Programovanie v Pythone, 3.vydanie

Andrej Blaho

04. sep 2018

Obsah

Fakulta matematiky, fyziky a informatiky

Univerzita Komenského v Bratislave

Autor Andrej Blaho

Názov Programovanie v Pythone (materiály k predmetom Programovanie (1) 1-AIN-130/16 a Programovanie (2) 1-AIN-170/13 na FMFI UK)

Vydavateľ Knižničné a edičné centrum FMFI UK

Rok vydania 2018

Miesto vydania Bratislava

Vydanie tretie prepracované vydanie

Počet strán 872

Internetová adresa http://input.sk/python2017/

Aktuálna adresa kurzu http://python.input.sk/

ISBN 978-80-8147-083-7

ISBN webovej verzie 978-80-8147-084-4

KAPITOLA 1

1. Úvod

1.1 Priebeh zimného semestra

Vyučovanie počas semestra sa skladá z

- týždenne dve dvojhodinové prednášky
- týždenne dve dvojhodinové cvičenia
- skoro každý týždeň jedno domáce zadanie
- jeden semestrálny projekt
- dva písomné testy
- praktická skúška pri počítači

prednášky

dvakrát týždenne po 2 hodiny v posluchárni A

- odporúčam nosiť si notebook s nainštalovaným Pythonom
- niektoré informácie z prednášok budú na webe, niečo sa dozviete len na prednáške
  - robte si podrobné poznámky
- prednášky nie sú povinné (povinné je len vedieť to, čo sa odprednášalo)

cvičenia

prebiehajú v počítačovej hale H3

- bude sa precvičovať látka hlavne z prednášok v danom týždni

- môžete pracovať na vlastnom notebooku
- spolupráca na cvičeniach nie je zakázaná - je odporúčaná
  - môžete pracovať aj po dvojiciach (dvaja riešia úlohy pri jednom počítači)
- na cvičeniach je povinná **aktívna účasť**
  - budú sa kontrolovať vyriešené úlohy
  - riešenie budete ukladať na úlohový server LIST
- počas semestra sú povolené maximálne **2** absencie

domáce zadania

počas semestra dostanete niekoľko povinných samostatných domácich zadaní

- na ich vyriešenie dostanete väčšinou 2 týždne
- vaše riešenia budú bodované (môžu mať rôzne ohodnotenia, väčšinou od 5 do 10 bodov)
  - podľa kvality riešenia získate od 0 až po maximálny počet bodov
  - riešenie budete ukladať na úlohový server LIST

priebežné hodnotenie

aktívna účasť na cvičeniach a body získané za domáce zadania sú priebežným hodnotením semestra

- zo všetkých cvičení počas semestra môžete mať maximálne **2 neúčasti**
- zo všetkých domácich zadaní musíte získať spolu aspoň **50% bodov**
- ak nesplníte podmienky priebežného hodnotenia, získavate známku **Fx** (bez možnosti robiť skúšku)

semestrálny projekt

v priebehu semestra dostávate možnosť riešiť jeden semestrálny projekt:

- tému si zvolíte z ponúknutej množiny tém (najčastejšie to budú jednoduché počítačové hry)
- za načas odovzdaný projekt môžete získať maximálne 10 bodov - tieto sa pripočítavajú k bodom ku skúške

skúška

sa skladá z dvoch častí:

1. dva písomné testy (v strede semestra **13.11.** a posledný týždeň semestra **18.12.**) - spolu max. 40 bodov
2. praktická skúška pri počítači (v skúškovom období) - max. 60 bodov

- máte nárok na 2 opravné termíny

hodnotenie skúšky

spočítajú sa body z písomného testu, praktickej skúšky, príp. bodov za semestrálny projekt:

- **:-)** známka **A** aspoň 90 bodov
- **:-)** známka **B** aspoň 80 bodov
- **:-)** známka **C** aspoň 70 bodov
- **:-)** známka **D** aspoň 60 bodov
- **:-)** známka **E** aspoň 50 bodov
- **:-(** známka **Fx** menej ako 50 bodov

užitočné linky

základná stránka Pythonu:

- Python Programming Language – Official Website
  - stiahnite si najnovšiu 3.x verziu (aspoň 3.6.3)
  - Python v3.x documentation

voľne stiahnuteľné materiály:

- How to Think Like a Computer Scientist: Interactive Edition
- Think Python: How to Think Like a Computer Scientist
- python.cz
- www.py.cz

prečítajte si

Teach Yourself Programming in Ten Years alebo slovenský preklad Ako sa naučiť programovať za desať rokov

1.2 Jazyk Python

Python je moderný programovací jazyk, ktorého popularita stále rastie.

- jeho autorom je Guido van Rossum (vymyslel ho v roku 1989)
- používajú ho napr. v Google, YouTube, Dropbox, Mozilla, Quora, Facebook, Rasperry Pi, ...
- na mnohých špičkových univerzitách sa učí ako úvodný jazyk, napr. MIT, Carnegie Mellon, Berkeley, Cornell, Caltech, Illinois, ...
- beží na rôznych platformách, napr. Windows, Linux, Mac. Je to *freeware* a tiež *open source*.

Na rozdiel od mnohých iných jazykov, ktoré sú kompilačné (napr. Pascal, C/C++, C#) je Python interpreter. To znamená, že

- interpreter nevytvára spustiteľný kód (napr. .exe súbor vo Windows)
- na spustenie programu musí byť v počítači nainštalovaný Python
- interpreter umožňuje aj interaktívnu prácu s prostredím

Hlavné vlastnosti jazyka Python:

- veľmi jednoduchá a dobre čitateľná syntax a keďže Python je aj vysoko interaktívny, je veľmi vhodný aj pre vyučovanie programovania
- na rozdiel od staticky typovaných jazykov, pri ktorých je treba dopredu deklarovať typy všetkých dát, je Python dynamicky typovaný, čo znamená, že neexistujú žiadne deklarácie
- Python obsahuje pokročilé črty moderných programovacích jazykov, napr. podpora práce s dátovými štruktúrami, objektovo-orientovaná tvorba softvéru, ...
- je to univerzálny programovací jazyk, ktorý poskytuje prostriedky na tvorbu moderných aplikácií, takých ako analýza dát, spracovanie médií, sieťové aplikácie a pod.
- Python má obrovskú komunitu programátorov a expertov, ktorí sú ochotní svojimi radami pomôcť aj začiatočníkom

1.2.1 Naštartujeme Python

Ako ho získať

- zo stránky https://www.python.org/ stiahnete najnovšiu verziu Pythonu - momentálne je to verzia **3.6.3**
- spustite inštalačný program (napr. `python-3.6.3.exe`)
- POZOR! Nesťahujte verziu začínajúcu 2 (napr. 2.7.14) - tá nie je kompatibilná s verziou 3.x

Alternatívne vývojové prostredia

- Inštalácia najnovšej verzie Pythonu obsahuje aj vývojové prostredie **IDLE**. Toto prostredie je pre úplného začiatočníka ideálne a budeme ho používať aj v tomto kurze.
- Pre skúsenejších odporúčame vybrať si jedno z týchto prostredí (mali by ste už mať nainštalovaný Python)
  - PyCharm Edu - Easy and Professional Tool to Learn & Teach Programming with Python
  - Wing Personal - A free Python IDE for students and hobbyists

Spustíme IDLE

IDLE (Python GUI) je vývojové prostredie (Integrated Development Learning Environment), vidíme informáciu o verzii Pythonu a tiež riadok s tromi znakmi >>> (tzv. výzva, t.j. **prompt**). Za túto výzvu budeme písať príkazy pre Python.

```
Python 3.6.3 (default, Oct  6 2017, 08:44:35)
[GCC 5.4.0 20160609] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
```

Ako to funguje

- Python je interpreter a pracuje v niekoľkých možných režimoch
- teraz sme ho spustili v **interaktívnom režime** (niekedy hovoríme aj príkazový režim): očakáva zadávanie textových príkazov (do riadka za znaky >>>), každý zadaný príkaz vyhodnotí a vypíše prípadnú reakciu (alebo **chybovú správu**, ak sme zadali niečo nesprávne)

- po skončení vyhodnocovania riadka sa do ďalšieho riadka znovu vypíšu znaky >>> a očakáva sa opätovné zadávanie ďalšieho príkazu
- takémuto interaktívnemu oknu hovoríme **shell**
- niekedy sa môžete dočítať aj o tzv. REP cykle interprétera, znamená to **Read**, **Evaluate**, **Print**, teda prečítaj, potom tento zápis vyhodnoť a na koniec vypíš výsledok, toto celé stále opakuj

Môžeme teda zadávať, napr. nejaké matematické výrazy:

```
>>> 12345
12345
>>> 123 + 456
579
>>> 1 * 2 * 3 * 4 * 5 * 6
720
>>>
```

V tomto príklade sme pracovali s celými číslami a niektorými celočíselnými operáciami. Python poskytuje niekoľko rôznych **typov** údajov; na začiatok sa zoznámime s tromi základnými typmi: celými číslami, desatinnými číslami a znakovými reťazcami.

celé čísla

- majú rovnaký význam, ako ich poznáme z matematiky: zapisujú sa v desiatkovej sústave a môžu začínať znamienkom plus alebo mínus
- ich veľkosť (počet cifier) je obmedzená len kapacitou pracovnej pamäte Pythonu (hoci aj niekoľko miliónov cifier)

Pracovať môžeme aj s **desatinnými číslami** (tzv. **floating point**), napr.

```
>>> 22 / 7
3.142857142857143
>>> .1 + .2 + .3 + .4
1.0
>>> 9999999999 * 99999999999
999999999890000000001
>>> 9999999999 * 99999999999.
9.9999999989e+20
>>>
```

desatinné čísla

- obsahujú desatinnú bodku alebo exponenciálnu časť (napr. 1e+15)
- môžu vzniknúť aj ako výsledok niektorých operácií (napr. delením dvoch celých čísel)
- majú obmedzenú presnosť (približne 16-17 platných cifier)

Všimnite si, že 3. výraz `9999999999*99999999999` násobí dve celé čísla a aj výsledkom je celé číslo. Hneď ďalší výraz `9999999999*99999999999.` obsahuje jedno desatinné číslo (číslo s bodkou) a teda aj výsledok je desatinné číslo.

V ďalšom príklade by sme chceli pracovať s nejakými textami. Ak ale zadáme:

```
>>> ahoj
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    ahoj
NameError: name 'ahoj' is not defined
>>>
```

dostaneme chybovú správu **NameError: name ‚ahoj‘ is not defined**, t.j. Python nepozná takéto meno. Takýmto spôsobom sa texty ako postupnosti nejakých znakov nezadávajú: na to potrebujeme špeciálny zápis: texty zadávame uzavreté medzi apostrofy, resp. úvodzovky. Takýmto textovým zápisom hovoríme **znakové reťazce**. Keď ich zapíšeme do príkazového riadka, Python ich vyhodnotí (v tomto prípade s nimi neurobí nič) a vypíše ich hodnotu. Napr.

```
>>> 'ahoj'
'ahoj'
>>> "hello folks"
'hello folks'
>>> 'úvodzovky "v" ret'azci'
'úvodzovky "v" ret'azci'
>>> "a tiež apostrofy 'v' ret'azci"
"a tiež apostrofy 'v' ret'azci"
>>>
```

znakové reťazce

- ich dĺžka (počet znakov) je obmedzená len kapacitou pracovnej pamäte Pythonu
- uzatvárame medzi apostrofy `'text'` alebo úvodzovky `"text"`
  - oba zápisy sú rovnocenné - reťazec musí končiť tým istým znakom ako začal (apostrof alebo úvodzovka)
  - takto zadaný reťazec nesmie presiahnuť jeden riadok
- môže obsahovať aj písmená s diakritikou
- prázdny reťazec má dĺžku 0 a zapisujeme ho ako `''`

Zatiaľ sme nepísali príkazy, ale len zadávali výrazy (číselné a znakové) - ich hodnoty sa vypísali v ďalšom riadku. Toto funguje len v tomto príkazovom režime.

1.2.2 Základné typy údajov

Videli sme, že hodnoty (konštanty alebo výrazy) môžu byť rôznych typov. V Pythone má každý typ svoje meno:

- `int` pre **celé čísla**, napr. `0`, `1`, `15`, `-123456789`, ...
- `float` pre **desatinné čísla**, napr. `0.0`, `3.14159`, `2.0000000001`, `33e50`, ...
- `str` pre **znakové reťazce**, napr. `'a'`, `"abc"`, `''`, `"I'm happy"`

Typ ľubovoľnej hodnoty vieme v Pythone zistiť pomocou štandardnej funkcie `type()`, napr.

```
>>> type(123)
<class 'int'>
>>> type(22 / 7)
<class 'float'>
>>> type(':-)')
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
<class 'str'>
>>>
```

Rôzne typy hodnôt majú zadefinované rôzne operácie.

Celočíselné operácie

- oba operandy musia byť celočíselného typu

`+`	sčitovanie	`1 + 2` má hodnotu `3`
`-`	odčitovanie	`2 - 5` má hodnotu `-3`
`*`	násobenie	`3 * 37` má hodnotu `111`
`//`	celočíselné delenie,	`22 // 7` má hodnotu `3`
`%`	zvyšok po delení	`22 % 7` má hodnotu `1`
`**`	umocňovanie	`2 ** 8` má hodnotu `256`

- zrejme nemôžeme deliť 0

Operácie s desatinnými číslami

- aspoň jeden z operandov musí byť desatinného typu (okrem delenia `/`)

`+`	sčitovanie	`1 + 0.2` má hodnotu `1.2`
`-`	odčitovanie	`6 - 2.86` má hodnotu `3.14`
`*`	násobenie	`1.5 * 2.5` má hodnotu `3.75`
`/`	delenie	`23 / 3` má hodnotu `7.666666666666667`
`//`	delenie zaokrúhlené nadol	`23.0 // 3` má hodnotu `7.0`
`%`	zvyšok po delení	`23.0 % 3` má hodnotu `2.0`
`**`	umocňovanie	`3 ** 3.` má hodnotu `27.0`

- zrejme nemôžeme deliť 0

Operácie so znakovými reťazcami

`+`	zreťazenie (spojenie dvoch reťazcov)	`'a' + 'b'` má hodnotu `'ab'`
`*`	viacnásobné zreťazenie reťazca	`3 * 'x'` má hodnotu `'xxx'`

Zreťazenie dvoch reťazcov je bežné aj v iných programovacích jazykoch. Viacnásobné zreťazenie je dosť výnimočné. Na príklade vidíme, ako to funguje:

```
>>> 'ahoj' + 'Python'
'ahojPython'
>>> 'ahoj' + ' ' + 'Python'
'ahoj Python'
>>> '#' + '#' + '#' + '#'
'####'
>>> 4 * '#'
'####'
>>> '#' * 4
'####'
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> 10 * ' :-) '
' :-)  :-)  :-)  :-)  :-)  :-)  :-)  :-)  :-)  :-) '
>>>
```

1.2.3 Premenné a priradenie

Doteraz sme pracovali len s hodnotami, t.j. s číslami a znakovými reťazcami. Teraz ukážeme, ako si zapamätať nejakú hodnotu tak, aby sme ju mohli použiť aj neskôr. Na toto slúžia tzv. **premenné**. Na rozdiel od premenných napr. v Pascale alebo C, kde premenná je pomenovanie nejakého konkrétneho vyhradeného pamäťového miesta (nejakej veľkosti a typu), v Pythone hovoríme, že premenná je pomenovaná nejaká existujúca hodnota v pamäti.

Premenná **vzniká** nie zadeklarovaním a spustením programu (ako v Pascale a v C), ale vykonaním priraďovacieho príkazu (nejakej existujúcej hodnote sa priradí meno).

Meno premennej:

- môže obsahovať písmená, číslice a znak podčiarkovník
- pozor na to, že v Pythone sa rozlišujú malé a veľké písmená
- musí sa líšiť od Pythonovských príkazov, tzv. **rezervovaných slov** (napr. `for`, `if`, `def`, `return`, ...)

Priraďovací príkaz

Zapisujeme:

```
premenna = hodnota
```

Tento zápis znamená, že do *premennej* (na ľavej strane príkazu pred znakom =) sa má priradiť zadaná *hodnota* (výraz na pravej strane za znakom =), t.j. zadaná hodnota dostáva meno a pomocou tohto mena s ňou budeme vedieť pracovať.

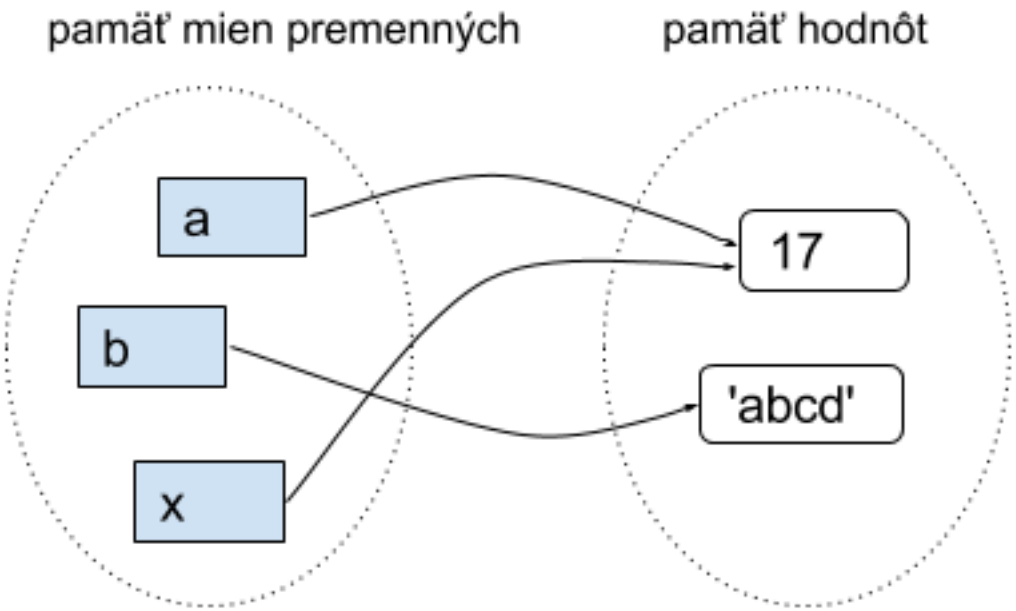
Premenná sa vytvorí priraďovacím príkazom (ak ešte doteraz neexistovala):

- v skutočnosti sa v Pythone do premennej priradí **referencia** (odkaz, adresa) na danú hodnotu (a nie samotná hodnota)
- ďalšie priradenie do tej istej premennej zmení túto referenciu
- na tú istú hodnotu sa môže odkazovať aj viac premenných
- meno môže referencovať (mať priradenú) maximálne jednu hodnotu (hoci samotná hodnota môže byť dosť komplexná)

Python si v svojej pamäti udržuje všetky premenné (v tzv. pamäti mien premenných) a všetky momentálne vytvorené hodnoty (v tzv. pamäti hodnôt). Po vykonaní týchto troch priraďovacích príkazov:

```
>>> a = 17
>>> b = 'abcd'
>>> x = a
```

To v pamäti Pythonu vyzerá nejako takto:



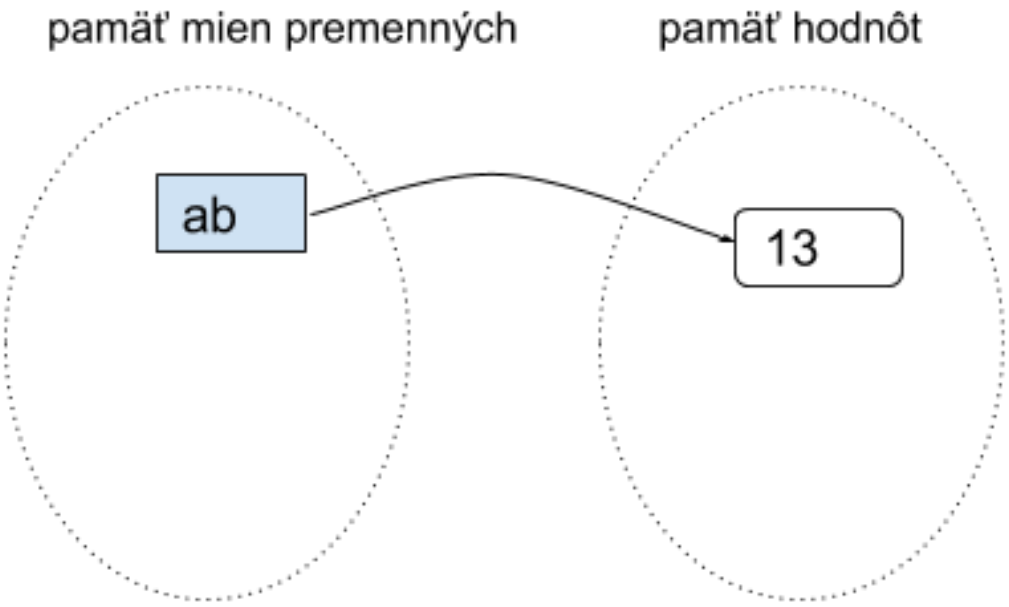
Vidíme, že

- ak priradíme do premennej nejakej hodnotu inej premennej (napr. `x = a`) neznamená to referenciu na meno ale na jej hodnotu
- najprv sa zistí hodnota na pravej strane príkazu a až potom sa spraví referencovanie (priradenie) do premennej na ľavej strane

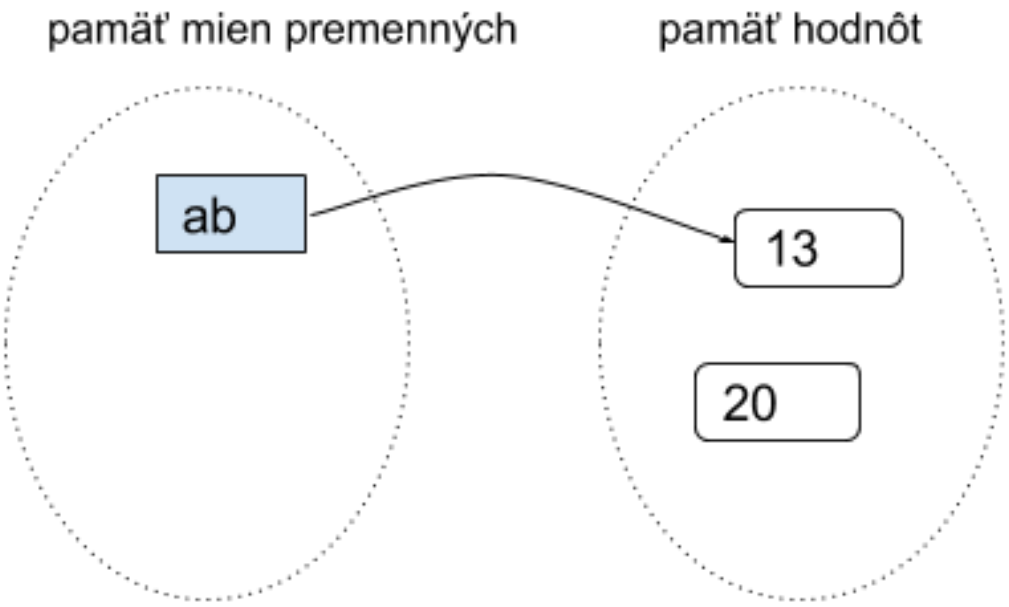
V ďalšom príklade vidíme, ako to funguje, keď vo výraze na pravej strane (kde je priraďovaná hodnota) sa nachádza tá istá premenná, ako na ľavej strane (teda kam priraďujeme):

```
>>> ab = 13
>>> ab = ab + 7
```

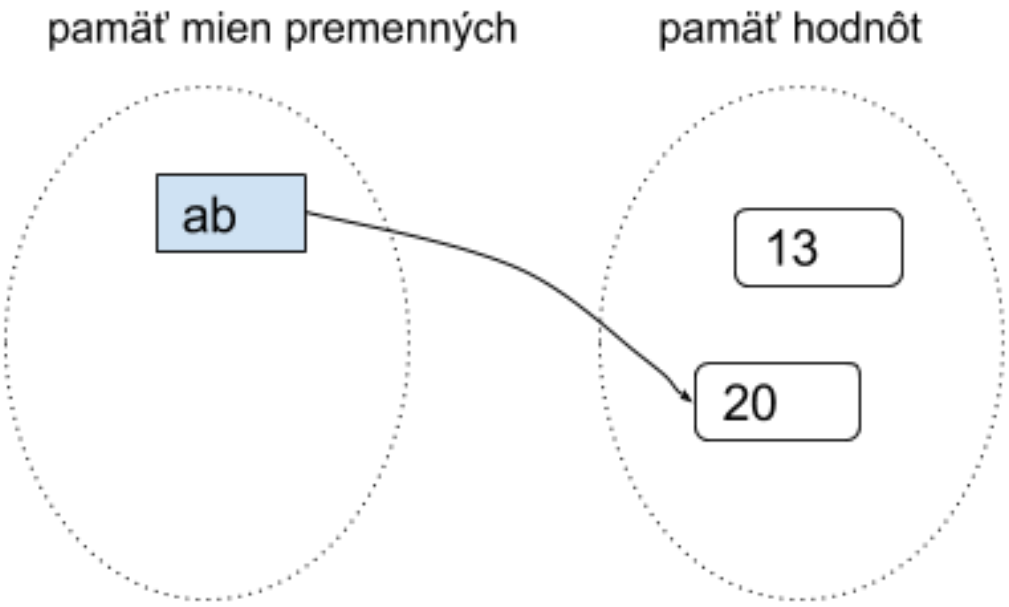
1. `ab` má najprv hodnotu `13`



2. potom sa vypočíta nová hodnota `20` (ako súčet `ab + 7`)



3. do premennej `ab` sa priradí nová hodnota



Desatinné čísla sa v počítači ukladajú s presnosťou len na istý počet desatinných miest a k tomu sa ešte pamätá aj tzv. exponenciálna časť, t.j. akou mocninou desiatky sa to celé vynásobí. Pozrite tento príklad:

```
>>> 10000.0
10000.0
>>> 1000e0
1000.0
>>> 1e4
10000.0
>>> 0.0001e8
10000.0
>>> 1000000e-3
1000.0
```

Všetky zápisy reprezentujú tú istú hodnotu (desatinné číslo 1000), ale okrem prvého všetky obsahujú exponenciálnu časť, t.j. číslo uvedené za písmenom `e`. Ale táto exponenciálna časť má svoje limity a nemôže byť ľubovoľne veľká. Ilustruje to nasledovný príklad:

```
>>> y = 4.3 * 10 ** 100          # velmi velke desatinne cislo
>>> y
4.3e+100
>>> y ** 2
1.849e+201
>>> y ** 3
7.9507e+301
>>> y ** 4
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    y ** 4
OverflowError: (34, 'Result too large')
```

Všimnite si, že exponenciálna časť môže byť väčšia ako 300 ale nesmie presiahnuť 400. `OverflowError: ..`

označuje chybovú správu pretečenia aritmetiky s desatinnými číslami.

Programovací režim

V IDLE vytvoríme nové okno (menu **File** a položka **New Window**), resp. stlačíme <Ctrl-N>. Otvorí sa nové textové okno, ale už bez promptu >>>. Do tohto okna nebudeme zadávať výrazy, ale príkazy. Použijeme priraďovacie príkazy, s ktorými sme pracovali vyššie:

```
a = 17
b = 'abcd'
x = a
```

Takto vytvorený program (hovoríme mu aj **skript**) treba uložiť (najčastejšie s príponou **.py**) a spustiť (**Run**, alebo klávesom <F5>). V okne shell sa objaví:

```
========== RESTART ==========
>>>
```

Po spustení sa v pôvodnom okne **shell** najprv celý Python reštartuje (zabudne, všetko, čo sme ho doteraz naučili, t.j. vyčistí pamäť premenných) a takto vyčistený vykoná všetky príkazy programu. V našom prípade sa vykonali len tri priradenia. Môžeme otestovať:

```
>>> a
17
>>> b
'abcd'
```

Ak by sme chceli, aby priamo náš program vypísal tieto hodnoty, takýto zápis tomu nepomôže:

```
a = 17
b = 'abcd'
x = a

a
b
```

Po spustení tohto programu sa opäť nič nevypíše. Zapamätajte si:

- po zadaní výrazov v interaktívnom režime (za promptom >>>) sa tieto vyhodnotili a hneď aj vypísali
- po zadaní výrazov v programovom režime sa tieto tiež vyhodnotia, ale ich **hodnota sa nevypíše ale ignoruje**
- ak chceme, aby sa táto hodnota neignorovala, musíme ju spracovať napr. priraďovacím príkazom alebo príkazom `print()` na výpis hodnôt (`print()` je v skutočnosti funkcia, uvidíme to neskôr)

Opravíme náš program:

```
a = 17
b = 'abcd'
x = a

print(a)
print(b)
print('výpočet je', 3 + x * 2)
```

Príkaz `print()` vypíše hodnoty uvedené v zátvorkách. Teda po spustení nášho programu dostávame:

```
========== RESTART ==========
17
abcd
výpočet je 37
>>>
```

Programy môžeme spúšťať nielen z vývojového prostredia IDLE, ale aj dvojkliknutím na súbor v operačnom systéme.

spustenie skriptu priamo zo systému

- ak je Python v operačnom systéme nainštalovaný korektne, dvojkliknutie na príponu súboru `.py` pre neho znamená spustenie programu:
  - otvorí sa nové konzolové okno, vykonajú sa príkazy a okno sa hneď aj zatvorí
- aby sa takéto okno nezatvorilo hneď, ale počkalo, kým nestlačíme napr. kláves **ENTER**, pridáme do programu nový riadok s príkazom `input()`

Do nášho skriptu dopíšeme nový riadok:

```
a = 17
b = 'abcd'
x = a

print(a)
print(b)
print('výpočet je', 3 + x * 2)

input()
```

Po spustení takéhoto programu sa najprv vypíšu zadané texty a okno sa nezavrie, kým nestlačíme **ENTER**. Príkaz `input()` môže obsahovať aj nejaký text, ktorý sa potom vypíše pre čakaním na **ENTER**, napr. takýto nový program:

```
# môj prvý program
print('programujem v Pythone')
print()

input('stlač ENTER')
```

Po spustení v operačnom systéme (nie v IDLE) sa v konzolovom okne objaví:

```
programujem v Pythone

stlač ENTER
```

Až po stlačení klávesu **ENTER** sa okno zatvorí.

Všimnite si, že do prvého riadka programu sme zapísali tzv. **komentár**, t.j. text za znakom `#`, ktorý sa Pythonom ignoruje.

Zhrňme oba tieto nové príkazy:

`print()`

- uvidíme neskôr, že je to volanie špeciálnej funkcie

- táto funkcia vypisuje hodnoty výrazov, ktoré sú uvedené medzi zátvorkami
- hodnoty sú pri výpise oddelené medzerami
- `print()` bez parametrov spôsobí len zariadkovanie výpisu, teda vloží na momentálne miesto prázdny riadok

`input()`

- je tiež funkcia, ktorá najprv vypíše zadaný znakový reťazec (ak je zadaný) a potom čaká na vstupný reťazec ukončený ENTER
- funkcia vráti tento nami zadaný znakový reťazec

Funkciu `input()` môžeme otestovať aj v priamom režime:

```
>>> input()
pisem nejaky text
'pisem nejaky text'
```

Príkaz `input()` tu čaká na stlačenie **ENTER**. Kým ho nestlačíme, ale píšeme nejaký text, tento sa postupne zapamätáva. Stlačenie **ENTER** (za napísaný text `pisem nejaky text`) spôsobí, že sa tento zapamätaný text vráti ako výsledok, teda v príkazovom režime sa vypísala hodnota zadaného znakového reťazca. Druhý príklad najprv vypíše zadaný reťazec `'? '` a očakáva písanie textu s ukončením pomocou klávesu ENTER:

```
>>> input('? ')
? matfyz
'matfyz'
>>>
```

Môžeme to otestovať aj takýmto programom:

```
meno = input('ako sa volas? ')
print('ahoj', meno)
```

V tomto príklade sa využíva funkcia `input()`, ktorá najprv zastaví bežiaci výpočet, vypýta si od používateľa, aby zadal nejaký text a tento uloží do premennej `meno`. Na koniec toto zadané meno vypíše. Program spustíme klávesom **F5**:

```
ako sa volas? Jozef
ahoj Jozef
>>>
```

Týmto program skončil a môžeme pokračovať aj v skúmaní premenných, napr. v programovom režime zistíme hodnotu premennej `meno`:

```
>>> meno
'Jozef'
```

V našich budúcich programoch bude bežné, že na začiatku sa vypýtajú hodnoty nejakých premenných a ďalej program pracuje s nimi.

Ďalší program ukazuje, ako to vyzerá, keď chceme načítať nejaké číslo:

```
# prevod euro na ceske koruny

suma = input('zadaj eura: ')
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
koruny = suma * 26
print(suma, 'euro je', koruny, 'korun')
```

Po spustení programu dostávame:

```
=========== RESTART ===========
zadaj eura: 100
100 euro je 100100100100100100100100100100100100100100100100100100100100100100100100100100␣
↪korun
>>>
```

Takýto výsledok je zrejme nesprávny: očakávali sme 2600 korún. Problémom tu bolo to, že funkcia `input()` zadanú hodnotu vráti nie ako číslo, ale ako znakový reťazec, teda `'100'`. Na tomto mieste potrebujeme takýto reťazec **prekonvertovať** na desatinné číslo. Využijeme jednu z troch konvertovacích (pretypovacích) funkcií:

pretypovanie hodnôt

Mená typov `int`, `float` a `str` zároveň súžia ako mená pretypovacích funkcií, ktoré dokážu z jedného typu vyrobiť hodnotu iného typu:

- `int(hodnota)` z danej hodnoty vyrobí celé číslo, napr.
  - `int(3.14)` => `3`
  - `int('37')` => `37`
- `float(hodnota)` z danej hodnoty vyrobí desatinné číslo, napr.
  - `float(333)` => `333.0`
  - `float('3.14')` => `3.14`
- `str(hodnota)` z danej hodnoty vyrobí znakový reťazec, napr.
  - `str(356)` => `'356'`
  - `str(3.14)` => `'3.14'`

Zrejme pretypovanie reťazca na číslo bude fungovať len vtedy, keď je to správne zadaný reťazec, inak funkcia vyhlási chybu.

Program by mal vyzerať správne takto:

```
# prevod euro na ceske koruny

retazec = input('zadaj eura: ')
suma = float(retazec)                    # pretypovanie
koruny = suma * 26
print(suma, 'euro je', koruny, 'korun')
```

Prečítaný reťazec sa najprv prekonvertuje na desatinné číslo a až potom sa s týmto číslom pracuje ako so zadanou cenou jedného výrobku v eurách. Po spustení dostávame:

```
zadaj eura: 100
100.0 euro je 2600.0 korun
```

Kvôli čitateľnosti programu sme tu použili tri rôzne premenné: `retazec`, `suma` a `koruny`. Neskôr budeme takto jednoduché programy zapisovať kompaktnejšie. Tu je príklad, ktorý by vám mal ukázať, že takto zatiaľ programy

nezapisujte: sú len pre pokročilého čitateľa a hlavne sa v takomto zápise ťažšie hľadajú a opravujú chyby. Nasledovné programy robia skoro presne to isté ako náš predchádzajúci program:

```
suma = float(input('zadaj eura: '))
koruny = suma * 26
print(suma, 'euro je', koruny, 'korun')
```

```
suma = float(input('zadaj eura: '))
print(suma, 'euro je', suma * 26, 'korun')
```

```
print('zadana suma v euro je', float(input('zadaj eura: ')) * 26, 'korun')
```

Veľa našich programov bude začínať načítaním niekoľkých vstupných hodnôt. Podľa typu požadovanej hodnoty môžeme prečítaný reťazec hneď prekonvertovať na správny typ, napr. takto:

```
cele = int(input('zadaj celé číslo: '))                # konverovanie na celé číslo
desatinne = float(input('zadaj desatinné číslo: '))    # konverovanie na desatinné␣
↪číslo
retazec = input('zadaj znakový ret'azec: ')             # ret'azec netreba konvertovat'
```

Úprava pythonovského programu

Programátori majú medzi sebou dohodu, ako správne zapisovať pythonovský kód (oficiálny dokument je PEP 8). My zatiaľ píšeme len veľmi jednoduché zápisy, ale je dôležité si zvykať už od začiatku na správne zápisy. Takže niekoľko základných pravidiel:

- mená premenných obsahujú len malé písmená
- pre znak = v priraďovacom príkaze dávame medzeru pred aj za
- operácie v aritmetických výrazoch sú väčšinou tiež oddelené od operandov medzerami
- riadky programu by nemali byť dlhšie ako 79 znakov
- za čiarky, napr. ktoré oddeľujú parametre v príkaze `print()` dávame vždy medzeru

Postupne sa budeme zoznamovať aj s ďalšími takýmito odporúčaniami.

V Pythone je zadefinovaných niekoľko štandardných funkcií, ktoré pracujú s číslami. Ukážeme si dve z nich: výpočet absolútnej hodnoty a zaokrúhľovaciu funkciu:

abs() absolútna hodnota

`abs`(*cislo*)

Parametre **`cislo`** – celé alebo desatinné číslo

Funkcia `abs(cislo)` vráti absolútnu hodnotu zadaného čísla, napr.

- `abs(13)` => `13`
- `abs(-3.14)` => `3.14`

Funkcia nemení typ parametra, s ktorým bola zavolaná, t.j.

```
>>> type(abs(13))
<class 'int'>
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> type(abs(-3.14))
<class 'float'>
```

Ak vyskúšame zistiť typ nie výsledku volania funkcie, ale samotnej funkcie, dostávame:

```
>>> type(abs)
<class 'builtin_function_or_method'>
```

Totiž aj každá funkcia (teda aj `print` aj `input`) je hodnotou, s ktorou sa dá pracovať podobne ako s číslami a reťazcami, teda ju môžeme napr. priradiť do premennej:

```
>>> a = abs
>>> print(a)
<built-in function abs>
```

Zatiaľ je nám toto úplne na nič, ale je dobre o tom vedieť už teraz. Keď už budeme dobre rozumieť mechanizmu priraďovania mien premenných rôznymi hodnotami, bude nám jasné, prečo funguje:

```
>>> vypis = print
>>> vypis
<built-in function print>
>>> vypis('ahoj', 3 * 4)
ahoj 12
```

Ale môže sa nám „prihodiť“ aj takýto nešťastný preklep:

```
>>> print=('ahoj')
>>> print('ahoj')
...
TypeError: 'str' object is not callable
```

Do premennej `print`, ktorá obsahovala referenciu na štandardnú funkciu, sme omylom priradili inú hodnotu (znakový reťazec `'ahoj'`) a tým sme znefunkčnili vypisovanie hodnôt pomocou pôvodného obsahu tejto premennej.

Ďalšia funkcia `help()` nám niekedy môže pomôcť v jednoduchej nápovedi k niektorým funkciám a tiež typom. Ako parameter pošleme buď meno funkcie, alebo hodnotu nejakého typu:

```
>>> help(abs)
Help on built-in function abs in module builtins:

abs(...)
    abs(number) -> number

    Return the absolute value of the argument.

>>> help(0)
Help on int object:

class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
...
```

... ďalej pokračuje dlhý výpis informácií o celočíselnom type.

Druhou štandardnou číselnou funkciou je zaokrúhľovanie.

round() zaokrúhľovanie čísla

round (*cislo*)
round (*cislo*, *pocet*)

Parametre

- **cislo** – celé alebo desatinné číslo
- **pocet** – celé číslo, ktoré vyjadruje na koľko desatinných miest sa bude zaokrúhľovať; ak je to záporné číslo, zaokrúhľuje sa na počet mocnín desiatky

Funkcia `round(cislo)` vráti zaokrúhlenú hodnotu zadaného čísla na celé číslo. Funkcia `round(cislo, pocet)` vráti zaokrúhlené číslo na príslušný počet desatinných miest, napr.

- `round(3.14)` => `3`
- `round(-0.74)` => `-1`
- `round(3.14, 1)` => `3.1`
- `round(2563, -2)` => `2600`

Tiež si o tom môžete prečítať pomocou:

```
>>> help(round)
Help on built-in function round in module builtins:

round(...)
    round(number[, ndigits]) -> number

    Round a number to a given precision in decimal digits (default 0 digits).
    This returns an int when called with one argument, otherwise the
    same type as the number. ndigits may be negative.
```

1.2.4 Ešte priraďovacie príkazy

Vráťme sa k priraďovaciemu príkazu:

```
meno_premennej = hodnota
```

- najprv sa zistí hodnota na pravej strane priraďovacieho príkazu => táto hodnota sa vloží do **pamäte hodnôt**
- ak sa toto `meno_premennej` ešte nenachádzalo v **pamäti mien premenných**, tak sa vytvorí toto nové meno
- `meno_premennej` dostane referenciu na novú vytvorenú hodnotu

Pozrime sa na takéto priradenie:

```
>>> ab = ab + 5
...
NameError: name 'ab' is not defined
```

Ak premenná `ab` ešte neexistovala, Python nevie vypočítať hodnotu `ab + 5` a hlási chybovú správu. Skúsme najprv do `ab` niečo priradiť:

```
>>> ab = 13
>>> ab = ab + 5
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> ab
18
```

Tomuto hovoríme aktualizácia (update) premennej: hodnotu premennej `ab` sme zvýšili o 5 (najprv sa vypočítalo `ab` + 5, čo je 18) a potom sa toto priradilo opäť do premennej `ab`. Konkrétne takto sme zvýšili (inkrementovali) obsah premennej. Podobne by fungovali aj iné operácie, napr.

```
>>> ab = ab * 11
>>> ab
198
>>> ab = ab // 10
>>> ab
19
```

Python na tieto prípady aktualizácie nejakej premennej ponúka špeciálny zápis priraďovacieho príkazu:

```
meno_premennej += hodnota         # meno_premennej = meno_premennej + hodnota
meno_premennej -= hodnota         # meno_premennej = meno_premennej - hodnota
meno_premennej *= hodnota         # meno_premennej = meno_premennej * hodnota
meno_premennej /= hodnota         # meno_premennej = meno_premennej / hodnota
meno_premennej //= hodnota        # meno_premennej = meno_premennej // hodnota
meno_premennej %= hodnota         # meno_premennej = meno_premennej % hodnota
meno_premennej **= hodnota        # meno_premennej = meno_premennej ** hodnota
```

Každý z týchto zápisov je len skrátenou formou bežného priraďovacieho príkazu. Nemusíte ho používať, ale verím, že časom si naň zvyknete a bude pre vás veľmi prirodzený.

Všimnite si, že fungujú aj tieto zaujímavé prípady:

```
>>> x = 45
>>> x -= x           # to isté ako x = 0
>>> x += x           # to isté ako x *= 2
>>> z = 'abc'
>>> z += z
>>> z
'abcabc'
```

Ďalším užitočným tvarom priraďovacieho príkazu je možnosť naraz priradiť tej istej hodnoty do viacerých premenných. Napr.

```
x = 0
sucet = 0
pocet = 0
ab = 0
```

Môžeme to nahradiť jediným priradením:

```
x = sucet = pocet = ab = 0
```

V takomto hromadnom priradení dostávajú všetky premenné tú istú hodnotu, teda referencujú na tú istú hodnotu v pamäti hodnôt.

Posledným užitočným variantom priradenia je tzv. paralelné priradenie: naraz priraďujeme aj rôzne hodnoty do viacerých premenných. Napr.

```
x = 120
y = 255
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
meno = 'bod A'
pi = 3.14
```

Môžeme zapísať jedným paralelným priradením:

```
x, y, meno, pi = 120, 255, 'bod A', 3.14
```

Samozrejme, že na oboch stranách priraďovacieho príkazu musí byť rovnaký počet mien premenných a počet hodnôt. Veľmi užitočným využitím takéhoto paralelného priradenia je napr. výmena obsahov dvoch premenných:

```
>>> a = 3.14
>>> b = 'hello'
>>> a, b = b, a          # paralelné priradenie
>>> a
'hello'
>>> b
3.14
```

Paralelné priradenie totiž funguje takto:

- najprv sa zistí postupnosť všetkých hodnôt na pravej strane priraďovacieho príkazu (bola to dvojica `b`, `a`, teda hodnoty `'ahoj'` a `3.14`)
- tieto dve zapamätané hodnoty sa naraz priradia do dvoch premenných `a` a `b`, teda sa vymenia ich obsahy

Zamyslite sa, čo sa dostane do premenných po týchto príkazoch:

```
>>> p1, p2, p3 = 11, 22, 33
>>> p1, p2, p3 = p2, p3, p1
```

alebo

```
>>> x, y = 8, 13
>>> x, y = y, x+y
```

Paralelné priradenie funguje aj v prípade, že na pravej strane príkazu je jediný znakový reťazec nejakej dĺžky a na pravej strane je presne toľko premenných, ako je počet znakov, napr.

```
>>> a, b, c, d, e, f = 'Python'
>>> print(a, b, c, d, e, f)
P y t h o n
```

Keďže tento znakový reťazec je vlastne postupnosť 6 znakov, priradením sa táto postupnosť 6 znakov paralelne priradí do 6 premenných.

1.2.5 Znakové reťazce

Ak znakový reťazec obsahuje dvojicu znakov `'\n'`, tieto označujú, že pri výpise funkciou `print()` sa namiesto nich prejde na nový riadok. Napr.

```
>>> a = 'prvý riadok\nstredný\ntretí riadok'
>>> a
'prvý riadok\nstredný\ntretí riadok'
>>> print(a)
prvý riadok
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
stredný
tretí riadok
```

Takáto dvojica znakov `'\n'` zaberá v reťazci len jeden znak.

Python umožňuje pohodlnejšie vytvárania takýchto „viacriadkových“ reťazcov. Ak reťazec začína tromi apostrofmi `'''` (alebo úvodzovkami `"""`), môže prechádzať aj cez viac riadkov, ale opäť musí byť ukončený rovnakou trojicou, ako začal. Prechody na nový riadok v takomto reťazci sa nahradia špeciálnym znakom `'\n'`. Napr.

```
>>> ab = '''prvý riadok
stredný
tretí riadok'''
>>> ab
'prvý riadok\nstredný\ntretí riadok'
>>> print(ab)
prvý riadok
stredný
tretí riadok
```

Takýto reťazec môže obsahovať aj apostrofy a úvodzovky.

Niekedy potrebujeme vytvárať znakový reťazec pomocou komplikovanejšieho zápisu, v ktorom ho budeme skladať (zreťaziť) z viacerých hodnôt, napr.

```
>>> meno, x, y = 'A', 180, 225
>>> r = 'bod ' + meno + ' na súradniciach (' + str(x) + ',' + str(y) + ')'
>>> r
'bod A na súradniciach (180,225)'
```

Python poskytuje špeciálny typ reťazca (tzv. formátovací znakový reťazec), pomocou ktorého môžeme vytvárať aj takto komplikované výrazy. Základom je formátovacia šablóna, do ktorej budeme vkladať ľubovoľné aj číselné hodnoty. V našom prípade bude šablónou reťazec `f'bod {meno} na súradniciach ({x},{y})'`. V tejto šablóne sa každá dvojica znakov `{...}` nahradí príslušnou hodnotou, v našom prípade týmito hodnotami sú postupne `meno`, `x`, `y`. Všimnite si, znak `f` pred začiatkom reťazca. Zápis takejto formátovacej metódy bude:

```
>>> meno, x, y = 'A', 180, 225
>>> r = f'bod {meno} na súradniciach ({x},{y})'
>>> r
'bod A na súradniciach (180,225)'
```

V Pythone existuje aj špeciálny typ funkcie (tzv. metódu znakového reťazca), aj nej môžeme vytvárať takto komplikované výrazy. Základom je opäť formátovacia šablóna, do ktorej budeme vkladať ľubovoľné aj číselné hodnoty. V našom prípade bude šablónou reťazec `'bod {} na súradniciach ({},{})'`. V tejto šablóne sa každá dvojica znakov `{}` nahradí nejakou konkrétnou hodnotou, v našom prípade týmito hodnotami sú postupne `meno`, `x`, `y`. Zápis takejto formátovacej metódy bude:

```
>>> meno, x, y = 'A', 180, 225
>>> r = 'bod {} na súradniciach ({},{})'.format(meno, x, y)
>>> r
'bod A na súradniciach (180,225)'
```

To znamená, že za reťazec šablóny píšeme znak bodka a hneď za tým volanie funkcie `format()` s hodnotami, ktoré sa do šablóny dosadia (zrejme ich musí byť rovnaký počet ako dvojíc `{}`). Neskôr sa zoznámime aj s ďalšími veľmi užitočnými špecialitami takéhoto formátovania.

1.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Zistite, čo sa vypočíta
   - skontrolujte

```
>>> 1/0
>>> 1//0
>>> 1%0
>>> 1--2
>>> 1---2
>>> 1----2
>>> 1-----2
```

2. Vieme, že desatinné čísla sú obmedzené nejakou konkrétnou maximálnou hodnotou. Pokusmi nájdite taký čo najväčší exponent v semilogaritmickom zápise desatinného čísla `1e+xxx`, že už o 1 väčší by vzniklo niečo iné ako desatinné číslo
   - napr.

```
>>> 1e+100
1e+100
>>> 1e+101
1e+101
```

3. Pre dané odvesny `a` a `b` vypočítajte preponu `c`
   - napr.

```
>>> a = 10
>>> b = 14
>>> c = ...
>>> c
17.204650534085253
```

4. V premenných `meno` a `priezvisko` sú priradené nejaké dva znakové reťazce. Do premennej `desat` priraďte reťazec, ktorý obsahuje desaťkrát meno a priezvisko oddelené čiarkou.
   - napr.

```
>>> meno = 'Janko'
>>> priezvisko = 'Hraško'
>>> desat = ...
>>> desat
'Janko Hraško, Janko Hraško, Janko Hraško, Janko Hraško, Janko Hraško,
Janko Hraško, Janko Hraško, Janko Hraško, Janko Hraško, Janko Hraško'
```

5. V dvoch premenných `cislo1` a `cislo2` sú priradené nejaké dve celé čísla (`cislo2` bude učite trojciferné). Vytvorte novú hodnotu do premennej `cislo3`, ktorá zlepí tieto dve čísla do jedného celého čísla. Nepoužívajte znakové reťazce.
   - napr.

```
>>> cislo1 = 4567
>>> cisko2 = 912
>>> cislo3 = ...
>>> cislo3
4567912
```

6. Zapíšte prevod českej koruny na euro a opačne (predpokladajte, že kurz je 26 korún za 1 euro).
   - napr. ak priradíme do `ck` 1000 korún, vypočíta sa euro a potom sa 30 euro prepočíta na české koruny

```
>>> ck = 1000
>>> euro = ...
>>> euro
38.46153846153846
>>> euro = 30
>>> ck = ...
>>> ck
780
```

7. Vypočítajte stav účtu, na ktorom je na 4% úrok na začiatku 100 euro. V premennej `pocet_rokov` je priradená hodnota počtu rokov, ktoré nás zaujímajú.
   - napr.

```
>>> pocet_rokov = 10
>>> stav_uctu = ...
>>> stav_uctu
148.02442849183444
```

8. Napíšte skript, ktorý vypíše vašu vizitku: bude orámovaná znakmi `'-'` a `'|'` a obsahovať by mala aspoň vaše meno, adresu a telefón (adresu a telefón si môžete vymyslieť).
   - napr.

```
+---------------------+
| Janko Hraško        |
| Pri brázde 17       |
| mobil: 0999 123456  |
+---------------------+
```

9. Napíšte skript, ktorý najprv do troch premenných `prvy`, `druhy` a `treti` priradí tri rôzne znakové reťazce a potom vypíše všetky rôzne poradia týchto reťazcov (permutácie).
   - napr. pre `prvy = 'biela'`, `druhy = 'modrá'` a `treti = 'červená'` vypíše

```
biela modrá červená
modrá červená biela
...
```

10. Napíšte skript, v krtorom sa najprv do premennej `cislo` priradí nejaké celé číslo a potom do troch premenných `c1`, `c2`, `c3` priradí posledné tri cifry daného čísla. Potom tieto tri cifry vypíše.
    - napr. pre `cislo = 12345678` vypíše

```
posledné tri cifry čísla 12345678 sú 6 7 8
```

11. Napíšte skript, ktorý najprv do premennej `n` priradí nejakú celočíselnú hodnotu, pre ktorú `n >= 10`. Potom na základe tejto hodnoty vypíše orámovaný text `'PYTHON'`. Tento text bude v rámiku „prirazený“ k ľavému okraju, pričom medzi textom a rámikom bude práve jedna medzera.

- napr. pre `n = 12` program vypíše:

```
************
*          *
* PYTHON   *
*          *
************
```

12. Napíšte skript, ktorý najprv do premennej `n` priradí nejakú celočíselnú hodnotu, pre ktorú `n >= 10`. Potom na základe tejto hodnoty vypíše orámovaný text `'PYTHON'`, ktorý bude ale vycentrovaný (pozor na nepárne `n`).

- napr. pre `n = 20` program vypíše:

```
********************
*                  *
*       PYTHON     *
*                  *
********************
```

KAPITOLA 2

2. Opakované výpočty

Doteraz sme sa naučili pracovať s tromi základnými príkazmi:

- priraďovací príkaz vytvorí alebo zmení obsah nejakej premennej
- výpis nejakých hodnôt do textovej plochy pomocou `print()`
- prečítanie hodnoty zadanej klávesnicou pomocou `input()`

Z týchto troch typov príkazov sme skladali programy (skripty), ktorých príkazy sa potom vykonávali postupne jeden za druhým. Lenže pri programovaní reálnych programov budeme potrebovať, aby sme nejaké časti programov mohli vykonávať viackrát za sebou bez toho, aby sme to museli viackrát rozpísať.

Napr. namiesto:

```
print('programujem v Pythone')
print('programujem v Pythone')
print('programujem v Pythone')
print('programujem v Pythone')
print('programujem v Pythone')
```

by sme potrebovali zapísať:

```
opakuj nasledovný príkaz 5-krát:
    print('programujem v Pythone')
```

Na toto v Pythone slúži konštrukcia **for-cyklus**

2.1 For-cyklus

Postupne ukážeme niekoľko základných typov použitia for-cyklu.

2.1.1 Cyklus s daným počtom opakovaní

Táto programová konštrukcia má takýto tvar:

```
for premenna in range(pocet):
    blok prikazov
```

Opakuje zadaný **počet** krát príkazy odsunutého bloku príkazov (tzv. **indentation**). Samotný riadok konštrukcie **for** obsahuje meno nejakej premennej a je ukončený znakom dvojbodka. Za tým nasleduje **blok príkazov** - sú to príkazové riadky, napr. `print()`, ktoré sú odsunuté o 4 medzery.

Zapíšme program, ktorý 5-krát vypíše zadaný text:

```
for prem in range(5):
    print('programujem v Pythone')
```

Blok príkazov môže obsahovať nielen jeden príkaz ale aj viac, napr.

```
for prem in range(5):
    print('studujem na matfyze a')
    print('programujem v Pythone')
print('============')
```

Blok príkazov, ktorý sa má opakovať v danom cykle končí napr. vtedy, keď sa objaví riadok s príkazmi na úrovni riadku s for-cyklom. Teda posledný riadok so znakmi `'============'` sa vypíše až po skončení cyklu, teda iba raz:

```
studujem na matfyze a
programujem v Pythone
studujem na matfyze a
programujem v Pythone
studujem na matfyze a
programujem v Pythone
studujem na matfyze a
programujem v Pythone
studujem na matfyze a
programujem v Pythone
============
```

Zatiaľ nevieme, na čo slúži premenná cyklu (v našom príklade `prem`). Python tejto premennej automaticky nastavuje hodnotu podľa toho, koľký krát sa už cyklus vykonal. Teda zápis:

```
for prem in range(n):
    prikazy
```

v skutočnosti znamená:

```
prem = 0
prikazy
prem = 1
prikazy
prem = 2
prikazy
...
prem = n-1
prikazy
```

Napr.

```
for i in range(4):
    print(i, 'riadok')
```

označuje

```
i = 0
print(i, 'riadok')
i = 1
print(i, 'riadok')
i = 2
print(i, 'riadok')
i = 3
print(i, 'riadok')
```

Teda program vypíše:

```
0 riadok
1 riadok
2 riadok
3 riadok
```

For-cyklus začne byť zaujímavý až keď sa v tele cyklu budú robiť nejaké výpočty. Začnime jednoduchým pripočítavaním 1:

```
n = int(input('zadaj n: '))
pocet = 0
for i in range(n):
    pocet = pocet + 1
print('pocet prechodov cyklu =', pocet)
```

Premennú `pocet` sme ešte pred začiatkom cyklu vynulovali. Výstup bude napr. takýto

```
zadaj n: 17
pocet prechodov cyklu = 17
```

Ak budeme namiesto 1 pripočítavať hodnotu premennej cyklu:

```
n = int(input('zadaj n: '))
sucet = 0
for i in range(n):
    sucet = sucet + i
print('sucet =', sucet)
```

dostávame súčet čísel od 0 do n-1, teda 0 + 1 + 2 + 3 + . . . + n-2 + n-1, napr.

```
zadaj n: 17
sucet = 136
```

Malou zmenou spočítame druhé mocniny tejto postupnosti:

```
n = int(input('zadaj n: '))
sucet = 0
for i in range(n):
    sucet = sucet + i * i    # alebo sucet += i ** 2
print('sucet =', sucet)
```

Uvedomte si, že takýmto algoritmom úplne zbytočne pripočítavame aj 0 na začiatku cyklu.

2.1.2 Cyklus s vymenovanými hodnotami

Ukážme tento typ for-cyklu na príklade:

```
for i in 1, 2, 3, 4, 5:
    blok prikazov
```

Namiesto funkcie `range(n)`, ktorá pre nás vygenerovala postupnosť čísel od 0 do `n-1`, sme vymenovali presné poradie hodnôt, pre ktoré sa postupne v cykle vykoná `blok prikazov`. Vymenované hodnoty musia byť oddelené čiarkou a mali by byť aspoň dve.

Otestujme:

```
sucin = 1
for cislo in 1, 2, 3, 4, 5, 6:
    sucin = sucin * cislo
print('6 faktorial =', sucin)
```

a naozaj dostávame:

```
6 faktorial = 720
```

Ďalší program počíta druhé mocniny niektorých zadaných čísel:

```
for x in 5, 7, 11, 13, 23:
    x2 = x ** 2
    print('druhá mocnina', x, 'je', x2)
```

Po spustení dostávame:

```
druhá mocnina 5 je 25
druhá mocnina 7 je 49
druhá mocnina 11 je 121
druhá mocnina 13 je 169
druhá mocnina 23 je 529
```

Vymenované hodnoty sa môžu aj ľubovoľne opakovať, napr.

```
i = 1
for prem in 2, 1, 7, 2, 3:
    print(i, 'prechod cyklu s hodnotou', prem)
    i = i + 1
```

vypíše:

```
1 prechod cyklu s hodnotou 2
2 prechod cyklu s hodnotou 1
3 prechod cyklu s hodnotou 7
4 prechod cyklu s hodnotou 2
5 prechod cyklu s hodnotou 3
```

Ďalší príklad ukazuje výpočet dní v roku ako súčet počtov dní v jednotlivých mesiacoch:

```
pocet = 0
for mesiac in 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31:
    pocet += mesiac
print('pocet dni v beznom roku =', pocet)
```

Zrejme, takýto typ cyklu môžeme použiť len vtedy, keď máme k dispozícii presný zoznam hodnôt a nie ľubovoľný počet, ktorý predtým zvládla funkcia `range()`.

Okrem toho, že sa hodnoty môžu opakovať, nemáme obmedzenia ani na typy vymenovaných hodnôt. Ukážme to na príklade, v ktorom spočítame ceny jednotlivých položiek nákupu a okrem tejto sumy vypočítame aj priemernú hodnotu:

```
pocet = 0
suma = 0
for cena in 1.75, 2.20, 1.03, 4.00, 3.50, 2.90, 1.89:
    suma = suma + cena
    pocet = pocet + 1
print('nakupil si', pocet, 'poloziek')
print('za', suma, 'euro')
print('priemerna cena bola', round(suma / pocet, 2), 'euro')
```

a výsledkom je:

```
nakupil si 7 poloziek
za 17.27 euro
priemerna cena bola 2.47 euro
```

Všimnite si, že opäť sme použili rovnakú schému na sčitovanie pomocou cyklu, ako sme to robili vyššie. Niekedy sa tomuto hovorí **pripočítavacia šablóna**: ešte pred cyklom inicializujeme nejakú pripočítavaciu premennú (v našom príklade dokonca dve premenné `pocet` a `suma`) a v tele cyklu hodnotu tejto premennej zvyšujeme podľa potreby (napr. pripočítame 1, alebo pripočítame premennú cyklu, alebo jej mocninu, alebo vynásobíme niečím, alebo vydelíme, ...). Po skončené cyklu máme v tejto pomocnej pripočítavacej premennej očakávaný výsledok.

Aj ďalšie dva príklady ilustrujú to, že vymenované hodnoty pre for-cyklu môžu byť rôznych typov:

```
for slovo in 'Python', 'Bratislavu', 'Matfyz':
    print('mam rad', slovo)
```

Hodnotami sú znakové reťazce a výsledkom bude:

```
mam rad Python
mam rad Bratislavu
mam rad Matfyz
```

V nasledovnom príklade sú vymenované hodnoty najrôznejších typov, dokonca jednou z hodnôt je aj funkcia `abs`. Cyklus vypíše hodnotu premennej cyklu a potom aj jej typ:

```
for hodnota in 3.14, abs(7 - 123), 'text', 100 / 4, abs, '42':
    print(hodnota, type(hodnota))
```

a výpis:

```
3.14 <class 'float'>
116 <class 'int'>
text <class 'str'>
25.0 <class 'float'>
<built-in function abs> <class 'builtin_function_or_method'>
42 <class 'str'>
```

2.1.3 Cyklus s prvkami znakového reťazca

Už sme videli, že znakové reťazce sa môžu nachádzať medzi vymenovanými hodnotami for-cyklu. Ale znakový reťazec v Pythone je v skutočnosti **postupnosť znakov**. Vďaka tomu for-cyklus môže prechádzať aj prvky tejto postupnosti. Premenná cyklu potom postupne nadobúda hodnoty jednotlivých znakov, čo sú vlastne jednoznakové reťazce. Teda

```
for znak in 'python':
    print(znak)
```

je to isté ako:

```
for znak in 'p', 'y', 't', 'h', 'o', 'n':
    print(znak)
```

a zrejme sa vypíše:

```
p
y
t
h
o
n
```

Ďalší príklad ilustruje použitie **pripočítavacej šablóny** aj pre znakové reťazce:

```
vstup = input('zadaj: ')
pocet = 0
retazec1 = retazec2 = ''
for znak in vstup:
    retazec1 = retazec1 + znak
    retazec2 = znak + retazec2
    pocet = pocet + 1
print('pocet znakov retazca =', pocet)
print('retazec1 =', retazec1)
print('retazec2 =', retazec2)
```

Dostávame:

```
zadaj: Python
pocet znakov retazca = 6
retazec1 = Python
retazec2 = nohtyP
```

Všimnite si, že `retazec2` obsahuje prevrátené poradie znakov pôvodného reťazca. Otestujte, že takýto for-cyklus bude fungovať nielen s jednoznakovým reťazcom, ale aj s prázdnym.

2.1.4 Funkcia range() aj pre iné postupnosti celých čísel

Videli sme, že funkcia `range(n)` nahrádza vymenovanie celočíselných hodnôt od `0` do `n-1`. Táto funkcia je v skutočnosti trochu univerzálnejšia: dovolí nám zadať nielen koncovú hodnotu vygenerovanej postupnosti ale aj počiatočnú. V tomto prípade funkciu zavoláme s dvomi parametrami:

- prvý parameter potom označuje počiatočnú hodnotu postupnosti
- druhý parameter označuje hodnotu, pri ktorej generovanie postupnosti končí, t.j. postupnosť bude obsahovať len hodnoty menšie ako tento druhý parameter

Napr. `range(5, 15)` označuje rastúcu postupnosť celých čísel, ktorá začína hodnotou `5` a všetky ďalšie prvky sú menšie ako `15`, teda vygenerovaná postupnosť by bola: `5, 6, 7, 8, 9, 10, 11, 12, 13, 14`. Ak teda potrebujeme postupnosť čísel od `1` do zadaného `n`, musíme zapísať:

```
n = int(input('zadaj n: '))
for cislo in range(1, n+1):
    print('hodnota v cykle', cislo)
print('koniec cyklu')
```

a výstupom je napr.

```
zadaj n: 7
hodnota v cykle 1
hodnota v cykle 2
hodnota v cykle 3
hodnota v cykle 4
hodnota v cykle 5
hodnota v cykle 6
hodnota v cykle 7
koniec cyklu
```

Teraz môžeme pomocou **pripočítavacej šablóny** vypočítať aj faktoriál pre ľubovoľnú zadanú hodnotu:

```
n = int(input('zadaj cislo: '))
faktorial = 1
for cislo in range(2, n+1):
    faktorial = faktorial * cislo
print(n, 'faktorial =', faktorial)
```

Spustíme s rôznymi hodnotami:

```
zadaj cislo: 1
1 faktorial = 1

zadaj cislo: 6
6 faktorial = 720

zadaj cislo: 20
20 faktorial = 2432902008176640000
```

Ak by sme nasledovný program spustili:

```
for i in range(100, 200):
    print(i)
```

dostali by sme 100-riadkový výpis s číslami od 100 do 199. Teraz by sa nám ale hodilo, keby `print()` v niektorých situáciách nekončil prechodom na nový riadok. Využijeme nový typ parametra funkcie `print()`:

funkcia `print()`

`print` (..., *end='ret'azec'*)

Parametre `end='ret'azec'` – tento reťazec nahradí štandardný `'\n'` na ľubovoľný iný, najčastejšie je to jedna medzera `' '` alebo prázdny reťazec `''`

Tento parameter musí byť v zozname parametrov funkcie `print()` uvedený ako posledný za všetkými vypisovanými hodnotami. Vďaka nemu po vypísaní týchto hodnôt sa namiesto prechodu na nový riadok vypíše zadaný

reťazec.

Napr.

```
for i in range(100, 200):
    print(i, end=' ')
```

teraz vypíše:

```
100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119
120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139
140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179
180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199
```

Ale s prázdnym reťazcom pre parameter `end`:

```
for i in range(100, 200):
    print(i, end='')
```

vypíše:

```
100101102103104105106107108109110111112113114115116117118119120121122123124125126127128129130131132133134135136137138139
140141142143144145146147148149150151152153154155156157158159160161162163164165166167168169170171172173174175176177178179
180181182183184185186187188189190191192193194195196197198199
```

Takýto zápis využijeme hlavne pri výpise väčšieho počtu hodnôt, ale aj vtedy, keď jeden riadok výpisu potrebujeme poskladať z viacerých častí v rôznych častiach programu, napr.

```
print('programujem', end='_')
print(10, end='...')
print('rokov')
```

vypíše:

```
programujem_10...rokov
```

Funkcii `range()` môžeme zadať aj tretí parameter, pričom všetky parametre musia byť celočíselné hodnoty. Zhrňme všetky tri varianty tejto funkcie:

funkcia `range()`

`range`(*stop*)
`range`(*start*, *stop*)
`range`(*start*, *stop*, *krok*)

Parametre

- **`start`** – prvý prvok vygenerovanej postupnosti (ak chýba, predpokladá sa 0)
- **`stop`** – hodnota, na ktorej sa už generovanie ďalšej hodnoty postupnosti zastaví - táto hodnota už v postupnosti nebude
- **`krok`** – hodnota, o ktorú sa zvýši každý nasledovný prvok postupnosti, ak tento parameter chýba, predpokladá sa 1

Najlepšie si to ukážeme na príkladoch rôzne vygenerovaných postupností celých čísel. V tabuľke vidíme výsledky pre rôzne parametre:

`range(10)`	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
`range(0, 10)`	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
`range(0, 10, 1)`	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
`range(3, 10)`	3, 4, 5, 6, 7, 8, 9
`range(3, 10, 2)`	3, 5, 7, 9
`range(10, 100, 10)`	10, 20, 30, 40, 50, 60, 70, 90
`range(10, 1)`	*prázdna postupnosť*
`range(1, 1)`	*prázdna postupnosť*

Nasledovný príklad ilustruje použitie parametra krok vo funkcii `range()`. Potrebujeme spočítať súčet všetkých nepárnych čísel do 1000. Zrejme začneme s `1` a každé ďalšie číslo je o 2 väčšie. Teda

```
sucet = 0
for cislo in range(1, 1000, 2):
    sucet = sucet + cislo
print('sucet neparnych cisel je', sucet)
```

Špeciálnym prípadom je záporný krok, t.j. keď požadujeme klesajúcu postupnosť čísel. Napr. zápis `range(15, 5, -1)` označuje, že prvý člen postupnosti bude `15`, všetky ďalšie budú o `1` menšie (parameter `krok` je -1) a posledný z nich nebude **menší** ako `5` (parameter `stop`). Otestujeme:

```
for i in range(15, 5, -1):
    print(i, end=' ')
```

a dostávame postupnost:

```
15 14 13 12 11 10 9 8 7 6
```

čo je vlastne prevrátené poradie postupnosti `range(6, 16)`. Ak sa vám záporný krok pri volaní `range()` nie veľmi páči, Python to umožňuje zapísať aj elegantnejšie pomocou funkcie `reversed()` a funkcie `range()` takto:

```
for i in reversed(range(6, 16)):
    print(i, end=' ')
```

čím dostávame rovnakú postupnosť ako v predchádzajúcom príklade. Ešte skontrolujme:

```
for i in reversed(range(10)):
    print(i, end=' ')
```

s výsledkom:

```
9 8 7 6 5 4 3 2 1 0
```

čo je veľakrát čitateľnejšie ako použitie `range(9, -1, -1)`.

2.2 Moduly math a random

My už poznáme niektoré štandardné funkcie, ktoré sú zadefinované už pri štarte Pythonu:

- funkcia `type()` vráti typ zadanej hodnoty
- funkcie `int()`, `float()` a `str()` pretypujú zadanú hodnotu na iný typ

- funkcie `print()` a `input()` sú určené na výpis textov a prečítanie textu zo vstupu
- funkcie `abs()` a `round()` počítajú absolútne hodnoty čísel a zaokrúhľujú desatinné čísla
- funkcie `round()` a `reversed()` generujú postupnosť čísel, resp. ju otáčajú

Štandardných funkcií je oveľa viac a z mnohými z nich sa zoznámime neskôr. Teraz sa zoznámime s dvoma novými modulmi (predstavme si ich ako nejaké knižnice užitočných funkcií), ktoré hoci nie sú štandardne zabudované, my ich budeme často potrebovať. Ak potrebujeme pracovať s nejakým modulom, musíme to najprv Pythonu nejako oznámiť. Slúži na to príkaz `import`.

2.2.1 Modul math

Pomocou takéhoto zápisu:

```
import math
```

umožníme našim programom pracovať s knižnicou matematických funkcií. V skutočnosti týmto príkazom Python vytvorí novú premennú `math`. Knižnica v tomto module obsahuje napr. tieto matematické funkcie: `sin()`, `cos()`, `sqrt()`. Lenže s takýmito funkciami nemôžeme pracovať priamo: Python nepozná ich mená, pozná jediné meno a to meno modulu `math`. Keďže tieto funkcie sa nachádzajú práve v tomto module, budeme k nim pristupovať, tzv. bodkovou notáciou (**dot notation**), t.j. za meno modulu uvedieme prvok (v tomto prípade funkciu) z daného modulu. Napr. `math.sin()` označuje volanie funkcie **sínus** a `math.sqrt()` označuje výpočet druhej odmocniny čísla. Otestujme v interaktívnom režime:

```
>>> math
<module 'math' (built-in)>
>>> type(math)
<class 'module'>
>>> math.sin
<built-in function sin>
>>> math.sin()
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    math.sin()
TypeError: sin() takes exactly one argument (0 given)
```

Posledná chybová správa oznamuje, že funkciu `sin` musíme volať práve s jedným parametrom (volanie bez parametrov sa Pythonu nepáči).

Ak zadáme `dir(math)`, Python nám vypíše všetky prvky, ktoré sa nachádzajú v tomto module:

```
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh'
, 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite',
'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf',
'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
'trunc']
```

Väčšina prvkov modulu `math` nás zatiaľ nebude zaujímať, ale môžeme tam vidieť napr. aj funkcie `exp()`, `log()`, `tan()`, `radians()` ale aj známe konštanty `e` a `pi`. Ak chceme poznať detaily nejakého prvku modulu, môžeme použiť štandardnú funkciu `help()`, napr.

```
>>> help(math.log)
Help on built-in function log in module math:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
log(...)
    log(x[, base])

    Return the logarithm of x to the given base.
    If the base not specified, returns the natural logarithm (base e) of x.
```

Môžeme sa dozvedieť, že funkcia `math.log()` počíta logaritmus čísla buď so základom `e` (prirodzené logaritmy) alebo s daným základom `base`.

alebo

```
>>> help(math.sin)
Help on built-in function sin in module math:

sin(...)
    sin(x)

    Return the sine of x (measured in radians).
```

Toto označuje, že funkcia `sin()` z modulu `math` naozaj počíta sínus, ale uhol musíme zadať v radiánoch. Preto napr. pre výpočet `sin(45)` zapíšeme:

```
>>> math.sin(45 * 3.14159 / 180)
0.7071063120935576
>>> math.sin(45 * math.pi / 180)
0.7071067811865475
>>> math.sin(math.radians(45))
0.7071067811865475
```

Druhý a tretí výpočet využívajú buď konštantu `pi` alebo konverznú funkciu `radians()`, ktorá prevádza stupne na radiány. Zrejme najčastejšie budeme používať tretí variant pomocou `radians()`.

Vytvorme tabuľku hodnôt sínusov aj kosínus pre uhly od 0 do 90 stupňov krokom 5:

```
import math

for uhol in range(0, 91, 5):
    uhol_v_radianoch = math.radians(uhol)
    sin_uhla = math.sin(uhol_v_radianoch)
    cos_uhla = math.cos(uhol_v_radianoch)
    print(uhol, sin_uhla, cos_uhla)
```

Výpis nie je veľmi pekný - obsahuje čísla vypísané zbytočne na veľ desatinných miest:

```
0 0.0 1.0
5 0.08715574274765817 0.9961946980917455
10 0.17364817766693033 0.984807753012208
15 0.25881904510252074 0.9659258262890683
...
```

Urobme z toho zarovnanú tabuľku s tromi stĺpcami. Využijeme to, že vo formátovacom reťazci môžeme zadať to, aby sa desatinné číslo vypisovalo na šírku 6 znakov pričom sú 3 desatinné miesta. Hodnota v `{}` zátvorkách môže za znakom `':'` obsahovať šírku výpisu. Všimnite si posledný riadok s volaním `print()`:

```
import math
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
for uhol in range(0, 91, 5):
    uhol_v_radianoch = math.radians(uhol)
    sin_uhla = math.sin(uhol_v_radianoch)
    cos_uhla = math.cos(uhol_v_radianoch)
    print(f'{uhol:3} {sin_uhla:6.3f} {cos_uhla:6.3f}')
```

Prvé riadky výpisu teraz už vyzerajú takto:

```
  0  0.000  1.000
  5  0.087  0.996
 10  0.174  0.985
 15  0.259  0.966
 20  0.342  0.940
 25  0.423  0.906
...
```

Pokúsme sa nakresliť (v textovej ploche pomocou nejakých znakov) priebeh funkcie sínus. Keďže oborom hodnôt tejto funkcie je interval reálnych čísel `<-1, 1>` a my chceme tieto hodnoty natiahnuť na šírku výpisu do 80 znakov, zapíšeme:

```
import math

for uhol in range(0, 361, 10):
    uhol_v_radianoch = math.radians(uhol)
    sin_uhla = math.sin(uhol_v_radianoch)
    stlpec = int(sin_uhla * 35 + 40)
    print(' ' * stlpec + 'SIN')
```

Sínusovka v textovej ploche potom vyzerá takto:

```
                                    SIN
                                          SIN
                                               SIN
                                                     SIN
                                                          SIN
                                                              SIN
                                                                   SIN
                                                                     SIN
                                                                       SIN
                                                                        SIN
                                                                       SIN
                                                                     SIN
                                                                   SIN
                                                              SIN
                                                          SIN
                                                     SIN
                                               SIN
                                          SIN
                                    SIN
                               SIN
                          SIN
                     SIN
                SIN
           SIN
      SIN
  SIN
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
SIN
SIN
SIN
  SIN
    SIN
        SIN
            SIN
                 SIN
                      SIN
                           SIN
                                SIN
                                     SIN
```

Všimnite si, že všetky tri premenné `uhol_v_radianoch`, `sin_uhla` a `stlpec` v tele cyklu tu slúžia „len“ na zvýšenie čitateľnosti kódu a mohli by sme to zapísať aj bez nich:

```
import math

for uhol in range(0, 361, 10):
    print(' ' * int(math.sin(math.radians(uhol)) * 35 + 40) + 'SIN')
```

Takýto zápis je menej prehľadný a najmä pre začiatočníkov sa neodporúča.

2.2.2 Modul random

Aj tento modul obsahuje knižnicu funkcií, ale tieto umožňujú generovanie náhodných čísel. My z tejto knižnice využijeme najmä tieto dve funkcie:

- `randrange()` vyberie náhodné číslo z postupnosti celých čísel
- `choice()` vyberie náhodný prvok z nejakej postupnosti, napr. zo znakového reťazca (postupnosti znakov)

Aby sme mohli pracovať s týmito funkciami, nesmieme zabudnúť zapísať:

```
import random
```

Nasledovná ukážka ilustruje volanie funkcie `randrange()`. Parametre tejto funkcie majú presne rovnaký význam ako pre nás známa funkcia `range()`. Preto každé volanie `random.randrange(1, 7)` **náhodne** vyberie jednu z hodnôt postupnosti `1, 2, 3, 4, 5, 6`. Môžeme si to predstaviť ako hod hracou kockou, na ktorej sú čísla od 1 do 6. Program vypíše postupnosť 100 náhodných hodov kocky:

```
import random

for i in range(100):
    nahodne = random.randrange(1, 7)
    print(nahodne, end=' ')
```

a spustenie dá podobné výsledky:

```
4 1 3 5 1 1 6 6 1 6 5 2 2 4 4 6 1 2 5 1 5 5 5 4 3 2 5 3 2 6 1 2 2 2 4 3 5 3 4 1
3 4 4 4 5 4 3 6 6 1 3 3 4 3 5 5 4 6 3 2 2 4 3 2 6 1 5 5 3 6 5 6 6 5 4 5 5 6 3 6
6 5 6 3 2 1 5 4 5 2 4 1 2 5 1 1 2 2 5 4
```

Veľmi podobne funguje aj druhá funkcia `choice()`. Táto má len jeden parameter, ktorým je nejaká postupnosť hodnôt. Pre nás je v súčasnosti najvhodnejšou postupnosťou postupnosť znakov, teda ľubovoľný znakový reťazec. Napr. volanie:

```
random.choice('aeiouy')
```

vyberie **náhodnú** hodnotu z postupnosti šiestich znakov - postupnosti samohlások. Podobne by sme mohli zapísať:

```
random.choice('bcdfghjklmnpqrstvwxz')
```

aj toto volanie vyberie **náhodné** písmeno z postupnosti spoluhlások. Keď to teraz dáme dokopy, dostaneme generátor náhodne vygenerovaných slov:

```
import random

slovo = ''
for i in range(3):
    spoluhlaska = random.choice('bcdfghjklmnpqrstvwxz')
    samohlaska = random.choice('aeiouy')
    slovo = slovo + spoluhlaska + samohlaska
print(slovo)
```

Program vygeneruje 3 **náhodné** dvojice spoluhlások a samohlások, teda dvojpísmenových slabík. Vždy, keď budeme potrebovať ďalšie náhodné slovo, musíme spustiť tento program (napr. pomocou F5). Môžeme dostať napr. takéto náhodné slová:

```
gugaqo

lupiha

cyxebi
```

Ak by sme potrebovali vygenerovať napr. naraz 10 slov, použijeme znovu for-cyklus. Preto cely náš program (okrem úvodného `import`) obalíme konštrukciou `for`, t.j. všetky riadky súčasného programu posunieme o 4 znaky vpravo:

```
import random

for j in range(10):
    slovo = ''
    for i in range(3):
        spoluhlaska = random.choice('bcdfghjklmnpqrstvwxz')
        samohlaska = random.choice('aeiouy')
        slovo = slovo + spoluhlaska + samohlaska
    print(slovo)
```

Všimnite si, že v tele vonkajšieho for-cyklu (s premennou cyklu `j`) sa nachádzajú tri príkazy: priradenie, potom tzv. **vnorený** for-cyklus a na koniec volanie funkcie `print()`.

2.2.3 Vnorené cykly

Na nasledovných príkladoch ukážeme niekoľko

Napíšme najprv program, ktorý vypíše čísla od 0 do 99 do 10 riadkov tak, že v prvom stĺpci sú čísla od 0 do 9, v druhom od 10 do 19, ... v poslednom desiatom sú čísla od 90 do 99:

```
for i in range(10):
    print(i, i+10, i+20, i+30, i+40, i+50, i+60, i+70, i+80, i+90)
```

Po spustení dostaneme:

```
0 10 20 30 40 50 60 70 80 90
1 11 21 31 41 51 61 71 81 91
2 12 22 32 42 52 62 72 82 92
3 13 23 33 43 53 63 73 83 93
4 14 24 34 44 54 64 74 84 94
5 15 25 35 45 55 65 75 85 95
6 16 26 36 46 56 66 76 86 96
7 17 27 37 47 57 67 77 87 97
8 18 28 38 48 58 68 78 88 98
9 19 29 39 49 59 69 79 89 99
```

Riešenie tohto príkladu využíva for-cyklus len na vypísanie 10 riadkov, pričom obsah každého riadka sa vyrába bez cyklu jedným príkazom `print()`. Toto je ale nepoužiteľný spôsob riešenia v prípadoch, ak by tabuľka mala mať premenlivý počet stĺpcov, napr. keď je počet zadaný zo vstupu. Vytvorenie jedného riadka by sme teda tiež mali urobiť for-cyklom, t.j. budeme definovať for-cyklus, ktorý je vo vnútri iného cyklu, tzv. **vnorený** cyklus. Všimnite si, že celý tento cyklus musí byť odsadený o ďalšie 4 medzery:

```
for i in range(10):
    for j in range(0, 100, 10):
        print(i + j, end=' ')
    print()
```

Vnútorný for-cyklus vypisuje 10 čísel, pričom premenná cyklu `i` postupne nadobúda hodnoty `0`, `10`, `20`, ... `90`. K tejto hodnote sa pripočítava číslo riadka tabuľky, teda premennú `j`. Tým dostávame rovnakú tabuľku, ako predchádzajúci program. Rovnaký výsledok vytvorí aj nasledovné riešenie:

```
for i in range(10):
    for j in range(i, 100, 10):
        print(j, end=' ')
    print()
```

V tomto programe má vnútorný cyklus tiež premennú cyklu s hodnotami s krokom 10, ale v každom riadku sa začína s inou hodnotou.

Túto istú ideu využijeme, aj keď budeme vytvárať tabuľku čísel od 0 do 99, ale organizovanú inak: v prvom riadku sú čísla od 0 do 9, v druhom od 10 do 19, ... v poslednom desiatom sú čísla od 90 do 99:

```
for i in range(0, 100, 10):
    for j in range(i, i + 10):
        print(j, end=' ')
    print()
```

Možných rôznych zápisov riešení tejto úlohy je samozrejme viac.

Ešte dve veľmi podobné úlohy:

1. Prečítať celé číslo `n` a vypísať tabuľku čísel s `n` riadkami, pričom v prvom je len `1`, v druhom sú čísla `1 2`, v treťom `1 2 3`, atď. až v poslednom sú čísla od 1 do `n`:

   ```
   pocet = int(input('zadaj počet riadkov: '))
   for riadok in range(1, pocet + 1):
       for cislo in range(1, riadok + 1):
           print(cislo, end=' ')
       print()
   ```

   Všimnite si mená oboch premenných cyklov `riadok` a `cislo`, vďaka čomu môžeme lepšie pochopiť, čo sa v ktorom cykle deje. Spustíme, napr.

```
zadaj počet riadkov: 7
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
```

2. Zadanie je podobné, len tabuľka v prvom riadku obsahuje `1`, v druhom `2 3`, v treťom `4 5 6`, atď. každý ďalší riadok obsahuje o jedno číslo viac ako predchádzajúci a tieto čísla v každom ďalšom riadku pokračujú v číslovaní. Zapíšeme jedno z možných riešení:

```
pocet = int(input('zadaj počet riadkov: '))
cislo = 1
for riadok in range(1, pocet + 1):
    for stlpec in range(1, riadok + 1):
        print(cislo, end=' ')
        cislo += 1
    print()
```

```
zadaj počet riadkov: 7
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
16 17 18 19 20 21
22 23 24 25 26 27 28
```

V tomto riešení využívame **pomocnú premennú** `cislo`, ktorú sme ešte pred cyklom nastavili na 1, vo vnútornom cykle vypisujeme jej hodnotu (a nie premennú cyklu) a zakaždým ju zvyšujeme o 1.

2.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/
- **v riešeniach úloh používajte len konštrukcie a funkcie, ktoré sme sa učili na prvých dvoch prednáškach**

1. Program prečíta nejaké slovo a jedno celé číslo `n`, potom vypíše `n`-riadkov, pričom v každom bude niekoľko znakov bodka a toto zadané slovo: v prvom riadku výpisu bude pred slovom `n`-1 znakov `'.'`, v každom ďalšom bude o bodku menej

   - napr.

   ```
   zadaj slovo: Python
   zadaj n: 5

   ....Python
   ...Python
   ..Python
   ```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
.Python
Python
```

2. Program prečíta nejaké slovo a jedno celé číslo `n`, potom vypíše `n`-riadkov: v prvom je zadané slovo raz, v druhom je 2-krát (slová sú oddelené medzerou), v treťom 3-krát (tiež s medzerou medzi slovami), atď. Použite len jeden cyklus.

   - napr.

```
zadaj slovo: slon
zadaj n: 4

slon
slon slon
slon slon slon
slon slon slon slon
```

3. Program prečíta dve celé čísla `od` a `do` a vypíše tabuľku druhých a tretích mocnín všetkých čísel z intervalu <`od`, `do`>.

   - napr.

```
zadaj od: 5
zadaj do: 11

5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
11 121 1331
```

4. Program dá rovnaký výsledok ako v úlohe (3), ale najprv pripraví celý výstup (všetky riadky s trojicami čísel) do jedného znakového reťazca (aj so znakmi `'\n'`) a až po skončení cyklu tento reťazec vypíše.

   - napr.

```
zadaj od: -2
zadaj do: 3

-2 4 -8
-1 1 -1
0 0 0
1 1 1
2 4 8
3 9 27
```

5. Program prečíta celé číslo a vypíše počet cifier a tiež ciferný súčet tohto čísla. Číslo preveďte na znakový reťazec (resp. ho po prečítaní neprevádzajte na číslo) a potom postupne pomocou for-cyklu prechádzajte jeho cifry (jednoznakové reťazce), každú preveďte na číslo a pripočítajte k výslednému súčtu.

   - napr.

```
zadaj cislo: 53124
pocet cifier = 5
ciferny sucet = 15
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
zadaj cislo: 126765060022822940149670320537 6
pocet cifier = 31
ciferny sucet = 115
```

6. Program prečíta celé číslo (maximálne 8-ciferné) a potom v cykle (8-krát) postupne toto číslo skracuje o poslednú cifru, zároveň vypisuje skrátené číslo aj poslednú cifru, o ktorú bolo skrátené.

   - napr.

```
zadaj cislo: 31841624

3184162 4
318416 2
31841 6
3184 1
318 4
31 8
3 1
0 3
```

7. Program prečíta slovo (znakový reťazec) a vytvorí z neho nový reťazec. Tento bude mať pred každým znakom číslo vyjadrujúce jeho pozíciu v pôvodnom reťazci, teda postupne pre každý znak pridá reťazce `'0'`, `'1'`, `'2'`,...

   - napr.

```
zadaj slovo: python
vysledok = 0p1y2t3h4o5n
```

8. Program prečíta číslo `n` a potom do jedného riadka postupne vypisuje `n` čísel tak, aby sa stále striedali týchto sedem čísel: 0 1 2 3 4 5 6.

   - napr.

```
zadaj n: 18
0 1 2 3 4 5 6 0 1 2 3 4 5 6 0 1 2 3
```

   - možno využijete operáciu zvyšok po delení `cislo % 7`

9. Program prečíta celé číslo `n` a potom do `n` riadkov postupne na striedačku vypíše texty `'cervena'` a `'modra'`

   - napr.

```
zadaj n: 5
cervena
modra
cervena
modra
cervena
```

   - možno použijete dve pomocné premenné s textami `'cervena'` a `'modra'` a v cykle budete vypisovať hodnotu prvej z nich a potom budete vymieňať ich obsah

10. Program najprv prečíta jedno celé číslo `n` a potom prečíta nasledovných `n` desatinných čísel. Na záver vypíše súčet týchto čísel aj ich priemer.

    - napr.

```
zadaj pocet: 4
zadaj cislo: 14.2
zadaj cislo: 1
zadaj cislo: 5.7
zadaj cislo: 22

sucet je 42.9
priemer je 10.725
```

11. Program prečíta dve celé čísla a zo znakov `'+'`, `'-'` a `'|'` vypíše obdĺžnik veľkosti podľa zadaných čísel.
    - napr.

```
zadaj pocet stlpcov: 5
zadaj pocet riadkov: 2

+-----+
|     |
|     |
+-----+
```

12. Program prečíta tri reťazce a potom pomocou vnorených cyklov vypíše všetky dvojice týchto reťazcov.
    - napr.

```
zadaj 1. slovo: aaa
zadaj 2. slovo: bbb
zadaj 3. slovo: ccc

aaa aaa
aaa bbb
aaa ccc
bbb aaa
bbb bbb
bbb ccc
ccc aaa
ccc bbb
ccc ccc
```

13. Program prečíta celé číslo `n` a vypíše výpočet faktoriálu tohto čísla v tvare `n! = 1*2*...*n = číslo`.
    - napr.

```
zadaj n: 5

5! = 1*2*3*4*5 = 120
```

14. Program vypíše tabuľku výpočtu prvých 10 faktoriálov (vo formáte z predchádzajúceho príkladu). Použite vnorené cykly.
    - výpis by mal byť v tomto tvare

```
1! = 1 = 1
2! = 1*2 = 2
3! = 1*2*3 = 6
4! = 1*2*3*4 = 24
...
```

15. Program prečíta celé číslo `n` a vypíše tabuľku `n` x `n` čísel od 1 do `n**2`, ktoré sú usporiadané do stĺpcov.

- napr.

```
zadaj n: 5

1 6 11 16 21
2 7 12 17 22
3 8 13 18 23
4 9 14 19 24
5 10 15 20 25
```

16. Program prečíta celé číslo `n` a vypíše tabuľku čísel s `n` riadkami, pričom v prvom sú čísla od 1 do `n`, každý ďalší riadok je o 1 kratší (bez posledného čísla), v poslednom je 1.

    - napr.

```
zadaj pocet: 5

1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

17. Upravte riešenie predchádzajúceho príkladu tak, že výpis každého riadku (okrem prvého) bude o 1 posunutý vpravo oproti predchádzajúcemu.

    - napr.

```
zadaj pocet: 5

1 2 3 4 5
 1 2 3 4
  1 2 3
   1 2
    1
```

KAPITOLA 3

3. Grafika

Doteraz sme pracovali len v textovom režime: do textovej plochy (shell) sme z programu zapisovali len pomocou `print()`. V súčasných operačných systémoch (windows, linux, os, ...) sú skoro všetky aplikácie grafické. Preto to skúsime aj my.

Budeme pracovať s modulom **tkinter**, ktorý slúži na vytváranie grafických aplikácii. My ho budeme využívať prakticky len na kreslenie na tzv. **plátno**, hoci tento modul zvláda aj iné typy grafických prvkov. Príkladom grafickej aplikácie, ktorá je kompletne napísaná v Pythone a používa **tkinter** je vývojové prostredie **IDLE**.

Užitočné informácie k **tkinter** nájdete napr. v materiáli:

- Tkinter 8.5 reference: a GUI for Python

Základná grafická aplikácia

Minimálna grafická aplikácia je táto:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

# tu sa bude kreslit do grafickej plochy

tkinter.mainloop()
```

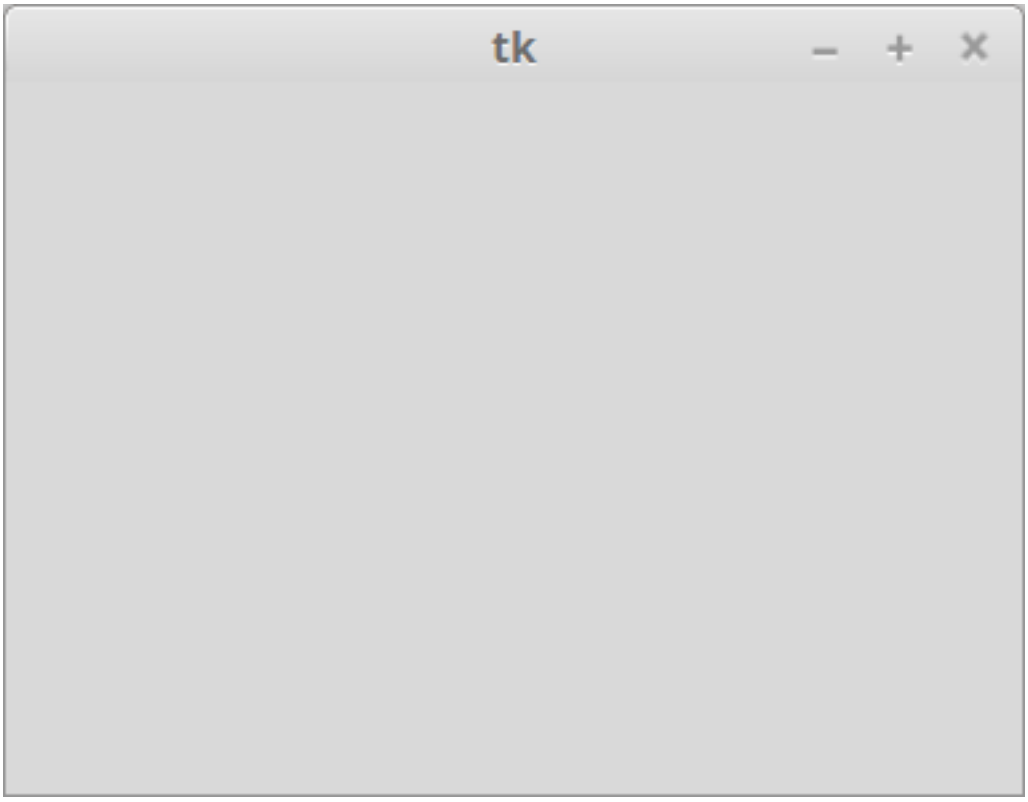
Príkazy majú postupne takéto vysvetlenie:

- `import tkinter` - vytvorí premennú `tkinter`, pomocou ktorej budeme mať prístup (bodkovou notáciou) k funkciám v module
- `canvas = tkinter.Canvas()` - vytvorí plátno grafickej aplikácie - priradili sme ho do premennej `canvas` (mohlo sa to volať hocijako inak, ale takto budeme lepšie rozumieť aj cudzím programom)
  - týmto vznikne malá grafická aplikácia (malé okno), aj plátno, ktoré sa ale zatiaľ v tejto aplikácii nenachádza
- `canvas.pack()` - umiestni naše plátno do grafickej aplikácie (do okna) - teraz je plátno pripravené, aby sme do neho mohli kresliť

- `tkinter.mainloop()` - grafická aplikácia vďaka tomuto príkazu v operačnom systéme naozaj „žije“, t.j. reaguje na klikanie, presúvanie, zmenu veľkosti, prekresľovanie, ...

Posledný príkaz `tkinter.mainloop()` môžeme pri spustení pod IDLE vynechať, lebo práve IDLE ho spraví za nás

Plátno (anglicky *canvas*) označuje ...



3.1 Grafické príkazy

Každý grafický príkaz vloží do plochy nejaký objekt, alebo s existujúcim objektom nejako manipuluje. Príkazy na vloženie nového grafického objektu do plochy majú tento tvar:

```
canvas.create_...(x, y, x, y, ..., pomenovane_parametre)
```

kde

- bodka za premennou `canvas` označuje **bodkovú notáciu**, teda budeme pracovať s funkciou, ktorá sa nachádza v (resp. patrí do) plátne `canvas`
- každé meno príkazu za bodkou začína **create_** a za tým nasleduje jedno z mien objektu:
  - **create_text** - vloží nejaký text (znakový reťazec)
  - **create_rectangle** - vloží obdĺžnik
  - **create_oval** - vloží elypsu
  - **create_line** - vloží úsečku alebo krivku zloženú z nadväzujúcich úsečiek
  - **create_polygon** - vyplní farbou oblasť, zadanú nejakou postupnosťou bodov

  - **create_image** - vloží obrázok (grafický súbor)
- parametre príkazov začínajú postupnosťou súradníc (dvojíc **x** a **y**), za ktorými môžu nasledovať pomenované parametre

3.1.1 Súradnicová sústava

Je oproti súradnicovej sústave, ktorú poznáme z matematiky, trochu pozmenená:

- **x**-ová os prechádza po hornej hrane plátna grafickej plochy zľava doprava
- **y**-ová os prechádza po ľavej hrane plátna zhora nadol
- počiatok (0, 0) je v ľavom hornom rohu plátna
- môžeme používať aj záporné súradnice, vtedy označujeme bod, ktorý je mimo grafickú plochu

Tak, ako sme vytvorili plátno pomocou `tkinter.Canvas()`, jej rozmery sú **379x265** pixelov, preto pravý dolný roh má súradnice (378, 264).

3.1.2 Grafický objekt text

Základný tvar príkazu je:

```
canvas.create_text(x, y, text='retazec')
```

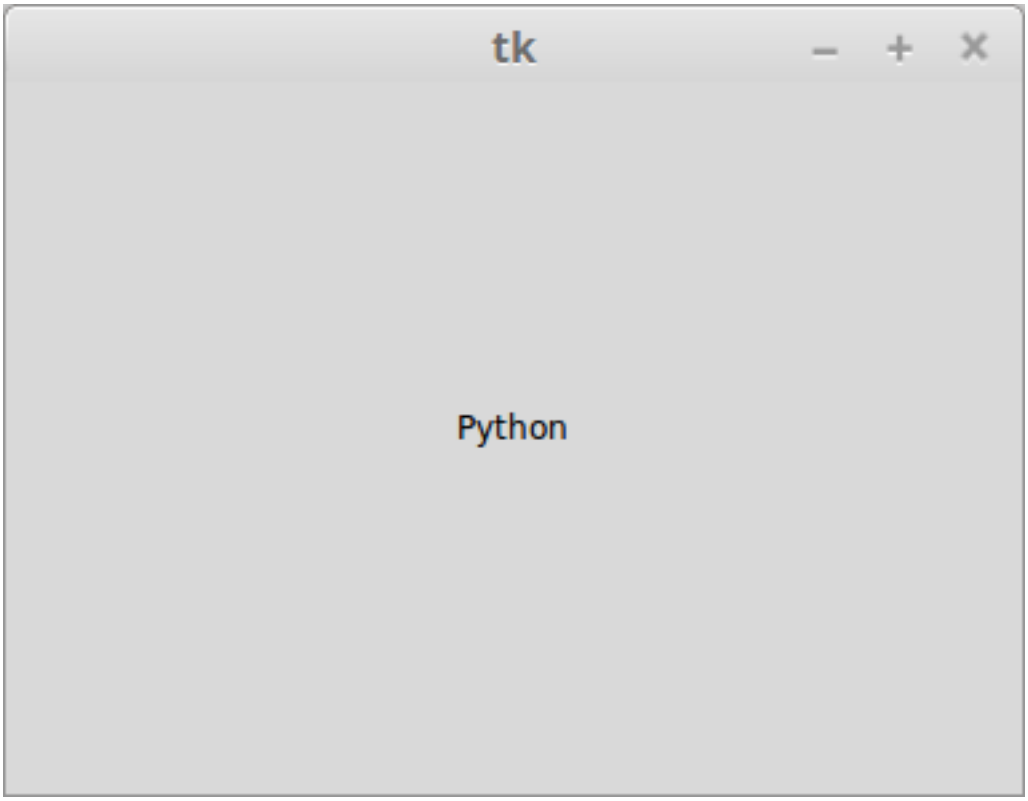
Pomocou tohto príkazu sa do grafickej plochy umiestni zadaný text, pričom súradnice `x` a `y` určujú polohu stredu textu. Napr.

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

canvas.create_text(190, 130, text='Python')
```

Vypíše text približne do stredu grafickej plochy. Tretí parameter `text='Python'` príkazu je tzv. **pomenovaný** parameter s menom **text** (toto nie je meno premennej, ale meno parametra).



Lenže tento text je veľmi malý. Použijeme ďalší pomenovaný parameter `font='meno vel'kost''`:

```
canvas.create_text(190, 130, text='Python', font='arial 40')
```

Teraz bude vypisovaný text výrazne väčší (nastavili sme mu font **Arial**). Do programu ešte vložíme ďalšie štyri texty, aby sme si zvykli na súradnú sústavu:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

canvas.create_text(190, 130, text='Python', font='arial 40')
canvas.create_text(40, 130, text='vlavo')
canvas.create_text(340, 130, text='vpravo')
canvas.create_text(190, 50, text='hore')
canvas.create_text(190, 210, text='dole')
```

a vidíme:

3.1.3 Grafický objekt obdĺžnik

Týmto príkazom sa umiestni obdĺžnik, ktorého strany sú rovnobežné so súradnicovými osami, t.j. hranami grafickej plochy. Základný tvar je:

```
canvas.create_rectangle(x1, y1, x2, y2)
```

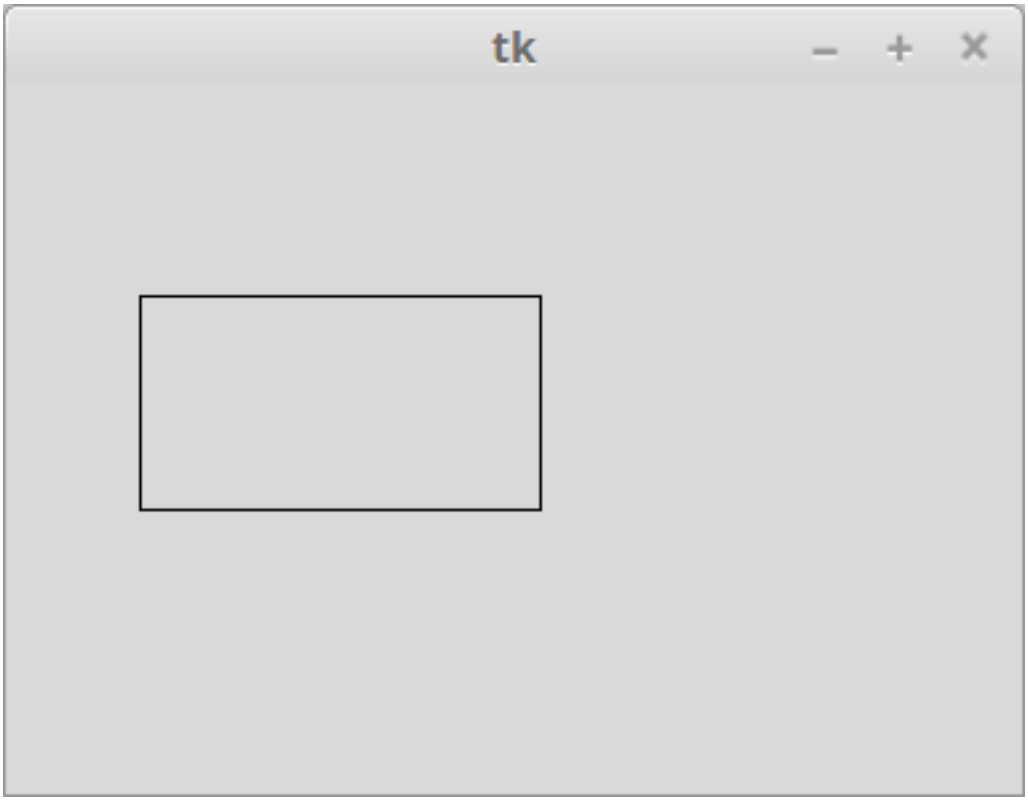
Hodnoty (x1, y1) a (x2, y2) sú súradnice dvoch ľubovoľných vrcholov tohto obdĺžnika, ktoré sú v obdĺžniku protiľahlé. Vyskúšajme:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

canvas.create_rectangle(50, 80, 200, 160)
```

Grafická plocha je teraz:

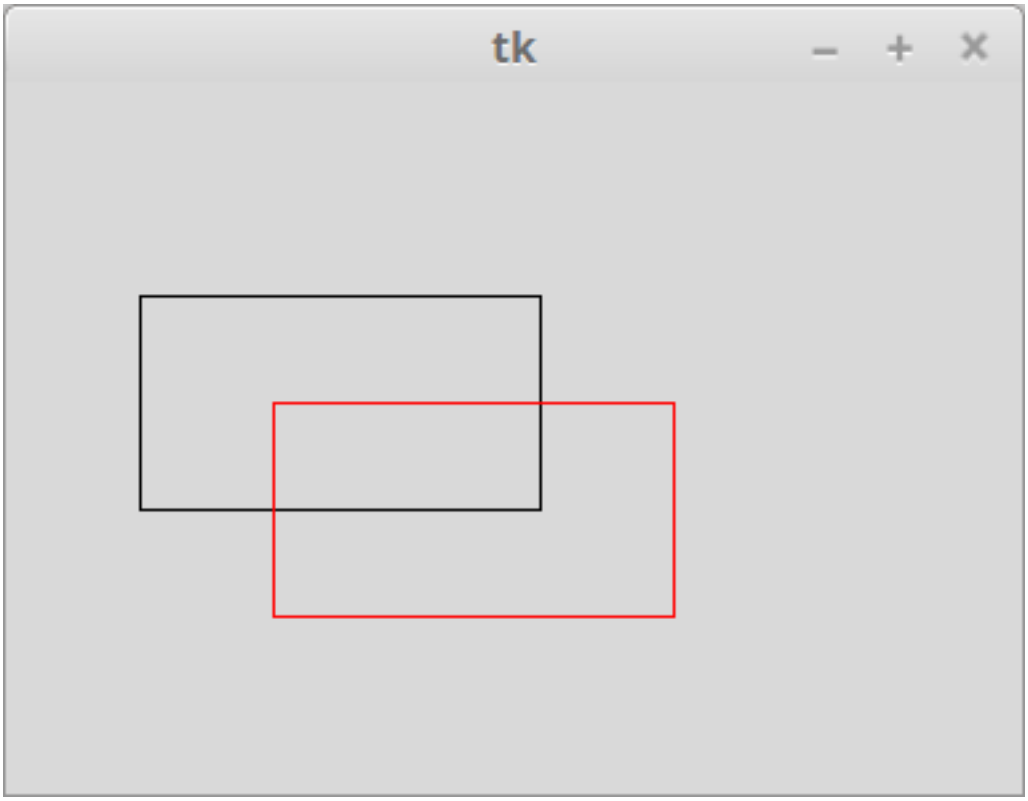


Uvedomte si, že `(50, 80)` a `(200, 160)` sú dva protiľahlé vrcholy obdĺžnika, Ďalšie dva vrcholy sú `(50, 160)` a `(200, 80)`. Veľkosti strán obdĺžnika sú `150` a `80`.

Pomocou pomenovaného parametra `outline='farba'` nastavíme farbu obrysu kresleného obdĺžnika, napr.

```
canvas.create_rectangle(100, 120, 250, 200, outline='red')
```

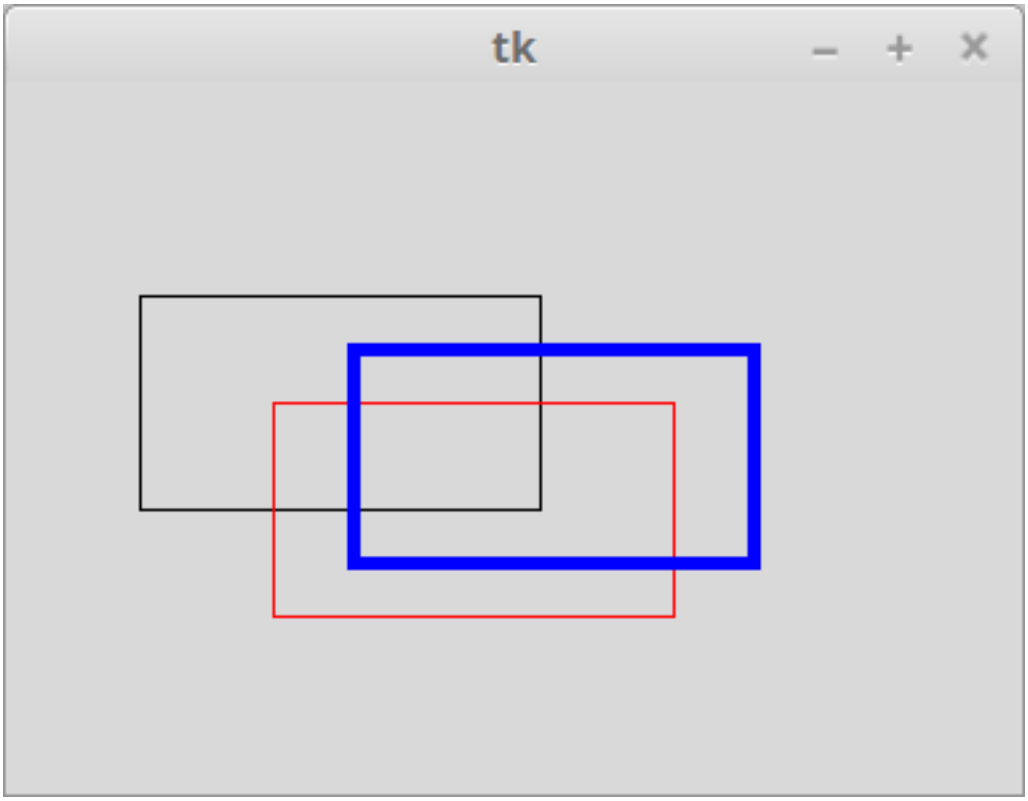
Tento nový obdĺžnik vyzerá takto:



ďalší pomenovaný parameter `width=číslo` nastaví hrúbku obrysu kresleného obdĺžnika. Napr.

```
canvas.create_rectangle(130, 100, 280, 180, outline='blue', width=5)
```

Tretí obdĺžnik má hrubší modrý obrys:



Asi najužitočnejším pomenovaným parametrom pre kreslenia obdĺžnika je nastavenie farby výplne `fill='farba'`. Všetky doterajšie obdĺžniky boli nakreslené bez výplne (hovoríme, že mali tzv. priesvitnú výplň). Štvrtý obdĺžnik bude mať žltú výplň:

```
canvas.create_rectangle(80, 70, 230, 150, fill='yellow')
```

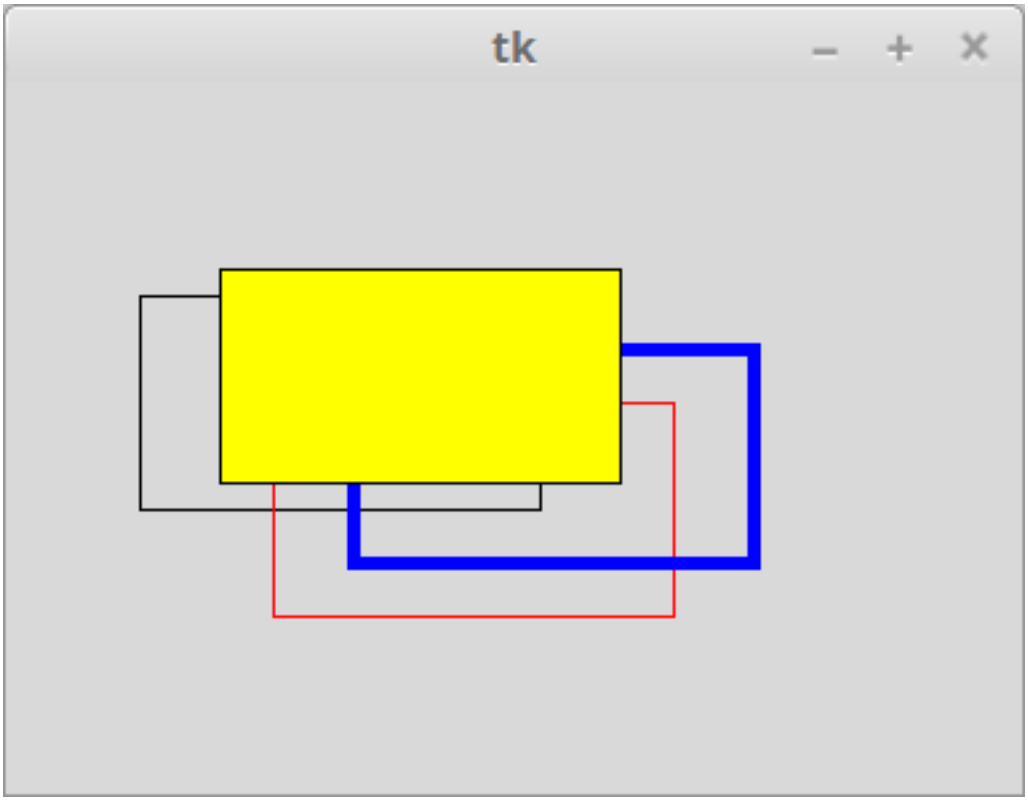
Všetky štyri obdĺžniky majú rovnaké rozmery ale rôzne nastavené niektoré parametre. Kompletný program:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

canvas.create_rectangle(50, 80, 200, 160)
canvas.create_rectangle(100, 120, 250, 200, outline='red')
canvas.create_rectangle(130, 100, 280, 180, outline='blue', width=5)
canvas.create_rectangle(80, 70, 230, 150, fill='yellow')
```

Grafická plocha:



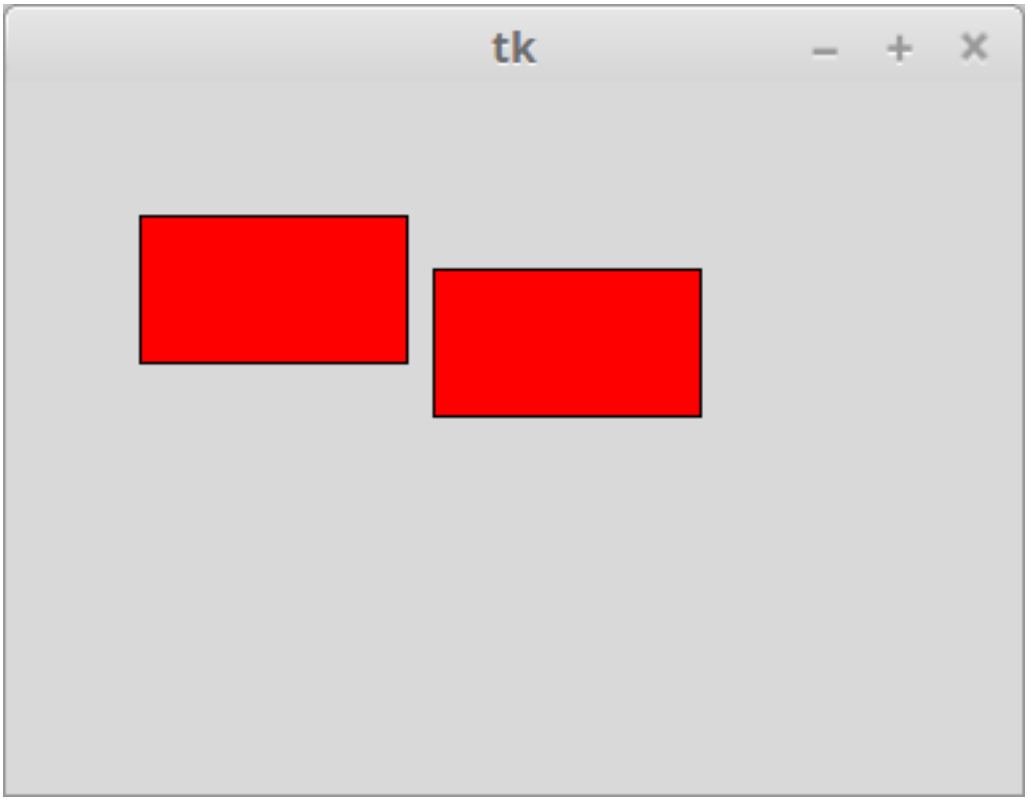
Pri kreslení grafických objektov je vhodné pracovať s premennými, v ktorých máme uložené napr. súradnice nejakých bodov, alebo veľkosti útvarov. Zapíšme nakreslenie obdĺžnika, v ktorom ľavý horný roh má súradnice `(x, y)`, jeho šírka je `s` a výška `v`:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

x, y = 50, 50
s, v = 100, 55
farba = 'red'
canvas.create_rectangle(x, y, x+s, y+v, fill=farba)
x, y = 160, 70
canvas.create_rectangle(x, y, x+s, y+v, fill=farba)
```

Všimnite si, že oba nakreslené obdĺžniky majú rovnaké rozmery a farbu, zmenili sme len polohu. Samotné vykreslenie `create_rectangle()` je v oboch prípadoch rovnaké:



Na minulej prednáške sme sa naučili pracovať s for-cyklami. Nakreslime rad rovnakých obdĺžnikov, ktoré budú uložené vedľa seba. Aby sa nám vedľa seba zmestilo viac obdĺžnikov, zväčšíme aj rozmery grafickej plochy:

```
import tkinter

canvas = tkinter.Canvas(width=600, height=300)
canvas.pack()

x, y = 10, 50
s, v = 30, 50
farba = 'red'
for i in range(15):
    canvas.create_rectangle(x, y, x+s, y+v, fill=farba)
    x = x + s + 2
```

Všimnite si riadok, v ktorom sa vytvára grafická plocha `tkinter.Canvas(...)`. Pridali sme do neho definovanie veľkosti plátna na šírku **600** a výšku **300**. Rad obdĺžnikov vyzerá takto:

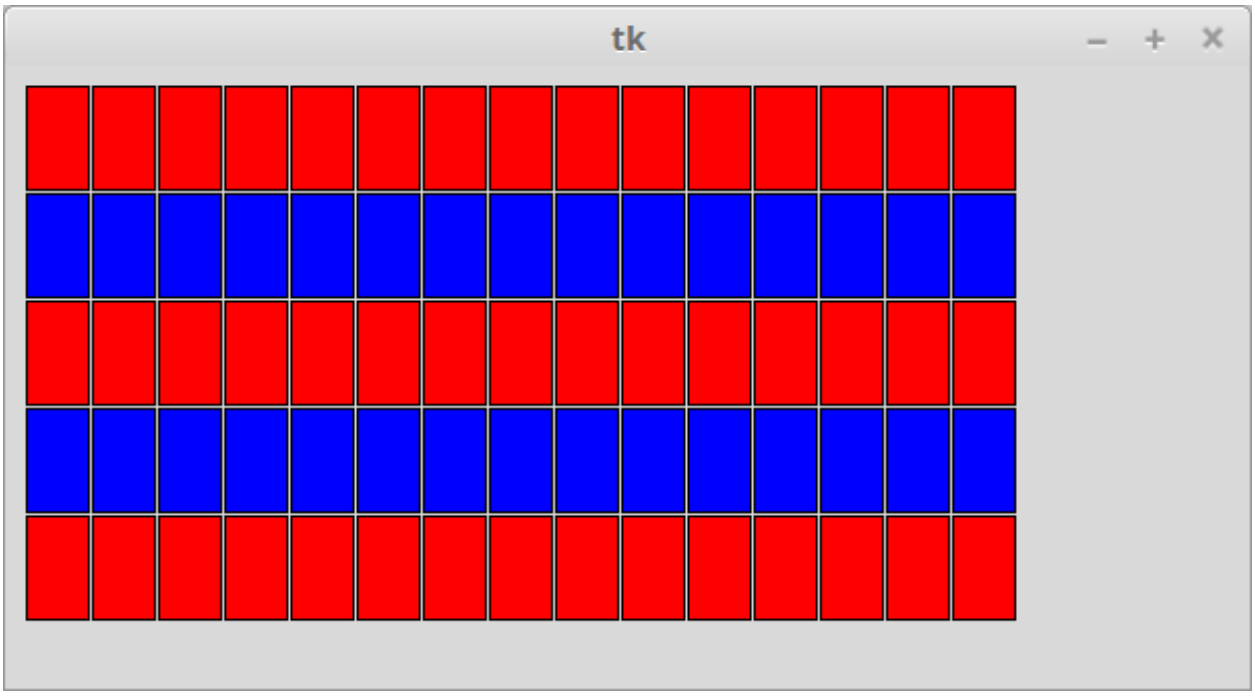
Pridajme ďalší cyklus, vďaka ktorému sa tento riadok obdĺžnikov nakreslí viackrát pod seba, pričom striedame dve farby výplne:

```
import tkinter

canvas = tkinter.Canvas(width=600, height=300)
canvas.pack()

y = 10
s, v = 30, 50
farba, farba2 = 'red', 'blue'
for j in range(5):
    x = 10
    for i in range(15):
        canvas.create_rectangle(x, y, x+s, y+v, fill=farba)
        x = x + s + 2
    y = y + v + 2
    farba, farba2 = farba2, farba
```

Dostávame:



3.1.4 Farby v grafickej ploche

Skôr, ako prejdeme na ďalšie grafické objekty, vysvetlime si, ako sa pre **tkinter** pracuje s farbami.

Zatiaľ sme používali tieto základné mená farieb (zopár sme ich ešte pridali):

- `'red'` - červená
- `'blue'` - modrá
- `'yellow'` - žltá
- `'green'` - zelená
- `'black'` - čierna
- `'white'` - biela
- `'gray'` - šedá

Preddefinovaných mien farieb je výrazne viac. **tkinter** používa zaužívané mená farieb z HTML stránok. Môžeme ich nájsť napr. na internete: HTML Color Names. Vyskúšajte niekoľko farieb z tohto zoznamu, ktoré sa vám páčia, či budú fungovať aj v našich programoch (napr. pri vyfarbovaní obdĺžnikov).

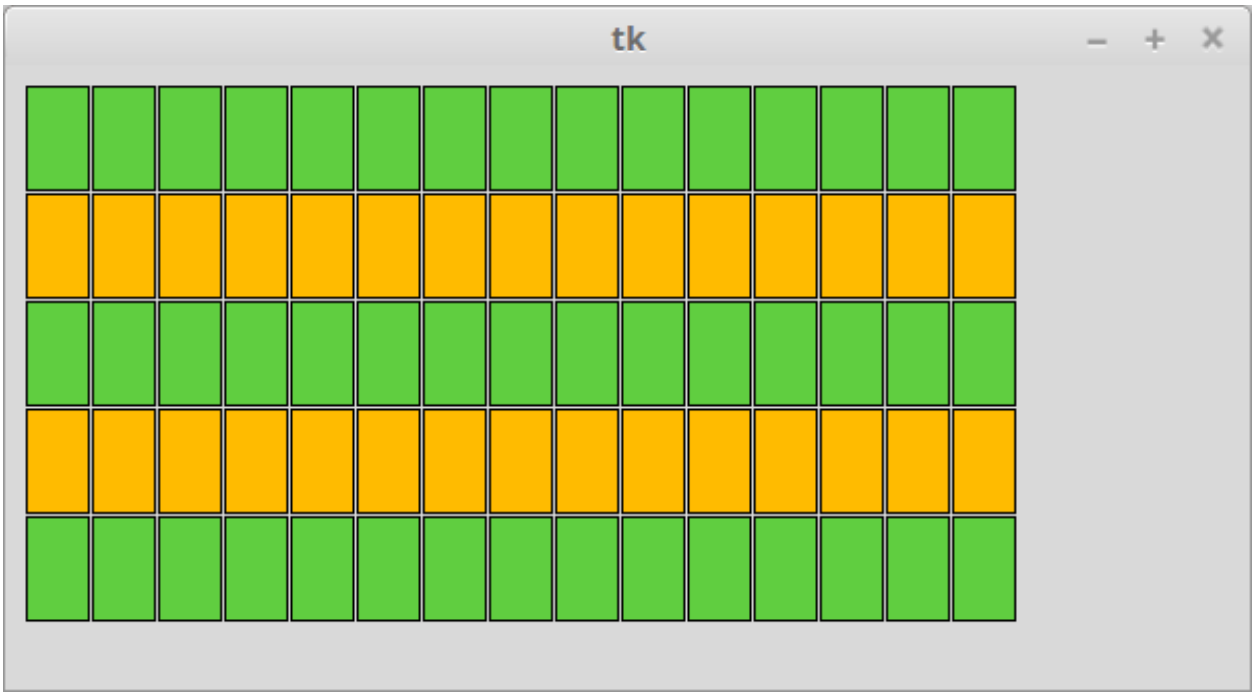
Iste ste už počuli o tzv. **RGB** modeli farieb. Môžeme predpokladať, že všetky farby v počítači sú namiešané z troch základných farieb: červenej, zelenej a modrej (teda **Red**, **Green**, **Blue**). Farba závisí od toho, ako je v nej zastúpená každá z týchto troch farieb. Zastúpenie jednotlivej farby vyjadrujeme číslom od 0 do 255 (zmestí sa do jedného bajtu, teda ako 2-ciferné šestnástkové číslo). Napr. žltá farba vznikne, ak namiešame 255 červenej, 255 zelenej a 0 modrej. Ak budeme zastúpenie každej farby trochu meniť, napríklad 250 červenej, 240 zelenej a hoci 100 modrej, stále to bude žltá, ale v inom odtieni.

Pri kreslení v **tkinter** zadávame farby buď menami alebo zakódujeme ich RGB-trojicu do šestnástkovej sústavy ako 3 dvojciferné čísla (spolu 6 cifier). Pred takéto šestnástkové číslo musíme ešte na začiatok pridať znak ‚#‘, aby to tkinter odlíšil od mena farby. Môžete s týmto poexperimentovať napr. na stránke: Colors RGB - RGB Calculator. Na tejto stránke pri nastavovaní rôznych hodnôt pre zložky RGB dostávame rôzne výsledné farby a zakaždým vidíme zodpovedajúcu 16-ovú reprezentáciu ako 6-ciferné 16-ové číslo, pred ktorým je znak `'#'`.

Vyskúšajte takto vytvoriť nejaké dve zaujímavé farby a otestujte to v našom poslednom programe s piatimi radmi obdĺžnikov. Napr. ak nastavíte:

```
farba, farba2 = '#60ce40', '#ffbb00'
```

dostaneme takýto obrázok:



Ak máme dané nejaké tri čísla, ktoré reprezentujú **RGB**, napr.

```
r, g, b = 205, 92, 9
```

a chceme z nich vyrobiť farbu pre **tkinter**, môžeme využiť formátovací reťazec, ktorý dokáže previesť číslo do 16-ovej sústavy:

```
f'#{r:02x}{g:02x}{b:02x}'
```

Formátovací parameter `02x` na tomto mieste znamená, že dané číslo sa prevedie do 16-ovej sústavy s dvoma ciframi, pričom sa toto číslo doplní zľava `0`, ak bolo iba jednociferné. Vďaka tomuto zápisu vieme v našich programoch veľmi elegantne vygenerovať ľubovoľné **RGB**. Pozmeňme tento program tak, že namiesto striedania dvoch farieb, každý jeden obdĺžnik zafarbíme náhodnou farbou. Zrejme každá farebná zložka bude náhodné číslo od 0 do 255, t.j. `random.randrange(256)`. Ešte sme upravili obdĺžniky tak, aby mali rovnakú výšku ako šírku, teda budú to štvorčeky:

```
import tkinter
import random

canvas = tkinter.Canvas(width=600, height=300)
canvas.pack()

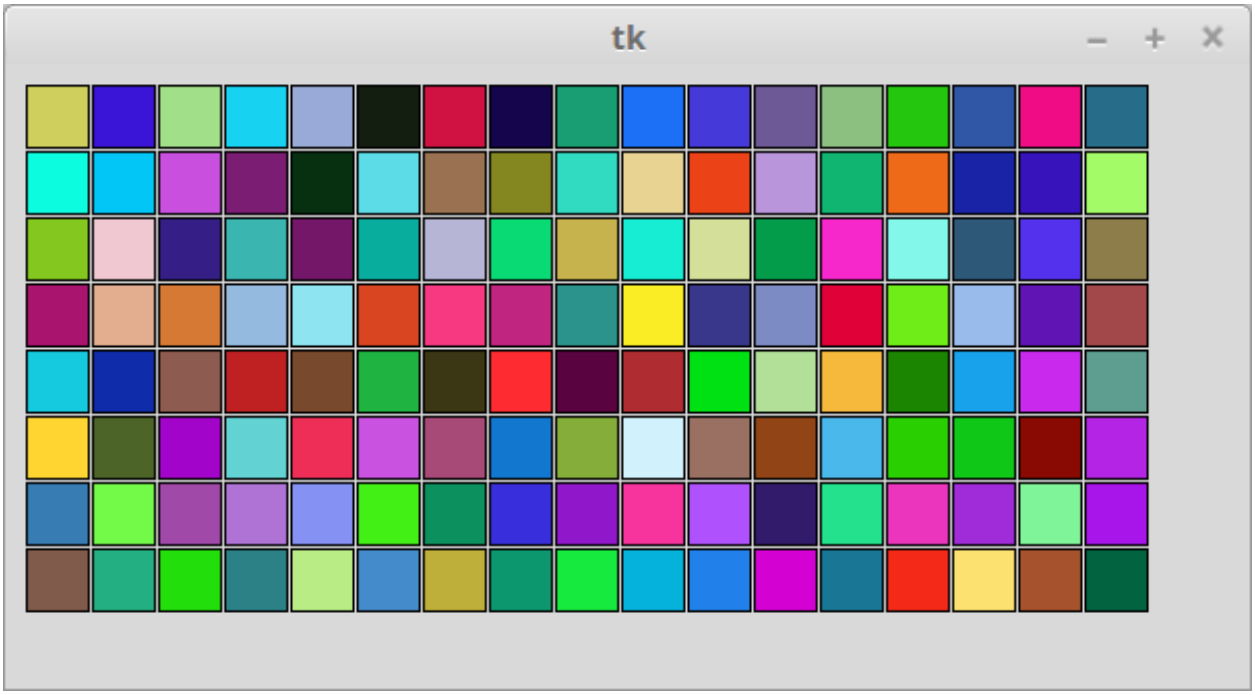
x0, y0 = 10, 10
s = 30
for y in range(y0, 250, s+2):
    for x in range(x0, 550, s+2):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        r = random.randrange(256)
        g = random.randrange(256)
        b = random.randrange(256)
        farba = f'#{r:02x}{g:02x}{b:02x}'
        canvas.create_rectangle(x, y, x+s, y+s, fill=farba)
```

Všimnite si, že sme tu aj trochu inak organizovali for-cykly. Dostávame nejako takto zafarbené štvorčeky:



3.1.5 Grafický objekt elipsa

Kreslenie elíps je v **tkinter** veľmi blízke kresleniu obdĺžnikov: elipsu budeme totiž definovať prostredníctvom „neviditeľného“ obdĺžnika, v ktorom bude táto elipsa vpísaná. Takže, ak máme program, v ktorom máme nakreslené nejaké obdĺžniky a príkazy `create_rectangle()` nahradíme `create_oval`, tak presne na týchto miestach sa nakreslia elipsy.

Preto

```
canvas.create_oval(x1, y1, x2, y2)
```

Nakreslí elipsu, ktorá by bola vpísaná v

```
canvas.create_rectangle(x1, y1, x2, y2)
```

Môžeme vyskúšať:

```
import tkinter

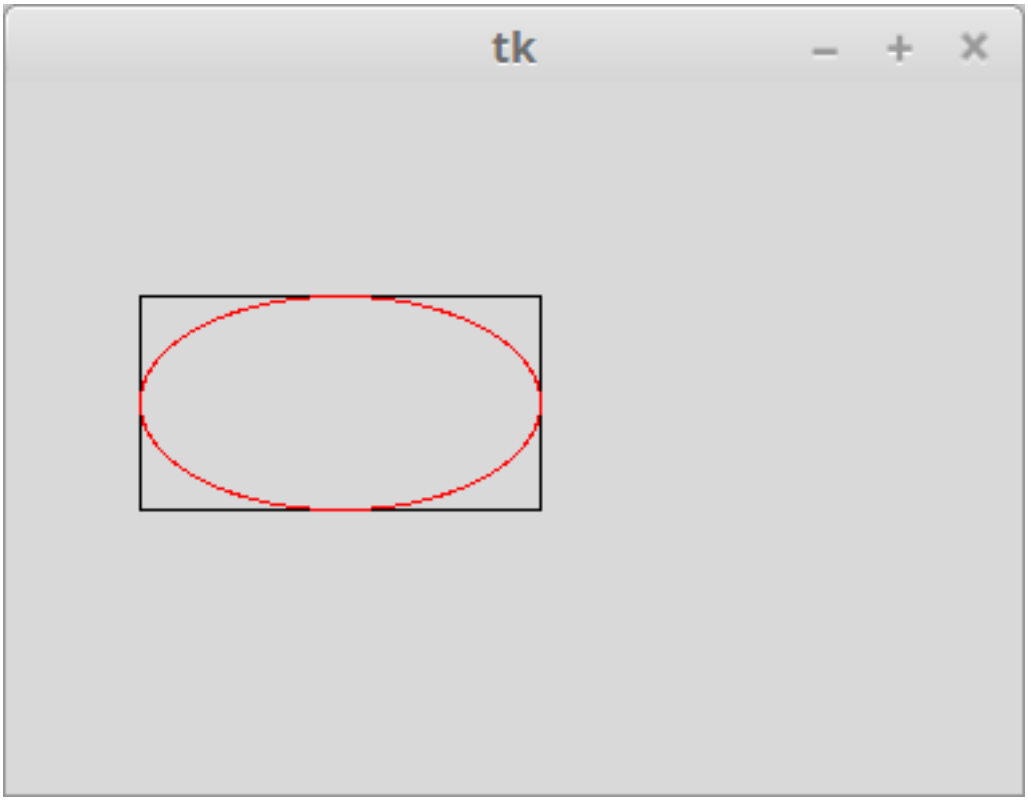
canvas = tkinter.Canvas()
canvas.pack()
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
canvas.create_rectangle(50, 80, 200, 160)
canvas.create_oval(50, 80, 200, 160, outline='red')
```

a vidíme:



Samozrejme, že keď vpíšeme elipsu do štvorca, dostávame kružnicu s priemerom veľkosti strany štvorca. Otestujeme kreslenie niekoľkých kružníc, ktoré majú spoločný stred ale rôzne polomery. Ak by sme mali daný stred `(x, y)` a polomer `r`, tak kružnicu nakreslíme:

```
canvas.create_oval(x-r, y-r, x+r, y+r)
```

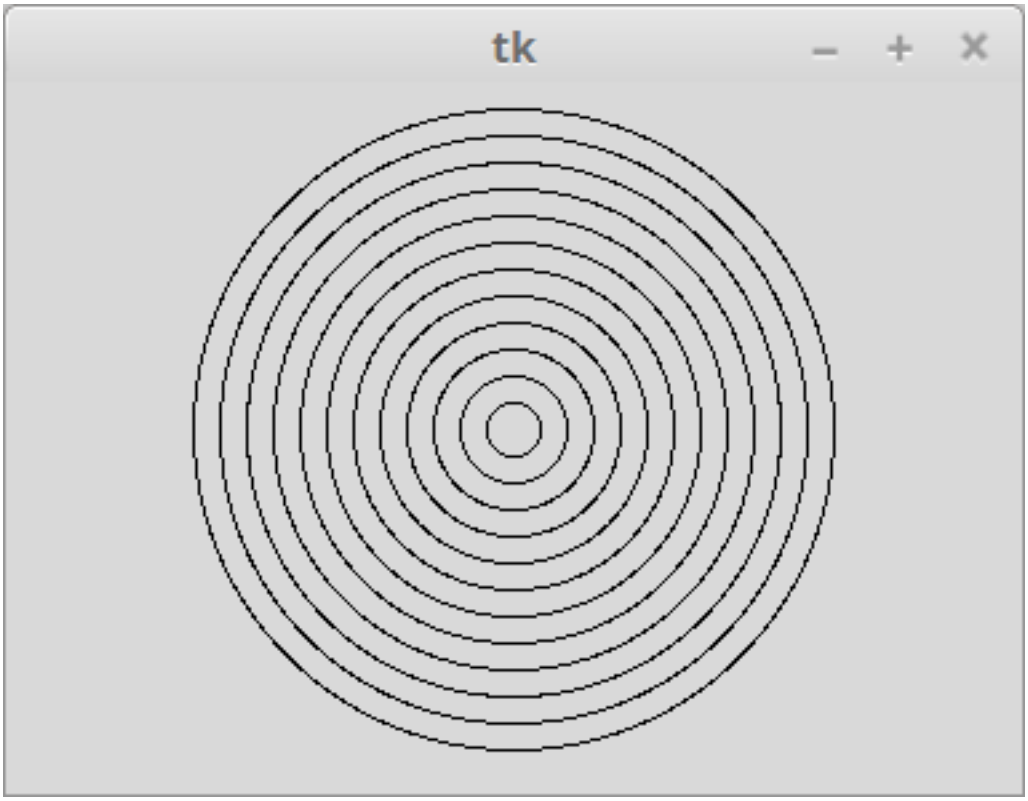
Vyskúšajme:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

x, y = 190, 130
for r in range(10, 130, 10):
    canvas.create_oval(x-r, y-r, x+r, y+r)
```

a sústredné kružnice vyzerajú takto:



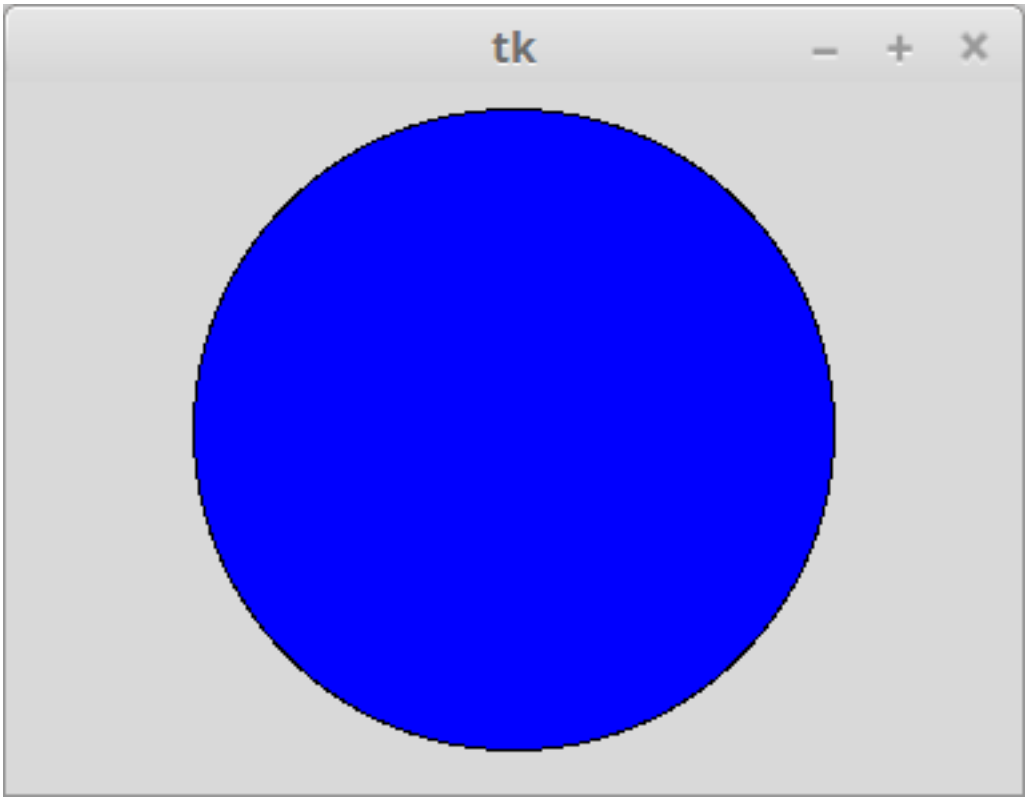
Keďže ďalšie parametre príkazu `create_oval` sú rovnaké ako pre obdĺžniky, môžeme vyfarbiť tieto kružnice na striedačku dvoma farbami:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

x, y = 190, 130
f1, f2 = 'red', 'blue'
for r in range(10, 130, 10):
    canvas.create_oval(x-r, y-r, x+r, y+r, fill=f1)
    f1, f2 = f2, f1
```

Žiaľ, vidíme len naposledy nakreslený najväčší kruh:



Keďže sme začali kresliť od najmenších kruhov, každý ďalší väčší prekryje všetky predchádzajúce.

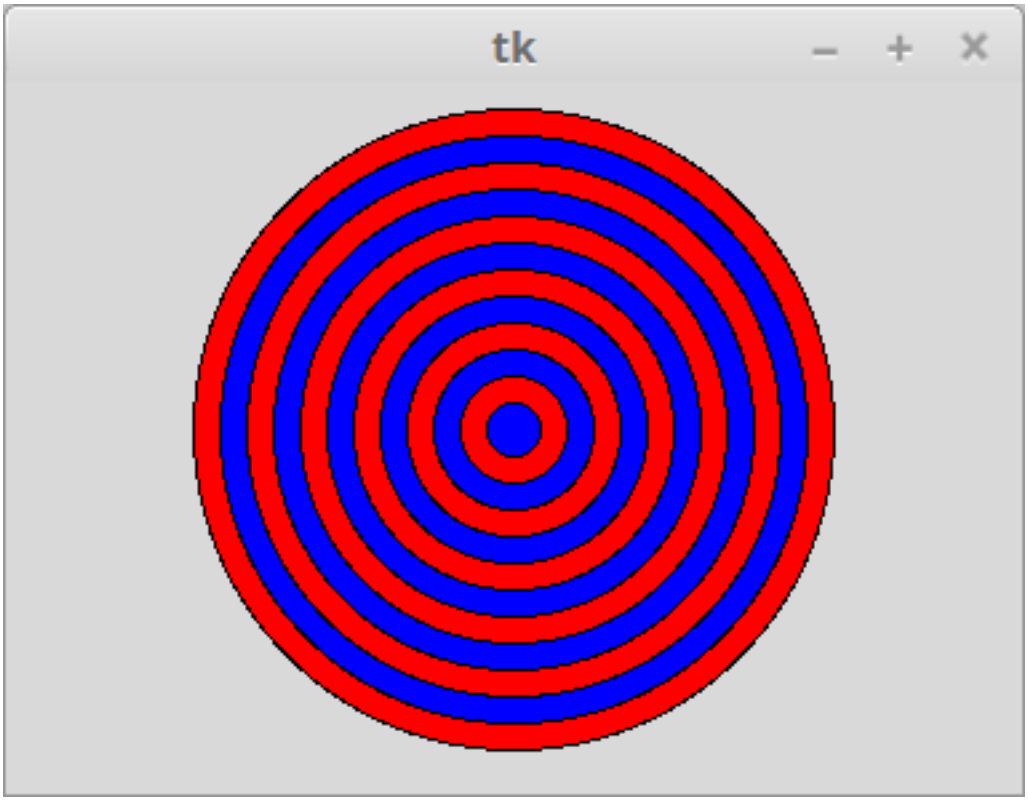
Asi najelegantnejšie riešenie tohto problému bude to, keď nakreslíme všetky kruhy v opačnom poradí: od najväčšieho po najmenší. Využijeme štandardnú funkciu `reversed()`, pomocou ktorej otočíme poradie postupnosti čísel `range(10, 130, 10)`:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

x, y = 190, 130
f1, f2 = 'red', 'blue'
for r in reversed(range(10, 130, 10)):
    canvas.create_oval(x-r, y-r, x+r, y+r, fill=f1)
    f1, f2 = f2, f1
```

a očakávaný výsledok:



Využijeme program, v ktorom sme nakreslili niekoľko radov náhodne zafarbených štvorčekov. Nahradíme v ňom `create_rectangle` volaním `create_oval`:

```
import tkinter
import random

canvas = tkinter.Canvas(width=600, height=300)
canvas.pack()

x0, y0 = 10, 10
s = 30
for y in range(y0, 250, s+2):
    for x in range(x0, 550, s+2):
        r = random.randrange(256)
        g = random.randrange(256)
        b = random.randrange(256)
        farba = f'#{r:02x}{g:02x}{b:02x}'
        canvas.create_oval(x, y, x+s, y+s, fill=farba)
```

Dostávame:

3.1.6 Grafický objekt pre úsečky a lomené čiary

Základný formát príkazu:

```
canvas.create_line(x1, y1, x2, y2, ...)
```

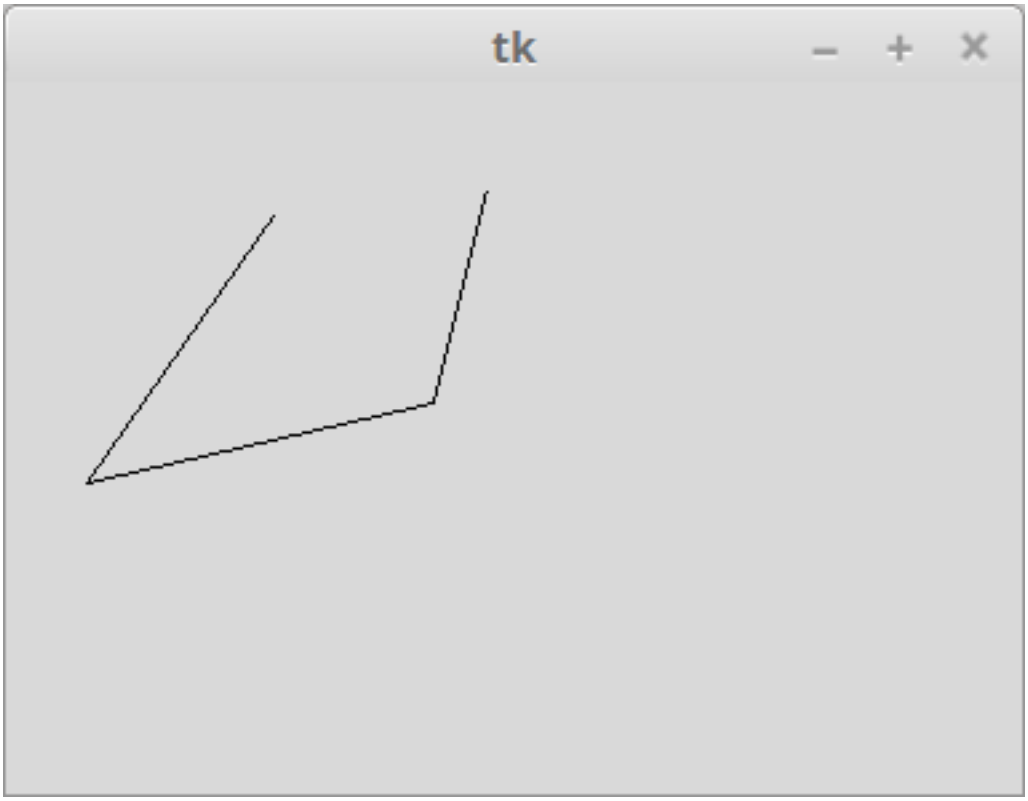
Príkaz musí dostať aspoň dve dvojice súradníc. Grafický objekt potom postupne pospája všetky tieto body a vytvorí jednu lomenú čiaru. Napr.

```
canvas.create_line(100, 50, 30, 150)
```

nakreslí jednu úsečku s koncovými vrcholmi `(100, 50)`, `(30, 150)`. Ak do príkazu uvedieme viac bodov, napr.

```
canvas.create_line(100, 50, 30, 150, 160, 120, 180, 40)
```

nakreslí sa lomená čiara (4 vrcholy, čo sú 3 úsečky):



Skúsme túto istú lomenú čiaru zapísať trochu inak. Najprv pomenujme jednotlivé vrcholy a zapíšme:

```
a = (100, 50)
b = (30, 150)
c = (160, 120)
d = (180, 40)
```

Z matematiky by sme mohli týmto zápisom rozumieť tak, že napr. `a` je vrchol, ktorého súradnice sú `(100, 500)`. Naozaj to v Pythone funguje a premenná `a` naozaj obsahuje **dvojicu** čísel. Čo je ešte lepšie, že takto označené vrcholy (premenné `a`, `b`, ...) môžeme použiť pri kreslení lomenej čiary:

```
canvas.create_line(a, b, c, d)
```

Čo je ešte zaujímavejšie, takéto dvojice môžeme použiť napr. aj vo for-cykle. Zakreslime do každého vrcholu tejto krivky červený znak plus:

```
import tkinter

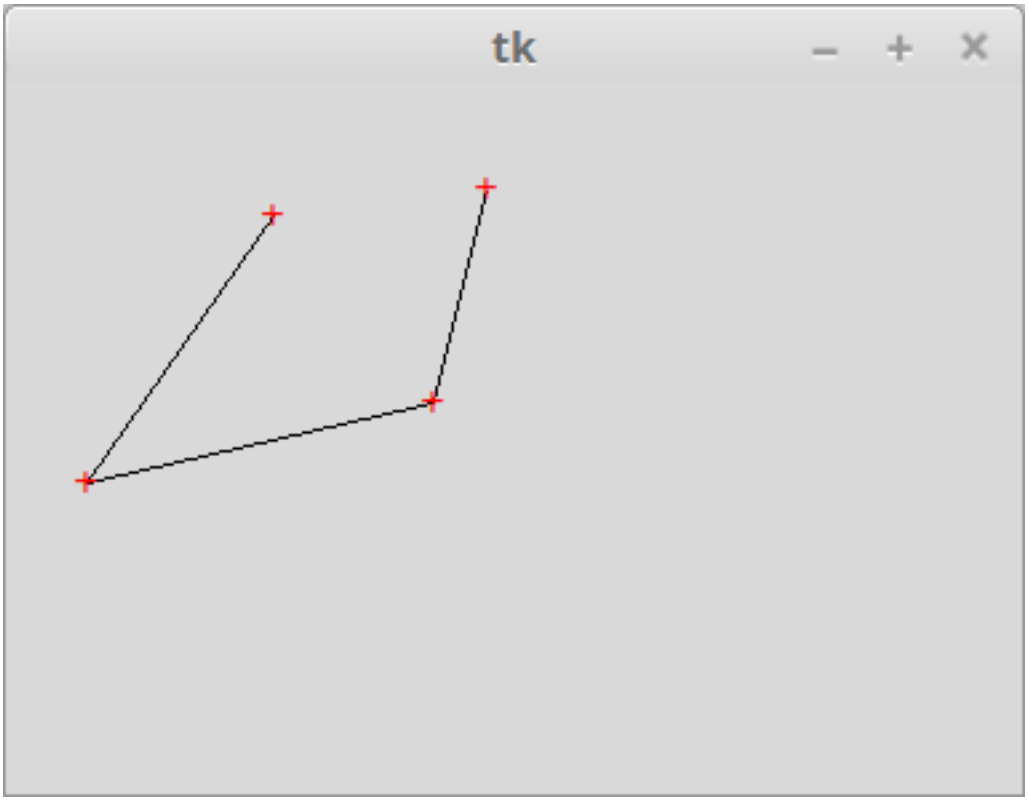
canvas = tkinter.Canvas()
canvas.pack()

a = (100, 50)
b = (30, 150)
c = (160, 120)
d = (180, 40)
canvas.create_line(a, b, c, d)

for bod in a, b, c, d:
    canvas.create_text(bod, text='+', fill='red')
```

A vidíme, že takéto **dvojice** naozaj fungujú ako vymenované hodnoty pre for-cyklus. Premenná cyklu `bod` potom

postupne nadobúda tieto hodntoty, teda dvojice čísel. Tento bod sa potom použije pre vypisovanie znaku pomocou `create_text`:



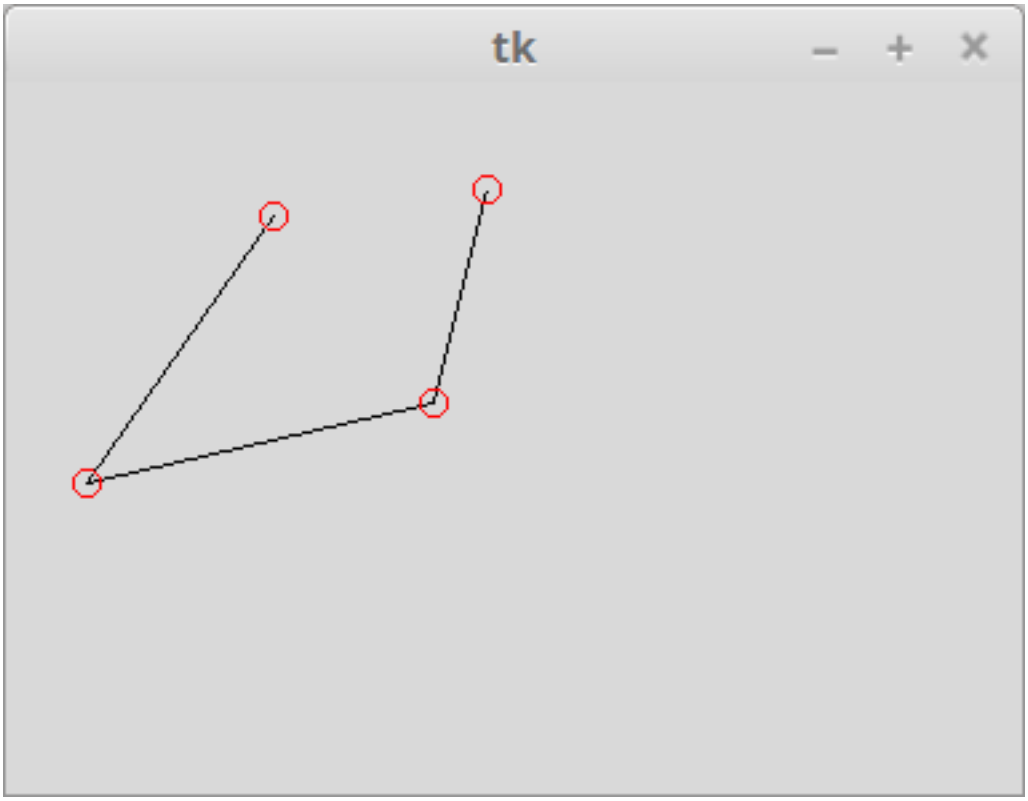
Ak by sme tento postup chceli použiť pre kreslenie malých krúžkov v každom vrchole, museli by sme v tele cyklu premennú `bod`, ktorá je typu **dvojica** čísel, previesť na dve premenné `x` a `y`. Našťastie funguje priraďovací príkaz, ktorý dokáže **dvojicu** priradiť do dvoch premenných. Môžeme zapísať:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

a = (100, 50)
b = (30, 150)
c = (160, 120)
d = (180, 40)
canvas.create_line(a, b, c, d)

for bod in a, b, c, d:
    x, y = bod
    canvas.create_oval(x-5, y-5, x+5, y+5, outline='red')
```



Tento cyklus môžeme zapísať ešte o trochu zaujímavejšie:

```
for x, y in a, b, c, d:
    canvas.create_oval(x-5, y-5, x+5, y+5, outline='red')
```

Namiesto jednej premennej cyklu (premenná `bod`) sme tu uviedli dve (premenné `x` a `y`). Python v tomto prípade postupne berie vymenované hodnoty a každú jednu **rozoberie** na dve čísla a priradí ich do `x` a `y`.

3.1.7 Grafický objekt pre polygon

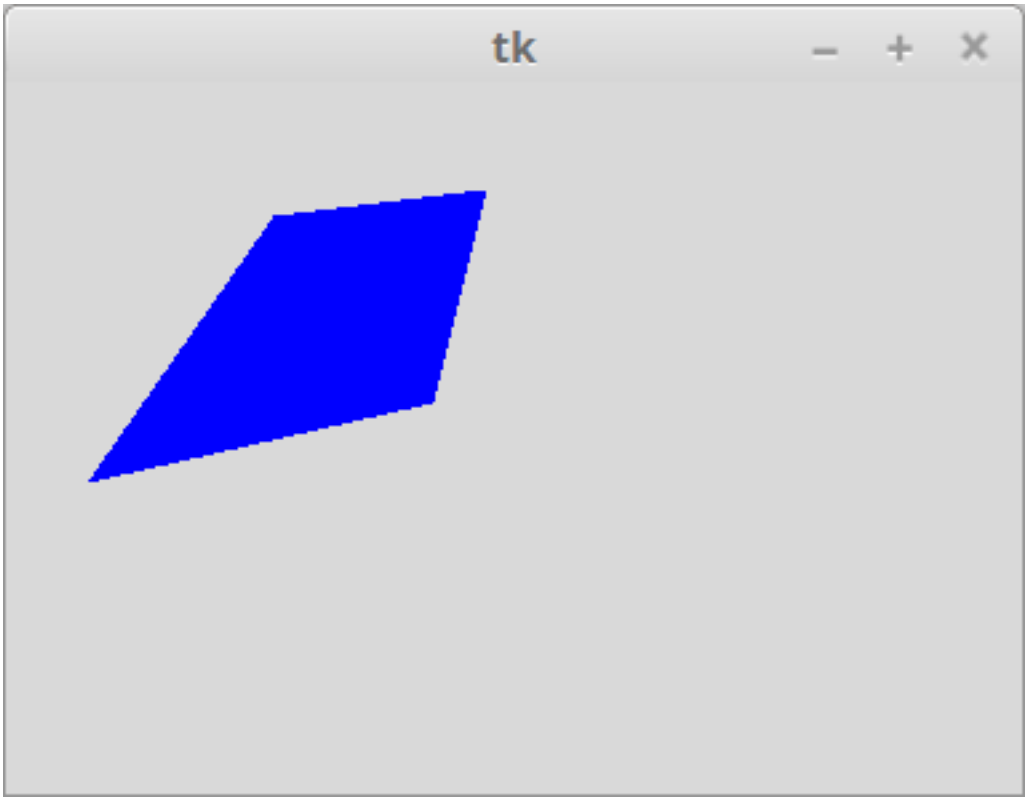
Polygonom voláme oblasť, ktorá je ohraničená zadanou lomenou čiarou (aspoň s troma vrcholmi) a vyplní sa nejakou farbou. Body zadávame podobne ako pre `create_line`, len dva body by bolo zrejme málo. Táto oblasť bude zafarbená čiernou farbou, prípadne ju môžeme zmeniť, napr.

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

a = (100, 50)
b = (30, 150)
c = (160, 120)
d = (180, 40)
canvas.create_polygon(a, b, c, d, fill='blue')
```

a vyzerá to takto:



3.1.8 Grafický objekt obrázok

Aby sme do plochy mohli nakresliť nejaký obrázok, musíme najprv vytvoriť „obrázkový objekt“ (pomocou `tkinter.PhotoImage()` prečítať obrázok zo súboru) a až tento poslať ako parameter do príkazu na kreslenie obrázkov `canvas.create_image()`.

Obrázkový objekt vytvoríme špeciálnym príkazom:

```
premenna = tkinter.PhotoImage(file='meno suboru')
```

v ktorom `meno suboru` je súbor s obrázkom vo formáte **png** alebo **gif**. Takýto obrázkový objekt môžeme potom vykresliť do grafickej plochy ľubovoľný počet-krát.

Samotná funkcia `canvas.create_image()` na vykreslenie obrázka má tri parametre: prvé dva sú súradnice stredu vykresľovaného obrázka a ďalší pomenovaný parameter určuje obrázkový objekt. Príkaz má tvar:

```
canvas.create_image(x, y, image=premenna)
```

Napr.

```
obr = tkinter.PhotoImage(file='python.png')
canvas.create_image(500, 100, image=obr)
```

3.1.9 Parametre grafickej plochy

Pri vytváraní grafickej plochy (pomocou `tkinter.Canvas()`) môžeme nastaviť veľkosť plochy ale aj farbu pozadia grafickej plochy. Môžeme uviesť tieto parametre:

- `bg=` nastavuje farbu pozadia (z anglického „background“)
- `width=` nastavuje šírku grafickej plochy
- `height=` výšku plochy

Napr.

```
canvas = tkinter.Canvas(bg='white', width=400, height=200)
```

Vytvorí bielu grafickú plochu, ktorá má šírku 400 a výšku 200.

3.2 Zmeny nakreslených útvarov

Všetky útvary, ktoré kreslíme do grafickej plochy si systém pamätá tak, že ich dokáže dodatočne meniť (napr. ich farbu), posúvať po ploche, ale aj mazať. Všetky útvary sú v ploche vykresľované presne v tom poradí, ako sme zadávali jednotlivé grafické príkazy: skôr nakreslené útvary sú pod neskôr nakreslenými a môžu ich prekrývať.

Každý grafický príkaz (napr. `canvas.create_line()`) je v skutočnosti funkciou, ktorá vracia celé číslo - identifikátor nakresleného útvaru. Toto číslo nám umožní neskoršie modifikovanie, resp. jeho zmazanie.

Okrem tohto identifikačného čísla (každý objekt má jedinečné číslo), môžeme grafickým objektom prideľovať štítky (pomenovaný parameter `tag=`) a potom môžeme vo všetkých nasledovných príkazov namiesto identifikátora používať tento pridelený štítok. Pritom rôzne objekty môžu mať rovnaké štítky a tým môžeme jedným modifikačným príkazom zmeniť naraz viac objektov.

3.2.1 Zrušenie nakresleného útvaru

Na zrušenie ľubovoľných grafických objektov z grafickej plochy slúži funkcia:

```
canvas.delete(oznacenie)
```

kde parameter `oznacenie` je jedno z

- číselný identifikátor
- pridelený štítok (`tag`)
- reťazec `'all'` označuje všetky útvaru v ploche

Napr.

```
id1 = canvas.create_line(10, 20, 30, 40)
id2 = canvas.create_oval(10, 20, 30, 40)
canvas.create_text(100, 100, text='ahoj', tag='t')
canvas.create_rectangle(80, 90, 120, 110, fill='gray', tag='t')
...
canvas.delete(id1)
canvas.delete('t')
```

Zmaže prvý grafický objekt, t.j. úsečku, pričom druhý objekt kružnica ostáva bez zmeny. Druhý príkaz zmaže oba textový objekt aj obdĺžnik, ktorým sme pridelili štítok ‚t‘.

3.2.2 Posúvanie útvarov

Objekty môžeme posúvať určením buď identifikátora útvaru alebo jeho štítku (vtedy ich môže byť aj viac). Ostatné útvary sa pri tom nehýbu. Tvar funkcie je

```
canvas.move(oznacenie, dx, dy)
```

kde

- `označenie` je buď identifikátor alebo štítok grafických útvarov (bude fungovať aj reťazec `'all'`)
- `dx` a `dy` označujú číselné hodnoty zmeny súradníc útvaru, t.j. posun v smere osi **x** a v smere osi **y**

Napr.

```
id1 = canvas.create_line(10, 20, 30, 40)
id2 = canvas.create_oval(10, 20, 30, 40)
canvas.create_text(100, 100, text='ahoj', tag='t')
canvas.create_rectangle(80, 90, 120, 110, fill='gray', tag='t')
...
canvas.move(id1, -5, 10)
canvas.move('t', -5, 10)
```

posunie prvý nakreslený útvar, teda úsečku, druhý útvar (kružnicu) pri tom nehýbe, zároveň s tým posunie naraz obdĺžnik s textom.

3.2.3 Zmena parametrov útvaru

Táto funkcia umožňuje meniť ľubovoľné pomenované parametre už nakresleným útvarom. Jeho tvar je:

```
canvas.itemconfig(oznacenie, parametre)
```

kde

- `označenie` je buď identifikátor alebo štítok grafických útvarov
- `parametre` sú pomenované parametre v rovnakom formáte, aký bol pri ich vytváraní

3.2.4 Zhrnutie pomenovaných parametrov

`canvas.create_text()`	
	`text=` vypisovaný text
	`font=` písmo a veľkosť
	`fill=` farba textu
	`angle=` uhol otočenia
`canvas.create_rectangle()`	
	`width=` hrúbka obrysu
	`outline=` farba obrysu
	`fill=` farba výplne
`canvas.create_oval()`	
	`width=` hrúbka obrysu
	`outline=` farba obrysu
	`fill=` farba výplne
`canvas.create_line()`	
	`width=` hrúbka obrysu
	`fill=` farba obrysu
	`arrow=` šípka: jedno z `'first'`, `'last'`, `'both'`
`canvas.create_polygon()`	
	`width=` hrúbka obrysu
	`outline=` farba obrysu
	`fill=` farba výplne
`canvas.create_image()`	
	`image=` obrázkový objekt

3.2.5 Zmena súradníc

Okrem posúvania útvarov im môžeme zmeniť aj ich kompletnú postupnosť súradníc. Napr. pre `canvas.create_line()` alebo `canvas.create_polygon()` môžeme zmeniť aj počet bodov útvaru. Tvar tejto funkcie je:

```
canvas.coords(oznacenie, postupnost)
```

kde

- `označenie` je buď identifikátor alebo štítok grafických útvarov
- `postupnost` je ľubovoľná postupnosť súradníc, ktorá je vhodná pre daný útvar - táto postupnosť musí obsahovať párny počet čísel (celých alebo desatinných)

Napr.

```
i1 = canvas.create_line(10, 20, 30, 40)
canvas.coords(i1, 30, 40, 50, 60, 70, 90)
```

3.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Napíšte program, ktorý vypíše nejaký zadaný text na rôzne miesta grafickej plochy tak, aby každý mal iný font, inú farbu aj veľkosť.
   - napr.

```
text = 'programovanie'
# vykreslenie tohto textu rôznymi fontami
```

   - ak sa meno fontu skladá z viac slov, napr. `'Cooper Black'`, `'Brush Script MT'` alebo `'Courier New'`... musíme pomenovaný parameter `font='...'` zmeniť na zápis napr. takto `font=('Courier New', 40)`
2. Napíšte program, ktorý nakreslí pyramídu: tvoria ju 3 na sebe položené obdĺžniky veľkosti 100x20, 60x20, 20x20, tieto obdĺžniky sú vycentrované. Zafarbite ich tromi rôznymi farbami.
3. Napíšte program, ktorý nakreslí podobnú pyramídu z predchádzajúceho príkladu, pričom všetky obdĺžniky majú výšku 10 a ich šírky sú postupne čísla 200, 180, 160, ... 60, 40, 20. Obdĺžniky zafarbujte náhodnými farbami.
4. Napíšte program, ktorý nakreslí rad `n` kruhov: všetky sú tesne vedľa seba a striedajú sa polomery 20 a 30. Zvoľte nejaké tri farby, ktoré sa pritom striedajú. Mená farieb vyberte zo stránky Colors RGB tak, aby boli v 16-ovom tvare `'#......'`. Kreslenie kruhov zapíšte tak, aby sa nekreslil ich čierny obrys.
   - napr.

```
n = 12
r = 15
# vykreslenie n kruhov vedla seba
```

5. Napíšte program, ktorý pre zadaný polomer `r` nakreslí do stredu grafickej plochy kružnicu a do nej pomocou `canvas.create_rectangle()` nakreslí vpísaný štvorec (jeho vrcholy musia ležať na obvode kružnice).
   - napr.

```
r = 180
# nakresli kruznicu s polomerom r a do nej vpisany stvorec
```

6. Napíšte program, ktorý pre tri zadané hodnoty `r1`, `r2`, `r3` nakreslí snehuliaka, ktorý sa skladá z troch „skoro“ bielych kruhov s týmito polomermi. Snehuliak je nakreslený v strede grafickej plochy na bledomodrom pozadí. Farbu bielych kruhov zvoľte tromi rôznymi odtieňmi bielej.
   - napr.

```
r1, r2, r3 = 20, 40, 60
# nakresli snehuliaka
```

7. Napíšte program, ktorý nakreslí rovnoramenný trojuholník so základňou `a` a výškou `v`. Hodnoty `a` a `v` program prečíta zo vstupu (`input()`).
   - napr.

```
a, v = 200, 170
# nakresli trojuholnik
```

8. Napíšte program, ktorý nakreslí takýto domček: skladá sa z červenej strechy (**rovnostranný** trojuholník so základňou `z`) a modrého štvorca (veľkosti `a` x `a`). Ľavý horný vrchol štvorca nech je na súradniciach `(x, y)` a strecha nech je vycentrovaná.

   - napr.

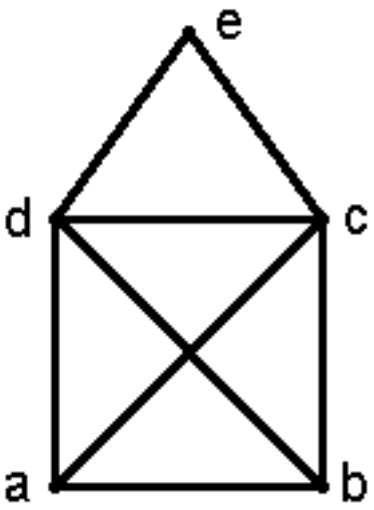
```
x, y = 50, 120
z, a = 100, 80
# nakresli domček
```

9. Napíšte program, ktorý nakreslí `n` žltých kruhov na náhodných pozíciách, pričom v strede každého sú postupne čísla 1, 2, ..., `n`. Polomer kruhov nech je `r`.

   - napr.

```
n = 30
r = 20
# nakresli n kruhov s číslami
```

10. Najprv zadefinujte päť vrcholov do premenných `a`, `b`, `c`, `d`, `e`, v ktorých budú súradnice bodov v rovine (ako **dvojice** čísel). Potom pomocou jediného volania `create_line()` nakreslite tento domček tak, aby sa po každej čiare prešlo len raz. Ako parametre `create_line()` použite týchto 5 premenných.



   - napr.

```
a = (..., ...)
b =
c =
d =
e =
# nakresli domcek jednym tahom
```

11. Podobne ako v úlohe (10) využijete päť vrcholov v premenných `a`, `b`, `c`, `d`, `e`. Zadefinujte ešte premennú `f`, ktorá bude obsahovať súradnice priesečníku uhlopriečok štvorca. Pomocou piatich volaní `create_polygon()` zafarbite každý trojuholník v domčeku inou farbou (len tie najmenšie trojuholníky v obrázku, nemyslíme napr. trojuholník `abc`). Zabezpečte, aby sa pri kreslení polygonov urobili ich čierne obrysy. V tejto úlohe nevolajte `create_line()`.

12. Body na kružnici so stredom `(x0, y0)` a polomerom `r` sa dajú vyjadriť vzorcom:

```
x = x0 + r * math.cos(math.radians(uhol))
y = y0 + r * math.sin(math.radians(uhol))
```

kde `uhol` je číslo od 0 do 360 stupňov. Ak budete takto vypočítané body postupne spájať úsečkami (napr. `canvas.create_line()`), dostanete kružnicu. Nakreslite týmto postupom kružnicu, pričom otestujte kreslenie pre rôznu hustotu bodov na kružnici (pre rôzne hodnoty zväčšovania uhla, t.j. `krok`, napr. s krokom 30, alebo 10 alebo 2, ...). Napr.

```
x0, y0 = 300, 200
r = 150
krok = 30
# nakresli kružnicu
```

13. V programe z predchádzajúcej úlohy (12) nebudete spájať susedné vrcholy, ktoré ležia na obvode kružnice, ale budeme spájať tieto vrcholy so stredom kružnice (zvoľte žlté hrubé pero). Na koniec nakreslite žltý kruh (`canvas.create_oval()`) s rovnakým stredom ako naša kružnica ale s menším polomerom. Takto dostanete slnko s lúčmi. Napíšte program:

```
x0, y0 = 200, 170
pocet_lucov = 10
dlzka_lucov = 150
velkost_slnka = 80
farba_pozadia = 'navy'
# nakreslí žlté slnko
```

14. Zvoľte si nejaký zaujímavý obrázok vo formáte `.png` alebo `.gif` (napr. The Python Logo) a nakreslite ho 10-krát na náhodných pozíciách.

KAPITOLA 4

4. Podmienky

4.1 Podmienený príkaz

Pri programovaní často riešime situácie, keď sa program má na základe nejakej podmienky rozhodnúť medzi viacerými možnosťami. Napr. program má vypísať, či zadaný počet bodov stačí na známku z predmetu. Preto si najprv vyžiada číslo - získaný počet bodov, **porovná** túto hodnotu s požadovanou hranicou, napr. 50 bodov a na základe toho vypíše, buď že je to dosť na známku, alebo nie je:

```
body = int(input('Zadaj získaný počet bodov: '))
if body >= 50:
    print(body, 'bodov je dostačujúci počet na známku')
else:
    print(body, 'bodov je málo na získanie známky')
```

Použili sme tu podmienený príkaz (príkaz vetvenia) `if`. Jeho zápis vyzerá takto:

```
if podmienka:     # ak podmienka plati, vykonaj 1. skupinu prikazov
    prikaz
    prikaz
    ...
else:             # ak podmienka neplati, vykonaj 2. skupinu prikazov
    prikaz
    prikaz
    ...
```

V našom príklade je v oboch skupinách príkazov len po jednom príkaze `print()`. Odsadenie skupiny príkazov (blok príkazov) má rovnaký význam ako vo for-cykle: budeme ich odsadzovať vždy presne o 4 medzery.

V pravidlách predmetu programovanie máme takéto kritériá na získanie známky:

- známka **A** aspoň 90 bodov
- známka **B** aspoň 80 bodov
- známka **C** aspoň 70 bodov

- známka **D** aspoň 60 bodov
- známka **E** aspoň 50 bodov
- známka **Fx** menej ako 50 bodov

Podmienka pre získanie známky **A**:

```
if body >= 90:
    print('za', body, 'bodov získavaš známku A')
else:
    ...
```

Ak je bodov menej ako 90, už to môže byť len horšia známka: dopíšeme testovanie aj známky **B**:

```
if body >= 90:
    print('za', body, 'bodov získavaš známku A')
else:
    if body >= 80:
        print('za', body, 'bodov získavaš známku B')
    else:
        ...
```

Všetky riadky v druhej skupine príkazov (za `else`) musia byť odsadené o 4 medzery, preto napr. `print()`, ktorý vypisuje správu o známke **B** je odsunutý o 8 medzier. Podobným spôsobom zapíšeme všetky zvyšné podmienky:

```
body = int(input('Zadaj získaný počet bodov: '))
if body >= 90:
    print('za', body, 'bodov získavaš známku A')
else:
    if body >= 80:
        print('za', body, 'bodov získavaš známku B')
    else:
        if body >= 70:
            print('za', body, 'bodov získavaš známku C')
        else:
            if body >= 60:
                print('za', body, 'bodov získavaš známku D')
            else:
                if body >= 50:
                    print('za', body, 'bodov získavaš známku E')
                else:
                    print('za', body, 'bodov si nevyhovel a máš známku Fx')
```

Takéto odsadzovanie príkazov je v Pythone veľmi dôležité a musíme byť pritom veľmi presní. Príkaz `if`, ktorý sa nachádza vo vnútri niektorej vetvy iného `if`, sa nazýva **vnorený príkaz if**.

V Pythone existuje konštrukcia, ktorá uľahčuje takúto vnorenú sériu `if`-ov:

```
if podmienka_1:       # ak podmienka_1 plati, vykonaj 1. skupinu prikazov
    prikaz
    ...
elif podmienka_2:     # ak podmienka_1 neplati, ale plati podmienka_2, ...
    prikaz
    ...
elif podmienka_3:     # ak ani podmienka_1 ani podmienka_2 neplatia, ale plati␣
↪podmienka_3, ...
    prikaz
    ...
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
else:                # ak ziadna z podmienok neplati, ...
    prikaz
    ...
```

Predchádzajúci program môžeme zapísať aj takto:

```
body = int(input('Zadaj získaný počet bodov: '))
if body >= 90:
    print('za', body, 'bodov získavaš známku A')
elif body >= 80:
    print('za', body, 'bodov získavaš známku B')
elif body >= 70:
    print('za', body, 'bodov získavaš známku C')
elif body >= 60:
    print('za', body, 'bodov získavaš známku D')
elif body >= 50:
    print('za', body, 'bodov získavaš známku E')
else:
    print('za', body, 'bodov si nevyhovel a máš známku Fx')
```

Ukážme ešte jedno riešenie tejto úlohy - jednotlivé podmienky zapíšeme ako intervaly:

```
body = int(input('Zadaj získaný počet bodov: '))
if body >= 90:
    print('za', body, 'bodov získavaš známku A')
if 80 <= body < 90:
    print('za', body, 'bodov získavaš známku B')
if 70 <= body < 80:
    print('za', body, 'bodov získavaš známku C')
if 60 <= body < 70:
    print('za', body, 'bodov získavaš známku D')
if 50 <= body < 60:
    print('za', body, 'bodov získavaš známku E')
if body < 50:
    print('za', body, 'bodov si nevyhovel a máš známku Fx')
```

V tomto riešení využívame to, že `else`-vetva v príkaze `if` môže chýbať a teda pri neplatnej podmienke, sa nevykoná nič:

```
if podmienka:        # ak podmienka plati, vykonaj skupinu prikazov
    prikaz
    prikaz
    ...              # ak podmienka neplati, nevykonaj nic
```

Mohli by sme to zapísať aj takto:

```
if podmienka:        # ak podmienka plati, vykonaj skupinu prikazov
    prikaz
    prikaz
    ...
else:
    pass             # ak podmienka neplati, nevykonaj nic
```

kde `pass` je tzv. **prázdny príkaz**, t.j. príkaz, ktorý nerobí nič. Môžeme ho použiť všade, kde chceme zapísať tzv. prázdny blok príkazov. Hoci je tento zápis správny, nezvykne sa takto používať.

Zrejme každý príkaz `if` po kontrole podmienky (a prípadnom výpise správy) pokračuje na ďalšom príkaze, ktorý

nasleduje za ním (a má rovnaké odsadenie ako `if`). Okrem toho vidíme, že teraz sú niektoré podmienky trochu zložitejšie, lebo testujeme, či sa hodnota nachádza v nejakom intervale. (podmienku `80 <= body < 90` sme mohli zapísať aj takto `90 > body >= 80`)

V Pythone môžeme zapisovať podmienky podobne, ako je to bežné v matematike:

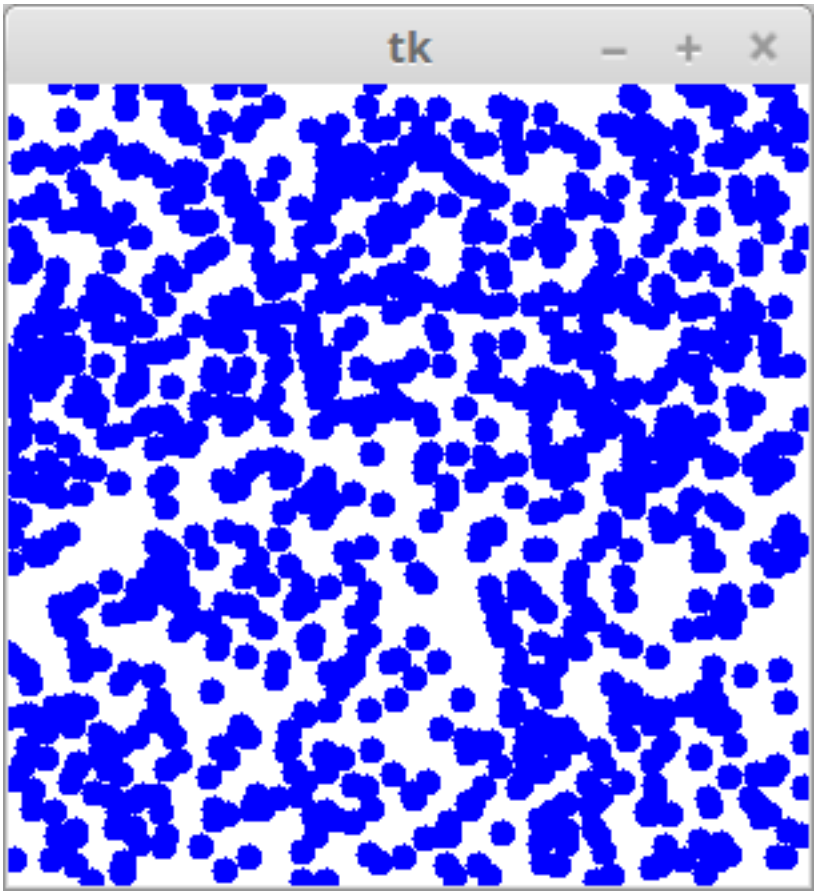
`body < 90`	je menšie ako
`body <= 50`	je menšie alebo rovné
`body == 50`	rovná sa
`body != 77`	nerovná sa
`body > 100`	je väčšie ako
`body >= 90`	je väčšie alebo rovné
`40 < body <= 50`	je väčšie ako ... a zároveň menšie alebo rovné ...
`a < b < c`	`a` je menšie ako `b` a zároveň je `b` menšie ako `c`

Ukážme použitie podmieneného príkazu aj v grafickom programe. Začneme s programom, ktorý na náhodné pozície nakreslí 1000 malých krúžkov:

```
import tkinter

canvas = tkinter.Canvas(bg='white', width=300, height=300)
canvas.pack()

for i in range(1000):
    x = random.randrange(300)
    y = random.randrange(300)
    farba = 'blue'
    canvas.create_oval(x-5, y-5, x+5, y+5, fill=farba, outline='')
```

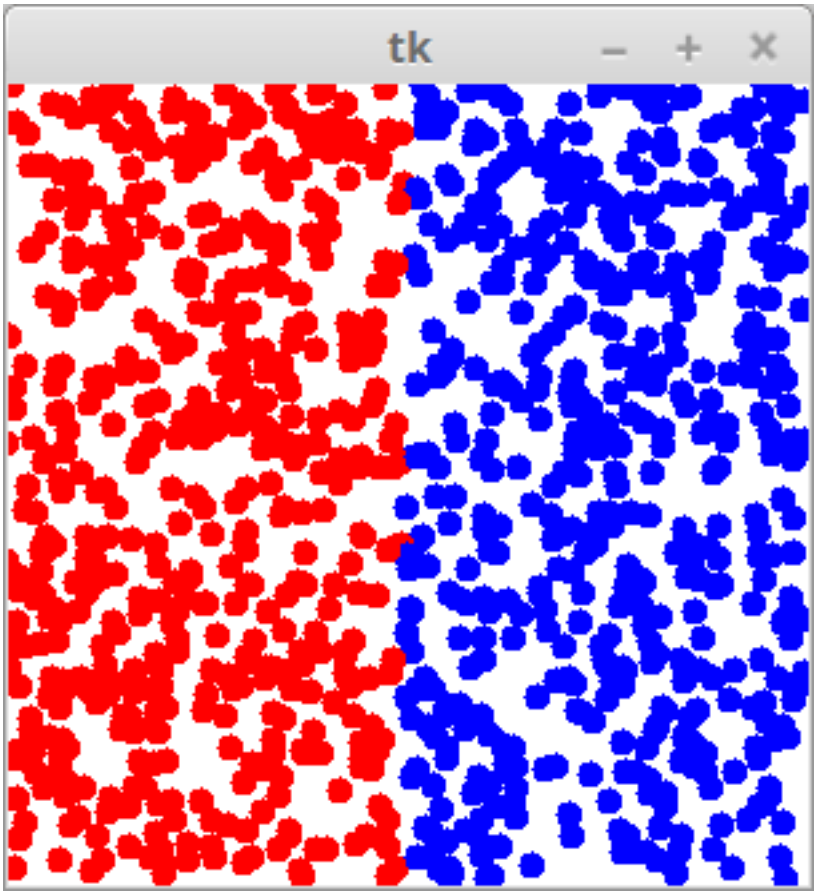


Teraz niektoré z týchto krúžkov zafarbíme tak, že tie z nich, ktoré sú v ľavej polovici plochy budú červené a zvyšné v pravej polovici (teda `else` vetva) budú modré:

```
import tkinter
import random

canvas = tkinter.Canvas(bg='white', width=300, height=300)
canvas.pack()

for i in range(1000):
    x = random.randrange(300)
    y = random.randrange(300)
    if x < 150:
        farba = 'red'
    else:
        farba = 'blue'
    canvas.create_oval(x-5, y-5, x+5, y+5, fill=farba, outline='')
```

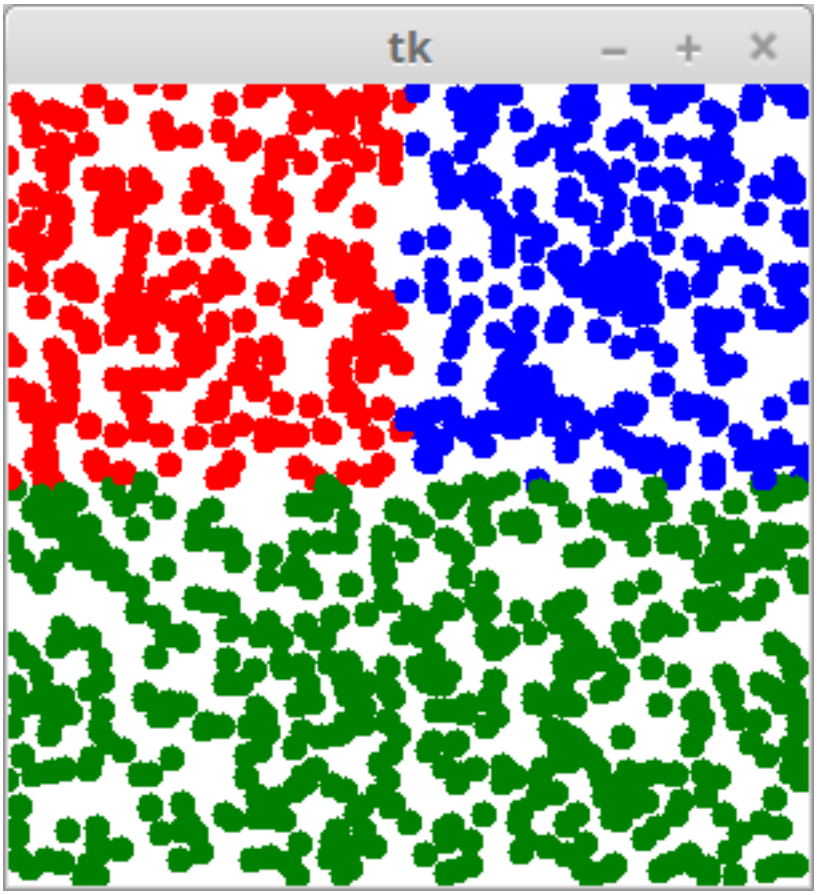


Skúsme pridať ešte jednu podmienku: všetky bodky v spodnej polovici (`y > 150`) budú zelené, takže rozdelenie na červené a modré bude len v hornej polovici. Jedno z možných riešení:

```
import tkinter
import random

canvas = tkinter.Canvas(bg='white', width=300, height=300)
canvas.pack()

for i in range(1000):
    x = random.randrange(300)
    y = random.randrange(300)
    if y < 150:
        if x < 150:
            farba = 'red'
        else:
            farba = 'blue'
    else:
        farba = 'green'
    canvas.create_oval(x-5, y-5, x+5, y+5, fill=farba, outline='')
```

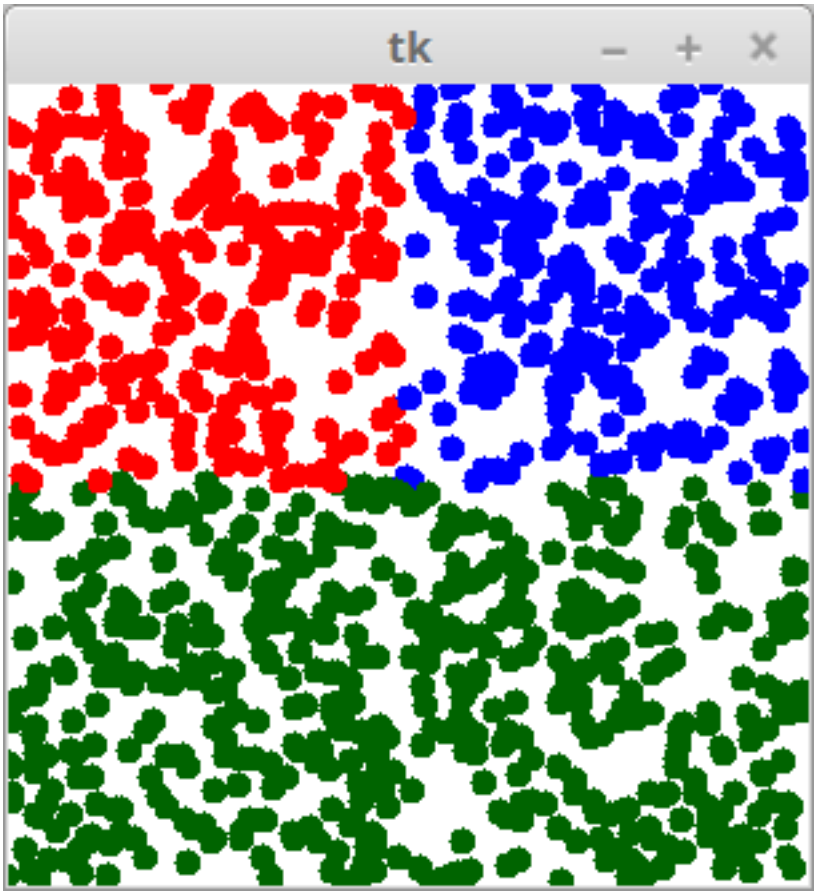


Podobne, ako sme to robili s intervalmi bodov pre rôzne známky, môžeme aj toto riešenie zapísať tak, že použijeme komplexnejšiu podmienku:

```
import tkinter
import random

canvas = tkinter.Canvas(bg='white', width=300, height=300)
canvas.pack()

for i in range(1000):
    x = random.randrange(300)
    y = random.randrange(300)
    if y < 150 and x < 150:
        farba = 'red'
    elif y < 150 and x >= 150:
        farba = 'blue'
    else:
        farba = 'darkgreen'
    canvas.create_oval(x-5, y-5, x+5, y+5, fill=farba, outline='')
```



Podmienky v Pythone môžu obsahovať logické operácie a tieto majú obvyklý význam z matematiky:

- podmienka1 `and` podmienka2 ... (a súčasne) znamená, že musia platiť obe podmienky
- podmienka1 `or` podmienka2 ... (alebo) znamená, že musí platiť aspoň jedna z podmienok
- `not` podmienka ... (neplatí) znamená, že daná podmienka neplatí

Otestovať rôzne kombinácie podmienok môžeme napr. takto:

```
>>> a = 10
>>> b = 7
>>> a < b
False
>>> a >= b + 3
True
>>> b < a < 2 * b
True
>>> a != 7 and b == a - 3
True
>>> a == 7 or b == 10
False
>>> not a == b          # to iste ako  a != b
True
>>> 1 == '1'
False
>>> 1 < '2'             # nemozeme takto porovnavat cisla a znaky
...
TypeError: unorderable types: int() < str()
```

Všimnite si, že podmienky ktoré platia, majú hodnotu `True` a ktoré neplatia, majú `False` - sú to dve špeciálne hodnoty, ktoré Python používa ako výsledky porovnávania - tzv. logických výrazov. Sú **logického typu**, tzv. `bool`. Môžeme to skontrolovať:

```
>>> type(1 + 2)
<class 'int'>
>>> type(1 / 2)
<class 'float'>
>>> type('12')
<class 'str'>
>>> type(1 < 2)
<class 'bool'>
```

4.1.1 Logické operácie

Pozrime sa podrobnejšie na logické operácie `and`, `or` a `not`. Tieto operácie samozrejme fungujú pre logické hodnoty `True` a `False`.

Tabuľka 1: Logický súčin **a súčasne**

A	B	A and B	
`False`	`False`	`False`	
`True`	`False`	`False`	
`False`	`True`	`False`	
`True`	`True`	`True`	

Tabuľka 2: Logický súčet **alebo**

A	B	A or B	
`False`	`False`	`False`	
`True`	`False`	`True`	
`False`	`True`	`True`	
`True`	`True`	`True`	

Tabuľka 3: Negácia **neplatí**

A	not A	
`False`	`True`	
`True`	`False`	

Logické operácie fungujú nielen pre logický typ, ale aj pre skoro všetky ďalšie typy. V tomto prípade Python pre **každý typ** definuje prípady, ktoré on (Python) chápe ako `False` a zrejme všetky ostatné hodnoty tohto typu chápe ako `True`. Ukážme prípady pre doteraz známe typy, ktoré označujú logickú hodnotu `False`:

Tabuľka 4: **False** z iných typov

typ	`False`	`True`	
`int`	`x == 0`	`x != 0`	
`float`	`x == 0.0`	`x != 0.0`	
`str`	`x == ''`	`x != ''`	

Tejto tabuľke budeme rozumieť takto: keď bude Python niekde očakávať logickú hodnotu (napr. v príkaze `if`) a bude tam namiesto toho celé číslo, tak, ak má hodnotu **0**, pochopí to ako `False` a ak má hodnotu rôznu od **0**, bude

to pre neho znamenať `True`. Podobne aj s reťazcami: ak ako podmienku `if` uvedieme znakový reťazec, tento bude reprezentovať `False`, ak je prázdny a `True`, ak je neprázdny.

Napr.

```
pocet = int(input('zadaj: '))
if pocet:
    print('pocet je rôzny od 0')
else:
    print('pocet je 0')
meno = input('zadaj: ')
if meno:
    print('meno nie je prázdny ret'azec')
else:
    print('meno je prázdny ret'azec')
```

Logické operácie `and`, `or` a `not` majú v skutočnosti trochu rozšírenú interpretáciu:

operácia: prvý `and` druhý

- ak **prvý** nie je `False`, tak
  - výsledkom je **druhý**
- inak (teda **prvý** je `False`)
  - výsledkom je **prvý**

Môžeme upraviť aj tabuľku pravdivostných hodnôt:

Tabuľka 5: Logický súčin **a súčasne**

A	B	A and B	
`False`	*hocičo*	**A**	
`True`	*hocičo*	**B**	

operácia: prvý `or` druhý

- ak **prvý** nie je `False`, tak
  - výsledkom je **prvý**
- inak (teda **prvý** je `False`)
  - výsledkom je **druhý**

Tabuľka:

Tabuľka 6: Logický súčet **alebo**

A	B	A or B	
`False`	*hocičo*	**B**	
`True`	*hocičo*	**A**	

operácia: `not` prvý

- ak **prvý** nie je `False`, tak
  - výsledkom je `False`
- inak
  - výsledkom je `True`

Napr.

```
>>> 1 + 2 and 3 + 4        # kedze 1+2 nie je False, vysledkom je 3+4
7
>>> 'ahoj' or 'Python'     # kedze 'ahoj' nie je False, vysledkom je 'ahoj'
'ahoj'
>>> '' or 'Python'         # kedze '' je False, vysledkom je 'Python'
'Python'
>>> 3 < 4 and 'kuk'        # kedze 3<4 nie je False, vysledkom je 'kuk'
'kuk'
```

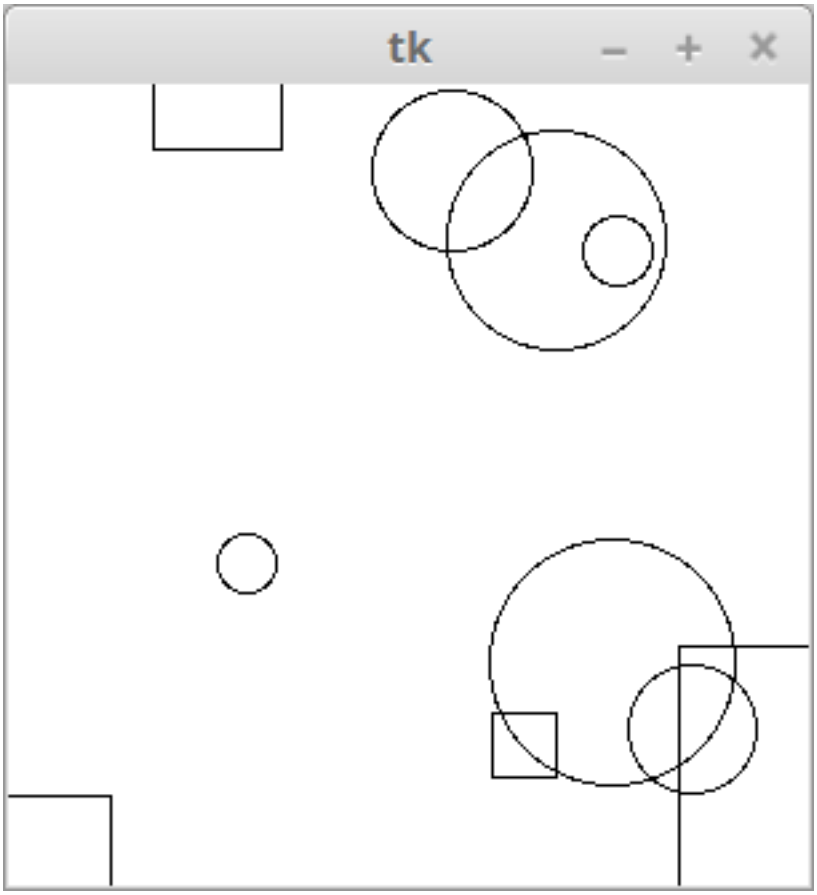
Podmienený príkaz sa často používa pri náhodnom rozhodovaní. Napr. hádžeme mincou (náhodné hodnoty 0 a 1) a ak padne 1, kreslíme náhodnú kružnicu, inak nakreslíme náhodný štvorec. Toto opakujeme 10-krát:

```
import tkinter
import random

canvas = tkinter.Canvas(bg='white', width=300, height=300)
canvas.pack()

for i in range(10):
    x = random.randrange(300)
    y = random.randrange(300)
    a = random.randrange(5, 50)

    if random.randrange(2):                # t.j. random.randrange(2) != 0
        canvas.create_oval(x-a, y-a, x+a, y+a)
    else:
        canvas.create_rectangle(x-a, y-a, x+a, y+a)
```



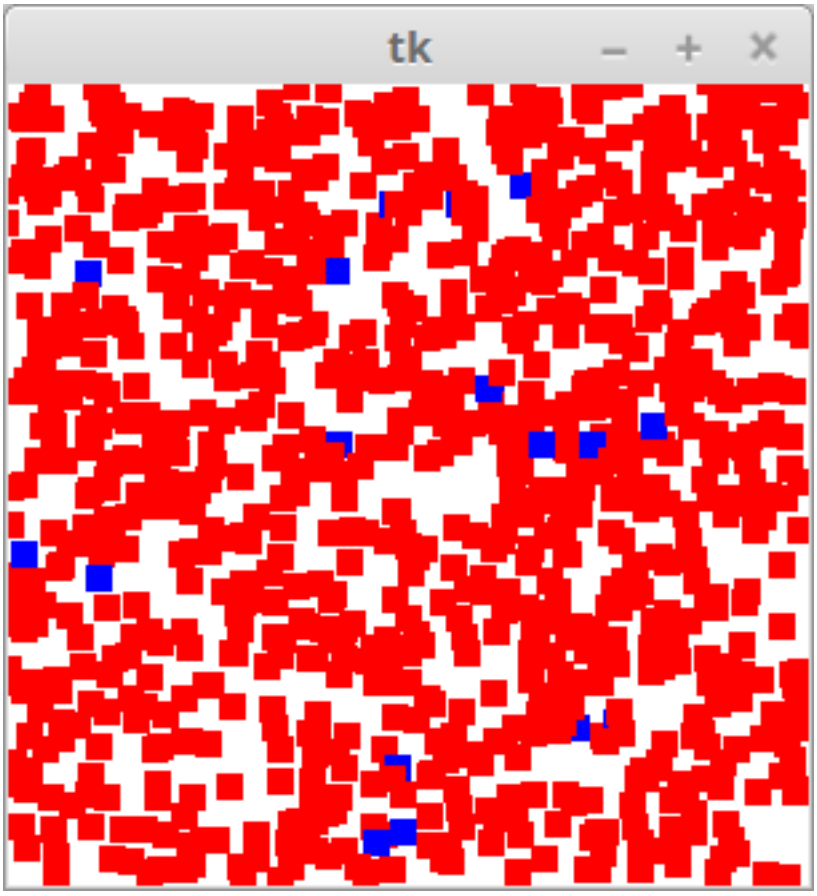
Približne rovnaké výsledky by sme dostali, ak by sme hádzali kockou so 6 možnosťami (`random.randrange(1, 7)`) a pre čísla 1, 2, 3 by sme kreslili kružnicu inak štvorec.

Túto ideu môžeme využiť aj pre takúto úlohu: vygenerujte 1000 farebných štvorčekov - modré a červené, pričom ich pomer je **1:50**, t.j. na 50 červených štvorčekov pripadne približne 1 modrý:

```
import tkinter
import random

canvas = tkinter.Canvas(bg='white', width=300, height=300)
canvas.pack()

for i in range(1000):
    x = random.randrange(300)
    y = random.randrange(300)
    if random.randrange(50):              # t.j. random.randrange(50) != 0
        farba = 'red'
    else:
        farba = 'blue'
    canvas.create_rectangle(x-5, y-5, x+5, y+5, fill=farba, outline='')
```



Ďalší príklad zisťuje, akých deliteľov má zadané číslo:

```
cislo = int(input('Zadaj číslo: '))
pocet = 0
print('delitele:', end=' ')
for delitel in range(1, cislo+1):
    if cislo % delitel == 0:        # mohli by sme zapisat aj  if not cislo % delitel:
        pocet += 1
        print(delitel, end=' ')
print()
print('počet delitel'ov:', pocet)
```

Výstup môže byť napríklad takýto:

```
Zadaj číslo: 100
delitele: 1 2 4 5 10 20 25 50 100
počet delitel'ov: 9
```

Malou modifikáciou tejto úlohy vieme urobiť ďalšie dva programy. Prvý zisťuje, či je zadané číslo prvočíslo:

```
cislo = int(input('Zadaj číslo: '))
pocet = 0
for delitel in range(1, cislo+1):
    if cislo % delitel == 0:
        pocet += 1
if pocet == 2:
    print(cislo, 'je prvočíslo')
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
else:
    print(cislo, 'nie je prvočíslo')
```

Po spustení napr.

```
Zadaj číslo: 101
101 je prvočíslo
```

Ďalší program zisťuje, či je nejaké číslo dokonalé, t.j. súčet všetkých deliteľov menších ako samotné číslo sa rovná samotnému číslu. Na základe tohto nájde (postupne preverí) všetky dokonalé čísla do 10000:

```
print('dokonalé čísla do 10000 sú', end=' ')
for cislo in range(1,10001):
    sucet = 0
    for delitel in range(1, cislo):
        if cislo % delitel == 0:
            sucet += delitel
    if sucet == cislo:
        print(cislo, end=', ')
print()
print('=== viac ich už nie je ===')
```

Program vypíše:

```
dokonalé čísla do 10000 sú 6, 28, 496, 8128,
=== viac ich už nie je ===
```

Vráťme sa k programu, ktorý zisťuje, či je nejaké číslo prvočíslo:

```
cislo = int(input('Zadaj číslo: '))
pocet = 0
for delitel in range(1, cislo+1):
    if cislo % delitel == 0:
        pocet += 1
if pocet == 2:
    print(cislo, 'je prvočíslo')
```

Tento for-cyklus prejde vždy `cislo`-krát, pričom nám by stačilo zistiť, či existuje aspoň jeden deliteľ, ktorý je väčší ako **1** a menší ako **cislo**. Ak taký nájdeme (t.j. platí `cislo % delitel == 0`), cyklus už ďalej nemusí preverovať zvyšné delitele. Predstavme si, že chceme zisťovať, či číslo **1000000** je prvočíslo. Tento náš program už pri treťom prechode cyklu (keď `delitel` má hodnotu 3) „vie“, že `pocet` je viac ako 2 a teda to určite nebude prvočíslo. Úplne zbytočne 999997-krát zisťuje, či nejaký deliteľ delí alebo nedelí naše `cislo`. V tomto prípade by sa vykonávanie cyklu mohlo prerušiť a dostali by sme oveľa rýchlejšie rovnako správny výsledok. Na prerušovanie for-cyklu využijeme špeciálny príkaz **break**, ktorý ale môžeme použiť len v tele cyklu.

Prerušenie cyklu pomocou break

Príkaz `break` v tele cyklu spôsobí ukončenie cyklu, t.j. vykonávanie príkazov cyklu sa v tomto momente preruší a pokračuje sa až za prvým príkazom za cyklom. Premenná cyklu bude mať potom hodnotu naposledy vykonávaného prechodu. Príkaz `break` sa musí použiť v nejakom vnorenom podmienenom príkaze `if`. Napr.

```
for premenna in ...:
    príkazy1
    if podmienka:
        break
    príkazy2
príkazy_za_cyklom
```

V každom prechode cyklu sa najprv vykonajú `príkazy1`. Potom odkontroluje `podmienka`: ak je pravdivá, vykonávanie cyklu končí a pokračuje sa na `príkazy_za_cyklom`. Ak je podmienka nepravdivá (neplatí), cyklus pokračuje vykonávaním `príkazy2` a prípadnými ďalšími prechodmi cyklu.

Vylepšime cyklus na zisťovanie prvočísel s prerušením pomocou `break`:

```
cislo = int(input('Zadaj číslo: '))
pocet = 0
for delitel in range(1, cislo+1):
    if cislo % delitel == 0:
        pocet += 1
        if pocet > 2:
            break
if pocet == 2:
    print(cislo, 'je prvočíslo')
else:
    print(cislo, 'nie je prvočíslo')
```

Otestujte zisťovaním, či je 1000000000 prvočíslo. Pomocou predchádzajúcej verzie programu by sme sa výsledku nedočkali, ale teraz sa dozvieme veľmi rýchlo, že:

```
Zadaj číslo: 1000000000
1000000000 nie je prvočíslo
```

Všimnite si, že v tomto prípade vôbec nepotrebujeme premennú `pocet`, ktorá počítala počet deliteľov. Podľa hodnoty premennej `delitel` po skončení cyklu, vieme presne povedať, či to bolo prvočíslo alebo nie. Upravme:

```
cislo = int(input('Zadaj číslo: '))
for delitel in range(2, cislo+1):
    if cislo % delitel == 0:
        break
if delitel == cislo:
    print(cislo, 'je prvočíslo')
else:
    print(cislo, 'nie je prvočíslo')
```

4.2 Podmienený cyklus

V Pythone existuje konštrukcia cyklu, ktorá opakuje vykonávanie postupnosti príkazov v závislosti od nejakej podmienky:

```
while podmienka:              # opakuj prikazy, kym plati podmienka
    prikaz
    prikaz
    ...
```

Vidíme podobnosť s podmieneným príkazom `if` - vetvením. Tento nový príkaz postupne:

- zistí hodnotu podmienky, ktorá je zapísaná za slovom `while`
- ak má táto podmienka hodnotu `False`, blok príkazov, ktorý je telom cyklu, sa preskočí a pokračuje sa na nasledovnom príkaze za celým while-cyklom (podobne ako v príkaze `if`), hovoríme, že sa ukončilo vykonávanie cyklu
- ak má podmienka hodnotu `True`, vykonajú sa všetky príkazy v tele cyklu (odsunutom bloku príkazov)

- a znovu sa testuje podmienka za slovom `while`, t.j. celé sa to opakuje

Najprv zapíšeme pomocou tohto cyklu, to čo už vieme pomocou for-cyklu:

```
for i in range(1, 21):
    print(i, i * i)
```

Vypíše tabuľku druhých mocnín čísel od 1 do 20. Prepis na cyklus `while` znamená, že zostavíme podmienku, ktorá bude testovať, napr. premennú `i`: tá nesmie byť väčšia ako 20. Samozrejme, že už pred prvou kontrolou premennej `i` v podmienke cyklu `while`, musí mať nejakú hodnotu:

```
i = 1
while i < 21:
    print(i, i * i)
    i += 1
```

V cykle sa vykoná `print()` a zvýši sa hodnota premennej `i` o jedna.

`while`-cykly sa ale častejšie používajú vtedy, keď zápis pomocou `for`-cyklu je príliš komplikovaný, alebo sa ani urobiť nedá.

Ukážeme to na programe, ktorý bude vedľa seba kresliť stále sa zväčšujúce štvorce postupne so stranami 10, 20, 30, ... Pritom bude dávať pozor, aby naposledy nakreslený štvorec „nevypadol“ z plochy - teda chceme skončiť skôr, ako by sme nakreslili štvorec, ktorý sa už celý nezmestí do grafickej plochy. Štvorce so stranou `a` budeme kresliť takto:

```
canvas.create_rectangle(x, 200, x+a, 200-a)
```

vďaka čomu, všetky ležia na jednej priamke (`y=200`). Keď teraz budeme posúvať x-ovú súradnicu vždy o veľkosť nakresleného štvorca, ďalší bude ležať tesne vedľa neho.

Program pomocou while-cyklu zapíšeme takto:

```
import tkinter

sirka = int(input('šírka plochy: '))

canvas = tkinter.Canvas(bg='white', width=sirka)
canvas.pack()

x = 5
a = 10
while x + a < sirka:
    canvas.create_rectangle(x, 200, x+a, 200-a)
    x = x + a
    a = a + 10
# príkazy za cyklom
```

Program pracuje korektne pre rôzne šírky grafickej plochy. Ak zväčšovanie strany štvorca `a = a + 10` nahradíme `a = 2 * a`, program bude pracovať aj s takto zväčšovanými štvorcami (strany budú postupne 10, 20, 40, 80, ...).

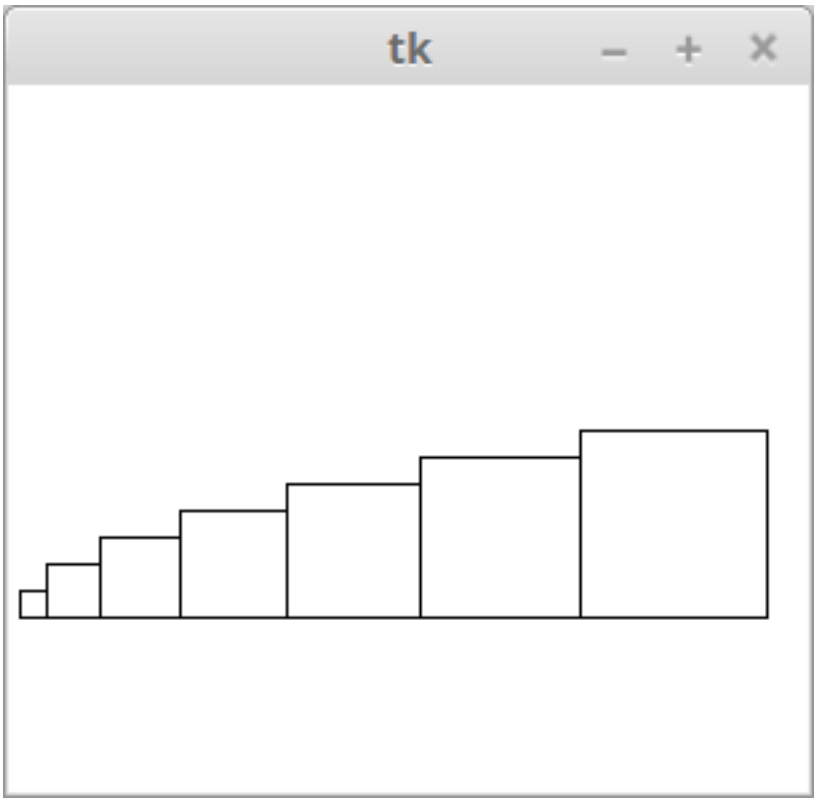
Zhrňme, ako funguje tento typ cyklu:

1. vyhodnotí sa podmienka `x + a < sirka`, t.j. pravý okraj štvorca, ktorý práve chceme nakresliť, sa ešte celý zmestí do grafickej plochy
2. ak je podmienka splnená (pravdivá), postupne sa vykonajú všetky príkazy, t.j. nakreslí sa ďalší štvorec so stranou `a` a potom sa posunie ľavý okraj budúceho štvorca o veľkosť práve nakresleného štvorca `a` a tiež sa ešte zmení veľkosť budúceho štvorca `a` o 10
3. po vykonaní tela cyklu sa pokračuje v 1. kroku, t.j. opäť sa vyhodnotí podmienka

4. ak podmienka nie je splnená (nepravda), cyklus končí a ďalej sa pokračuje v príkazoch za cyklom

Uvedomte si, že podmienka nehovorí, kedy má cyklus skončiť, ale naopak - kým podmienka platí, vykonávajú sa všetky príkazy v tele cyklu.

Pre šírku grafickej plochy 300 dostávame takýto obrázok:



Konkrétne tento program s while-cyklom vieme jednoducho prepísať na for-cyklus:

```
import tkinter

sirka = int(input('šírka plochy: '))

canvas = tkinter.Canvas(bg='white', width=sirka)
canvas.pack()

x = 5
for a in range(10, sirka, 10):
    if x + a >= sirka:
        break
    canvas.create_rectangle(x, 200, x+a, 200-a)
    x = x + a
```

Vyššie sme zostavili program, ktorý zisťoval, či je zadané číslo prvočíslo. Použili sme for-cyklus, v ktorom sme zadané číslo postupne delili všetkými číslami, ktoré sú menšie ako samotné číslo. Zistili sme, že na zisťovanie prvočísla nepotrebujeme skutočný počet deliteľov, ale malo by nám stačiť zistenie, či existuje aspoň jeden deliteľ. Keď sa vyskytne prvý deliteľ (t.j. platí `cislo % delitel != 0`), cyklus môžeme ukončiť a vyhlásiť, že číslo nie je prvočíslo. Ak ani jedno číslo nie je deliteľom nášho čísla, hodnota premennej `delitel` dosiahne `cislo` a to je situácia, keď cyklus tiež skončí (t.j. keď `delitel == cislo`, našli sme prvočíslo). Zapíšeme to while-cyklom:

```
cislo = int(input('Zadaj číslo: '))
delitel = 2
while delitel < cislo and cislo % delitel != 0:
    delitel = delitel + 1

if delitel == cislo:
    print(cislo, 'je prvočíslo')
else:
    print(cislo, 'nie je prvočíslo')
```

Do podmienky while-cyklu sme pridali novú časť. Operátor `and` tu označuje to, že aby sa cyklus opakoval, musia byť splnené obe časti. Uvedomte si, že **cyklus skončí vtedy**, keď prestane platiť zadaná podmienka, t.j. (a ďalej to matematicky upravíme):

- `not (delitel < cislo and cislo % delitel != 0)`
- `not delitel < cislo or not cislo % delitel != 0`
- `delitel >= cislo or cislo % delitel == 0`

while-cyklus teda skončí vtedy, keď `delitel >= cislo`, alebo `cislo % delitel == 0` (teda, že deliteľ by bol už zbytočne veľký alebo našli sme hodnotu, ktorá delí naše číslo).

4.2.1 Zisťovanie druhej odmocniny

Ukážeme, ako zistíme druhú odmocninu čísla aj bez volania funkcie `math.sqrt(x)`, resp. umocňovaním na jednu polovicu `x**0.5`.

Prvé riešenie:

```
cislo = float(input('zadaj cislo:'))

x = 1
while x ** 2 < cislo:
    x = x + 1

print('odmocnina', cislo, 'je', x)
```

Takto nájdené riešenie je veľmi nepresné, lebo `x` zvyšujeme o 1, takže, napr. odmocninu z 26 vypočíta ako 6. Skúsme zjemniť krok, o ktorý sa mení hľadané `x`:

```
cislo = float(input('zadaj cislo:'))

x = 1
while x ** 2 < cislo:
    x = x + 0.001

print('odmocnina', cislo, 'je', x)
```

Teraz dáva program lepšie výsledky, ale pre väčšiu zadanú hodnotu mu to citeľne dlhšie trvá - skúste zadať napr. 10000000. Keďže mu vyšiel výsledok približne 3162.278 a dopracoval sa k nemu postupným pripočítavaním čísla 0.001 k štartovému 1, musel urobiť vyše 3 miliónov pripočítaní a tiež toľkokrát testov vo while-cykle (podmienky `x ** 2 < cislo`). Kvôli tomuto je takýto algoritmus nepoužiteľne pomalý.

Využijeme inú ideu:

- zvolíme si interval, v ktorom sa určite bude nachádzať hľadaný výsledok (hľadaná odmocnina), napr. nech je to interval `<1, cislo>` (pre čísla väčšie ako 1 je aj odmocnina väčšia ako 1 a určite je menšia ako samotne

`cislo`)

- ako `x` (prvý odhad nášej hľadanej odmocniny) zvolíme stred tohto intervalu
- zistíme, či je druhá mocnina tohto `x` väčšia ako zadané `cislo` alebo menšia
- ak je väčšia, tak upravíme predpokladaný interval, tak že jeho hornú hranicu zmeníme na `x`
- ak je ale menšia, upravíme dolnú hranicu intervalu na `x`
- tým sa nám interval zmenšil na polovicu
- toto celé opakujeme, kým už nie je nájdené `x` dostatočne blízko k hľadanému výsledku, t.j. či sa nelíši od výsledku menej ako zvolený rozdiel (epsilón)

Zapíšme to:

```
cislo = float(input('zadaj cislo:'))

od = 1
do = cislo

x = (od + do)/2

pocet = 0
while abs(x ** 2 - cislo) > 0.001:
    if x ** 2 > cislo:
        do = x
    else:
        od = x
    x = (od + do) / 2
    pocet += 1

print('druhá odmocnina', cislo, 'je', x)
print('počet prechodov while-cyklom bol', pocet)
```

Ak spustíme program pre 10000000 dostávame:

```
zadaj cislo:10000000
druhá odmocnina 10000000.0 je 3162.2776600480274
počet prechodov while-cyklom bol 44
```

čo je výrazné zlepšenie oproti predchádzajúcemu riešeniu, keď prechodov while-cyklom (hoci jednoduchších) bolo vyše 3 miliónov.

4.2.2 Nekonečný cyklus

Cyklus s podmienkou, ktorá má stále hodnotu `True`, bude nekonečný. Napr.

```
i = 0
while i < 10:
    i -= 1
```

Nikdy neskončí, lebo premenná `i` bude stále menšia ako 10. Takéto výpočty môžeme prerušiť stlačením klávesov **Ctrl/C**.

Aj nasledovný cyklus je úmyselne nekonečný:

```
while 1:
    pass
```

Pripomíname, že príkaz `pass` je **prázdny príkaz**, ktorý nerobí nič. V tomto príklade `pass` označuje prázdne telo cyklu a teda tento cyklus bude nikdy nekončiaci.

Už vieme, že príkaz `break` môžeme použiť v tele for-cyklu a vtedy sa zvyšok cyklu nevykoná. Nasledovný príklad ilustruje použitie `break` aj vo while-cykle:

```
sucet = 0
while True:
    retazec = input('zadaj číslo: ')
    if retazec == '':
        break
    sucet += int(retazec)
print('súčet prečítaných čísel =', sucet)
```

V tomto príklade sa čítajú čísla zo vstupu, kým nezadáme prázdny reťazec: vtedy cyklus končí a program vypíše súčet prečítaných čísel. Napr.

```
zadaj číslo: 5
zadaj číslo: 17
zadaj číslo: 2
zadaj číslo:
súčet prečítaných čísel = 24
```

4.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/
- v riešeniach úloh používajte len konštrukcie a funkcie, ktoré sme sa učili na doterajších prednáškach

1. Zistite, akú hodnotu `True` alebo `False` (alebo inú) majú tieto výrazy:
   - najprv to skúste bez počítača, potom to skontrolujte v Pythone

   ```
   x, y = 7, 'ab'
   8 < x <= 7
   x <= 3 + x // 2
   y != 2 * x or 2 * y == 'abab'
   x < x + 1 < 2 * x

   x // 8 or x * y
   x or y
   x and y
   not y
   not x % 2
   ```

2. Napíšte program, ktorý najprv prečíta 3 desatinné čísla `a`, `b`, `c` a potom vypíše, **koľko reálnych ale rôznych koreňov** má kvadratická rovnica so zadanými koeficientami (zrejme výsledkom bude 0, alebo 1, alebo 2)
   - napr.

   ```
   zadaj a: 1
   zadaj b: 2
   zadaj c: 1
   ```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
kvadraticka rovnica ma jeden koren

zadaj a: 1
zadaj b: 0
zadaj c: 1
kvadraticka rovnica nema realne korene

zadaj a: 1
zadaj b: 2
zadaj c: -3
kvadraticka rovnica ma dva rozne korene
```

3. Máme daný tento program.

   - ručne bez počítača zistite, čo vypočíta pre vstupnú hodnotu **11**:

```
cislo1 = 0
cislo2 = int(input('? '))
while cislo2 > 0:
    if cislo2 % 2 == 0:
        cislo2 //= 2
    else:
        cislo2 -= 1
    cislo1 += 1
print(cislo1)
```

   - Vedeli by ste matematikovi, ktorý nevie programovať ale pozná dvojkovú sústavu, vysvetliť, čo tento program vypočíta?

4. Napíšte program, ktorý najprv prečíta celé číslo a potom, kým je toto číslo väčšie ako 0, opakuje tento blok príkazov: vypíše jeho poslednú cifru a potom ho ešte vydelí 10. Takto by ste mali dosiahnuť postupný výpis všetkých cifier vstupného čísla ale od konca (od poslednej cifry). Cifry vypisujte do jedného riadku.

   - napr.

```
zadaj cislo: 50273
3 7 2 0 5
```

5. Prerobte riešenie predchádzajúceho príkladu (4) tak, že cifry sa nebudú vypisovať, ale sčitovať. Takto by ste mali dostať **ciferný súčet** daného čísla

   - napr.

```
zadaj cislo: 50273
ciferny sucet je 17
```

6. Využite ideu riešenia predchádzajúceho príkladu a vyriešte: program zistí, koľkokrát sa v zadanom čísle objaví nejaká konkrétna cifra

   - napr.

```
zadaj cislo: 12345678912345678123456 7
zadaj cifru: 8
cifra 8 sa vyskytla 2 krat

zadaj cislo: 12345678912345678123456 7
zadaj cifru: 0
cifra 0 sa vyskytla 0 krat
```

- vyskúšajte to aj pre nejaké veľké číslo, napr. `2 ** 1000`

7. Napíšte program, ktorý najprv prečíta celé číslo a vypíše jeho rozklad na prvočinitele (ako súčin prvočísel).
   - snažte sa vyrobiť výpis v takomto tvare:

```
zadaj cislo: 24
24 = 2 * 2 * 2 * 3

zadaj cislo: 31
31 = 31

zadaj cislo: 65536
65536 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2
```

   - rozklad na prvočinitele môžete naprogramovať pomocou while-cyklu takto: nastavte `delitel` na 2, potom kým je dané číslo väčšie ako 1: ak je deliteľné hodnotou `delitel`, treba tento `delitel` vypísať a dané číslo ním vydeliť, ak dané číslo nie je deliteľné hodnotou `delitel`, premennú `delitel` treba zvýšiť o 1

8. Napíšte program, ktorý nájde také najväčšie `n`, pre ktoré `n!` (faktoriál) je menší ako 1000000.
   - použite while-cyklus

9. **Fibonacciho postupnosť** sa skladá z čísel 0,1,1,2,3,5,8,13,21, ..., teda každé ďalšie v postupnosti je súčtom dvoch predchádzajúcich. Napíšte program, ktorý zistí najväčšie fibonacciho číslo, ktoré nie je väčšie ako 1000000.
   - použite while-cyklus

10. Máme daný tento program.
    - najprv bez počítača odhadnite čo urobí

```
for i in range(10):
    while random.randrange(4):
        print(end='x')
    print()
```

    - uvažujte nad tým, čo nové ste sa na tomto príklade naučili

11. Grafická plocha má veľkosť `vel` x `vel` (`vel` je konštanta v programe). Generujeme do nej `n` náhodných bodiek (malé krúžky s polomerom 3), pričom súradnice `x` a `y` sú z intervalu <0, `vel`). Ak vzdialenosť vygenerovanej bodky od bodu (0, 0) je menšia ako `vel`, bodka bude červená, inak bude modrá. Počet bodiek `n` je tiež konštanta v programe. Počas generovania bodiek spočítajte, koľko z nich je červených. Na záver program vypíše pomer počtu červených bodiek ku všetkým vygenerovaným krát 4.
    - nastavte napr.

```
vel = 500
n = 1000
# program počíta počet vygenerovaných červených bodiek
print('pomer =', ...)
```

    - Vedeli by ste zdôvodniť, prečo sa tento pomer pre veľké `n` blíži k číslu `pi`?

12. Grafická plocha má veľkosť 300x200 a generujeme do nej náhodné bodky (malé krúžky s polomerom 3), pričom súradnica `x` je z intervalu <0,300) a `y` <0, 200). Ak vzdialenosť vygenerovanej bodky od bodu (100, 100) je menšia ako 80, bodka bude červená, inak ak vzdialenosť od bodu (180, 100) je menšia ako 90, bodka bude modrá, inak bodka bude čierna. Vygenerujte `n` takýchto bodiek.
    - `n` je konštanta programu, napr.

```
n = 1000
```

13. Nastavte grafickú plochu na veľkosť `sirka, vyska = 250, 250`. Nakreslite do nej šachovnicu 8x8 štvorčekov každý bude veľkosti 30x30, pričom ľavý horný štvorček má ľavý horný roh na súradniciach (5, 5). Pri kreslení striedavo zafarbujte políčka šachovnice dvomi farbami, napr. červenou a modrou (dajte pozor na rozostavenie farieb ako na šachovnici).
    - riešte dvomi vnorenými for-cyklami
14. Predchádzajúci príklad (13) riešte tak, že sa nenakreslí šachovnica veľkosti 8x8, ale šachovnica, v ktorej je len toľko štvorčekov v riadku, resp. v stĺpci, aby sa každý z nich zmestil do grafickej plochy. Napr. pre `sirka, vyska = 200, 150` bude mať šachovnica v každom riadku len 6 štvorčekov a riadky budú len 4.
    - namiesto dvoch vnorených for-cyklov použite while-cykly

4.4 1. Domáce zadanie

L.I.S.T.

- riešenie odovzdávajte na úlohový server https://list.fmph.uniba.sk/
- v riešení úlohy používajte len konštrukcie, ktoré sme sa učili na doterajších prednáškach

Napíšte program, ktorý prečíta niekoľko vstupných údajov a na ich základe vypíše niekoľko riadkov textu.

Program postupne prečíta tieto 3 vstupné hodnoty:

- **n** (`int`), nezáporné celé číslo
- **znak1** (`str`), jednoznakový reťazec
- **znak2** (`str`), jednoznakový reťazec

Pre ich prečítanie použite príkaz `input()` bez parametra.

Program vypíše takýto text:

- zo znakov **znak1** sa nakreslí štvorec veľkosti **n** riadkov po **n** stĺpcov
- vnútro štvorca obsahuje vykreslené aj obe uhlopriečky so znakov **znak1**
- na zvyšné políčka štvorca použite znak **znak2**

Napr. pre **n** = `9`, **znak1** = `'*'`, **znak2** = `'.'`, vyrobí takýto výstup:

```
*********
**.....**
*.*...*.*
*..*.*..*
*...*...*
*..*.*..*
*.*...*.*
**.....**
*********
```

Váš odovzdaný program s menom `riesenie1.py` musí začínať tromi riadkami komentárov:

```
# 1. zadanie: štvorec s uhlopriečkami
# autor: Janko Hraško
# datum: 10.10.2017
```

V programe používajte len konštrukcie jazyka Python, ktoré sme sa učili na prvých 4 prednáškach.

Projekt `riesenie1.py` odovzdávajte na úlohový server https://list.fmph.uniba.sk/ najneskôr do 23:00 **17. októbra**, kde ho môžete nechať otestovať. Odovzdať projekt aj ho testovať môžete ľubovoľný počet krát. Môžete zaň získať **4 body**.

KAPITOLA 5

5. Podprogramy

5.1 Funkcie

Doteraz sme pracovali so štandardnými funkciami, napr.

- vstup a výstup `input()` a `print()`
- aritmetické funkcie `abs()` a `round()`
- generovanie postupnosti čísel pre for-cyklus `range()`

Všetky tieto funkcie niečo vykonali (vypísali, prečítali, vypočítali, ...) a niektoré z nich vrátili nejakú hodnotu, ktorú sme mohli ďalej spracovať. Tiež sme videli, že niektoré majú rôzny počet parametrov, prípadne sú niekedy volané bez parametrov.

Okrem toho sme pracovali aj s funkciami, ktoré boli definované v iných moduloch:

- keď napíšeme `import random`, môžeme pracovať napr. s funkciami `random.randint()` a `random.randrange()`
- keď napíšeme `import math`, môžeme pracovať napr. s funkciami `math.sin()` a `math.cos()`
- keď napíšeme `import tkinter`, môžeme pracovať napr. s funkciami `tkinter.Canvas()` a `tkinter.mainloop()`

Všetky tieto a tisícky ďalších v Pythone naprogramovali programátori pred nejakým časom, aby nám neskôr zjednodušili samotné programovanie. Vytváranie vlastných funkcií pritom vôbec nie je komplikované a teraz sa to naučíme aj my.

Funkcie

Funkcia je pomenovaný blok príkazov (niekedy sa tomu hovorí aj podprogram). Popisujeme (**definujeme**) ju špeciálnou konštrukciou:

```
def meno_funkcie():       # zapamataj si blok prikazov ako novy prikaz
    prikaz
    prikaz
    ...
```

Keď zapíšeme definíciu funkcie, zatiaľ sa z bloku príkazov (hovoríme tomu **telo funkcie**) nič nevykoná. Táto definícia sa „len“ zapamätá a jej **referencia** sa priradí k zadanému menu - vlastne sa do premennej `meno_funkcie` priradí referencia na telo funkcie. Je to podobné tomu, ako sa priraďovacím príkazom do premennej priradí hodnota z pravej strany príkazu.

Ako prvý príklad zapíšme takúto definíciu funkcie:

```
def vypis():
    print('**********')
    print('**********')
```

Zadefinovali sme funkciu s menom `vypis`, pričom telo funkcie obsahuje dva príkazy na výpis riadkov s hviezdičkami. Celý blok príkazov je odsunutý o 4 medzery rovnako ako sme odsúvali príkazy v cykloch a aj v podmienených príkazoch. Definícia tela funkcie končí vtedy, keď sa objaví riadok, ktorý už nie je odsunutý. Touto definíciou sa ešte žiadne príkazy z tela funkcie nevykonávajú. Na to potrebujeme túto funkciu **zavolať**.

Volanie funkcie

Volanie funkcie je taký zápis, ktorým sa začnú vykonávať príkazy z definície funkcie. Stačí zapísať meno funkcie so zátvorkami a funkcie sa spustí:

```
meno_funkcie()
```

Samozrejme, že funkciu môžeme zavolať až vtedy, keď už Python pozná jej definíciu.

Zavolajme funkciu `vypis` v príkazovom režime:

```
>>> vypis()
**********
**********
>>>
```

Vidíme, že sa vykonali oba príkazy z tela funkcie a potom Python ďalej čaká na ďalšie príkazy. Zapíšme volanie funkcie aj s jej definíciou priamo do skriptu (teda v programovom režime):

```
def vypis():
    print('**********')
    print('**********')

print('hello')
vypis()
print('* Python *')
vypis()
```

Skôr, ako to spustíme, si uvedomme, čo sa udeje pri spustení:

- zapamätá sa definícia funkcie v premennej `vypis`
- vypíše sa slovo `'hello'`
- zavolá sa funkcia `vypis()`
- vypíše riadok s textom `'* Python *'`
- znovu sa zavolá funkcia `vypis()`

A teraz to spustíme:

```
hello
**********
**********
* Python *
**********
**********
```

Zapíšme teraz presné kroky, ktoré sa vykonajú pri volaní funkcie:

1. preruší sa vykonávanie práve bežiaceho programu (Python si presne zapamätá miesto, kde sa to stalo)
2. skočí sa na začiatok volanej funkcie
3. postupne sa vykonajú všetky príkazy
4. keď sa príde na koniec funkcie, zrealizuje sa **návrat** na zapamätané miesto, kde sa prerušilo vykonávanie programu a pokračuje sa vo vykonávaní ďalších príkazov za volaním funkcie

Pre volanie funkcie sú veľmi dôležité okrúhle zátvorky. Bez nich to už nie je volanie, ale len zisťovanie referencie na hodnotu, ktorá je priradená pre toto meno. Napr.

```
>>> vypis()
**********
**********
>>> vypis
<function vypis at 0x0205CB28>
```

Ak by sme namiesto volania funkcie takto zapísali len meno funkcie bez zátvoriek, ale v skripte (teda nie v interaktívnom režime), táto hodnota referencie by sa nevypísala, ale odignorovala. Toto býva dosť častá chyba začiatočníkov, ktorá sa ale ťažšie odhaľuje.

Ak zavoláme funkciu, ktorú sme ešte nedefinovali, Python vyhlási chybu, napr.

```
>>> vipis()
...
NameError: name 'vipis' is not defined
```

Samozrejme, že môžeme volať len definované funkcie.

```
>>> vypis()
**********
**********
>>> vypis = 'ahoj'
>>> vypis
'ahoj'
>>> vypis()
...
TypeError: 'str' object is not callable
```

Hodnotou premennej `vypis` je už teraz znakový reťazec, a ten sa „nedá zavolať“, t.j. nie je „callable“ (tento objekt nie je zavolateľný ako funkcia).

5.1.1 Funkcie môžu kresliť v grafickej ploche

Napíšme teraz funkciu, ktorá do grafickej plochy na náhodnú pozíciu napíše nejaký text, napr. `'Python'`:

```
def text():
    x = random.randrange(300)
    y = random.randrange(300)
    canvas.create_text(x, y, text='Python', font='arial 20')
```

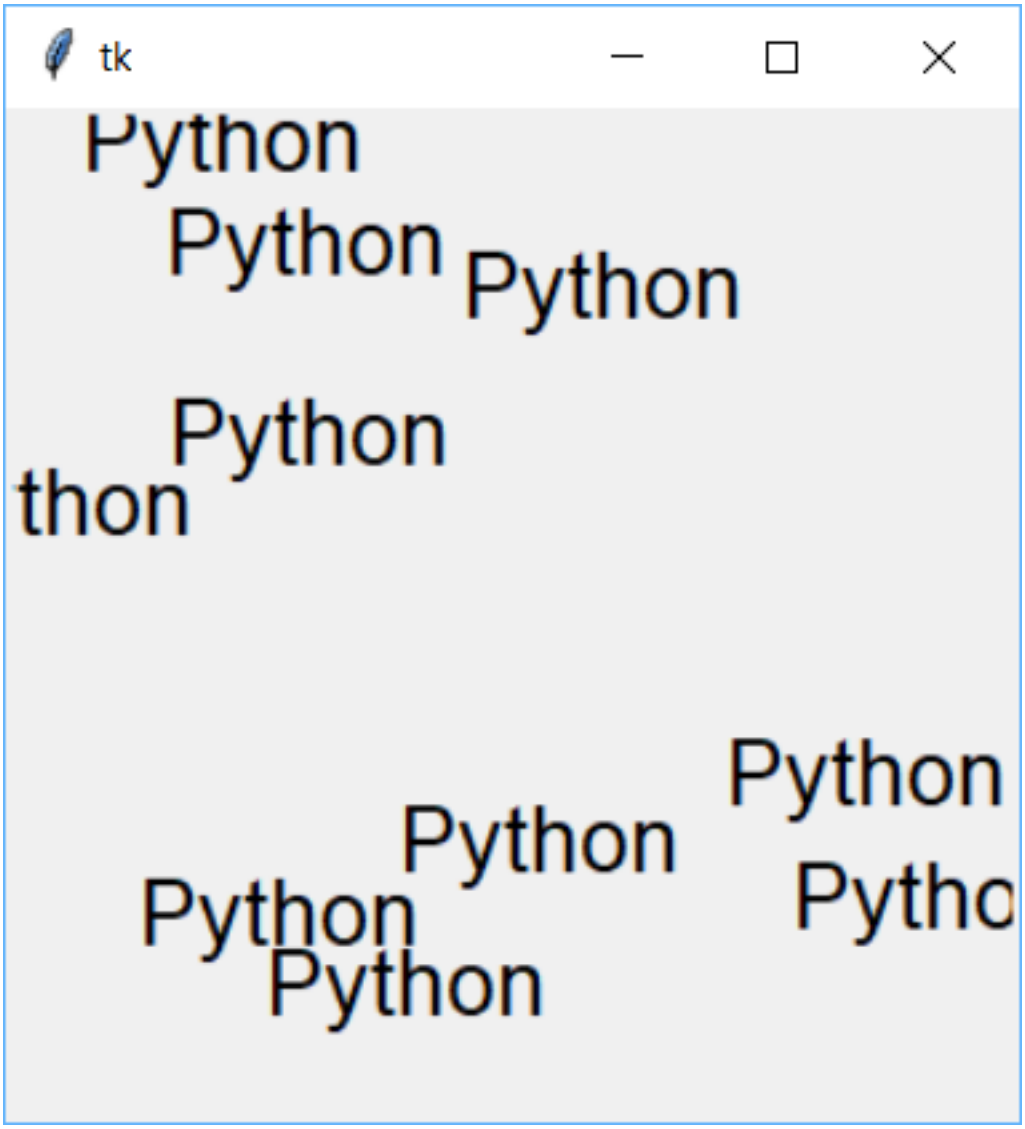
Aby sme túto funkciu mohli zavolať, musí už exitovať `random` aj `canvas`, teda

```
import tkinter
import random

canvas = tkinter.Canvas(width=300, height=300)
canvas.pack()
for i in range(10):
    text()
```

V tejto malej ukážke vidíme tieto novinky:

- v tele funkcie môžeme používať premenné, ktoré sú definované mimo tela funkcie (`random` a `canvas`) - neskôr uvidíme, že im budeme hovoriť **globálne premenné**
- v tele funkcie sme do dvoch premenných `x` a `y` priradili nejaké hodnoty a ďalej sme ich používali v ďalšom príkaze funkcie - neskôr uvidíme, že im budeme hovoriť **lokálne premenné**



5.2 Parametre funkcií

Hotové funkcie, s ktorými sme doteraz pracovali, napr. `print()` alebo `random.randint()`, mali aj parametre, vďaka čomu riešili rôzne úlohy. Parametre slúžia na to, aby sme mohli funkcii lepšie oznámiť, čo špecifické má urobiť: čo sa má vypísať, z akého intervalu má vygenerovať náhodné číslo, akú úsečku má nakresliť, prípadne akej farby, ...

Parametre funkcie

Parametrom funkcie je **dočasná premenná**, ktorá vzniká pri volaní funkcie a prostredníctvom ktorej, môžeme do funkcie *poslať* nejakú hodnotu. Parametre funkcií definujeme počas definovania funkcie v **hlavičke funkcie** a ak ich je viac, oddeľujeme ich čiarkami:

```
def meno_funkcie(parameter):
    prikaz
    prikaz
    ...
```

Môžeme napríklad zapísať:

```
def vypis_hviezdiciek(pocet):
    print('*' * pocet)
```

V prvom riadku definície funkcie (hlavička funkcie) pribudla jedna premenná `pocet` - parameter. Táto premenná vznikne automaticky pri volaní funkcie, preto musíme pri volaní oznámiť hodnotu tohto parametra. Volanie zapíšeme:

```
>>> vypis_hviezdiciek(30)
******************************
>>> for i in range(1, 10):
        vypis_hviezdiciek(i)

*
**
***
****
*****
******
*******
********
*********
```

Pri volaní sa „skutočná hodnota“ **priradí** do parametra funkcie (premenná `pocet`).

Už predtým sme popísali mechanizmus volania funkcie, ale to sme ešte nepoznali parametre. Teraz doplníme tento postup o spracovanie parametrov. Najprv trochu terminológie:

- pri definovaní funkcie v hlavičke funkcie uvádzame tzv. **formálne parametre**: sú to nové premenné, ktoré vzniknú až pri volaní funkcie
- pri volaní funkcie musíme do zátvoriek zapísať hodnoty, ktoré sa stanú tzv. **skutočnými parametrami**: tieto hodnoty sa pri volaní priradia do formálnych parametrov

Mechanizmus volania vysvetlíme na volaní `vypis_hviezdiciek(30)`:

1. zapamätá sa návratová adresa volania
2. vytvorí sa **nová** premenná `pocet` (**formálny parameter**) a priradí sa do nej hodnota **skutočného parametra** `30`
3. vykonajú sa všetky príkazy v definícii funkcie (**telo funkcie**)
4. zrušia sa všetky premenné, ktoré vznikli počas behu funkcie
5. riadenie sa vráti na miesto, kde bolo volanie funkcie

Zapíšme novú funkciu `cerveny_kruh()`, ktorá bude mať dva parametre: súradnice stredu kruhu. Funkcia nakreslí kruh s polomerom 20 s daným stredom:

```
def cerveny_kruh(x, y):
    canvas.create_oval(x-20, y-20, x+20, y+20, fill='red')
```

a môžeme ju zavolať napr. takto:

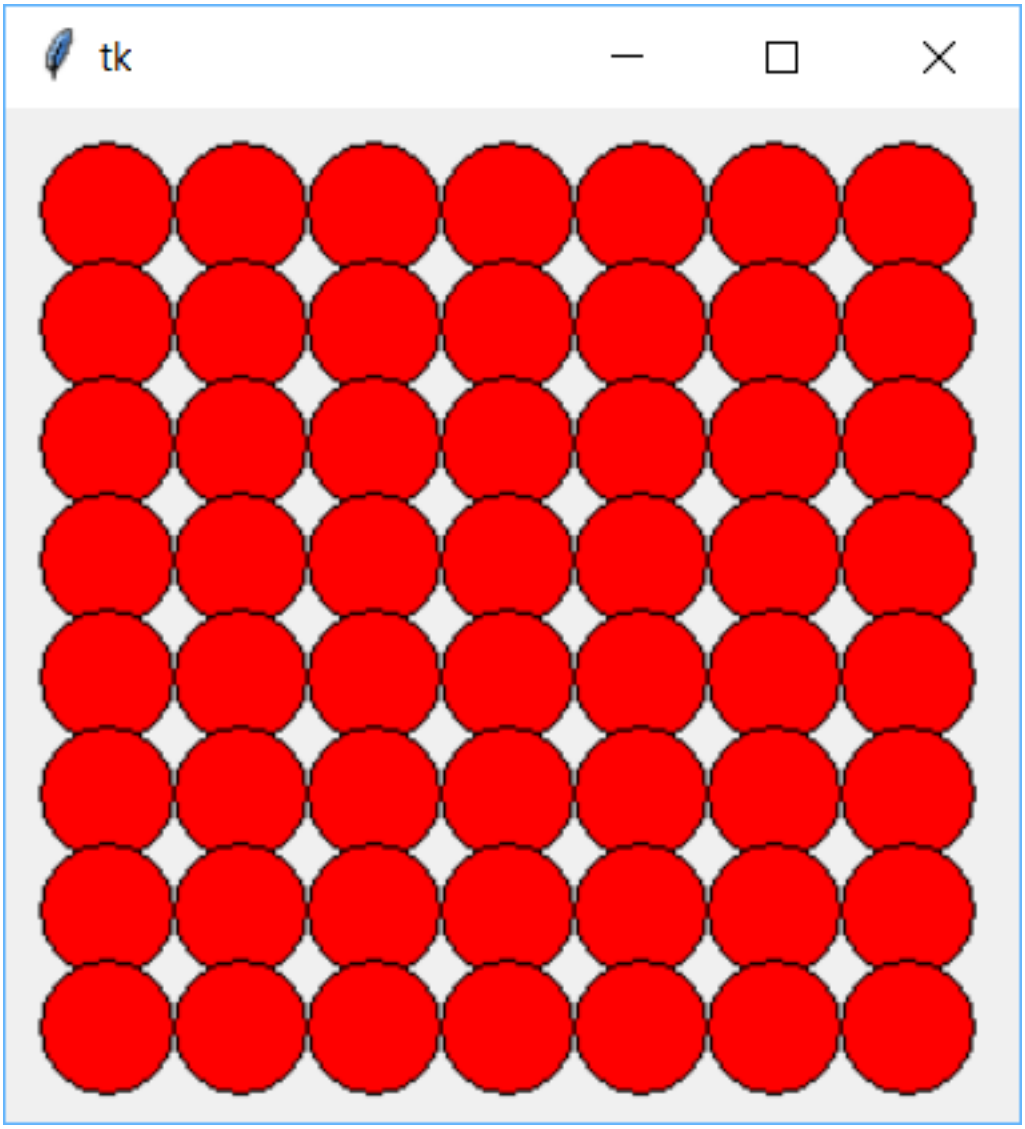
```
import tkinter
import random

canvas = tkinter.Canvas(width=300, height=300)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
canvas.pack()
for x in range(30, 280, 40):
    for y in range(30, 280, 35):
        cerveny_kruh(x, y)
```



Aj v tomto príklade si popíšme **Mechanizmus volania** funkcie:

1. zapamätá sa návratová adresa volania
2. vytvoria sa dve **nové** premenné `x` a `y` (**formálne parametre**) a priradia sa do nej hodnoty **skutočných parametrov** `30` a `20` (prvé volanie funkcie)
3. vykonajú sa všetky príkazy v definícii funkcie (**telo funkcie**)
4. zrušia sa všetky premenné, ktoré vznikli počas behu funkcie, teda `x` aj `y`
5. riadenie sa vráti na miesto, kde bolo volanie funkcie

Už vieme, že priraďovací príkaz vytvára premennú a referenciou ju spojí s nejakou hodnotou. Premenné, ktoré **vzniknú počas behu funkcie**, sa stanú **lokálnymi premennými**: budú existovať len počas tohto behu a po skončení

funkcie, sa automaticky zrušia. Aj parametre vznikajú pri štarte funkcie a zanikajú pri jej skončení: tieto premenné sú pre funkciu tiež lokálnymi premennými.

V nasledovnom príklade funkcie `vypis_sucet()` počítame a vypisujeme súčet čísel od 1 po zadané `n`:

```
def vypis_sucet(n):
    sucet = 1
    print(1, end=' ')
    for i in range(2, n + 1):
        sucet = sucet + i
        print('+', i, end=' ')
    print('=', sucet)
```

Pri volaní funkcie sa pre parameter `n` = 10 vypíše:

```
>>> vypis_sucet(10)
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55
```

Počas behu vzniknú 2 lokálne premenné (`sucet` a `i`) a jeden parameter, ktorý je pre funkciu tiež lokálnou premennou:

- `n` vznikne pri štarte funkcie aj s hodnotou 10
- `sucet` vznikne pri prvom priradení `sucet = 1`
- `i` vznikne pri štarte for-cyklu

Po skončení behu funkcie sa všetky tieto premenné automaticky zrušia.

Pozrime sa na **lokálne premenné**, ktoré vznikajú vo funkcii `nahodne_kruhy(n, farba)`:

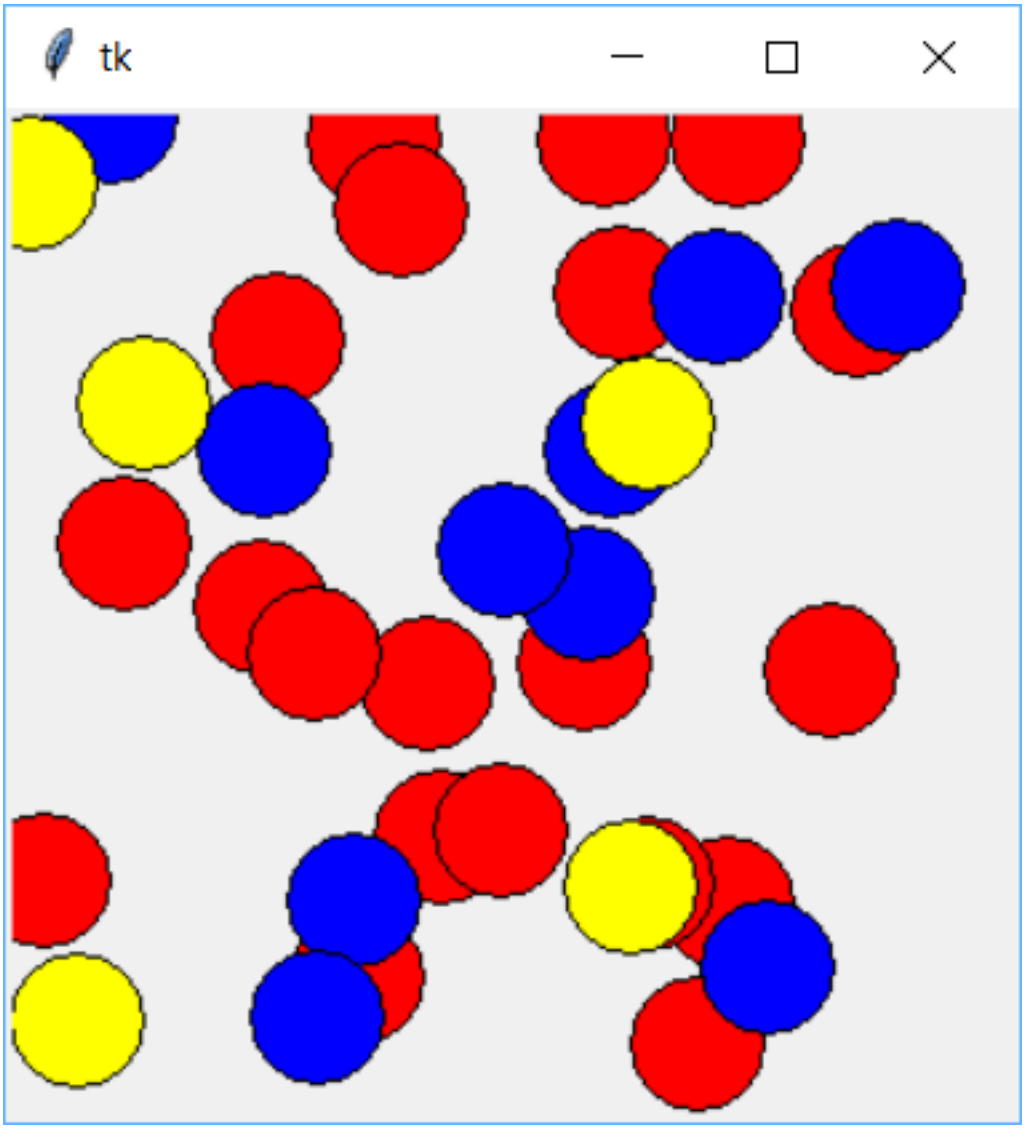
```
def nahodne_kruhy(n, farba):
    for i in range(n):
        x = random.randrange(300)
        y = random.randrange(300)
        canvas.create_oval(x-20, y-20, x+20, y+20, fill=farba)
```

Napr.

```
import tkinter
import random

canvas = tkinter.Canvas(width=300, height=300)
canvas.pack()
nahodne_kruhy(20, 'red')
nahodne_kruhy(10, 'blue')
nahodne_kruhy(5,  'yellow')
```

Program postupne nakreslí nanáhodné pozície 20 červených, 10 modrých a 5 žltých kruhov.



Všimnite si vo funkcii `nahodne_kruhy()` tri lokálne premenné (`i`, `x`, `y`) a dva parametre (`n`, `farba`), ktoré sú pre funkciu tiež lokálne premenné.

5.2.1 Menný priestor

Aby sme lepšie pochopili ako naozaj fungujú **lokálne premenné**, musíme rozumieť, čo to je a ako funguje **menný priestor** (namespace). Najprv trochu ďalšej terminológie: všetky identifikátory v Pythone sú jedným z troch typov (Python má pre identifikátory 3 rôzne tabuľky mien):

- **štandardné**, napr. `int`, `print`, ...
  - hovorí sa tomu **builtins**
- **globálne** - definujeme ich na najvyššej úrovni mimo funkcií, napr. funkcie `vypis_hviezdiciek`, `vypis_sucet`, `nahodne_kruhy`, alebo aj premenné `random`, `tkinter`, `canvas` (zrejme `random` a `tkinter` sú referenciami na dva importované moduly)
  - hovorí sa tomu **main**
- **lokálne** - vznikajú počas behu funkcie

Tabuľka štandardných mien (builtins) je pre celý program len jedna, tiež tabuľka globálnych mien (main) je len jedna, ale každá funkcia má svoju „súkromnú“ lokálnu tabuľku mien, ktorá vznikne pri štarte (zavolaní) funkcie a zruší sa pri konci vykonávania funkcie.

Keď na nejakom mieste použijeme identifikátor, Python ho najprv hľadá (v tzv. **menných priestoroch**):

- v lokálnej tabuľke mien, ak tam tento identifikátor nenájde, hľadá ho
- v globálnej tabuľke mien, ak tam tento identifikátor nenájde, hľadá ho
- v štandardnej tabuľke mien

Ak nenájde v žiadnej z týchto tabuliek, hlási chybu `NameError: name 'identifikátor' is not defined`.

Príkaz (štandardná funkcia) `dir()` vypíše tabuľku globálnych mien. Hoci pri štarte Pythonu by táto tabuľka mala byť prázdna, obsahuje niekoľko špeciálnych mien, ktoré začínajú aj končia znakmi `'__'`:

```
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
```

Keď teraz vytvoríme nejaké nové globálne mená, objavia sa aj v tejto globálnej tabuľke:

```
>>> premenna = 2015
>>> def funkcia():
        pass

>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__',
 'funkcia', 'premenna']
```

Podobne sa vieme dostať aj k tabuľke štandardných mien (builtins):

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', ...
```

Takto sa vypíšu všetky preddefinované mená. Vidíme medzi nimi napr. `'int'`, `'print'`, `'range'`, `'str'`,...

S týmito tabuľkami súvisí aj príkaz na zrušenie premennej.

príkaz `del`

Príkazom `del` zrušíme identifikátor z tabuľky mien. Formát príkazu:

```
del premenná
```

Príkaz najprv zistí, v ktorej tabuľke sa identifikátor nachádza (najprv pozrie do lokálnej a keď tam nenájde, tak do globálnej tabuľky) a potom ho z tejto tabuľky vyhodí. Príkaz ale nefunguje pre štandardné mená.

Ukážme to na príklade: identifikátor `print` je menom štandardnej funkcie (v štandardnej tabuľke mien). Ak v priamom režime (čo je globálna úroveň mien) do premennej `print` priradíme nejakú hodnotu, toto meno vznikne v globálnej tabuľke:

```
>>> print('ahoj')
ahoj
>>> print=('ahoj')          # do print sme priradili nejaku hodnotu
>>> print
'ahoj'
>>> print('ahoj')
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
...
TypeError: 'str' object is not callable
```

Teraz už `print` nefunguje ako funkcia na výpis hodnôt, ale len ako obyčajná globálna premenná. Ale v štandardnej tabuľke mien `print` stále existuje, len je táto premenná **prekrytá** globálnym menom. Python predsa najprv prehľadáva globálnu tabuľku a až keď sa tam nenájde, hľadá sa v štandardnej tabuľke. A ako môžeme vrátiť funkčnosť štandardnej funkcie `print`? Stačí vymazať identifikátor z globálnej tabuľky:

```
>>> del print
>>> print('ahoj')
ahoj
```

Vymazaním globálneho mena `print` ostane definovaný len identifikátor v tabuľke štandardných mien, teda opäť začne fungovať funkcia na výpis hodnôt.

Pozrime sa teraz na prípad, keď sa v tele funkcie bude nachádzať volanie inej funkcie (tzv. **vnorené volanie**), napr.

```
def vypis_hviezdiciek(pocet):
    print('*' * pocet)

def trojuholnik(n):
    for i in range(1, n + 1):
        vypis_hviezdiciek(i)
```

Pri ich definovaní v globálnom mennom priestore vznikli dva identifikátory: `vypis_hviezdiciek` a `trojuholnik`. Zavoláme funkciu `trojuholnik`:

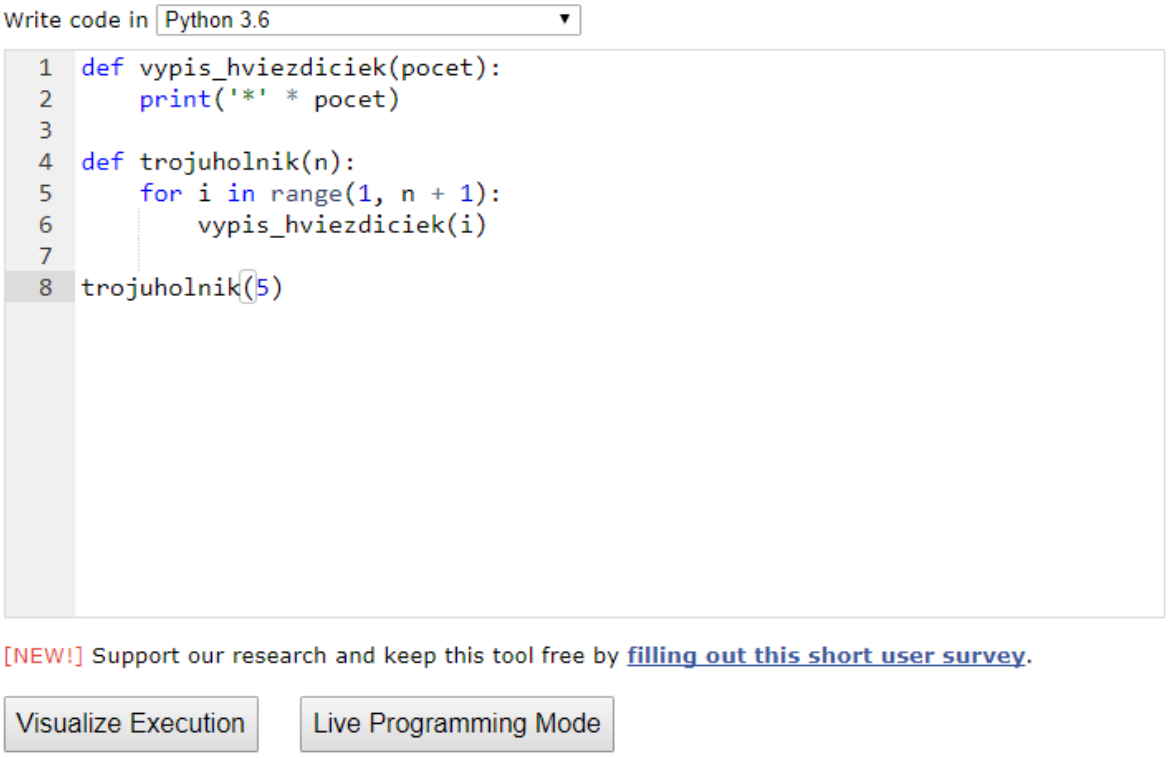
```
>>> trojuholnik(5)
```

Najprv sa pre túto funkciu vytvorí jej menný priestor (lokálna tabuľka mien) s dvomi lokálnymi premennými: `n` a `i`. Teraz **pri každom** (vnorenom) volaní `vypis_hviezdiciek(i)` sa pre túto funkciu:

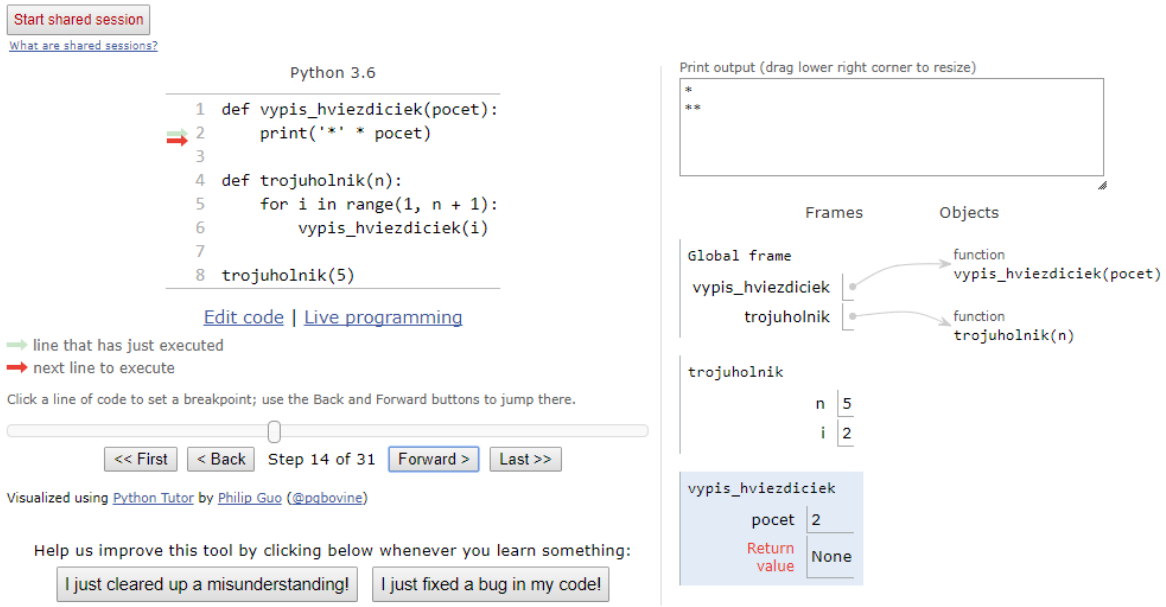
- vytvorí nový menný priestor s jedinou premennou `pocet`
- vykoná sa príkaz `print()`
- nakoniec sa zruší jej menný priestor, t.j. zanikne premenná `pocet`

Môžeme to odkrokovať pomocou http://www.pythontutor.com/visualize.html#mode=edit (zapneme voľbu `Python 3.6`):

- najprv do editovacieho okna zapíšeme nejaký program, napr.



- spustíme vizualizáciu pomocou tlačidla **Visualize Execution** a potom niekoľkokrát tlačíme tlačidlo **Forward >**



Všimnite si, že v pravej časti tejto stránky sa postupne zobrazujú menné priestory (tu sa nazývajú **frame**):

- najprv len globálny priestor s premennými `vypis_hviezdiciek` a `trojuholnik`
- potom sa postupne objavujú a aj miznú lokálne priestory týchto dvoch funkcií - na obrázku vidíme oba tieto menné priestory tesne pred ukončením vykonávania funkcie `trojuholnik` s parametrom `2`

5.2.2 Funkcie s návratovou hodnotou

Väčšina štandardných funkcií v Pythone na základe parametrov vráti nejakú hodnotu, napr.

```
>>> abs(-5.5)
5.5
>>> round(2.36, 1)
2.4
```

Funkcie, ktoré sme zatiaľ vytvárali my, takú možnosť nemali: niečo počítali, niečo vypisovali, ale žiadnu návratovú hodnotu nevytvárali. Aby funkcia mohla vrátiť nejakú hodnotu ako výsledok volania funkcie, musí sa v jej tele objaviť príkaz `return`, napr.

```
def meno(parametre):
    prikaz
    prikaz
    ...
    return hodnota                  # takato funkcia bude vracat vyslednu hodnotu
```

Príkazom `return` sa ukončí výpočet funkcie (zruší sa jej menný priestor) a uvedená hodnota sa stáva výsledkom funkcie, napr.

```
def eura_na_koruny(eura):                   # prepočítanie na české koruny
    koruny = round(eura * 25.9, 2)
    return koruny
```

môžeme otestovať:

```
>>> print('mas', 123, 'euro, co je', eura_na_koruny(123), 'ceskych korun')
mas 123 euro, co je 3185.7 ceskych korun
```

Niekedy potrebujeme návratovú hodnotu počítať nejakým cyklom, napr. nasledovná funkcia počíta súčet čísel od 1 do `n`:

```
def suma(n):
    vysledok = 0
    while n > 0:
        vysledok += n
        n -= 1
    return vysledok
```

Zároveň vidíme, že formálny parameter (je to predsa lokálna premenná) môžeme v tele funkcie modifikovať.

Už sme videli, že rozlišujeme dva typy funkcií:

- také, ktoré niečo robia (napr. vypisujú, kreslia, ...), ale nevracajú návratovú hodnotu (neobsahujú `return` s nejakou hodnotou)
- také, ktoré niečo vypočítajú a vrátia nejakú výslednú hodnotu - musia obsahovať `return` s návratovou hodnotou

Ďalej ukážeme, že rôzne funkcie môžu vracať hodnoty rôznych typov. Najprv číselné funkcie.

Výsledkom funkcie je číslo

Nasledovná funkcia počíta `n`-tú mocninu dvojky a tento výsledok ešte zníži o 1:

```
def pocitaj(n):
    return 2 ** n - 1
```

Zrejme výsledkom je vždy len číslo.

Ak chceme funkciu otestovať, buď ju spustíme s konkrétnym parametrom, alebo napíšeme cyklus, ktorý našu funkciu spustí s konkrétnymi hodnotami (niekedy na testovanie píšeme ďalšiu testovaciu funkciu, ktorá nerobí nič iné, „len“ testuje funkciu pre rôzne hodnoty a porovnáva ich s očakávanými výsledkami), napr.

```
>>> pocitaj(5)
31
>>> for i in 1, 2, 3, 8, 10, 16, 20, 32:
        print(f'pocitaj({i}) = {pocitaj(i)}')

pocitaj(1) = 1
pocitaj(2) = 3
pocitaj(3) = 7
pocitaj(8) = 255
pocitaj(10) = 1023
pocitaj(16) = 65535
pocitaj(20) = 1048575
pocitaj(32) = 4294967295
```

Ďalšia funkcia zisťuje dĺžku (počet znakov) zadaného reťazca. Využíva to, že for-cyklus vie prejsť všetky znaky reťazca a s každým môže niečo urobiť, napr. zvýšiť počítadlo o 1:

```
def dlzka(retazec):
    pocet = 0
    for znak in retazec:
        pocet += 1
    return pocet
```

Otestujeme:

```
>>> dlzka('Python')
6
>>> dlzka(10000 * 'ab')
20000
```

5.2.3 Výsledkom funkcie je logická hodnota

Funkcie môžu vracať aj hodnoty iných typov, napr.

```
def parne(n):
    return n % 2 == 0
```

vráti `True` alebo `False` podľa toho či je `n` párne (zvyšok po delení 2 bol 0), vtedy vráti `True`, alebo nepárne (zvyšok po delení 2 nebol 0) a vráti `False`. Túto istú funkciu môžeme zapísať aj tak, aby bolo lepšie vidieť tieto dve rôzne návratové hodnoty:

```
def parne(n):
    if n % 2 == 0:
        return True
    else:
        return False
```

Hoci táto verzia robí presne to isté ako predchádzajúca, skúsení programátori radšej používajú kratšiu prvú verziu. Keď chceme túto funkciu otestovať, môžeme zapísať:

```
>>> parne(10)
True
>>> parne(11)
False
>>> for i in range(20, 30):
        print(i, parne(i))

20 True
21 False
22 True
23 False
24 True
25 False
26 True
27 False
28 True
29 False
```

5.2.4 Výsledkom funkcie je reťazec

Napíšme funkciu, ktorá vráti nejaký reťazec v závislosti od hodnoty parametra:

```
def farba(ix):
    if ix == 0:
        return 'red'
    elif ix == 1:
        return 'blue'
    else:
        return 'yellow'
```

Funkcia vráti buď červenú, alebo modrú, alebo žltú farbu v závislosti od hodnoty parametra.

Opäť by ju bolo dobre najprv otestovať, napr.

```
>>> for i in range(6):
        print(i, farba(i))

0 red
1 blue
2 yellow
3 yellow
4 yellow
5 yellow
```

Uvedomte si, prečo ju môžeme zapísať aj takto bez `else` vetiev:

```
def farba(ix):
    if ix == 0:
        return 'red'
    if ix == 1:
        return 'blue'
    return 'yellow'
```

V takýchto prípadoch je na vás, ktorý zápis použijete, ktorý z nich sa vám zdá čitateľnejší.

Typy parametrov a typ výsledku

Python nekontroluje typy parametrov, ale kontroluje, čo sa s nimi robí vo funkcii. Napr. funkcia:

```
def pocitaj(x):
    return 2 * x + 1
```

bude fungovať pre čísla, ale pre reťazec spadne:

```
>>> pocitaj(5)
11
>>> pocitaj('a')
...
TypeError: Can't convert 'int' object to str implicitly
```

V tele funkcie ale môžeme kontrolovať typ parametra, napr. takto

```
def pocitaj(x):
    if type(x) == str:
        return 2 * x + '1'
    else:
        return 2 * x + 1
```

a potom volanie

```
>>> pocitaj(5)
11
>>> pocitaj('a')
'aa1'
```

Neskôr sa naučíme testovať typ nejakých hodnôt správnejším spôsobom, ale zatiaľ nám bude stačiť, keď to budeme riešiť takto jednoducho.

Napriek tomuto niektoré funkcie môžu fungovať rôzne pre rôzne typy, napr.

```
def urob(a, b):
    return 2 * a + 3 * b
```

niekedy funguje pre čísla aj pre reťazce. Otestujte.

5.2.5 Grafické funkcie

Zadefinujeme funkcie, pomocou ktorých sa nakreslí 5000 náhodných farebných bodiek, ktoré budú zafarbené podľa nejakých pravidiel:

```
import tkinter
import random
import math

def vzd(x1, y1, x2, y2):
    return math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

def kresli_bodku(x, y, farba):
    canvas.create_oval(x-5, y-5, x+5, y+5, fill=farba, outline='')

def farebne_bodky(pocet):
```

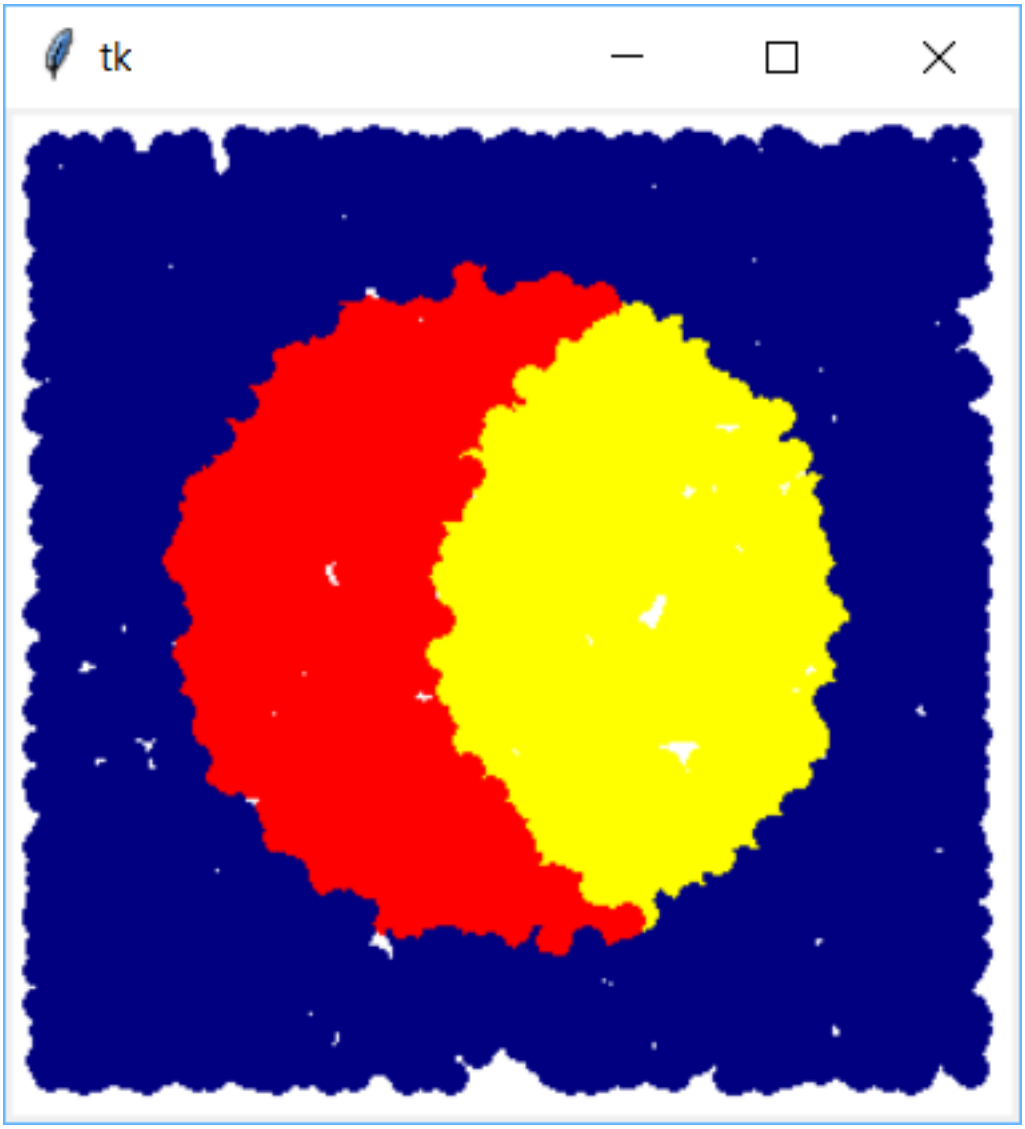
(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    for i in range(pocet):
        x = random.randint(10, 290)
        y = random.randint(10, 290)
        if vzd(x, y, 150, 150) > 100:
            kresli_bodku(x, y, 'navy')
        elif vzd(x, y, 230, 150) > 100:
            kresli_bodku(x, y, 'red')
        else:
            kresli_bodku(x, y, 'yellow')

canvas = tkinter.Canvas(bg='white', width=300, height=300)
canvas.pack()
farebne_bodky(5000)
```

Funkcia `vzd()` počíta vzdialenosť dvoch bodov `(x1, y1)` a `(x2, y2)` v rovine - tu sa použil známy vzorec z matematiky. Táto funkcia nič nevypisuje, ale vracia číselnú hodnotu (desatinné číslo). Ďalšia funkcia `kresli_bodku()` nič nevracia, ale vykreslí v grafickej ploche malý kruh s polomerom 5, ktorý je zafarbený zadanou farbou. Tretia funkcia `farebne_bodky()` dostáva ako parameter počet bodiek, ktoré má nakresliť: funkcia na náhodné pozície nakreslí príslušný počet bodiek, pričom tie, ktoré sú od bodu `(150, 150)` vzdialené viac ako 100, budú tmavomodré (farba `'navy'`), tie, ktoré sú od bodu `(230, 150)` vzdialené viac ako 100, budú červené a všetku ostatné budú žlté. Všimnite si, že sme za definíciami všetkých funkcií napísali samotný program, ktorý využíva práve zadefinované funkcie. Po spustení dostávame približne takýto obrázok:



5.2.6 Náhradná hodnota parametra

Naučíme sa zadefinovať parametre funkcie tak, aby sme pri volaní nemuseli uviesť všetky hodnoty skutočných parametrov, ale niektoré sa automaticky dosadia, tzv. náhradnou hodnotou (default), napr.

```
def kresli_bodku(x, y, farba='red', r=5):
    canvas.create_oval(x-r, y-r, x+r, y+r, fill=farba, outline='')
```

V hlavičke funkcie môžeme k niektorým parametrom uviesť náhradnú hodnotu (vyzerá to ako priradenie). V tomto prípade to označuje, že ak tomuto formálnemu parametru nebude zodpovedať skutočný parameter, dosadí sa práve táto náhradná hodnota. Pritom musí platiť, že keď nejakému parametru v definícii funkcie určíte, že má náhradnú hodnotu, tak náhradnú hodnotu musíte zadať aj všetkým ďalším formálnym parametrom, ktoré sa nachádzajú v zozname parametrov za ním (ak sme zadefinovali náhradnú hodnotu pre parameter `farba`, musíme nejakú zadefinovať aj pre parameter `r`).

Teraz môžeme zapísať aj takéto volania tejto funkcie:

```
kresli_bodku(100, 200, 'blue', 3)     # farba bude 'blue' a r bude 3
kresli_bodku(150, 250, 'blue')        # farba bude 'blue' a r bude 5
kresli_bodku(200, 200)                # farba bude 'red'  a r bude 5
```

5.2.7 Parametre volané menom

Predpokladajme rovnakú definíciu funkcie `kresli_bodku`:

```
def kresli_bodku(x, y, farba='red', r=5):
    canvas.create_oval(x-r, y-r, x+r, y+r, fill=farba, outline='')
```

Python umožňuje funkcie s parametrami volať tak, že skutočné parametre neurčujeme pozične (prvému skutočnému zodpovedá prvý formálny, druhému druhý, atď.) ale priamo pri volaní uvedieme meno parametra. Takto môžeme určiť hodnotu ľubovoľnému parametru. Napr. všetky tieto volania sú korektné:

```
kresli_bodku(10, 20, r=10)
kresli_bodku(farba='green', x=10, y=20)
kresli_bodku(r=7, farba='yellow', y=20, x=30)
```

Samozrejme aj pri takomto volaní môžeme vynechať len tie parametre, ktoré majú určenú náhradnú hodnotu, všetky ostatné parametre sa musia v nejakom poradí objaviť v zozname skutočných parametrov.

5.2.8 Farebný model RGB

Keďže už vieme vytvárať reťazce so šestnástkovým zápisom čísel (napr. `'f{číslo:02x}'` alebo `'{:02x}'.format(číslo)`) zapíšeme funkciu `rgb()`, ktorá bude vytvárať farby pomocou RGB-modelu:

```
def rgb(r, g, b):
    return f'#{r:02x}{g:02x}{b:02x}'
```

alebo

```
def rgb(r, g, b):
    return '#{:02x}{:02x}{:02x}'.format(r, g, b)
```

otestujme:

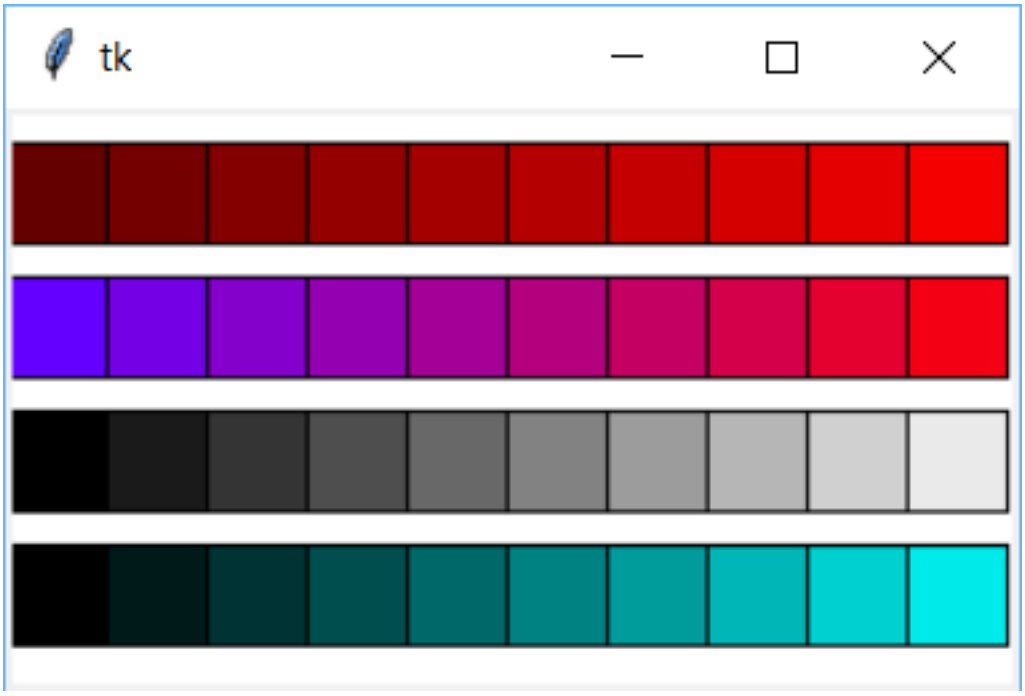
```
>>> rgb(255, 255, 0)
'#ffff00'
>>> rgb(0, 100, 0)
'#006400'
```

Funkciu `rgb()` môžeme využiť napr. na kreslenie farebných štvorcov:

```
def stvorec(strana, x, y, farba=''):
    canvas.create_rectangle(x, y, x+strana, y+strana, fill=farba)

for i in range(10):
    stvorec(30, i*30, 10, rgb(100+16*i, 0, 0))
    stvorec(30, i*30, 50, rgb(100+16*i, 0, 255-26*i))
    stvorec(30, i*30, 90, rgb(26*i, 26*i, 26*i))
    stvorec(30, i*30, 130, rgb(0, 26*i, 26*i))
```

Tento program nakreslí takýchto 40 zafarbených štvorcov:



Náhodné farby

Ak potrebujeme generovať náhodnú farbu, ale stačí nám iba jedna z dvoch možností, môžeme to urobiť napr. takto:

```
def nahodna2_farba():
    if random.randrange(2):
        return 'blue'
    return 'red'
```

Podobne by sa zapísala funkcia, ktorá generuje náhodnú farbu jednu z troch a pod.

Ak ale chceme úplne náhodnú farbu z celej množiny všetkých farieb, využijeme RGB-model napr. takto

```
def rgb(r, g, b):
    return f'#{r:02x}{g:02x}{b:02x}'

def nahodna_farba():
    return rgb(random.randrange(256), random.randrange(256), random.randrange(256))
```

Keďže takto sa vlastne vygeneruje náhodné šestnástkové šesťciferné číslo, toto isté vieme zapísať aj takto:

```
def nahodna_farba():
    return f'#{random.randrange(256*256*256):06x}'      # co je aj 256**3
```

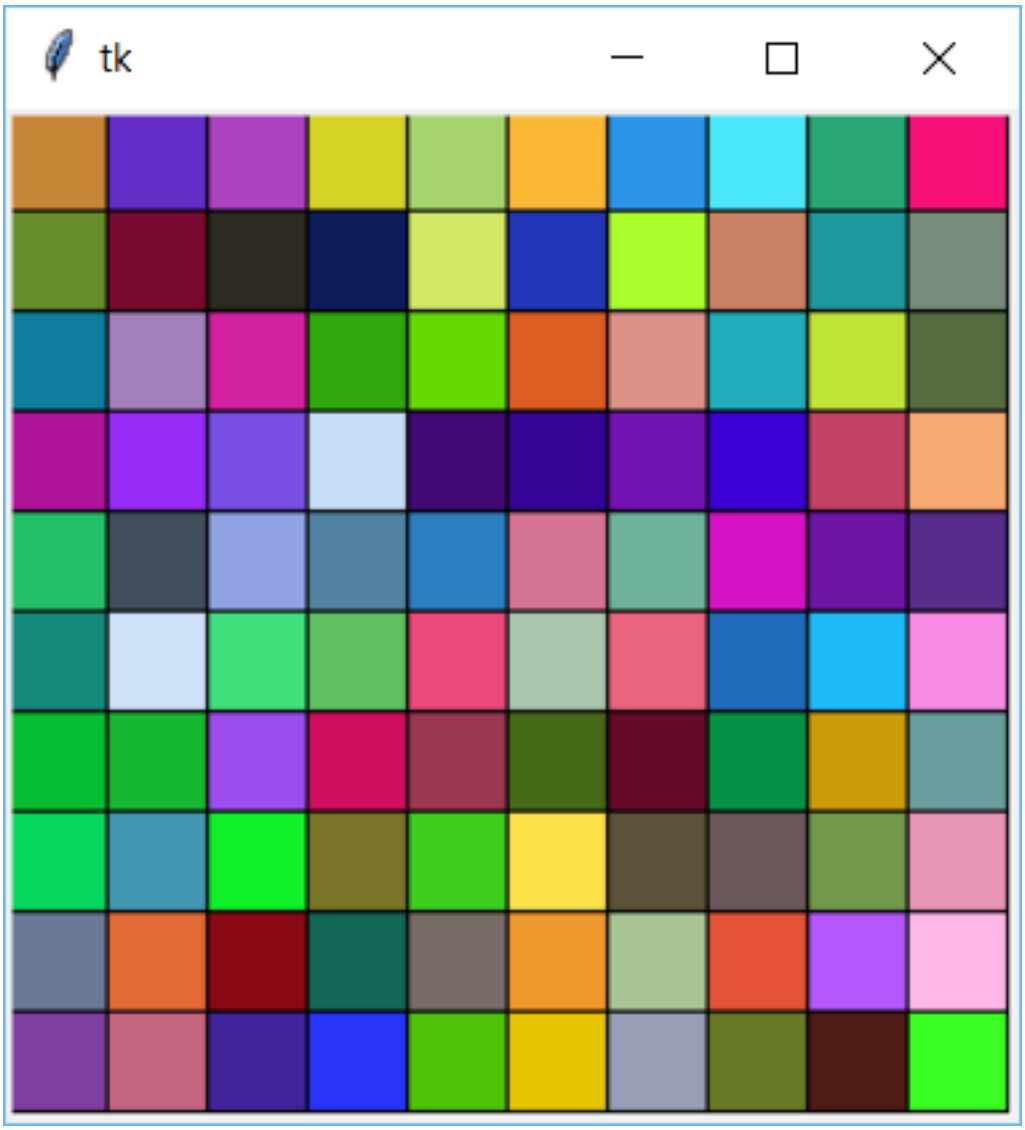
alebo

```
def nahodna_farba():
    return '#{:06x}'.format(random.randrange(256*256*256))   # co je aj 256**3
```

Môžeme vygenerovať štvorcovú sieť náhodných farieb:

```
for y in range(0, 300, 30):
    for x in range(0, 300, 30):
        stvorec(30, x, y, nahodna_farba())
```

Dostaneme nejaký takýto obrázok:



5.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/
- v riešeniach úloh používajte len konštrukcie a funkcie, ktoré sme sa učili na doterajších prednáškach

1. Napíšte funkciu `vypis_delitele(cislo)`, ktorá vypíše do jedného riadka všetky delitele daného čísla.

- napr.

```
>>> vypis_delitele(24)
1 2 3 4 6 8 12 24
```

2. Napíšte funkciu `sucet_delitelov(cislo)`, ktorá vráti súčet všetkých deliteľov daného čísla. Funkcia nič nevypisuje, funkcia vracia (pomocou `return`) nejakú hodnotu.

   - napr.

```
>>> x = sucet_delitelov(24)
>>> x
60
```

3. Napíšte funkciu `je_dokonale(cislo)`, ktorá pomocou funkcie `sucet_delitelov()` zistí, či je dané číslo **dokonalé**, t.j. že súčet všetkých menších deliteľov ako samotné číslo sa rovná samotnému číslu. Napr. delitele čísla 6 (menšie ako 6) sú 1, 2, 3. Ich súčet je 6. Preto je číslo 6 dokonalé. Funkcia nič nevypisuje, funkcia vracia (pomocou `return`) nejakú hodnotu.

   - napr.

```
>>> je_dokonale(6)
True
>>> je_dokonale(24)
False
```

4. Napíšte funkciu `vsetky_dokonale(od, do)`, ktorá vypíše dokonalé čísla v danom intervale. Táto funkcia využije funkciu `je_dokonale()`.

   - napr.

```
>>> vsetky_dokonale(1, 30)
6 je dokonale
28 je dokonale
```

5. Napíšte funkciu `nsd(a, b)`, ktorá počíta najväčší spoločný deliteľ dvoch čísel. Funkcia nič nevypisuje, funkcia vracia (pomocou `return`) nejakú hodnotu.

   - napr.

```
>>> nsd(21, 15)
3
>>> nsd(1000000, 17)
1
```

   - môžete využiť tzv. Euklidov algoritmus
   - odkrokujte vaše riešenie pomocou stránky http://www.pythontutor.com/visualize.html#mode=edit

6. Napíšte funkciu `pocet_delitelov(cislo)`, ktorá pre dané číslo zistí počet všetkých deliteľov. Napr. delitele čísla 6 sú 1, 2, 3, 6, preto funkcia vráti 4. Funkcia nič nevypisuje, funkcia vracia (pomocou `return`) nejakú hodnotu.

   - napr.

```
>>> pocet_delitelov(6)
4
>>> pocet_delitelov(17)
2
```

7. Napíšte funkciu `je_prvocislo(cislo)`, ktorá pomocou funkcie `pocet_delitelov()` zistí (vráti `True` alebo `False`), či je to prvočíslo (je deliteľné len 1 a samým sebou).

   - napr.

```
>>> je_prvocislo(6)
False
>>> je_prvocislo(17)
True
```

8. Napíšte funkciu `vsetky_prvocisla(od, do)`, ktorá do jedného riadka vypíše všetky prvočísla v danom intervale.

   - napr.

```
>>> vsetky_prvocisla(1, 30)
2 3 5 7 11 13 17 19 23 29
```

   - odkrokujte vaše riešenie pomocou stránky http://www.pythontutor.com/visualize.html#mode=edit

9. Napíšte funkciu `sucet_mocnin2(n)`, ktorá vráti súčet mocnín dvojky s exponentmi menšími ako `n`. Napr. `sucet_mocnin2(5)`, vráti hodnotu 1+2+4+8+16, t.j. hodnotu 31.

   - otestujte

```
>>> for i in range(7):
        print(i, sucet_mocnin2(i))

0 0
1 1
2 3
3 7
4 15
5 31
6 63
```

   - pokúste sa odhadnúť, akým vzorcom by sme vedeli vypočítať rovnaký výsledok ako dáva funkcia `sucet_mocnin2()` ale bez použitia cyklu.

10. Napíšte funkciu `sucet_mocnin2a(n)`, ktorá vráti súčet prevrátených mocnín dvojky s exponentmi menšími ako `n`. Napr. `sucet_mocnin2a(5)`, vráti hodnotu 1/1+1/2+1/4+1/8+1/16, t.j. hodnotu 1.9375.

    - otestujte

```
>>> for i in range(7):
        print(i, sucet_mocnin2a(i))

0 0
1 1.0
2 1.5
3 1.75
4 1.875
5 1.9375
6 1.96875
```

    - všimnite si, že výsledok sa pre väčšie `n` blíži k nejakej konštante

11. Napíšte funkciu `kocka()` bez parametrov, ktorá pri každom zavolaní vráti náhodné číslo z intervalu od 1 do 6.

    - otestujte

```
>>> for i in range(20):
        print(kocka(), end=' ')
...
```

- opravte tento testovací cyklus tak, že v postupnosti vypisovaných čísel za každé, ktoré je rovnaké ako predchádzajúce, vypíšete aj znak `'*'`, napr.

```
>>> ... for i in range(20): ...
6 4 4 * 4 * 3 5 4 3 5 5 * 1 2 1 1 * 1 * 1 * 2 2 * 6 1
```

12. Napíšte funkciu `pocet_samohlasok(text)`, ktorá pre zadaný znakový reťazec zistí počet samohlások, ktoré obsahuje.

    - napr.

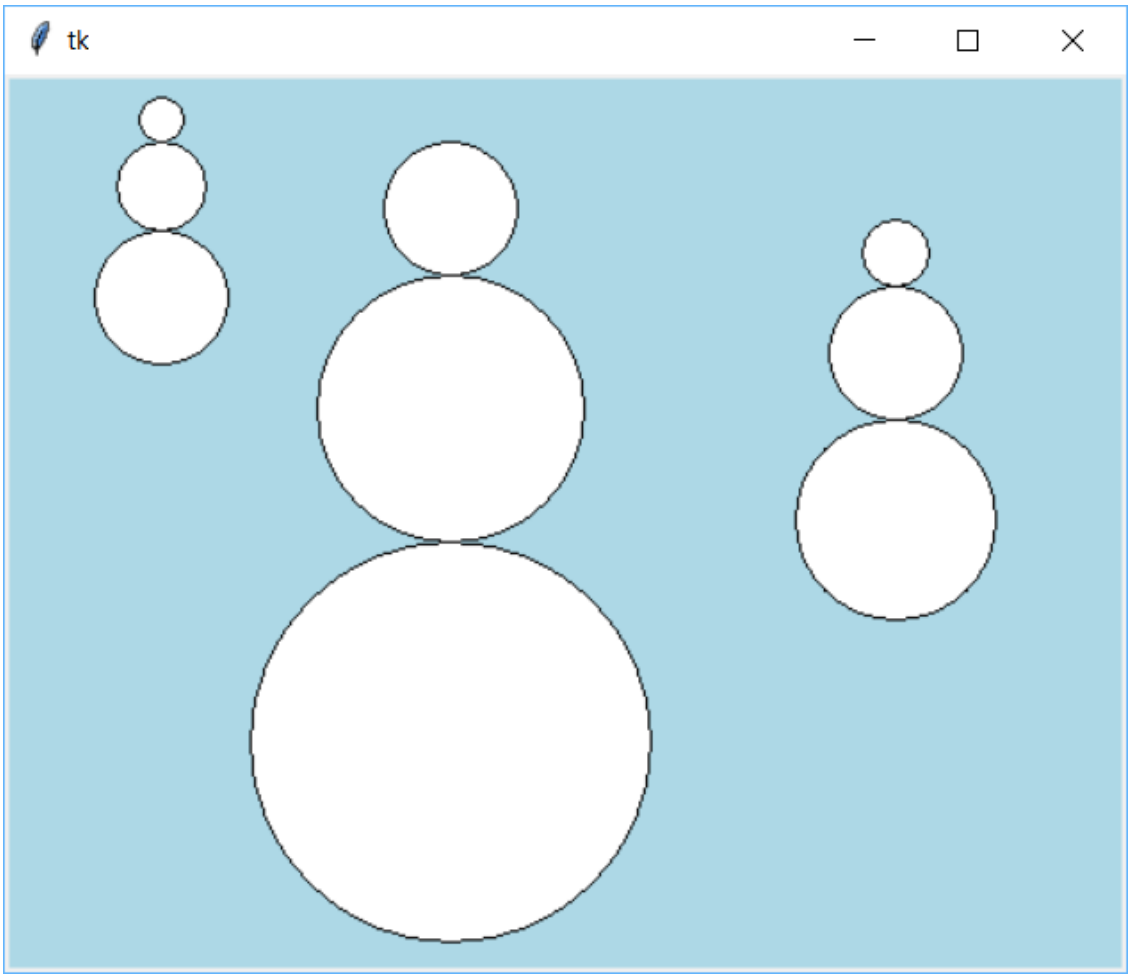
```
>>> pocet_samohlasok('python')
2
>>> pocet_samohlasok('strc prst skrz krk')
0
```

13. Napíšte funkcie `gula(x, y, r)` a `snehuliak(x, y, r)`. Funkcia `gula()` nakreslí biely kruh so stredom `(x, y)` a s polomerom `r`. Funkcia `snehuliak()` pomocou troch volaní `gula()` nakreslí snehuliaka, v ktorom spodná najväčšia guľa má stred `(x, y)` a polomer `r`. Stredná má polomer 2/3 veľkej a najmenšia je polovičná strednej. Otestujte na bledomodrom pozadí.

    - napr.

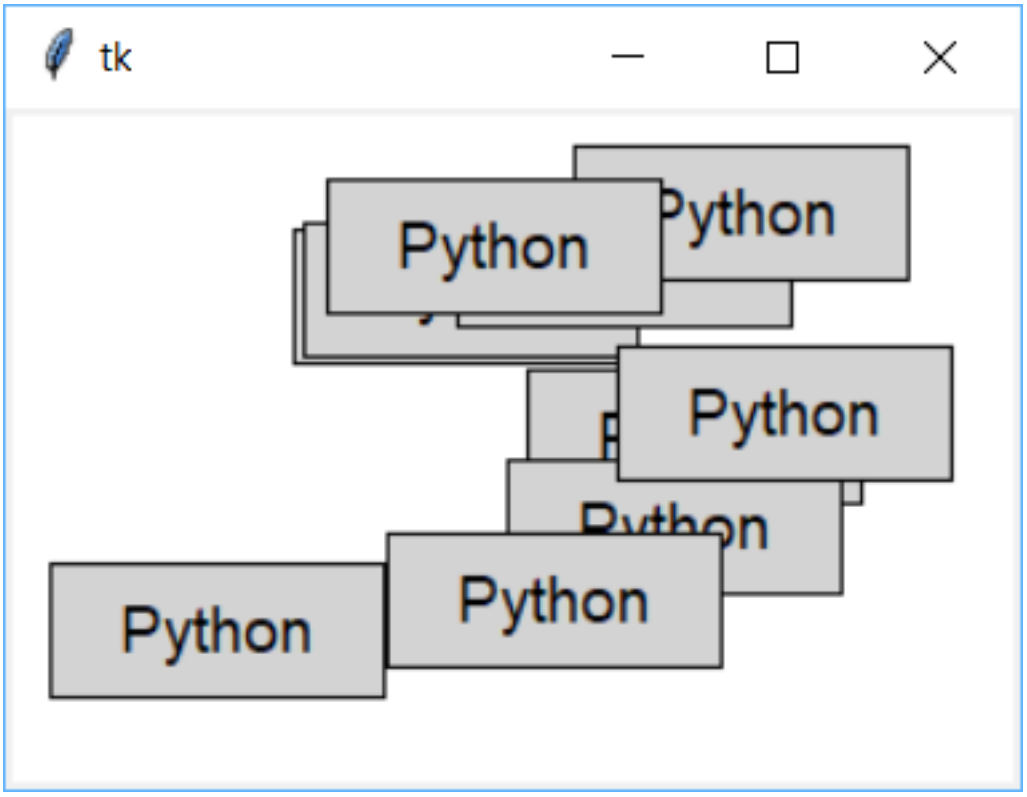
```
snehuliak(200, 300, 90)
snehuliak(400, 200, 45)
snehuliak(70, 100, 30)
```

    nakreslí:

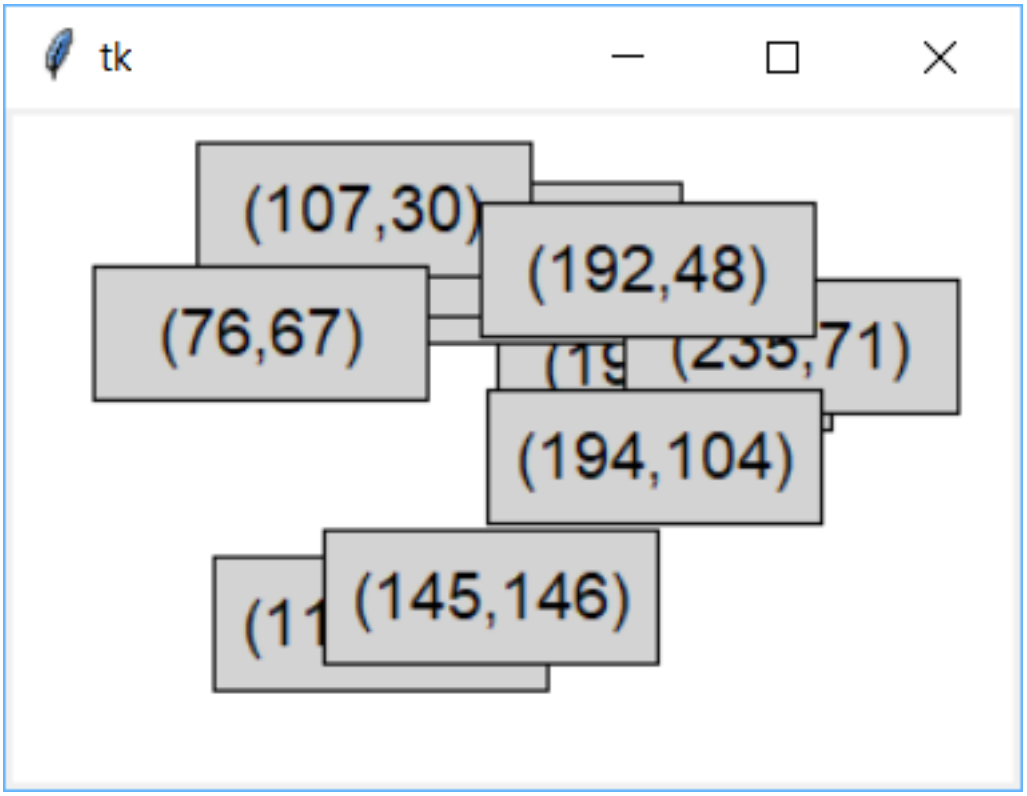


14. Napíšte funkciu `karticka(x, y, text)`, ktorá do grafickej plochy nakreslí bledošedý obdĺžnik a do jeho stredu vypíše zadaný text. Stred kartičky má súradnice `(x, y)` a jej strany majú dĺžky 100 a 40. Font písma nech je napr. `'arial 14'`.

    - otestujte náhodným vygenerovaním 10 kartičiek, napr. s textom `'Python'`, napr.



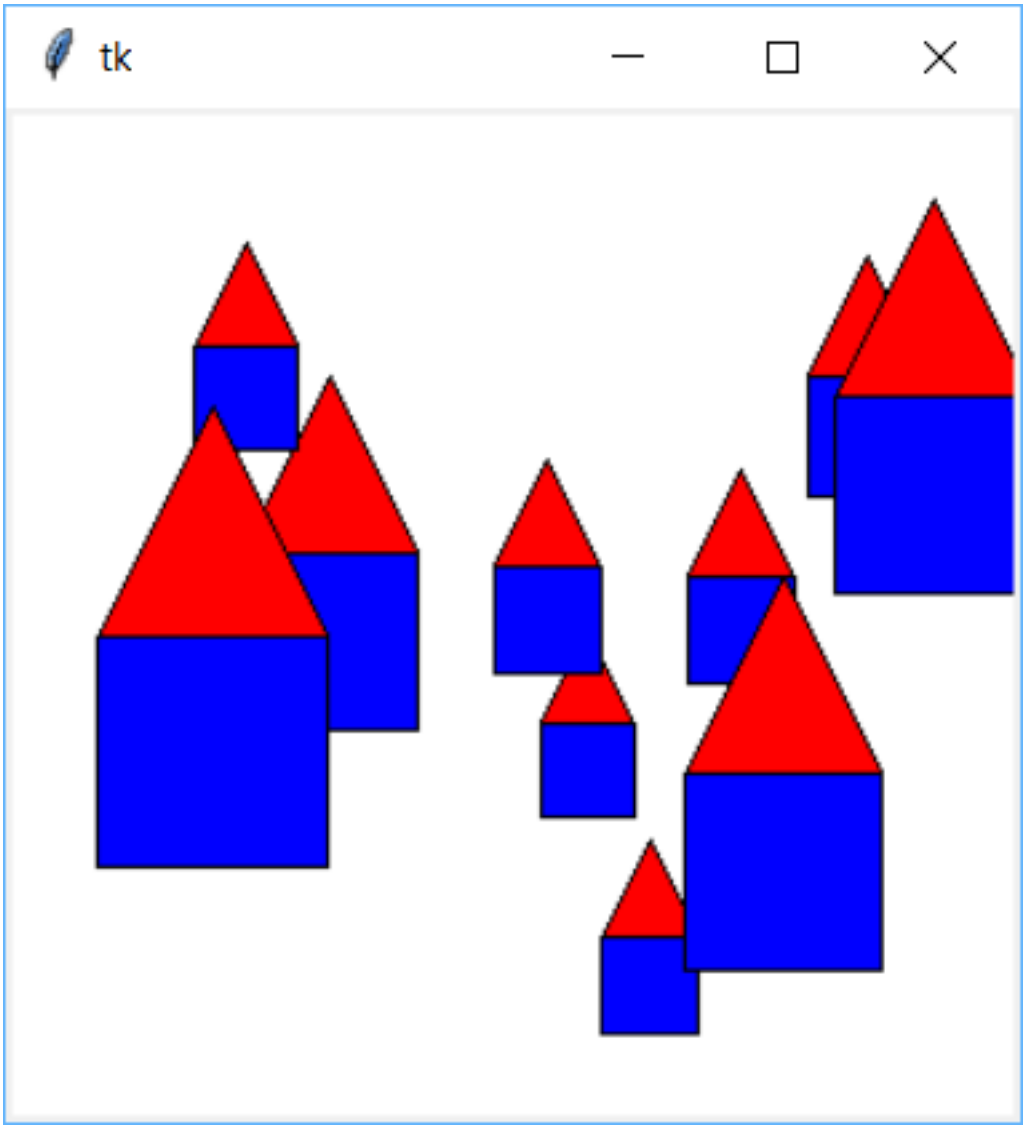
- otestujte náhodným vygenerovaním 10 kartičiek s textom súradníc kartičky, napr. v tvare `'(354,211)'`:



15. Napíšte funkcie `stvorec(x, y, a, farba)`, `trojuholnik(x, y, a, farba)` a `domcek(x, y, a=50, farba1='blue', farba2='red')`. Funkcia `stvorec()` nakreslí farebný štvorec s ľa-

vým horným vrcholom na `(x, y)` a stranou `a`. Funkcia `trojuholnik()` nakreslí rovnoramenný trojuholník s ľavým dolným vrcholom v `(x, y)` so stranou aj výškou `a`. Funkcia `domcek()` nakreslí domček pomocou štvorca a trojuholníka.

- otestujte vykreslením 10 domčekov na náhodných pozíciách, napr.



16. Napíšte funkcie `kruh(x, y, r)` a `sustredne(n, x, y)`. Funkcia `kruh()` nakreslí na zadané súradnice kruh daného polomeru a vyplní ho náhodnou farbou. Funkcia `sustredne()` pomocou `kruh()` nakreslí `n` sústredných farebných kruhov s polomermi 5, 10, 15, 20, ... Použite funkciu `nahodna_farba()` z prednášky.

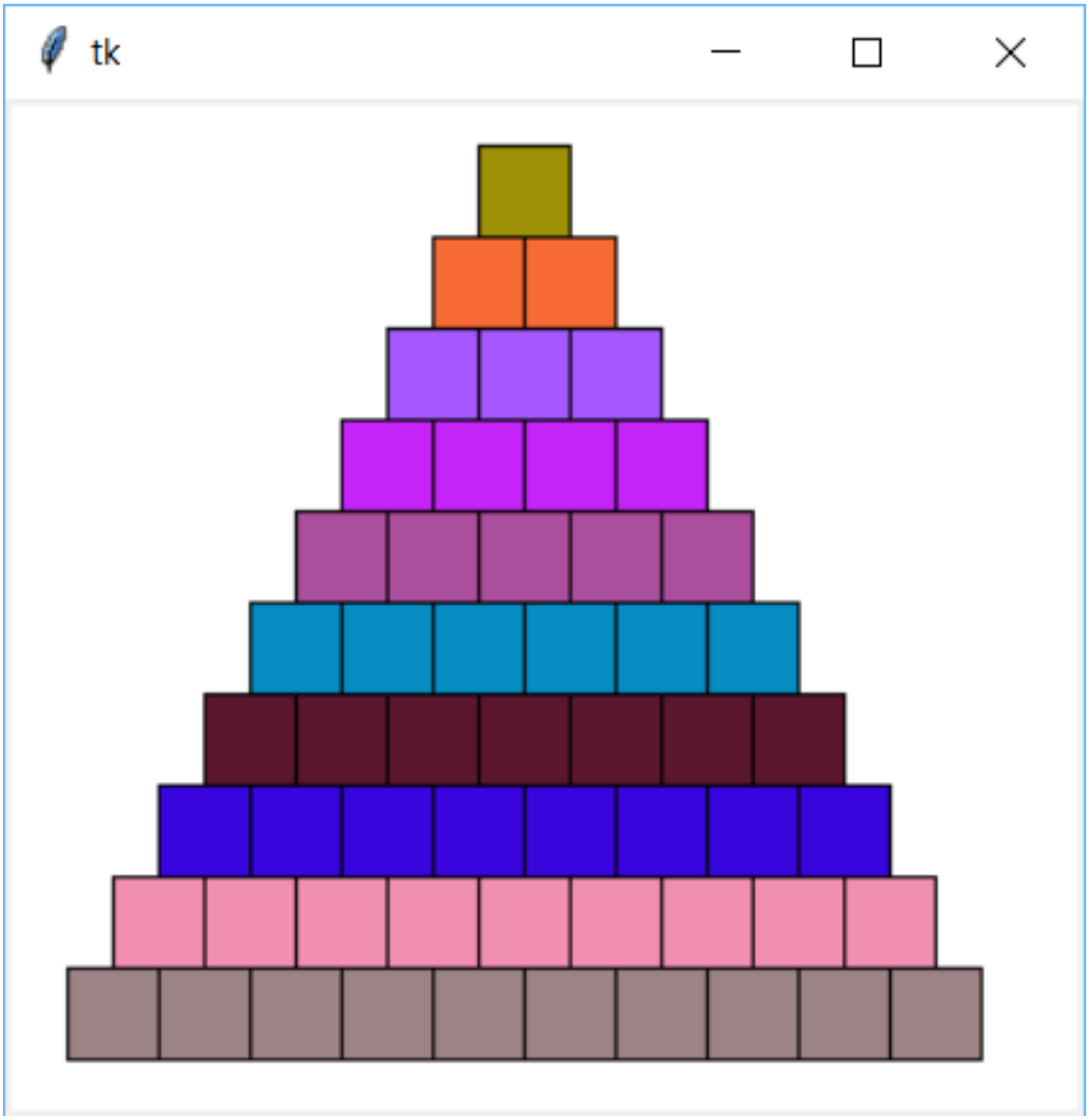
    - otestujte niekoľkými volaniami `sustredne()` na náhodných pozíciách

17. Napíšte funkcie: `kocka(x, y, a, farba)`, ktorá nakreslí farebný štvorec s daným stredom a danou stranou; funkcia `rad(n, x, y, a)` nakreslí vedľa seba `n` štvorcov (prvý ľavý z nich na zadaných súradniciach), pričom sú zafarbené rovnakou náhodnou farbou; funkcia `pyramida(n, x, y, a)` pomocou funkcie `rad()` nakreslí pyramídu výšky `n`, t.j. zloženú z `n` radov dĺžky 1, 2, 3, ... `n`. Najvyššia kocka pyramídy je na zadaných súradniciach. Každý nižší rad kociek sa nakreslí o `a` nižšie a o `a/2` odsunutý vľavo.

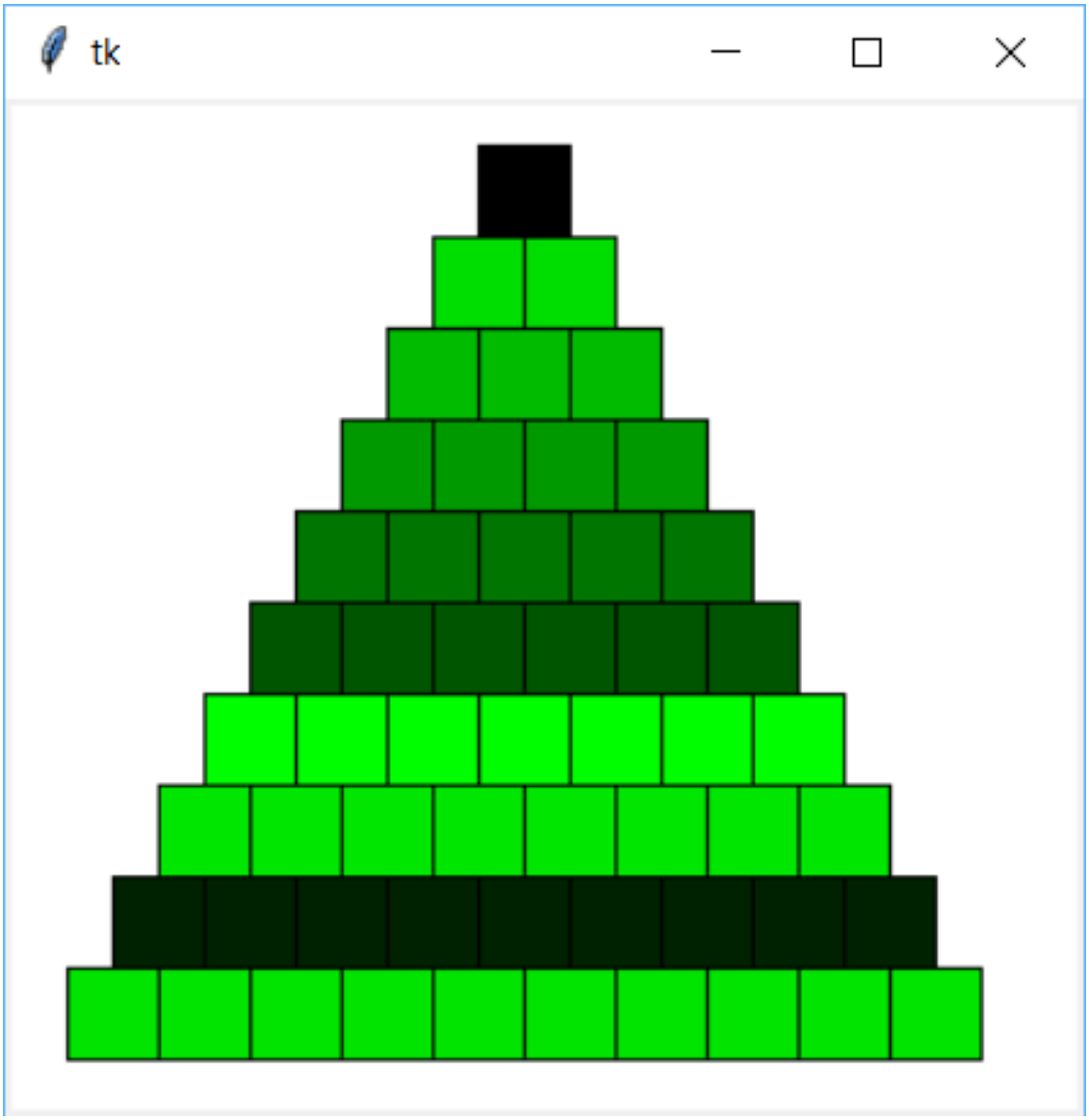
    - otestujte napr.

    ```
    pyramida(10, 170, 30, 30)
    ```

    dostaneme napr.



- skuste zmeniť generátor náhodnej farby tak, aby sa vytvárali náhodné odtiene len jednej farby, napr. `rgb(0, g, 0)`, kde `g` je náhodné číslo od 100 do 255 bude generovať len zelené farby, napr.


tk

KAPITOLA 6

6. Znakové reťazce

6.1 Typ string

Čo už vieme o znakových reťazcoch:

- reťazec je postupnosť znakov uzavretá v apostrofoch `''` alebo v úvodzovkách `""`
- vieme priradiť reťazec do premennej
- zreťaziť (zlepiť) dva reťazce
- násobiť (zlepiť viac kópií) reťazca
- načítať zo vstupu (pomocou `input()`) a vypisovať (pomocou `print()`)
- vyrobiť z čísla reťazec (`str()`), z reťazca číslo (`int()`, `float()`)
- rozobrať reťazec vo `for`-cykle

Postupne prejdeme tieto možnosti práce s reťazcami a doplníme ich o niektoré novinky.

Keďže znakový reťazec je postupnosť znakov uzavretá v apostrofoch `''` alebo v úvodzovkách `""`, platí:

- môže obsahovať ľubovoľné znaky (okrem znaku apostrof `'` v `''` reťazci, a znaku úvodzovka `"` v úvodzovkovom `""` reťazci)
- musí sa zmestiť do jedného riadka (nesmie prechádzať do druhého riadka)
- môže obsahovať špeciálne znaky (zapisujú sa dvomi znakmi, ale pritom v reťazci reprezentujú len jeden), vždy začínajú znakom `'\'` (opačná lomka):
  - `\n` - nový riadok
  - `\t` - tabulátor
  - `\'` - apostrof
  - `\"` - úvodzovka
  - `\\` - opačná lomka

Napríklad

```
>>> 'Monty\nPython'
'Monty\nPython'
>>> print('Monty\nPython')
Monty
Python
>>> print('Monty\\nPython')
Monty\nPython
```

6.1.1 Viacriadkové reťazce

platí:

- reťazec, ktorý začína trojicou buď apostrofov `'''` alebo úvodzoviek `"""` môže obsahovať aj `'` a `"`, môže prechádzať cez viac riadkov (automaticky sa sem doplní `\n`)
- musí byť ukončený rovnakou trojicou `'''` alebo `"""`

```
>>> macek = '''Išiel Macek
do Malacek
šošovičku mlácic'''
>>> macek
'Išiel Macek\ndo Malacek\nšošovičku mlácic'
>>> print(macek)
Išiel Macek
do Malacek
šošovičku mlácic
>>> '''tento retazec obsahuje " aj ' a funguje'''
'tento retazec obsahuje " aj \' a funguje'
>>> print('''tento retazec obsahuje " aj ' a funguje''')
tento retazec obsahuje " aj ' a funguje
```

6.1.2 Dĺžka reťazca

Štandardná funkcia `len()` vráti dĺžku reťazca (špeciálne znaky ako `'\n'`, `'\''`, a pod. reprezentujú len 1 znak):

```
>>> a = 'Python'
>>> len(a)
6
>>> len('Peter\'s dog')
11
>>> len('\\\\\\')
3
```

Túto funkciu už vieme naprogramovať aj sami, ale v porovnaní so štandardnou funkciou `len()` bude oveľa pomalšia:

```
def dlzka(retazec):
    pocet = 0
    for znak in retazec:
        pocet += 1
    return pocet
```

```
>>> dlzka('Python')
6
>>> a = 'x' * 100000000
>>> dlzka(x)
100000000
>>> len(x)
100000000
```

6.1.3 Operácia in

Aby sme zistili, či sa v reťazci nachádza nejaký konkrétny znak, doteraz sme to museli riešiť takto:

```
def zisti(znak, retazec):
    for z in retazec:
        if z == znak:
            return True
    return False
```

```
>>> zisti('y', 'Python')
True
>>> zisti('T', 'Python')
False
```

Pritom existuje binárna operácia `in`, ktorá zisťuje, či sa zadaný podreťazec nachádza v nejakom konkrétnom reťazci. Jej tvar je

```
podretazec in retazec
```

Najčastejšie sa bude využívať v príkaze `if` a v cykle `while`, napr.

```
>>> 'nt' in 'Monty Python'
True
>>> 'y P' in 'Monty Python'
True
>>> 'tyPy' in 'Monty Python'
False
>>> 'pyt' in 'Monty Python'
False
```

Na rozdiel od našej vlastnej funkcie `zisti()`, operácia `in` funguje nielen pre zisťovanie jedného znaku, ale aj pre ľubovoľne dlhý podreťazec.

Ak niekedy budeme potrebovať negáciu tejto podmienky, môžeme zapísať:

```
if not 'a' in retazec:
    ...
if 'a' not in retazec:
    ...
```

Pričom sa odporúča druhý spôsob zápisu `not in`.

6.1.4 Operácia indexovania [ ]

Pomocou tejto operácie vieme pristupovať k jednotlivým znakom postupnosti (znakový reťazec je postupnosť znakov). Jej tvar je

```
ret'azec[číslo]
```

Celému číslu v hranatých zátvorkách hovoríme **index**:

- znaky v reťazci sú indexované od `0` do `len()-1`, t.j. prvý znak v reťazci má index 0, druhý 1, ... posledný má index `len()-1`
- výsledkom indexovania je vždy 1-znakový reťazec (čo je nový reťazec s kópiou 1 znaku z pôvodného reťazca) alebo chybová správa, keď indexujeme mimo znaky reťazca

Očíslujme znaky reťazca:

Tabuľka 1: reťazec ‚Monty Python‘

M	o	n	t	y		P	y	t	h	o	n	
0	1	2	3	4	5	6	7	8	9	10	11	

Napr. do premennej `abc` priradíme reťazec 12 znakov a pristupujeme ku niektorým znakom pomocou indexu:

```
>>> abc = 'Monty Python'
>>> abc[3]
't'
>>> abc[9]
'h'
>>> abc[12]
...
IndexError: string index out of range
>>> abc[len(abc)-1]
'n'
```

Vidíme, že posledný znak v reťazci má index **dĺžka reťazca-1**. Ak indexujeme väčším číslom ako 11, vyvolá sa chybová správa **IndexError: string index out of range**.

Často sa indexuje v cykle, kde premenná cyklu nadobúda správneho správne hodnoty indexov, napr.

```
>>> a = 'Python'
>>> for i in range(len(a)):
        print(i, a[i])

0 P
1 y
2 t
3 h
4 o
5 n
```

Funkcia `range(len(a))` zabezpečí, že cyklus prejde postupne pre všetky `i` od `0` do `len(a)-1`.

6.1.5 Indexovanie so zápornými indexmi

Keďže často potrebujeme pristupovať ku znakom na konci reťazca, môžeme to zapisovať pomocou záporných indexov:

```
abc[-5] == abc[len(abc)-5]
```

Znaky reťazca sú indexované od `-1` do `-len()` takto:

Tabuľka 2: reťazec ‚Monty Python‘

M	o	n	t	y		P	y	t	h	o	n	
0	1	2	3	4	5	6	7	8	9	10	11	
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	

Napríklad:

```
>>> abc = 'Monty Python'
>>> abc[len(abc)-1]
'n'
>>> abc[-1]
'n'
>>> abc[-7]
' '
>>> abc[-13]
...
IndexError: string index out of range
```

alebo aj for-cyklom:

```
>>> a = 'Python'
>>> for i in range(1, len(a)+1):
        print(-i, a[-i])

-1 n
-2 o
-3 h
-4 t
-5 y
-6 P
```

alebo for-cyklom so záporným krokom:

```
>>> a = 'Python'
>>> for i in range(-1, -len(a)-1, -1):
        print(i, a[i])

-1 n
-2 o
-3 h
-4 t
-5 y
-6 P
```

6.1.6 Podreťazce

Indexovať môžeme nielen jeden znak, ale aj nejaký podreťazec celého reťazca. Opäť použijeme operátor indexovania, ale index bude obsahovať znak `':'`:

```
retˊazec[prvý : posledný]
```

kde

- `prvý` je index začiatku podreťazca

- `posledný` je index prvku **jeden za**, t.j. musíme písať index prvku o 1 viac
- takejto operácii hovoríme **rez** (alebo po anglicky **slice**)
- ak takto indexujeme mimo reťazec, nenastane chyba, ale prvky mimo sú prázdny reťazec

Ak indexujeme rez od 6. po 11. prvok:

Tabuľka 3: reťazec ‚Monty Python‘

M	o	n	t	y		P	y	t	h	o	n	
						^					^	
0	1	2	3	4	5	6	7	8	9	10	11	

prvok s indexom 11 už vo výsledku nebude:

```
>>> abc = 'Monty Python'
>>> abc[6:11]
'Pytho'
>>> abc[6:12]
'Python'
>>> abc[6:len(abc)]
'Python'
>>> abc[6:12]
'Python'
>>> abc[10:16]
'on'
```

Podreťazce môžeme vytvárať aj v cykle:

```
>>> a = 'Python'
>>> for i in range(len(a)):
        print(f'{i}:{i+3} {a[i:i+3]}')

0:3 Pyt
1:4 yth
2:5 tho
3:6 hon
4:7 on
5:8 n
```

alebo

```
>>> a = 'Python'
>>> for i in range(len(a)):
        print(f'{i}:{len(a)} {a[i:len(a)]}')

0:6 Python
1:6 ython
2:6 thon
3:6 hon
4:6 on
5:6 n
```

6.1.7 Predvolená hodnota

Ak neuvedieme prvý index v podreťazci, bude to označovať rez **od začiatku reťazca**. Zápis je takýto:

```
ret'azec[ : posledný]
```

Ak neuvedieme druhý index v podreťazci, označuje to, že chceme rez **až do konca reťazca**. Teda:

```
ret'azec[prvý : ]
```

Ak neuvedieme ani jeden index v podreťazci, označuje to, že chceme **celý reťazec**, t.j. vytvorí sa kópia pôvodného reťazca:

```
ret'azec[ : ]
```

Tabuľka 4: reťazec ‚Monty Python‘

M	o	n	t	y		P	y	t	h	o	n	
0	1	2	3	4	5	6	7	8	9	10	11	
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	

napríklad

```
>>> abc = 'Monty Python'
>>> abc[6:]                  # od 6. znaku do konca
'Python'
>>> abc[:5]                  # od zaciatku po 4. znak
'Monty'
>>> abc[-4:]                 # od 4. od konca az do konca
'thon'
>>> abc[16:]                 # indexujeme mimo retazca
''
```

6.1.8 Podreťazce s krokom

Podobne ako vo funkcii `range()` aj pri indexoch podreťazca môžeme určiť aj krok indexov:

```
ret'azec[prvý : posledný : krok]
```

kde `krok` určuje o koľko sa bude index v reťazci posúvať od `prvý` po `posledný`. Napríklad:

```
>>> abc = 'Monty Python'
>>> abc[2:10:2]
'nyPt'
>>> abc[::3]
'MtPh'
>>> abc[9:-7:-1]
'htyP'
>>> abc[::-1]
'nohtyP ytnoM'
>>> abc[6:] + ' ' + abc[:5]
'Python Monty'
>>> abc[4::-1] + ' ' + abc[:5:-1]
'ytnoM nohtyP'
>>> (abc[6:] + ' ' + abc[:5])[::-1]
'ytnoM nohtyP'
>>> 'kobyla ma maly bok'[::-1]
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
'kob ylam am alybok'
>>> abc[4:9]
'y Pyt'
>>> abc[4:9][2]          # aj podretazce mozeme dalej indexovat
'P'
>>> abc[4:9][2:4]
'Py'
>>> abc[4:9][::-1]
'tyP y'
```

6.1.9 Reťazce sú v pamäti nemenné (nemeniteľné)

Typ `str`, t.j. znakové reťazce, je nemeniteľný typ (**immutable**). To znamená, že hodnota reťazca sa v pamäti zmeniť nedá. Ak budeme potrebovať reťazec, v ktorom je nejaká zmena, budeme musieť skonštruovať nový. Napr.

```
>>> abc[6] = 'K'
TypeError: 'str' object does not support item assignment
```

Všetky doterajšie manipulácie s reťazcami nemenili reťazec, ale zakaždým vytvárali úplne nový (niekedy to bola len kópia pôvodného), napr.

```
>>> cba = abc[::-1]
>>> abc
'Monty Python'
>>> cba
'nohtyP ytnoM'
```

Takže, keď chceme v reťazci zmeniť nejaký znak, budeme musieť skonštruovať nový reťazec, napr. takto:

```
>>> abc[6] = 'K'
...
TypeError: 'str' object does not support item assignment
>>> novy = abc[:6] + 'K' + abc[7:]
>>> novy
'Monty Kython'
>>> abc
'Monty Python'
```

Alebo, ak chceme opraviť prvý aj posledný znak:

```
>>> abc = 'm' + abc[1:-1] + 'N'
>>> abc
'monty PythoN'
```

6.1.10 Porovnávanie jednoznakových reťazcov

Jednoznakové reťazce môžeme porovnávať relačnými operátormi ==, !=, <, <=, >, >=, napr.

```
>>> 'x' == 'x'
True
>>> 'm' != 'M'
True
>>> 'a' > 'm'
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
False
>>> 'a' > 'A'
True
```

Python na porovnávanie používa vnútornú reprezentáciu **Unicode (UTF-8)**. S touto reprezentáciou môžeme pracovať pomocou funkcií `ord()` a `chr()`:

- funkcia `ord(znak)` vráti vnútornú reprezentáciu znaku (kódovanie v pamäti počítača)

```
>>> ord('a')
97
>>> ord('A')
65
```

- opačná funkcia `chr(číslo)` vráti jednoznakový reťazec, pre ktorý má tento znak danú číselnú reprezentáciu

```
>>> chr(66)
'B'
>>> chr(244)
'ô'
```

Pri porovnávaní dvoch znakov sa porovnávajú ich vnútorné reprezentácie, t.j.

```
>>> ord('a') > ord('A')
True
>>> 97 > 65
True
>>> 'a' > 'A'
True
```

Vnútornú reprezentáciu niektorých znakov môžeme zistiť napr. pomocou for-cyklu:

```
>>> for i in range(ord('A'), ord('J')):
        print(i, chr(i))

65 A
66 B
67 C
68 D
69 E
70 F
71 G
72 H
73 I
```

6.1.11 Porovnávanie dlhších reťazcov

Dlhšie reťazce Python porovnáva postupne po znakoch:

- kým sú v oboch reťazcoch rovnaké znaky, preskakuje ich
- pri prvom rôznom znaku, porovná tieto dva znaky

Napr. pri porovnávaní dvoch reťazcov ‚kocur' a ‚kohut':

- porovná 0. znaky: `'k' == 'k'`
- porovná 1. znaky: `'o' == 'o'`

- porovná 2. znaky: `'c' < 'h'` a tu aj skončí porovnávanie týchto reťazcov

Preto platí, že `'kocur' < 'kohut'`. Treba si dávať pozor **na znaky s diakritikou**, lebo, napr. `ord('č') = 269 > ord('h') = 104`. Napr.

```
>>> 'kocúr' < 'kohút'
True
>>> 'kočka' < 'kohut
False
>>> 'PYTHON' < 'Python' < 'python'
True
```

6.1.12 Prechádzanie reťazca v cykle

Už sme videli, že prvky znakového reťazca môžeme prechádzať for-cyklom, v ktorom indexujeme celý reťazec postupne od ‚‘,0‘,‘ do ‚‘,len()-1‘,‘:

```
>>> a = 'Python'
>>> for i in range(len(a)):
        print('.' * i, a[i])

 P
. y
.. t
... h
.... o
..... n
```

Tiež vieme, že for-cyklom môžeme prechádzať nielen postupnosť indexov (t.j. `range(len(a))`), ale priamo postupnosť znakov, napr.

```
>>> for znak in 'python':
        print(znak * 5)

ppppp
yyyyy
ttttt
hhhhh
ooooo
nnnnn
```

Zrejme reťazec vieme prechádzať aj while-cyklom, napr.

```
>>> a = '.....vel'a bodiek'
>>> print(a)
.....vel'a bodiek
>>> while len(a) != 0 and a[0] == '.':
        a = a[1:]

>>> print(a)
vel'a bodiek
```

Cyklus sa opakoval, kým bol reťazec neprázdny a kým boli na začiatku reťazca znaky bodky `'.'`. Vtedy sa v tele cyklu reťazec skracoval o prvý znak.

6.2 Reťazcové funkcie

Už poznáme tieto štandardné funkcie:

- `len()` - dĺžka reťazca
- `int()` - prevod reťazca na celé číslo
- `float()` - prevod reťazca na desatinné číslo
- `str()` - prevod čísla (aj ľubovoľnej inej hodnoty) na reťazec
- `ord()`, `chr()` - prevod do a z **Unicode**

Okrem nich existujú ešte aj tieto tri užitočné štandardné funkcie:

- `bin()` - prevod celého čísla do reťazca, ktorý reprezentuje toto číslo v dvojkovej sústave
- `hex()` prevod celého čísla do reťazca, ktorý reprezentuje toto číslo v šestnástkovej sústave
- `oct()` - prevod celého čísla do reťazca, ktorý reprezentuje toto číslo v osmičkovej sústave

Napríklad

```
>>> bin(123)
'0b1111011'
>>> hex(123)
'0x7b'
>>> oct(123)
'0o173'
```

Zápisy celého čísla v niektorej z týchto sústav fungujú ako celočíselné konštanty:

```
>>> 0b1111011
123
>>> 0x7b
123
>>> 0o173
123
```

6.2.1 Vlastné funkcie

Môžeme vytvárať vlastné funkcie, ktoré majú aj reťazcové parametre, resp. môžu vracať reťazcovú návratovú hodnotu. Niekoľko námetov:

- funkcia vráti `True` ak je daný znak (jednoznakový reťazec) číslicou:

  ```
  def je_cifra(znak):
      return '0' <= znak <= '9'
  ```

  alebo inak

  ```
  def je_cifra(znak):
      return znak in '0123456789'
  ```

- funkcia vráti `True` ak je daný znak (jednoznakový reťazec) malé alebo veľké písmeno (anglickej abecedy)

  ```
  def je_pismeno(znak):
      return 'a' <= znak <= 'z' or 'A' <= znak <= 'Z'
  ```

- parametrom funkcie je reťazec s menom a priezviskom (oddelené sú práve jednou medzerou) - funkcia vráti reťazec, v ktorom bude najprv priezvisko a až za tým meno (oddelené medzerou)

```
def meno(r):
    ix = 0
    while ix < len(r) and r[ix] != ' ':      # najde medzeru
        ix += 1
    return r[ix+1:] + ' ' + r[:ix]
```

- funkcia vráti prvé slovo vo vete, ktoré obsahuje len malé a veľké písmená (využijeme funkciu `je_pismeno`)

```
def slovo(veta):
    for i in range(len(veta)):
        if not je_pismeno(veta[i]):
            return veta[:i]
    return veta
```

6.2.2 Reťazcové metódy

Je to špeciálny spôsob zápisu volania funkcie (bodková notácia):

```
ret'azec.metóda(parametre)
```

kde `metóda` je meno niektorej z metód, ktoré sú v systéme už definované pre znakové reťazce. My si ukážeme niekoľko užitočných metód, s niektorými ďalšími sa zoznámime neskôr:

- `ret'azec.count(podret'azec)` - zistí počet výskytov podreťazca v reťazci
- `ret'azec.find(podret'azec)` - zistí index prvého výskytu podreťazca v reťazci
- `ret'azec.lower()` - vráti reťazec, v ktorom prevedie všetky písmená na malé
- `retazec.upper()` - vráti reťazec, v ktorom prevedie všetky písmená na veľké
- `ret'azec.replace(podret'azec1, podret'azec2)` - vráti reťazec, v ktorom nahradí všetky výskyty `podret'azec1` iným reťazcom `podret'azec2`
- `ret'azec.strip()` - vráti reťazec, v ktorom odstráni medzery na začiatku a na konci reťazca (odfiltruje pritom aj iné oddeľovacie znaky ako `'\n'` a `'\t'`)
- `ret'azec.format(hodnoty)` - vráti reťazec, v ktorom nahradí formátovacie prvky `'{}'` zadanými hodnotami

Ak chceme o niektorej z metód získať **help**, môžeme zadať, napr.

```
>>> help(''.find)
Help on built-in function find:

find(...) method of builtins.str instance
    S.find(sub[, start[, end]]) -> int

    ...
```

metóda `ret'azec.count()`

`ret'azec.`**`count`**`(`*`podret'azec`*`)`

Parametre

- **ret'azec** – reťazec, v ktorom sa budú hľadať všetky výskyty nejakého zadaného podreťazca
- **podret'azec** – hľadaný podreťazec

Metóda zistí počet všetkých výskytov podreťazca v danom reťazci. Napr.

```
>>> 'Python'.count('th')
1          # ret'azec 'th' sa nachádza v 'Python' iba raz
>>> 'Python'.count('to')
0          # ret'azec 'to' sa v 'Python' nenachádza ani raz
>>> 'Pyp ypY Ypy yPy yPY'.count('Py')
2          # ret'azec 'Py' sa tu nachádza na 2 miestach
```

metóda ret'azec.find()

ret'azec.**find**(*podret'azec*)

Parametre

- **ret'azec** – reťazec, v ktorom sa budú hľadať prvý výskyt nejakého zadaného podreťazca
- **podret'azec** – hľadaný podreťazec

Metóda nájde prvý najľavejší výskyt podreťazca v danom reťazci. Napr.

```
>>> 'Python'.find('th')
2                          # retazec 'th' sa nachadza v 'Python' od 2. indexu
>>> 'Python'.find('to')
-1                         # retazec 'to' sa v 'Python' nenachadza
>>> 'abcd ce abced'.find('c')
5                          # prvy vyskyt retazca 'ce' je na indexe 5
```

metóda ret'azec.lower()

ret'azec.**lower**()

Parametre **ret'azec** – reťazec, z ktorého sa vyrobí nový ale s malými písmenami

Metóda vyrobí kópiu daného reťazca, v ktorej všetky veľké písmená prerobí na malé. Nepísmenové znaky nemení. Napr.

```
>>> 'PyTHon'.lower()
'python'
>>> '1+2'.lower()
'1+2'
```

6.2.3 Formátovanie reťazca

Možnosti formátovania pomocou formátovacích reťazcov `f'{x}'` ale aj pomocou metódy `format()` sme už videli predtým. Teraz ukážeme niekoľko užitočných formátovacích prvkov. Volania majú tvar:

```
f'formátovací ret'azec s hodnotami v {}'
'formátovací ret'azec'.format(parametre)
```

Tieto dva varianty sú skoro identické a je na vás, na ktorý z nich si zvyknete lepšie. Tvar `f'{x}'` funguje v Pythone až od verzie **3.6**, preto sa v starších učebných textoch nevyskytuje.

Formátovací reťazec v oboch prípadoch obsahuje formátovacie prvky, ktoré sa nachádzajú v `'{}'` zátvorkách. Vo variante `f'{x}'` je v týchto zátvorkách priamo vypisovaná hodnota a za ňou sa môže nachádzať špecifikácia oddelená znakom `':'`. Vo variante metódy `format` môže byť v `'{}'` zátvorkách len špecifikácia a vypisované hodnoty sa nachádzajú ako parametre `format`. Napr. tieto dva zápisy dávajú rovnaký výsledok:

```
>>> x = 1237
>>> a = f'vysledok {x+8:5}'
>>> b = 'vysledok {:5}'.format(x+8)
```

alebo

```
>>> farba1 = f'#{r:02x}{g:02x}{b:02x}'
>>> farba2 = '#{:02x}{:02x}{:02x}'.format(r, g, b)
```

Zrejme pri volaní metódy `format()` musí sedieť počet formátovacích prvkov `'{}'` s počtom parametrov funkcie. Metóda `format()` potom dosadí hodnoty svojich parametrov za zodpovedajúce dvojice `'{}'`.

Špecifikácia formátu

V zátvorkách `'{}'` sa môžu nachádzať rôzne upresnenia formátovania, napr.:

- `'{:10}'` - šírka výpisu 10 znakov
- `'{:>7}'` - šírka 7, zarovnané vpravo
- `'{:<5d}'` - šírka 5, zarovnané vľavo, parameter musí byť celé číslo (bude sa vypisovať v 10-ovej sústave)
- `'{:12.4f}'` - šírka 12, parameter desatinné číslo vypisované na 4 desatinné miesta
- `'{:06x}'` - šírka 6, zľava doplnená nulami, parameter celé číslo sa vypíše v 16-ovej sústave
- `'{:^20s}'` - šírka 20, vycentrované, parametrom je reťazec

Zhrňme najpoužívanejšie písmená pri označovaní typu parametra:

- `d` - celé číslo v desiatkovej sústave
- `b` - celé číslo v dvojkovej sústave
- `x` - celé číslo v šestnástkovej sústave
- `s` - znakový reťazec
- `f` - desatinné číslo (možno špecifikovať počet desatinných miest, inak default 6)
- `g` - desatinné číslo vo všeobecnom formáte

6.2.4 Dokumentačný reťazec pri definovaní funkcie

Ak funkcia vo svojom tele hneď ako **prvý riadok** obsahuje znakový reťazec (zvykne byť viacriadkový s `'''`), tento sa stáva, tzv. **dokumentačným reťazcom** (**docstring**). Pri vykonávaní tela funkcie sa takéto reťazce ignorujú (preskakujú). Tento reťazec (docstring) sa ale môže neskôr vypísať, napr. štandardnou funkciou `help()`.

Zadefinujme reťazcovú funkciu a hneď do nej dopíšeme aj niektoré základné informácie:

```
def pocet_vyskytov(podretazec, retazec):
    '''funkcia vráti počet výskytov podret'azca v ret'azci

    prvý parameter podretazec - l'ubovol'ný neprázdny ret'azec, o ktorom sa
                                bude zist'ovat' počet výskytov
    druhý parameter retazec - ret'azec, v ktorom sa hl'adajú výskyty

    ak je prvý parameter podretazec prázdny ret'azec, funkcia vráti len(retazec)
    '''
    pocet = 0
    for ix in range(len(retazec)):
        if retazec[ix:ix+len(podretazec)] == podretazec:
            pocet += 1
    return pocet
```

Takto definovaná funkcia funguje rovnako, ako keby žiaden dokumentačný reťazec neobsahovala, ale teraz bude fungovať aj:

```
>>> help(pocet_vyskytov)
Help on function pocet_vyskytov in module __main__:

pocet_vyskytov(podretazec, retazec)
    funkcia vráti počet výskytov podret'azca v ret'azci

    prvý parameter podretazec - l'ubovol'ný neprázdny ret'azec, o ktorom sa
                                bude zist'ovat' počet výskytov
    druhý parameter retazec - ret'azec, v ktorom sa hl'adajú výskyty

    ak je prvý parameter podretazec prázdny ret'azec, funkcia vráti len(retazec)
```

Tu môžeme vidieť užitočnú vlastnosť Pythonu: programátor, ktorý vytvára nejaké nové funkcie, môže hneď vytvárať aj malú dokumentáciu o jej používaní pre ďalších programátorov. Asi ľahko uhádneme, ako funguje napr. aj toto:

```
>>> help(hex)
Help on built-in function hex in module builtins:

hex(number, /)
    Return the hexadecimal representation of an integer.

    >>> hex(12648430)
    '0xc0ffee'
```

Pri takomto spôsobe samodokumentácie funkcií si treba uvedomiť, že Python v tele funkcie ignoruje nielen všetky reťazce, ale aj iné konštanty:

- ak napr. zavoláme funkciu, ktorá vracia nejakú hodnotu a túto hodnotu ďalej nespracujeme (napr. priradením do premennej, použitím ako parametra inej funkcie, ...), vyhodnocovanie funkcie takúto návratovú hodnotu ignoruje
- ak si uvedomíme, že meno funkcie bez okrúhlych zátvoriek nespôsobí volanie tejto funkcie, ale len hodnotu referencie na funkciu, tak aj takýto zápis sa ignoruje

Napr. všetky tieto zápisy sa v tele funkcie (alebo aj v programovom režime mimo funkcie) ignorujú:

```
s.replace('a', 'b')
print
g.pack
pocet + 1
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
i == i + 1
math.sin(uhol)
```

Python pri nich nehlási ani žiadnu chybu.

6.2.5 Príklad s kreslením a reťazcami

Navrhnime malú aplikáciu, v ktorej budeme pohybovať mysleným perom. Toto pero sa bude hýbať v jednom zo štyroch smerov: `'s'` pre sever, `'v'` pre východ, `'j'` pre juh, `'z'` pre západ. Dĺžka kroku pera nech je nejaká malá konštanta, napr. 10:

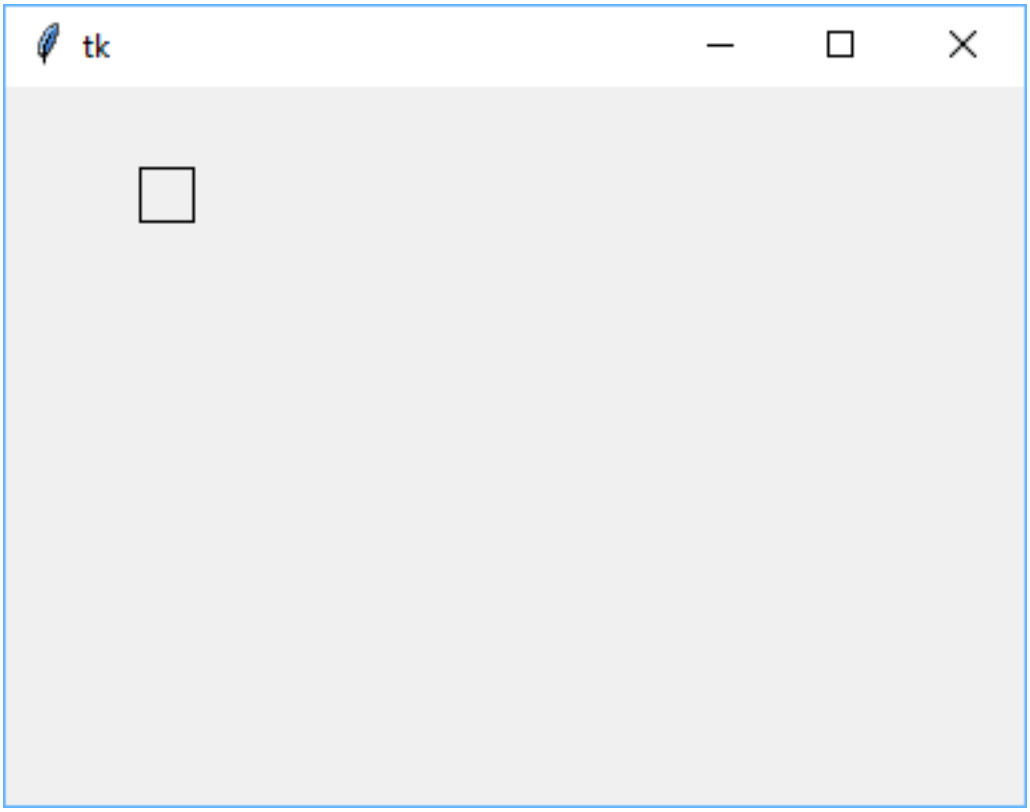
```
import tkinter

def kresli(retazec):
    x, y = 50, 50
    for znak in retazec:
        x1, y1 = x, y
        if znak == 's':
            y1 -= 10
        elif znak == 'v':
            x1 += 10
        elif znak == 'j':
            y1 += 10
        elif znak == 'z':
            x1 -= 10
        else:
            print('nerozumiem "' + znak + '"')
            return
        canvas.create_line(x, y, x1, y1)
        x, y = x1, y1

canvas = tkinter.Canvas()
canvas.pack()

kresli('ssvvjjzz')
```

Po spustení dostaneme:



Zrejme rôzne reťazce znakov, ktoré obsahujú len naše štyri písmená pre smery pohybu, budú kresliť rôzne útvary. Napr.

```
kresli('vvvvvvvjjjjjjjzzzzzzzsssssss')
```

nakreslí trochu väčší štvorec. Toto vieme zapísať napr. aj takto:

```
kresli('v'*7 + 'j'*7 + 'z'*7 + 's'*7)
```

Alebo

```
def stvorec(n):
    return 'v'*n + 'j'*n + 'z'*n + 's'*n

kresli(stvorec(7))
```

Na cvičeniach budeme rôzne vylepšovať túto ideu funkcie `kresli()`

6.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/
- v riešeniach úloh používajte len konštrukcie a funkcie, ktoré sme sa učili na doterajších prednáškach

1. Ručne zistite, čo sa vypíše:

- najprv bez počítača

```
>>> x, y = 'Bratislava', 'Praha'
>>> y[1] + x[4] + y[3] + x[-4] + y[-5]
...
>>> x[5:8] + 3 * x[3] + y[2:]
...
>>> y[:2] + x[-2:]
...
>>> x[1::2] + y[2::2] + x[2::3]
...
>>> x.replace('a', 'e') + y.replace('a', 'i')
...
>>> (y + x).replace('ra', '').replace('a', 'xa')
...
```

- potom to skontrolujte pomocou Pythonu

2. Napíšte funkciu `porovnaj(slovo1, slovo2)`, ktorá porovná dva dané reťazce a vypíše jednu z týchto možností: `slovo1 < slovo2`, `slovo1 == slovo2`, `slovo1 > slovo2`.

   - napr.

```
>>> porovnaj('abc', 'def')
abc < def
>>> porovnaj('2', '11')
2 > 11
>>> porovnaj('xxx', 'x'*3)
xxx == xxx
```

3. Napíšte funkciu `sucet(retazec)`, ktorá dostáva znakový reťazec s dvomi celými číslami oddelenými jednou medzerou. Funkcia vráti (nič nevypisuje) celé číslo, ktoré je súčtom dvoch čísel v reťazci.

   - napr.

```
>>> sucet('12 9')
21
>>> sucet('987654321 99999')
987754320
```

4. Napíšte funkciu `vymen_slova(retazec)`, ktorá dostáva znakový reťazec s tromi slovami. Tieto slová sú navzájom oddelené znakom `','`. Funkcia nič nevypisuje ale vráti nový znakový reťazec, v ktorom sú prvé a posledné slová navzájom vymenené.

   - napr.

```
>>> vymen_slova('prve,stredne,posledne')
'posledne,stredne,prve'
>>> vymen_slova(' prve slovo ,  stredne  ,   posledne   ')
'   posledne   ,  stredne  , prve slovo '
```

5. Napíšte funkciu `rozdel_na_slova(veta)`, ktorá zo zadaného znakového reťazca vypíše všetky slová. Predpokladáme, že v tomto reťazci sú slová oddelené po jednej medzere, kde slovo je postupnosť znakov rôzna od medzery.

   - napr.

```
>>> rozdel_na_slova('isiel Macek do Malaciek')
isiel
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
Macek
do
Malaciek
>>> rozdel_na_slova('Juraj_Janosik')
Juraj_Janosik
```

6. Napíšte funkciu `riadky(retazec)`, ktorá vypíše daný viacriadkový reťazec, ale pritom každý riadok očísluje číslami od 1 do počet riadkov.

   - napr.

```
>>> riadky('prvy riadok\n\ntreti je posledny')
1. prvy
2.
3. treti je posledny
>>> riadky('len \\n jeden riadok')
1. len \n jeden riadok
```

7. Napíšte funkciu `vypis(retazec)`, ktorá postupne pre každý znak reťazca vypíše tieto štyri hodnoty: poradové číslo v reťazci, samotný znak, jeho Unicode kódovanie a znak, ktorý má o jedna vyššie kódovanie (je v Unicode tabuľke za ním).

   - napr.

```
>>> vypis('Monty Python')
0 M 77 N
1 o 111 p
2 n 110 o
3 t 116 u
4 y 121 z
5   32 !
6 P 80 Q
7 y 121 z
8 t 116 u
9 h 104 i
10 o 111 p
11 n 110 o
>>> vypis('0123456789')
0 0 48 1
1 1 49 2
2 2 50 3
3 3 51 4
4 4 52 5
5 5 53 6
6 6 54 7
7 7 55 8
8 8 56 9
9 9 57 :
```

8. Metóda `'ret'azec'.count(podret'azec)` zistí počet výskytov podreťazca v reťazci. Napíšte funkciu `pocet(retazec, podretazec)`, ktorá robí to isté, ale bez použitia tejto metódy.

   - napr.

```
>>> pocet('mama ma emu a ema ma mamu', 'ma ')
4
>>> pocet('mama ma emu a ema ma mamu', 'am')
2
```

9. Znakový reťazec vieme prevrátiť pomocou zápisu `retazec[::-1]`. Napíšte funkciu `prevrat(retazec)`, ktorá len pomocou cyklu a zreťazovania prevráti zadaný reťazec. Funkcia nič nevypisuje, jej výsledkom (`return`) je nový znakový reťazec.

   - napr.

```
>>> prevrat('tseb eht si nohtyP')
'Python is the best'
```

10. Napíšte funkciu `bez_medzier(text)`, ktorá z daného textu vyhodí všetky medzery. Funkcia nič nevypisuje, jej výsledkom (`return`) je znakový reťazec.

    - napr.

```
>>> bez_medzier('  Mon  tyPy thon   ')
'MontyPython'
```

11. Napíšte funkciu `nahrad_samo(text, znaky)`, ktorá v zadanom texte nahradí všetky samohlásky (`'aeiouy'`) zadaným reťazcom. Funkcia nič nevypisuje, ale vráti nový znakový reťazec.

    - napr.

```
>>> nahrad_samo('sedi mucha na stene', 'a')
'sada macha na stana'
>>> nahrad_samo('sedi mucha na stene', 'uo')
'suoduo muochuo nuo stuonuo'
```

12. Napíšte funkciu `do_desiatkovej(cislo)`, ktorá prevedie zadané celé číslo do znakového reťazca v desiatkovej sústave. Nepoužite pritom funkciu `str()` a ani formátovacie reťazce. Funkcia by mala postupne deliť dané číslo desiatimi a zo zvyškov po delení skladať výsledný reťazec. Funkcia nič nevypisuje, ale vráti tento výsledný znakový reťazec.

    - napr.

```
>>> do_desiatkovej(370042)
'370042'
>>> do_desiatkovej(-13)
'-13'
```

13. Napíšte funkciu `zo_sestnastkovej(retazec)`, ktorá z reťazca, ktorý reprezentuje číslo v 16-ovej sústave, vráti celé číslo. Nepoužite štandardnú funkciu `int()` (môžete ju použiť na kontrolu správnosti výsledku).

    - napr.

```
>>> zo_sestnastkovej('a9EF')
43503
>>> zo_sestnastkovej('abcdef')
11259375
```

14. Napíšte funkciu `rozsekaj(text, sirka)`, ktorá vypíše zadaný text do viacerých riadkov, pričom každý (možno okrem posledného) má presne `sirka` znakov.

    - napr.

```
>>> rozsekaj('Anicka dusicka, kde si bola', 5)
Anick
a dus
icka,
 kde
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
si bo
la
```

15. Vymysleli sme takéto tajné zašifrovanie textu: v celý text sa rozdelí na dvojice znakov a v každej sa navzájom oba znaky vymenia, teda prvý s druhým, tretí so štvrtým, piaty so šiestym, atď. Ak je nepárny počet znakov, tak posledný znak sa nevymieňa. Napíšte funkciu `sifra2(text)`, ktorá takto zašifruje zadaný text (zrejme sa dá použiť aj na odšifrovanie).

    - napr.

```
>>> sifra2('programujem')
rpgoarumejm
>>> sifra2('rpgoarumejm')
programujem
```

16. Ďalší typ šifrovania bude modifikovať každý jeden znak v zadanom texte: ak je týmto znakom malé alebo veľké písmeno, tak ho v abecede cyklicky posunie o `x` pozícii vpravo (za písmenom `'z'` nasleduje `'a'`). Ak znakom v text nie je písmeno, tak takýto znak ostáva bez zmeny. Napíšte funkciu `sifra1(text, x)`, ktorá takto zašifruje (teda aj odšifruje) zadaný text. Ak je `x` záporné, posun v abecede je v opačnom smere.

    - napr.

```
>>> sifra1('Python', 1)
'Qzuipo'
>>> sifra1('Qzuipo', -1)
'Python'
>>> sifra1(sifra1('Why Python?', 20), 6)
'Why Python?'



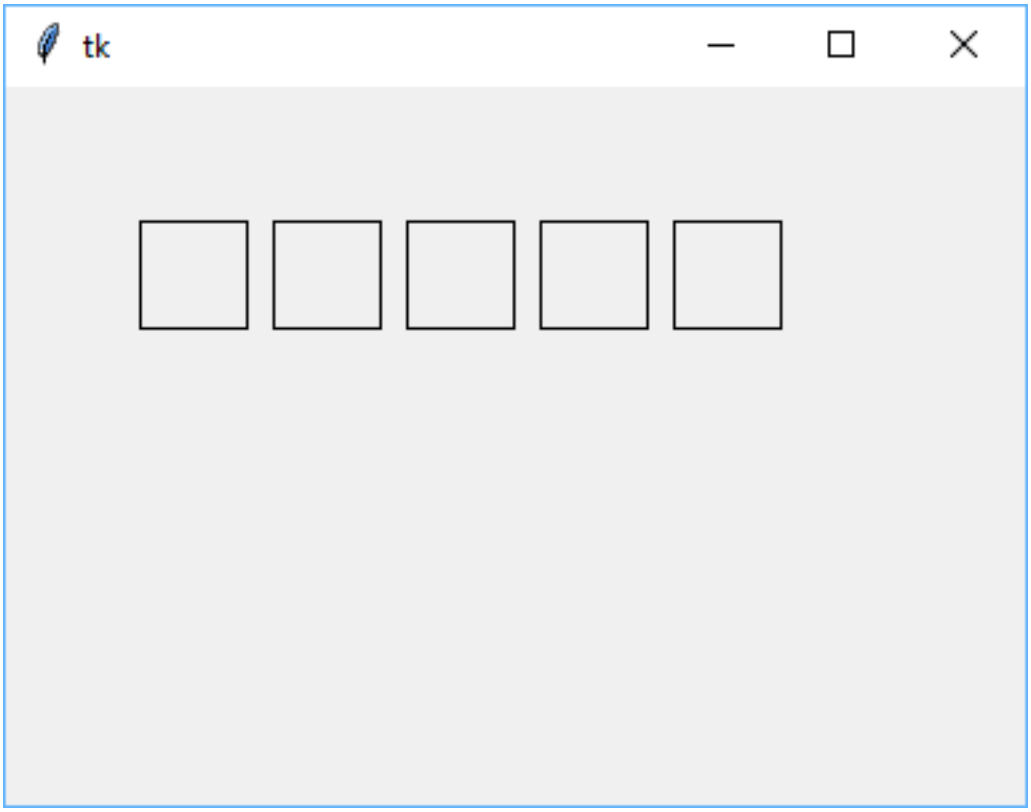
- ret'azec po znakoch - ord, chr+1
```

17. Na konci prednášky je funkcia `kresli(retazec)`, pomocou ktorej môžeme vytvárať nejakú kresbu zakódovanú písmenami `'svjz'`. Dopíšte do tejto funkcie spracovanie týchto ďalších znakov:

    - `'h'` - kresliace pero sa bude odteraz pohybovať bez kreslenia (pero hore)
    - `'d'` - kresliace pero bude odteraz pri pohybe kresliť (pero dole)
    - číslice od `'1'` do `'9'` - nasledovný príkaz (jeden z `'svjz'`) sa vykoná príslušný počet krát
    - napr.

```
>>> kresli('4v4j4z4sh5vd'*5)
```

    nakreslí vedľa seba 5 štvorcov:

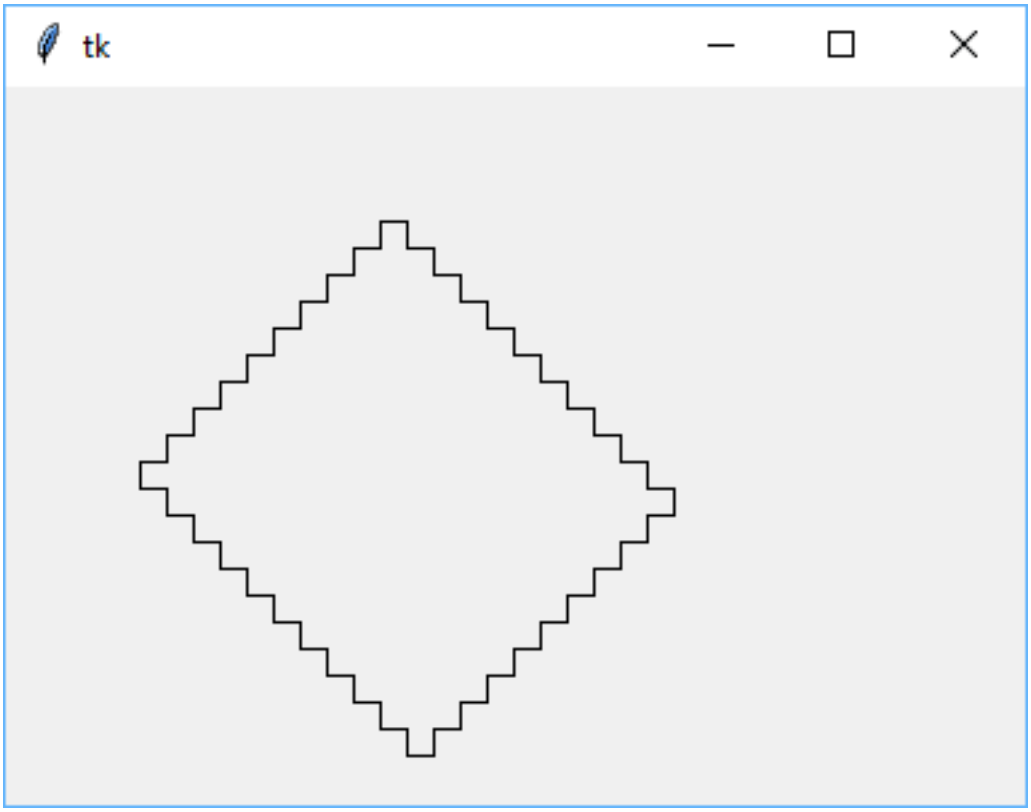


18. Napíšte funkcie, ktoré vygenerujú znakové reťazce pre funkciu `kresli()`:

    - funkcia `schody(pocet, smer)` - vygeneruje príslušný počet stupienkov daným smerom, kde smer=0 označuje smer na sever a východ, smer=1 označuje juh a východ, smer=2 označuje juh a západ a smer=3 označuje sever a západ, napr. (posunuli sme štart)

```
>>> schody(4, 0) + schody(4, 1)
'svsvsvsvjvjvjvjv'
>>> kresli(schody(10, 0) + schody(10, 1) + schody(10, 2) + schody(10, 3))
```

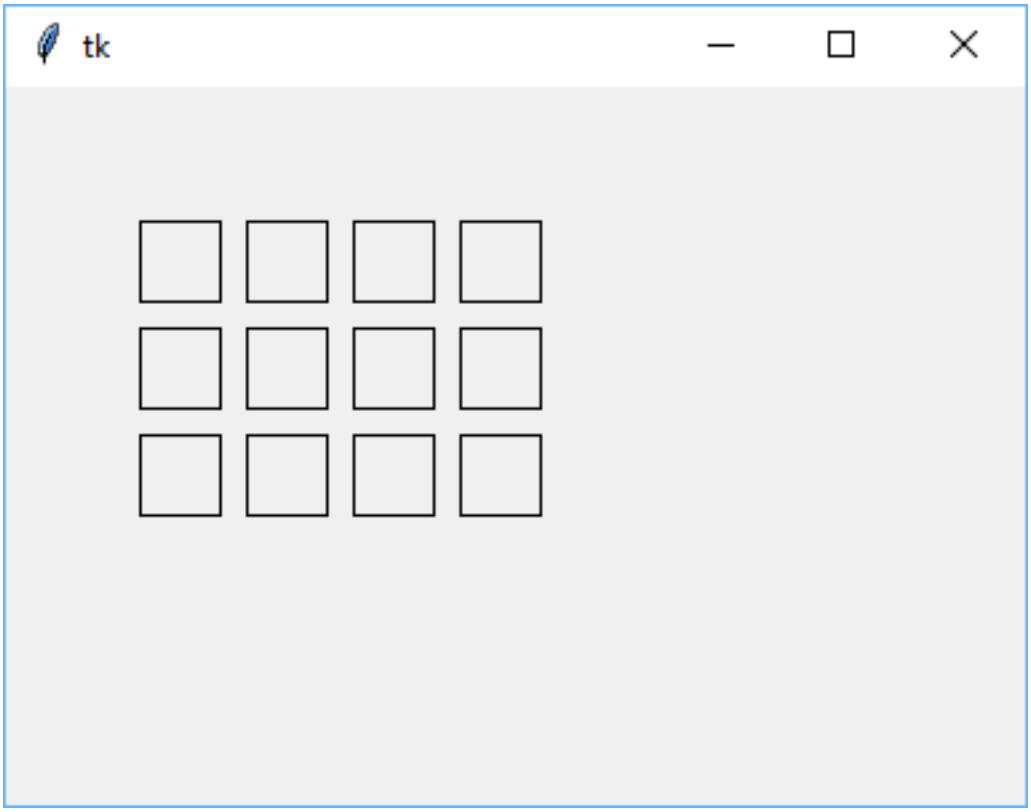
    nakreslí:



- funkcia `stvorce(n, m)` - vygeneruje siet stvorcekov velkosti 3 (t.j. `'3v3j3z3s'`): `n` je počet riadkov a `m` je počet stĺpcov, napr.

```
>>> kresli(stvorce(3, 4))
```

nakreslí:



6.4 2. Domáce zadanie

L.I.S.T.

- riešenie odovzdávajte na úlohový server https://list.fmph.uniba.sk/

Napíšte pythonovský skript, ktorý bude definovať tieto 4 funkcie:

- `pocet_dni_v_mesiaci(mesiac, priestupny=False)` - vráti číslo od 28 do 31, mesiace číslujeme od 1 pre január do 12 pre december
  - zrejme `pocet_dni_v_mesiaci(2, True)` vráti 29
- `pocet_dni_medzi(mesiac1, rok1, mesiac2, rok2)` - pre dva dátumy vypočíta počet dní, ktoré sú medzi nimi
  - prvý dátum je `1. mesiac1 rok1`, druhý dátum je `1. mesiac2 rok2`
  - môžete predpokladať, že prvý dátum je pred alebo rovný druhému dátumu, keď sa oba dátumy rovnajú, funkcia vráti 0
  - oba parametre roky budú v intervale <1901, 2099>
  - napr.

```
>>> pocet_dni_medzi(9, 2017, 10, 2017)       # medzi 1.9.2017 a 1.10.2017
30
>>> pocet_dni_medzi(9, 2017, 9, 2018)        # medzi 1.9.2017 a 1.9.2018
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
365
>>> pocet_dni_medzi(1, 1999, 10, 2017)      # medzi 1.1.1999 a 1.10.2017
6848
```

  - zrejme využijete funkciu `pocet_dni_v_mesiaci()` a to, že priestupný rok (`rok%4==0`) má 366 dní a nepriestupný 365

- `den_v_tyzdni(den, mesiac, rok)` - vráti číslo od 1 do 7, ktoré označujú pondelok až nedeľu
  - môžete to počítať tak, že najprv zistíte počet dní, ktoré uplynuli od dátumu **1.1.1901** a keďže vieme, že vtedy bol **utorok**, ľahko z toho vypočítate deň v týždni (bude to nejaký zvyšok po delení 7)
  - môžete predpokladať, že dátum bude zadaný korektne a bude z intervalu <1.1.1901, 31.12.2099>
  - napr.

```
>>> den_v_tyzdni(18, 10, 2017)
3
>>> den_v_tyzdni(1, 1, 1901)
2
>>> den_v_tyzdni(23, 6, 1912)
7
```

  - Viete zistiť, čím je dátum 23. júna 1912 zaujímavý?

- `kalendar(mesiac, rok)` - vypíše (pomocou `print()`) kalendár pre jeden mesiac v takomto tvare
  - napr.

```
>>> kalendar(10, 2017)
10. mesiac 2017
po ut st st pi so ne
                    1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

  - prvý riadok obsahuje číslo mesiaca a rok, druhý riadok obsahuje mená dní v týždni presne v tomto poradí, ďalej nasledujú dni v mesiaci, ktoré sú naformátované presne do zodpovedajúcich stĺpcov
  - treba si dať pozor na medzery

Na otestovanie správnosti vašich funkcii môžete využiť napr. tento štandardný modul:

```
>>> import calendar
>>> calendar.prmonth(2017, 10)
    October 2017
Mo Tu We Th Fr Sa Su
                    1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

Váš odovzdaný program s menom `riesenie2.py` musí začínať tromi riadkami komentárov:

```
# 2. zadanie: kalendár
# autor: Janko Hraško
# datum: 18.10.2017
```

V programe používajte len konštrukcie jazyka Python, ktoré sme sa učili na prvých 6 prednáškach. Nepoužívajte príkaz `import` ani indexovanie pomocou hranatých zátvoriek `[ ]`.

Projekt `riesenie2.py` odovzdávajte na úlohový server https://list.fmph.uniba.sk/ najneskôr do 23:00 **31. októbra**, kde ho môžete nechať otestovať. Odovzdať projekt aj ho testovať môžete ľubovoľný počet krát. Môžete zaň získať **10 bodov**.

KAPITOLA 7

7. Textové súbory

So súbormi súvisia znakové reťazce ale aj príkazy `input()` a `print()` na vstup a výstup:

- znakové reťazce sú postupnosti znakov (v kódovaní Unicode), ktoré môžu obsahovať aj znaky konca riadka `'\n'`
- funkcia `input()` prečíta zo štandardného vstupu znakový reťazec, t.j. **číta** z klávesnice
- funkcia `print()` **zapíše** reťazec na štandardný výstup, t.j. do textového konzolového okna

Textový súbor

Je postupnosť znakov, ktorá môže obsahovať aj znaky konca riadka. Väčšinou je táto postupnosť uložená na disku v súbore.

Na textový súbor sa môžeme pozerať ako na postupnosť riadkov (môže to byť aj prázdna postupnosť), pričom každý riadok je postupnosťou znakov (znakový reťazec), ktorá je ukončená špeciálnym znakom `'\n'`.

So súbormi sa vo všeobecnosti pracuje takto:

- najprv musíme vytvoriť spojenie medzi našim programom a súborom, ktorý je veľmi často v nejakej externej pamäti - tomuto hovoríme **otvoriť súbor**
- teraz sa dá so súborom pracovať, t.j. môžeme z neho čítať, alebo do neho zapisovať
- keď so súborom skončíme prácu, musíme zrušiť spojenie, hovoríme tomu **zatvoriť súbor**

7.1 Čítanie zo súboru

Najprv sa naučíme čítať zo súboru, čo označuje, že postupne sa náš program dozvedá, aký je obsah súboru.

7.1.1 Otvorenie súboru na čítanie

Súbor otvárame volaním štandardnej funkcie `open()`, ktorej oznámime meno súboru, ktorý chceme čítať. Táto funkcia vráti **referenciu** na súborový objekt (hovoríme tomu aj **dátový prúd**, t.j. **stream**):

```
premenná = open('meno_súboru', 'r')
```

Do **súborovej premennej** sa priradí **spojenie** s uvedeným súborom. Najčastejšie je tento súbor umiestnený v tom istom priečinku, v ktorom sa nachádza samotný Python skript. Meno súboru môže obsahovať aj celú cestu k súboru, prípadne môže byť relatívna k umiestneniu skriptu.

Meno súborovej premennej je vhodné voliť tak, aby nejako zodpovedalo vzťahu k súboru, napr.

```
subor = open('pribeh.txt', 'r')
kniha = open('c:/dokumenty/python.txt', 'r')
file = open('../texty/prvy.txt', 'r')
```

V týchto príkladoch otvárania súborov vidíte, že meno súboru môže byť kompletná cesta `'c:/dokumenty/python.txt'` alebo relatívna k pozícii skriptu `'../texty/prvy.txt'`.

7.1.2 Čítanie zo súboru

Najčastejšie sa informácie zo súboru čítajú po celých riadkoch. Možností, ako takto čítať je viac. Základný spôsob je:

```
riadok = subor.readline()
```

Funkcia `readline()` je **metódou** súborovej premennej, preto za meno súborovej premennej píšeme bodku a meno funkcie. Funkcia vráti znakový reťazec - prečítaný riadok aj s koncovým znakom `'\n'`. Súborová premenná si zároveň zapamätá, kde v súbore sa práve nachádza toto čítanie, aby každé ďalšie zavolanie `readline()` čítalo ďalšie a ďalšie riadky.

Funkcia vráti prázdny reťazec `''`, ak sa už prečítali všetky riadky a teda pozícia čítania v súbore je na konci súboru.

Zrejme prečítaný riadok nemusíme priradiť do premennej, ale môžeme ho spracovať aj inak, napr.

```
subor = open('subor.txt', 'r')
print(subor.readline())
print('dlzka =', len(subor.readline()))
print(subor.readline()[::-1])
```

V tomto programe sa najprv vypíše obsah prvého riadka, potom dĺžka druhého riadka (aj s koncovým znakom `'\n'`) a na záver sa vypíše tretí riadok ale otočený (aj s koncovým znakom `'\n'`, ktorý sa vypíše ako prvý).

7.1.3 Zatvorenie súboru

Keď skončíme prácu so súborom, **uzavrieme** otvorené spojenie volaním:

```
subor.close()
```

Tým sa uvoľnia všetky systémové zdroje (resources), ktoré boli potrebné pre otvorený súbor. Zrejme so zatvoreným súborom sa už nedá ďalej pracovať a napr. čítanie by vyvolalo chybovú správu.

Ďalej ukážeme, ako môžeme pracovať s textovým súborom. Predpokladajme, že máme pripravený nejaký textový súbor, napr. `'subor.txt'`:

```
prvy riadok
druhy riadok
    treti riadok

posledny riadok
```

Tento súbor má 5 riadkov (štvrtý je prázdny) a preto ho môžeme celý prečítať a vypísať takto:

```
t = open('subor.txt', 'r')

for i in range(5):
    riadok = t.readline()
    print(riadok)

t.close()
```

program vypíše:

```
prvy riadok

druhy riadok

    treti riadok



posledny riadok
```

Keďže metóda `readline()` prečíta zo súboru celý riadok aj s koncovým `'\n'`, príkaz `print()` k tomu pridáva ešte jeden svoj `'\n'` a preto je za každým vypísaným riadkom ešte jeden prázdny. Buď príkazu `print()` povieme, aby na koniec riadka nevkladal prechod na nový riadok (napr. `print(riadok, end='')`), alebo pred samotným výpisom z reťazca `riadok` vyhodíme posledný znak, napr.

```
t = open('subor.txt', 'r')
for i in range(5):
    riadok = t.readline()
    print(riadok[:-1])
t.close()
```

Takéto vyhadzovanie posledného znaku z reťazca môže nefungovať celkom správne pre posledný riadok súboru, ktorý nemusí byť ukončený znakom `'\n'`.

7.1.4 Zistenie konca súboru

Predchádzajúci program má najväčší nedostatok v tom, že predpokladá, že vstupný súbor obsahuje presne 5 riadkov. Ak by sme tento počet dopredu nepoznali, musíme použiť nejaký iný spôsob. Keďže metóda `readline()` vráti na konci súboru **prázdny reťazec** `''` (pozor, nie jednoznakový reťazec `'\n'`), môžeme práve túto podmienku využiť na testovanie konca súboru:

```
t = open('subor.txt', 'r')
riadok = t.readline()
while riadok != '':
    print(riadok, end='')
    riadok = t.readline()
t.close()
```

Tento program už správne vypíše všetky riadky súboru, hoci nevidíme, či je štvrtý riadok prázdny alebo obsahuje aj nejaké medzery:

```
prvy riadok
druhy riadok
    treti riadok

posledny riadok
```

šablóna čítania s while-cyklom

Takto budú často vyzerať naše programy, ktoré budú čítať súbor po riadkoch:

```
subor = open('meno súboru', 'r')

riadok = subor.readline()
while riadok != '':
    # ... spracuj riadok
    riadok = subor.readline()

subor.close()
```

Môžeme to prepísať aj s použitím `break`:

```
t = open('subor.txt', 'r')
while True:
    riadok = t.readline()
    if riadok == '':
        break
    print(riadok, end='')
t.close()
```

Niekedy sa nám môže hodiť taký výpis prečítaného reťazca, ktorý napr. zobrazí nielen medzery na konci reťazca, ale aj ukončovací znak `'\n'`. Využijeme na to štandardnú funkciu `repr()`.

funkcia `repr()`

Volanie štandardnej funkcie:

```
repr(ret'azec)
```

vráti takú reťazcovú reprezentáciu daného parametra, aby sme po jeho vypísaní (napr. funkciou `print()`) dostali presný taký tvar, aký očakáva Python pri zadávaní konštanty, teda aj s apostrofmi, prípadne aj so znakom `'\'` pri špeciálnych znakoch.

Môže sa použiť aj pri ladení a testovaní, lebo máme lepší prehľad o skutočnom obsahu reťazca. Napr.

```
>>> a = 'ahoj   \naj "apostrof" \' v texte  \n'
>>> print(a)
ahoj
aj "apostrof" ' v texte

>>> print(repr(a))
'ahoj   \naj "apostrof" \' v texte  \n'
```

Doplníme do while-cyklu o volanie funkcie `repr()`:

```
t = open('subor.txt', 'r')
riadok = t.readline()
while riadok != '':
    print(repr(riadok))
    riadok = t.readline()
t.close()
```

Po spustení vidíme, že sa vypíše:

```
'prvy riadok\n'
'druhy riadok\n'
'    treti riadok     \n'
'\n'
'posledny riadok\n'
```

Namiesto `while riadok != '':` môžeme zapísať `while riadok:`.

Vidíme, že tretí riadok obsahuje medzery aj na konci riadka. Ak by sme pri čítaní súboru nepotrebovali informácie o medzerách na začiatku a konci riadkov, môžeme využiť reťazcovú metódu `strip()`:

```
t = open('subor.txt', 'r')
riadok = t.readline()
while riadok:
    print(repr(riadok.strip()))
    riadok = t.readline()
t.close()
```

vypíše:

```
'prvy riadok'
'druhy riadok'
'treti riadok'
''
'posledny riadok'
```

Všimnite si, že takto sme sa zbavili aj záverečného znaku `'\n'`. Ak by sme namiesto `riadok.strip()` použili `riadok.rstrip()`, vyhodia sa medzerové znaky len od konca reťazca (sprava) a na začiatku riadkov medzery ostávajú.

Použitie for-cyklu pre čítanie zo súboru

Python umožňuje použiť for-cyklus, aj pre súbory, o ktorých dopredu nevieme, koľko majú riadkov. For-cyklus má vtedy tvar:

```
for riadok in súborová_premenná:
    prikazy
```

kde `riadok` je ľubovoľná premenná cyklu, do ktorej sa budú postupne priraďovať všetky prečítané riadky - POZOR! aj s koncovým `'\n'`, **súborová_premenná** musí byť otvoreným súborom na čítanie.

Program sa teraz výrazne zjednoduší:

```
t = open('subor.txt', 'r')
for riadok in t:
    print(repr(riadok))
t.close()
```

šablóna čítania s for-cyklom

S použitím for-cyklu môžu byť naše programy, ktoré čítajú súbor po riadkoch, ešte čitateľnejšie:

```
subor = open('meno súboru', 'r')

for riadok in subor:
    # ... spracuj riadok

subor.close()
```

Takýto for-cyklus bude fungovať aj vtedy, keď sme už zo súboru niečo čítali a potrebujeme spracovať už len zvyšok súboru, napr.

```
t = open('subor.txt', 'r')
riadok = t.readline()
print('najprv som precital:', repr(riadok.rstrip()))
print('v subore ostali este tieto riadky:')
for riadok in t:
    print(repr(riadok.rstrip()))
t.close()
```

Teraz sa vypíše:

```
najprv som precital: 'prvy riadok'
v subore ostali este tieto riadky:
'druhy riadok'
'    treti riadok'
''
'posledny riadok'
```

Prečítanie celého súboru do jedného reťazca

Zapíšme takúto úlohu: do jednej reťazcovej premennej prečítame všetky riadky súboru, pričom im ponecháme koncové `'\n'`. Zrejme, ak by sme takýto reťazec naraz celý vypísali (pomocou `print()`), dostali by sme kompletný výpis. Napr.

```
t = open('subor.txt', 'r')
cely_subor = ''
for riadok in t:
    cely_subor = cely_subor + riadok
t.close()
print(cely_subor, end='')
```

Všimnite si, že riadok programu `cely_subor = cely_subor + riadok` by sme mohli zapísať aj takto `cely_subor += riadok`

To, čo sme prácne skladali cyklom, za nás urobí metóda `read()`, teda

```
t = open('subor.txt', 'r')
cely_subor = t.read()
t.close()
print(cely_subor, end='')
```

Samozrejme, takéto skladanie súboru do jednej reťazcovej premennej môžeme urobiť len vtedy, ak spracovávaný súbor nie je väčší ako kapacita pamäte pre Python (závisí to od vášho počítača, ale je to od niekoľkých 100MB po GB).

metóda súbor.read()

súbor.**read**(*počet_znakov*)

súbor.**read**()

Parametre počet_znakov – určuje počet znakov, ktoré budem chcieť zo suboru prečítať

Metóda prečíta zo súboru a vráti ako výsledok funkcie zadaný počet znakov. Ak už v súbore toľko znakov nie je, alebo sme nezadali počet znakov, vráti maximálny možný počet (alebo aj prázdny reťazec). Napr.

```
>>> t = open('subor.txt')
>>> print('prvy znak =', repr(t.read(1)))
>>> print('dalsich 10 znakov =', repr(t.read(10)))
>>> print('zvysok suboru =', repr(t.read()))
```

7.1.5 Slovenčina v súbore

Hoci znakové reťazce v Pythone sú ukladané v kódovaní **Unicode**, pri práci so súbormi, ktoré obsahujú znaky s diakritikou, musíme upresniť aj kódovanie v súbore. Samotné súbory môžu mať pri uložení na disku rôzne kódovania (závisí to aj od vášho operačného systému), napr. `'cp1250'`, `'iso88591'`, `'utf-8'`, ... a pri ich otváraní treba toto kódovanie uvádzať ako parameter funkcie `open()`. Súbory, ktoré dostanete na čítanie v tomto kurze, budú mať väčšinou kódovanie `'utf-8'`, ale vaše vlastné súbory môžu mať aj iné kódovanie.

Preto súbor s diakritikou najčastejšie otvárame takto:

```
subor = open(meno_suboru, 'r', encoding='utf-8')
```

7.2 Zápis do súboru

Doteraz sme čítali už existujúci súbor. Teraz sa naučíme textový súbor aj vytvárať. Bude to veľmi podobné ako pri čítaní súboru.

7.2.1 Otvorenie súboru

do **súborovej premennej** sa priradí **spojenie** so súborom:

```
subor = open('meno_súboru', 'w')
```

Súbor bude umiestnený v tom istom priečinku, kde sa nachádza samotný Python skript (resp. treba uviesť cestu). Ak tento súbor ešte neexistoval, tento príkaz ho vytvorí (vytvorí sa prázdny súbor). Ak takýto súbor už existoval, tento príkaz ho vyprázdni. Treba si dávať pozor, lebo omylom môžeme prísť o dôležitý súbor.

Možností, ako zapisovať riadky do súboru je viac. My si postupne ukážeme dva z nich: zápis pomocou základnej metódy pre zápis `write()` a pomocou nám známej štandardnej funkcie `print()`. Najprv metóda `write()`:

7.2.2 Zápis do súboru

Zápis nejakého reťazca do súboru urobíme pomocou volania:

```
subor.write(ret'azec)
```

Táto metóda zapíše zadaný reťazec na momentálny koniec súboru. Ak chceme, aby sa v súbore objavili aj koncové znaky `'\n'`, musíme ich pridať do reťazca.

Niekoľko za sebou idúcich zápisov do súboru môžeme spojiť do jedného, napr.

```
subor.write('prvy')
subor.write(' riadok')
subor.write('\n')
subor.write('druhy riadok\n')
```

môžeme zapísať jediným volaním metódy `write()`:

```
subor.write('prvy riadok\ndruhy riadok\n')
```

7.2.3 Zatvorenie súboru

Tak ako pri čítaní súboru sme na záver súbor zatvárali, musíme zatvárať súbor aj pri vytváraní súboru:

```
subor.close()
```

Metóda skončí prácu so súborom, t.j. zruší **spojenie** s fyzickým súborom na disku. Bez volania tejto metódy nemáme zaručené, že Python naozaj fyzicky stihol zapísať všetky reťazce z volania `write()` na disk. Tiež operačný systém by mohol mať problém so znovu otvorením ešte nezatvoreného súboru.

Zápis do súboru ukážeme na príklade, v ktorom vytvoríme niekoľko riadkový súbor `'subor1.txt'`:

```
subor = open('subor1.txt', 'w')
subor.write('zoznam prvocisel:\n')
for ix in 2, 3, 5, 7, 11, 13:
    subor.write(f'cislo {ix} je prvocislo\n')
subor.close()
```

Program najprv do súboru zapísal jeden riadok ‚zoznam prvocisel:‘ a za ním ďalších 6 riadkov:

```
zoznam prvocisel:
cislo 2 je prvocislo
cislo 3 je prvocislo
cislo 5 je prvocislo
cislo 7 je prvocislo
cislo 11 je prvocislo
cislo 13 je prvocislo
```

7.2.4 Zápis do súboru pomocou print()

Doteraz sme štandardný príkaz `print()` používali na výpis do textovej plochy Shellu. Veľmi jednoducho, môžeme presmerovať výstup z už odladeného programu do súboru.

Program vytvorí súbor `'nahodne_cisla.txt'`, do ktorého zapíše pod seba 100 náhodných čísel:

```
import random
subor = open('nahodne_cisla.txt', 'w')
for i in range(100):
    print(random.randint(1, 100), file=subor)
subor.close()
```

Všimnite si nový parameter pri volaní funkcie `print()`, pomocou ktorého presmerujeme výstup do nášho súboru (tu musíme uviesť súborovú premennú už otvoreného súboru na zápis).

Ak by sme chceli, aby boli čísla v súbore nie v jednom stĺpci ale v jednom riadku oddelené medzerou, zapísali by sme:

```
import random
subor = open('nahodne_cisla.txt', 'w')
for i in range(100):
    print(random.randint(1, 100), end=' ', file=subor)
subor.close()
```

Kopírovanie súboru

Ak potrebujeme obsah jedného súboru prekopírovať do druhého (pritom možno niečo spraviť s každým riadkom), môžeme použiť 2 súborové premenné, napr.

```
odkial = open('subor.txt', 'r')
kam = open('subor2.txt', 'w')
for riadok in odkial:
    riadok = riadok.strip()
    if riadok != '':
        kam.write(riadok + '\n')
odkial.close()
kam.close()
```

Program postupne prečíta všetky riadky, vyhodí medzery zo začiatku a z konca každého riadka, a ak je takýto riadok neprázdny, zapíše ho do druhého súboru (keďže `strip()` vyhodil z riadka aj koncové `'\n'`, museli sme ho tam vo `write()` pridať).

Táto istá úloha by sa dala riešiť aj pomocou jednej súborovej premennej - najprv súbor čítame a do jednej reťazcovej premennej pripravujeme obsah nového súboru, nakoniec ho celý zapíšeme:

```
t = open('subor.txt', 'r')
cely = ''
for riadok in t:
    riadok = riadok.strip()
    if riadok != '':
        cely += riadok + '\n'
t.close()
t = open('subor2.txt', 'w')
t.write(cely)
t.close()
```

Ak by sme pri kopírovaní riadkov nepotrebovali meniť nič, môžeme použiť metódu `read()`, napr.

```
t = open('subor.txt', 'r')
cely = t.read()
t.close()
t = open('subor2.txt', 'w')
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
t.write(cely)
t.close()
```

Na prácu so súbormi môžeme využiť špeciálnu programovú konštrukciu, pomocou ktorej bude Python vedieť, že sme už so súborom skončili pracovať a teda ho treba zatvoriť. Samotný príkaz má aj iné využitie ako pre prácu so súbormi, ale v tomto kurze sa s tým nestretneme.

7.3 Konštrukcia with

Všeobecný tvar príkazu je:

```
with open(...) as premenna:
    prikaz
    prikaz
    ...
```

Touto príkazovou konštrukciou sa otvorí požadovaný súbor a referencia na súbor sa priradí do premennej uvedenej za `as`. Ďalej sa vykonajú všetky príkazy v bloku a po ich skončení sa súbor **automaticky** zatvorí. Urobí sa skoro to isté, ako

```
premenna = open(...)
prikaz
prikaz
...
premenna.close()
```

Odporúčame pri práci so súbormi používať čo najviac práve túto konštrukciu, čo oceníme napr. aj prácu so súbormi vo funkciách, v ktorých príkaz `return`, ak sa použije vo vnútri bloku `with`, automaticky zatvorí otvorené súbory.

Ukážme niekoľko príkladov zápisu pomocou `with`:

1. Prečítaj a vypíš obsah celého súboru:

   ```
   with open('subor.txt') as subor:
       print(subor.read())
   ```

2. Vytvor súbor s tromi riadkami:

   ```
   with open('subor.txt','w') as file:
       print('prvy\ndruhy\ntreti\n', file=file)
   ```

   Všimnite si tu použitie mena súborovej premennej: nazvali sme ju `file` rovnako ako meno parametra vo funkcii `print()`, preto musíme presmerovanie do súboru zapísať ako `print(..., file=file)`: formálnemu parametru `file` priradíme hodnotu skutočného parametra `file`.

3. Vytvor súbor 100 náhodných čísel:

   ```
   import random
   with open('cisla.txt','w') as file:
       for i in range(100):
           file.write(str(random.randrange(1000))+' ')
   ```

7.3.1 Automatické zatváranie súboru

Python sa v jednoduchých prípadoch snaží korektne zatvoriť otvorené súbory, keď už sme s nimi skončili pracovať a pritom sme nepoužili metódu `close()`. V serióznych aplikáciách toto nebudeme používať, ale pri jednoduchých testoch a ukážkach sa to objaviť môže.

V nasledovných príkladoch využívame to, že funkcia `open()` vracia ako výsledok súborovú premennú, t.j. spojenie na súbor. Ak toto spojenie potrebujeme použiť len jednorázovo, nemusíme to priradíme do premennej, ale použijeme ho priamo napr. s volaním nejakej metódy.

Ak do súboru zapisujeme len jedenkrát a hneď ho zatvárame, nemusíme na to vytvárať súborovú premennú, ale priamo pri otvorení urobíme jeden zápis. Vtedy sa súbor automaticky zatvorí. Napr.

```
>>> open('subor2.txt', 'w').write('first line\nsecond line\nend of file\n')
```

Týmto jedným príkazom sme vytvorili nový súbor `'subor3.txt'`, zapísali sme do neho 3 riadky a automaticky sa na záver zatvoril (dúfajme...). Korektný zápis. ktorý by sme použili v programe:

```
with open('subor2.txt', 'w') as f:
    f.write('first line\nsecond line\nend of file\n')
```

Podobne by sme to zapísali aj pomocou príkazu `print()`:

```
>>> print('first line\nsecond line\nend of file', file=open('subor3.txt', 'w'))
```

alebo radšej korektne:

```
with open('subor3.txt', 'w') as f:
    print('first line\nsecond line\nend of file', file=f)
```

Nezabudnite, že ak súbor `'subor3.txt'` niečo pred tým obsahoval, týmto príkazom sa celý prepíše.

Vyššie uvedený príklad, ktorý kopíroval kompletný súbor:

```
t = open('subor.txt', 'r')
cely = t.read()
t.close()
t = open('subor2.txt', 'w')
t.write(cely)
t.close()
```

by sa dal úsporne zapísať takto:

```
>>> open('subor2.txt', 'w').write(open('subor.txt', 'r').read())
```

čo je zrejme veľmi ťažko čitateľné, a my to určite budeme zapisovať radšej takto korektne:

```
with open('subor.txt', 'r') as r:
    with open('subor2.txt', 'w') as w:
        w.write(r.read())
```

Niekedy to môžete vidieť aj v takomto tvare:

```
with open('subor.txt', 'r') as r, open('subor2.txt', 'w') as w:
    w.write(r.read())
```

To znamená, že do príkazu `with` môžeme zadať naraz viac otvorených súborov (oddelených čiarkou). Po skončení bloku príkazov sa všetky takto otvorené súbory automaticky zatvoria.

Hoci teraz už vieme zapísať príkaz, ktorý na konzolu vypíše obsah nejakého textového súboru takto:

```
>>> print(open('readme.txt').read())
```

budeme to zapisovať korektnejšie:

```
>>> with open('readme.txt') as t:
        print(t.read())
```

alebo

```
>>> with open('readme.txt') as t: print(t.read())
```

Všimnite si, že sme pri `open()` nepoužili parameter `'r'` pre označenie otvorenia na čítanie. Keď totiž pri otváraní nezapíšeme `'r'`, Python si domyslí práve otváranie súboru na čítanie.

Ak očakávame, že otvorený súbor je príliš veľký a my ho naozaj nepotrebujeme vypísať celý, zapíšeme:

```
>>> with open('readme.txt') as t: print(t.read()[:1000])
```

7.3.2 Pridávanie riadkov do súboru

Videli sme dva rôzne typy otvárania textového súboru:

- `t = open('subor.txt', 'r')` - súbor sa otvoril na len čítanie, ak ešte neexistoval, program spadne
- `t = open('subor.txt', 'w')` - súbor sa otvoril na len zapisovanie, ak ešte neexistoval, tak sa vytvorí prázdny, inak sa zruší doterajší obsah (zapíše sa prázdny obsah)

Zoznámime sa s ešte jednou voľbou, pri ktorej sa súbor otvorí na zápis, ale nezruší sa jeho pôvodný obsah. Namiesto toho sa nové riadky budú pridávať na koniec súboru. Napr. ak máme súbor `'subor3.txt'` s tromi riadkami:

```
first line
second line
end of file
```

môžeme do neho pripísať ďalšie riadky, napr. takto: namiesto `'r'` a `'w'` pri otváraní súboru použijeme `'a'`, ktoré označuje anglické **append**:

```
t = open('subor3.txt', 'a')
t.write('pridany riadok na koniec\na este jeden\n')
t.close()
```

v súbore je teraz:

```
first line
second line
end of file
pridany riadok na koniec
a este jeden
```

Zrejme by sme to zvládli naprogramovať aj bez tejto voľby, len pomocou pôvodného čítania a zápisu, ale bolo by to časovo náročnejšie riešenie, napr. takto:

```
with open('subor3.txt', 'r') as t:
    cely = t.read()                         # zapamätá si pôvodný obsah
with open('subor3.txt', 'w') as t:          # vymaže všetko
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    t.write(cely)                           # vráti tam pôvodný obsah
    t.write('pridany riadok na koniec\na este jeden\n')
```

Zistite čo urobí:

```
for i in range(100):
    with open('subor4.txt', 'a') as file:
        print('vypisujem', i, 'riadok', file=file)
```

Uvedomte si, že takéto riešenie je veľmi neefektívne.

7.3.3 Príklad s grafikou

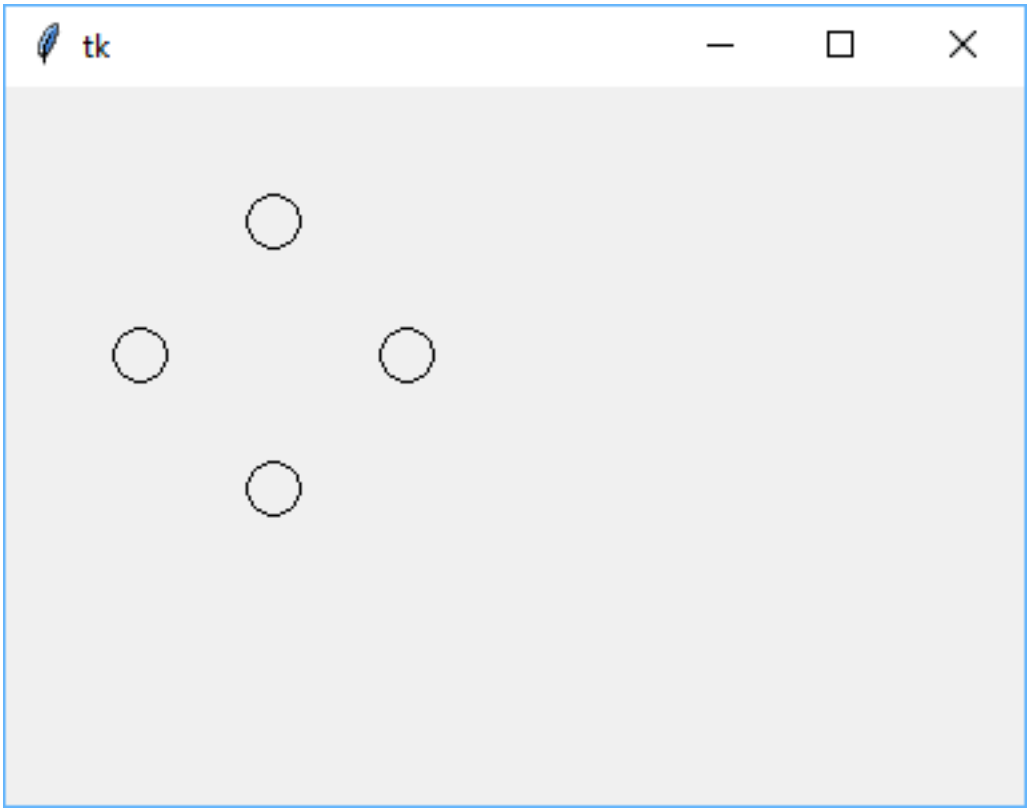
Máme pripravený textový súbor, v každom riadku ktorého sú dve celé čísla - súradnice nejakých bodov v grafickej ploche. Napr.

```
100 50
150 100
50 100
100 150
```

Napíšme program, ktorý prečíta súradnice týchto bodov a do grafickej plochy na príslušné miesta nakreslí malé krúžky:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()
with open('subor.txt') as subor:
    for riadok in subor:
        i = riadok.find(' ')
        x, y = int(riadok[:i]), int(riadok[i:])
        canvas.create_oval(x-10, y-10, x+10, y+10)
```



Na cvičeniach budeme ďalej pracovať s touto ideou. Napr. budeme tieto body spájať úsečkami, alebo budeme generovať vlastné textové súbory s nejakými kresbami (postupnosťami bodov).

7.4 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/
- v riešeniach úloh používajte len konštrukcie a funkcie, ktoré sme sa učili na doterajších prednáškach

K zadaniam, ktoré čítajú textový súbor, pripravte nejaké vhodné testovacie súbory.

1. Napíšte program, ktorý si vypýta meno súboru a potom vypíše prvý, tretí a piaty znak z prvého riadka tohto súboru.
   - napr. ak súbor `'text1.txt'` obsahuje

     ```
     programujem v Pythone
     ```

   - spustenie programu:

     ```
     meno suboru: text1.txt
     tri znaky: 'por'
     ```

2. Napíšte program, ktorý si vypýta meno súboru a potom vypíše počet riadkov a dĺžku najdlhšieho riadka tohto súboru.

- pripravte si vhodný viacriadkový textový súbor a potom výpis môže vyzerať napr. takto

```
meno suboru: text2.txt
pocet riadkov suboru: 13
najdlhsi ma 40 znakov
```

- riešenie zapíšte pomocou:
  (a) čítaním pomocou `readline()`
  (b) čítaním riadkov pomocou for-cyklu
  (c) prečítaním naraz celého súboru pomocou `read()`
  (d) čítaním po jednom znaku pomocou `read(1)`

3. Napíšte funkciu `riadky_s_textom(meno_suboru, text)`, ktorá otvorí zadaný súbor (najlepšie nejaký súbor s pythonovským programom) a vypíše z neho len tie riadky, ktoré obsahujú zadaný text.
   - napr.

```
>>> riadky_s_textom('ries.py', 'if ')
if a != b:
elif b < 7:
    if x:
```

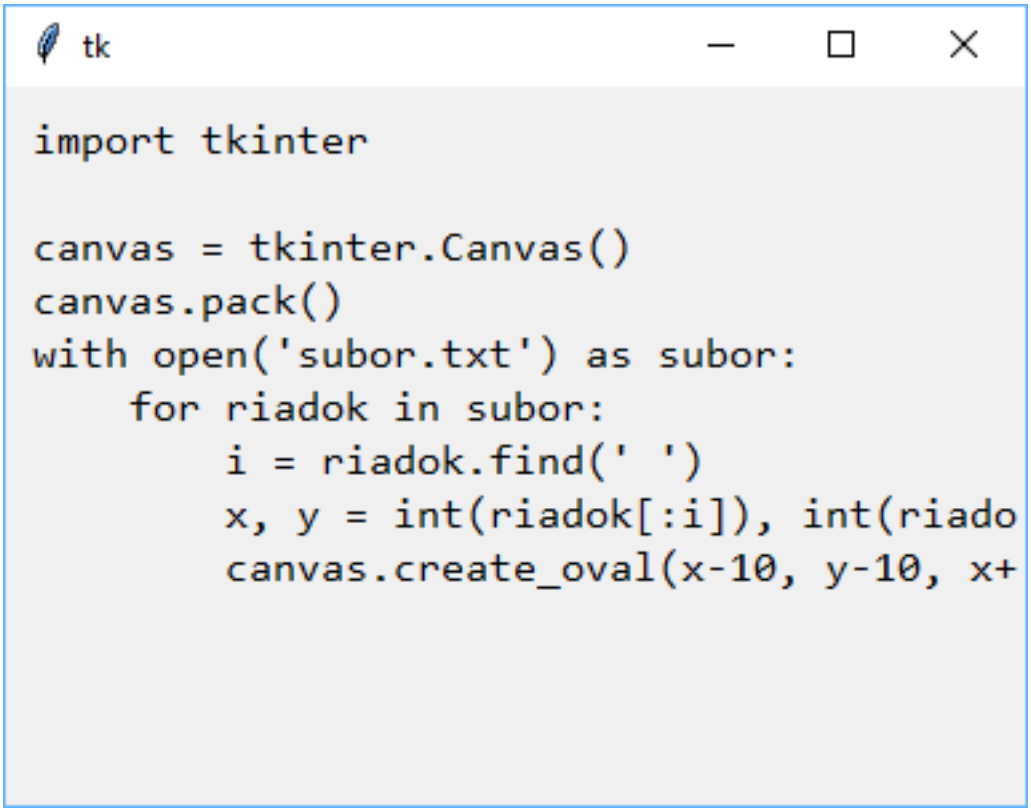
4. Napíšte a otestujte tieto funkcie. Ich parametrom je meno nejakého textového súboru a všetky vrátia (`return`) nejaký riadok súboru:
   - funkcia `posledny_riadok(meno_suboru)`
   - funkcia `predposledny_riadok(meno_suboru)`
   - funkcia `najdlhsi_riadok(meno_suboru)`
5. Napíšte funkciu `priemer(meno_suboru)`: funkcia číta súbor, v ktorom je v každom riadku jedno celé číslo, funkcia zistí priemer týchto hodnôt.
   - napr. ak by súbor obsahoval 10 riadkov s číslami 1 až 10, tak po spustení dostávame

```
>>> print('priemer =', priemer('cisla.txt'))
priemer = 5.5
```

6. Vypíšte obsah textového súboru do grafickej plochy. Súbor obsahuje niekoľko riadkov a funkcia `vypis_subor(meno_suboru)` tieto riadky vypíše pod sebou nejakým fontom. V globálnej premennej `canvas` je referencia na grafickú plochu.
   - napr.

```
import tkinter
canvas = ...
vypis_subor('program.py')
```

   vypíše riadky zadaného súboru:



- pre zarovnanie textu na ľavý okraj môžete vo volaní `create_text()` použiť pomenovaný parameter `anchor='nw'`

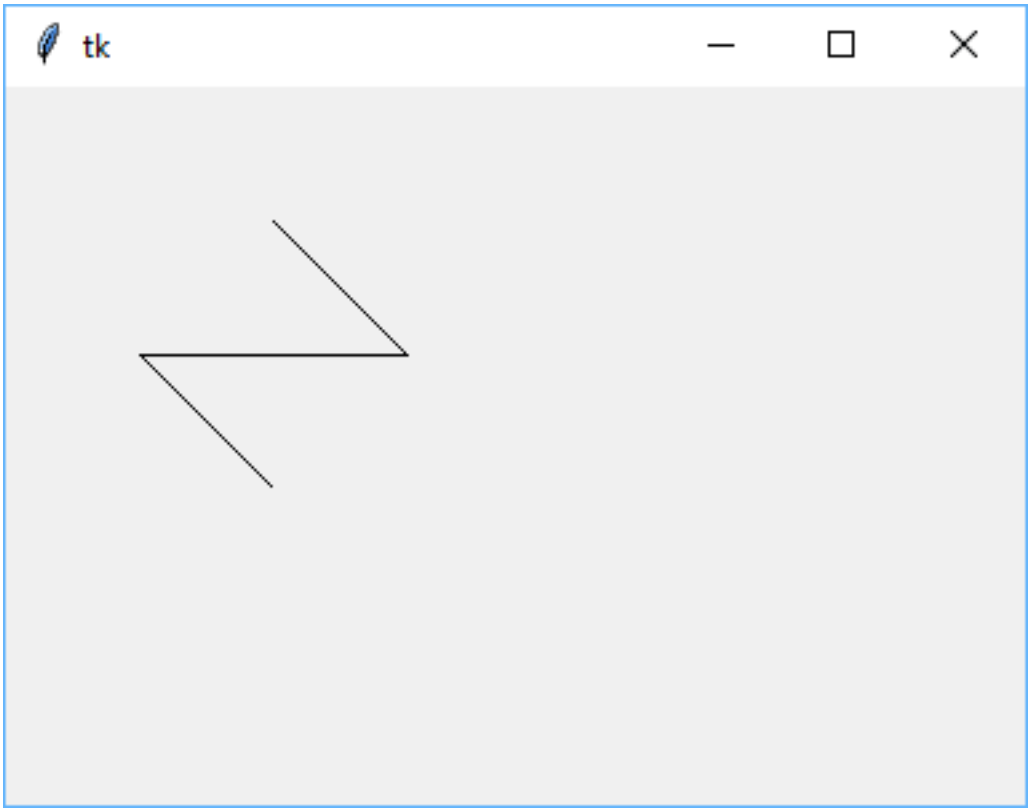
7. Na prednáške bol program, ktorý z textového súboru čítal súradnice bodov x, y a do grafickej plochy na príslušných miestach kreslil malé krúžky. Zapíšte to ako funkciu `kresli(meno_suboru)`, ktorá namiesto kreslenia krúžkov, bude postupne spájať body zo súboru, t.j. nakreslí sa úsečka z prvého bodu do druhého, z druhého do tretieho, ...

   - napr. ak súbor `'body.txt'` obsahuje tieto 4 riadky

   ```
   100 50
   150 100
   50 100
   100 150
   ```

   ```
   import tkinter
   canvas = ...
   kresli('body.txt')
   ```

   nakreslí tieto tri úsečky:



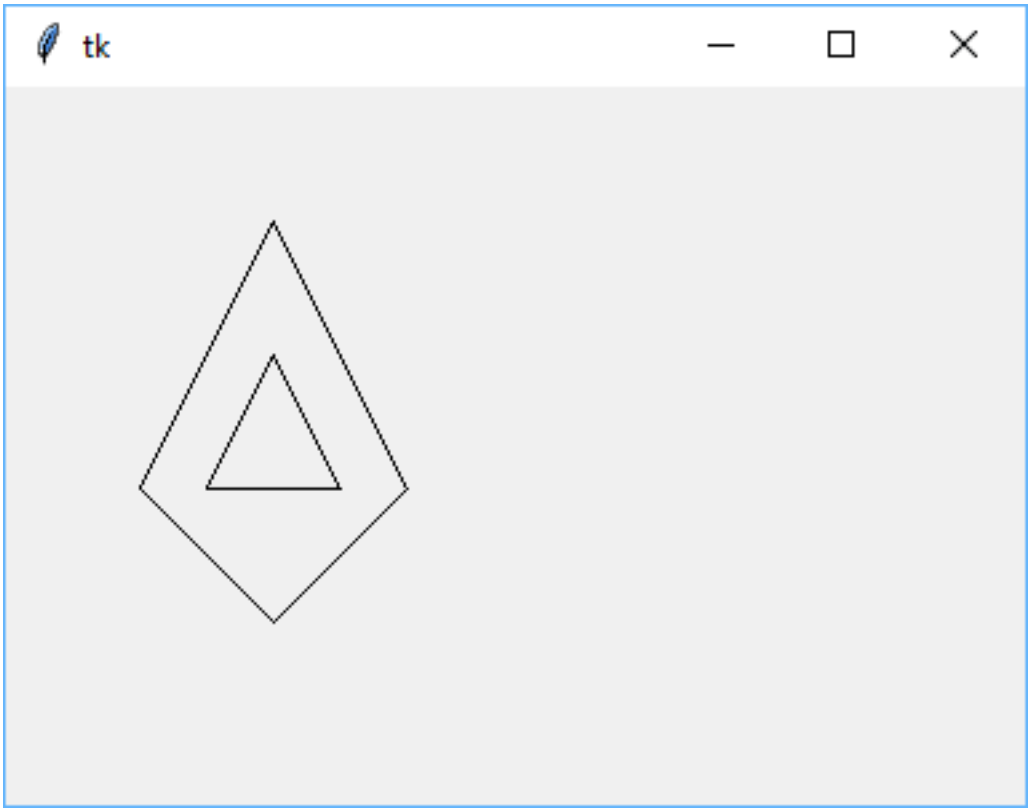
8. Funkciu `kresli()` z predchádzajúceho príkladu vylepšite takto: ak sa vo vstupnom súbore nachádza prázdny riadok, tento označuje, že za ním nasleduje ďalšia skupina bodov, ktorá ale nie je s predchádzajúcimi bodmi spojená.

   - napr. ak súbor obsahuje

```
100 50
150 150
100 200
50 150
100 50

100 100
125 150
75 150
100 100
```

   nakreslí tieto dva útvary:



9. Textový súbor obsahuje v každom riadku jedno meno farby. Napíšte funkciu `do_riadkov(meno_suboru, sirka)`, ktorá vypíše rad štvorčekov (veľkosti 30x30). Každý z týchto štvorčekov zafarbí príslušnou farbou zo súboru. Po vykreslení `sirka` štvorčekov, pokračuje pod týmito v ďalšom rade štvorčekov s ďalšími farbami zo súboru. Takto pokračuje, kým sa neminú všetky farby zo súboru.

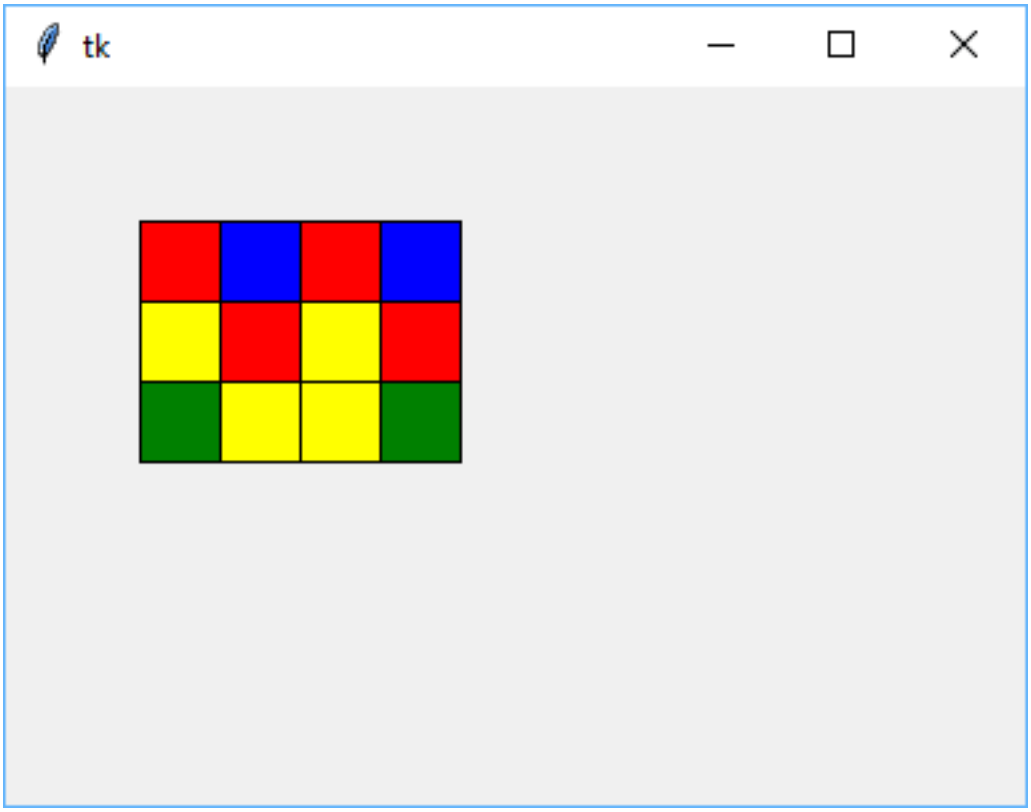
   - napr. ak súbor obsahuje:

```
red
blue
red
blue
yellow
red
yellow
red
green
yellow
yellow
green
```

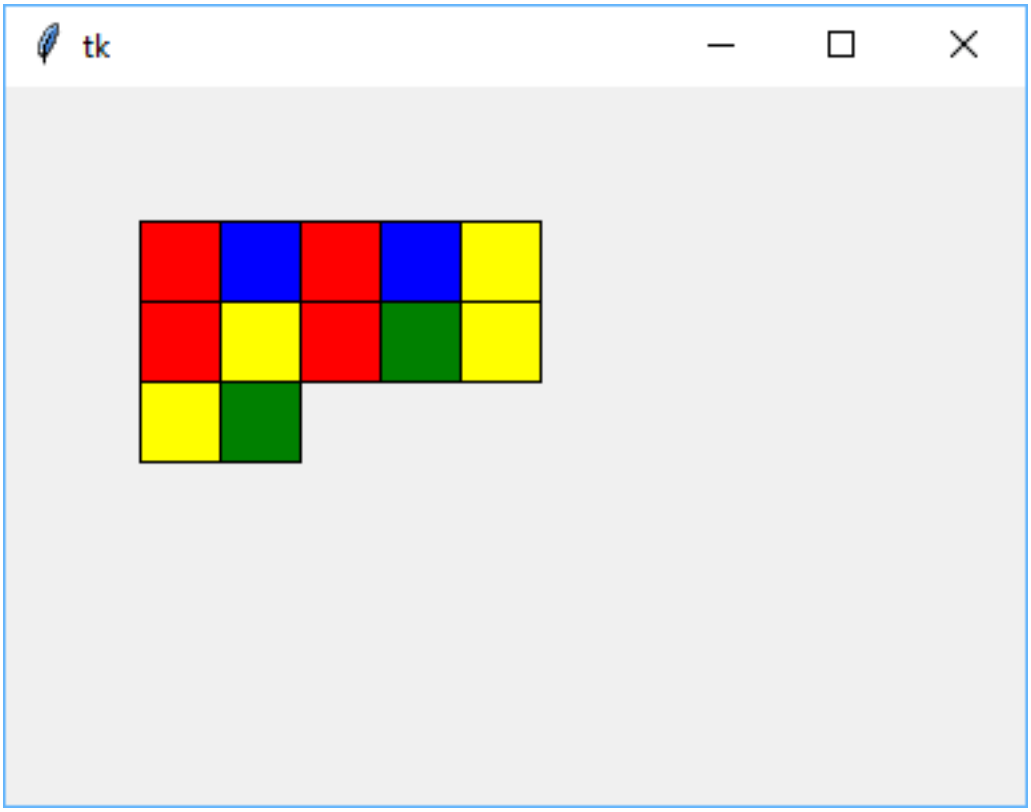
   volanie

```
import tkinter
canvas = ...
do_riadkov('farby.txt', 4)
```

   nakreslí:



- ak by sme tento súbor vypisovali do 5 stĺpcov, dostali by sme:



10. V každom riadku súboru je po jednom slove. Napíšte funkciu `vypis_slova(meno_suboru)`, ktorá po-

stupne vypíše všetky slová súboru do viacerých riadkov takto: v prvom riadku bude iba prvé slovo, v druhom ďalšie dve (oddelené medzerou), v treťom ďalšie tri, ...

- napr. ak súbor postupne obsahuje riadky so slovami `pon`, `uto`, `str`, `stv`, `pia`, `sob`, `ned`, funkcia vypíše:

```
>>> vypis_slova('slova.txt')
pon
uto str
stv pia sob
ned
```

11. Napíšte funkciu `nahodne_cisla(meno_suboru, pocet)`. Funkcia vytvorí textový súbor s trojciferný mi náhodnými číslami (zrejme z intervalu <100, 999>). V každom riadku bude jedno číslo, riadkov bude zadaný počet.

    - napr.

```
>>> nahodne_cisla('cisla.txt', 5)
... vytvoril sa 5 riadkový súbor čísel, napr.
272
598
822
927
233
```

12. Napíšte funkciu `vyrob(meno_suboru, text)`, ktorá vytvorí textový súbor s daným menom. Tento súbor bude pythonovským skriptom, ktorý 100-krát vypíše (pomocou `print()`) zadaný text

    - napr.

```
>>> vyrob('skript.py', 'Programujem v Pythone')
... vytvoril sa nový program, ten ked' teraz spustíme, výpíše 100-krát
Programujem v Pythone
Programujem v Pythone
...
```

13. Napíšte funkciu `pridaj(meno_suboru, text)`, ktorá do textového súboru **pridá na koniec** nový riadok so zadaným textom.

    - napr. ak súbor obsahoval riadky:

```
prvy riadok
druhy riadok
```

    potom volania:

```
>>> pridaj('subor.txt', 'predposledny')
>>> pridaj('subor.txt', 'posledny riadok')
```

    zmenia tento súbor:

```
prvy riadok
druhy riadok
predposledny
posledny riadok
```

14. Napíšte funkciu `vyhod_riadok(meno_suboru, index)`, ktorá z textového súboru odstráni zadaný riadok (`index` je číslo riadka od 0). Ak sa `index` rovná **-1**, funkcia vyhodí posledný riadok. Ak riadok so zadaným indexom neexistuje, funkcia nerobí nič.

- napr. pre 4-riadkový `'subor.txt'` z predchádzajúceho príkladu, volanie:

```
>>> vyhod_riadok('subor.txt', 1)
```

odstráni riadok s indexom 1, teda druhý v poradí:

```
prvy riadok
predposledny
posledny riadok
```

15. Treba otvoriť dva textové súbory a vytvoriť z nich tretí, v ktorom sa striedajú riadky z prvého a druhého súboru. Ak je niektorý zo súborov kratší, ďalej sa zapisujú len riadky z ďalšieho súboru. Napíšte funkciu `zluc_subory(meno_suboru1, meno_suboru2, meno_suboru3)`, ktorá zo `meno_suboru1` a `meno_suboru2` vytvorí `meno_suboru3`.

- napr. ak vstupné súbory obsahujú riadky `prvy subor 1`, `prvy subor 2`, ... a `druhy subor 1`, `druhy subor 2`, ... zlúčený výstupný súbor bude onbsahovať

```
prvy subor 1
druhy subor 1
prvy subor 2
druhy subor 2
prvy subor 3
druhy subor 3
...
```

16. Treba kopírovať vstupný súbor do výstupného tak, aby mal každý riadok presne zadanú šírku riadkov `sirka` znakov: dlhšie riadky sa orežú a kratšie sa sprava doplnia medzerami. Napíšte funkciu `orezat(meno_suboru1, meno_suboru2, sirka)`

- napr. pre vstupný súbor:

```
prvy riadok
druhy
    treti riadok

a posledny riadok
```

```
>>> orezat('vstup.txt', 'vystup.txt', 9)
>>> for riadok in open('vystup.txt'):
        riadok
'prvy riad'
'druhy    '
'   treti '
'         '
'a posledn'
```

17. Pre dané dva súbory funkcia `rovnake(meno_suboru1, meno_suboru2)` zistí, či tieto súbory majú identický obsah.

- napr.

```
>>> rovnake('subor11.txt', 'subor12.txt')
True
```

KAPITOLA 8

8. Zoznamy

Už poznáme tieto typy údajov:

- **jednoduché**: číselné (`int` a `float`), logické (`bool`)
- **postupnosti**: znakov (`str`), riadkov (otvorený textový súbor), čísel (pomocou `range()`)

8.1 Dátová štruktúra zoznam

- podobá sa na **pole** v iných jazykoch (napr. v Pascale je to dynamické pole, ktorého ale hodnoty musia byť rovnakého typu) - často im tak budeme hovoriť aj v Pythone
- v Pythone sa tento typ volá **list**
- je to vlastne postupnosť hodnôt ľubovoľných typov
- hovoríme, že typ zoznam sa skladá z **prvkov**
- okrem názvu **zoznam**, môžeme používať aj názov **tabuľka** alebo **pole** (väčšinou pre zoznamy hodnôt rovnakého typu)

Zoznamy vytvárame vymenovaním prvkov v hranatých zátvorkách, v príklade ich hneď aj priradíme do rôznych premenných:

```
>>> teploty = [10, 13, 15, 18, 17, 12, 12]
>>> nakup = ['chlieb', 'mlieko', 'rozky', 'jablka']
>>> studenti = ['Juraj Janosik', 'Zora Kolinska', 'Ludovit Stur', 'Janko Hrasko',
↪'Margita Figuli']
>>> zviera = ['pes', 'Dunco', 8, 35.7, 'hneda']
>>> prazdny = []                                    # prázdny zoznam
>>> print(teploty)
[10, 13, 15, 18, 17, 12, 12]
>>> type(zviera)
<class 'list'>
```

Všimnite si, že niektoré z týchto zoznamov majú všetky prvky rovnakého typu (napr. všetky sú celé čísla alebo všetky sú reťazce).

8.1.1 Operácie so zoznamami

Základné operácie so zoznamami fungujú skoro presne rovnako, ako ich vieme používať so znakovými reťazcami:

- **indexovanie** pomocou hranatých zátvoriek `[ ]` - je úplne rovnaké ako pri reťazcoch: indexom je celé číslo od 0 do počet prvkov zoznamu - 1, alebo je to záporné číslo, napr.

```
>>> zviera[0]
'pes'
>>> nakup[1]
'mlieko'
>>> studenti[-1]
'Margita Figuli'
>>> for i in range(10):
        farba = ['red', 'blue', 'yellow', 'green'][i % 4]
        print(farba)  # mohlo byt' create_oval(..., fill=farba)

red
blue
yellow
green
red
blue
yellow
green
red
blue
>>> ['red', 'blue', 'yellow'][2][4]
'o'
```

- **zreťazovanie** pomocou operácie + označuje, že vytvoríme nový väčší zoznam, ktorý bude obsahovať najprv prvky prého zoznamu a za tým všetky prvky druhého zoznamu, napr.

```
>>> nakup2 = ['zosity', 'pero', 'vreckovky']
>>> nakup + nakup2
['chlieb', 'mlieko', 'rozky', 'jablka', 'zosity', 'pero', 'vreckovky']
>>> studenti = studenti + ['Karel Capek']
>>> studenti
['Juraj Janosik', 'Zora Kolinska', 'Ludovit Stur', 'Janko Hrasko', 'Margita Figuli
↪', 'Karel Capek']
>>> prazdny + prazdny
[]
>>> [1] + [2] + [3, 4] + [] + [5]
[1, 2, 3, 4, 5]
```

- **viacnásobné zreťazenie** pomocou operácie $\star$ označuje, že daný zoznam sa navzájom zreťazí určený počet krát, napr.

```
>>> jazyky = ['Python', 'Pascal', 'C++', 'Java', 'C#']
>>> vela = 3 * jazyky
>>> vela
['Python', 'Pascal', 'C++', 'Java', 'C#', 'Python', 'Pascal', 'C++', 'Java', 'C#',
↪ 'Python',
 'Pascal', 'C++', 'Java', 'C#']
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> sto_krat_nic = 100 * [None]
>>> sto_krat_nic
[None, None, None, None, None, None, None, None, None, None, None, None, None,␣
↪None, None,
 None, None, None, None, None, None, None, None, None, None, None, None, None,␣
↪None, None,
 None, None, None, None, None, None, None, None, None, None, None, None, None,␣
↪None, None,
 None, None, None, None, None, None, None, None, None, None, None, None, None,␣
↪None, None,
 None, None, None, None, None, None, None, None, None, None, None, None, None,␣
↪None, None,
 None, None, None, None, None, None, None, None, None, None, None, None, None,␣
↪None, None,
 None, None, None, None, None, None, None, None, None, None]
>>> prazdny * 1000
[]
```

- **zisťovanie** pomocou `in` označuje, či sa nejaká hodnota nachádza v danom zozname, napr.

```
>>> 'Janko Hrasko' in studenti
True
>>> 'pero' in nakup
False
>>> 'pascal' in jazyky
False
>>> prazdny in sto_krat_nic
False
>>> teploty = [10, 13, 15, 18, 17, 12, 12]
>>> 18 in teploty
True
>>> [18, 17] in teploty
False
```

V poslednom príklade testujeme, či sa dvojprvkový zoznam `[18, 17]` nachádza niekde v zozname `teploty`. Lenže tento zoznam obsahuje len celé čísla a žiaden prvok nie je typu zoznam. Hoci pre znakové reťazce fungovalo hľadanie podreťazca, napr.

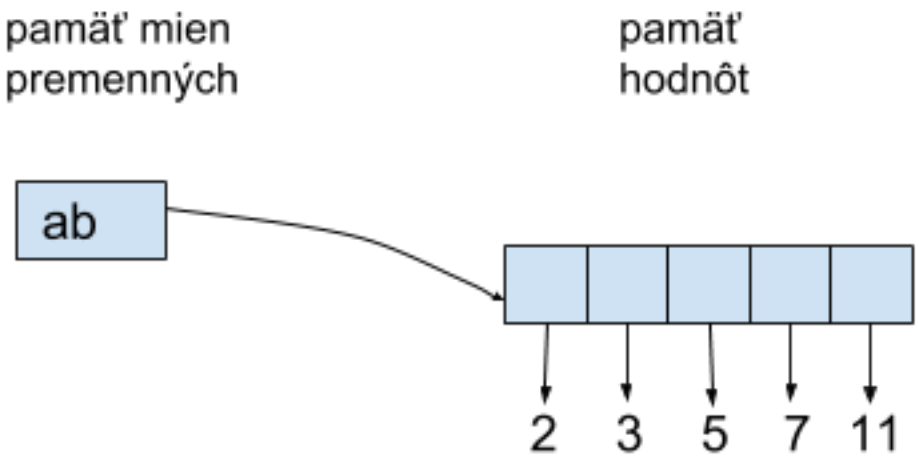
```
>>> 'th' in 'Python'
True
```

pre zoznamy táto analógia nefunguje.

Pripomeňme si, ako vyzerajú premenné a ich hodnoty v pamäti Pythonu. Urobme toto priradenie:

```
ab = [2, 3, 5, 7, 11]
```

Do pamäti mien (globálny menný priestor) pribudne jeden identifikátor premennej `ab` a tiež referencia na päťprvkový zoznam `[2, 3, 5, 7, 11]`. Tento zoznam môžeme v pamäti hodnôt vizualizovať ako päť vedľa seba položených škatuliek, pričom v každej je referencia na príslušnú hodnotu:



Je dobre si uvedomiť, že momentálne máme v pamäti 6 premenných, jedna z nich je `ab` (je typu `list`) a zvyšných päť je `ab[0]`, `ab[1]`, `ab[2]`, `ab[3]` a `ab[4]` (všetky sú typu `int`).

8.1.2 Prechádzanie prvkov zoznamu

Tzv. **iterovanie** najčastejšie pomocou for-cyklu. Napr.

```
>>> teploty
[10, 13, 15, 18, 17, 12, 12]
>>> for i in range(7):
        print(f'{i}. den {teploty[i]}')

0. den 10
1. den 13
2. den 15
3. den 18
4. den 17
5. den 12
6. den 12
```

Využili sme indexovanie prvkov zoznamu indexmi od 0 do 6. Ak nepotrebujeme pracovať s indexmi ale stačia nám samotné hodnoty, zapíšeme:

```
>>> for prvok in teploty:
        print(prvok, end=', ')

10, 13, 15, 18, 17, 12, 12,
```

Môžeme zapísať aj komplexnejší výpočet s prvkami zoznamu, napr.

```
teploty = [10, 13, 15, 18, 17, 12, 12]

sucet = 0
maximum = minimum = teploty[0]
for prvok in teploty:
    sucet += prvok
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    if prvok < minimum:
        minimum = prvok
    if prvok > maximum:
        maximum = prvok
priemer = sucet / 7
print(f'priemerna teplota je {priemer:.1f}')  # fornatovanie desatinneho cisla na␣
↪jedno miesto
print(f'minimalna teplota je {minimum}')
print(f'maximalna teplota je {maximum}')
```

```
priemerna teplota je 13.9
minimalna teplota je 10
maximalna teplota je 18
```

8.1.3 Zmena hodnoty prvku zoznamu

Dátová štruktúra zoznam je **meniteľný** typ (tzv. **mutable**) - môžeme meniť hodnoty prvkov zoznamu, napr.

```
>>> studenti[3]
'Janko Hrasko'
>>> studenti[3] = 'Jano Hraso'
>>> studenti[2] = 'Guido van Rossum'
>>> studenti
['Juraj Janosik', 'Zora Kolinska', 'Guido van Rossum', 'Jano Hraso', 'Margita Figuli',
↪ 'Karel Capek']
```

Môžeme zmeniť hodnoty prvkov zoznamu aj v cykle, ale k prvkom musíme pristupovať pomocou indexov, napr. sme zistili, že náš teplomer ukazuje o 2 stupne menej ako je reálna teplota, preto opravíme všetky prvky zoznamu:

```
teploty = [10, 13, 15, 18, 17, 12, 12]
for i in range(7):
    teploty[i] = teploty[i] + 2
print(teploty)
```

```
[12, 15, 17, 20, 19, 14, 14]
```

Krajšie by sme to zapísali s využitím štandardnej funkcie `len()`, ktorá vráti počet prvkov zoznamu:

```
teploty = [10, 13, 15, 18, 17, 12, 12]
for i in range(len(teploty)):
    teploty[i] += 2
print(teploty)
```

```
[12, 15, 17, 20, 19, 14, 14]
```

Uvedomte si, že ak by sme prvky zoznamu neindexovali, ale prechádzali sme ich priamo cez premennú cyklu `prvok`:

```
teploty = [10, 13, 15, 18, 17, 12, 12]
for prvok in teploty:
    prvok += 2
print(teploty)
```

nebude to fungovať:

```
[10, 13, 15, 18, 17, 12, 12]
```

Samotný zoznam sa tým nezmení: menili sme len obsah premennej cyklu `prvok`, ale tým sa nezmení obsah zoznamu.

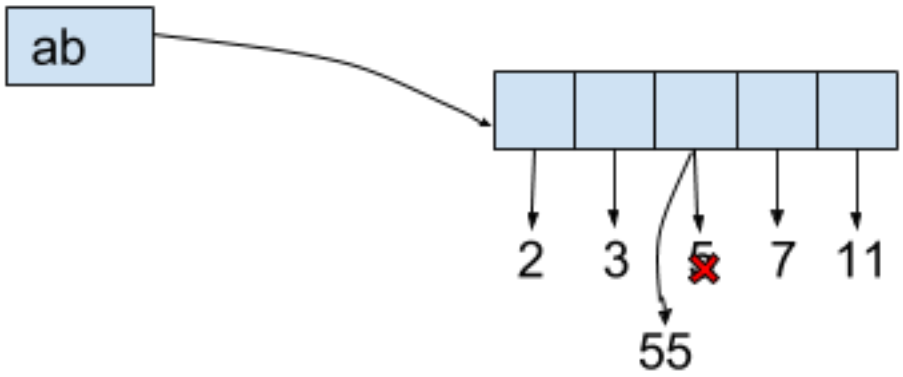
Je dobré si predstaviť, čo sa deje v pamäti pri zmene hodnoty prvku zoznamu. Zoberme si pôvodný pať prvkový zoznam prvočísel:

```
ab = [2, 3, 5, 7, 11]
```

Zmenou obsahu jedného prvku zoznamu sa zmení **jediná referencia**, všetko ostatné zostáva bez zmeny:

```
ab[2] = 55
```

dostávame:



Priraďovaním do jedného prvku zoznamu sa tento zoznam modifikuje, hovoríme, že priradenie do prvku je **mutable** operácia.

8.1.4 Štandardné funkcie so zoznamami

Nasledovné funkcie fungujú nielen so zoznamami, ale s ľubovoľnou postupnosťou hodnôt. V niektorých prípadoch však nemajú zmysel a vyhlásia chybu (napr. číselný súčet prvkov znakového reťazca).

- funkcia `len(postupnost')` -> vráti počet prvkov postupnosti
- funkcia `sum(postupnost')` -> vypočíta číselný súčet prvkov postupnosti
- funkcia `max(postupnost')` -> vráti maximálny prvok postupnosti (t.j. jeho hodnotu)
- funkcia `min(postupnost')` -> vráti minimálny prvok postupnosti

Predchádzajúci príklad, v ktorom sme počítali priemernú, minimálnu aj maximálnu teplotu, prepíšeme:

```
teploty = [10, 13, 15, 18, 17, 12, 12]

sucet = sum(teploty)
maximum = max(teploty)
minimum = min(teploty)
priemer = sucet / len(teploty)
print(f'priemerna teplota je {priemer:.1f}')
print(f'minimalna teplota je {minimum}')
print(f'maximalna teplota je {maximum}')
```

```
priemerna teplota je 13.9
minimalna teplota je 10
maximalna teplota je 18
```

Čo sa dá zapísať úspornejšie, ale menej čitateľne aj takto:

```
teploty = [10, 13, 15, 18, 17, 12, 12]

print(f'priemerna teplota je {sum(teploty) / len(teploty):.1f}')
print(f'minimalna teplota je {min(teploty)}')
print(f'maximalna teplota je {max(teploty)}')
```

8.1.5 Funkcia list()

Už máme nejaké skúsenosti s tým, že v Pythone každý základný typ má definovanú svoju **konverznú funkciu**, pomocou ktorej sa dajú niektoré hodnoty rôznych typov prekonvertovať na daný typ. Napr.

- `int(3.14)` -> vráti celé číslo `3`
- `int('37')` -> vráti celé číslo `37`
- `str(22 / 7)` -> vráti reťazec `'3.142857142857143'`
- `str(2 < 3)` -> vráti reťazec `'True'`

Podobne funguje aj funkcia `list(hodnota)`:

- parametrom musí byť **iterovateľná** hodnota, t.j. nejaká postupnosť, ktorá sa dá prechádzať (iterovať), napr. for-cyklom
- funkcia `list()` túto postupnosť rozoberie na prvky a z týchto prvkov poskladá nový zoznam
- parameter môže chýbať, vtedy vygeneruje prázdny zoznam

Napr.

```
>>> list(zviera)                              # kópia existujúceho zoznamu
['pes', 'Dunco', 8, 35.7, 'hneda']
>>> list(range(5, 16))
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
>>> list('Python')
['P', 'y', 't', 'h', 'o', 'n']
>>> list()                                    # prázdny zoznam
[]
>>> list(3.14)
...
TypeError: 'float' object is not iterable
```

Ak nejaký súbor obsahuje tieto riadky:

```
prvy
druhy
   treti riadok

a posledny
```

aj toto Python vidí ako **postupnosť** (otvorený súbor sa dá prechádzať for-cyklom ako postupnosť riadkov) a preto:

```
>>> list(open('subor.txt'))
['prvy\n', 'druhy\n', '   treti riadok\n', '\n', 'a posledny\n']
```

8.1.6 Rezy

Keď sme v znakovom reťazci potrebovali zmeniť nejaký znak, zakaždým sme museli vyrobiť kópiu reťazca, napr

```
>>> retazec = 'Monty Python'
>>> retazec[4] = 'X'                    # takto sa to nedá
...
TypeError: 'str' object does not support item assignment
>>> retazec = retazec[:4] + 'X' + retazec[5:]
>>> retazec
'MontX Python'
```

Využili sme tu **rezy** (slice), t.j. získavanie podreťazcov. To isté sa dá použiť aj pri práci so zoznamami, lebo aj s nimi fungujú rezy, napr.

```
>>> jazyky
['Python', 'Pascal', 'C++', 'Java', 'C#']
>>> jazyky[1:3]
['Pascal', 'C++']
>>> jazyky[-3:]
['C++', 'Java', 'C#']
>>> jazyky[:-1]
['Python', 'Pascal', 'C++', 'Java']
```

Samozrejme, že pritom funguje aj určovanie kroku, napr.

```
>>> jazyky[1::2]
['Pascal', 'Java']
>>> jazyky[::-1]
['C#', 'Java', 'C++', 'Pascal', 'Python']
```

Uvedomte si, že takéto rezy nemenia obsah samotného zoznamu a preto hovoríme, že sú **immutable**.

8.1.7 Priraďovanie do rezu

Keď iba vyberáme nejaký podzoznam pomocou rezu, napr. `zoznam[od:do:krok]`, takáto operácia s pôvodným zoznamom nič nerobí (len vyrobí úplne nový zoznam). Lenže my môžeme obsah zoznamu meniť aj takým spôsobom, že zmeníme len jeho nejakú časť. Takže rez zoznamu môže byť na ľavej strane priraďovacieho príkazu a potom na pravej strane priraďovacieho príkazu musí byť nejaká **postupnosť** (nemusí to byť zoznam). Priraďovací príkaz teraz túto postupnosť prejde, zostrojí z nej zoznam a ten vloží namiesto udaného rezu.

Preštudujte nasledovné príklady:

```
>>> zoz = list(range(0, 110, 10))
>>> zoz
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
>>> zoz[3:6] = ['begin', 'end']                    # tri prvky sa nahradili dvoma
>>> zoz
[0, 10, 20, 'begin', 'end', 60, 70, 80, 90, 100]
```

```
>>> zoz[6:7] = [111, 222, 333]                    # jeden prvok sa nahradil tromi
>>> zoz
[0, 10, 20, 'begin', 'end', 60, 111, 222, 333, 80, 90, 100]
```

```
>>> abc = list('Python')
>>> abc
['P', 'y', 't', 'h', 'o', 'n']
>>> abc[2:2]                                      # rez dĺžky 0
[]
>>> abc[2:2] = ['dve', 'slova']                   # rez dĺžky 0 sa nahradí dvomi␣
↪prvkami
>>> abc
['P', 'y', 'dve', 'slova', 't', 'h', 'o', 'n']
```

```
>>> prvo = [2, 3, 5, 7, 11]
>>> prvo[1:-1]
[3, 5, 7]
>>> prvo[1:-1] = []                               # rez dĺžky 3 sa nahradí žiadnymi␣
↪prvkami
>>> prvo                                          # prvky sa takto vyhodili
[2, 11]
```

Pozor! Všetky tieto príklady modifikujú pôvodný zoznam, teda priraďovanie do rezu je **mutable operácia**.

8.1.8 Porovnávanie zoznamov

Zoznamy môžeme navzájom porovnávať (na rovnosť, alebo menší/väčší). Funguje to na rovnakom princípe ako porovnávanie znakových reťazcov:

- postupne sa prechádzajú prvky jedného aj druhého zoznamu, kým sú rovnaké
- ak je jeden so zoznamov kratší, tak ten sa považuje za menší ako ten druhý
- ak pri postupnom porovnávaní prvkov nájde rôzne hodnoty, výsledok porovnania týchto dvoch rôznych hodnôt je výsledkom porovnania celých zoznamov
- každé dva porovnávané prvky musí Python vedieť porovnať: na rovnosť je to bez problémov, ale relačné operácie < a > nebudú fungovať napr. pre porovnávanie čísel a reťazcov

Napr.

```
>>> [1, 2, 5, 3, 4] > [1, 2, 4, 8, 1000]
True
>>> [1000, 2000, 3000] < [1000, 2000, 3000, 0, 0]
True
>>> [1, 'ahoj'] == ['ahoj', 1]
False
>>> [1, 'ahoj'] < ['ahoj', 1]
...
TypeError: '<' not supported between instances of 'int' and 'str'
```

8.1.9 Zoznam ako parameter funkcie

Už predtým sme zisťovali priemernú teplotu zo zoznamu nameraných hodnôt. Teraz z toho vieme urobiť funkciu, napr.

```
def priemer(zoznam):
    sucet = 0
    pocet = 1
    for prvok in zoznam:
        sucet += prvok
        pocet += 1
    return sucet / pocet
```

Keďže pomocou štandardných funkcií `sum()` a `len()` vieme veľmi rýchlo zistiť súčet aj počet prvkov zoznamu, môžeme to zapísať elegantnejšie:

```
def priemer(zoznam):
    return sum(zoznam) / len(zoznam)
```

Ďalšia funkcia zisťuje počet výskytov nejakej konkrétnej hodnoty v zozname:

```
def pocet(zoznam, hodnota):
    vysl = 0
    for prvok in zoznam:
        if prvok == hodnota:
            vysl += 1
    return vysl
```

a naozaj funguje:

```
>>> pocet([1, 2, 3, 2, 1, 2], 4)
0
>>> pocet([1, 2, 3, 2, 1, 2], 2)
3
```

Zaujímavé je aj to, že funkcia funguje nielen pre zoznamy, ale pre ľubovoľnú postupnosť (iterovateľnú štruktúru), napr.

```
>>> pocet('bla-bla-bla', 'l')
3
>>> pocet('bla-bla-bla', 'la')      # v postupnosti znakov sa 'la' nenachádza
0
```

Pre znaky táto funkcia robí skoro to isté ako **metóda** `count()`, teda zápis:

```
>>> 'bla-bla-bla'.count('l')
3
```

naša funguje úplne rovnako ako funkcia `pocet()`. Aj pre zoznamy existuje metóda `count`, ktorá robí presne to isté ako naša funkcia `pocet()`.

8.2 Metódy

Už vieme, že metódami voláme také funkcie, ktoré fungujú s nejakou hodnotou: za túto hodnotu dávame bodku (preto tzv. **bodková notácia**) a samotné meno funkcie s prípadnými parametrami. V prípade zoznamov to vyzerá takto:

```
zoznam.funkcia(parametre)
```

Pre zoznamy existujú tieto dve metódy, ktoré nemodifikujú ich obsah a preto vieme, že sú **immutable**.

metóda `count()`

Volanie metódy:

```
zoznam.count(hodnota)
```

vráti počet výskytov danej hodnoty v zozname. Táto metóda je **immutable** lebo nemení obsah zoznamu.

Metódu môžeme použiť nielen s premennou typu zoznam, napr.

```
>>> zoz = [1, 2, 3, 2, 1, 2]
>>> zoz.count(2)
3
>>> zoz.count(4)
0
```

Ale aj priamo s hodnotou zoznam, napr.

```
>>> [0, 1, 0, 0, 1, 0, 1, 0, 0].count(1)
3
```

Alebo s výrazom, ktorý je typu zoznam:

```
>>> ([3, 7] * 100 + [7, 8] * 50).count(7)
150
```

metóda `index()`

Volanie metódy:

```
zoznam.index(hodnota)
```

vráti index prvého výskytu danej hodnoty v zozname. Táto metóda je **immutable** lebo nemení obsah zoznamu. Funkcia spadne na chybe, ak sa daná hodnota v zozname nenachádza.

Aj túto metódu môžeme použiť rôznym spôsobom, napr. s premennou typu zoznam:

```
>>> farby = ['red', 'blue', 'red', 'blue', 'yellow']
>>> farby.index('blue')
1
>>> farby.index('green')
...
ValueError: 'green' is not in list
```

Pri používaní tejto metódy musíme dávať pozor, aby nám program nepadal, keď sa daná hodnota v zozname nenachádza. Napr. takto:

```
>>> farby = ['red', 'blue', 'red', 'blue', 'yellow']
>>> if 'green' in farby:
        index = farby.index('green')
    else:
        print('green sa v zozname nenachadza')
```

```
green sa v zozname nenachadza
```

Všetky ďalšie metódy, ktoré tu uvedieme, sú **mutable**, teda budú modifikovať samotný zoznam.

metóda `append()`

Volanie metódy:

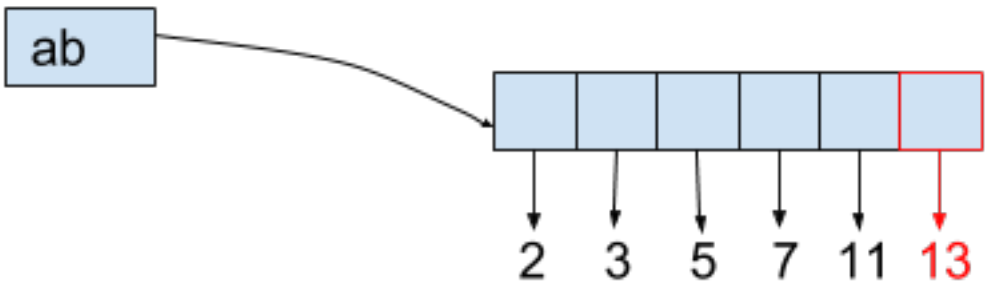
```
zoznam.append(hodnota)
```

pridá na koniec zoznamu nový prvok - zoznam sa takto predĺži o 1. Táto metóda je **mutable** lebo mení obsah zoznamu. Funkcia nič nevracia, preto nemá zmysel priraďovať jej volanie do nejakej premennej (teda vracia hodnotu `None`).

Napr. volanie:

```
>>> ab = [2, 3, 5, 7, 11]
>>> ab.append(13)
>>> ab
[2, 3, 5, 7, 11, 13]
```

takto sa zmení vizualizácia pamäte pre premennú `ab`:



Zrejme nemá zmysel volať túto metódu s hodnotou typu zoznam namiesto premennej. Hoci to funguje dobre, nemáme šancu zistiť, ako vyzerá daný zoznam s pridaným prvkom:

```
>>> [1, 2, 3].append(4)
```

Niekde v pamäti hodnôt sa vyrobil zoznam `[1, 2, 3, 4]`, na ktorý ale nemáme žiadnu referenciu.

metóda `pop()`

Volanie metódy:

```
zoznam.pop()
```

odoberie z konca zoznamu posledný prvok - zoznam sa takto skráti o 1. Táto metóda je **mutable** lebo mení obsah zoznamu. Funkcia vracia hodnotu odobratého prvku. Ak bol zoznam prázdny, funkcia nič nevracia ale spadne na chybe.

Napr.

```
>>> abc = ['raz', 'dva', 'tri']
>>> for i in range(4):
        abc.pop()
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
'tri'
'dva'
'raz'
...
IndexError: pop from empty list
```

metóda `insert()`

Volanie metódy:

```
zoznam.insert(index, hodnota)
```

pridá na dané miesto zoznamu nový prvok - zoznam sa takto predĺži o 1. Táto metóda je **mutable** lebo mení obsah zoznamu. Funkcia nič nevracia, preto nemá zmysel priraďovať jej volanie do nejakej premennej (teda vracia hodnotu `None`).

Napr.

```
>>> abc = ['raz', 'dva', 'tri']
>>> abc.insert(10, 'koniec')
>>> abc
['raz', 'dva', 'tri', 'koniec']
>>> abc.insert(2, 'stred')
>>> abc
['raz', 'dva', 'stred', 'tri', 'koniec']
>>> abc.insert(0, 'zaciatok')
>>> abc
['zaciatok', 'raz', 'dva', 'stred', 'tri', 'koniec']
>>> abc.insert(-1, 'predposledny')
>>> abc
['zaciatok', 'raz', 'dva', 'stred', 'tri', 'predposledny', 'koniec']
```

Uvedomte si, že `zoznam.insert(len(zoznam), hodnota)` pridáva vždy na koniec zoznamu, teda robí to isté ako `zoznam.append(hodnota)`.

metóda `pop()` s indexom

Volanie metódy:

```
zoznam.pop(index)
```

odoberie zo zoznamu príslušný prvok (daný indexom) - zoznam sa takto skráti o 1. Táto metóda je **mutable** lebo mení obsah zoznamu. Funkcia vracia hodnotu odobratého prvku. Ak bol zoznam prázdny, funkcia nič nevracia ale spadne na chybe.

Napr.

```
>>> abc = ['raz', 'dva', 'tri', 'styri']
>>> abc.pop(7)
...
IndexError: pop index out of range
>>> abc.pop(-1)                    # to isté ako abc.pop()
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
'styri'
>>> abc.pop(0)                          # vyhadzuje prvý prvok
'raz'
>>> abc
['dva', 'tri']
>>> abc.pop(1)
'tri'
>>> abc.pop(0)
'dva'
>>> abc.pop(0)
...
IndexError: pop from empty list
>>> abc
[]
```

Posledné volanie metódy `pop()` sa snaží vybrať prvý prvok z prázdneho zoznamu - spôsobilo to vyvolanie správy o chybe.

metóda `remove()`

Volanie metódy:

```
zoznam.remove(hodnota)
```

odoberie zo zoznamu prvý výskyt prvku s danou hodnotou - zoznam sa takto skráti o 1. Táto metóda je **mutable** lebo mení obsah zoznamu. Funkcia nič nevracia (teda vracia `None`). Ak sa daná hodnota v zozname nenachádza, funkcia spadne na chybe.

Napr.

```
>>> abc = ['raz', 'dva', 'tri', 'dva']
>>> abc.remove('dva')
>>> abc
['raz', 'tri', 'dva']
>>> abc.remove('styri')
...
ValueError: list.remove(x): x not in list
```

metóda `sort()`

Volanie metódy:

```
zoznam.sort()
```

zmení poradie prvkov zoznamu tak, aby boli **usporiadané vzostupne** - zoznam takto nemení svoju dĺžku. Táto metóda je **mutable** lebo mení obsah zoznamu. Funkcia nič nevracia (teda vracia `None`). Ak sa prvky v zozname nedajú navzájom porovnávať (napr. sú tam čísla aj reťazce), funkcia spadne na chybe.

Napr.

```
>>> abc = ['raz', 'dva', 'tri', 'styri']
>>> abc.sort()
>>> abc
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
['dva', 'raz', 'styri', 'tri']
>>> post = list(reversed(range(10)))
>>> post
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> post.sort()
>>> post
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Častou začiatočníckou chybou býva priradenie výsledku tejto metódy (teda `None`) do premennej, napr.

```
>>> abc = ['raz', 'dva', 'tri', 'styri']
>>> abc = abc.sort()                        # vždy vráti None
>>> print(abc)
None
```

Takto si pokazíme referenciu na pekne utriedený zoznam.

8.2.1 Zoznam ako výsledok funkcie

Prvá funkcia vráti novo vytvorený zoznam rovnakých hodnôt:

```
def urob_zoznam(n, hodnota=0):
    return [hodnota] * n
```

Môžeme to použiť napr. takto (premenné sme nazvali `pole`, lebo sa to trochu podobá na pascalovské polia):

```
>>> pole1 = urob_zoznam(30)
>>> pole2 = urob_zoznam(25, None)
>>> pole3 = urob_zoznam(3, 'Python')
>>> pole1
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> pole2
[None, None, None, None, None, None, None, None, None, None,
 None, None, None, None, None, None, None, None, None, None,
 None, None, None, None, None]
>>> pole3
['Python', 'Python', 'Python']
```

Ďalšia funkcia vytvorí z danej postupnosti nový zoznam:

```
def zoznam(postupnost):
    vysl = []
    for prvok in postupnost:
        vysl.append(prvok)
    return vysl
```

Uvedomte si, že to robí skoro to isté ako volanie funkcie `list()`:

```
>>> py = zoznam('Python')
>>> py
['P', 'y', 't', 'h', 'o', 'n']
>>> cisla = zoznam(range(3, 25, 4))
>>> cisla
[3, 7, 11, 15, 19, 23]
```

Funkcia `pridaj()` na základe nejakého zoznamu vytvorí nový zoznam, na koniec ktorého pridá nový prvok:

```
def pridaj(zoznam, hodnota):
    return zoznam + [hodnota]
```

Všimnite si, že pôvodný zoznam pritom ostal nezmenený:

```
>>> zoz = ['raz', 'dva', 'tri']
>>> novy = pridaj(zoz, 'styri')
>>> novy
['raz', 'dva', 'tri', 'styri']
>>> zoz
['raz', 'dva', 'tri']
```

Takejto funkcii budeme hovoriť, že je **immutable**, lebo nemení hodnotu žiadneho predtým existujúceho zoznamu.

Ak by sme túto funkciu zapísali takto:

```
def pridaj1(zoznam, hodnota):
    zoznam.append(hodnota)
    return zoznam
```

Volaním tejto funkcie by sme dostali veľmi podobné výsledky:

```
>>> zoz = ['raz', 'dva', 'tri']
>>> novy = pridaj1(zoz, 'styri')
>>> novy
['raz', 'dva', 'tri', 'styri']
>>> zoz
['raz', 'dva', 'tri', 'styri']
```

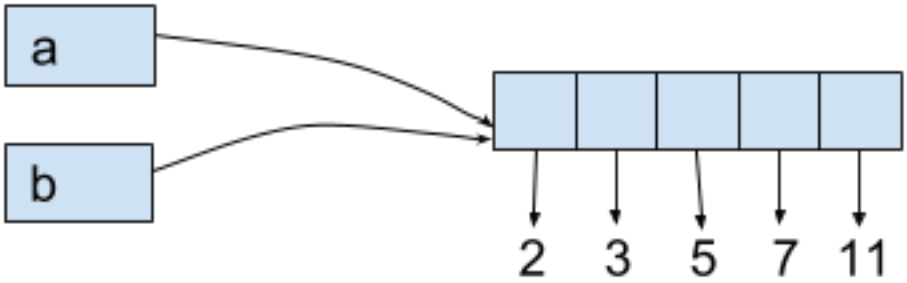
Ale zmenila sa pritom aj hodnota prvého parametra - premennej `zoz` typu zoznam. Táto funkcia je **mutable** - mení svoj parameter `zoz` typu `list`.

8.2.2 Dve premenné referencujú na ten istý zoznam

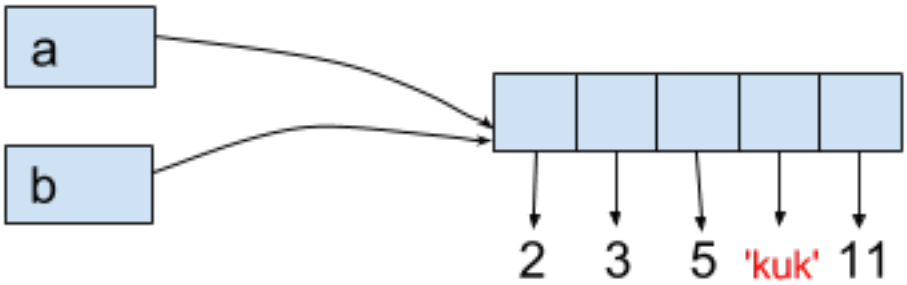
Už sme získali predstavu o tom, že priradenie zoznamu do premennej označuje, že sme v skutočnosti do premennej priradili referenciu na zoznam. Lenže na ten istý zoznam v pamäti môžeme mať viac referencií, napr.

```
>>> a = [2, 3, 5, 7, 11]
>>> b = a
>>> b[3] = 'kuk'
>>> a
[2, 3, 5, 'kuk', 11]
```

Menili sme obsah premennej `b` (zmenili sme jej prvok s indexom 3), ale tým sa zmenil aj obsah premennej `a`. Totiž obe premenné referencujú na ten istý zoznam:



Keď teraz meníme obsah premennej `b` (ale len pomocou **mutable** operácií!), zmení sa aj obsah premennej `a`:



8.2.3 Zhrňme

Vkladanie do zoznamu

Videli sme viac rôznych spôsobov, ako môžeme pridať jednu hodnotu do zoznamu. Vkladanie nejakej `hodnoty` pred prvok s indexom `i`:

- pomocou rezu (**mutable**):

```
zoznam[i:i] = [hodnota]
```

- pomocou metódy `insert()` (**mutable**):

```
zoznam.insert(i, hodnota)
```

- ak `i == len(zoznam)`, pridávame na koniec (za posledný prvok), môžeme použiť metódu `append()` (**mutable**):

```
zoznam.append(hodnota)
```

  to isté dosiahneme aj takto (**mutable**):

```
zoznam += [hodnota]
```

Vo vašich programoch použijete ten zápis, ktorý sa vám bude najlepšie hodiť, ale zápis s rezom `zoznam[i:i]` je najmenej čitateľný a používa sa veľmi zriedkavo.

- zrejme funguje aj (**immutable**):

```
zoznam = zoznam[:i] + [hodnota] + zoznam[i:]
```

toto priradenie nemodifikuje pôvodný zoznam, ale vytvára nový s pridanou hodnotou

Vyhadzovanie zo zoznamu

Aj vyhadzovanie prvku zo zoznamu môžeme robiť viacerými spôsobmi. Ak vyhadzujeme prvok na indexe `i`, môžeme zapísať:

- pomocou rezu (**mutable**):

```
zoznam[i:i+1] = []
```

- pomocou príkazu `del` (**mutable**):

```
del zoznam[i]
```

- pomocou metódy `pop()`, ktorá nám aj vráti vyhadzovanú hodnotu (**mutable**):

```
hodnota = zoznam.pop(i)
```

- veľmi neefektívne pomocou metódy `remove()`, ktorá ako parameter očakáva nie index ale vyhadzovanú hodnotu (**mutable**):

```
zoznam.remove(zoznam[i])
```

  tento spôsob je veľmi neefektívny (zbytočne sa hľadá prvok v zozname) a okrem toho niekedy môže vyhodiť nie `i`-ty prvok, ale prvok s rovnakou hodnotou, ktorý sa v zozname nachádza skôr ako na indexe `i`.

- zrejme funguje aj (**immutable**):

```
zoznam = zoznam[:i] + zoznam[i+1:]
```

  toto priradenie nemodifikuje pôvodný zoznam, ale vytvára nový bez prvku s daným indexom

Vyhodenie všetkých prvkov zo zoznamu

- najjednoduchší spôsob (**immutable**):

```
zoznam = []
```

  môžeme použiť len vtedy, keď nepotrebujeme uchovať referenciu na zoznam - toto priradenie nahradí momentálnu referenciu na zoznam referenciou na úplne nový zoznam; ak to použijeme vo vnútri funkcie, stratí sa tým referencia na pôvodný zoznam

Ďalšie spôsoby uchovávajú referenciu na zoznam:

- všetky prvky zoznamu postupne vyhodíme pomocou while-cyklu (**mutable**):

```
while zoznam:
    zoznam.pop()
```

  toto je zbytočne veľmi neefektívne riešenie

- priradením do rezu (**mutable**):

```
zoznam[:] = []
```

  je ťažšie čitateľné a menej pochopiteľné riešenie

- metódou `clear()` (**mutable**):

```
zoznam.clear()
```

je asi najčitateľnejší zápis

Vytvorenie kópie zoznamu

Ak potrebujeme vyrobiť kópiu celého zoznamu, dá sa to urobiť:

- pomocou cyklu:

```
kopia = []
for prvok in zoznam:
    kopia.append(prvok)
```

- môžeme využiť aj rez:

```
kopia = zoznam[:]
```

- keďže funguje funkcia `list()`, môžeme zapísať:

```
kopia = list(zoznam)
```

8.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/
- v riešeniach úloh používajte len konštrukcie a funkcie, ktoré sme sa učili na doterajších prednáškach
- pozrite si *Riešenie 8. cvičenia*

1. Napíšte funkciu `sucin(zoznam)`, ktorá vráti súčin prvkov zoznamu (obsahuje len čísla). Ak je zoznam prázdny, funkcia vráti `1`.
   - napr.

```
>>> c = [2, 3, 5, 7, 11]
>>> sucin(c)                    # co je 2*3*5*7*11
2310
>>> sucin(list(range(1, 11)))   # co je 10 faktorial
3628800
>>> sucin([2] * 20)             # co je 2 ** 20
1048576
```

2. Napíšte funkciu `mocniny(n)`, ktorá vráti `n` prvkový zoznam druhých mocnín celých čísel `[1, 4, 9, 16, 25, ..., n**2]`
   - napr.

```
>>> x = mocniny(7)
>>> x
[1, 4, 9, 16, 25, 36, 49]
```

3. Napíšte funkciu `len_parne(zoznam)`, ktorá z daného zoznamu celých čísel, vyrobí nový, ale ponechá v ňom len párne hodnoty

   - napr.

```
>>> print(len_parne([2, 5, 7, 10, 13]))
[2, 10]
>>> print(len_parne([1, 3, 5, 7, 9]))
[]
```

4. Napíšte funkciu `spoj(zoznam)`, ktorá pre daný zoznam slov (znakových reťazcov) vytvorí jeden znakový reťazec zlepením všetkých slov v zozname. Slová v tomto výslednom reťazci budú oddelené medzerami. Nepoužite metódu `join()`.

   - napr.

```
>>> slova = ['nepi', 'Jano', 'nepi', 'vodu']
>>> veta = spoj(slova)
>>> veta
'nepi Jano nepi vodu'
```

5. Funkcia `zisti(zoznam)` zistí počet celých čísel v zadanom zozname, ktoré sú deliteľné 7

   - napr.

```
>>> print('pocet =', zisti([4, 7.0, 14, '7', 0]))
pocet = 2
```

   - to, či je nejaká hodnota celým číslom, môžete zistiť testom `type(hodnota) == int`

6. Funkcia `zoznam2(n, hodn1, hodn2)` vytvorí `n` prvkový zoznam (predpokladajte, že `n` je párne), ktorý bude obsahovať striedajúce sa hodnoty `hodn1` a `hodn2`

   - napr.

```
>>> s = zoznam2(10, 7, 'w')
>>> s
[7, 'w', 7, 'w', 7, 'w', 7, 'w', 7, 'w']
```

7. Funkcia `na_parnych(zoznam)` z daného zoznamu vytvorí nový, ktorý obsahuje len prvky na párnych indexoch (funkcia nemodifikuje vstupný zoznam)

   - napr.

```
>>> a = list('programovanie')
>>> b = na_parnych(a)
>>> b
['p', 'o', 'r', 'm', 'v', 'n', 'e']
```

8. Funkcia `zostupne(zoznam)` zistí, či je daný `zoznam` utriedený **zostupne** (každý ďalší prvok v zozname nie je väčší ako predchádzajúci). Funkcia vráti `True` alebo `False` (funkcia nemodifikuje vstupný zoznam). Nepoužite štandardné funkcie ani metódy.

   - napr.

```
>>> zostupne([5, 3, 3, 2, 0, 0])
True
>>> p = [1, 2, 3]
>>> zostupne(p)
False
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> p
[1, 2, 3]
```

9. Funkcia `vyrob1(zoznam)` vyrobí (vráti) kópiu celočíselného zoznamu, ale každé párne číslo pritom zväčší o 1 (funkcia nemodifikuje vstupný zoznam)

   - napr.

```
>>> zoz1 = [3, 5, 6, 8, 9, 10, 11, 13]
>>> zoz2 = vyrob1(zoz1)
>>> zoz2
[3, 5, 7, 9, 9, 11, 11, 13]
>>> zoz1
[3, 5, 6, 8, 9, 10, 11, 13]
```

10. Funkcia `vyrob2(zoznam)` vyrobí kópiu vstupného zoznamu ale v kópii ponechá len tie prvky, ktoré sú znakové reťazce (funkcia nemodifikuje vstupný zoznam)

    - napr.

```
>>> zoz1 = [1, 2.2, ['a', 'b'], 'tri', 4, '']
>>> zoz2 = vyrob2(zoz1)
>>> zoz2
['tri', '']
>>> zoz1
[1, 2.2, ['a', 'b'], 'tri', 4, '']
```

11. Funkcia `zoznam_cifier(cislo)` z daného nezáporného celého čísla vytvorí (vráti) zoznam cifier

    - napr.

```
>>> c = zoznam_cifier(123789)
>>> c
[1, 2, 3, 7, 8, 9]
>>> zoznam_cifier(0)
[0]
```

12. Funkcia `gener(a, b, c=1)` vytvorí (vráti) zoznam, ktorého prvky sú celočíselné hodnoty od `a` do `b-1` krokom `c` (rovnako ako `range(a, b, c)`). Nepoužite pritom štandardnú funkciu `range()`.

    - napr.

```
>>> aa = gener(1, 11)
>>> aa
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> gener(5, 20, 3)
[5, 8, 11, 14, 17]
>>> gener(4, 0, -1)
[4, 3, 2, 1]
```

13. Funkcia `cele(zoznam)` zo zoznamu desatinných čísel vyrobí (vráti) zoznam celých čísel - ich celých častí (volaním funkcie `int()`). Funkcia nemodifikuje vstupný zoznam.

    - napr.

```
>>> d = [3.14, -7.0, 0.99]
>>> a = cele(d)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> a
[3, -7, 0]
```

14. Napíšte funkciu `vymen(zoznam)`, ktorá dostane **dvojprvkový zoznam** a vráti nový zoznam, ktorý má tieto prvky vymenené
    - napr.

```
>>> dvojica = ['ahoj', 123]
>>> novy = vymen(dvojica)
>>> novy
[123, 'ahoj']
```

15. Napíšte funkciu `vymen2(zoznam)`, ktorá dostane **dvojprvkový zoznam** a navzájom vymení jeho prvky (bude **mutable**). Funkcia nič nevracia.
    - napr.

```
>>> dvojica = ['ahoj', 123]
>>> vymen(dvojica)
>>> dvojica
[123, 'ahoj']
```

16. Napíšte funkciu `vyhod(zoznam, hodnota)`, ktorá z pôvodného zoznamu vytvorí nový ale už bez prvkov s danou hodnotou.
    - napr.

```
>>> zoz = [37, 'hello', -7, 3.14, 'hello', 2]
>>> novy = vyhod(zoz, 'hello')
>>> novy
[37, -7, 3.14, 2]
>>> zoz
[37, 'hello', -7, 3.14, 'hello', 2]
```

17. Napíšte funkciu `vyhod2(zoznam, hodnota)`, ktorá v danom zozname vyhodí všetky výskyty danej hodnoty. Funkcia nič nevracia, ale zmení obsah zoznamu (**mutable**).
    - napr.

```
>>> zoz = [37, 'hello', -7, 3.14, 'hello', 2]
>>> vyhod2(zoz, 'hello')
>>> zoz
[37, -7, 3.14, 2]
```

18. Predpokladáme, že nejaký textový súbor má v každom riadku 1 alebo 2 celé čísla. Napíšte funkciu `zo_suboru(meno_suboru)`, ktorá prečíta tento súbor a vráti takýto zoznam: z každého riadku súboru vytvorí jeden prvok výsledného zoznamu, pričom, ak bolo v riadku len jedno číslo, toto bude priamo prvkom zoznamu, ak tam boli dve čísla, vytvorí dvojprvkový zoznam čísel
    - napr. ak súbor `'subor.txt'` obsahuje

```
100 200
300
400
500 600
```

    - dostaneme štvorprvkový zoznam

```
>>> vysl = zo_suboru('subor.txt')
>>> vysl
[[100, 200], 300, 400, [500, 600]]
```

19. Napíšte funkciu `do_suboru(meno_suboru, zoznam)`, ktorá zapíše do súboru daný zoznam. Každý prvok vstupného zoznamu bude zapísaný do samostatného riadku. Predpokladáme, že prvkami zoznamu môžu byť buď čísla, alebo dvojprvkové zoznamy čísel. Ak je prvkom číslo, v riadku súboru bude priamo toto číslo inak bude riadok obsahovať dve čísla oddelené medzerou (rovnaký formát súboru ako bol v predchádzajúcom príklade)

    - napr.

```
>>> xy = [[100, 200], 300, 400, [500, 600]]
>>> do_suboru('subor1.txt', xy)
```

    - vytvorí súbor

```
100 200
300
400
500 600
```

20. Funkcia `krat2(zoznam)` vynásobí každý prvok zoznamu číslom 2; funkcia nič nevracia len mení obsah zoznamu (**mutable**)

    - napr.

```
>>> z = [5, 3.14, [1, 2], -4, 'ab']
>>> krat2(z)
>>> z
[10, 6.28, [1, 2, 1, 2], -8, 'abab']
```

21. Funkcia `zdvoj(zoznam)` do daného zoznamu pridá nové hodnoty tak, že každý prvok z pôvodného obsahu sa tu objaví dvakrát za sebou, funkcia nič nevracia, len modifikuje daný zoznam (**mutable**)

    - napr.

```
>>> tab = [1, 'Python', 2, 'Java', 3, 'C#']
>>> zdvoj(tab)
>>> tab
[1, 1, 'Python', 'Python', 2, 2, 'Java', 'Java', 3, 3, 'C#', 'C#']
```

22. Prvkami zoznamu sú čísla 0, 1 alebo 2 (v ľubovoľnom poradí). Napíšte funkciu `uprac(zoznam)`, ktorá preusporiada prvky zoznamu tak, že na začiatku zoznamu budú všetky 2, za tým 0 a na koniec 1, funkcia nič nevracia, len modifikuje daný zoznam (**mutable**)

    - napr.

```
>>> f = [0, 1, 2, 0, 0, 2]
>>> uprac(f)
>>> f
[2, 2, 0, 0, 0, 1]
```

23. Funkcia `nahodny_zoznam(n, vyber)` vyrobí (vráti pomocou `return`) `n` prvkový zoznam, ktorého prvky sú náhodne vybrané hodnoty zo zoznamu `vyber`

    - napr.

```
>>> m = nahodny_zoznam(8, [7, 'red', None])
>>> m
['red', 7, 'red', 7, 'red', None, None, 'red']
>>> nahodny_zoznam(13, [2, 3])
[2, 3, 2, 2, 3, 2, 3, 3, 3, 2, 3, 2, 3]
```

8.4 3. Domáce zadanie

L.I.S.T.

- riešenie odovzdávajte na úlohový server https://list.fmph.uniba.sk/

Napíšte pythonovský skript, ktorý bude definovať túto funkciu:

```
vypis(meno_suboru, sirka)
```

Funkcia `vypis()` dostáva ako prvý parameter meno textového súboru a dryhým parametrom je celé číslo, ktoré udáva šírku výpisu. Funkcia tento súbor prečíta a celý ho vypíše do textovej plochy (do konzoly) tak, že bude zarovnaný na danú šírku.

Textový súbor sa skladá z **odsekov**, ktoré sa skladajú zo **slov**. Odseky sú navzájom oddelené aspoň jedným púrázdnym riadkom. Slová v odesku sú navzákom oddelené aspoň jednou medzerou alebo koncom riadka.

Napr. `"subor1.txt"` sa skladá z týchto riadkov:

```
Ján Botto:
  Žltá l'alija


Stojí, stojí mohyla.

Na mohyle   zlá    chvíl'a,
na mohyle tŕnie,   chrastie
 a v tom tŕní, chrastí rastie,
  rastie, kvety rozvíja
jedna   žltá   l'alija.


  Tá l'alija smutno vzdychá:

  Hlávku moju tŕnie pichá
 a  nožičky  oheň  páli --
pomôžte mi v mojom žiali!
```

Tento súbor obsahuje 5 „odsekov“, pričom najkratší je druhý a má 3 „slová“. Najdlhší je tretí odsek má 20 slov.

Volanie `vypis('subor1.txt', 20)` vypíše:

```
Ján    Botto:   Žltá
l'alija

Stojí, stojí mohyla.

Na     mohyle    zlá
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
chvíľa,   na  mohyle
tŕnie,  chrastie a v
tom   tŕní,  chrastí
rastie,      rastie,
kvety  rozvíja jedna
žltá ľalija.

Tá   ľalija   smutno
vzdychá:

Hlávku   moju  tŕnie
pichá a nožičky oheň
páli  -- pomôžte mi v
mojom žiali!
```

Pričom `vypis('subor1.txt', 60)` vypíše:

```
Ján Botto: Žltá ľalija

Stojí, stojí mohyla.

Na  mohyle  zlá  chvíľa,  na  mohyle tŕnie, chrastie a v tom
tŕní,  chrastí  rastie,  rastie,  kvety  rozvíja  jedna žltá
ľalija.

Tá ľalija smutno vzdychá:

Hlávku  moju  tŕnie pichá a nožičky oheň páli -- pomôžte mi v
mojom žiali!
```

Všimnite si, že všetky riadky v odseku okrem posledného sú zarovnané vpravo ma zadanú šírku, pričom ak by bola dĺžka takéhoto riadka kratšia ako zadaná šírka, medzi slová sú rovnomerne vložené medzery. Ak nejaký riadok obsahuje len jedno slovo, tak ani tento sa nezarovnáva na pravý okraj.

Váš odovzdaný program s menom `riesenie3.py` musí začínať tromi riadkami komentárov:

```
# 3. zadanie: zarovnaj
# autor: Janko Hraško
# datum: 5.11.2017
```

Projekt `riesenie3.py` odovzdávajte na úlohový server https://list.fmph.uniba.sk/ najneskôr do 23:00 **19. novembra**, kde ho môžete nechať otestovať. Testovač bude spúšťať vašu funkciu s rôznymi textovými súbormi, ktoré si môžete stiahnuť z L.I.S.T.u. Odovzdať projekt aj ho testovať môžete ľubovoľný počet krát. Môžete zaň získať **10 bodov**.

KAPITOLA 9

9. Zoznamy a n-tice (tuple)

9.1 Používanie zoznamov

Na minulej prednáške sme videli, že v Pythone je veľmi jednoduché vyrobiť dve premenné, ktoré referencujú na jeden a ten istý zoznam. Potom každá (**mutable**) operácia s jednou z týchto premenných zrejme zmení aj druhú premennú. Pozrime si takéto školské príklady:

- napíšeme funkciu, ktorá ako parameter dostáva zoznam čísel a jej úlohou je z tohto zoznamu vypísať dva zoznamy - zoznam záporných a zoznam nezáporných čísel; elegantným riešením by mohlo byť:

```
def vypis_dva_zoznamy(zoznam):
    zoz1 = zoz2 = []
    for prvok in zoznam:
        if prvok < 0:
            zoz1.append(prvok)
        else:
            zoz2.append(prvok)
    print('zaporne =', zoz1)
    print('nezaporne =', zoz2)
```

  Prekvapením môže byť otestovanie tejto funkcie:

```
>>> vypis_dva_zoznamy([5, 7.3, 0, -3, 0.0, 1, -3.14])
zaporne = [5, 7.3, 0, -3, 0.0, 1, -3.14]
nezaporne = [5, 7.3, 0, -3, 0.0, 1, -3.14]
```

  Všimnite si priradenie `zoz1 = zoz2 = []`. Týmto priradením obe premenné získali referenciu v pamäti na ten istý zoznam. Keďže ďalej s oboma premenným pracujeme len pomocou **mutable** operácií (metóda `append()`), stále sa uchovávajú referencie na ten istý zoznam. Riešením by mohlo byť, buď opraviť úvodnú inicializáciu premenných napr. na `zoz1, zoz2 = [], []`, alebo používanie **immutable** operácií (namiesto `zoz1.append(prvok)` použijeme `zoz1 = zoz1 + [prvok]`).

- ďalšia funkcia z daného reťazca vytvorí kópiu, v ktorej ale vynechá všetky reťazce (ponechá napr. čísla):

```
def zoznam_bez_retazcov(zoznam):
    kopia = zoznam
    for prvok in zoznam:
        if type(prvok) == str:
            kopia.remove(prvok)
    return kopia
```

Riešenie je dostatočne čitateľné. Otestujme:

```
>>> nejaky = [2, '+', 5, 'je', 7]
>>> zoznam_bez_retazcov(nejaky)
[2, 5, 7]
>>> nejaky
[2, 5, 7]
>>> nejaky = [1, 'prvy', 'druhy', 'treti', 'stvrty']
>>> zoznam_bez_retazcov(nejaky)
[1, 'druhy', 'stvrty']
>>> nejaky
[1, 'druhy', 'stvrty']
```

Prvý test dal dobrý výsledok, ale funkcia pokazila aj pôvodný zoznam. V druhom teste dokonca dostávame aj chybný výsledok aj pokazený vstupný zoznam. Aspoň trochu skúsenejší pythonista vidí problém v priradení `kopia = zoznam`. Premenná `kopia` referencuje ten istý zoznam ako je v parametri `zoznam`. Preto vyhadzovanie prvkov z premennej `kopia` ich bude vyhadzovať aj zo `zoznam`. Stačí opraviť úvodné priradenie napr. na `kopia = list(zoznam)` a tým sa naozaj do `kopia` vyrobí nový zoznam. Teraz to už funguje, hoci skúsenejšiemu pythonistovi sa nemusí páčiť konštrukcia `kopia.remove(prvok)`. Táto metóda vyhľadá v zozname `kopia` daný prvok a jeho prvý výskyt vyhodí. Skôr by tu (namiesto vyhadzovania) zapísal skladanie nového reťazca pomocou `append()` ale už bez reťazcov, napr.

```
def zoznam_bez_retazcov(zoznam):
    novy = []
    for prvok in zoznam:
        if type(prvok) != str:
            novy.append(prvok)
    return novy
```

- nasledovná funkcia by mala vyčistiť obsah zadaného zoznamu:

```
def cisti(zoznam):
    zoznam = []
```

Samozrejme, že to nefunguje:

```
>>> ab = [1, 'dva', 3.14]
>>> cisti(ab)
>>> ab
[1, 'dva', 3.14]
```

Volanie funkcie `cisti()` vytvorí lokálnu premennú `zoznam` a ten dostáva hodnotu skutočného parametra, teda referenciu na zoznam `[1, 'dva', 3.14]`. Lenže priradenie `zoznam = []` je **immutable** operácia, teda do lokálnej premennej `zoznam` priradíme novú referenciu na prázdny zoznam. Tým sa ale nezmení referencia pôvodnej premennej `ab`, ktorá bola skutočným parametrom volania funkcie `cisti()`. Riešením by asi bolo použitie nejakej **mutable** operácie, ktorá vyčistí obsah zoznamu, napr.

```
def cisti(zoznam):
    zoznam.clear()
```

Predpokladáme, že takéto príklady vás presvedčia, že pri práci so zoznamami vo funkciách musíte byť veľmi ostražití a uvedomovať si dôsledky referencií na zoznamy.

S referenciami môžeme mať problémy nielen v situáciách, keď dve premenné odkazujú na ten istý zoznam, ale aj v prípadoch, keď zoznam obsahuje nejaký podzoznam, t.j. referenciu na iný zoznam. Vyskúšajme:

```
>>> x = ['prvy', [2, 3], 'styri']
>>> y = x[1]
>>> y
[2, 3]
>>> y.append('kuk')
>>> y
[2, 3, 'kuk']
```

Zatiaľ to vyzerá dobre: do premennej `y` sme priradili prvok zoznamu `x` s indexom 1. Vidíme, že je to dvojprvkový zoznam `[2, 3]` a preto do neho na koniec pridáme nejaký reťazec pomocou `y.append('kuk')`. Aj toto funguje dobre: v premennej `y` je teraz `[2, 3, 'kuk']`. Lenže teraz sa zmenil aj pôvodný zoznam `x`:

```
>>> x
['prvy', [2, 3, 'kuk'], 'styri']
```

Pekne tento efekt vidieť aj na ďalšom príklade:

```
>>> a = ['a']
>>> zoz = [a, a, a]
>>> zoz
[['a'], ['a'], ['a']]
```

My už vieme, že zoznam `zoz` má tri prvky a všetky tri sú referenciami na ten istý **mutable** objekt. Ak sa tento zmení **mutable** operáciou, zmenia sa obsahy všetkých troch prvkov zoznamu:

```
>>> a.insert(0, 'b')
>>> a
['b', 'a']
>>> zoz
[['b', 'a'], ['b', 'a'], ['b', 'a']]
```

Nielen začiatočníka prekvapí takáto chyba: plánujeme vytvoriť 100-prvkový zoznam, ktorý na začiatok bude obsahovať prázdne zoznamy `[]`. Potom budeme niektoré z týchto zoznamov meniť, napr.

```
>>> pole = [[]] * 100
>>> pole[3].append(1)
>>> pole[7].insert(0, 0)
>>> pole[9] += [2]
>>> pole
[[0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2],
[0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2],
[0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2],
[0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2],
[0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2],
...]
```

Vidíme, že napriek tomu, že sme menili len niektoré tri prvky zoznamu `pole`, malo to vplyv na všetkých 100 prvkov. Zrejme platí:

```
>>> pole[2] == pole[3]
True
>>> pole[3] = [0, 1, 2]
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> pole[2] == pole[3]
True
```

Obsahy 2. a 3. prvkov sú rovnaké aj keď nemajú tú istú referenciu. Môžeme to vidieť, keď vymažeme obsah, napr. 0. prvku:

```
>>> pole[0].clear()
>>> pole
[[], [], [], [0, 1, 2], [], [], [], [], [], [], [], [], [], [], [], [], [], [],
[], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [],
[], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [],
[], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [],
[], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [],
[], []]
```

Všetky prvky s rovnakou referenciou sa vymazali okrem 3., ktorý mal rovnaký obsah, ale inú referenciu. V niektorých situáciách nám pomôže nová operácia `is`, pomocou ktorej vieme zistiť, či majú dva objekty identickú referenciu. Napr.

```
>>> pole[1] = []
>>> pole[0] == pole[1]
True
>>> pole[0] is pole[1]
False
>>> pole[0] is pole[2]
True
```

Zmenili sme 1. prvok na prázdny zoznam, preto platí `pole[0] == pole[1]`, ale sú to rôzne referencie, preto neplatí `pole[0] is pole[1]`.

9.1.1 Zoznamy a reťazce

V Pythone existuje niekoľko metód, v ktorých zoznamy spolupracujú so znakovými reťazcami. Ukážeme dve reťazcové metódy, ktoré budeme odteraz veľmi intenzívne využívať.

metóda split()

metóda `split()`

Keďže je to reťazcová metóda, má tvar:

```
ret'azec.split()
```

Metóda rozbije daný reťazec na samostatné reťazce a uloží ich do zoznamu (teda vracia zoznam reťazcov). Predpokladáme, že tieto podreťazce sú navzájom oddelené „medzerovými“ znakmi (medzera, znak konca riadku, tabulátor). V helpe (napr. `help('').split`) sa môžete dozvedieť ďalšie možnosti tejto funkcie.

Najlepšie to pochopíte na niekoľkých príkladoch. Metóda `split()` sa často využíva pri rozdelení prečítaného reťazca zo vstupu (`input()` alebo `subor.readline()`) na viac častí, napr.

```
>>> ret = input('zadaj 2 čísla: ')
zadaj 2 čísla: 15 999
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> pole = ret.split()
>>> pole
['15', '999']
>>> a, b = pole
>>> ai, bi = int(pole[0]), int(pole[1])
>>> a, b, ai, bi
('15', '999', 15, 999)
```

Niekedy môžeme vidieť aj takýto zápis:

```
>>> meno, priezvisko = input('zadaj meno a priezvisko: ').split()
zadaj meno a priezvisko: Janko Hraško
>>> meno
'Janko'
>>> priezvisko
'Hraško'
```

Metóda `split()` môže dostať ako parameter oddeľovač, napr. ak sme prečítali čísla oddelené čiarkami:

```
sucet = 0
for prvok in input('zadaj čísla: ').split(','):
    sucet += int(prvok)
print('ich súčet je', sucet)
```

```
zadaj čísla: 10,20,30,40
ich súčet je 100
```

metóda join()

metóda `join()`

Opäť je to reťazcová metóda. Má tvar:

```
oddeľovač.join(zoznam)
```

Metóda zlepí všetky reťazce z daného zoznamu reťazcov do jedného, pričom ich navzájom oddelí uvedeným `oddeľovačom`, t. j. nejakým zadaným reťazcom. Ako zoznam môžeme uviesť ľubovoľnú postupnosť (iterovateľný objekt) reťazcov.

Ukážme to na príklade:

```
>>> pole = ['prvý', 'druhý', 'tretí']
>>> pole
['prvý', 'druhý', 'tretí']
>>> ''.join(pole)
'prvýdruhýtretí'
>>> '...'.join(pole)
'prvý...druhý...tretí'
>>> list(str(2013))
['2', '0', '1', '3']
>>> '.'.join(list(str(2013)))
'2.0.1.3'
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> '.'.join('Python')
'P.y.t.h.o.n'
```

Preštudujte:

```
>>> veta = 'kto druhemu jamu kope'
>>> ' '.join(veta[::-1].split()[::-1])
'otk umehurd umaj epok'

>>> ' '.join(input('? ').split()[::-1])
? anicka dusicka kde si bola ked si si cizmicky zarosila
'zarosila cizmicky si si ked bola si kde dusicka anicka'
```

9.1.2 Funkcia enumerate()

Najčastejšie sa využíva v takýchto for-cykloch:

```
i = 0
for prvok in postupnost:
    print(i, prvok)
    i += 1
```

alebo (toto nefunguje napr. pre otvorený súbor)

```
for i in range(len(postupnost)):
    prvok = postupnost[i]
    print(i, prvok)
```

Teda v týchto príkladoch by sa nám zišiel taký for-cyklus, v ktorom máme pri každom prechode k dispozícii prvok postupnosti a aj index (poradové číslo prechodu cyklu). Pomocou štandardnej funkcie `enumerate()` môžeme takýto cyklus prepísať:

```
for i, prvok in enumerate(postupnost):
    print(i, prvok)
```

Toto znamená, že for-cyklus má dve premenné cyklu: prvá premenná `i` je počítadlo prechodov (číslo od 0) a druhá premenná `prvok` bude postupne nadobúdať hodnoty všetkých prvkov postupnosti.

9.2 n-tice (tuple)

Sú to vlastne len nemeniteľné (**immutable**) zoznamy. Pythonovský typ `tuple` dokáže robiť skoro všetko to isté ako `list` okrem **mutable** operácií. Takže to najprv zhrňme a potom si ukážeme, ako to pracuje:

- operácia + na zreťazovanie (spájanie dvoch n-tíc)
- operácia * na viacnásobné zreťazovanie (viacnásobné spájanie jednej n-tice)
- operácia `in` na zisťovanie, či sa nejaká hodnota nachádza v n-tici
- operácia indexovania `[i]` na zistenie hodnoty prvku na zadanom indexe
- operácia rezu `[i:j:k]` na zistenie hodnoty nejakej podčasti n-tice
- relačné operácie ==, !=, <, <=, >, >= na porovnávanie obsahu dvoch n-tíc

- metódy `count()` a `index()` na zisťovanie počtu výskytov, resp. indexu prvého výskytu
- prechádzanie n-tice pomocou for-cyklu (iterovanie)
- štandardné funkcie `len()`, `sum()`, `min()`, `max()` ktoré zisťujú niečo o prvkoch n-tice

Takže **n-tice**, tak ako aj zoznamy sú **štruktúrované typy**, t.j. sú to typy, ktoré obsahujú hodnoty nejakých (možno rôznych) typov (sú to tzv. **kolekcie**):

- konštanty typu n-tica uzatvárame do okrúhlych zátvoriek a navzájom oddeľujeme čiarkami
- funkcia `len()` vráti počet prvkov n-tice, napr.

```
>>> stred = (150, 100)
>>> zviera = ('slon', 2013, 'gray')
>>> print(stred)
(150, 100)
>>> print(zviera)
('slon', 2013, 'gray')
>>> nic = ()
>>> print(nic)
()
>>> len(stred)
2
>>> len(zviera)
3
>>> len(nic)
0
>>> type(stred)
<class 'tuple'>
```

Vidíte, že n-tica môže byť aj prázdna, označujeme ju `()` a vtedy má počet prvkov 0 (teda `len(())` je 0). Ak n-tica nie je prázdna, hodnoty sú navzájom oddelené čiarkami.

9.2.1 n-tica s jednou hodnotou

n-ticu s jednou hodnotou **nemôžeme** zapísať takto:

```
>>> p = (12)
>>> print(p)
12
>>> type(p)
<class 'int'>
>>>
```

Ak zapíšeme ľubovoľnú hodnotu do zátvoriek, nie je to n-tica (v našom prípade je to len jedno celé číslo). Pre jednoprvkovú n-ticu musíme do zátvoriek zapísať aj čiarku:

```
>>> p = (12,)
>>> print(p)
(12,)
>>> len(p)
1
>>> type(p)
<class 'tuple'>
>>>
```

Pre Python sú dôležitejšie čiarky ako zátvorky. V mnohých prípadoch si Python zátvorky domyslí (čiarky si nedomyslí nikdy):

```
>>> stred = 150, 100
>>> zviera = 'slon', 2013, 'gray'
>>> p = 12,
>>> print(stred)
(150, 100)
>>> print(zviera)
('slon', 2013, 'gray')
>>> print(p)
(12,)
```

Uvedomte si rozdiel medzi týmito dvoma priradeniami:

```
>>> a = 3.14
>>> b = 3,14
>>> print(a, type(a))
3.14 <class 'float'>
>>> print(b, type(b))
(3, 14) <class 'tuple'>
```

9.2.2 Operácie s n-ticami

Operácie fungujú presne rovnako ako fungovali s reťazcami a zoznamami:

- operácia + **zreťazí** dve n-tice, to znamená, že **vyrobí novú n-ticu**, ktorá obsahuje najprv všetky prvky prvej n-tice a za tým všetky prvky druhej n-tice; zrejme oba operandy musia byť n-tice: nemôžeme zreťazovať n-ticu s hodnotou iného typu
- operácia * zadané číslo-krát zreťazí jednu n-ticu, to znamená, že **vyrobí novú n-ticu**, ktorá požadovaný-krát obsahuje všetky prvky zadanej n-tice; operácia viacnásobného zreťazovania má jeden operand typu n-tica a druhý musí byť celé číslo
- operácia `in` vie zistiť, či sa nejaký prvok nachádza v n-tici

Napríklad:

```
>>> stred = (150, 100)
>>> zviera = ('slon', 2013, 'gray')
>>> stred + zviera
(150, 100, 'slon', 2013, 'gray')
>>> zviera + stred
('slon', 2013, 'gray', 150, 100)
>>> stred * 5
(150, 100, 150, 100, 150, 100, 150, 100, 150, 100)
>>> 50 * (1)
50
>>> 50 * (1,)
(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
>>> 50 * (1, 2)
(1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1,
2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2,
1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1,
2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
```

n-tica môže obsahovať ako svoje prvky aj iné n-tice:

```
>>> stred = (150, 100)
>>> p = ('stred', stred)
>>> p
('stred', (150, 100))
>>> len(p)
2
>>> p = (stred, stred, stred, stred)
>>> p
((150, 100), (150, 100), (150, 100), (150, 100))
>>> (stred,) * 4
((150, 100), (150, 100), (150, 100), (150, 100))
>>> (stred) * 4                                    # čo je to isté ako stred * 4
(150, 100, 150, 100, 150, 100, 150, 100)
```

Operátorom `in` vieme zistiť, či sa nejaká hodnota nachádza v n-tici ako jeden jeho prvok. Takže

```
>>> p = (stred, stred, stred, stred)
>>> p
((150, 100), (150, 100), (150, 100), (150, 100))
>>> stred in p
True
>>> 150 in p
False
>>> 150 in stred
True
>>> zviera = ('slon', 2013, 'gray')
>>> 2013 in zviera
True
>>> (2013, 'gray') in zviera
False
```

9.2.3 Funkcia tuple()

Vyrobí n-ticu z ľubovoľnej postupnosti (z iterovateľného objektu, t.j. takého objektu, ktorý sa dá prechádzať for-cyklom), napr. zo znakového reťazca, zo zoznamu, vygenerovanej postupnosti celých čísel pomocou `range()`, ale iterovateľný je aj otvorený textový súbor. Znakový reťazec funkcia `tuple()` rozoberie na znaky:

```
>>> tuple('Python')
('P', 'y', 't', 'h', 'o', 'n')
```

Vytvorenie n-tice pomocou `range()`:

```
>>> tuple(range(1, 16))
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
>>> a = tuple(range(1000000))
>>> len(a)
1000000
```

Podobne môžeme skonštruovať n-ticu z textového súboru. Predpokladajme, že súbor obsahuje tieto 4 riadky:

```
prvy
druhy
treti
stvrty
```

potom

```
>>> with open('abc.txt') as t:
        obsah = tuple(t)

>>> obsah
('prvy\n', 'druhy\n', 'treti\n', 'stvrty\n')
```

Riadky súboru sa postupne stanú prvkami n-tice.

9.2.4 for-cyklus s n-ticami

for-cyklus je programová konštrukcia, ktorá postupne prechádza všetky prvky nejakého „iterovateľného“ objektu. Doteraz sme sa stretli s iterovaním pomocou funkcie `range()`, prechádzaním prvkov reťazca `str` a zoznamu `list`, aj celých riadkov textového súboru. Ale už od 2. prednášky sme používali aj takýto zápis:

```
for i in 2, 3, 5, 7, 11, 13:
    print('prvocislo', i)
```

V tomto zápise vidíme n-ticu (tuple) `2, 3, 5, 7, 11, 13`. Len sme tomu nemuseli dávať zátvorky. Keďže aj n-tica je iterovateľný objekt, Môžeme ju používať vo for-cykle rovnako ako iné iterovateľné typy. To isté, ako predchádzajúci príklad, by sme zapísali napr. aj takto:

```
cisla = (2, 3, 5, 7, 11, 13)
for i in cisla:
    print('prvocislo', i)
```

Keďže teraz už vieme manipulovať s n-ticami, môžeme zapísať napr.

```
>>> rozne = ('retazec', (100, 200), 3.14, len)
>>> for prvok in rozne:
        print(prvok, type(prvok))

retazec <class 'str'>
(100, 200) <class 'tuple'>
3.14 <class 'float'>
<built-in function len> <class 'builtin_function_or_method'>
```

Tu vidíme, že prvkami n-tice môžu byť najrôznejšie objekty, hoci aj funkcie (tu je to štandardná funkcia `len`).

Pomocou operácií s n-ticami vieme zapísať aj zaujímavejšie postupnosti čísel, napr.

```
>>> for i in 10 * (1,):
        print(i, end=' ')

1 1 1 1 1 1 1 1 1 1
>>> for i in 10 * (1, 2):
         print(i, end=' ')

1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
>>> for i in 10 * tuple(range(10)):
         print(i, end=' ')

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
3 4 5 6 7 8 9
```

Ďalej pomocou for-cyklu vieme n-tice skladať podobne, ako sme to robili so zoznamami. V nasledovnom príklade vytvoríme n-ticu zo všetkých deliteľov nejakého čísla:

```
cislo = int(input('zadaj cislo: '))
delitele = ()
for i in range(1, cislo+1):
    if cislo % i == 0:
        delitele = delitele + (i,)
print('delitele', cislo, 'su', delitele)
```

po spustení:

```
zadaj cislo: 124
delitele 124 su (1, 2, 4, 31, 62, 124)
```

Všimnite si, ako sme pridali jeden prvok na koniec n-tice: `delitele = delitele + (i,)`. Museli, sme vytvoriť jednoprvkovú n-ticu `(i,)` a tú sme zreťazili s pôvodnou n-ticou `delitele`. Mohli sme to zapísať aj takto: `delitele += (i,)`. Zrejme, keby sme toto riešili pomocou zoznamov, použili by sme metódu `append()`.

9.2.5 Funkcia enumerate()

Pred chvíľou sme videli použitie štandardnej funkcie `enumerate()`. Napr.

```
pole = (2, 3, 5, 7, 9, 11, 13, 17, 19)
for ix, pr in enumerate(pole):
    print(f'{ix}. prvocislo je {pr}')
```

```
0. prvocislo je 2
1. prvocislo je 3
2. prvocislo je 5
3. prvocislo je 7
4. prvocislo je 9
5. prvocislo je 11
6. prvocislo je 13
7. prvocislo je 17
8. prvocislo je 19
```

Funkcia `enumerate()` v skutočnosti z jednej ľubovoľnej postupnosti vygeneruje postupnosť dvojíc (ktoré sú už typu `tuple`). Túto postupnosť ďalej preliezame for-cyklom. Mohli by sme to zapísať aj takto:

```
pole = (2, 3, 5, 7, 9, 11, 13, 17, 19)
for dvojica in enumerate(pole):
    ix, pr = dvojica
    print(f'{ix}. prvocislo je {pr} ... dvojica = {dvojica}')
```

po spustení:

```
0. prvocislo je 2 ... dvojica = (0, 2)
1. prvocislo je 3 ... dvojica = (1, 3)
2. prvocislo je 5 ... dvojica = (2, 5)
3. prvocislo je 7 ... dvojica = (3, 7)
4. prvocislo je 9 ... dvojica = (4, 9)
5. prvocislo je 11 ... dvojica = (5, 11)
6. prvocislo je 13 ... dvojica = (6, 13)
7. prvocislo je 17 ... dvojica = (7, 17)
8. prvocislo je 19 ... dvojica = (8, 19)
```

Ak by sme skúmali výsledok z tejto funkcie, dozvedeli by sme sa:

```
>>> enumerate(pole)
<enumerate object at 0x02F59B48>
>>> list(enumerate(pole))
[(0, 2), (1, 3), (2, 5), (3, 7), (4, 9), (5, 11), (6, 13), (7, 17), (8, 19)]
```

čo je naozaj postupnosť skutočných dvojíc.

9.2.6 Indexovanie

n-tice indexujeme rovnako ako sme indexovali zoznamy:

- prvky postupnosti môžeme indexovať v `[]` zátvorkách, pričom index musí byť od 0 až po počet prvkov-1
- pomocou rezu (slice) vieme indexovať časť n-tice (niečo ako podreťazec) tak, že `[]` zátvoriek zapíšeme aj dvojbodku:
  - `ntica[od:do]` n-tica z prvkov s indexmi `od` až po `do-1`
  - `ntica[:do]` n-tica z prvkov od začiatku až po prvok s indexom `do-1`
  - `ntica[od:]` n-tica z prvkov s indexmi `od` až po koniec n-tice
  - `ntica[od:do:krok]` n-tica z prvkov s indexmi `od` až po `do-1`, pričom berieme každý `krok` prvok

Niekoľko príkladov

```
>>> prvo = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
>>> prvo[2]
5
>>> prvo[2:5]
(5, 7, 11)
>>> prvo[4:5]
(11,)
>>> prvo[:5]
(2, 3, 5, 7, 11)
>>> prvo[5:]
(13, 17, 19, 23, 29)
>>> prvo[::2]
(2, 5, 11, 17, 23)
>>> prvo[::-1]
(29, 23, 19, 17, 13, 11, 7, 5, 3, 2)
```

Vidíme, že s n-ticami pracujeme veľmi podobne ako sme pracovali so znakovými reťazcami a so zoznamami. Keď sme ale do znakového reťazca chceli pridať jeden znak (alebo aj viac), museli sme to robiť rozoberaním a potom skladaním:

```
>>> prvo = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
>>> prvo = prvo[:5] + ('fuj',) + prvo[5:]
>>> prvo
(2, 3, 5, 7, 11, 'fuj', 13, 17, 19, 23, 29)
```

Pred 5-ty prvok vloží nejaký znakový reťazec.

Rovnako ako nemôžeme zmeniť hodnotu nejakého znaku reťazca obyčajným priradením:

```
>>> ret = 'Python'
>>> ret[2] = 'X'
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
...
TypeError: 'str' object does not support item assignment
>>> ret = ret[:2] + 'X' + ret[3:]
>>> ret
'PyXhon'
>>> ntica = (2, 3, 5, 7, 11, 13)
>>> ntica[2] = 'haha'
...
TypeError: 'tuple' object does not support item assignment
>>> ntica = ntica[:2] + ('haha',) + ntica[3:]
>>> ntica
(2, 3, 'haha', 7, 11, 13)
```

Všimnite si, že Python vyhlásil rovnakú chybu pre `tuple` ako pre `str`. Hovoríme, že ani reťazce ani n-tice nie sú meniteľné (teda sú **immutable**).

9.2.7 Porovnávanie n-tíc

Porovnávanie n-tíc je veľmi podobné ako porovnávanie reťazcov a zoznamov. Pripomeňme si, ako je to pri zoznamoch:

- postupne porovnáva i-te prvky oboch zoznamov, kým sú rovnaké; pri prvej nerovnosti je výsledkom porovnanie týchto dvoch hodnôt
- ak je pri prvej nezhode v prvom zozname menšia hodnota ako v druhom, tak prvý zoznam je menší ako druhý
- ak je prvý zoznam kratší ako druhý a zodpovedajúce prvky sa zhodujú, tak prvý zoznam je menší ako druhý

Hovoríme tomu **lexikografické** porovnávanie.

Teda aj pri porovnávaní n-tíc sa budú postupne porovnávať zodpovedajúce si prvky a pri prvej nerovnosti sa skontroluje, ktorý z týchto prvkov je menší. Treba tu ale dodržiavať jedno veľmi dôležité pravidlo: porovnávať hodnoty napr. na menší môžeme len vtedy, keď sú zhodných typu:

```
>>> 5 < 'a'
...
TypeError: unorderable types: int() < str()
>>> (1, 5, 10) < (1, 'a', 10)
...
TypeError: unorderable types: int() < str()
>>> (1, 5, 10) != (1, 'a', 10)
True
```

Najlepšie je porovnávať také n-tice, ktoré majú prvky rovnakého typu. Pri n-ticiach, ktoré majú zmiešané typy si musíme dávať väčší pozor

```
>>> ('Janko', 'Hrasko', 'Zilina') < ('Janko', 'Jesensky', 'Martin')
True
>>> (1, 2, 3, 4, 5, 5, 6, 7, 8) < tuple(range(1, 9))
True
>>> ('Janko', 'Hrasko', 2008) < ('Janko', 'Hrasko', 2007)
False
```

9.2.8 Viacnásobné priradenie

Tu len pripomenieme, ako funguje viacnásobné priradenie: ak je pred znakom priradenia = viac premenných, ktoré sú oddelené čiarkami, tak za znakom priradenia musí byť iterovateľný objekt, ktorý má presne toľko hodnôt, ako počet premenných. Iterovateľným objektom môže byť zoznam (`list`), n-tica (`tuple`), znakový reťazec (`str`), generovaná postupnosť čísel (`range()`) ale aj otvorený textový súbor (`open()`), ktorý má presne toľko riadkov, koľko je premenných v priradení.

Ak do jednej premennej priraďujeme viac hodnôt oddelených čiarkou, Python to chápe ako priradenie n-tice. Pozrite nasledovné priradenia.

Priradíme n-ticu:

```
>>> a1, a2, a3, a4 = 3.14, 'joj', len, (1, 3, 5)
>>> print(a1, a2, a3, a4)
3.14 joj <built-in function len> (1, 3, 5)
>>> a, b, c, d, e, f = 3 * (5, 7)
>>> print(a, b, c, d, e, f)
5 7 5 7 5 7
```

Priradíme vygenerovanú postupnosť 4 čísel:

```
>>> a, b, c, d = range(2, 6)
>>> print(a, b, c, d)
2 3 4 5
```

Priradíme znakový reťazec:

```
>>> d, e, f, g, h, i = 'Python'
>>> print(d, e, f, g, h, i)
P y t h o n
```

Priradíme riadky textového súboru:

```
>>> with open('dva.txt', 'w') as f:
        f.write('first\nsecond\n')

>>> with open('dva.txt') as subor:
        prvy, druhy = subor

>>> prvy
'first\n'
>>> druhy
'second\n'
```

Vieme zapísať aj takto:

```
>>> prvy, druhy = open('dva.txt')
```

Tento posledný príklad je veľmi umelý a v praxi sa asi priamo do premenných takto čítať nebude.

Viacnásobné priradenie používame napr. aj na výmenu obsahu dvoch (aj viac) premenných:

```
>>> x, y, z = y, z, x
```

Aj v tomto príklade je na pravej strane priradenia (za =) n-tica: `(y, z, x)`.

9.2.9 n-tica ako návratová hodnota funkcie

V Pythone sa dosť využíva to, že návratovou hodnotou funkcie môže byť n-tica, t.j. výsledkom funkcie je naraz niekoľko návratových hodnôt. Napr. nasledovný príklad počíta celočíselné delenie a súčasne zvyšok po delení:

```
def zisti(a, b):
    return a // b, a % b
```

Funkciu môžeme použiť napríklad takto:

```
>>> podiel, zvysok = zisti(153, 33)
>>> print('podiel =', podiel, 'zvysok =', zvysok)
podiel = 4 zvysok = 21
```

Ak z výsledku takejto funkcie potrebujeme použiť len jednu z hodnôt, môžeme zapísať:

```
>>> print('zvysok =', zisti(153, 33)[1])
```

Ďalšia funkcia vráti postupnosť všetkých deliteľov nejakého čísla:

```
def delitele(cislo):
    vysl = ()
    for i in range(1, cislo+1):
        if cislo % i == 0:
            vysl = vysl + (i,)
    return vysl
```

Otestujeme:

```
>>> deli = delitele(24)
>>> print(deli)
(1, 2, 3, 4, 6, 8, 12, 24)
>>> if 2 in deli:
        print('parne')

parne
>>> print('sucet delitelov =', sum(deli))
sucet delitelov = 60
>>> print('je prvocislo =', len(delitele(int(input('zadaj cislo: '))))==2)
zadaj cislo: 11213
je prvocislo = True
>>> print('je prvocislo =', len(delitele(int(input('zadaj cislo: '))))==2)
zadaj cislo: 1001
je prvocislo = False
```

Príklad ukazuje, že keď je výsledkom n-tica, môžeme ju ďalej rôzne spracovať alebo testovať.

9.2.10 Ďalšie funkcie a metódy

S n-ticami vedia pracovať nasledovné štandardné funkcie:

- `len(ntica)` - vráti počet prvkov n-tice
- `sum(ntica)` - vypočíta súčet prvkov n-tice (všetky musia byť čísla)
- `min(ntica)` - zistí najmenší prvok n-tice (prvky sa musia dať navzájom porovnať, nemôžeme tu miešať rôzne typy)

- `max(ntica)` - zistí najväčší prvok n-tice (ako pri `min()` ani tu sa nemôžu typy prvkov miešať)

Na rozdiel od zoznamov a znakových reťazcov, ktoré majú veľké množstvo metód, n-tice majú len dve:

- `ntica.count(hodnota)` - zistí počet výskytov nejakej hodnoty v n-tici
- `ntica.index(hodnota)` - vráti index (poradie) v n-tici prvého (najľavejšieho) výskytu danej hodnoty, ak sa hodnota v n-tici nenachádza, metóda spôsobí spadnutie na chybe (`ValueError: tuple.index(x): x not in tuple`)

Ukážme tieto funkcie na malom príklade. V n-tici uložíme niekoľko nameraných teplôt a potom vypíšeme priemernú, minimálnu aj maximálnu teplotu:

```
>>> teploty = (14, 22, 19.5, 17.1, 20, 20.4, 18)
>>> print('počet nameraných teplôt: ', len(teploty))
počet nameraných teplôt:  7
>>> print('minimálna teplota: ', min(teploty))
minimálna teplota:  14
>>> print('maximálna teplota: ', max(teploty))
maximálna teplota:  22
>>> print('priemerná teplota: ', round(sum(teploty) / len(teploty), 2))
priemerná teplota:  18.71
```

Ďalej môžeme zistiť, kedy bola nameraná konkrétna hodnota:

```
>>> teploty.index(20)
4
>>> teploty.index(20.1)
...
ValueError: tuple.index(x): x not in tuple
```

Môžeme zistiť, koľko-krát sa nejaká konkrétna teplota vyskytla v našich meraniach:

```
>>> teploty.count(20)
1
>>> teploty.count(20.1)
0
```

9.2.11 n-tice a grafika

Nasledovný príklad predvedie použitie n-tíc v grafickom režime. Zadefinujeme niekoľko bodov v rovine a potom pomocou nich kreslíme nejaké farebné polygóny. Začnime takto:

```
a = (70, 150)
b = (200, 200)
c = (150, 250)
d = (120, 70)
e = (50, 220)

canvas = tkinter.Canvas()
canvas.pack()
canvas.create_polygon(a, b, c, d, fill='red')
```

Ak by sme chceli jedným priradením priradiť dva body do premenných `a` aj `b`, zapíšeme:

```
a, b = (100, 150), (180, 200)
```

čo je vlastne:

```
a, b = ((100, 150), (180, 200))
```

Polygónov môžeme nakresliť aj viac (zrejme väčšinou záleží na poradí ich kreslenia):

```
canvas.create_polygon(e, a, c, fill='green')
canvas.create_polygon(e, d, b, fill='yellow')
canvas.create_polygon(a, b, c, d, fill='red')
canvas.create_polygon(a, c, d, b, fill='blue')
```

Vidíme, že niektoré postupnosti bodov tvoria jednotlivé útvary, preto zapíšme:

```
utvar1 = e, a, c
utvar2 = e, d, b
utvar3 = a, b, c, d
utvar4 = a, c, d, b

canvas.create_polygon(utvar1, fill='green')
canvas.create_polygon(utvar2, fill='yellow')
canvas.create_polygon(utvar3, fill='red')
canvas.create_polygon(utvar4, fill='blue')
```

Volanie funkcie `canvas.create_polygon()` sa tu vyskytuje 4-krát, ale s rôznymi parametrami. Prepíšme to do for-cyklu:

```
utvar1 = e, a, c
utvar2 = e, d, b
utvar3 = a, b, c, d
utvar4 = a, c, d, b

for param in (utvar1, 'green'), (utvar2, 'yellow'), (utvar3, 'red'), (utvar4, 'blue'):
    utvar, farba = param
    canvas.create_polygon(utvar, fill=farba)
```

Dostávame to isté. Vo for-cykle sa najprv do premennej `param` priradí dvojica s dvoma prvkami: útvar a farba, a v tele cyklu sa táto premenná s dvojicou priradí do dvoch premenných `utvar` a `farba`. Potom sa zavolá funkcia `canvas.create_polygon()` s týmito parametrami.

Už sme videli aj predtým, že pre for-cyklus existuje vylepšenie: ak sa do premennej cyklu postupne priraďujú nejaké dvojice hodnôt a tieto by sa na začiatku tela rozdelili do dvoch premenných, môžeme priamo tieto dve premenné použiť ako premenné cyklu (ako keby viacnásobné priradenie). Podobnú ideu sme mohli vidieť pri použití `enumerate()`. Predchádzajúci príklad prepíšeme:

```
utvar1 = e, a, c
utvar2 = e, d, b
utvar3 = a, b, c, d
utvar4 = a, c, d, b

for utvar, farba in (utvar1, 'green'), (utvar2, 'yellow'), (utvar3, 'red'),(utvar4,
↪'blue'):
    canvas.create_polygon(utvar, fill=farba)
```

Pozrime sa na n-ticu, ktorá sa prechádza týmto for-cyklom:

```
>>> cyklus = (utvar1, 'green'), (utvar2, 'yellow'), (utvar3, 'red'), (utvar4, 'blue')
>>> cyklus
((((50, 220), (70, 150), (150, 250)), 'green') (((50, 220), (120, 70),
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
(200, 200)), 'yellow') (((70, 150), (200, 200), (150, 250), (120, 70)),
'red') (((70, 150), (150, 250), (120, 70), (200, 200)), 'blue'))
```

Vidíme, že n-tica v takomto tvare je dosť ťažko čitateľná, ale for-cyklus jej normálne rozumie. Zrejme by sme teraz mohli zapísať:

```
for utvar, farba in cyklus:
   canvas.create_polygon(utvar, fill=farba)
```

9.2.12 Zoznamy a grafika

Už vieme, že väčšina grafických príkazov, napr. `create_line()`, `create_polygon()`, ... akceptujú ako parametre nielen čísla, ale aj n-tice alebo aj zoznamy čísel, resp. zoznamy/dvojice čísel, napr.

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()
utvar = ((100, 50), (200, 120))
canvas.create_rectangle(utvar, fill='blue')
canvas.create_oval(utvar, fill='yellow')
utvar2 = list(utvar)                # z n-tice sa vyrobí pole
utvar2.append((170, 20))
canvas.create_polygon(utvar2, fill='red')
```

alebo môžeme generovať náhodnú krivku:

```
import tkinter, random

canvas = tkinter.Canvas(bg='white')
canvas.pack()
krivka = []
for i in range(30):
    krivka.append()
canvas.create_line(krivka)
```

Ak by sme chceli využiť grafickú funkciu `coords()`, ktorá modifikuje súradnice nakreslenej krivky, nemôžeme jej poslať pole súradníc (dvojíc čísel x a y), ale táto vyžaduje plochý zoznam (alebo n-ticu) čísel. Predchádzajúci príklad mierne zmeníme:

```
import tkinter, random

canvas = tkinter.Canvas(bg='white')
canvas.pack()
poly = canvas.create_polygon(0, 0, 0, 0, fill='yellow', outline='blue')
krivka = []
for i in range(100):
    bod = [random.randrange(350), random.randrange(250)]
    krivka.extend(bod)      # to isté ako    krivka += bod
    canvas.coords(poly, krivka)
    canvas.update()
    canvas.after(300)
```

Použili sme tu dvojicu nových príkazov pre grafickú plochu `canvas`:

```
canvas.update()
canvas.after(300)
```

Tieto dva príkazy označujú, že sa grafická plocha prekreslí (aktualizuje) pomocou metódy `update()` a potom sa na **300** milisekúnd pozdrží výpočet pomocou metódy `after()`. Vďaka tejto dvojici príkazov budeme vidieť postupné pridávanie náhodných bodov vytváranej krivky.

Okrem toho sme tu využili novú metódu `extend()`, ktorá k existujúcemu zoznamu prilepí na koniec nový zoznam. Napr.

```
>>> a = [1, 2, 3, 4]
>>> b = ['xx', 'yy']
>>> a.extend(b)
>>> a
[1, 2, 3, 4, 'xx', 'yy']
```

Zrejme to isté by sme dosiahli aj takto:

```
>>> a = [1, 2, 3, 4]
>>> b = ['xx', 'yy']
>>> a += b
>>> a
[1, 2, 3, 4, 'xx', 'yy']
```

9.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/
- v riešeniach úloh používajte len konštrukcie a funkcie, ktoré sme sa učili na doterajších prednáškach
- pozrite si *Riešenie 9. cvičenia*

1. Napíšte funkciu `zisti_sucet(zoznam, sucet)`, ktorá nájde taký maximálny index do daného zoznamu, pre ktorý súčet všetkých prvkov pred daným indexom nebol väčší ako zadaná hodnota `sucet`. Riešte pomocou while-cyklus, ale nepoužite štandardnú funkciu `sum()`.
   - napr.

   ```
   >>> zoz = [5, 4, 3, 2, 1, 2, 1, 3, 4, 2]
   >>> zisti_sucet(zoz, 10)
   2
   >>> zisti_sucet(zoz, 19)
   7
   >>> sum(zoz[:7])
   18
   ```

2. Napíšte funkciu `zluc(zoz1, zoz2)`, ktorá dostáva dva utriedené zoznamy a vytvorí z nich nový, ktorý obsahuje tie isté prvky ako boli v oboch zoznamoch a tiež je utriedený. Nepoužite `sort()` ani `sorted()`.
   - napr.

```
>>> a = zluc([5, 6, 10], [1, 6, 7, 12])
>>> a
[1, 5, 6, 6, 7, 10, 12]
```

3. Napíšte funkciu `vnoreny_sucet(zoznam)`, pre ktorú vstupný zoznam (alebo n-tica) obsahuje len vnorené zoznamy alebo n-tice celých čísel a funkcia vráti súčet všetkých týchto čísel.
   - napr.

```
>>> pole = [(1, 2), [3], (), [4, 5, 6]]
>>> suc = vnoreny_sucet(pole)
>>> pole
21
>>> vnoreny_sucet(((), ()))
0
```

4. Napíšte funkciu `daj_max(zoznam)`, ktorá vyhodí z daného zoznamu maximálny prvok (jeho prvý výskyt) a túto hodnotu vráti ako výsledok funkcie (`return`). Nepoužite metódu `remove()` ani štandardnú funkciu `max()`.
   - napr.

```
>>> tab = [5, 2, 6, 3, 6, 5]
>>> m = daj_max(tab)
>>> m
6
>>> tab
[5, 2, 3, 6, 5]
```

5. Napíšte funkciu `zisti(meno_suboru)`, ktorá prečíta daný súbor a vypíše počet slov, najdlhšie slovo a slovo, ktoré je posledné v abecede (najväčšie zo všetkých). Nepoužívajte štandardnú funkciu `max()`.
   - napr.

```
>>> zisti('dobs.txt')
pocet slov: 21827
najdlhsie: 'zlostnikomkocisom'
posledne: 'zvyky'
```

   - otestujte na súboroch dobs.txt a twain.txt

6. Napíšte funkciu `vyrovnaj(ntica)`, ktorá dostáva ako parameter n-ticu len s prvkami n-tíc. Funkcia vytvorí n-ticu, ktorá obsahuje všetky prvky vnorených n-tíc.
   - napr.

```
>>> vysl = vyrovnaj(((1, 3), ('a',), (7, 0)))
>>> vysl
(1, 3, 'a', 7, 0)
```

7. Napíšte funkciu `histogram(pole)`, ktorá vypíše stĺpcový histogram z daného zoznamu, resp. n-tice kladných čísel.
   - napr. (každý stĺpec hviezdičiek zodpovedá jednému číslu v zozname)

```
>>> histogram([2, 5, 3, 4])
  *
  *   *
  * * *
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
* * * *
* * * *
>>> histogram([2, 1] * 6)
*   *   *   *   *   *
* * * * * * * * * * * *
```

8. Hovoríme, že dve slová tvoria **anagram**, ak jedno slovo vznikne zmenou poradia písmen v druhom slove. Napr. tieto tri slová sú navzájom anagramami: ‚reklama', ‚makrela', ‚karamel'. Napíšte funkciu `anagram(slovo1, slovo2)`, ktorá pre dve slová zistí, či sú anagramom. Funkcia vráti `True` alebo `False`.

   - napr.

```
>>> anagram('lampa', 'palma')
True
>>> anagram('python', 'thynon')
False
```

   - môžete využiť takú vlastnosť anagramových slov, že majú rovnaký utriedený zoznam svojich písmen

9. Napíšte funkciu `ma_duplikaty(zoznam)`, ktorá pre daný zoznam, resp. n-ticu zistí, či sa tam nejaký prvok nachádza viackrát. Funkcia nemodifikuje vstupný zoznam a vráti `True` alebo `False`.

   - napr.

```
>>> abc = [6, 7, 13, 8, 13, 5]
>>> ma_duplikaty(abc)
True
>>> abc
[6, 7, 13, 8, 13, 5]
>>> ma_duplikaty(tuple(range(1000)))
False
```

10. Napíšte funkciu `pocet_dni_v_mesiaci(mesiac)`, ktorá vráti číslo od 28 do 31 pre mesiac od 1 do 12. Nepoužite príkaz `if`.

    - napr.

```
>>> pocet_dni_v_mesiaci(2)
28
>>> pocet_dni_v_mesiaci(4)
30
```

11. Napíšte funkciu `nahodny_datum()`, ktorá vygeneruje dvojicu (typu `tuple`), ktorá reprezentuje jeden deň v roku, t.j. `(den, mesiac)`. Počítajte len s nepriestupným rokom, teda každý dátum má pravdepodobnosť 1/365. Funkcia nič nevypisuje, ale vráti dvojicu.

    - napr.

```
>>> nahodny_datum()
(15, 5)
>>> nahodny_datum()
(26, 10)
```

12. Napíšte funkciu `narodeninovy_problem(pocet)`, ktorá najprv vygeneruje príslušný `pocet` náhodných dátumov (využije funkciu z predchádzajúcej úlohy) a vráti `True`, ak sa medzi týmito dátumami nejaký opakuje. Dá sa matematicky ukázať, že už pre počet **23** je takáto pravdepodobnosť väčšia ako 50% (pozri narodeninový problém na wikipedii).

    - napr.

```
>>> narodeninovy_problem(10)
False
>>> narodeninovy_problem(30)
True
```

13. Napíšte funkciu `rozdel(retazec)`, ktorá vráti zoznam podreťazcov daného reťazca, ktorý sa rozdelil (pomocou `split()`) podľa riadkov a nie slov. Prázdne riadky na konci súboru odfiltrujte.

    - napr. ak súbor obsahuje

    ```
    prvy riadok
    druhy je posledny
    ```

    dostaneme:

    ```
    >>> rozdel(open('subor.txt'))
    ['prvy riadok', 'druhy je posledny']
    ```

14. Napíšte funkciu `na_cisla(pole)`, ktorá zo zadanej postupnosti (zoznam alebo n-tica) reťazcov vráti (`return`) zoznam celých čísel. Predpokladáme, že reťazce v postupnosti obsahujú len celé čísla.

    - napr.

    ```
    >>> na_cisla('12 422 700'.split())
    [12, 422, 700]
    >>> p = na_cisla(tuple(str(20317)))
    >>> p
    [2, 0, 3, 1, 7]
    ```

15. Napíšte funkciu `ciferny_sucet(cislo)`, ktorá pomocou funkcie `na_cisl()` z predchádzajúcej úlohy vypočíta ciferný súčet daného čísla.

    - napr.

    ```
    >>> ciferny_sucet(20317)
    13
    ```

16. Napíšte funkciu `len_int(ntica)`, ktorá v danej n-tici ponechá len celé čísla. Funkcia nič nevypisuje len vráti `tuple`.

    - napr.

    ```
    >>> len_int(('12', 2.3, (), 42, (1,), -55))
    (42, -55)
    ```

17. Napíšte funkciu `zip(zoz1, zoz2)`, ktorá dostáva dva rovnako dlhé zoznamy (alebo n-tice) a vráti (`return`) jeden zoznam dvojíc (teda `list`, ktorého prvkami sú `tuple`): v každej dvojici je prvý prvok z prvého `zoz1` a druhý z druhého `zoz2`. Nepoužívajte štandardnú funkciu `zip()`.

    - napr.

    ```
    >>> z = zip(['a', 12, 'bc', 3.1], (1, 3, 5, 7))
    [('a', 1), (12, 3), ('bc', 5), (3.1, 7)]
    ```

18. Napíšte funkciu `ocisluj(zoznam)`, ktorá vráti zoznam dvojíc (`list` s prvkami `tuple`): prvý prvok v dvojici je číslo od 0 do počet prvkov zoznamu - 1 a druhým prvkom je zodpovedajúci prvok daného zoznamu.

    - napr.

```
>>> en = ocisluj(list('Python'))
>>> en
[(0, 'P'), (1, 'y'), (2, 't'), (3, 'h'), (4, 'o'), (5, 'n')]
```

- najprv vyriešte použitím štandardnej funkcie `enumerate()` potom pomocou funkcie `zip()` z predchádzajúcej úlohy

19. Napíšte funkciu `do_dvojic(zoznam)`, ktorá z daného zoznamu (alebo n-tice) vyrobí zoznam dvojíc (`list` s prvkami `tuple`): prvým prvkom dvojice budú postupne prvky daného zoznamu (okrem posledného) a druhým jeho nasledovným prvkom.

    - napr.

    ```
    >>> x = do_dvojic(('11', 22, '3', 4))
    [('11', 22), (22, '3'), ('3', 4)]
    ```

    - riešte najprv bez použitia funkcie `zip()` z predchádzajúcej úlohy (teda pomocou for-cyklu) a potom pomocou tejto funkcie (bez cyklu)

20. Napíšte funkciu `body(n, r, x, y)`, ktorá vráti zoznam dvojíc čísel. Tieto dvojice desatinných čísel reprezentujú daný počet bodov na kružnici s polomerom `r` a so stredom (`x`, `y`). Body sú rovnomerne rozložené na kružnici. Funkcia nič nekreslí ani nevypisuje len vráti `n`-prvkový zoznam súradníc.

    - napr.

    ```
    >>> b = body(7, 100, 150, 120)
    >>> b
    [(...
    ```

21. Napíšte funkciu `uhlopriecky(n, r, x, y)`, ktorá vykreslí (pomocou `tkinter`) všetky strany a uhlopriečky `n`-uholníka, ktorého vrcholy ležia na kružnici s polomerom `r` a so stredom (`x`, `y`). Využite funkciu `body()` z predchádzajúcej úlohy.

    - napr.

    ```
    >>> uhlopriecky(7, 100, 150, 120)
    ```

22. Napíšte funkciu `vektorovy_sucet(nt1, nt2)`, ktorá dostáva dve rovnako veľké n-tice čísel a vráti (`return`) n-ticu, v ktorej každý prvok súčtom zodpovedajúcich prvkov vo vstupných n-ticiach.

    - napr.

    ```
    >>> vektorovy_sucet((1, 2, 3), (4.1, 5.2, 6.3))
    (5.1, 7.2, 9.3)
    ```

23. Napíšte funkciu `indexy(zoznam, hodnota)`, ktorá pre zadaný zoznam (alebo n-ticu) vráti postupnosť indexov (ako `tuple`), na ktorých sa zadaná hodnota v danom zozname vyskytuje.

    - napr.

    ```
    >>> indexy([7, 77, 7, (7, 7), 7], 7)
    (0, 2, 4)
    >>> indexy(('7', '77'), 7)
    ()
    ```

KAPITOLA 10

10. Udalosti v grafickej ploche

Naučíme sa v našich programoch využívať tzv. **udalosti**, ktoré vznikajú v bežiacej grafickej aplikácii buď aktivitami používateľa (klikanie myšou, stláčanie klávesov) alebo operačného systému (tikanie časovača). Na úvod si pripomeňme, čo už vieme o grafickej ploche. Pomocou metód grafickej plochy `Canvas` (definovanej v module `tkinter`) kreslíme grafické objekty:

- `canvas.create_line()` - kreslí úsečku alebo krivku z nadväzujúcich úsečiek
- `canvas.create_oval()` - kreslí elipsu
- `canvas.create_rectangle()` - kreslí obdĺžnik
- `canvas.create_text()` - vypíše text
- `canvas.create_polygon()` - kreslí vyfarbený útvar zadaný bodmi na obvode
- `canvas.create_image()` - kreslí obrázok (prečítaný zo súboru .gif alebo .png)

Ďalšie pomocné metódy manipulujú s už nakreslenými objektami:

- `canvas.delete()` - zruší objekt
- `canvas.move()` - posunie objekt
- `canvas.coords()` - zmení súradnice objektu
- `canvas.itemconfig()` - zmení ďalšie parametre objektu (napr. farba, hrúbka, text, obrázok, ...)

Ďalšie metódy umožňujú postupne zobrazovať vytváranú kresbu:

- `canvas.update()` - zobrazí nové zmeny v grafickej ploche
- `canvas.after()` - pozdrží beh programu o zadaný počet milisekúnd

Udalosť

Udalosťou voláme akciu, ktorá vznikne mimo behu programu a program môže na túto situáciu reagovať. Najčastejšie sú to udalosti od pohybu a klikania myši, od stláčania klávesov, od časovača (vnútorných hodín OS), od rôznych

zariadení, ... V našom programe potom môžeme nastaviť, čo sa má udiať pri ktorej udalosti. Tomuto sa zvykne hovoriť **udalosťami riadené programovanie** (event-driven programming).

Naučíme sa, ako v našich grafických programoch reagovať na udalosti od myši a klávesnice.

Aby grafická plocha reagovala na klikania myšou, musíme ju zviazať (**bind**) s príslušnou udalosťou (**event**).

metóda `bind()`

Táto metóda grafickej plochy slúži na zviazanie niektorej konkrétnej udalosti s nejakou funkciou, ktorá sa bude v programe starať o spracovanie tejto udalosti. Jej formát je:

```
canvas.bind(meno_udalosti, funkcia)
```

kde `meno_udalosti` je znakový reťazec s popisom udalosti (napr. pre kliknutie tlačidlom myši) a `funkcia` je **referencia** na funkciu, ktorá by sa mala spustiť pri vzniku tejto udalosti. Táto funkcia, musí byť definovaná s práve jedným parametrom, v ktorom nám systém prezradí detaily vzniknutej udalosti.

10.1 Klikanie a ťahanie myšou

Ukážeme tieto tri „myšacie“ udalosti:

- **kliknutie** (zatlačenie tlačidla myši) - reťazec `'<Button-1>'` - **1** označuje ľavé tlačidlo myši, **2** by tu znamenala stredné tlačidlo, **3** pravé tlačidlo
- **ťahanie** (posúvanie myšou so zatlačeným tlačidlom) - reťazec `'<B1-Motion>'`
- **pustenie myši** - reťazec `'<ButtonRelease-1>'`

10.1.1 Klikanie myšou

Kliknutie myšou do grafickej plochy vyvolá udalosť s menom `'<Button-1>'`. Ukážme ako vyzerá samotné zviazanie funkcie:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

canvas.bind('<Button-1>', print)
```

Druhý parameter metódy `bind()` musí byť **referencia na funkciu**, ale nie hocijakú, na funkciu, ktorá má jeden parameter. Tu sme použili štandardnú funkciu `print` a keďže do `bind()` treba poslať referenciu na túto funkciu, nesmieme za identifikátor `print` písať zátvorky `()`.

Keď teraz tento malý testovací program spustíte, objaví sa prázdne grafické okno a program čaká, čo sa bude diať (Ak budete tento program spúšťať mimo IDLE, nezabudnite na záver pripísať volanie `mainloop()`, napr. `canvas.mainloop()`). Keďže sme grafickej ploche udalosť kliknutie myšou **zviazali** s funkciou `print()`, každé kliknutie do plochy automaticky vyvolá túto funkciu. Pri klikaní dostávame nejaký takýto výpis (pri každom kliknutí do plochy sa vypíše jeden riadok):

```
<ButtonPress event state=Mod1 num=1 x=194 y=117>
<ButtonPress event state=Mod1 num=1 x=194 y=173>
<ButtonPress event state=Mod1 num=1 x=328 y=31>
<ButtonPress event state=Mod1 num=1 x=17 y=9>
```

Zviazanie udalosti s nejakou funkciou teda znamená, že každé vyvolanie udalosti (kliknutie ľavým tlačidlom myši do grafickej plochy) automaticky zavolá zviazanú funkciu. To, čo nám pritom vypisuje `print()`, je ten jeden parameter, ktorý `tkinter` posiela pri každom jeho zavolaní.

Vytvorme si teraz vlastnú funkciu, ktorú zviažeme s udalosťou kliknutia:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

def klik(parameter):
    print('klik')

canvas.bind('<Button-1>', klik)
```

Vytvorili sme funkciu `klik()`, ktorá sa bude automaticky volať pri každom kliknutí do plochy. Nezabudli sme do hlavičky funkcie pridať jeden formálny parameter, inak by Python pri vzniku udalosti protestoval, že našu funkciu `klik()` chcel zavolať s jedným parametrom a my sme ho nezadeklarovali. Ak by sme tento program spustili, pri každom kliknutí do plochy by sa do textovej plochy shellu mal vypísať text `'klik'`.

Teraz k samotnému parametru v našej funkcii `klik()`: tento parameter slúži na to, aby nám `tkinter` mohol nejakým spôsobom posielať informácie o udalosti. My vieme, že funkcia `klik()` sa zavolá vždy, keď sa niekam klikne, ale nevieme, kde presne do plochy sa kliklo. Práve na toto slúži tento parameter: z neho vieme vytiahnuť, napr. x-ovú a y-ovú súradnicu kliknutého miesta. V nasledovnom príklade vidíme, ako sa to robí. Ešte sme tento `parameter` premenovali na `event` (udalosť po anglicky), aby sme lepšie rozlíšili to, že s týmto parametrom prišla udalosť. Preto tieto súradnice získame ako `event.x` a `event.y`:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

def klik(event):
    print('klik', event.x, event.y)

canvas.bind('<Button-1>', klik)
```

V tomto programe sa pri každom kliknutí vypíšu do shellu aj súradnice kliknutého miesta.

V ďalšom príklade ukážeme, ako využijeme súradnice kliknutého bodu v ploche:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

def klik(event):
    x, y = event.x, event.y
    canvas.create_oval(x-10, y-10, x+10, y+10, fill='red')

canvas.bind('<Button-1>', klik)
```

Teraz sa pri kliknutí nakreslí červený kruh a využijú sa pritom súradnice kliknutého miesta: stred kruhu je kliknuté miesto.

Akcia, ktorá sa vykoná pri kliknutí môže byť veľmi jednoduchá, napr. spájanie kliknutého bodu s nejakým bodom grafickej plochy:

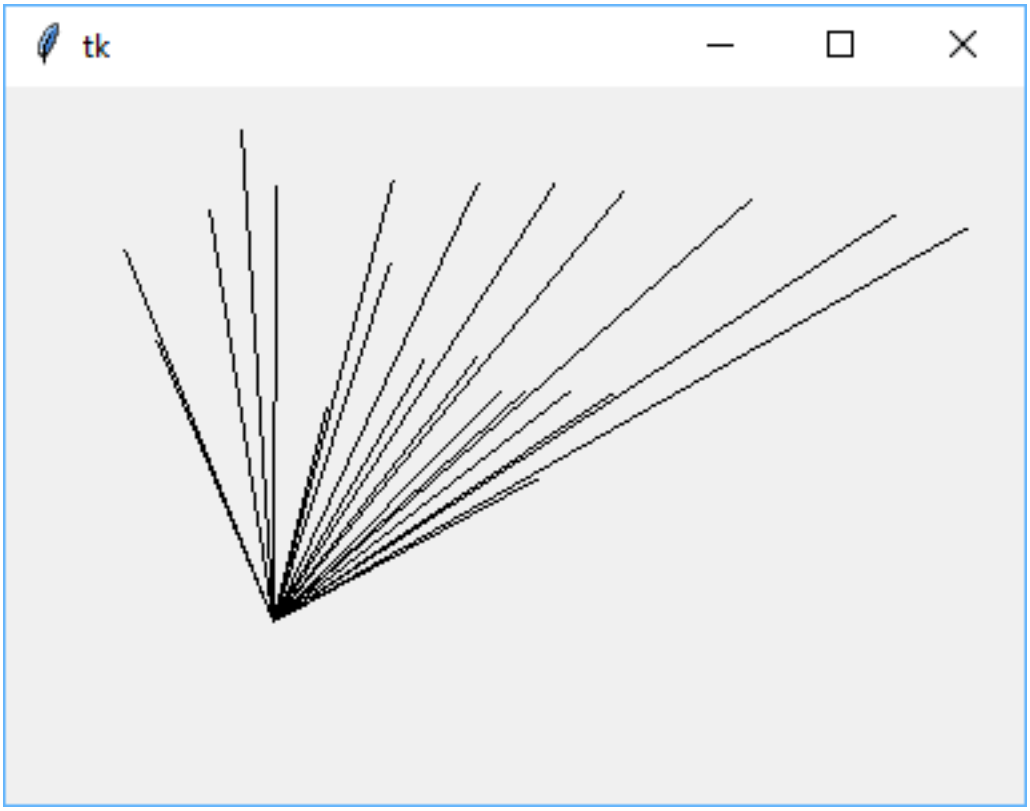
```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

def klik(event):
    canvas.create_line(100, 200, event.x, event.y)

canvas.bind('<Button-1>', klik)
```

Napr.



Ale môžu sa nakresliť aj komplexnejšie kresby, napr. 10 sústredných farebných kruhov:

```
import tkinter
import random

canvas = tkinter.Canvas()
canvas.pack()

def klik(event):
    x, y = event.x, event.y
    for r in range(50, 0, -5):
        farba = f'#{random.randrange(256**3):06x}'
```

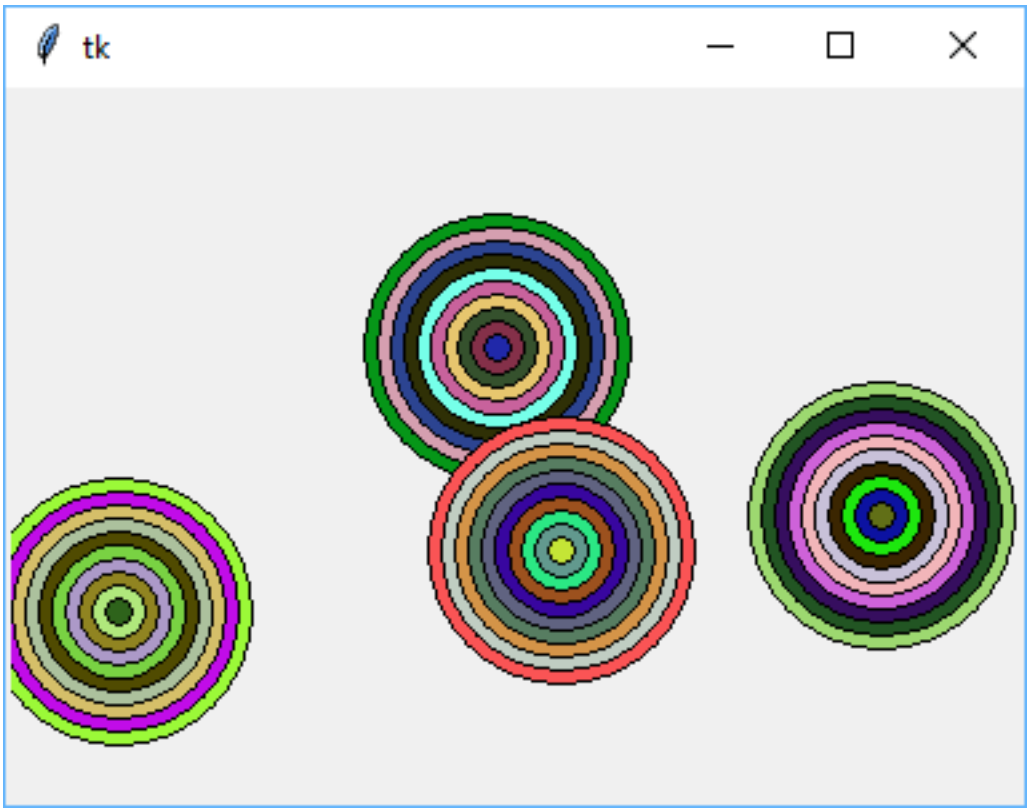
(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        canvas.create_oval(x-r, y-r, x+r, y+r, fill=farba)

canvas.bind('<Button-1>', klik)
```

Napr.



Vráťme sa k príkladu, v ktorom sme kreslili malé krúžky:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

def klik(event):
    x, y = event.x, event.y
    canvas.create_oval(x-5, y-5, x+5, y+5, fill='red')

canvas.bind('<Button-1>', klik)
```

Do tohto programu chceme pridať takéto správanie: tieto kliknuté body (červené krúžky) sa budú postupne spájať úsečkami (zrejme sa bude úsečka kresliť až od druhého kliknutia). Pridáme dve globálne premenné `xx` a `yy`, v ktorých si budeme pamätať predchádzajúci kliknutý bod. Pred prvým kliknutím sme do `xx` priradili `None`, čo bude označovať, že predchádzajúci vrchol ešte nebol:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
xx = yy = None

def klik(event):
    x, y = event.x, event.y
    canvas.create_oval(x-5, y-5, x+5, y+5, fill='red')
    if xx != None:
        canvas.create_line(xx, yy, x, y)
    xx, yy = x, y

canvas.bind('<Button-1>', klik)
```

Žiaľ to nefunguje: po spustení a kliknutí sa dozvieme:

```
...
File ..., line 11, in klik
    if xx != None:
UnboundLocalError: local variable 'xx' referenced before assignment
```

Problémom sú tu **globálne premenné**. Používať globálne premenné vo vnútri funkcii môžeme, len dovtedy, kým ich **nemeníme**. Priraďovací príkaz vo funkcii totiž znamená, že vytvárame novú **lokálnu premennú** (v mennom priestore funkcie `klik()`). Takže Python to v tejto funkcii pochopil takto: do premenných `xx` a `yy` sa vo vnútri funkcie priraďuje nejaká hodnota, takže obe sú lokálne premenné. Keď ale príde vykonávanie funkcie na podmienený príkaz `if xx != None:`, Python už vie, že `xx` je lokálna premenná, ktorá nemá zatiaľ priradenú žiadnu hodnotu. A preto nám oznámil túto chybovú správu: `UnboundLocalError: local variable 'xx' referenced before assignment`.

Takže s globálnymi premennými vo funkcii sa bude musieť pracovať nejako inak. Zrejme, kým do takejto premennej nepotrebujeme vo funkcii nič priradzovať, iba ju používať, problémy nie sú. Problém nastáva vtedy, keď chceme (pomocou priraďovacieho príkazu) meniť obsah globálnej premennej.

Toto nám pomôže vyriešiť nový príkaz `global`:

príkaz `global`

príkaz má tvar:

```
global premenná
global premenná, premenná, premenná, ...
```

Príkaz sa používa vo funkcii vtedy, keď v nej chceme pracovať s globálnou premennou (alebo aj s viac premennými), ale nechceme, aby ju Python vytvoril aj v lokálnom mennom priestore, ale ponechal len v globálnom.

Po doplnení tohto príkazu do predchádzajúceho príkladu všetko funguje tak, ako má:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

xx = yy = -1

def klik(event):
    global xx, yy
    x, y = event.x, event.y
```

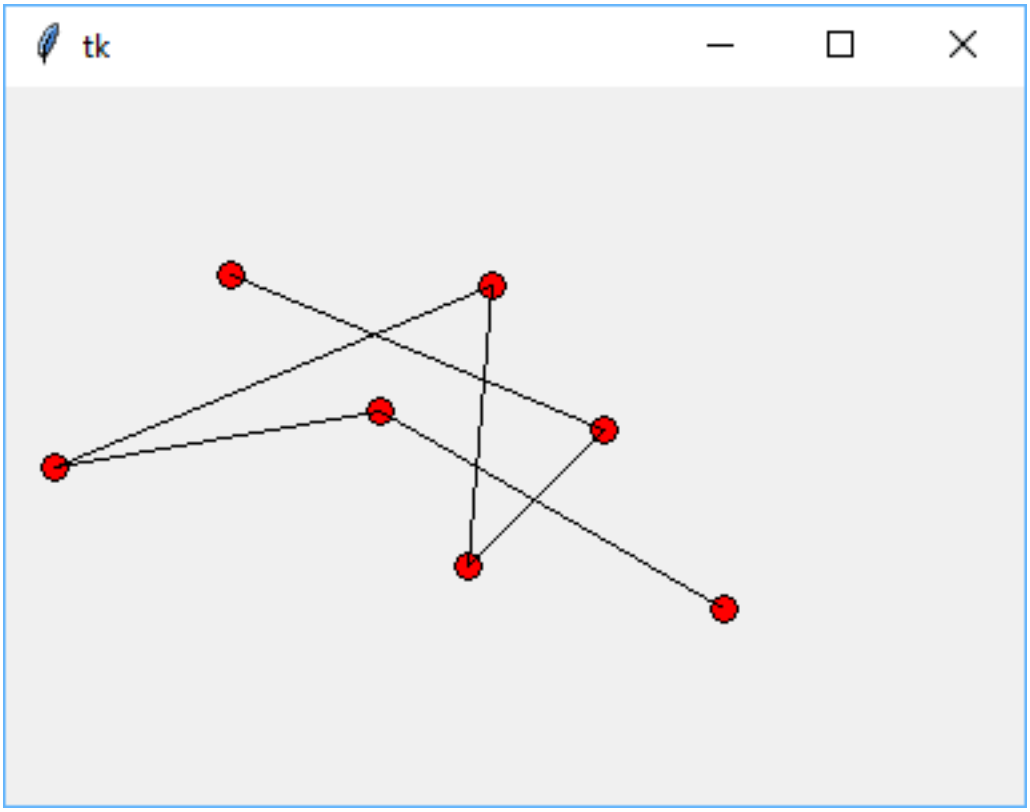
(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    canvas.create_oval(x-5, y-5, x+5, y+5, fill='red')
    if xx >= 0:
        canvas.create_line(xx, yy, x, y)
    xx, yy = x, y

canvas.bind('<Button-1>', klik)
```

Po spustení dostávame takýto obrázok:



Nebezpečné: Príkaz **global** umožňuje modifikovať globálne premenné vo funkciách, teda vlastne robiť **vedľajší účinok** (*side effect*) na globálnych premenných. Toto je ale veľmi **nesprávny spôsob programovania** (*bad programming practice*) a väčšinou svedčí o programátorovi začiatočníkovi, amatérovi.

Kým sa nenaučíme, ako to obísť, budeme to používať, ale veľmi opatrne. Neskôr to využijeme veľmi výnimočne, najmä pri ladení. Správne sa takéto problémy riešia najčastejšie definovaním vlastných tried a použitím atribútov tried.

10.1.2 Ťahanie myšou

Obsluha udalosti ťahanie myšou (pohyb myši so zatlačeným tlačidlom) je veľmi podobná klikaniu. Udalosť má meno `'<B1-Motion>'`. Pozrime, čo sa zmení, keď kliknutie `'<Button-1>'` nahradíme ťahaním `'<B1-Motion>'`:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
def klik(event):
    x, y = event.x, event.y
    canvas.create_oval(x-10, y-10, x+10, y+10, fill='red')

canvas.bind('<B1-Motion>', klik)      # '<B1-Motion>' namiesto '<Button-1>'
```

Funguje to veľmi dobre: pri ťahaní sa na pozícii myši kreslia červené kruhy. Pri pomalom ťahaní sú kruhy nakreslené veľmi nahusto. Často budeme v našich programoch spracovávať obe udalosti: kliknutie aj ťahanie. Niekedy je to tá istá funkcia, inokedy sú rôzne a preto je dobre ich pomenovať zodpovedajúcimi názvami, napr.

```
import tkinter

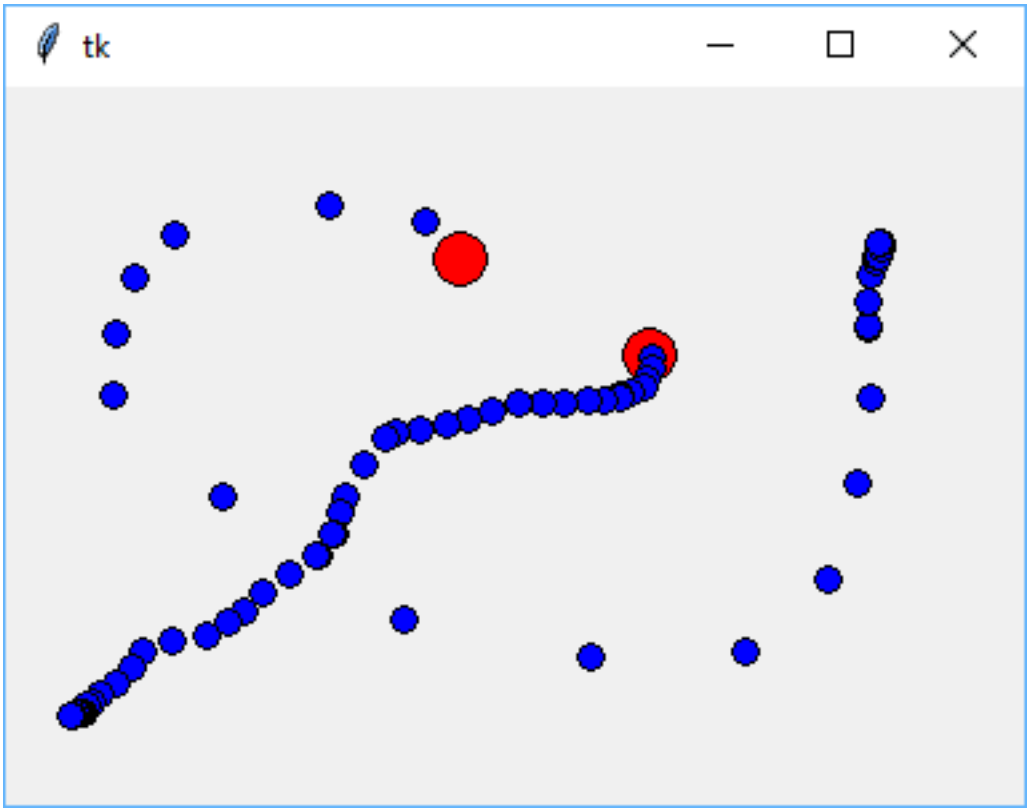
canvas = tkinter.Canvas()
canvas.pack()

def klik(event):
    x, y = event.x, event.y
    canvas.create_oval(x-10, y-10, x+10, y+10, fill='red')

def tahanie(event):
    x, y = event.x, event.y
    canvas.create_oval(x-5, y-5, x+5, y+5, fill='blue')

canvas.bind('<Button-1>', klik)
canvas.bind('<B1-Motion>', tahanie)
```

Pri kliknutí (ešte bez ťahania) sa nakreslí červený kruh, pri ťahaní sa kreslia už len malé modré kruhy.



Na podobnom princípe môžeme upraviť aj kreslenie lúčov z bodu (100, 200) do pozície myši:

```
import tkinter

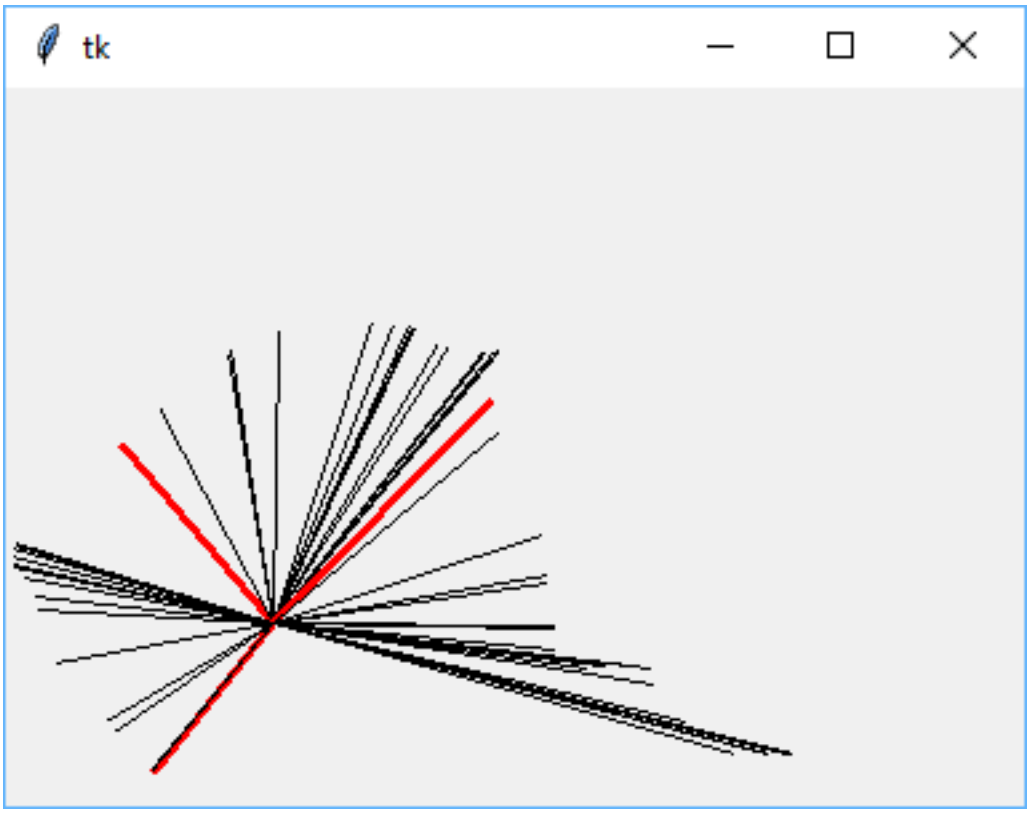
canvas = tkinter.Canvas()
canvas.pack()

def klik(event):
    canvas.create_line(100, 200, event.x, event.y, fill='red', width=3)

def tahanie(event):
    canvas.create_line(100, 200, event.x, event.y)

canvas.bind('<Button-1>', klik)
canvas.bind('<B1-Motion>', tahanie)
```

Pri kliknutí sa nakreslí červená úsečka, pri ťahaní sa kreslia už len čierne.



Alebo malou zmenou kliknutím definujeme pozíciu (globálny bod `(xx, yy)`), z ktorého sa budú kresliť lúče:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

def klik(event):
    global xx, yy
    xx, yy = event.x, event.y

def tahanie(event):
```

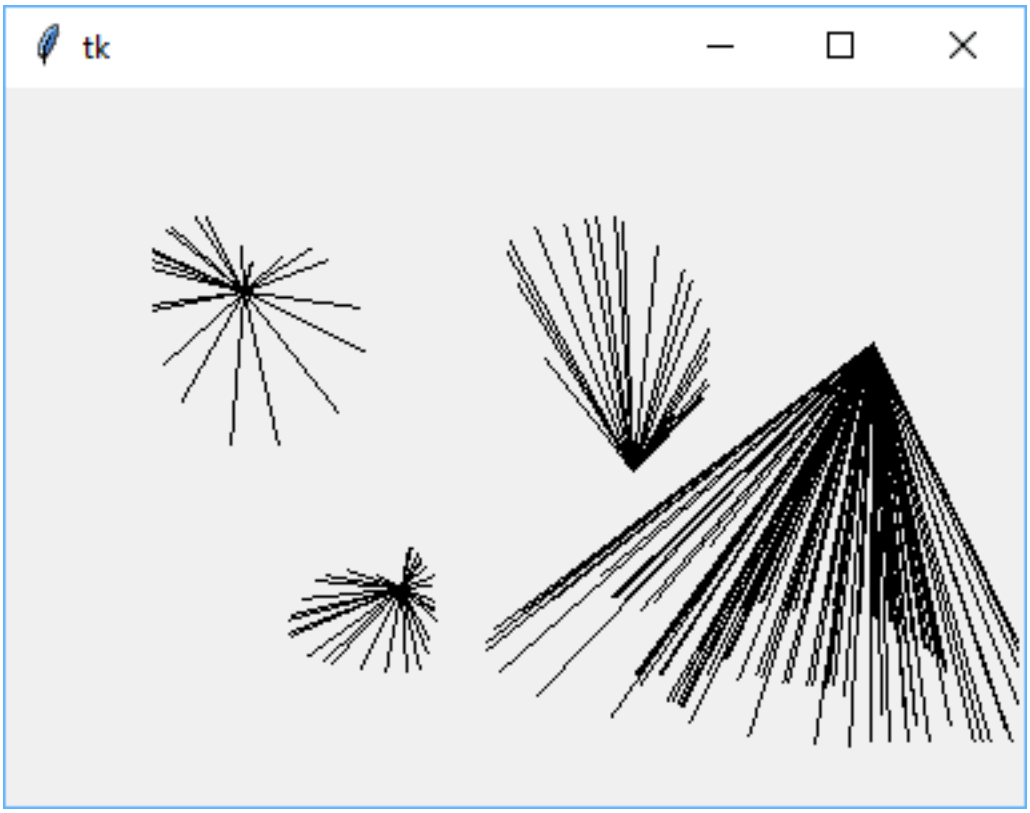
(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    canvas.create_line(xx, yy, event.x, event.y)

canvas.bind('<Button-1>', klik)
canvas.bind('<B1-Motion>', tahanie)
```

Potom dostávame takúto kresbu:



Ďalej nadviažeme na program, v ktorom sme postupne spájali kliknuté body. Pri tomto programe sme využili nový príkaz `global`, aby sme sa dostali ku globálnym premenným. Tento nie najvhodnejší príkaz môžeme obísť, keď využijeme meniteľný (**mutable**) typ zoznam.

Budeme ťahať (ťahaním myši) jednu dlhú lomenú čiaru, pričom si budeme ukladať súradnice prijaté z udalosti do zoznamu čísel:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

zoznam = []
ciara = canvas.create_line(0, 0, 0, 0)

def klik(event):
    zoznam[:] = [event.x, event.y]

def tahanie(event):
    zoznam.extend([event.x, event.y])
    canvas.coords(ciara, zoznam)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
canvas.bind('<Button-1>', klik)
canvas.bind('<B1-Motion>', tahanie)
```

Všimnite si, že vo funkciách používame 3 globálne premenné:

- `canvas` - referencia na grafickú plochu
- `ciara` - identifikátor objektu čiara, potrebujeme ho pre neskoršie menenie postupnosti súradníc príkazom `coords()`
- `zoznam` - zoznam súradníc je meniteľný objekt, teda môžeme meniť obsah zoznamu bez toho, aby sme do premennej `zoznam` priraďovali (priraďujeme buď do rezu alebo voláme metódu `extend()`, t. j. prilep nejakú postupnosť na koniec zoznamu)
  - modifikovanie zoznamu vo funkcii, pričom zoznam nie je parametrom funkcie, tiež nie je najvhodnejším spôsobom programovania, tiež je to nevhodný **vedľajší účinok** podobne ako príkaz `global`, zatiaľ to inak robiť nevieme, tak je to dočasne akceptovateľné

Ťahanie čiary v predchádzajúcom príklade žiaľ kreslí jedinú čiaru: každé ďalšie kliknutie a ťahanie začne kresliť novú čiaru, pričom stará čiara zmizne. Vyriešime to tak, že pri kliknutí začneme kresliť novú čiaru a tú starú necháme tak:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

zoznam = []

def klik(event):
    global ciara
    zoznam[:] = [event.x, event.y]
    ciara = canvas.create_line(0, 0, 0, 0)

def tahanie(event):
    zoznam.extend([event.x, event.y])
    canvas.coords(ciara, zoznam)

canvas.bind('<Button-1>', klik)
canvas.bind('<B1-Motion>', tahanie)
```

Poexperimentujeme:

Malou zmenou dosiahneme veľmi zaujímavý efekt. Vyskúšajte a poštudujte:

```
import tkinter
import random

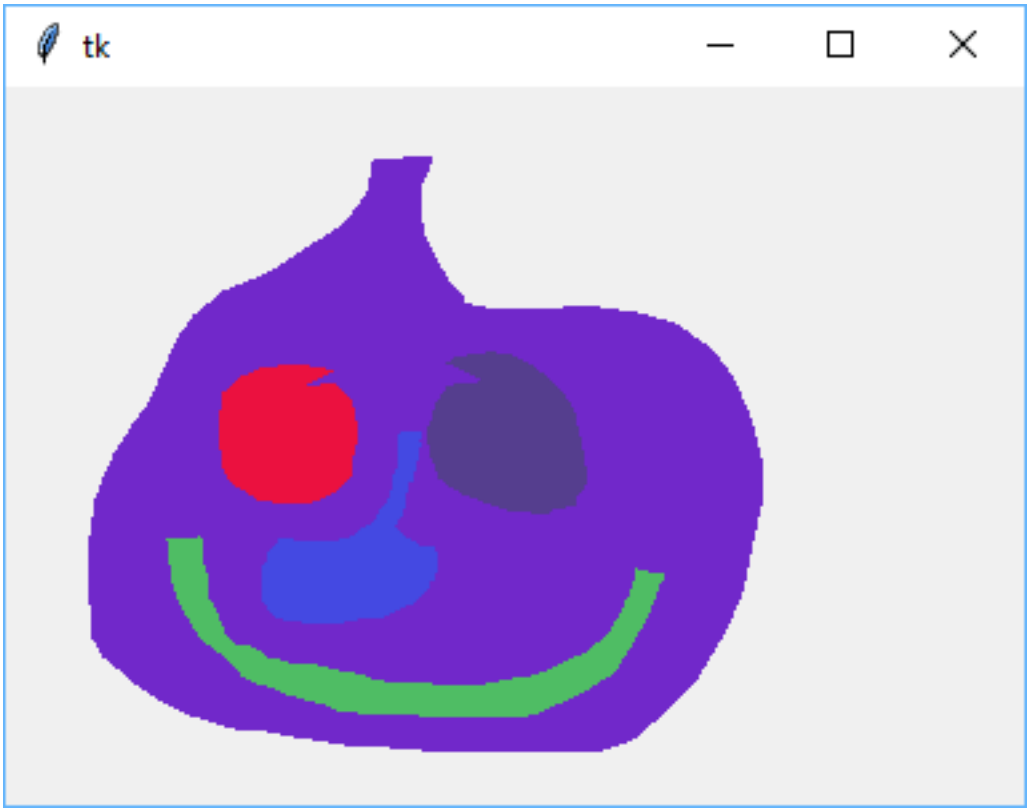
canvas = tkinter.Canvas()
canvas.pack()

zoznam = []

def klik(event):
    global poly
    zoznam[:] = [event.x, event.y]
    farba = f'#{random.randrange(256**3):06x}'
    poly = canvas.create_polygon(0, 0, 0, 0, fill=farba)

def tahanie(event):
    zoznam.extend([event.x, event.y])
    canvas.coords(poly, zoznam)

canvas.bind('<Button-1>', klik)
canvas.bind('<B1-Motion>', tahanie)
```



10.2 Udalosti od klávesnice

Aj každé zatlačenie nejakého klávesu na klávesnici môže vyvolať udalosť. Základnou univerzálnou udalosťou je `'<Key>'`, ktorá sa vyvolá pri každom zatlačení nejakého klávesu. Môžeme otestovať:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

def test(event):
    print(event.keysym)

canvas.bind_all('<Key>', test)
```

Všimnite si, že sme museli zapísať `bind_all()` namiesto `bind()`. Každé zatlačenie nejakého klávesu vypíše jeho reťazcovú reprezentáciu, napr.

```
a
Shift_L
A
Left
Right
Up
Down
Next
Escape
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
Return
F1
```

Pritom každý jeden kláves môže vyvolať aj samostatnú udalosť. Ako meno udalosti treba uviesť meno klávesu (jeho reťazcovú reprezentáciu) v `'<...>'` zátvorkách alebo samostatný znak, napr.

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

def test_vlavo(event):
    print('sipka vlavo')

def test_a(event):
    print('stlacil si klaves a')

canvas.bind_all('a', test_a)
canvas.bind_all('<Left>', test_vlavo)
```

Tento program reaguje len na stláčanie klávesu `'a'` a šípky vľavo. Všetky ostatné klávesy ignoruje.

Často sa samostatné udalosti pre jednotlivé šípky použijú podobne, ako v tomto príklade:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

def kresli(dx, dy):
    global x, y
    x += dx
    y += dy
    zoznam.extend((x, y))
    canvas.coords(ciara, zoznam)

def udalost_vlavo(event):
    kresli(-10, 0)

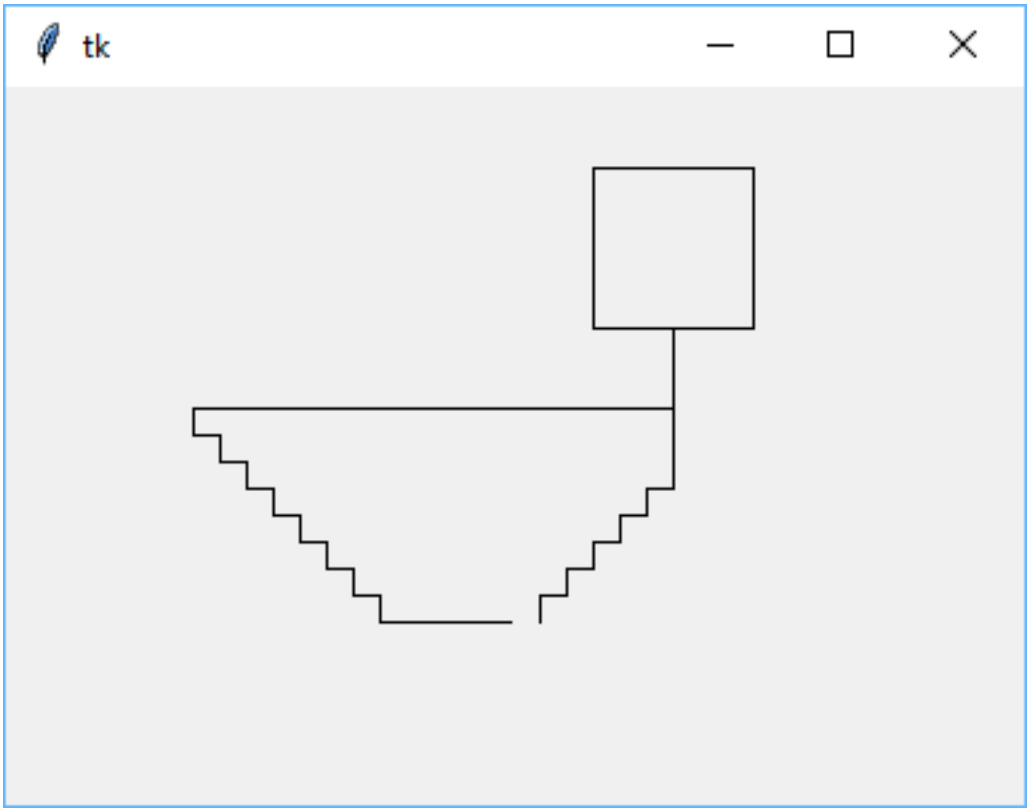
def udalost_vpravo(event):
    kresli(10, 0)

def udalost_hore(event):
    kresli(0, -10)

def udalost_dolu(event):
    kresli(0, 10)

x, y = 200, 200
zoznam = [x, y]
ciara = canvas.create_line(0, 0, 0, 0)         # zatial' prázdna čiara
canvas.bind_all('<Left>', udalost_vlavo)
canvas.bind_all('<Right>', udalost_vpravo)
canvas.bind_all('<Up>', udalost_hore)
canvas.bind_all('<Down>', udalost_dolu)
```

Kreslenie pomocou tohto programu môže pripomínať detskú hračku „magnetická tabuľka“:



10.3 Časovač

Pripomeňme si, ako by sme doteraz v grafickej ploche kreslili krúžky na náhodné pozície s nejakým časovým pozdržaním (napr. 100 ms):

```
import tkinter
import random

canvas = tkinter.Canvas()
canvas.pack()

def kresli():
    while True:
        x = random.randrange(350)
        y = random.randrange(260)
        canvas.create_oval(x-10, y-10, x+10, y+10, fill='red')
        canvas.update()
        canvas.after(100)

kresli()
print('hotovo')
```

Použili sme tu nekonečný cyklus a preto sa príkaz `print()` za volaním `kresli()` s nekonečným while-cyklom už nikdy nevykoná.

Metóda grafickej plochy `after()`, ktorá pozdrží výpočet o nejaký počet milisekúnd, je oveľa všestrannejšia: môžeme pomocou nej štartovať, tzv. **časovač**:

metóda `after()`

Metóda `after()` grafickej plochy môže mať jeden z týchto tvarov:

```
canvas.after(milisekundy)
canvas.after(milisekundy, funkcia)
```

Prvý parameter `milisekundy` už poznáme: výpočet sa pozdrží o príslušný počet milisekúnd. Lenže, ak je metóda zavolaná aj s druhým parametrom `funkcia`, výpočet sa naozaj nepozdrží, ale pozdrží sa vyvolanie zadanej funkcie (parameter `funkcia` musí byť **referencia na funkciu**, teda väčšinou bez okrúhlych zátvoriek). Táto vyvolaná funkcia musí byť definovaná bez parametrov.

S týmto druhým parametrom metóda `after()` naplánuje (niekedy v budúcnosti) spustenie nejakej funkcie a pritom výpočet pokračuje normálne ďalej na ďalšom príkaze za `after()` (bez pozdržania).

Tomuto mechanizmu hovoríme **časovač** (naplánovanie spustenia nejakej akcie), po anglicky **timer**. Najčastejšie sa používa takto:

```
def casovac():
    # príkazy
    canvas.after(cas, casovac)
```

V tomto prípade funkcia naplánuje spustenie samej seba po nejakom čase. Môžete si to predstaviť tak, že v počítači tikajú nejaké hodiny s udanou frekvenciou v milisekundách a pri každom tiknutí sa vykonajú príkazy v tele funkcie.

Najprv jednoduchý test:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

def casovac():
    print('tik')
    canvas.after(1000, casovac)

casovac()
```

Časovač každú sekundu vypíše do textovej plochy reťazec `'tik'`.

Predchádzajúci program s náhodnými červenými krúžkami teraz prepíšeme s použitím časovača:

```
import tkinter
import random

canvas = tkinter.Canvas()
canvas.pack()

def kresli():
    x = random.randrange(350)
    y = random.randrange(260)
    canvas.create_oval(x-10, y-10, x+10, y+10, fill='red')
    #canvas.update()
    canvas.after(100, kresli)

kresli()
print('hotovo')
```

Po spustení funkcie `kresli()` (tá nakreslí jeden kruh a zavolá `after()`, t. j. naplánuje ďalšie kreslenie) sa pokra-

čuje ďalším príkazom, t. j. sa vypíše `print('hotovo')`, program končí a v shelli môžeme zadávať ďalšie príkazy. Pritom ale stále beží náš rozbehnutý **časovač**. V shelli by sme mohli napr. zapísať:

```
>>> canvas.delete('all')
```

Tento príkaz vymaže grafickú plochu, ale bežiaci časovač (rozbehnutá funkcia `kresli()`) do nej bude pokračovať v kreslení červených krúžkov.

Keďže počas behu časovača môže program vykonávať ďalšie akcie, môže spustiť hoci aj ďalší časovač. Zapíšme:

```
import tkinter
import random

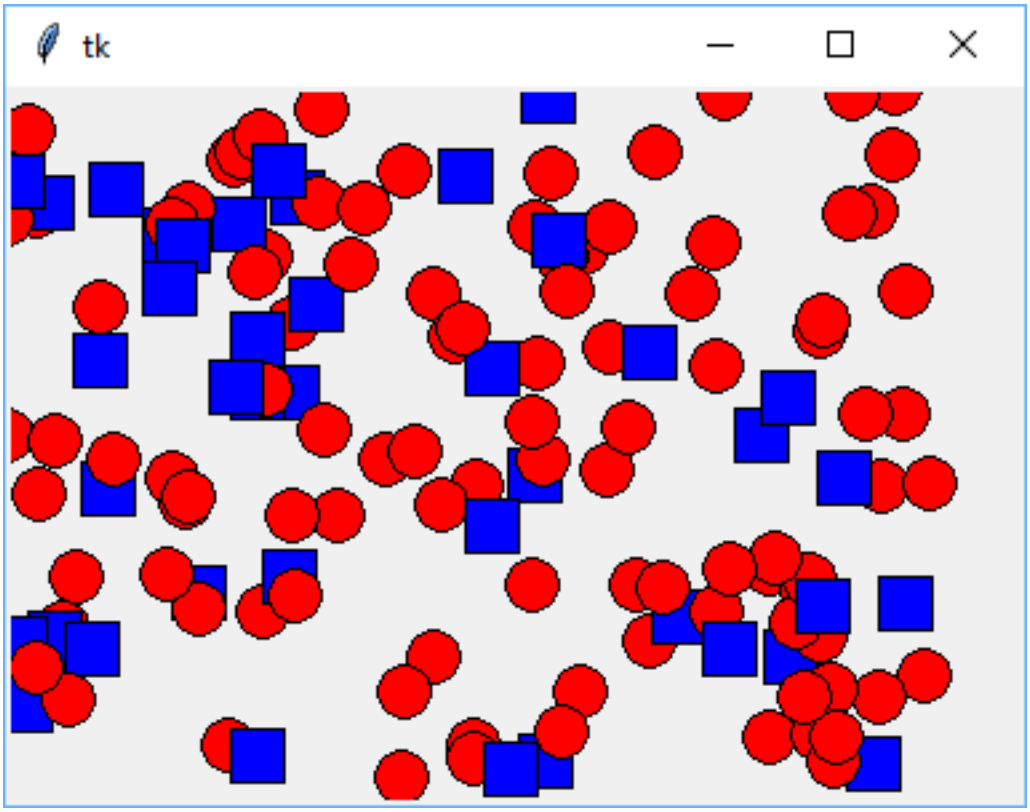
canvas = tkinter.Canvas()
canvas.pack()

def kresli():
    x = random.randrange(350)
    y = random.randrange(260)
    canvas.create_oval(x-10, y-10, x+10, y+10, fill='red')
    canvas.after(100, kresli)

def kresli1():
    x = random.randrange(350)
    y = random.randrange(260)
    canvas.create_oval(x-10, y-10, x+10, y+10, fill='blue')
    canvas.after(300, kresli1)

kresli()
kresli1()
print('hotovo')
```

Program teraz spustí oba časovače: kreslia sa červené krúžky a modré štvorčeky. Keďže druhý časovač má svoj interval 300 milisekúnd, teda „tiká“ 3-krát pomalšie ako prvý, kreslí 3-krát menej modrých štvorčekov ako prvý časovač červených krúžkov, napr.



10.3.1 Zastavovanie časovača

Na zastavenie časovača nemáme žiaden príkaz. Časovač môžeme zastaviť len tak, že on sám v svojom tele na konci nezavolá metódu `canvas.after()` a tým aj skončí. Upravíme predchádzajúci príklad tak, že zadefinujme dve globálne premenné, ktoré budú slúžiť pre oba časovače na zastavovanie:

```
import tkinter
import random

canvas = tkinter.Canvas()
canvas.pack()

def kresli():
    x = random.randrange(350)
    y = random.randrange(260)
    canvas.create_oval(x-10, y-10, x+10, y+10, fill='red')
    if bezi:
        canvas.after(100, kresli)

def kresli1():
    x = random.randrange(350)
    y = random.randrange(260)
    canvas.create_rectangle(x-10, y-10, x+10, y+10, fill='blue')
    if bezi1:
        canvas.after(300, kresli1)

bezi = bezi1 = True
kresli()
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
kresli1()
print('hotovo')
```

Teraz bežia oba časovače, ale stačí zavolať, napr.

```
>>> bezi = False
```

V tomto momente sa prvý časovač zastaví a beží iba druhý, teda sa kreslia len modré štvorčeky. Ak by sme chceli znovu naštartovať prvý časovač, nesmieme zabudnúť zmeniť premennú `bezi` a zavolať `kresli()`:

```
>>> bezi = True
>>> kresli()
```

Opäť bežia súčasne oba časovače.

Globálne premenné môžeme využiť aj na iné účely: môžeme nimi meniť „parametre“ bežiacich príkazov. Napr. farbu krúžkov, ale aj interval tikania časovača, napr.

```
import tkinter
import random

canvas = tkinter.Canvas()
canvas.pack()

def kresli():
    x = random.randrange(350)
    y = random.randrange(260)
    canvas.create_oval(x-vel, y-vel, x+vel, y+vel, fill=farba)
    if bezi:
        canvas.after(cas, kresli)

bezi = True
farba = 'red'
cas = 100
vel = 10
kresli()
```

Počas behu tohoto časovača ho môžeme nielen zastavovať, ale meniť mu rýchlosť (zmenou premennej `cas`), farbu aj veľkosť krúžkov, napr.

```
>>> vel, farba = 30, 'blue'
```

Od tohto momentu sa kreslia modré ale väčšie krúžky.

```
>>> cas = 30
```

Zrýchli časovač na 30 milisekúnd.

Ďalšia ukážka bude hýbať dvoma obrázkami autíčok (napr. auto1.png a auto2.png) rôznou rýchlosťou:

```
import tkinter
import random

canvas = tkinter.Canvas(width=600)
canvas.pack()

obr_auto1 = tkinter.PhotoImage(file='auto1.png')
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

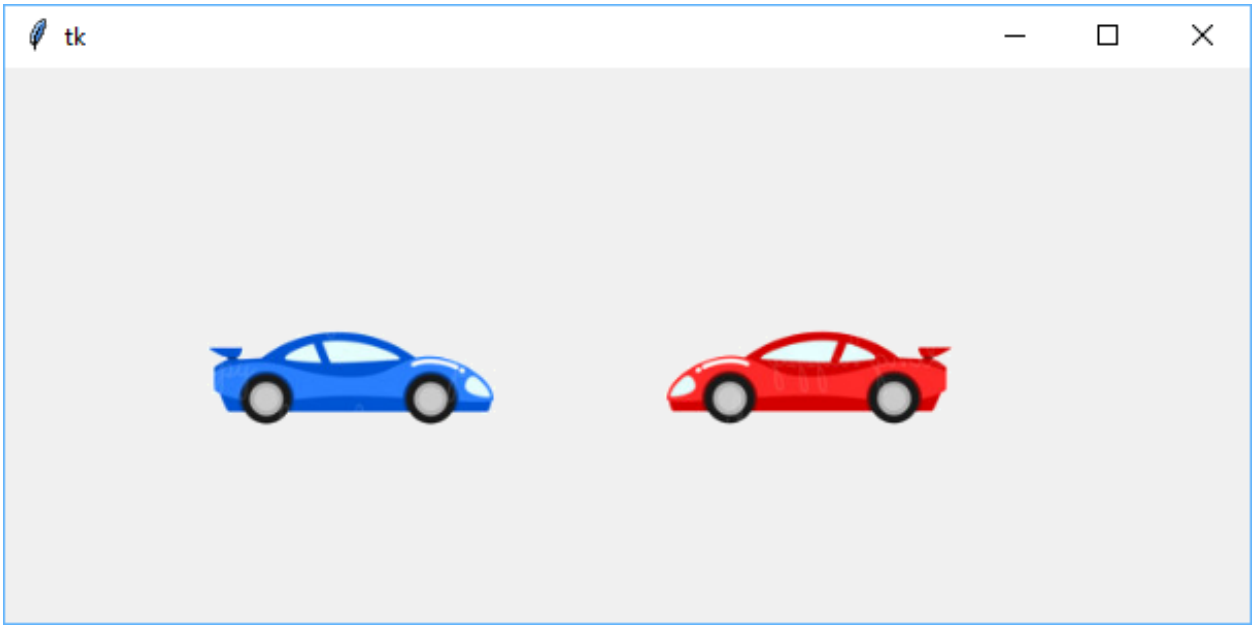
```
obr_auto2 = tkinter.PhotoImage(file='auto2.png')

auto1 = canvas.create_image(0, 150, image=obr_auto1)
auto2 = canvas.create_image(600, 150, image=obr_auto2)

def pohyb():
    canvas.move(auto1, 4, 0)
    canvas.move(auto2, -5, 0)
    canvas.after(30, pohyb)

pohyb()
```

Program rozbehne dve autíčka proti sebe, ale nekontroluje, či sa zrazia:



Aby sa autá nerozbehli už pri štarte programu, ale až po kliknutí do plochy, zapíšeme:

```
import tkinter
import random

canvas = tkinter.Canvas(width=600)
canvas.pack()

obr_auto1 = tkinter.PhotoImage(file='auto1.png')
obr_auto2 = tkinter.PhotoImage(file='auto2.png')

auto1 = canvas.create_image(0, 150, image=obr_auto1)
auto2 = canvas.create_image(600, 150, image=obr_auto2)

def start(event):
    canvas.coords(auto1, 0, 150)
    canvas.coords(auto2, 600, 150)
    pohyb()

def pohyb():
    canvas.move(auto1, 4, 0)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    canvas.move(auto2, -5, 0)
    canvas.after(30, pohyb)

canvas.bind('<Button-1>', start)
```

Teraz chceme pridať kontrolu, či sa už obe autá zrazili. Vtedy zastavíme časovač a vypíšeme nejakú správu. Keďže si nikde neuchovávame momentálnu pozíciu áut, využijeme metódu `coords()`, ktorá okrem zmeny súradníc grafického objektu, dokáže zistiť momentálne jeho súradnice. Napr. ak počas behu časovača zapíšeme:

```
>>> canvas.coords(auto1)
[164.0, 150.0]
>>> canvas.coords(auto2)
[395.0, 150.0]
```

Vidíme, že táto funkcia nám vracia polohu autíčka (súradnice stredu obrázka), preto môžeme zastaviť časovač testovaním x-ových súradníc autíčok:

```
import tkinter
import random

canvas = tkinter.Canvas(width=600)
canvas.pack()

obr_auto1 = tkinter.PhotoImage(file='auto1.png')
obr_auto2 = tkinter.PhotoImage(file='auto2.png')

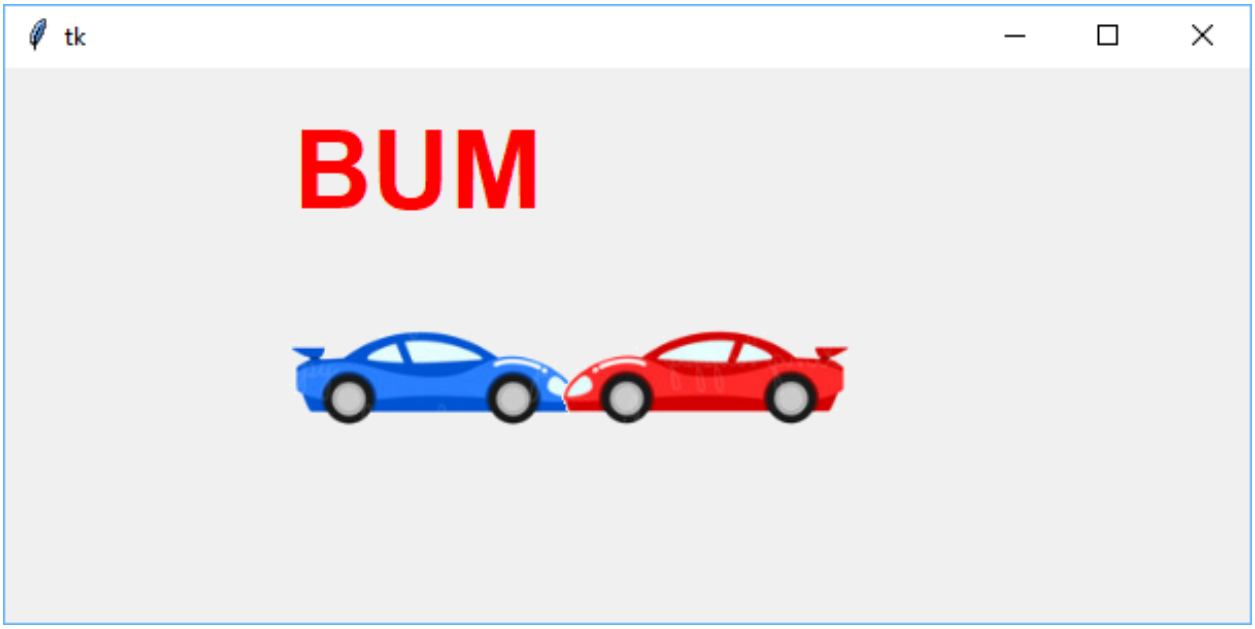
auto1 = canvas.create_image(0, 150, image=obr_auto1)
auto2 = canvas.create_image(600, 150, image=obr_auto2)

def start(event):
    canvas.coords(auto1, 0, 150)
    canvas.coords(auto2, 600, 150)
    pohyb()

def pohyb():
    canvas.move(auto1, 4, 0)
    canvas.move(auto2, -5, 0)
    x_auto1 = canvas.coords(auto1)[0]
    x_auto2 = canvas.coords(auto2)[0]
    if x_auto1 > x_auto2 - 140:
        canvas.create_text(200, 50, text='BUM', fill='red', font='arial 40 bold')
    else:
        canvas.after(30, pohyb)

canvas.bind('<Button-1>', start)
```

Pri stretnutí autíčok dostávame:



Hoci tento program funguje dobre, má niekoľko malých nedostatkov:

- ak počas pohybu autíčok (beží časovač `pohyb()`) znovu klikneme do plochy (vyvoláme udalosť `start()`), autíčka skočia do svojich štartových pozícií a znovu sa vyvolá časovač `pohyb()`; teraz to ale vyzerá, že autíčka idú dvojnásobnou rýchlosťou - totiž teraz bežia naraz dva časovače `pohyb()` aj `pohyb()`, ktoré oba pohnú oboma autíčkami - hoci je to zaujímavé, budeme sa snažiť tomuto zabrániť
- ak autíčka do seba nabúrajú, vypíše sa text `'BUM'`, ktorý tam bude svietiť aj po opätovnom naštartovaní autíčiek

Vylepšíme funkciu `start()` takto:

- keďže táto štartuje časovač `pohyb()`, zablokujeme opätovné kliknutie tým, že zrušíme zviazanie udalosti klik s funkciou `start()`
- ak svieti text `'BUM'`, tak ho vymažeme

Využijeme nový príkaz `unbind()`, pomocou ktorého vieme rozviazať nejakú konkrétnu udalosť s funkciou:

metóda `unbind()`

Metóda zruší zviazanie príslušnej udalosti:

```
canvas.unbind(meno_udalosti)
```

V časovači (vo funkcii) `pohyb()` pri výpise správy `'BUM'`, keďže zastavujeme časovač, opätovne zviažeme kliknutie do plochy s udalosťou `start()`:

```
import tkinter
import random

canvas = tkinter.Canvas(width=600)
canvas.pack()

obr_auto1 = tkinter.PhotoImage(file='auto1.png')
obr_auto2 = tkinter.PhotoImage(file='auto2.png')
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
auto1 = canvas.create_image(0, 150, image=obr_auto1)
auto2 = canvas.create_image(600, 150, image=obr_auto2)

text = None

def start(event):
    canvas.unbind('<Button-1>')              # zruší klikaciu udalost'
    canvas.coords(auto1, 0, 150)
    canvas.coords(auto2, 600, 150)
    canvas.delete(text)
    pohyb()

def pohyb():
    global text
    canvas.move(auto1, 4, 0)
    canvas.move(auto2, -5, 0)
    x_auto1 = canvas.coords(auto1)[0]
    x_auto2 = canvas.coords(auto2)[0]
    if x_auto1 > x_auto2 - 140:
        text = canvas.create_text(200, 50, text='BUM', fill='red', font='arial 40 bold
↪')
        canvas.bind('<Button-1>', start)     # obnoví klikaciu udalost'
    else:
        canvas.after(30, pohyb)

canvas.bind('<Button-1>', start)
```

10.4 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/
- vo všetkých úlohách zostavíte program, ktorý bude pracovať s grafickou plochou, preto najprv vytvoríte `canvas` vhodnej veľkosti a farby

Klikanie myšou

1. Na pozícii kliknutia myšou sa vypíšu súradnice kliknutého bodu v tvare `(120, 151)`
   - zvoľte malý font
2. Máme dané globálny zoznam reťazcov. Na každé kliknutie myšou sa na danej pozícii vypíše prvý reťazec zo zoznamu a zároveň sa z neho odstráni (metódou `pop()`)
   - reťazce vypisujte do plochy nejakým väčším fontom
   - keď je zoznam už prázdny, klikanie do plochy nerobí nič (v shelli môžeme do zoznamu priradiť nové reťazce)
   - napr.

```
>>> zoznam = list('PYTHON')
>>> ... # 6-krát klikneme do plochy
>>> zoznam = 'python java pascal c++ ruby php logo javascript'.split()
>>> ... # klikáme do plochy
```

3. Do plochy nakreslite tesne vedľa seba 8 štvorcov veľkosti 50x50. Na každé kliknutie sa príslušný kliknutý štvorec zafarbí na červeno (alebo na nejakú náhodnú farbu)
   - poradové číslo štvorca v rade zistite z x-ovej súradnice kliknutého bodu `event.x // 50`, kontrolujte, či aj `event.y` je v správnom intervale, t. j. kliklo sa do vnútra niektorého štvorca a nie niekde mimo
4. Predchádzajúci príklad doplňte tak, aby klikanie v každom štvorci striedalo 3 farby: `('white', 'blue', 'red')`
   - napr. na začiatku sú všetky biele, ak teraz postupne na všetky klikneme raz, budú sa prefarbovať na modro
5. V zozname máme čísla 0 až `n` v náhodnom poradí
   - napr.

   ```
   zoznam = [3, 0, 1, 2]
   ```

   Tento zoznam reprezentuje takýto hlavolam: máme `n+1` políčok (štvorčekov) v každom okrem jedného je číslo od 1 do `n`. Políčko s 0 vykreslíme ako prázdne ostatné so zodpovedajúcim číslom v strede. Kliknutie do jedného zo štvorčekov s číslom presťahuje toto číslo na voľné políčko (zároveň v zozname vymení 0 a toto kliknuté číslo). Cieľom riešenia hlavolamu je preusporiadať čísla tak aby v prvých `n` políčkach boli postupne čísla 1 až `n` a posledné políčko ostalo prázdne: vtedy program vypíše ‚HURA‘.

Ťahanie myšou

6. Gumená úsečka: kliknutie naštartuje vytváranie úsečky (prvý aj druhý bod úsečky je kliknutý bod, t. j. jej veľkosť je 0), ťahanie aktualizuje jej druhý bod (použite metódu `canvas.coords()`)
   - každé ďalšie kliknutie a ťahanie vytvára ďalšiu úsečku
7. Gumený obdĺžnik: kliknutie naštartuje vytváranie obdĺžnika (jeden vrchol je kliknutý bod a veľkosť je zatiaľ 0x0), ťahanie aktualizuje jeho veľkosť, t.j. protiľahlý vrchol
   - kliknutie nastaví náhodnú farbu výplne tohto obdĺžnika
   - každé ďalšie kliknutie a ťahanie vytvára ďalší obdĺžnik
   - otestujte tento program s „gumenou elipsou“:- namiesto `create_rectangle()` dajte `create_oval()`
8. V ploche sa nachádza jeden červený štvorček veľkosti 50x50. Keď klikneme do jeho vnútra (počíta sa aj obvod), môžeme ho ťahať, teda posúvať po ploche pomocou pohybov myši (inak ťahanie s kliknutím mimo štvorček nerobí nič).
   - dajte pozor, aby aj malé posunutie myši počas ťahania neurobilo jeho neúmerne veľký skok (môžeme ho chytiť a ťahať napr. aj za ľubovoľný vrchol)
9. Predchádzajúci príklad upravte tak, aby fungoval aj pre 2 rôzne veľké štvorce: jeden červený veľkosti 50x50, druhý modrý veľkosti 100x100
   - vedeli by ste tento program upraviť tak, aby fungoval pre ľubovoľný počet štvorcov v ploche?
10. V ďalšom programe bude kliknutie aj ťahanie robiť to isté:
    - funkcia bude využívať tieto globálne premenné:

```
pocet = 20
farba = 'green'
vzd = 20
```

- v okolí kliknutého bodu sa náhodne vygeneruje `pocet` malých farebných krúžkov (`create_oval(x-2, y-2, x+2, y+2, fill=farba, outline='')`), tieto náhodne vygenerované body majú `x`, resp. `y` v intervale plus mínus `vzd` od kliknutého bodu (napr. `x=event.x+random.randrange(-vzd, vzd+1)`)
- takto by mal vzniknúť efekt spreja (experimentujte s rôznymi hodnotami týchto troch premenných)

Klávesnica

11. Približne v strede sa zobrazí nejaký obrázok (nájdite na internete nejaký malý `.png` súbor), stláčaním šípok sa celý obrázok posúva daným smerom
    - každé stlačenie klávesu šípky posunie obrázok daným smerom o 10 pixelov
    - otestujte, či sa bude automaticky posúvať aj vtedy, keď kláves šípky podržíme zatlačený dlhšie
12. Stláčaním malých a veľkých písmen abecedy sa tieto vypisujú nejakým fontom pod seba. Keď už by to malo presiahnuť spodný okraj plochy, pokračuje sa odhora ale celý ďalší výpis je trochu posunutý vpravo
    - použite metódu `bind_all('<Key>', ...)` pričom vo viazanej funkcii pracujte s hodnotou `event.char`

Časovač

13. V globálnej premennej `farba = 'red'` je nastavená nejaká farba, nastavte časovač, ktorý každú 0.05 sekundy vypíše na náhodnú pozíciu náhodné písmeno (od `'a'` do `'z'`) s danou farbou (zvoľte vhodný font)
    - počas behu časovača, meňte nastavenú farbu a sledujte, čo sa robí
14. Do riešenia predchádzajúcej úlohy dorobte ďalší časovač, ktorý každých 5 sekúnd zmení obsah premennej `farba` na náhodnú farbu
    - porozmýšľajte aj o treťom časovači (napr. každých 8 sekúnd), ktorý zmení veľkosť vykresľovaných písmen na náhodné číslo od 10 do 30
15. V ploche je nakreslených 10 sústredných kružníc s náhodnými farbami a s polomermi 50, 45, 40, ... (ako bolo na prednáške), ich stred je (50, 100). Časovač všetky tieto kruhy pomaly (0.1 sekundy) posúva vpravo s krokom 1.
    - použite metódu `canvas.move()`
      - aby ste posúvali jedným `move()` naraz všetky nakreslené kružnice, môžete im pridať ďalší parameter

        `canvas.create_oval(..., tag='kruhy')`

        kde reťazec pre pomenovaný parameter `tag` môžete uviesť ľubovoľný text, potom volanie

        `canvas.move('kruhy', 1, 0)`

        posunie všetky grafické objekty, ktoré majú nastavený `tag` na reťazec `'kruhy'`
    - časovač sa pri vykreslení kružníc na pravom okraji plochy sám zastaví
16. V ploche je farebný štvorček, po kliknutí myšou (hocikde v ploche) sa začne posúvať smerom vpravo, keď narazí na okraj plochy posúvanie sa otočí opačným smerom, takto sa otočí aj na ľavom okraji, atď. Každé ďalšie kliknutie myšou buď zastaví alebo rozbehne posúvanie štvorčeka

- zastavené posúvanie si pamätá smer a opätovné kliknutie ho rozbehne v danom smere

17. Nechajte bežať na obrazovke veľké digitálky: čas je zobrazený v tvare `'17:22:34.5'` a mení sa každú 0.1 sekundu
    - použite jeden textový objekt (`create_text()`), ktorému pomocou `itemconfig()` meníte zobrazovanú hodnotu
18. Nakreslite úsečku (150, 150, 150, 50); táto sa bude pomaly otáčať okolo bodu (150, 150), tak aby sa každú sekundu otočila o uhol 6 stupňov vpravo (za minútu sa teda otočí o 360 stupňov)
    - môžete to testovať aj s kratším časovým intervalom, napr. s 0.1 sekundy
19. Naprogramujte takúto hru na postreh:
    - každých `interval` milisekúnd sa farebný kruh s polomerom `r` presunie na náhodnú pozíciu v ploche
    - keď klikneme do plochy a trafíme do vnútra kruhu, ku nášmu skóre sa pripočíta 1
    - keď klikneme do plochy, ale netrafíme do kruhu, skóre sa zníži o 1
    - aktuálne skóre sa vypisuje niekde v rohu obrazovky (ako grafický objekt `create_text()`)
    - `interval` a `r` sú nejaké globálne premenné, napr. s hodnotami 500 a 20
20. Na mieste kliknutia myšou sa nakreslí kruh s polomerom 50 a s náhodnou farbou výplne, ďalších 50x sa jej polomer zmenší o 1 s časovým intervalom 0.1 sekundy, keď bude polomer 0, časovač sa zastaví (nebude pokračovať v zmenšovaní kružnice)
    - zabezpečte, aby aj viac kliknutí tesne za sebou na rôzne miesta plochy paralelne vytvorilo viac kruhov a aby sa všetky postupne zmenšovali o 1 (každé kliknutie vytvorí nový časovač s vlastnou kružnicou - treba dať pozor na lokálne a globálne premenné)
21. V strede plochy je malý útvar (obrázok, alebo štvorček, alebo text, ...). Ďalej máme dve globálne premenné `dx=dy=0`. Po stlačení jednej zo šípok sa útvar začne pohybovať daným smerom, t. j. príslušná premenná `dx` alebo `dy` sa zvýši alebo zníži o 1 (podľa zatlačeného smeru šípky) a útvar sa bude posúvať o momentálne `(dx, dy)`.
    - na okrajoch plochy útvar zastane, t. j. nezrealizuje posunutie, ktoré by ho dostalo von z plochy
    - aj keď útvar stojí na mieste, môžeme stláčať šípky a tým mu meniť `(dx, dy)` a teda ho môžeme aj rozbehnúť

KAPITOLA 11

11. Korytnačky (turtle)

Dnes sa naučíme v Pythone pracovať trochu s inou grafikou ako sa pracovalo pomocou `tkinter`. Napr.

```
import tkinter
canvas = tkinter.Create()                    # vytvor grafickú plochu
canvas.pack()                                # zobraz ju do okna
g.create_oval(100, 50, 150, 80, fill='red')  # nakresli červenú elipsu
...
```

Pri programovaní takýchto úloh sme počítali s tým, že:

- počiatočný bod (0, 0) je v ľavom hornom rohu grafickej plochy
- x-ová súradnica ide vpravo vodorovne po hornej hrane grafickej plochy
- y-ová súradnica ide nadol zvislo po ľavej hrane grafickej plochy: smerom nadol sú kladné y-ové hodnoty, smerom nahor idú záporné hodnoty súradníc
- všetky kreslené útvary sú umiestnené absolútne k bodu (0, 0)
- otáčanie útvaru o nejaký uhol, resp. rozmiestňovanie bodov na kružnici môže byť dosť náročné a väčšinou vyžaduje použitie `math.sin()` a `math.cos()`

11.1 Korytnačia grafika

Korytnačka je grafické pero (grafický robot), ktoré si okrem pozície v grafickej ploche (`pos()`) pamätá aj smer natočenia (`heading()`). Korytnačku vytvárame podobne ako v `tkinter` grafickú plochu:

```
>>> import turtle
>>> t = turtle.Turtle()
>>> t.pos()
(0.00,0.00)
>>> t.heading()
0.0
```

Príkaz `t = turtle.Turtle()` vytvorí grafickú plochu a v jej strede korytnačku s natočením na východ. Volanie `t.pos()` vráti momentálnu pozíciu korytnačky (0, 0) a `t.heading()` vráti uhol 0.

Pre grafickú plochu korytnačej grafiky platí:

- súradná sústava má počiatok v strede grafickej plochy
- x-ová súradnica ide vpravo vodorovne od počiatku
- y-ová súradnica ide nahor zvislo od počiatku: smerom nahor sú kladné y-ové hodnoty, smerom nadol idú záporné hodnoty súradníc
- smer natočenia určujeme v stupňoch (nie v radiánoch) a v protismere otáčania hodinových ručičiek:
  - na východ je to 0
  - na sever je to 90
  - na západ je to 180
  - na juh je to 270
- pozícia a smer korytnačky je vizualizovaná malým čiernym trojuholníkom - keď sa bude korytnačka hýbať alebo otáčať, bude sa hýbať tento trojuholník.

Základnými príkazmi sú `forward(dĺžka)`, ktorý posunie korytnačku v momentálnom smere o zadanú dĺžku a `right(uhol)`, ktorý otočí korytnačku o zadaný uhol vpravo, napr.

```
>>> import turtle
>>> t = turtle.Turtle()
>>> t.forward(100)
>>> t.right(90)
>>> t.forward(100)
>>> t.right(90)
>>> t.forward(100)
>>> t.right(90)
>>> t.forward(100)
```

nakreslí štvorec so stranou 100. Častejšie budeme používať skrátené zápisy týchto príkazov: `fd()`, `rt()` a `lt()` (je skratkou príkazu `left()`, teda otočenie vľavo).

mainloop()

Takýto spôsob postupného testovania korytnačích príkazov a programov s korytnačkami nemusí fungovať mimo **IDLE**. Niektorí z vás už máte skúsenosti, že pri práci s grafikou musíte na záver programu pripísať `canvas.mainloop()`. Pri práci s korytnačou grafikou Python tiež využíva `tkinter`, preto aj v tomto prípade musíme na záver programu zadať napr.

```
turtle.mainloop()
```

Pripadne môžeme namiesto tohto zapísať

```
turtle.exitonclick()
```

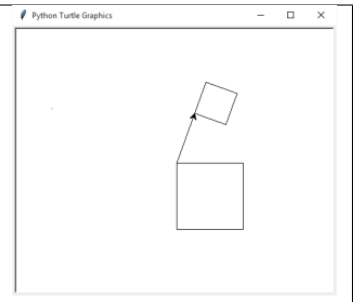
čo bude označovať, že grafické okno sa automaticky zatvorí po kliknutí niekam do okna.

Zapíšme funkciu, pomocou ktorej korytnačka nakreslí štvorec:

```
import turtle

def stvorec(dlzka):
    for i in range(4):
        t.fd(dlzka)
        t.rt(90)

t = turtle.Turtle()
stvorec(100)
t.lt(70)
t.fd(80)
stvorec(50)
```



Program nakreslí dva rôzne štvorce - druhý je posunutý a trochu otočený.

Aby sa nám ľahšie experimentovalo s korytnačkou, nemusíme stále reštartovať shell z programového režimu (napr. klávesom **F5**). Keď máme vytvorenú korytnačku `t`, stačí zmazať kresbu pomocou:

```
>>> t.clear()      # zmaže grafickú plochu a korytnačku nechá tam, kde sa momentálne␣
↪nachádza
```

alebo pri zmazaní plochy aj inicializuje korytnačku v strede plochy otočenú na východ:

```
>>> t.reset()      # zmaže grafickú plochu a inicializuje korytnačku
```

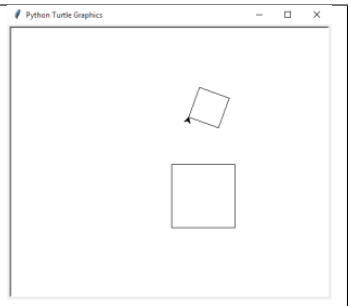
a teraz znovu kreslíme už do prázdnej plochy.

Korytnačka má pero, ktorým kreslí pri svojom pohybe po grafickej ploche. Toto pero môžeme zdvihnúť (`pen up`) - odteraz sa pohybuje bez kreslenia, alebo spustiť (`pen down`) - opäť bude pri pohybe kresliť. Na to máme dva príkazy `penup()` alebo `pendown()`, prípadne ich skratky `pu()` alebo `pd()`. Predchádzajúci príklad doplníme o dvíhanie pera:

```
import turtle

def stvorec(dlzka):
    for i in range(4):
        t.fd(dlzka)
        t.rt(90)

t = turtle.Turtle()
stvorec(100)
t.pu()
t.lt(70)
t.fd(80)
t.pd()
stvorec(50)
```



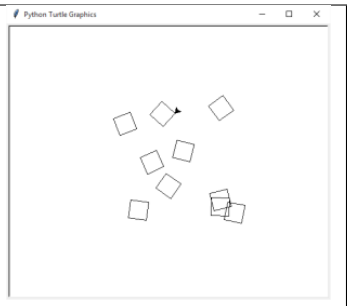
Napíšme funkciu `posun()`, ktorá presunie korytnačku na náhodnú pozíciu v ploche a dá jej aj náhodný smer:

```
import turtle
import random

def posun():
    t.pu()
    t.setpos(random.randint(-300, 300),
             random.randint(-300, 300))
    t.seth(random.randrange(360))
    t.pd()

def stvorec(dlzka):
    for i in range(4):
        t.fd(dlzka)
        t.rt(90)

t = turtle.Turtle()
for i in range(10):
    posun()
    stvorec(30)
```

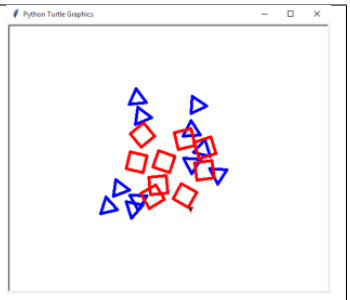


- funkcia na náhodné pozície nakreslí 10 malých štvorcov
- použili sme tu dva nové príkazy: `setpos(x, y)`, ktorá presunie korytnačku na novú pozíciu a `seth(uhol)` (skratka z `setheading()`), ktorá otočí korytnačku do daného smeru

Grafickému peru korytnačky môžeme meniť hrúbku a farbu:

- príkaz `pencolor(farba)` zmení farbu pera - odteraz bude korytnačka všetko kresliť touto farbou, až kým ju opäť nezmeníme
- príkaz `pensize(hrúbka)` zmení hrúbku pera (celé kladné číslo) - odteraz bude korytnačka všetko kresliť touto hrúbkou, až kým ju opäť nezmeníme

V nasledovnom príklade uvidíme aj príkaz `turtle.delay()`, ktorým môžeme urýchliť (alebo spomaliť) pohyb korytnačky (rýchlosť `turtle.delay(0)` je najrýchlejšia, `turtle.delay(10)` je pomalšia - parameter hovorí počet milisekúnd, ktorým sa zdržuje každé kreslenie):



```
import turtle
import random

def stvorec(dlzka):
    for i in range(4):
        t.fd(dlzka)
        t.rt(90)

def trojuholnik(dlzka):
    for i in range(3):
        t.fd(dlzka)
        t.rt(120)

def posun():
    t.pu()
    t.setpos(random.randrange(-300, 300),
             random.randrange(-300, 300))
    t.seth(random.randrange(360))
    t.pd()

turtle.delay(0)
t = turtle.Turtle()
t.pensize(5)
for i in range(20):
    posun()
    if random.randrange(2):
        t.pencolor('red')
        stvorec(30)
    else:
        t.pencolor('blue')
        trojuholnik(30)
```

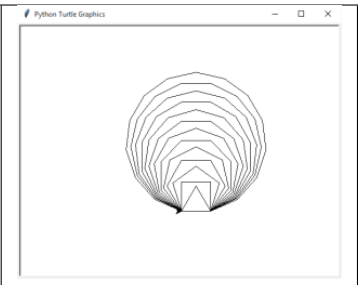
Program na náhodné pozície umiestni červené štvorce alebo modré trojuholníky. Zrejme korytnačka je v **globálnej premennej** `t` (v hlavnom mennom priestore) a teda na ňu vidia všetky naše funkcie.

Ďalší príklad predvedie funkciu, ktorá nakreslí ľubovoľný rovnostranný n-uholník a tiež príkaz `clear()`, ktorý zmaže nakreslený obrázok, aby sa mohol kresliť ďalší už v prázdnej grafickej ploche:

```
import turtle

def n_uholnik(n, d):
    for i in range(n):
        t.fd(d)
        t.lt(360 / n)

t = turtle.Turtle()
for n in range(3, 16):
    t.clear()
    n_uholnik(n, 100)
```



- ak by sme vyhodili príkaz `clear()`, mohli by sme v jednej kresbe vidieť všetky n-uholníky

Pomocou n-uholníkov môžeme nakresliť aj kružnicu (napr. ako 36-uholník s malou dĺžkou strany), ale aj len časti kružníc, napr. 18 strán z 36-uholníka nakreslí polkruh, a 9 strán nakreslí štvrťkruh.

Nasledovný príklad najprv definuje `oblúk` (štvrťkruh), potom `lupen` (dva priložené štvrťkruhy) a nakoniec kvet ako `n` lupeňov:

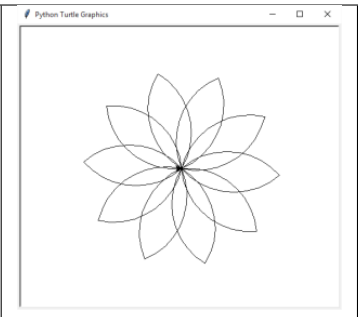
```
import turtle

def obluk(d):
    for i in range(9):
        t.fd(d)
        t.rt(10)

def lupen(d):
    for i in 1, 2:
        obluk(d)
        t.rt(90)

def kvet(n, d):
    for i in range(n):
        lupen(d)
        t.rt(360 / n)

turtle.delay(0)
t = turtle.Turtle()
kvet(10, 20)
```



11.1.1 Vyfarbenie útvaru

Útvary, ktoré nakreslí korytnačka sa dajú aj vyfarbiť. Predpokladajme, že korytnačka nakreslí nejaký útvar (napr. štvorec), a potom ho môže farbou výplne vyfarbiť. Princíp fungovania vypĺňania nejakou farbou je takýto:

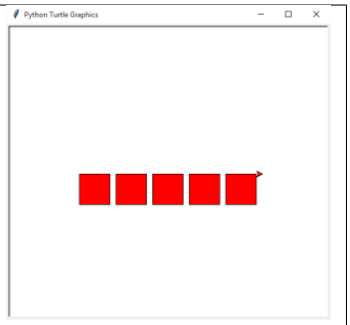
- na začiatok postupnosti korytnačích príkazov, ktoré definujú obrys útvaru, umiestnime príkaz `begin_fill()`
- na koniec tejto postupnosti dáme príkaz `end_fill()`, ktorý vyfarbí nakreslený útvar farbou výplne
- farbu výplne meníme príkazom `fillcolor(farba)` (na začiatku je nastavená čierna farba)
- ak nakreslíme krivku, ktorá netvorí uzavretý útvar, pri vypĺňaní sa táto uzavrie

Napr. nakreslíme farebné štvorce v jednom rade:

```
import turtle

def stvorec(d):
    for i in range(4):
        t.fd(d)
        t.rt(90)

turtle.delay(0)
t = turtle.Turtle()
t.fillcolor('red')
for i in range(5):
    t.begin_fill()
    stvorec(50)
    t.end_fill()
    t.pu()
    t.fd(60)
    t.pd()
```



Nakreslíme kvet zložený z farebných lupeňov (každý lupeň inou farbou):

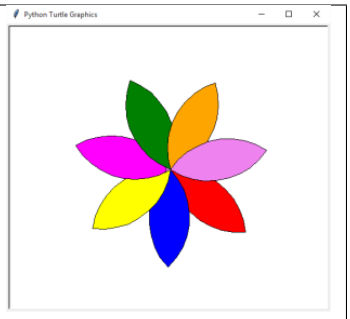
```
import turtle

def obluk(d):
    for i in range(9):
        t.fd(d)
        t.rt(10)

def lupen(d):
    for i in 1, 2:
        obluk(d)
        t.rt(90)

def kvet(d, farby):
    for f in farby:
        t.fillcolor(f)
        t.begin_fill()
        lupen(d)
        t.end_fill()
        t.rt(360/len(farby))

turtle.delay(0)
t = turtle.Turtle()
kvet(20, ['red', 'blue', 'yellow', 'magenta',
          'green', 'orange', 'violet'])
```

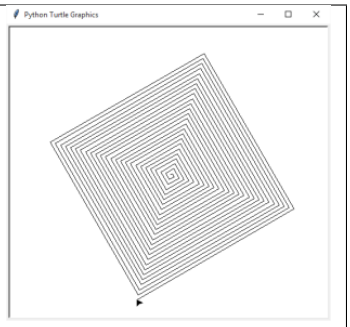


11.1.2 Špirály

Rôzne špirály môžeme kresliť tak, že opakujeme kreslenie stále sa zväčšujúcich čiar a za každým sa otočíme o pevný uhol, napr.

```
import turtle

t = turtle.Turtle()
t.lt(30)
for i in range(3, 300, 3):
    t.fd(i)
    t.rt(90)
```

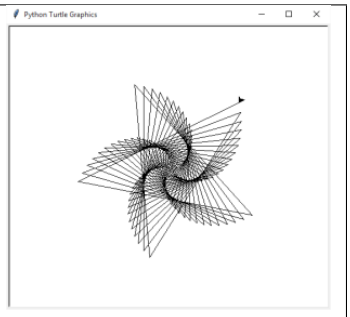


Program nakreslí štvorcovú špirálu.

S uhlami môžete experimentovať, napr.

```
import turtle
import random

turtle.delay(0)
t = turtle.Turtle()
while True:
    uhol = random.randrange(30, 170)
    print('spirala s uhlom', uhol)
    for i in range(3, 300, 3):
        t.fd(i)
        t.rt(uhol)
    t.reset()
```



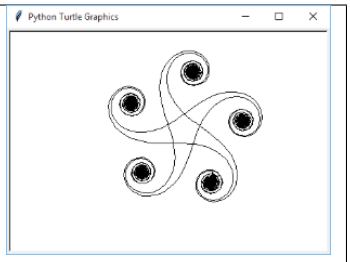
Tento program kreslí špirály s rôznymi náhodne generovanými uhlami. Zároveň do textovej plochy vypisuje informáciu o uhle momentálne kreslenej špirály.

Zaujímavé špirály vznikajú, keď nemeníme dĺžku čiar ale uhol, napr.

Tu môžeme vyskúšať rôzne malé zmeny uhla, o ktorý sa mení kreslenie čiar útvaru:

```
import turtle

turtle.delay(0)
t = turtle.Turtle()
for uhol in range(1, 2000):
    t.fd(8)
    t.rt(uhol + 0.1)
```



Vyskúšajte rôzne iné zmeny uhla v príkaze `t.rt()`.

11.1.3 Zhrnutie užitočných metód

metóda	variant	význam	príklad
`forward(d)`	`fd`	choď dopredu	`t.fd(100); t.fd(-50)`
`back(d)`	`backward, bk`	cúvaj	`t.bk(50); t.bk(-10)`
`right(u)`	`rt`	otoč sa vpravo	`t.rt(90); t.rt(-120)`
`left(u)`	`lt`	otoč sa vľavo	`t.lt(90); t.lt(-45)`
`penup()`	`pu, up`	zdvihni pero	`t.pu()`
`pendown()`	`pd, down`	spusti pero	`t.pd()`
`setpos(x, y)`	`setposition, goto`	choď na pozíciu	`t.setpos(50, 70)`
`pos()`	`position`	zisti pozíciu korytnačky	`t.pos()`
`xcor()`		zisti x-ovú súradnicu	`t.xcor()`
`ycor()`		zisti y-ovú súradnicu	`t.ycor()`
`heading()`		zisti uhol korytnačky	`t.heading()`
`setheading(s)`	`seth`	nastav uhol korytnačky	`t.seth(120)`
`pensize(h)`	`width`	nastav hrúbku pera	`t.pensize(5)`
`pencolor(f)`		nastav farbu pera	`t.pencolor('red')`
`pencolor()`		zisti farbu pera	`t.pencolor()`
`fillcolor(f)`		nastav farbu výplne	`t.fillcolor('blue')`
`fillcolor()`		zisti farbu výplne	`t.fillcolor()`
`color(f1, f2)`		nastav farbu pera aj výplne	`t.color('red'); t.color('blue', 'white')`
`color()`		zisti farbu pera aj výplne	`t.color()`
`reset()`		zmaž kresbu a inicializuj korytnačku	`t.reset()`
`clear()`		zmaž kresbu	`t.clear()`
`begin_fill()`		začiatok budúceho vyfarbenia	`t.begin_fill()`
`end_fill()`		koniec vyfarbenia	`t.end_fill()`

11.1.4 Globálne korytnačie funkcie

Modul `turtle` má ešte tieto funkcie, ktoré robia globálne nastavenia a zmeny (majú vplyv na všetky korytnačky):

- `turtle.delay(číslo)` - vykonávanie korytnačích metód sa spomalí na zadaný počet milisekúnd (štandardne je 10)
  - každú jednu korytnačku môžeme ešte individuálne zrýchľovať alebo spomaľovať pomocou `t.speed(číslo)`, kde `číslo` je od 0 do 10 (0 najrýchlejšie, štandardne je 3)
- `turtle.tracer(číslo)` - zapne alebo vypne priebežné zobrazovanie zmien v grafickej ploche (štandardne je `číslo` 1):
  - `turtle.tracer(0)` - vypne zobrazovanie zmien, t. j. teraz je vykresľovanie veľmi rýchle bez pozdržiavania, ale zatiaľ žiadnu zmenu v grafickej ploche nevidíme
  - `turtle.tracer(1)` - zapne zobrazovanie zmien, t. j. teraz je vykresľovanie už pomalé (podľa nastavených `turtle.delay()` a `t.speed()`), lebo vidíme všetky zmeny kreslenia v grafickej ploche
- `turtle.bgcolor(farba)` - zmení farbu pozadia grafickej plochy, pričom všetky kresby v ploche ostávajú bez zmeny

11.1.5 Tvar korytnačky

Korytnačkám môžeme meniť ich tvar - momentálne je to malý trojuholník.

Príkaz `shape()` zmení tvar na jeden s preddefinovaných tvarov (pre korytnačku `t`):

```
t.shape('arrow')          # tvarom korytnačky bude šípka
t.shape('turtle')         # tvarom korytnačky bude korytnačka
t.shape('circle')         # tvarom korytnačky bude kruh
t.shape('square')         # tvarom korytnačky bude štvorec
t.shape('triangle')       # tvarom korytnačky bude trojuholník
t.shape('classic')        # tvarom korytnačky bude hrot šípky
```

Default tvar je `'classic'`.

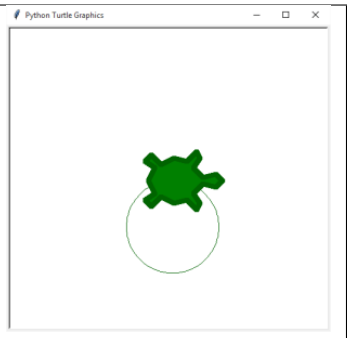
Príkaz `shapesize()` nastavuje zväčšenie tvaru a hrúbku obrysu tvaru (pre korytnačku `t`):

```
t.shapesize(sirka, vyska, hrubka)
```

Mohli ste si všimnúť, že keď korytnačke zmeníte farbu pera, zmení sa obrys jej tvaru. Podobne, keď sa zmení farba výplne, tak sa zmení aj výplň tvaru korytnačky. Napr.

```
import turtle

turtle.delay(0)
t = turtle.Turtle()
t.shape('turtle')
t.shapesize(5, 5, 8)
t.color('darkgreen', 'green')
for i in range(90):
    t.fd(5)
    t.rt(4)
```



V tomto príklade sa nastaví korytnačke zväčšený tvar a pomaly nakreslí kružnicu (90-uholník).

Zobrazovanie tvaru korytnačky môžeme skryť príkazom `hideturtle()` (skratka `ht()`) a opätovné zobrazovanie zapnúť príkazom `showturtle()` (skratka `st()`).

11.1.6 Náhodné prechádzky

Náhodnými prechádzkami budeme nazývať taký pohyb korytnačky, pri ktorom sa korytnačka veľa-krát náhodne otočí a prejde nejakú malú vzdialenosť. Napr.

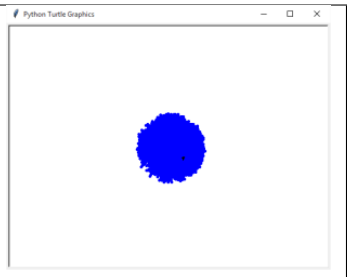
```
import turtle
import random

turtle.delay(0)
t = turtle.Turtle()
for i in range(10000):
    t.seth(random.randrange(360))
    t.fd(10)
```

Po čase odíde z grafickej plochy - upravíme tak, aby nevyšla z nejakej oblasti, napr.

```
import turtle
import random

turtle.delay(0)
t = turtle.Turtle()
t.pensize(5)
t.pencolor('blue')
for i in range(10000):
    t.seth(random.randrange(360))
    t.fd(10)
    if t.xcor() ** 2 + t.ycor() ** 2 > 50 ** 2:
        t.fd(-10)
```

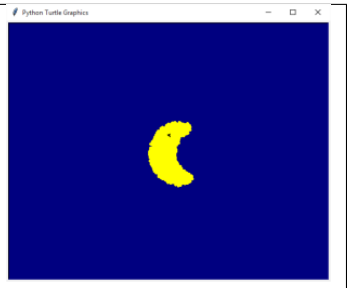


Príkaz `if` nedovolí korytnačke vzdialiť sa od (0,0) viac ako 50 - počítali sme tu vzdialenosť korytnačky od počiatku.

Môžeme využiť metódu `distance()`, ktorá vráti vzdialenosť korytnačky od nejakého bodu alebo inej korytnačky:

```
import turtle
import random

turtle.delay(0)
t = turtle.Turtle()
turtle.bgcolor('navy')
t.pensize(5)
t.pencolor('yellow')
for i in range(10000):
    t.seth(random.randrange(360))
    t.fd(10)
    if t.distance(20, 0) > 50 or t.distance(50, 0) < 50:
        t.fd(-10)
```



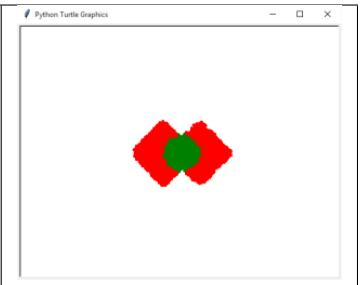
Korytnačka sa teraz pohybuje v oblasti, ktorá ma tvar mesiaca: nesmie vyjsť z prvého kruhu a zároveň vojsť do druhého.

Tvar stráženej oblasti môže byť definovaný aj zložitejšou funkciou, napr.

```
import turtle
import random

def fun(pos):
    x, y = pos              # pos je dvojica súradníc
    if abs(x - 30) + abs(y) < 50:
        return False
    return abs(x + 30) + abs(y) > 50

turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
t.pensize(5)
for i in range(10000):
    t.seth(random.randrange(360))
    if t.distance(0, 0) < 30:
        t.pencolor('green')
    else:
        t.pencolor('red')
    t.fd(5)
    if fun(t.pos()):        # funkcia fun stráži nejakú␣
↪oblast'
        t.fd(-5)
```



Okrem stráženia oblasti tu meníme farbu pera podľa nejakej podmienky

11.2 Viac korytnačiek

Doteraz sme pracovali len s jednou korytnačkou (vytvorili sme ju pomocou `t = turtle.Turtle()`). Korytnačiek ale môžeme vytvoriť ľubovoľne veľa. Aby rôzne korytnačky mohli využívať tú istú kresliacu funkciu (napr. `stvorec()`) musíme globálnu premennú `t` vo funkcii prerobiť na parameter:

```
import turtle

def stvorec(tu, velkost):
    for i in range(4):
        tu.fd(velkost)
        tu.rt(90)

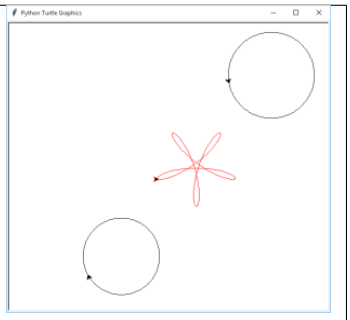
t = turtle.Turtle()
stvorec(t, 100)
```

Vytvorme ďalšiu korytnačku a necháme ju tiež kresliť štvorce tou istou funkciou:

```
t1 = turtle.Turtle()
t1.lt(30)
for i in range(5):
    stvorec(t1, 50)
    t1.lt(72)
```

Kým sme vytvárali funkcie, ktoré pracovali len pre jednu korytnačku, nemuseli sme ju posielať ako parameter. Ak ale budeme potrebovať funkcie, ktoré by mali pracovať pre ľubovoľné ďalšie korytnačky, vytvoríme vo funkcii nový parameter (najčastejšie ako prvý parameter funkcie) a ten bude v tele funkcie zastupovať tú korytnačku, ktorú do funkcie pošleme.

V ďalšom príklade vyrobíme 3 korytnačky: 2 sa pohybujú po nejakej stálej trase a tretia sa vždy nachádza presne v strede medzi nimi (ako keby bola v strede gumenej nite):



```
import turtle

def posun(t, pos):      # pos je pozícia v tvare (x,y)
    t.pu()
    t.setpos(pos)
    t.pd()

def stred(k1, k2):
    x = (k1.xcor() + k2.xcor()) / 2
    y = (k1.ycor() + k2.ycor()) / 2
    return (x, y)

turtle.delay(0)
t1 = turtle.Turtle()
posun(t1, (-100, -100))
t2 = turtle.Turtle()
posun(t2, (200, 100))
t3 = turtle.Turtle()
posun(t3, stred(t1, t2))
t3.pencolor('red')

while True:
    t1.fd(4)
    t1.rt(3)
    t2.fd(3)
    t2.lt(2)
    t3.setpos(stred(t1, t2))
```

11.2.1 Zoznam korytnačiek

Do premennej typu zoznam postupne priradíme vygenerované korytnačky, pričom každú presunieme na inú pozíciu (všetky ležia na x-ovej osi) a nastavíme jej iný smer:

```
import turtle

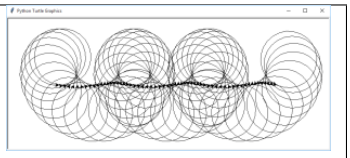
turtle.delay(0)
zoznam = []
for i in range(60):
    t = turtle.Turtle()
    t.pu()
    t.setpos(-300 + 10 * i, 0)
    t.pd()
    t.seth(i * 18)
    zoznam.append(t)
```

Necháme ich kresliť rovnaké kružnice:

```
import turtle

turtle.delay(0)
zoznam = []
for i in range(60):
    t = turtle.Turtle()
    t.pu()
    t.setpos(-300 + 10 * i, 0)
    t.pd()
    t.seth(i * 18)
    zoznam.append(t)

for t in zoznam:
    for i in range(24):
        t.fd(20)
        t.lt(15)
```



Takto kreslila jedna za druhou: ďalšia začala kresliť až vtedy, keď predchádzajúca skončila.

Pozmeňme to tak, aby všetky kreslili naraz:

```
import turtle

turtle.delay(0)
zoznam = []
for i in range(60):
    t = turtle.Turtle()
    t.pu()
    t.setpos(-300 + 10 * i, 0)
    t.pd()
    t.seth(i * 18)
    zoznam.append(t)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
for i in range(24):
    for t in zoznam:
        t.fd(20)
        t.lt(15)
```

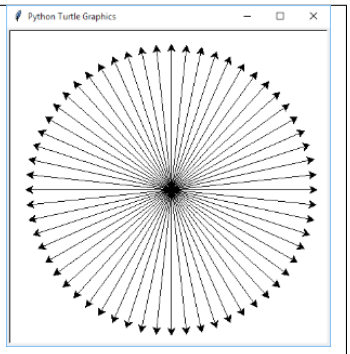
Tu sme zmenili len poradie for-cyklov.

V ďalšom príklade vygenerujeme všetky korytnačky v počiatku súradnej sústavy ale s rôznymi smermi a necháme ich prejsť dopredu rovnakú vzdialenosť:

```
import turtle

turtle.delay(0)
zoznam = []
for i in range(60):
    zoznam.append(turtle.Turtle())
    zoznam[-1].seth(i * 6)

for t in zoznam:
    t.fd(200)
```

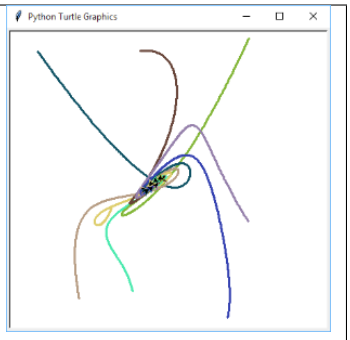


Všimnite si, ako pracujeme so zoznamom korytnačiek (`zoznam[-1]` označuje posledný prvok zoznamu, t. j. naposledy vygenerovanú korytnačku).

11.2.2 Korytnačky sa naháňajú

Na náhodných pozíciách vygenerujeme `n` korytnačiek a potom ich necháme, nech sa naháňajú podľa takýchto pravidiel:

- každá sa otočí smerom k nasledovnej (prvá k druhej, druhá k tretej, . . . , N-tá k prvej)
- každá prejde stotinu vzdialenosti k nasledovnej



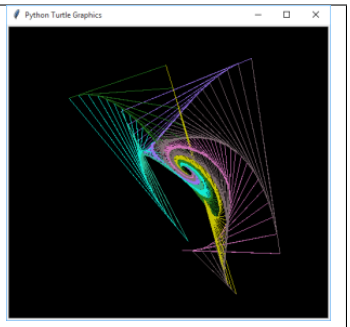

```
import turtle
import random

n = 8
t = []
turtle.delay(0)
for i in range(n):
    nova = turtle.Turtle()
    nova.pu()
    nova.setpos(random.randrange(-300, 300),
                random.randrange(-300, 300))
    nova.pencolor(f'#{random.randrange(256**3):06x}')
    nova.pensize(3)
    nova.pd()
    t.append(nova)

while True:
    for i in range(n):
        j = (i + 1) % n                # index nasledovnej
        uhol = t[i].towards(t[j])
        t[i].seth(uhol)
        vzdialenost = t[i].distance(t[j])
        t[i].fd(vzdialenost / 100)
```

Využili sme novú metódu `towards()`, ktorá vráti uhol otočenia k nejakému bodu alebo k pozícii inej korytnačky.

Trochu pozmeníme: okrem prejdenia 1/10 vzdialenosti k nasledovnej nakreslí aj celú spojnicu k nasledovnej:




```
import turtle
import random

turtle.delay(0)
while True:
    turtle.bgcolor('black')
    n = random.randrange(3, 9)
    t = []
    for i in range(n):
        nova = turtle.Turtle()
        nova.speed(0)
        nova.pu()
        nova.setpos(random.randrange(-300, 300),
                    random.randrange(-300, 300))
        nova.pencolor(f'#{random.randrange(256**3):06x}')
        nova.pd()
        nova.ht()
        t.append(nova)

    for k in range(100):
        for i in range(n):
            j = (i + 1) % n                 # index␣
↪nasledovnej
            uhol = t[i].towards(t[j])
            t[i].seth(uhol)
            vzdialenost = t[i].distance(t[j])
            t[i].fd(vzdialenost)
            t[i].fd(vzdialenost / 10 - vzdialenost)

    for tt in t:
        tt.clear()
```

Po dokreslení, obrázok zmaže a začne kresliť nový. Uvedomte si, že každý ďalší prechod while-cyklu vytvorí nové a nové korytnačky a tie pôvodné staré tam ostávajú, hoci majú skrytý tvar a teda ich nevidíme. Ak by sme naozaj chceli zrušiť korytnačky z grafickej plochy, tak namiesto záverečného cyklu:

```
for tt in t:
    tt.clear()
```

by sme mali zapísať:

```
turtle.getscreen().clear()
```

11.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

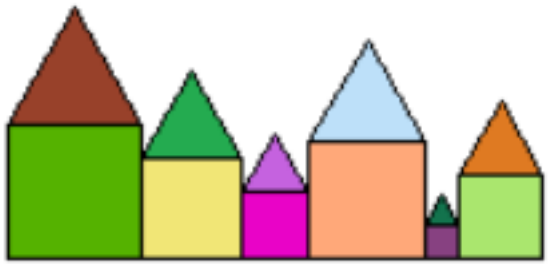
1. Napíšte funkcie `stvorec(strana, farba)` a `trojuholnik(strana, farba)`, ktoré nakreslia vyplnené útvary danou farbou. Potom pomocou nich zadefinujte funkciu `domcek(strana, farba1, farba2)`, ktorá nakreslí domček, pričom `farba1` je farba štvorca a `farba2` farba strechy (trojuholníka).
   - otestujte napr.

   ```
   >>> domcek(100, 'red', 'blue')
   ```

2. Napíšte funkciu `ulica(zoznam)`, ktorá nakreslí vedľa seba niekoľko domčekov (volaním funkcie `domcek()`), pričom farby trojuholníka aj štvorca sú vygenerované náhodne, veľkosti domčekov aj ich počet je daný vstupným zoznamom: napr. ak má zoznam hodnoty `[50, 20, 80]`, funkcia nakreslí tesne vedľa seba (na spoločnej základni) 3 domčeky postupne veľkostí 50, 20 a 80. Rad domčekov začnite kresliť na ľavom okraji grafickej plochy (napr. `(-300, 0)`).
   - otestujte napr.

   ```
   >>> ulica([40, 30, 20, 35, 10, 25])
   ```

3. Napíšte funkciu `stvorce(strana, farba1, farba2)`, ktorá nakreslí štvorec s danou stranou a tiež ďalší do neho vpísaný štvorec, tento menší štvorec má vrcholy v stredoch strán veľkého štvorca. Parameter `farba1` bude farbou výplne veľkého štvorca a `farba2` farbou výplne malého štvorca.
   - otestujte napr.

```
>>> stvorce(150, 'blue', 'yellow')
```

4. Napíšte funkciu `rad_stvorcov(n, strana)`, ktorá nakreslí vedľa seba do radu `n` štvorcov (úplne na tesno), pričom po dokreslení celého radu korytnačka skončí presne na tom mieste, kde začala prvý štvorec. Každý zo štvorcov nech je vyfarbený náhodnou farbou.
   - otestujte napr.

```
>>> rad_stvorcov(20, 15)
```

5. Napíšte funkciu `pyramida(n, strana)`, ktorá pomocou funkcie `rad_stvorcov()` z predchádzajúcej úlohy nakreslí `n`-poschodovú pyramídu: v spodnm rade je `n` štvorcov, každý rad nad ním má o 1 štvorec menej. Pyramídu začnite kresliť tak, aby sa zmestila do grafickej plochy.
   - otestujte napr.

```
>>> pyramida(10, 50)
```

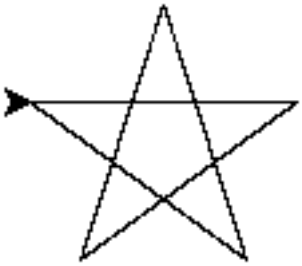
6. Napíšte funkciu `ciara()`, ktorá nakreslí úsečku dĺžky 100, pričom ju kreslí ako 1000 čiar dĺžky 0.1.
   - otestujte napr.

```
>>> ciara()
```

7. Funkcia `hviezda()` pomocou funkcie `ciara()` nakreslí 5-cípu hviezdu (5 úsečiek s otáčaním o 144 stupňov).

   - otestujte napr.

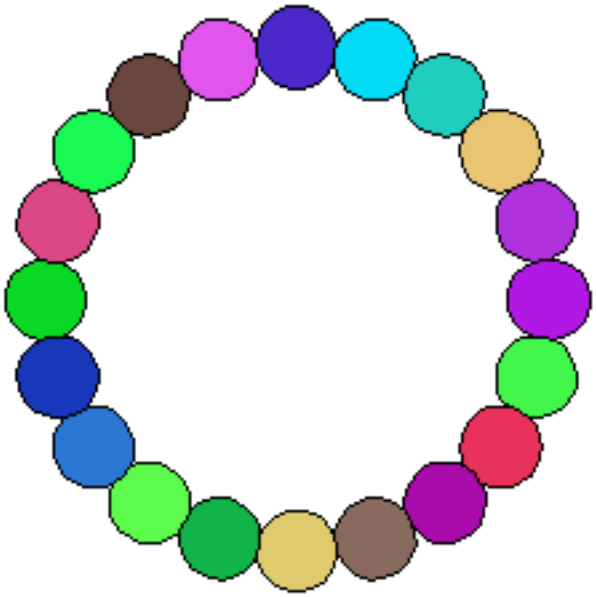
```
>>> hviezda()
```



8. Funkcia `kruh(velkost, farba)` nakreslí 24-uholník so stranou `velkost`, ktorý bude vyfarbený danou farbou. Potom funkcia `naramok(n, vel)` nakreslí `n` takýchto kruhov rozmiestnených na obvode nejakej kružnice. Kruhy vyfarbite náhodnou farbou.

   - otestujte napr.

```
>>> naramok(20, 4)
```



9. Funkcia `prechadzka1(a, b)` pomocou náhodnej prechádzky vyfarbí obdĺžnik veľkosti `a` x `b` so stredom `(0, 0)`.

   - otestujte napr.

```
>>> prechadzka1(100, 20)
```

10. Funkcia `prechadzka2(a, b)` pomocou náhodnej prechádzky vyfarbí dva na seba kolmé obdĺžniky `a` x `b` a `b` x `a` so spoločným stredom `(0, 0)`.

    - otestujte napr.

```
>>> prechadzka2(100, 20)
```

11. Funkcia `prechadzka3(r1, r2)` pomocou náhodnej prechádzky vyfarbí dva kruhy: jeden s polomerom `r1` a stredom `(0, 0)` a druhý s polomerom `r2` a stredom `(0, r1+r2-5)`.

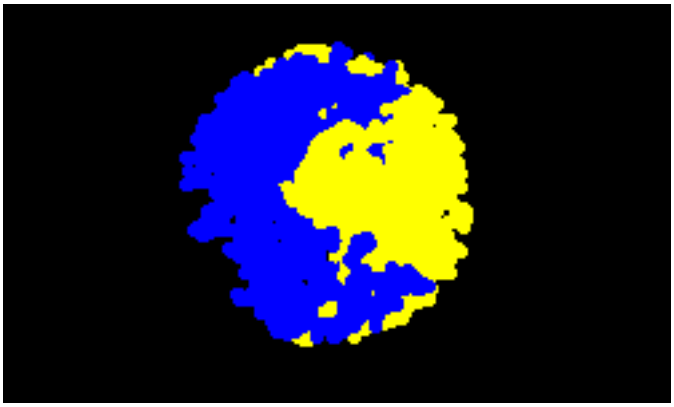
    - otestujte napr.

```
>>> prechadzka3(50, 30)
```

12. Vytvorte 2 korytnačky s hrúbkou pera 5 a farbami pera `blue` a `yellow`. Pomocou funkcie `prechadzka4(r)` sa budú obe paralelne prechádzať v kruhu s polomerom `r` (každa so svojimi náhodnými uhlami).

    - otestujte napr.

```
>>> prechadzka4(80)
```



13. Napíšte funkciu `nova(x, y, uhol=0)`, ktorá vytvorí novú korytnačku, umiestni ju na pozíciu `(x, y)` a nastaví jej daný uhol. Funkcia vráti túto korytnačku ako výsledok funkcie

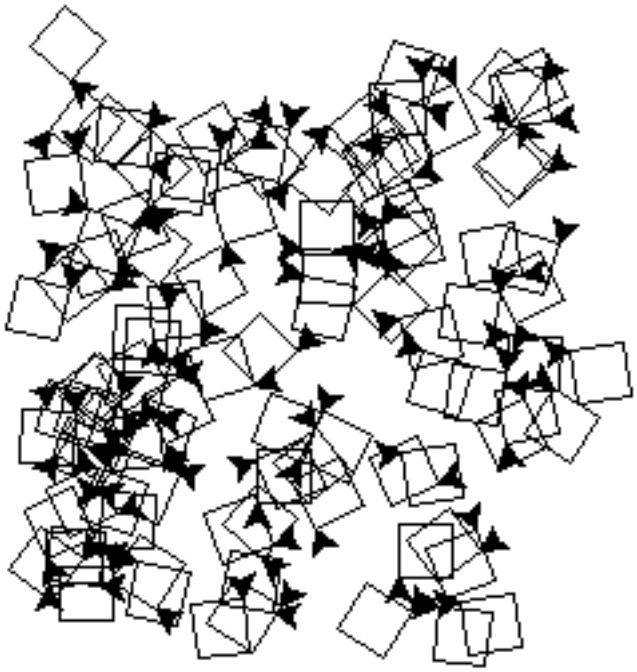
    - otestujte napr.

```
>>> t1 = nova(-100, 50, -90)
>>> t1.fd(100)
```

14. Otestujte funkciu `nova()` z predchádzajúcej úlohy tak, že pomocou nej vygenerujete 100 korytnačiek na náhodných pozíciách s náhodným uhlom a potom všetky naraz nakreslia štvorec so stranou 20, t. j. najprv všetky prejdú dopredu 20, potom sa všetky otočia o 90 stupňov a toto 4-krát opakujú.

    - otestujte napr.

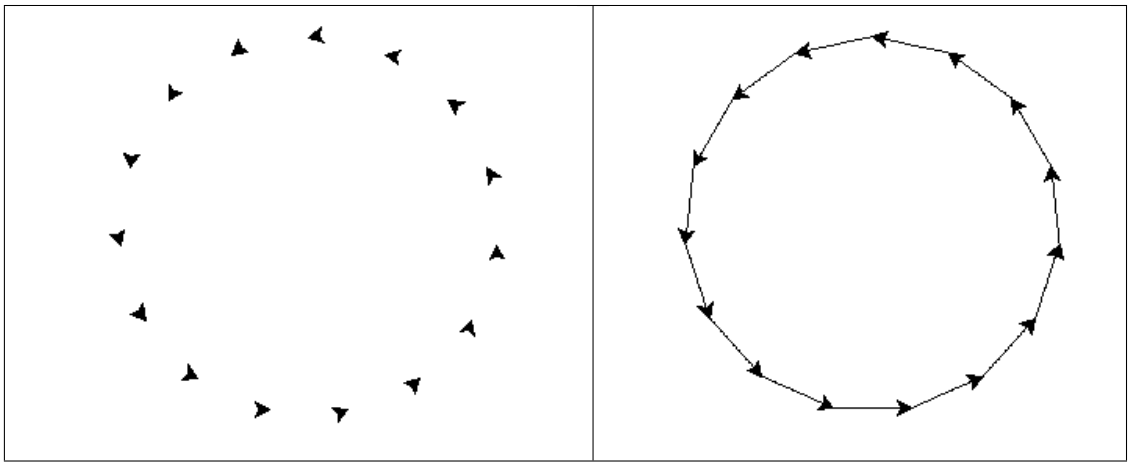
```
>>> test_nova()
```



15. Napíšte funkciu `n_uholnik(n, strana)`, pomocou ktorej korytnačka `t` prejde so zdvihnutým perom po obvode pravidelného `n`-uholníka s danou stranou a v každom vrchole vytvorí novú korytnačku, ktorá je natočená v smere príslušnej strany n-uholníka. Funkcia vráti **zoznam** takto vytvorených korytnačiek. Na záver sa všetky tieto korytnačky naraz pohybujú svojim smerom krokom 1 k nasledovnému vrcholu

- otestujte napr.

```
>>> zoznam = n_uholnik(15, 50)
>>> for i in range(50):
        ... každá korytnačka v zozname prejde dopredu 1 krok
```



11.4 4. Domáce zadanie

L.I.S.T.

- riešenie odovzdávajte na úlohový server https://list.fmph.uniba.sk/

Programovací jazyk **Logo** podobne ako my v Pythone riadi korytnačku v grafickej ploche. Syntax jazyka sa ale trochu líši od Pythonu. Nás z Loga budú zaujímať len týchto 8 príkazov:

- `fd 100` - zozpovedá pythonovskému `t.fd(100)`, parametrom je celé alebo desatinné číslo
- `rt 45` - zozpovedá pythonovskému `t.rt(45)`, parametrom je celé alebo desatinné číslo
- `lt 60` - zozpovedá pythonovskému `t.lt(60)`, parametrom je celé alebo desatinné číslo
- `pu` - zozpovedá pythonovskému `t.pu()`, príkaz je bez parametrov
- `pd` - zozpovedá pythonovskému `t.pd()`, príkaz je bez parametrov
- `setpc 'red'` - zozpovedá pythonovskému `t.pencolor('red')`, parametrom je znakový reťazec
- `setpw 5` - zozpovedá pythonovskému `t.pensize(5)`, parametrom je celé číslo
- `repeat 4 [fd 50 rt 90]` - označuje cyklus (bez premennej cyklu) s daným počtom opakovaní, pričom telom môže byť ľubovoľný logovský program, napr. tento cyklus sa môže preložiť takto:

```
for _ in range(4):
    t.fd(50)
    t.rt(90)
```

  premennú cyklu budeme zapisovať identifikátorm _

Program v Logu môže byť naformátovaný úplne voľne, t.j. medzi príkazmi sú buď medzery alebo nové riadky a tak isto aj parametre sú od príkazov oddelené aspoň jednou medzerou alebo aj prázdnym riadkom.

Vašou úlohou bude napísať pythonovský skript s funkciou `logo()`, ktorý na vstupe dostane logovský program (textový súbor s príponou `'.txt'`) a vyrobí z neho pythonovský skript (textový súbor s rovnakým menom ale s príponou `'.py'`), ktorým by sa nakreslil identický obrázok s logovským programom.

Napr. pre takýto vstupný súbor `'subor1.txt'`:

```
pu lt 30 fd 100 lt 60 setpc
'red' pd
fd 100 rt 120 fd
100    rt 120
     fd 100 rt
   60 setpc 'blue' fd 100 rt 120
fd 100
```

Váš progran vygeneruje takýto pythonovský skript `'subor1.py'`:

```
import turtle
t = turtle.Turtle()
t.pu()
t.lt(30)
t.fd(100)
t.lt(60)
t.pencolor('red')
t.pd()
t.fd(100)
t.rt(120)
t.fd(100)
t.rt(120)
t.fd(100)
t.rt(60)
t.pencolor('blue')
t.fd(100)
t.rt(120)
t.fd(100)
```

Alebo pre súbor `'subor2.txt'`:

```
lt 90 pu fd 100 rt 30 pd

repeat 4 [
  fd 50
  lt 30
  repeat 3[fd 30 rt 120]lt 60
] rt 30 fd 70
```

dostaneme:

```
import turtle
t = turtle.Turtle()
t.lt(90)
t.pu()
t.fd(100)
t.rt(30)
t.pd()
for _ in range(4):
    t.fd(50)
    t.lt(30)
    for _ in range(3):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        t.fd(30)
        t.rt(120)
    t.lt(60)
t.rt(30)
t.fd(70)
```

Váš odovzdaný program s menom `riesenie4.py` musí začínať tromi riadkami komentárov:

```
# 4. zadanie: logo
# autor: Janko Hraško
# datum: 28.12.2017
```

Modul musí obsahovať definíciu funkcie:

```
def logo(meno_suboru):
    ...
```

Túto funkciu bude volať testovač s rôznymi logovskými súbormi.

Projekt `riesenie4.py` odovzdávajte (bez ďalších dátových súborov) na úlohový server https://list.fmph.uniba.sk/ najneskôr do 23:00 **5. januára**, kde ho môžete nechať otestovať. Testovač bude spúšťať vašu funkciu s rôznymi vstupmi. Odovzdať projekt aj ho testovať môžete ľubovoľný počet krát. Môžete zaň získať **10 bodov**.

KAPITOLA 12

12. Rekurzia

Rekurzia je mechanizmus, vďaka ktorému môže funkcia zavolať samu seba. Najčastejšie sa bude používať na riešenie úloh, ktoré vieme rozložiť na menšie časti a niektoré z týchto (už menších) častí riešime opäť zavolaním samotnej funkcie. Keďže toto druhé (tzv. rekurzívne) volanie už rieši menšiu úlohu, ako bola pôvodná, je šanca, že sa takéto riešenie priblížilo k očakávanému výsledku.

Aby sme mali šancu lepšie porozumieť mechanizmu rekurzie, pripomeňme si mechanizmus volania funkcií:

- zapamätá sa návratová adresa
- vytvorí sa nový menný priestor funkcie
  - v ňom sa vytvárajú lokálne premenné aj parametre
- po skončení vykonania tela funkcie sa zruší menný priestor (aj so všetkými premennými)
- vykonávanie programu sa vráti na návratovú adresu

Nekonečná rekurzia

Rekurzia v programovaní teda znamená, že funkcia najčastejšie zavolá samú seba, t.j. že funkcia je definovaná pomocou samej seba. Na prvej ukážke vidíme rekurzívnu funkciu, ktorá nerobí nič iné, len volá samú seba:

```
def xy():
    xy()

xy()
```

Takéto volanie po krátkom čase skončí chybovou správou:

```
...
  [Previous line repeated 990 more times]
RecursionError: maximum recursion depth exceeded
```

To, že funkcia naozaj volala samú seba, môžeme vidieť, keď popri rekurzívnom volaní urobíme nejakú akciu, napr. budeme zvyšovať nejakú globálnu premennú:

```
def xy():
    global pocet
    pocet += 1
    xy()
```

```
>>> pocet = 0
>>> xy()
...
RecursionError: maximum recursion depth exceeded
>>> pocet
994
```

V tomto prípade program opäť spadne, hoci môžeme vidieť, že sa naozaj zvyšovalo počítadlo. Treba si uvedomiť, že každé zvýšenie počítadla znamená rekurzívne volanie a teda vidíme, že ich bolo skoro tisíc. My už vieme, že každé volanie funkcie (bez ohľadu na to, či je rekurzívna alebo nie) spôsobí, že Python si niekde zapamätá nielen **návratovú adresu**, aby po skončení funkcie vedel, kam sa má vrátiť, ale aj **menný priestor** tejto funkcie. Python má na tieto účely rezervu okolo 1000 vnorených volaní. Ak toto presiahneme, tak sa dozvieme správu **‚RecursionError: maximum recursion depth exceeded'** (hĺbka rekurzívnych volaní presiahla nastavené maximum).

```
import turtle

def xy(d):
    t.fd(d)
    t.lt(60)
    xy(d + 0.3)

turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
xy(1)
```

```
...
RecursionError: maximum recursion depth exceeded
```

Aj tento program, veľmi rýchlo spadne.

Odkrokujme si to:

- v príkazovom riadku je volanie funkcie `xy()` => vytvorí sa parameter `d` (čo je vlastne lokálna premenná), ktorej hodnota je 1,
- nakreslí sa čiara dĺžky `d` a korytnačka sa otočí doľava o 60 stupňov,
- znovu sa volá funkcia `xy()` s parametrom `d` zmeneným na `d+0.3`, t.j. vytvorí sa nová lokálna premenná `d` s hodnotou 1.3 (táto premenná sa vytvára v novom mennom priestore funkcie),
- toto sa robí donekonečna - našťastie to časom spadne na preplnení pythonovskej pamäte pre volania funkcií
- informácie o mennom priestore a aj návratovej adrese sa ukladajú v špeciálnej údajovej štruktúre **zásobník**

Zásobník (stack)

je údajová štruktúra, ktorá má tieto vlastnosti:

- nové prvky pridávame na vrch (napr. na vrch kopy tanierov, resp. na koniec radu čakajúcich)
- keď potrebujeme zo zásobníka nejaký prvok, vždy ho odoberáme z vrchu (odoberáme naposledy položený tanier, resp. posledný v rade čakajúcich)

Odborne sa tomu hovorí **LIFO** (last in first out), teda posledný prišiel, ale ako prvý odišiel (zrejme, keby to bol rad napr. v obchode, tak by sme ho považovali za **nespravodlivý rad**).

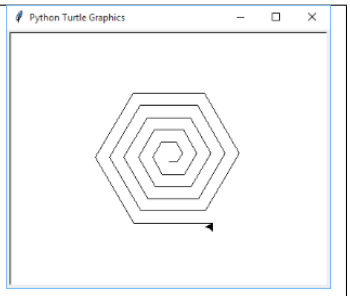
Zásobník sa používa na uchovávanie menných priestorov: každé ďalšie volanie funkcie, vytvorí nový menný priestor, ktorý sa uloží na koniec doteraz vytvorených menných priestorov. Keď ale príde ukončenie volania funkcie, z tohto zásobníka sa odstráni naposledy pridávaný menný priestor (hoci sme ho zaradili ako posledný, vybrali sme ho ako prvý, t.j. LIFO).

Chvostová rekurzia (nepravá rekurzia)

Aby sme nevytvárali nikdy nekončiace programy, t.j. nekonečnú rekurziu, niekde do tela rekurzívnej funkcie musíme vložiť test, ktorý zabezpečí, že v niektorých prípadoch rekurzia predsa len skončí. Najčastejšie to budeme riešiť tzv. **triviálnym prípadom**: na začiatok podprogramu umiestnime podmienený príkaz `if`, ktorý otestuje triviálny prípad, t.j. prípad, keď už nebudeme funkciu rekurzívne volať, ale vykonáme len nejaké „nerekurzívne“ príkazy (akcia triviálneho prípadu). Môžeme si to predstaviť aj takto: rekurzívna funkcia rieši nejaký komplexný problém a pri jeho riešení volá samu seba (rekurzívne volanie) väčšinou s nejakými pozmenenými údajmi. V niektorých prípadoch ale rekurzívne volanie na riešenie problému nepotrebujeme, ale vieme to vyriešiť „triviálne“ aj bez nej (riešenie takejto úlohy je už „triviálne“). V takto riešených úlohách vidíme, že funkcia sa skladá z dvoch častí:

- pri splnení nejakej podmienky, sa vykonajú príkazy bez rekurzívneho volania (triviálny prípad),
- inak sa vykonajú príkazy, ktoré v sebe obsahujú rekurzívne volanie.

Zrejme, toto má šancu fungovať len vtedy, keď po nejakom čase naozaj nastane podmienka triviálneho prípadu, t.j. keď sa tak menia parametre rekurzívneho volania, že sa k triviálnemu prípadu nejako blížime. V nasledujúcej ukážke môžete vidieť, že rekurzívna špirála sa kreslí tak, že sa najprv nakreslí úsečka dĺžky `d`, korytnačka sa otočí o 60 stupňov vľavo a dokreslí sa špirála väčšej veľkosti. Toto celé skončí, keď už budeme chcieť nakresliť špirálu väčšiu ako 100 - takáto špirála sa už nenakreslí. Akciou triviálneho prípadu je tu *nič*, t.j. žiadna akcia pre príliš veľké špirály:




```
import turtle

def spir(d):
    if d > 100:
        pass      # nerob nič
    else:
        t.fd(d)
        t.lt(60)
        spir(d + 3)

turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
spir(10)
```

Trasujme volanie so simuláciou na zásobníku s počiatočnou hodnotou parametra `d`, napr. 92, t.j. volanie `spir(92)`:

- na zásobníku vznikne nová lokálna premenná `d` s hodnotou 92 ... korytnačka nakreslí čiaru, otočí sa a volá `spir()` s parametrom 95,
- na zásobníku vznikne nová lokálna premenná `d` s hodnotou 95 ... korytnačka nakreslí čiaru, otočí sa a volá `spir()` s parametrom 98,
- na zásobníku vznikne nová lokálna premenná `d` s hodnotou 98 ... korytnačka nakreslí čiaru, otočí sa a volá `spir()` s parametrom 101,
- na zásobníku vznikne nová lokálna premenná `d` s hodnotou 101 ... korytnačka už nič nekreslí ani sa nič nevolá ... funkcia `spir()` končí, t.j.
  - zabudnú sa všetky lokálne premenné na tejto úrovni, t.j. premenná `d` s hodnotou 101 a riadenie sa vráti za posledné volanie funkcie `spir()` - tá ale končí, t.j.
  - zabudnú sa všetky lokálne premenné na tejto úrovni, t.j. premenná `d` s hodnotou 98 a riadenie sa vráti za posledné volanie funkcie `spir()` - tá ale končí, t.j.
- zabudnú sa všetky lokálne premenné na tejto úrovni, t.j. premenná `d` s hodnotou 95 a riadenie sa vráti za posledné volanie funkcie `Spir()` - tá ale končí, t.j.
- zabudnú sa všetky lokálne premenné na tejto úrovni, t.j. premenná `S` s hodnotou 92 a riadenie sa vráti za posledné volanie funkcie `Spir()` - teda do príkazového riadka

Toto nám potvrdia aj kontrolné výpisy vo funkcii:

```
def spir(d):
    print('volanie spir({})'.format(d))
    if d > 100:
        pass     # nerob nič
        print('... trivialny pripad - nerobim nic')
    else:
        t.fd(d)
        t.lt(60)
        print('... rekurzivne volam spir({})'.format(d+3))
        spir(d+3)
        print('... navrat z volania spir({})'.format(d+3))
```

```
>>> spir(92)
volanie spir(92)
... rekurzivne volam spir(95)
volanie spir(95)
... rekurzivne volam spir(98)
volanie spir(98)
... rekurzivne volam spir(101)
volanie spir(101)
... trivialny pripad - nerobim nic
... navrat z volania spir(101)
... navrat z volania spir(98)
... navrat z volania spir(95)
```

Nakoľko rekurzívne volanie funkcie je iba na jednom mieste, za ktorým už nenasledujú ďalšie príkazy funkcie, toto rekurzívne volanie sa dá ľahko prepísať cyklom `while`. Rekurzii, v ktorej za rekurzívnym volaním nie sú ďalšie príkazy, hovoríme **chvostová rekurzia** (jediné rekurzívne volanie je posledným príkazom funkcie). Predchádzajúcu ukážku môžeme prepísať napr. takto:

```
def spir(d):
    while d <= 100:
        t.fd(d);
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        t.lt(60);
        d = d + 3;
```

Rekurziu môžeme používať nielen pri kreslení pomocou korytnačky, ale napr. aj pri výpise pomocou `print()`. V nasledujúcom príklade vypisujeme vedľa seba čísla n, n-1, n-2, . . . , 2, 1:

```
def vypis(n):
    if n < 1:
        pass                # nič nerob len skonči
    else:
      print(n, end=', ')
      vypis(n - 1)          # rekurzívne volanie
```

```
>>> vypis(20)
20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
```

Zrejme je veľmi jednoduché prepísať to bez použitia rekurzie, napr. pomocou while-cyklu. Poexperimentujme, a vymeňme dva riadky: vypisovanie `print()` s rekurzívnym volaním `vypis()`. Po spustení vidíte, že aj táto nová rekurzívna funkcia sa dá prepísať len pomocou while-cyklu (resp. for-cyklu), ale jej činnosť už nemusí byť pre každého na prvý pohľad až tak jasná - odtrasujte túto zmenenú verziu:

```
def vypis(n):
    if n < 1:
        pass      # nič nerob len skonči
    else:
        vypis(n-1)
        print(n, end=', ')
```

```
>>> vypis(20)
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
```

12.1 Pravá rekurzia

Rekurzie, ktoré už nie sú obyčajné chvostové, sú na pochopenie trochu zložitejšie. Pozrime takéto kreslenie špirály:

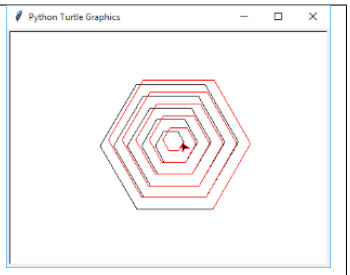
```
def spir(d):
    if d > 100:
        t.pencolor('red')      # a skonči
    else:
        t.fd(d)
        t.lt(60);
        spir(d + 3)
        t.fd(d)
        t.lt(60)

spir(1)
```

Nejaké príkazy sú pred aj za rekurzívnym volaním. Aby sme to lepšie rozlíšili, triviálny prípad nastaví inú farbu pera.

Aj takéto rekurzívne volanie sa dá prepísať pomocou dvoch cyklov:

```
def spir(d):
    pocet = 0
    while d <= 100:     # čo sa deje pred rekurzívnym
↪volaním
        t.fd(d)
        t.lt(60)
        d += 3
        pocet += 1
    t.pencolor('red')   # triviálny prípad
    while pocet > 0:    # čo sa deje po vynáraní z rekurzie
        d -= 3
        t.fd(d)
        t.lt(60)
        pocet -= 1
```



Aj v ďalších príkladoch môžete vidieť pravú rekurziu. Napr. vylepšená funkcia `vypis` vypisuje postupnosť čísel:

```
def vypis(n):
    if n < 1:
        pass          # skonči
    else:
        print(n, end=', ')
        vypis(n - 1)
        print(n, end=', ')

vypis(10)
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

Keď ako triviálny prípad pridáme výpis hviezdičiek, toto sa vypíše niekde medzi postupnosť čísel. Viete, kde sa vypíšu tieto hviezdičky?

```
def vypis(n):
    if n < 1:
        print('***', end=', ')          # a skonči
    else:
        print(n, end=', ')
        vypis(n - 1)
        print(n, end=', ')

vypis(10)
```

V ďalších príkladoch s korytnačkou využívame veľmi užitočnú funkciu `poly`:

```
def poly(pocet, dlzka, uhol):
    while pocet > 0:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        t.fd(dlzka)
        t.lt(uhol)
        pocet -= 1
```

Ktorú môžeme cvične prerobiť na rekurzívnu:

```
def poly(pocet, dlzka, uhol):
    if pocet > 0:
        t.fd(dlzka)
        t.lt(uhol)
        poly(pocet - 1, dlzka, uhol)
```

Zistite, čo kreslia funkcie `stvorec` a `stvorec1`:

```
def stvorec(a):
    if a > 100:
        pass           # nič nerob len skonči
    else:
        poly(4, a, 90)
        stvorec(a + 5)

def stvorec1(a):
    if a > 100:
        t.lt(180)        # a skonči
    else:
        poly(4, a, 90)
        stvorec1(a + 5)
        poly(4, a, 90)
```

Všetky tieto príklady s pravou rekurziou by ste mali vedieť jednoducho prepísať bez rekurzie pomocou niekoľkých cyklov.

V nasledujúcom príklade počítame **faktoriál** prirodzeného čísla `n`, pričom vieme, že

- 0! = 1 ... triviálny prípad
- n! = (n-1)!*n ... rekurzívne volanie

```
def faktorial(n):
    if n == 0:
        return 1
    return faktorial(n - 1) * n
```

Triviálnym prípadom je tu úloha, ako vyriešiť `0!`. Toto vieme aj bez rekurzie, lebo je to 1. Ostatné prípady sú už rekurzívne: na to, aby sme vyriešili zložitejší problém (`n` faktoriál), najprv vypočítame jednoduchší (‘n-1“ faktoriál) - zrejme pomocou rekurzie - a z neho skombinujeme (násobením) požadovaný výsledok. Hoci toto riešenie nie je chvostová rekurzia (po rekurzívnom volaní `faktorial` sa musí ešte násobiť), vieme ho jednoducho prepísať pomocou cyklu.

Pozrime ďalšiu jednoduchú rekurzívnu funkciu, ktorá otočí znakový reťazec (zrejme to vieme urobiť aj jednoduchšie pomocou `retazec[::-1]`):

```
def otoc(retazec):
    if len(retazec) <= 1:
        return retazec
    return otoc(retazec[1:]) + retazec[0]
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
print(otoc('Bratislava'))
print(otoc('Bratislava'*100))
```

Táto funkcia pracuje na tomto princípe:

- krátky reťazec (prázdny alebo jednoznakový) sa otáča jednoducho: netreba robiť nič, lebo on je zároveň aj otočeným reťazcom
- dlhšie reťazce otáčame tak, že z neho najprv odtrhneme prvý znak, otočíme zvyšok reťazca (to je už kratší reťazec ako pôvodný) a k nemu na koniec prilepíme odtrhnutý prvý znak

Toto funguje dobre, ale veľmi rýchlo narazíme na limity rekurzie: dlhší reťazec ako 1000 znakov už táto rekurzia nezvládne.

Vylepšime tento algoritmus takto:

- reťazec budeme skracovať o prvý aj posledný znak, takýto skrátený rekurzívne otočíme a tieto dva znaky opäť k reťazcu prilepíme, ale v opačnom poradí: na začiatok posledný znak a na koniec prvý:

```
def otoc(retazec):
    if len(retazec) <= 1:
        return retazec
    return retazec[-1] + otoc(retazec[1:-1]) + retazec[0]

print(otoc('Bratislava'))
print(otoc('Bratislava'*100))
print(otoc('Bratislava'*200))
```

Táto funkcia už pracuje pre 1000-znakový reťazec správne, ale opäť nefunguje pre reťazce dlhšie ako 2000.

Ďalšie vylepšenie tohto algoritmu už nie je také zrejmé:

- reťazec rozdelíme na dve polovice (pritom jedna z nich môže byť o 1 kratšia ako druhá)
- každú polovicu samostatne otočíme
- tieto dve otočené polovice opäť zlepíme dokopy, ale v opačnom poradí: najprv pôjde druhá polovica a za ňou prvá

```
def otoc(retazec):
    if len(retazec) <= 1:
        return retazec
    prva = otoc(retazec[:len(retazec)//2])
    druha = otoc(retazec[len(retazec)//2:])
    return druha + prva

print(otoc('Bratislava'))
print(otoc('Bratislava'*100))
print(otoc('Bratislava'*200))
r = otoc('Bratislava'*100000)
print(len(r), r == ('Bratislava'*100000)[::-1])
```

Zdá sa, že tento algoritmus už nemá problém s obmedzením na hĺbku vnorenia rekurzie. Zvládol aj 1000000 znakový reťazec.

Vidíme, že pri rozmýšľaní nad rekurzívnym riešením problému je veľmi dôležité správne rozdeliť **veľký** problém na jeden alebo aj viac menších, tie rekurzívne vyriešiť a potom to správne spojiť do jedného výsledku. Pri takomto rozhodovaní funguje matematická intuícia a tiež nemalá programátorská skúsenosť. Hoci nie vždy to ide tak elegantne, ako pri otáčaní reťazca.

Ďalší príklad ilustruje využitie rekurzie pri výpočte binomických koeficientov. Vieme, že **binomické kofícienty** sa dajú vypočítať pomocou matematického vzorca:

```
bin(n, k) = n! / (k! * (n-k)!)
```

Teda výpočtom nejakých troch faktoriálov a potom ich delením. Pre veľké `n` to môžu byť dosť veľké čísla, napr. `bin(1000,1)` potrebuje vypočítať `1000!` a tiež `999!`, čo sú dosť veľké čísla, ale ich vydelením dostávame výsledok len `1000`. Takýto postup počítať binomické koeficienty pomocou faktoriálov asi nie je najvhodnejší.

Tieto koeficienty ale vieme zobraziť aj pomocou **Pascalovho trojuholníka**, napr.:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5  10  10   5   1
```

Každý prvok v tomto trojuholníku zodpoveda `bin(n,k)`, kde `n` je riadok tabuľky a `k` je stĺpec. Pre túto tabuľku poznáme aj takýto vzťah:

```
bin(n,k) = bin(n-1,k-1) + bin(n-1,k)
```

t.j. každé číslo je súčtom dvoch čísel v riadku nad sebou, pričom na kraji tabuľky sú 1. To je predsa krásna rekurzia, v ktorej kraj tabuľky je triviálny prípad:

```
def bin(n, k):
    if k == 0 or n == k:
        return 1
    return bin(n - 1, k - 1) + bin(n - 1, k)

for n in range(6):
    for k in range(n + 1):
        print(bin(n, k), end=' ')
    print()
```

po spustení:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

Všimnite si, že v tomto algoritme nie je žiadne násobenie iba sčitovanie a ak by sme aj toto sčitovanie previedli na zreťazovanie reťazcov, videli by sme:

```
def bin_retazec(n, k):
    if k == 0 or n == k:
        return '1'
    return bin_retazec(n - 1, k - 1) + '+' + bin_retazec(n - 1, k)

print(bin(6, 3), '=', bin_retazec(6, 3))
```

```
20 = 1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1
```

Rekurzívny algoritmus pre výpočet binárnych koeficientov by mohol využívať vlastne len pričitávanie jednotky.

Fibonacciho čísla

Na podobnom princípe ako napr. výpočet faktoriálu, funguje aj **fibonacciho postupnosť** čísel: postupnosť začína dvomi členmi `0, 1`. Každý ďalší člen sa vypočíta ako súčet dvoch predchádzajúcich, teda:

- triviálny prípad: **fib(0) = 0**
- triviálny prípad: **fib(1) = 1**
- rekurzívny popis: **fib(n) = fib(n-1) + fib(n-2)**

Zapíšeme to v Pythone:

```
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

```
>>> for i in range(15):
        print(fib(i), end=', ')

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,
```

Tento rekurzívny algoritmus je ale **veľmi** neefektívny, napr. `fib(100)` asi nevypočítate ani na najrýchlejšom počítači.

V čom je tu problém? Veď nerekurzívne je to veľmi jednoduché, napr.:

```
def fib(n):
    a, b = 0, 1
    while n > 0:
        a, b = b, a + b
        n -= 1
    return a

for i in range(15):
    print(fib(i), end=', ')

print('\nfib(100) =', fib(100))
```

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,
fib(100) = 354224848179261915075
```

Pridajme do rekurzívnej verzie funkcie `fib()` globálne počítadlo, ktoré bude počítať počet zavolaní tejto funkcie:

```
def fib(n):
    global pocet
    pocet += 1
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

pocet = 0
print('fib(15) =', fib(15))
print('pocet volani funkcie =', pocet)
pocet = 0
print('fib(16) =', fib(16))
print('pocet volani funkcie =', pocet)
```

```
fib(15) = 610
pocet volani funkcie = 1973
fib(16) = 987
pocet volani funkcie = 3193
```

Vidíme, že tento počet volaní veľmi rýchlo rastie a je určite väčší ako samotné fibonnaciho číslo. Preto aj `fib(100)` by trvalo veľmi dlho (vyše **354224848179261915075** volaní funkcie).

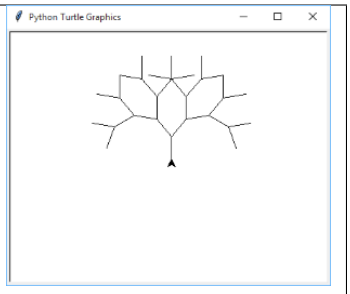
12.2 Binárne stromy

Medzi informatikmi sú veľmi populárne binárne stromy. Rekurzívne kresby binárnych stromov sa najlepšie kreslia pomocou grafického pera korytnačky. Aby sa nám lepšie o binárnych stromoch rozprávalo, zavedieme pojem **úroveň** stromu, t.j. číslo `n`, pre ktoré platí:

- ak je úroveň stromu `n` = 0, nakreslí sa len čiara nejakej dĺžky (kmeň stromu)
- pre `n` >= 1, sa najprv nakreslí čiara, potom sa na jej konci nakreslí najprv **vľavo** celý binárny strom úrovne `(n-1)` a potom **vpravo** opäť binárny strom úrovne `(n-1)` (hovoríme im podstromy) - po nakreslení týchto podstromov sa ešte vráti späť po prvej nakreslenej čiare
- po skončení kreslenia stromu ľubovoľnej úrovne sa korytnačka nachádza na mieste, kde začala kresliť
- ľavé aj pravé podstromy môžu mať buď rovnako veľké konáre ako kmeň stromu, alebo sa môžu v nižších úrovniach (teda v podstromoch) zmenšovať

Úroveň stromu nám hovorí o počte rekurzívnych vnorení pri kreslení stromu (podobne budeme neskôr definovať aj iné rekurzívne obrázky a často budeme pritom používať pojem úroveň).

Najprv ukážeme binárny strom, ktorý má vo všetkých úrovniach rovnako veľké podstromy:

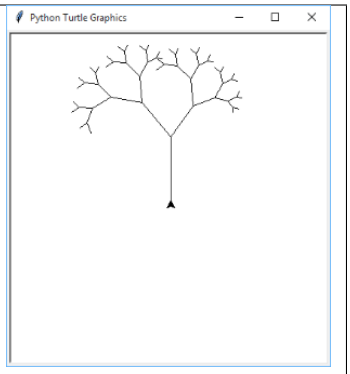



```
import turtle

def strom(n):
    if n == 0:
        t.fd(30)      # triviálny prípad
        t.bk(30)
    else:
        t.fd(30)
        t.lt(40)      # natoč sa na kreslenie ľavého␣
↪podstromu
        strom(n - 1) # nakresli ľavý podstrom (n-1).␣
↪úrovne
        t.rt(80)      # natoč sa na kreslenie pravého␣
↪podstromu
        strom(n - 1) # nakresli pravý podstrom (n-1).␣
↪úrovne
        t.lt(40)      # natoč sa do pôvodného smeru
        t.bk(30)      # vráť sa na pôvodné miesto

t = turtle.Turtle()
t.lt(90)
strom(4)
```

Binárne stromy môžeme rôzne vylepšovať, napr. vetvy stromu sa vo vyšších úrovniach môžu rôzne skracovať, uhol o ktorý je natočený ľavý a pravý podstrom môže byť tiež rôzny. V tomto riešení si všimnite, kde je skrytý triviálny prípad rekurzie:




```
import turtle

def strom(n, d):
    t.fd(d)
    if n > 0:
        t.lt(40)
        strom(n - 1, d * 0.7)
        t.rt(75)
        strom(n - 1, d * .6)
        t.lt(35)
    t.bk(d)

t = turtle.Turtle()
t.lt(90)
strom(5, 80)
```

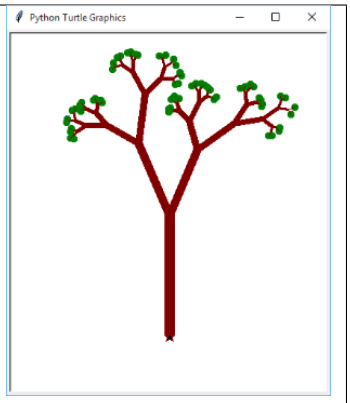
Algoritmus binárneho stromu môžeme zapísať aj bez parametra `n`, ktorý určuje úroveň stromu. V tomto prípade rekurzia končí, keď sú kreslené úsečky príliš malé:

```
import turtle

def strom(d):
    t.fd(d)
    if d > 5:
        t.lt(40)
        strom(d * 0.7)
        t.rt(75)
        strom(d * 0.6)
        t.lt(35)
    t.bk(d)

turtle.delay(0)
t = turtle.Turtle()
t.lt(90)
strom(80)
```

Ak využijeme náhodný generátor, môžeme vytvárať stromy, ktoré budú navzájom rôzne:




```
import turtle
import random

def strom(n, d):
    t.pensize(2 * n + 1)
    t.fd(d)
    if n == 0:
        t.dot(10, 'green')
    else:
        uhol1 = random.randint(20, 40)
        uhol2 = random.randint(20, 60)
        t.lt(uhol1)
        strom(n - 1, d * random.randint(40, 70) / 100)
        t.rt(uhol1 + uhol2)
        strom(n - 1, d * random.randint(40, 70) / 100)
        t.lt(uhol2)
    t.bk(d)

turtle.delay(0)
t = turtle.Turtle()
t.lt(90)
t.pencolor('maroon')
strom(6, 150)
```

V tomto riešení si všimnite, kde sme zmenili hrúbku pera, aby sa strom kreslil rôzne hrubý v rôznych úrovniach. Tiež sa tu na posledných „konároch“ nakreslili zelené listy - pridali sme ich v triviálnom prípade. Využili sme tu korytnačiu metódu `t.dot(vel'kost', farba)`, ktorá na pozícii korytnačky nakreslí bodku danej veľkosti a farby.

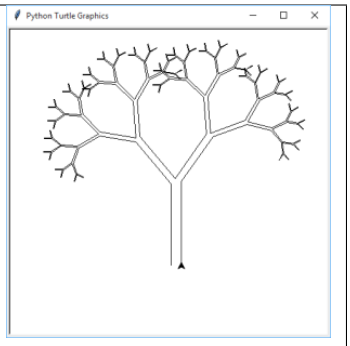
Každé spustenie tohto programu nakreslí trochu iný strom. Môžeme vytvoriť celú alej stromov, v ktorej bude každý strom trochu iný:

```
import turtle
import random

def strom(n, d):
    t.pensize(2 * n + 1)
    t.fd(d)
    if n == 0:
        t.dot(10, 'green')
    else:
        uhol1 = random.randint(20, 40)
        uhol2 = random.randint(20, 60)
        t.lt(uhol1)
        strom(n - 1, d * random.randint(40, 70) / 100)
        t.rt(uhol1 + uhol2)
        strom(n - 1, d * random.randint(40, 70) / 100)
        t.lt(uhol2)
    t.bk(d)

turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
t.lt(90)
t.pencolor('maroon')
for i in range(6):
    t.pu()
    t.setpos(100 * i - 250, -50)
    t.pd()
    strom(5, 50)
```

V nasledujúcom riešení vzniká zaujímavý efekt tým, že v triviálnom prípade urobí korytnačka malý úkrok vpravo a teda sa nevracia po tých istých čiarach a preto sa ani nevráti presne na to isté miesto, kde štartovala kresliť (pod)strom. Táto „chybička“ sa stále zväčšuje a zväčšuje, až pri nakreslení kmeňa stromu je už dosť veľká:



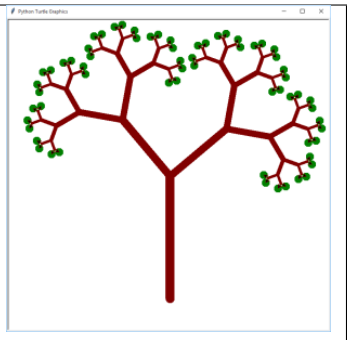

```
import turtle

def strom(n, d):
    t.fd(d)
    if n == 0:
        t.rt(90)
        t.fd(1)
        t.lt(90)
    else:
        t.lt(40)
        strom(n-1,d*.67)
        t.rt(75)
        strom(n-1,d*.67)
        t.lt(35)
    t.bk(d)

turtle.delay(0)
t = turtle.Turtle()
t.lt(90)
strom(6, 120)
```

Binárny strom sa dá nakresliť viacerými spôsobmi aj nerekurzívne. V jednom z nich využijeme zoznam korytnačiek, pričom každá z nich po nakreslení jednej úsečky „narodí“ na svojej pozícii ďalšiu korytnačku (vytvorí svoju kópiu), pričom ju ešte trochu otočí. Idea algoritmu je takáto:

- prvá korytnačka nakreslí kmeň stromu - prvú úsečku dĺžky `d`
- na jeho konci (na momentálnej pozícii tejto korytnačky) sa vyrobí jedna nová korytnačka, s novým relatívnym natočením o 40 stupňov vľavo (pripravuje sa, že bude kresliť ľavý podstrom) a sama sa otočí o 50 stupňov vpravo (pripravuje sa, že bude kresliť pravý podstrom)
- dĺžka `d` sa zníži napr. na `d` * 0.6
- všetky korytnačky teraz prejdú v svojom smere dĺžku `d` (nakreslia úsečku dĺžky `d`) a opäť sa na ich koncových pozíciách vytvoria nové korytnačky otočené o 40 a samé sa otočia o 50 stupňov, a `d` sa opäť zníži
- toto sa opakuje `n` krát a takto sa nakreslí kompletný strom



```
import turtle

def nova(pos, heading):
    t = turtle.Turtle()
    #t.speed(0)
    #t.ht()
    t.pu()
    t.setpos(pos)
    t.seth(heading)
    t.pd()
    return t

def strom(n, d):
    pole = [nova([0, -300], 90)]
    for i in range(n):
        for j in range(len(pole)):
            t = pole[j]
            t.pensize(3 * n - 3 * i + 1)
            t.pencolor('maroon')
            t.fd(d)
            if i == n - 1:
                t.dot(20, 'green')
            else:
                pole.append(nova(t.pos(), t.heading() +␣
↪40))
                t.rt(50)
        d *= 0.6

    print('pocet korytnaciek =', len(pole))

#turtle.delay(0)
strom(7, 300)
```

Pre korytnačky na poslednej úrovni sa už ďalšie nevytvárajú, ale na ich koncoch sa nakreslí zelená bodka. Program na záver vypíše celkový počet korytnačiek, ktoré sa takto vyrobili (je ich presne toľko, koľko je zelených bodiek ako listov stromu). Všimnite si pomocnú funkciu `nova()`, ktorá vytvorí novú korytnačku a nastaví jej novú pozíciu aj smer natočenia. Funkcia ako výsledok vráti túto novovytvorenú korytnačku. V tomto prípade program vypísal:

```
pocet korytnaciek = 64
```

12.3 Ďalšie rekurzívne obrázky

Napíšeme funkciu, ktorá nakreslí obrázok `stvorce` úrovne `n`, veľkosti `a` s týmito vlastnosťami:

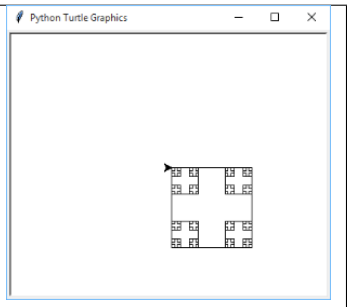
- pre `n` = 0 nekreslí nič
- pre `n` = 1 kreslí štvorec so stranou dĺžky `a`
- pre `n` > 1 kreslí štvorec, v ktorom v každom jeho rohu (smerom dnu) je opäť obrázok `stvorce` ale už zmenšený: úrovne `n`-1 a veľkosti `a`/3

Štvorce v každom rohu štvorca:

```
import turtle

def stvorce(n, a):
    if n == 0:
        pass
    else:
        for i in range(4):
            t.fd(a)
            t.rt(90)
            stvorce(n - 1, a / 3)
            # skúste: stvorce(n - 1, a * 0.45)

turtle.delay(0)
t = turtle.Turtle()
stvorce(4, 300)
```



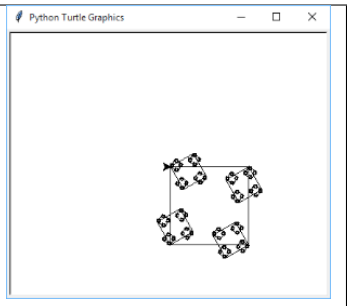
Uvedomte si, že toto nie je chvostová rekurzia.

Tesne pred rekurzívnym volaním otočíme korytnačku o 30 stupňov a po návrate z rekurzie týchto 30 stupňov vrátime:

```
import turtle

def stvorce(n, d):
    if n > 0:
        for i in range(4):
            t.fd(d)
            t.rt(90)
            t.lt(30)
            stvorce(n - 1, d / 3)
            t.rt(30)

turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
stvorce(5, 300)
```



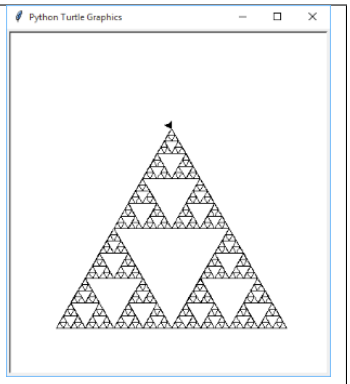
Sierpiňského trojuholník

Rekurzívny obrázok na rovnakom princípe ako boli vnorené štvorce ale trojuholníkového tvaru navrhol poľský matematik Sierpiňský ešte v roku 1915:

```
import turtle

def trojuholniky(n, a):
    if n > 0:
        for i in range(3):
            t.fd(a)
            t.rt(120)
            trojuholniky(n - 1, a / 2)

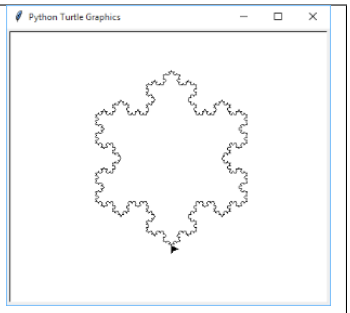
turtle.delay(0)
t = turtle.Turtle()
#t.speed(0)
t.rt(60)
trojuholniky(6, 400)
```



Zaujímavé je to, že táto rekurzívna krivka sa dá nakresliť aj jedným ťahom (po každej čiare sa prejde len raz). Porozmýšľajte ako.

Snehová vločka

Ďalej ukážeme veľmi známu rekurzívnu krivku - snehovú vločku (známu tiež ako Kochova krivka):



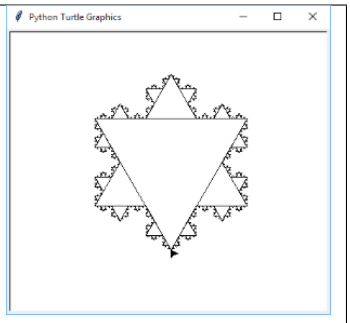

```
import turtle

def vlocka(n, d):
    if n == 0:
        t.fd(d)
    else:
        vlocka(n - 1, d / 3)
        t.lt(60)
        vlocka(n - 1, d / 3)
        t.rt(120)
        vlocka(n - 1, d / 3)
        t.lt(60)
        vlocka(n - 1, d / 3)

def sneh_vlocka(n, d):
    for i in range(3):
        vlocka(n, d)
        t.rt(120)

turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
t.lt(120)
sneh_vlocka(4, 300)
```

Ak namiesto jedného volania funkcie `vlocka()` zapíšeme nakreslenie aj všetkých predchádzajúcich úrovní krivky, dostávame tiež zaujímavú kresbu:




```
import turtle

def vlocka(n, d):
    if n == 0:
        t.fd(d)
    else:
        vlocka(n - 1, d / 3)
        t.lt(60)
        vlocka(n - 1, d / 3)
        t.rt(120)
        vlocka(n - 1, d / 3)
        t.lt(60)
        vlocka(n - 1, d / 3)

def sneh_vlocka(n, d):
    for i in range(3):
        vlocka(n, d)
        t.rt(120)

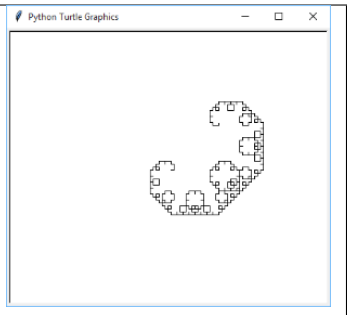
turtle.delay(0)
t = turtle.Turtle()
#t.speed(0)
t.lt(120)
for i in range(5):
    sneh_vlocka(i, 300)
```

Ďalšie fraktálové krivky

Špeciálnou skupinou rekurzívnych kriviek sú fraktály (pozri aj na wikipédii). Už pred érou počítačov sa s nimi „hrali“ aj významní matematici (niektoré krivky podľa nich dostali aj svoje meno, aj snehová vločka je fraktálom a vymyslel ju švédsky matematik Koch). Zjednodušene by sme mohli povedať, že fraktál je taká krivka, ktorá sa skladá z viacerých svojich zmenšených kópií. Keby sme sa na nejakú jej časť pozreli lupou, videli by sme opäť skoro tú istú krivku. Napr. aj binárne stromy a aj snehové vločky sú fraktály.

C-krivka

Začneme veľmi jednoduchou, tzv. C-krivkou:



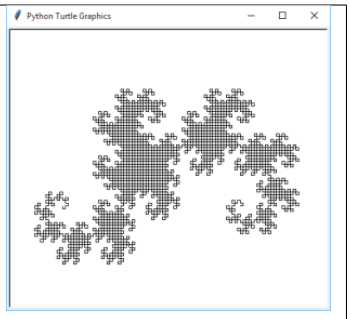

```
import turtle

def ckrivka(n, s):
    if n == 0:
        t.fd(s)
    else:
        ckrivka(n - 1, s)
        t.lt(90)
        ckrivka(n - 1, s)
        t.rt(90)

turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
t.ht()
ckrivka(9, 4)     # skúste aj: ckrivka(13, 2)
```

Dračia krivka

C-krivke sa veľmi podobá Dračia krivka, ktorá sa skladá z dvoch „zrkadlových“ funkcií: `ldrak` a `pdrak`. Všimnite si zaujímavú vlastnosť týchto dvoch rekurzívnych funkcií: prvá rekurzívne volá samu seba ale aj druhú a druhá volá seba aj prvú. Našťastie Python toto zvláda veľmi dobre:



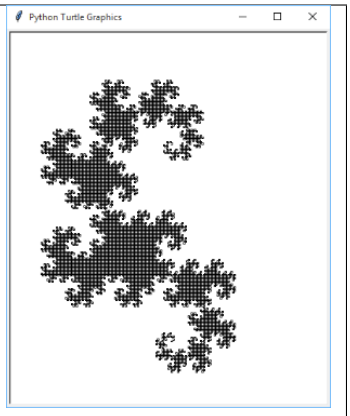

```
import turtle

def ldrak(n, s):
    if n == 0:
        t.fd(s)
    else:
        ldrak(n - 1, s)
        t.lt(90)
        pdrak(n - 1, s)

def pdrak(n, s):
    if n == 0:
        t.fd(s)
    else:
        ldrak(n - 1, s)
        t.rt(90)
        pdrak(n - 1, s)

turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
t.ht()
ldrak(12, 6)
```

Dračiu krivku môžeme nakresliť aj len jednou funkciou - táto bude mať o jeden parameter `u` viac a to, či je to ľavá alebo pravá verzia funkcie:




```
import turtle

def drak(n, s, u=90):
    if n == 0:
        t.fd(s)
    else:
        drak(n - 1, s, 90)
        t.lt(u)
        drak(n - 1, s, -90)

turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
t.ht()
drak(14, 2)
```

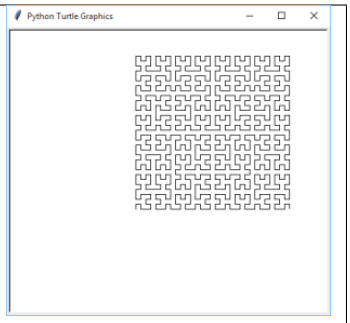
Hilbertova krivka

V literatúre je veľmi známou Hilbertova krivka, ktorá sa tiež skladá z dvoch zrkadlových častí (ako dračia krivka) a preto ich definujeme jednou funkciou a parametrom `u` (t.j. uhol pre ľavú a pravú verziu):

```
import turtle

def hilbert(n, s, u=90):
    if n > 0:
        t.lt(u)
        hilbert(n - 1, s, -u)
        t.fd(s)
        t.rt(u)
        hilbert(n - 1, s, u)
        t.fd(s)
        hilbert(n - 1, s, u)
        t.rt(u)
        t.fd(s)
        hilbert(n - 1, s, -u)
        t.lt(u)

turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
t.ht()
hilbert(5, 7)           # vyskúšajte: hilbert(7, 2)
```



Ďalšie inšpirácie na rekurzívne krivky môžete nájsť na wikipédii.

12.4 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/
- pozrite si *Riešenie 12. cvičenia*

Rekurzívne výpočty

1. Daná je funkcia `urob(a, b)`.
   - zistite bez spúšťania v Pythone, čo vypočíta `urob(7, 17)`

```
def urob(a, b):
    if a == 0:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        return 0
    return b + urob(a - 1, b)
```

- potom to môžete otestovať napr. vo Visualize Python

2. Daná funkcia `retazec(n)` na základe čísla `n` vráti nejaký znakový reťazec
   - zistite bez spúšťania v Pythone, čo sa vypíše

```
def retazec(n):
    if n == 0:
        return ''
    if n == 1:
        return 'a'
    return retazec(n-2) + '+' + retazec(n-1)

for i in range(5):
    print(i, repr(retazec(i)))
```

3. Máme danú funkciu `w(n)`.
   - zistite bez spúšťania v Pythone, čo vypočíta `w(3)`:

```
def w(n):
    s = 0
    while n:
        s += n + w(n-1)
        n -= 1
    return s
```

   - zamyslite sa nad tým, kde je v tejto rekurzívnej funkcii triviálny prípad

4. Funkcia `urob(a, b)` z prvej úlohy počíta bez cyklov súčin dvoch nezáporných celých čísel a to len pomocou sčitovania. Napíšte funkciu `umocni(a, b)`, ktorá pre dve celé čísla vypočíta mocninu `a**b` ale bez cyklu a bez násobenia - na násobenie použite funkciu `urob()`.
   - otestujte:

```
>>> umocni(3, 7)
```

5. Napíšte dve rekurzívne funkcie `tostr(cislo)` a `toint(retazec)`, ktoré bez cyklov a len pomocou štandardných funkcií `ord()` a `chr()` prevedú celé nezáporné číslo na znakový reťazec a naopak. Obe funkcie nič nevypisujú len vracajú nejakú hodnotu.
   - otestujte napr.

```
>>> tostr(1276)
'1276'
>>> toint('1276')
1276
```

6. Napíšte rekurzívnu funkciu `pocet(znak, retazec)`, ktorá bez cyklu a reťazcových metód zistí počet výskytov zadaného znaku vo vstupnom reťazci.
   - otestujte napr.

```
>>> pocet('m', 'mama ma emu a ema ma mamu')
8
```

7. Prechádzajúci príklad vyriešte tak, aby fungoval aj pre dlhšie reťazce. Inšpirujte sa funkciou `otoc()` z prednášky

   - otestujte napr.

```
>>> pocet('a', 'ab'*100000)
100000
```

8. Napíšte funkciu `vela(n)`, ktorá vráti znakový reťazec `'='*(2**n)`, t.j. obsahuje len znak `'='`, ktorý sa opakuje `2**n` krát. Riešte rekurzívne bez cyklov a viacnásobného zreťazovania reťazcov (bez operácie `*` s reťazcami).

   - otestujte napr.

```
>>> vela(0)
'='
>>> vela(5)
'================================'
```

9. Zapíšte funkciu `nsd(a, b)` (najväčší spoločný deliteľ) rekurzívne: triviálny prípad je vtedy, keď `a==b`, inak ak `a>b`, tak rekurzívne vypočíta `nsd(b, a)`, inak rekurzívne zavolá `nsd(a, b-a)`.

   - otestujte napr.

```
>>> nsd(40, 24)
8
```

10. Napíšte rekurzívnu funkciu `sucet(zoznam)`, ktorá bez cyklov zistí súčet prvkov zoznamu. Prvkami sú len celé čísla.

    - otestujte napr.

```
>>> sucet([2, 4, 6, 8])
20
>>> sucet([])
0
```

11. Upravte funkciu `sucet(zoznam)` z predchádzajúceho príkladu tak, aby prvkami zoznamu mohli byť nielen celé čísla, ale aj zoznamy, ktoré obsahujú celé čísla.

    - otestujte napr.

```
>>> sucet([2, 4, [1, 2, 3], 8])
20
```

    - vyriešte úlohu tak, aby funkcia fungovala aj pre ľubovoľný počet vnorení zoznamov, napr.

```
>>> sucet([2, [[4]], [1, [2, 3]], 8])
20
```

12. Napíšte funkciu `tolist(zoznam)`, ktorá z ľubovoľného zoznamu, ktoré obsahuje aj iné (vnorené) zoznamy vráti zoznam (typu `list`) všetkých prvkov, ktoré sa v týchto zoznamoch nachádzajú. Nepoužite cyklus ale rekurziu.

    - otestujte napr.

```
>>> tolist([2, [['4a']], [1, [2, 3.14]], 8])
[2, '4a', 1, 2, 3.14, 8]
>>> tolist([[[[], []], []])
[]
```

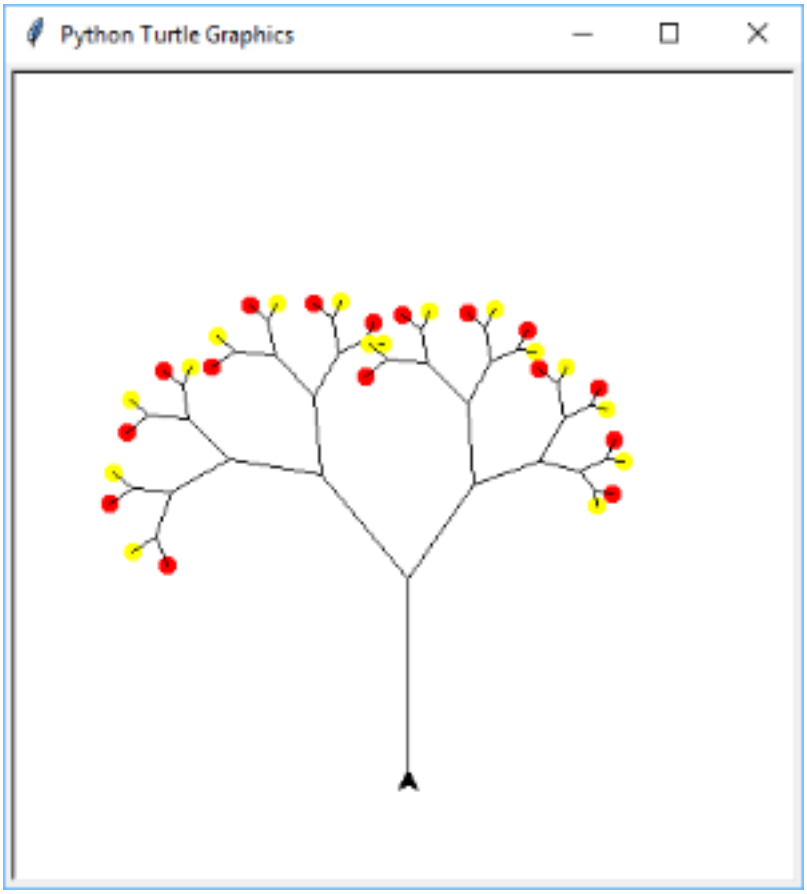
Rekurzívne krivky

13. Na prednáške sme kreslili binárny strom.

    - strom úrovne `n`:

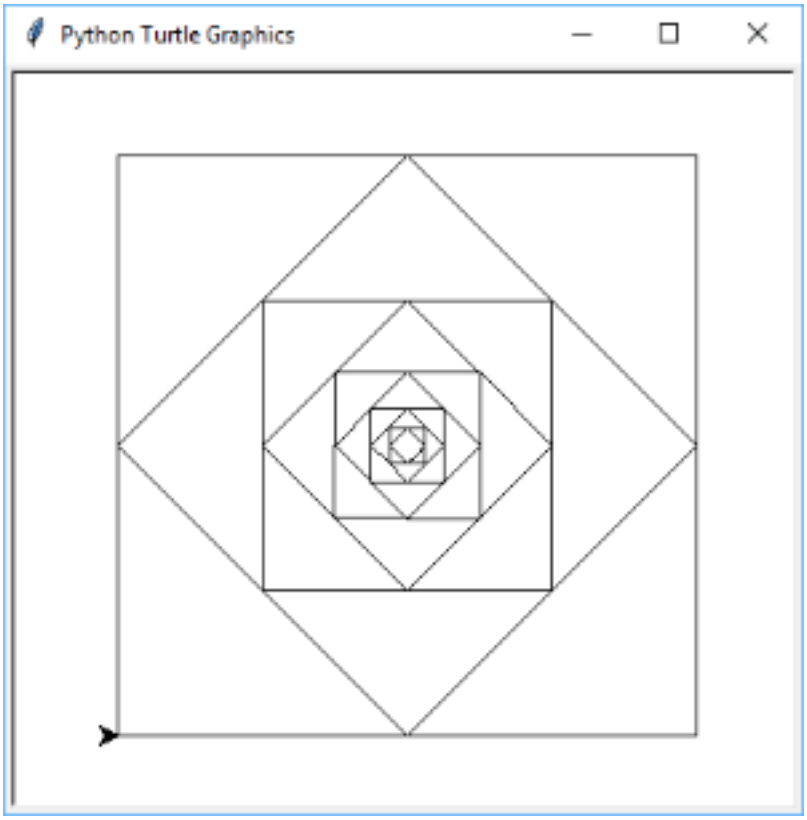
```
def strom(n, d):
    t.fd(d)
    if n > 0:
        t.lt(40)
        strom(n - 1, d * 0.7)
        t.rt(75)
        strom(n - 1, d * .6)
        t.lt(35)
    t.bk(d)
```

    - dopíšte do tejto funkcie kreslenie farebných bodiek (pomocou `t.dot(10, farba)`) na koncových vetvičkách tak, aby sa pravideľne striedali dve farby červená a žltá; okrem korytnačky `t` nepoužívajte iné globálne premenné
    - napr.

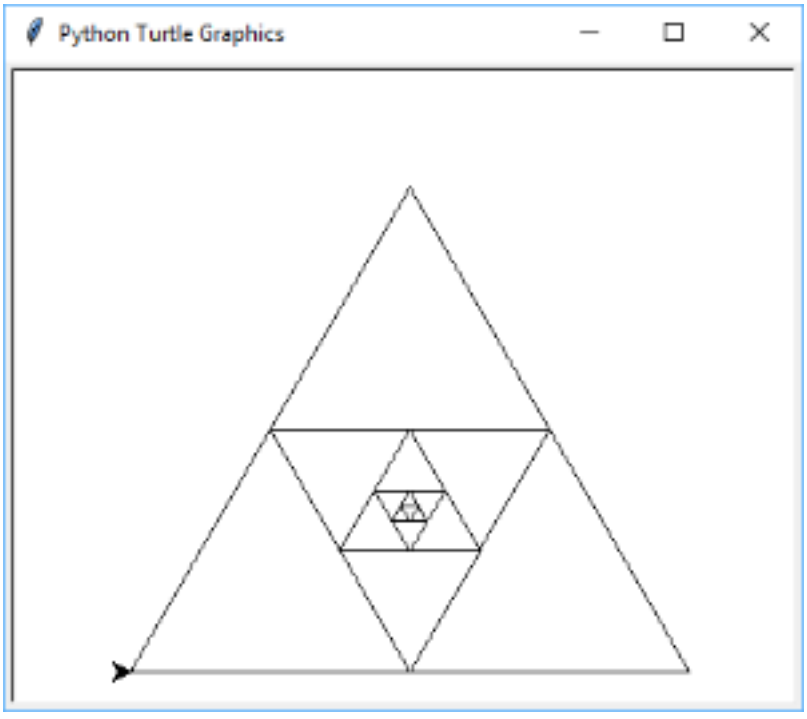


14. Nakreslite rekurzívnu krivku `stvorce(n, a)`, ktorá pre `n > 0` nakreslí štvorec so stranou `a`, v ktorom sú vpísané štvorce (s vrcholmi v stredoch strán vonkajšieho štvorca). Tieto vnorené štvorce vzniknú volaním `stvorce(n-1, ...)`. Pre `n=2` sme to riešili bez rekurzie na predchádzajúcom cvičení.

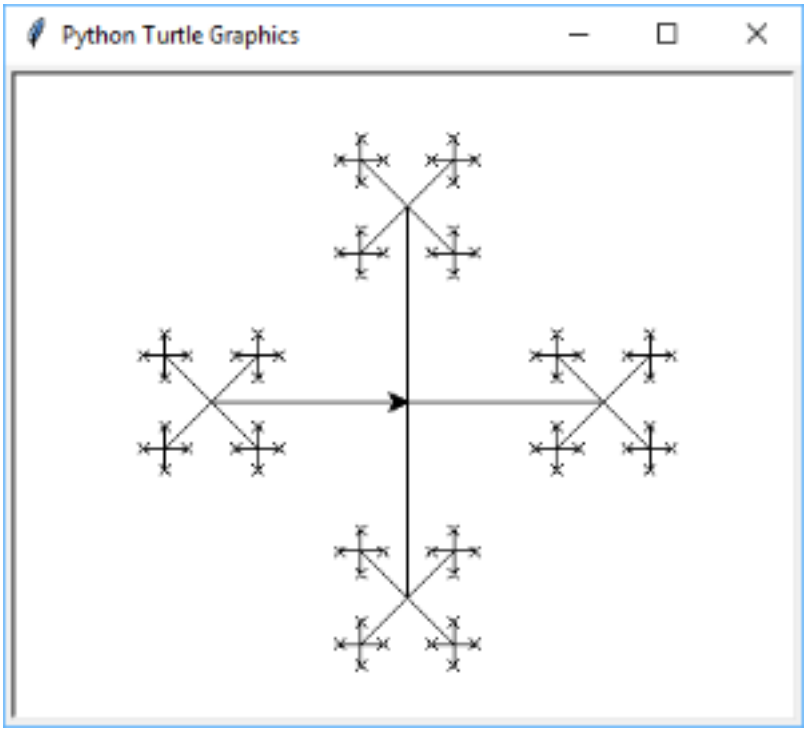
    - napr. pre volanie `stvorce(10, 300)` dostaneme:



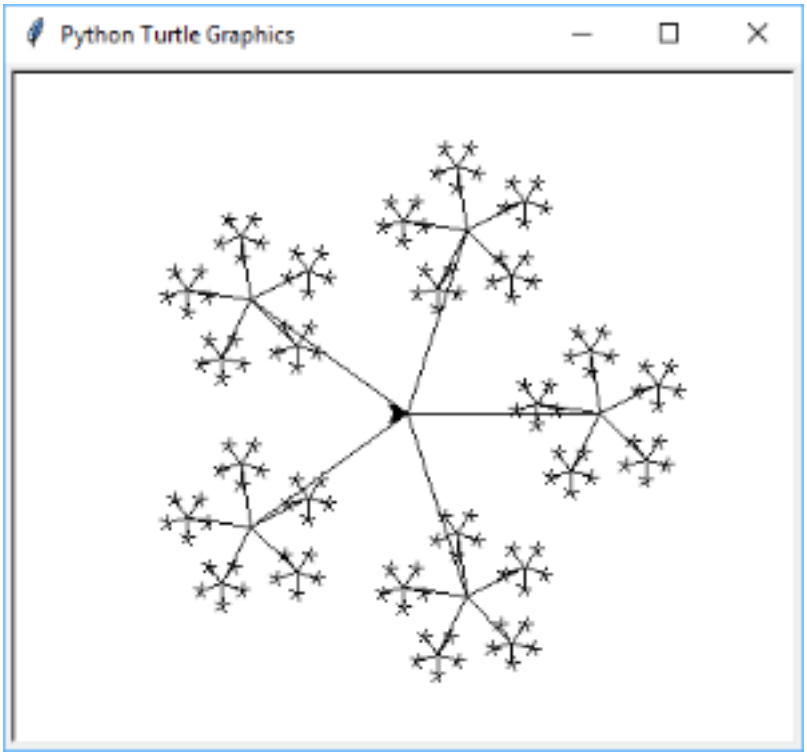
- zamyslite sa, či by bol veľký problém naprogramovať to tak, aby sa po žiadnej čiare neprešlo pri kreslení viackrát (aby sa útvar nakreslil jedným ťahom)
- malou zmenou rekurzívnej funkcie vieme nakresliť napr. vpísané trojuholníky, napr. pre volanie `trojuholniky(6, 300)`:



15. Zadefinujte funkciu `kriziky(n, d)`, ktorá nakreslí takúto rekurzívnu krivku:
    - pre `n=0` funkcia nerobí nič
    - pre `n=1` funkcia nakreslí kríž (4 na seba kolmé čiary zadanej dĺžky `d`) - kreslenie sa skončí tam, kde sa začalo
    - pre `n=2` funkcia opäť nakreslí kríž zadanej veľkosti, ale na konci všetkých 4 ramien kríža nakreslí tiež kríž ale s ramenami tretinovej veľkosti, pričom tieto menšie krížiky budú oproti ramenu otočené o 45 stupňov (nakreslí sa 1 kríž veľkosti `d` a 4 veľkosti `d/3`)
    - každá ďalšia úroveň krivky nakreslí menšie a menšie krížiky na konci ramien vnorených krížov (pre `n=3` sa nakreslí 1 kríž veľkosti `d`, 4 veľkosti `d/3` a 16 veľkosti `d/9`)
    - napr. pre volanie `kriziky(4, 100)` dostávame:



    - malou zmenou rekurzívnej funkcie vieme zmeniť 4 ramená krížikov na ľubovoľný iný počet, napr. pre volanie `kriziky(4, 300, 5)` dostávame:



12.5 5. Domáce zadanie

L.I.S.T.

- riešenie odovzdávajte na úlohový server https://list.fmph.uniba.sk/

Napíšte pythonovský skript, v ktorom zadefinujete štyri funkcie na prácu so zoznamami. Vstupom pre všetky tieto funkcie bude zoznam, ktorý môže obsahovať čísla, reťazce, ale aj ďalšie podzoznamy. Tieto podzoznamy môžu opäť obsahovať ďalšie podzoznamy, atď. Napríklad aj takýto zoznam `['a', ['dom', [2], 3], [], [[[2]]], 'b']` môže byť vstupom do vašich funkcií.

zoznam_prvkov()

Funkcia `zoznam_prvkov(zoznam)` vráti sploštený zoznam prvkov daného zoznamu, teda taký, ktorý už neobsahuje žiadne podzoznamy. Napr.

```
>>> print(zoznam_prvkov([1, 2, 3, [4, 5], 6, [[[7]]], [], 8]))
[1, 2, 3, 4, 5, 6, 7, 8]
>>> print(zoznam_prvkov(['a', ['dom', [2], 3], [], [[[2]]], 'b']))
['a', 'dom', 2, 3, 2, 'b']
>>> print(zoznam_prvkov([[], [[[]]], []]))
[]
>>> zoz = [[[7]], 8]
>>> print(zoznam_prvkov(zoz))
[7, 8]
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> zoz
[[[7]], 8]
```

Pôvodný zoznam pri tom ostane bez zmeny.

splosti()

Funkcia `splosti(zoznam)` sploští daný vstupný zoznam. Na rozdiel od predchádzajúcej funkcie táto nič nevracia, len modifikuje vstupný zoznam. Napr.

```
>>> zoz = [[[7]], 8]
>>> print(splosti(zoz))
None
>>> zoz
[7, 8]
>>> p = [1, 2, 3, [4, 5], 6, [[[7]]], [], 8]
>>> splosti(p)
>>> p
[1, 2, 3, 4, 5, 6, 7, 8]
```

nahradeny_zoznam()

Funkcia `nahradeny_zoznam(zoznam, hodnota1, hodnota2)` vráti kópiu pôvodného zoznamu, v ktorom budú všetky prvky vstupného zoznamu s hodnotou `hodnota1` nahradené hodnotou `hodnota2`. Ak sú nejakými prvkami opäť zoznamy, tak bude `hodnota1` nahradená `hodnotou2` aj v týchto zoznamoch, aj ich podzoznamoch atď. Napr.

```
>>> zoz = [[[7]], 8]
>>> print(nahradeny_zoznam(zoz, 7, 'a'))
[[['a']], 8]
>>> zoz
[[[7]], 8]
>>> print(nahradeny_zoznam([1, 2, 3, [1, 2], 3, [[[1]]], [], 2], 1, 'x'))
['x', 2, 3, ['x', 2], 3, [[['x']]], [], 2]
>>> print(nahradeny_zoznam([3, [33, [333, [13], 13]], 36], 3, 'q'))
['q', [33, [333, [13], 13]], 36]
>>> print(nahradeny_zoznam([3, [33, [333, [13], 13]], 36], [13], 'm'))
[3, [33, [333, 'm', 13]], 36]
```

nahrad()

Funkcia `nahrad(zoznam, hodnota1, hodnota2)` nahradí v danom zozname `zoznam` všetky prvky s hodnotou `hodnota1` hodnotou `hodnota2`. Ak sú nejakými prvkami opäť zoznamy, tak nahrádza aj v týchto zoznamoch, aj v ich podzoznamoch atď. Funkcia nič nevracia, len modifikuje vstupný zoznam. Napr.

```
>>> zoz = [[[7]], 8]
>>> print(nahrad(zoz, 7, 'a'))
None
>>> zoz
[[['a']], 8]
>>> p = [1, 2, 3, [1, 2], 3, [[[1]]], [], 2]
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> nahrad(p, 1, 'x')
>>> p
['x', 2, 3, ['x', 2], 3, [[['x']]], [], 2]
>>> p = [1, 2, 3, [1, 2], 3, [[[1]]], [], 2]
>>> nahrad(p, 4, 'z')
>>> p
[1, 2, 3, [1, 2], 3, [[[1]]], [], 2]
>>> p = ['a', ['dom', [2], 3], [], [[[2]]], 'b']
>>> nahrad(p, 2, 'abc')
>>> p
['a', ['dom', ['abc'], 3], [], [[['abc']]], 'b']
```

Nemeňte mená funkcií a parametrov. Zrejme využijete rekurziu.

Váš odovzdaný program s menom `riesenie5.py` musí začínať tromi riadkami komentárov:

```
# 5. zadanie: zoznamy
# autor: Janko Hraško
# datum: 28.12.2017
```

Projekt `riesenie5.py` odovzdávajte na úlohový server https://list.fmph.uniba.sk/ najneskôr do 23:00 **5. januára**, kde ho môžete nechať otestovať. Testovač bude spúšťať vašu funkciu s rôznymi vstupmi. Odovzdať projekt aj ho testovať môžete ľubovoľný počet krát. Môžete zaň získať **10 bodov**.

KAPITOLA 13

13. Dvojrozmerné tabuľky

Pythonovský zoznam `list` (blízky jednorozmerným poliam v iných programovacích jazykoch) slúži hlavne na uchovávanie nejakej postupnosti alebo skupiny údajov. V takejto štruktúre sa dajú uchovávať aj dvojrozmerné tabuľky ako zoznam zoznamov (opäť je to analógia k dvojrozmerným poliam). Dvojrozmerné údaje sa často vyskytujú, napríklad ako rôzne hracie plochy (štvorčekový papier pre piškvorky, šachovnica pre doskové hry, rôzne typy labyrintov), ale napríklad aj rastrové obrázky sú často uchovávané v dvojrozmernej tabuľke. Aj v matematike sa niekedy pracuje s dvojrozmernými tabuľkami čísel (tzv. matice).

Už vieme, že prvkami zoznamu môžu byť opäť postupnosti (zoznamy alebo n-tice). Práve táto vlastnosť nám poslúži pri reprezentácii dvojrozmerných tabuliek. Napr. takúto tabuľku (matematickú maticu 3x3):

```
1 2 3
4 5 6
7 8 9
```

môžeme v Pythone zapísať:

```
>>> m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Zoznam `m` má tri prvky: sú to tri riadky (zoznamy čísel - teda jednorozmerné polia). Našou prvou úlohou bude vypísať takýto zoznam do riadkov. Ale takýto jednoduchý výpis sa nám nie vždy bude hodiť:

```
>>> for riadok in m:
        print(riadok)
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
```

Častejšie to budeme robiť dvoma vnorenými cyklami. Zadefinujme funkciu `vypis()` s jedným parametrom dvojrozmernou tabuľkou:

```
def vypis(tab):
    for riadok in tab:
        for prvok in riadok:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            print(prvok, end=' ')
        print()
```

To isté vieme zapísať aj pomocou indexovania:

```
def vypis(tab):
    for i in range(len(tab)):
        for j in range(len(tab[i])):
            print(tab[i][j], end=' ')
        print()
```

Obe tieto funkcie sú veľmi častými šablónami pri práci s dvojrozmernými tabuľkami. Teraz výpis tabuľky vyzerá takto:

```
>>> vypis(m)
1 2 3
4 5 6
7 8 9
```

13.1 Vytváranie dvojrozmerných tabuliek

Pozrime, ako môžeme vytvárať nové dvojrozmerné tabuľky. Okrem priameho priradenia, napr.

```
>>> matica = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

ich môžeme poskladať z jednotlivých riadkov, napr.

```
>>> riadok1 = [1, 2, 3]
>>> riadok2 = [4, 5, 6]
>>> riadok3 = [7, 8, 9]
>>> matica = [riadok1, riadok2, riadok3]
```

Častejšie to ale bude pomocou nejakých cyklov. Závisí to od toho, či sa vo výslednej tabuľke niečo opakuje. Vytvorme dvojrozmernú tabuľku veľkosti 3x3, ktorá obsahuje samé `0`:

```
>>> matica = []
>>> for i in range(3):
        matica.append([0, 0, 0])
>>> matica
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> vypis(matica)
0 0 0
0 0 0
0 0 0
```

Využili sme tu štandardný spôsob vytvárania jednorozmerného zoznamu pomocou metódy `append()`. Obsah ľubovoľného prvku matice môžeme zmeniť obyčajným priradením:

```
>>> matica[0][1] = 9
>>> matica[1][2] += 1
>>> vypis(matica)
0 9 0
0 0 1
0 0 0
```

Prvý index v `[]` zátvorkách väčšinou bude pre nás označovať poradové číslo riadka, v druhých zátvorkách je poradové číslo stĺpca. Už sme si zvykli, že riadky aj stĺpce sú číslované od 0.

Hoci pri definovaní matice sa zdá, že sa 3-krát opakuje to isté. Zapíšme to pomocou viacnásobného zreťazenia (operácia $*$) zoznamov:

```
>>> matica1 = [[0, 0, 0]] * 3
```

Opäť sa potvrdzuje, že je to veľmi nesprávny spôsob vytvárania prvkov zoznamu: zápis `[0, 0, 0]` označuje **referenciu** na trojprvkový zoznam, potom `[[0, 0, 0]]*3` rozkopíruje túto jednu referenciu trikrát. Teda vytvorili sme zoznam, ktorý trikrát obsahuje referenciu na ten istý riadok. Presvedčíme sa o tom priradením do niektorých prvkov takéhoto zoznamu:

```
>>> matica1[0][1] = 9
>>> matica1[1][2] += 1
>>> vypis(matica1)
0 9 1
0 9 1
0 9 1
```

Uvedomte si, že zápis:

```
>>> matica1 = [[0, 0, 0]] * 3
```

v skutočnosti znamená:

```
>>> riadok = [0, 0, 0]
>>> matica1 = [riadok, riadok, riadok]
```

Zapamätajte si! Dvojrozmerné štruktúry nikdy nevytvárame tak, že viacnásobne zreťazujeme (násobíme) jeden riadok viackrát. Pritom

```
>>> matica2 = [[0]*3, [0]*3, [0]*3]
```

je už v poriadku, lebo v tomto zozname sme vytvorili tri rôzne riadky.

Niekedy sa na vytvorenie „prázdnej“ dvojrozmernej tabuľky definuje funkcia:

```
def vyrob(pocet_riadkov, pocet_stlpcov, hodnota=0):
    vysl = []
    for i in range(pocet_riadkov):
        vysl.append([hodnota] * pocet_stlpcov)
    return vysl
```

Otestujme:

```
>>> a = vyrob(3, 5)
>>> vypis(a)
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
>>> b = vyrob(2, 6, '*')
>>> vypis(b)
* * * * * *
* * * * * *
```

Niekedy potrebujeme do takto pripravenej tabuľky priradiť nejaké hodnoty, napr. postupným zvyšovaním nejakého počítadla:

```
def ocisluj(tab):
    poc = 0
    for i in range(len(tab)):
        for j in range(len(tab[i])):
            tab[i][j] = poc
            poc += 1
```

Všimnite si, že táto funkcia vychádza z druhej funkcie (šablóny) pre vypisovanie dvojrozmernej tabuľky: namiesto výpisu prvku (`print()`) sme do neho niečo priradili. Táto funkcia `ocisluj()` nič nevypisuje ani nevracia žiadnu hodnotu „len“ modifikuje obsah tabuľky, ktorá je parametrom tejto funkcie.

```
>>> a = vyrob(3, 5)
>>> ocisluj(a)
>>> vypis(a)
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
```

13.1.1 Niekoľko príkladov práce s dvojrozmernými tabuľkami

1. zvýšime obsah všetkých prvkov o 1:

   ```
   def zvys_o_1(tab):
       for riadok in tab:
           for i in range(len(riadok)):
               riadok[i] += 1
   ```

   Zrejme všetky prvky tejto tabuľky musia byť nejaké čísla, inak by funkcia spadla na chybe.

   ```
   >>> p = [[5, 6, 7], [0, 0, 0], [-3, -2, -1]]
   >>> zvys_o_1(p)
   >>> p
   [[6, 7, 8], [1, 1, 1], [-2, -1, 0]]
   ```

2. podobný cieľ má aj druhá funkcia: hoci nemení samotnú tabuľku, vytvorí novú, ktorej prvky sú o jedna väčšie ako v pôvodnej tabuľke:

   ```
   def o_1_viac(tab):
       nova_tab = []
       for riadok in tab:
           novy_riadok = [0] * len(riadok)
           for i in range(len(riadok)):
               novy_riadok[i] = riadok[i] + 1
           nova_tab.append(novy_riadok)
       return nova_tab
   ```

   To isté trochu inak:

   ```
   def o_1_viac(tab):
       nova_tab = []
       for riadok in tab:
           novy_riadok = list(riadok)          # kopia povodneho riadka
           for i in range(len(novy_riadok)):
               novy_riadok[i] += 1
           nova_tab.append(novy_riadok)
       return nova_tab
   ```

3. kópia dvojrozmernej tabuľky:

```
def kopia(tab):
    nova_tab = []
    for riadok in tab:
        nova_tab.append(list(riadok))
    return nova_tab
```

4. číslovanie prvkov tabuľky inak ako to robila funkcia `cisluj()`: nie po riadkoch ale po stĺpcoch. Predpokladáme, že všetky riadky sú rovnako dlhé:

```
def ocisluj_po_stlpcoch(tab):
    poc = 0
    for j in range(len(tab[0])):
        for i in range(len(tab)):
            tab[i][j] = poc
            poc += 1
```

Všimnite si, že táto funkcia má oproti `ocisluj()` vymenené len dva riadky for-cyklov.

```
>>> a = vyrob(3, 5)
>>> ocisluj_po_stlpcoch(a)
>>> vypis(a)
0 3 6 9 12
1 4 7 10 13
2 5 8 11 14
```

5. spočítame počet výskytov nejakej hodnoty:

```
def pocet(tab, hodnota):
    vysl = 0
    for riadok in tab:
        for prvok in riadok:
            if prvok == hodnota:
                vysl += 1
    return vysl
```

Využili sme tu prvú verziu funkcie (šablóny) pre výpis dvojrozmernej tabuľky. Ak si ale pripomenieme, že niečo podobné robí štandardná metóda `count()`, ale táto funguje len pre jednorozmerné zoznamy, môžeme našu funkciu vylepšiť:

```
def pocet(tab, hodnota):
    vysl = 0
    for riadok in tab:
        vysl += riadok.count(hodnota)
    return vysl
```

Otestujeme:

```
>>> a = [[1, 2, 1, 2], [4, 3, 2, 1], [2, 1, 3, 1]]
>>> pocet(a, 1)
5
>>> pocet(a, 4)
1
>>> pocet(a, 5)
0
```

5. funkcia zistí, či je nejaká matica (dvojrozmerný zoznam) symetrická, t. j. či sú prvky pod a nad hlavnou uhlopriečkou rovnaké, čo znamená, že má platiť `matica[i][j] == matica[j][i]` pre každé `i` a `j`:

```
def symetricka(matica, hodnota):
    vysl = True
    for i in range(len(matica)):
        for j in range(len(matica[i])):
            if matica[i][j] != matica[j][i]:
                vysl = False
    return vysl
```

Hoci je toto riešenie správne, má niekoľko nedostatkov:

- funkcia zbytočne testuje každú dvojicu prvkov `matica[i][j]` a `matica[j][i]` dvakrát, napr. či `matica[0][2] == matica[2][0]` aj `matica[2][0] == matica[0][2]`, tiež zrejme netreba kontrolovať prvky na hlavnej uhlopriečke, či `matica[i][i] == matica[i][i]`
- keď sa vo vnútornom cykle zistí, že sme našli dvojicu `matica[i][j]` a `matica[j][i]`, ktoré sú navzájom rôzne, zapamätá sa, že výsledok funkcie bude `False` a ďalej sa pokračuje prehľadávať dvojrozmerný zoznam - toto je zrejme zbytočné, lebo výsledok je už známy - asi by sme mali vyskočiť z týchto cyklov; POZOR! príkaz `break` ale neurobí to, čo by sa nám tu hodilo:

```
def symetricka(matica, hodnota):
    vysl = True
    for i in range(len(matica)):
        for j in range(len(matica[i])):
            if matica[i][j] != matica[j][i]:
                vysl = False
                break                       # vyskočí z cyklu
    return vysl
```

Takéto vyskočenie z cyklu nám veľmi nepomôže, lebo vyskakuje sa len z vnútorného a ďalej sa pokračuje vo vonkajšom. Našťastie my tu nepotrebujeme vyskakovať z cyklu, ale môžeme priamo ukončiť celú funkciu aj s návratovou hodnotou `False`.

Prepíšme funkciu tak, aby zbytočne dvakrát nekontrolovala každú dvojicu prvkov a aby sa korektne ukončila, keď nájde nerovnakú dvojicu:

```
def symetricka(matica, hodnota):
    for i in range(1, len(matica)):
        for j in range(i):
            if matica[i][j] != matica[j][i]:
                return False
    return True
```

6. funkcia vráti pozíciu prvého výskytu nejakej hodnoty, teda dvojicu `(riadok, stĺpec)`. Keďže budeme potrebovať poznať indexy konkrétnych prvkov zoznamu, použijeme šablónu s indexmi:

```
def index(tab, hodnota):
    for i in range(len(tab)):
        for j in range(len(tab[i])):
            if tab[i][j] == hodnota:
                return i, j
```

Funkcia skončí, keď nájde prvý výskyt hľadanej hodnoty (prechádza po riadkoch zľava doprava):

```
>>> a = [[1, 2, 1, 2], [1, 2, 3, 4], [2, 1, 3, 1]]
>>> index(a, 3)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
(1, 2)
>>> index(a, 5)
>>>
```

Na tomto poslednom príklade vidíme, že naša funkcia `index()` v nejakom prípade nevrátila „nič“. My už vieme, že vrátila špeciálnu hodnotu `None`, ktorá sa ale v príkazovom režime nevypíše. Ak by sme výsledok volania funkcie vypísali príkazom `print()`, dozvieme sa:

```
>>> print(index(a, 5))
None
>>>
```

hodnota `None`

Táto špeciálna hodnota je výsledkom všetkých funkcií, ktoré nevracajú žiadnu hodnotu pomocou `return`. To znamená, že každá funkcia ukončená bez `return` v skutočnosti vracia `None` ako keby posledným príkazom funkcie bol:

```
return None
```

Túto hodnotu môžeme často využívať v situáciách, keď chceme nejako oznámiť, že napr. výsledok hľadania bol neúspešný. Tak ako to bolo v prípade našej funkcie `index()`, ktorá v prípade, že sa v tabuľke hľadaná hodnota nenašla, vrátila `None`. Je zvykom takýto výsledok testovať takto:

```
vysledok = index(tab, hodnota)
if vysledok is None:
    print('nenasiel')
else:
    riadok, stlpec = vysledok
```

Teda namiesto testu `premenna == None` alebo `premenna != None` radšej používame `premenna is None` alebo `premenna is not None`.

13.1.2 Tabuľky s rôzne dlhými riadkami

Doteraz sme predpokladali, že všetky riadky dvojrozmernej štruktúry majú rovnakú dĺžku. Niekedy sa ale stretáme so situáciou, keď riadky budú rôzne dlhé. Napr.

```
>>> pt = [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1], [1, 5, 10, 10, 5, 1]]
>>> vypis(pt)
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

Tento zoznam obsahuje prvých niekoľko riadkov Pascalovho trojuholníka. Našťastie funkciu `vypis()` (obe verzie) sme napísali tak, že správne vypíšu aj zoznamy s rôzne dlhými riadkami.

Niektoré tabuľky nemusia mať takto pravidelný tvar, napr.

```
>>> delitele = [[6, 2, 3], [13, 13], [280, 2, 2, 2, 5, 7], [1]]
>>> vypis(delitele)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
6 2 3
13 13
280 2 2 2 5 7
1
```

Zoznam `delitele` má v každom riadku rozklad nejakého čísla (prvý prvok) na prvočinitele (súčin zvyšných prvkov).

Preto už pri zostavovaní funkcií musíme myslieť na to, že parametrom môže byť aj zoznam s rôznou dĺžkou riadkov. Zapíšme funkciu, ktorá nám vráti zoznam všetkých dĺžok riadkov danej dvojrozmernej štruktúry:

```
def dlzky(tab):
    vysl = []
    for riadok in tab:
        vysl.append(len(riadok))
    return vysl
```

Pre naše dva príklady zoznamov dostávame:

```
>>> dlzky(pt)
[1, 2, 3, 4, 5, 6]
>>> dlzky(delitele)
[3, 2, 6, 1]
```

Podobným spôsobom môžeme generovať nové dvojrozmerné štruktúry s rôznou dĺžkou riadkov, pre ktoré poznáme práve len tieto dĺžky:

```
def vyrob(dlzky, hodnota=0):
    vysl = []
    for dlzka in dlzky:
        vysl.append([hodnota] * dlzka)
    return vysl
```

Otestujeme:

```
>>> m1 = vyrob([3, 0, 1])
>>> m1
[[0, 0, 0], [], [0]]
>>> m2 = vyrob(dlzky(delitele), 1)
>>> vypis(m2)
1 1 1
1 1
1 1 1 1 1 1
1
```

Zamyslite sa, ako budú vyzerať tieto zoznamy:

```
>>> n = 7
>>> m3 = vyrob([n] * n)
>>> m4 = vyrob(range(n))
>>> m5 = vyrob(range(n, 0, -2))
```

13.1.3 Tabuľka farieb

Ukážme dve malé aplikácie, v ktorých vytvoríme dvojrozmerný zoznam náhodných farieb, potom ho vykreslíme do grafickej plochy ako postupnosť malých farebných štvorčekov - vznikne farebná mozaika a na záver to otestujeme klikaním myšou.

Prvý program vygeneruje dvojrozmerný zoznam náhodných farieb, vykreslí ho a uloží do textového súboru:

```
import tkinter
import random

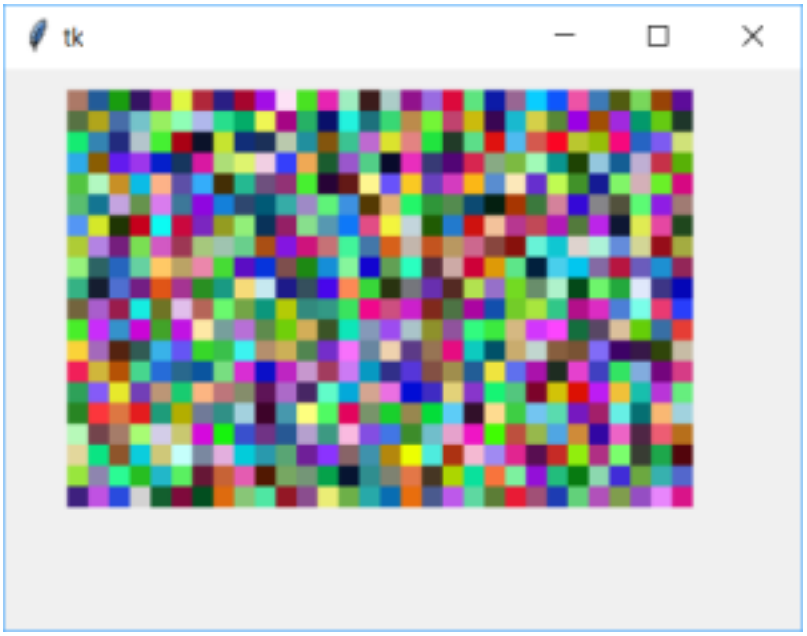
canvas = tkinter.Canvas()
canvas.pack()

tab = []
for i in range(20):
    p = []
    for j in range(30):
        p.append(f'#{random.randrange(256**3):06x}')
    tab.append(p)

d, x0, y0 = 10, 30, 10
for i in range(len(tab)):
    for j in range(len(tab[i])):
        x, y = d*j+x0, d*i+y0
        canvas.create_rectangle(x, y, x+d, y+d, fill=tab[i][j], outline='')

with open('obr.txt', 'w') as subor:
    for riadok in tab:
        print(' '.join(riadok), file=subor)
```

Vznikne približne takýto obrázok



V druhej časti programu už nebudeme generovať dvojrozmerný zoznam, ale prečítame ho z uloženého súboru. Keďže plánujeme klikaním meniť farby kliknutých štvorčekov, musíme si pamätať ich identifikačné čísla, ktoré vznikajú pri ich vykreslení pomocou `create_rectangle()` - použijeme na to pomocnú dvojrozmernú tabuľku `re` (rovnakých rozmerov ako tabuľka farieb). Na záver doplníme funkciu na zabezpečenie klikania: kliknutý štvorček sa zafarbí, napr. na bielo:

```
import tkinter

canvas = tkinter.Canvas()
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
canvas.pack()

tab = []
with open('obr.txt') as subor:
    for riadok in subor:
        tab.append(riadok.split())

# inicializuj pomocnú tabul'ku re[][] pre id nakreslených štvorčekov
re = []
for i in range(len(tab)):
    re.append([0] * len(tab[i]))
# vykresli a id. cisla uloz do zoznamu re[][]
d, x0, y0 = 10, 30, 10
for i in range(len(tab)):
    for j in range(len(tab[i])):
        x, y = d*j + x0, d*i + y0
        re[i][j] = canvas.create_rectangle(x, y, x+d, y+d, fill=tab[i][j], outline='')

def klik(event):
    stlpec, riadok = (event.x - x0) // d, (event.y - y0) // d
    if 0 <= riadok < len(tab) and 0 <= stlpec < len(tab[riadok]):
        canvas.itemconfig(re[riadok][stlpec], fill='white')
        #tab[riadok][stlpec] = 'white'

canvas.bind('<Button-1>', klik)
```

Všimnite si, ako sme počítali pozíciu kliknutého štvorčeka.

13.2 Hra LIFE

Informácie k tejto informatickej simulačnej hre nájdete na wikipedii

Pravidlá:

- v nekonečnej štvorcovej sieti žijú bunky, ktoré sa rôzne rozmnožujú, resp. umierajú
- v každom políčku siete je buď živá bunka, alebo je políčko prázdne (budeme označovať ako **1** a **0**)
- každé políčko má 8 susedov (vodorovne, zvislo aj po uhlopriečke)
- v každej generácii sa s každým jedným políčkom:
  - ak je na políčku bunka a má práve 2 alebo 3 susedov, tak táto bunka prežije aj do ďalšej generácie
  - ak je na políčku bunka a má buď 0 alebo 1 suseda, alebo viac ako 3 susedov, tak bunka na tomto políčku do ďalšej generácie neprežije (umiera)
  - ak má prázdne políčko presne na troch susediacich políčkach živé bunky, tak sa tu v ďalšej generácii narodí nová bunka

Štvorcovú sieť s `0` a `1` budeme ukladať v dvojrozmernej tabuľke veľkosti `n` x `n`. V tejto tabuľke je momentálna generácia bunkových živočíchov. Na to, aby sme vyrobili novú generáciu, si pripravíme pomocnú tabuľku rovnakej veľkosti a do nej budeme postupne zapisovať bunky novej generácie. Keď už bude celá táto pomocná tabuľka hotová, prekopírujeme ju do pôvodnej tabuľky. Dvojrozmernú tabuľku budeme vykresľovať do grafickej plochy.

```
import tkinter
import random
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
def inicializuj_siet():
    d, x0, y0 = 8, 30, 10
    re = []
    for i in range(n):
        re.append([0]*n)
        for j in range(n):
            x, y = d*j+x0, d*i+y0
            re[i][j] = canvas.create_rectangle(x, y, x+d, y+d, fill='white', outline=
↪'gray')
    return re

def nahodne():
    siet = []
    for i in range(n):
        p = []
        for j in range(n):
            p.append(random.randrange(2))
        siet.append(p)
    return siet

def kresli():
    for i in range(n):
        for j in range(n):
            farba = ['white','black'][siet[i][j]]
            canvas.itemconfig(re[i][j], fill=farba)
    canvas.update()

def pocet_susedov(rr, ss):
    pocet = 0
    for r in rr-1, rr, rr+1:
        for s in ss-1, ss, ss+1:
            if 0 <= r < n and 0 <= s < n:
                pocet += siet[r][s]
    return pocet - siet[rr][ss]

def nova():
    siet1 = []
    for i in range(n):
        siet1.append([0] * n)

    for i in range(n):
        for j in range(n):
            p = pocet_susedov(i, j)
            if p == 3 or p == 2 and siet[i][j]:
                siet1[i][j] = 1

    siet[:] = siet1
    kresli()

def rob(kolko=100):
    for i in range(kolko):
        nova()

# štart

canvas = tkinter.Canvas(width=600, height=500)
```

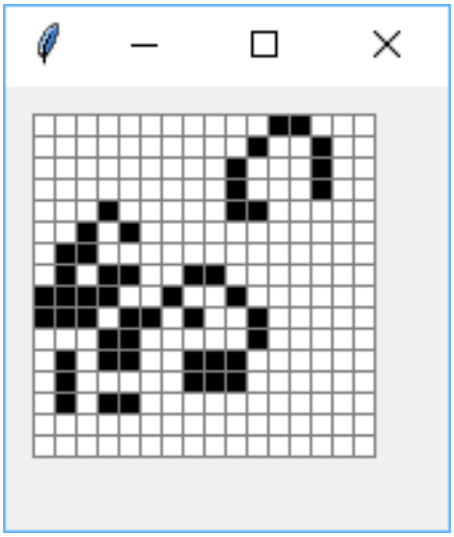
(pokračuje na ďalšej strane)

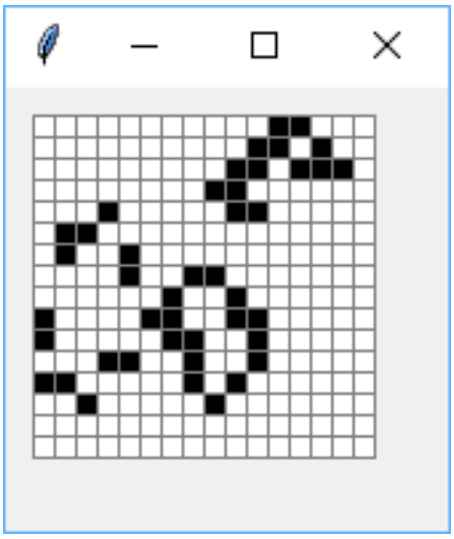
(pokračovanie z predošlej strany)

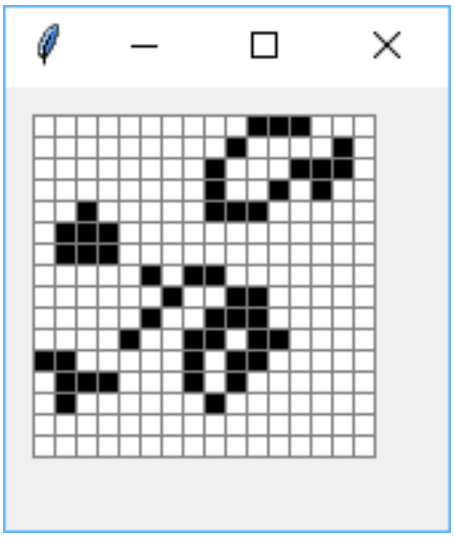
```
canvas.pack()

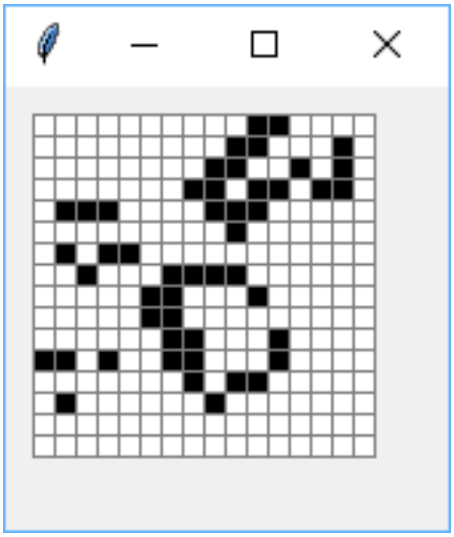
n = 50
re = inicializuj_siet()
siet = nahodne()
kresli()
rob()
```

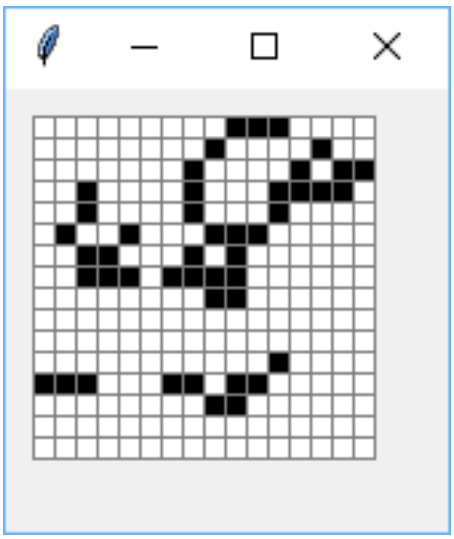
Na tejto sérii obrázkov môžete sledovať, ako sa s nejakej náhodnej pozície postupne generujú ďalšie generácie:

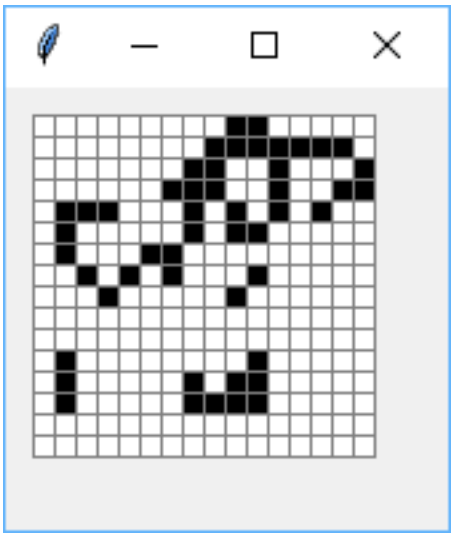


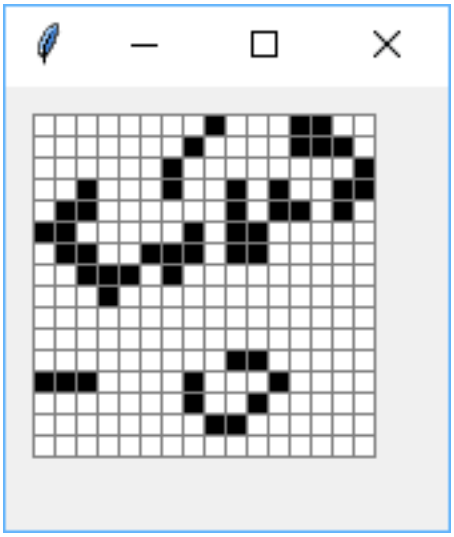


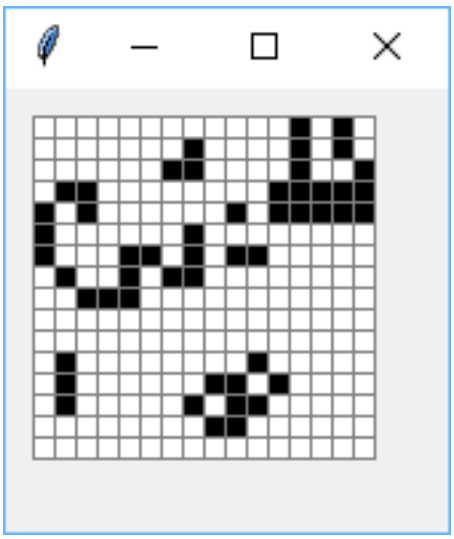








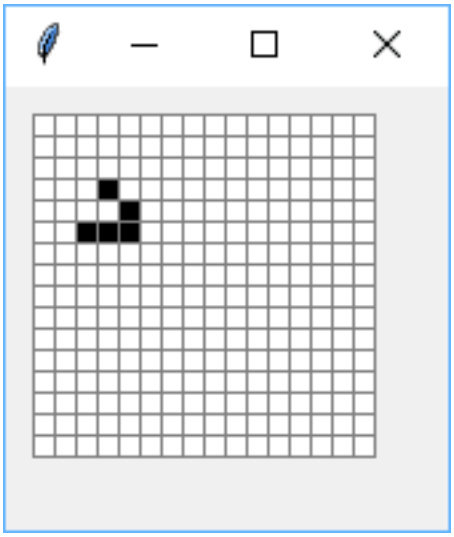


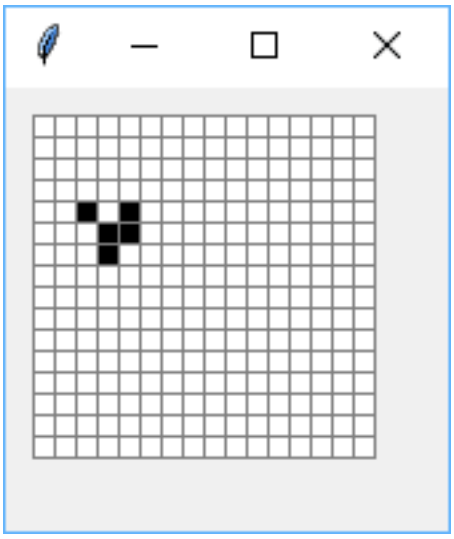


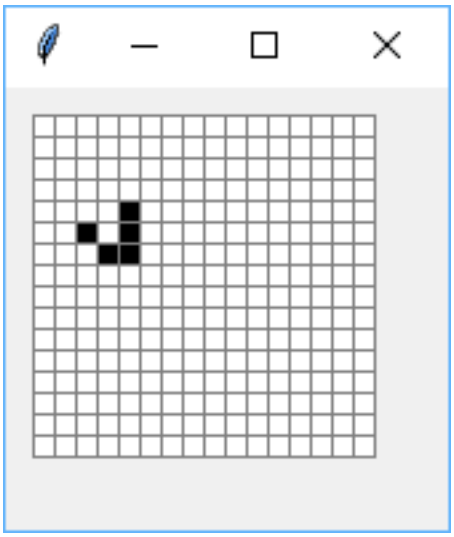
Namiesto náhodného obsahu môžeme vytvoriť prázdnu (vynulovanú) sieť, do ktorej priradíme:

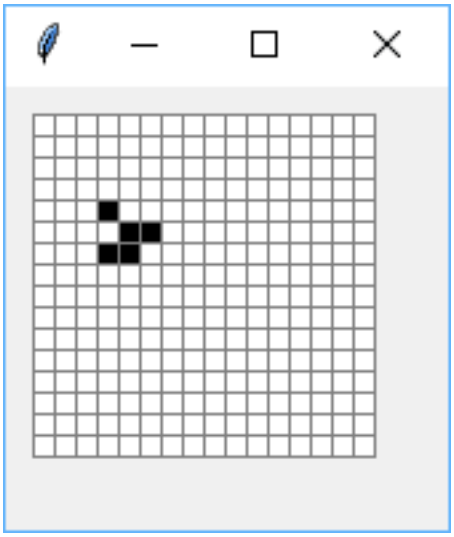
```
siet[5][2] = siet[5][3] = siet[5][4] = siet[4][4] = siet[3][3] = 1
```

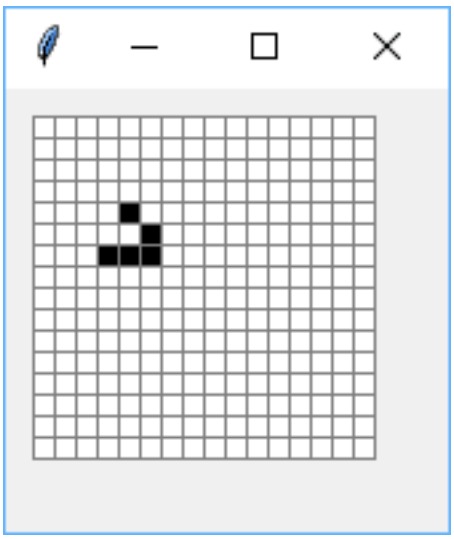
Dostávame takýto klzák (glider), ktorý sa pohybuje po ploche nejakým smerom:











Všimnite si, že po 4 generáciách má rovnaký tvar, ale je posunutý o 1 políčko dole a vpravo.

13.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Funkcia `vypis_sucty(tab)` vypíše súčty prvkov v jednotlivých riadkoch tabuľky, súčty vypisuje vedľa seba.
   - napr.

```
>>> vypis_sucty([[1, 2, 3], [4], [5, 6]])
6 4 11
```

2. Funkcia `zoznam_suctov(tab)` počíta súčty prvkov v riadkoch (podobne ako v predchádzajúcej úlohe), ale tieto súčty nevypisuje ale ukladá do výsledného zoznamu.
   - napr.

```
>>> suc = zoznam_suctov([[1, 2, 3], [4], [5, 6]])
>>> suc
[6, 4, 11]
```

3. Funkcia `pridaj_sucty(tab)` podobne ako predchádzajúce úlohy počíta súčty po riadkoch, ale ich ukladá na koniec každého riadka tabuľky.
   - napr.

```
>>> a = [[1, 2, 3], [4], [5, 6]]
>>> pridaj_sucty(a)
>>> a
[[1, 2, 3, 6], [4, 4], [5, 6, 11]]
```

4. Funkcia `preklop(matica)` vyrobí novú maticu (dvojrozmernú tabuľku), v ktorej bude pôvodná preklopená okolo hlavnej uhlopriečky. Predpokladáme, že všetky riadky majú rovnakú dĺžku.
   - napr.

```
>>> p = [[1, 2], [5, 6], [3, 4]]
>>> q = preklop(p)
>>> q
[[1, 5, 3], [2, 6, 4]]
```

5. Funkcia `preklop_sa(matica)` pracuje ako predchádzajúci príklad, ale namiesto výslednej matice (teda funkcia nič nevracia) funkcia zmení samotnú vstupnú maticu.
   - napr.

```
>>> p = [[1, 2], [5, 6], [3, 4]]
>>> preklop_sa(p)
>>> p
[[1, 5, 3], [2, 6, 4]]
```

6. Funkcia `zisti_dlzky(tab)` zistí, či sú všetky riadky vstupnej tabuľky rovnako dlhé, ak áno, funkcia vráti túto dĺžku, inak vráti `None`.
   - napr.

```
>>> p = [[1, 2], [3, 4], [5, 6]]
>>> zisti_dlzky(p)
2
>>> zisti_dlzky([[1, 2, 3]])
3
>>> zisti_dlzky([[], [1, 2, 3]])     # vráti None
>>>
```

7. Funkcia `dopln(tab)` doplní do vstupnej tabuľky do každého riadka minimálny počet `None` tak, aby mali všetky riadky rovnakú dĺžku.
   - napr.

```
>>> a = [[5, 6], [1, 2, 3], [4]]
>>> dopln(a)
>>> a
[[5, 6, None], [1, 2, 3], [4, None, None]]
```

8. Zistite, čo počíta
   - funkcia:

```
def test(mat):
    vysl, n = 0, len(mat)
    for i in range(n):
        for j in range(n):
            vysl += abs(mat[i][j] - mat[j][i])
    return vysl == 0
```

   - čo vráti:

```
>>> test([[1, 2], [1, 1]])
>>> test([[1, 2, 3], [2, 2, 1], [3, 1, 3]])
```

9. Funkcia `zisti(tab1, tab2)` zistí, či majú dve vstupné tabuľky úplne rovnaké rozmery, t. j. majú rovnaký počet rovnakodlhých riadkov.
   - napr.

```
>>> a = [[5, 6], [1, 2, 3], [4]]
>>> b = [[0, 0], [0, 0, 0], [0]]
>>> zisti(a, b)
True
>>> del b[-1][-1]
>>> zisti(a, b)
False
```

10. Funkcia `sucet(tab1, tab2)` vráti novú tabuľku, ktorá je súčtom dvoch vstupných rovnakoveľkých číselných tabuliek. Funkcia vráti takú tabuľku, v ktorej je každý prvok súčtom dvoch prvkov zo vstupných tabuliek s rovnakým indexom.
    - napr.

```
>>> a = [[5, 6], [1, 2, 3], [4]]
>>> b = [[-1, -3], [-2, 0, 1], [2]]
>>> c = sucet(a, b)
>>> c
[[4, 3], [-1, 2, 4], [6]]
```

11. Textový súbor v každom riadku obsahuje niekoľko slov, oddelených medzerou (riadok môže byť aj prázdny). Funkcia `citaj(meno_suboru)` prečíta tento súbor a vyrobí z neho dvojrozmernú tabuľku: každý riadok tabuľky zodpovedá jednému riadku súboru,

    - napr. ak súbor `text.txt`:

```
anicka dusicka
kde si bola
ked si si cizmicky
zarosila
```

    - potom

```
>>> s = citaj('text.txt')
>>> s
[['anicka', 'dusicka'], ['kde', 'si', 'bola'], ['ked', 'si', 'si', 'cizmicky
↪'], ['zarosila']]
```

12. Funkcia `zapis(tab, meno_suboru)` je opačná k predchádzajúcemu príkladu: zapíše danú dvojrozmernú tabuľku slov do súboru.

    - napr.

```
>>> s = [['anicka', 'dusicka'], ['kde', 'si', 'bola'], ['ked', 'si', 'si',
↪'cizmicky'], ['zarosila']]
>>> zapis(s, 'text1.txt')
```

      vytvorí rovnaký súbor ako bol `text.txt`

    - uvedomte si, že ak by vstupná dvojrozmerná tabuľka obsahovala čísla, táto funkcia vytvorí korektný súbor čísel, napr.

```
>>> zapis([[1, 11, 21], [345], [-5, 10]], 'cisla.txt')
```

      vytvorí súbor:

```
1 11 21
345
-5 10
```

13. Funkcia `citaj_cisla(meno_suboru)` bude podobná funkcii `citaj(meno_suboru)` z (11) úlohy, let táto predpokladá, že vstupný súbor obsahuje len celé čísla. Funkcia vráti dvojrozmernú tabuľku čísel.

    - napr. pre textový súbor z (12) úlohy:

```
>>> tab = citaj_cisla('cisla.txt')
>>> tab
[[1, 11, 21], [345], [-5, 10]]
```

14. Funkcia `prvky(tab)` z dvojrozmernej tabuľky vyrobí (funkcia vráti) jednorozmernú: všetky prvky postupne pridáva do výsledného zoznamu.

    - napr.

```
>>> a = [[5, 6], [1, 2, 3], [4]]
>>> b = prvky(a)
>>> b
[5, 6, 1, 2, 3, 4]
```

15. Funkcia `vyrob(pr, ps, hodnoty)` vyrobí dvojrozmernú tabuľku s počtom riadkov `pr` a počtom stĺpcov `ps`. Prvky zoznamu `hodnoty` postupne priradzuje po riadkoch do novovytváranej tabuľky. Ak je vstupný zoznam hodnôt kratší ako potrebujeme, začne z neho čítať od začiatku.

    - napr.

```
>>> xy = vyrob(3, 2, [3, 5, 7])
>>> xy
[[3, 5], [7, 3], [5, 7]]
>>> vyrob(3, 3, list(range(1, 20, 2)))
[[1, 3, 5], [7, 9, 11], [13, 15, 17]]
```

16. Vytvorte (napr. v notepade) textový súbor, ktorý obsahuje aspoň 5 riadkov s piatimi farbami (len mená farieb). Napíšte funkciu `kruhy(meno_suboru)`, ktorá prečíta tento súbor a farby zo súboru vykreslí ako farebné kruhy. Tieto budú vykreslené tesne vedľa seba po riadkoch. Súbor najprv prečítajte do dvojrozmernej tabuľky farieb a potom vykreslite.

    - napr. súbor môže vyzerať takto:

```
yellow yellow blus yellow yellow
yellow blue yellow blue yellow
blue yellow red yellow blue
yellow blue yellow blue yellow
yellow yellow blue yellow yellow
```

    - volanie:

```
>>> kruhy('farby.txt')
# vykreslí 25 kruhov v piatich radoch po 5
```

17. Predchádzajúci príklad upravte tak, aby ak by bol v súbore namiesto nejakej farby `None`, bude to označovať, že sa príslušný kruh vynechá (ostane po ňom prázdne miesto).

    - napr. súbor môže vyzerať aj takto:

```
yellow yellow blus yellow yellow
yellow blue None blue yellow
blue None red None blue
yellow blue None blue yellow
yellow yellow blue yellow yellow
```

    - volanie:

```
>>> kruhy('farby.txt')
# vykreslí 21 kruhov v piatich radoch po 5, 4, 3, 4, 5 kruhoch
```

18. Textový súbor v prvom riadku obsahuje dve čísla: počet riadkov a stĺpcov dvojrozmernej tabuľky. V každom ďalšom sa nachádza trojica čísel: číslo riadka, číslo stĺpca, hodnota. Funkcia `precitaj(meno_suboru)` z tohto súboru vytvorí dvojrozmernú tabuľku čísel, v ktorej budú na zadaných pozíciách dané hodnoty.

    - napr. pre súbor:

```
4 5
3 1 7
0 1 1
3 3 3
2 4 9
```

```
>>> tab = precitaj('subor.txt')
>>> tab
[[0, 1, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 9], [0, 7, 0, 3, 0]]
```

KAPITOLA 14

14. Triedy a objekty

Čo už vieme:

- poznáme základné typy: `int`, `float`, `bool`, `str`, `list`, `tuple`
- niektoré ďalšie typy sme získali z iných modulov: `tkinter.Canvas`, `turtle.Turtle`
- premenné v Pythone sú vždy referencie na príslušné hodnoty
- pre rôzne typy máme v Pythone definované:
  - **operácie**: `7 * 8 + 9`, `'a' * 8 + 'b'`, `7 * [8] + [9]`
  - **funkcie**: `len('abc')`, `sum(zoznam)`, `min(ntica)`
  - **metódy**: `'11 7 234'.split()`, `zoznam.append('novy')`, `g.create_line(1,2,3,4)`, `t.fd(100)`
- funkcia `type(hodnota)` vráti **typ** hodnoty

14.1 Vlastný typ

V Pythone sú všetky typy objektové, t.j. popisujú objekty, a takýmto typom hovoríme **trieda** (po anglicky **class**). Všetky hodnoty (teda aj premenné) sú nejakého objektového typu, teda typu trieda, hovoríme im **inštancia triedy** (namiesto *hodnota alebo premenná typu trieda*).

Zadefinujme vlastný typ, teda novú triedu:

```
class Student:
    pass
```

Trochu sa to podobá definícii funkcie bez parametrov s prázdnym telom, napr.

```
def funkcia():
    pass
```

Pomocou konštrukcie `class Student:` sme vytvorili prázdnu triedu, t.j. nový typ `Student`, ktorý zatiaľ nič nevie. Keďže je to typ, môžeme vytvoriť premennú tohto typu (teda skôr hodnotu typu `Student`, na ktorú do premennej priradíme referenciu):

```
>>> fero = Student()        # inštancia triedy
>>> type(fero)
<class '__main__.Student'>
>>> fero
<__main__.Student object at 0x022C4FF0>
```

Objektovú premennú, teda **inštanciu triedy** vytvárame zápisom `MenoTriedy()` (neskôr budú v zátvorkách nejaké parametre). V našom prípade premenná `fero` obsahuje referenciu na objekt nášho nového typu `Student`. Podobne to funguje aj s typmi, ktoré už poznáme, ale zatiaľ sme to takto často nerobili:

```
>>> i = int()
>>> type(i)
<class 'int'>
>>> zoznam = list()
>>> type(zoznam)
<class 'list'>
```

Všimnite si, že inštanciu sme tu vytvorili volaním `meno_typu()`. Všetky doterajšie štandardné typy majú svoj identifikátor zapísaný len malými písmenami: `int`, `float`, `bool`, `str`, `list`, `tuple`. Medzi pythonistami je ale dohoda, že nové typy, ktoré budeme v našich programoch definovať, budeme zapisovať s prvým písmenom veľkým. Preto sme zapísali napr. typ `Student`.

Spomeňte si, ako sme definovali korytnačku:

```
>>> import turtle
>>> t = turtle.Turtle()
```

Premenná `t` je referenciou na objekt triedy `Turtle`, ktorej definícia sa nachádza v module `turtle` (preto sme museli najprv urobiť `import turtle`, aby sme dostali prístup k obsahu tohto modulu). Už vieme, že `t` je inštanciou triedy `Turtle`.

14.1.1 Atribúty

O objektoch hovoríme, že sú to **kontajnery na dáta**. V našom prípade premenná `fero` je referenciou na prázdny kontajner. Pomocou priradenia môžeme objektu vytvárať nové súkromné premenné, tzv. **atribúty**. Takéto súkromné premenné nejakého objektu sa správajú presne rovnako ako bežné premenné, ktoré sme používali doteraz, len sa nenachádzajú v hlavnej pamäti (v globálnom mennom priestore) ale v „pamäti objektu“. Atribút vytvoríme tak, že za meno objektu `fero` zapíšeme meno tejto súkromnej premennej, pričom medzi nimi musíme zapísať bodku. Ak takýto atribút ešte neexistoval, vytvoríme ho priradením:

```
>>> fero.meno = 'Frantisek'
```

Týmto zápisom sme vytvorili novú premennú (atribút objektu) a priradili sme jej hodnotu reťazec `'Frantisek'`. Ak ale chceme zistiť, čo sa zmenilo v objekte `fero`, nestačí zapísať:

```
>>> print(fero)
<__main__.Student object at 0x022C4FF0>
```

Totiž `fero` je stále referenciou na objekt typu `Student` a Python zatiaľ netuší, čo znamená, že takýto objekt chceme nejako slušne vypísať. Musíme zadať:

```
>>> fero.meno
'Frantisek'
```

Pridajme do objektu `fero` ďalší atribút:

```
>>> fero.priezvisko = 'Fyzik'
```

Tento objekt teraz obsahuje dve súkromné premenné `meno` a `priezvisko`. Aby sme ich vedeli slušne vypísať, môžeme vytvoriť pomocnú funkciu `vypis`:

```
def vypis(st):
    print('volam sa', st.meno, st. priezvisko)
```

Funkcia má jeden parameter `st` a ona z tohto objektu (všetko v Pythone sú objekty) vyberie dve súkromné premenné (atribúty `meno` a `priezvisko`) a tieto vypíše:

```
>>> vypis(fero)
volam sa Frantisek Fyzik
```

Do tejto funkcie by sme mohli poslať ako parameter hodnotu ľubovoľného typu nielen `Student`: táto hodnota ale musí byť objektom s atribútmi `meno` a `priezvisko`, inak dostávame takúto chybu:

```
>>> i = 123
>>> vypis(i)
...
AttributeError: 'int' object has no attribute 'meno'
```

Teda chyba oznamuje, že celé čísla nemajú atribút `meno`. Vytvorme ďalšiu inštanciu triedy `Student`:

```
>>> zuzka = Student()
>>> type(zuzka)
<class '__main__.Student'>
```

Aj `zuzka` je objekt typu `Student` - je to zatiaľ prázdny kontajner atribútov. Ak zavoláme:

```
>>> vypis(zuka)
...
AttributeError: 'Student' object has no attribute 'meno'
```

dostali sme rovnakú správu, ako keď sme tam poslali celé číslo. Ak chceme, aby to fungovalo aj s týmto novým objektom, musíme tieto dve súkromné premenné vytvoriť, napr.

```
>>> zuzka.meno = 'Zuzana'
>>> zuzka.priezvisko = 'Matikova'
>>> vypis(zuzka)
volam sa Zuzana Matikova
```

14.1.2 Objekty sú meniteľné (mutable)

Atribúty objektu sú súkromné premenné, ktoré sa správajú presne rovnako ako „obyčajné“ premenné. Premenným môžeme meniť obsah, napr.

```
>>> fero.meno = 'Ferdinand'
>>> vypis(fero)
volam sa Ferdinand Fyzik
```

Premenná `fero` stále obsahuje referenciu na rovnaký objekt (kontajner), len sa trochu zmenil jeden z atribútov. Takejto vlastnosti objektov sme doteraz hovorili **meniteľné** (**mutable**):

- napr. zoznamy sú **mutable**, lebo niektoré operácie zmenia obsah zoznamu ale nie referenciu na objekt (`zoznam.append('abc')` pridá do zoznamu nový prvok)
- ak dve premenné referencujú ten istý objekt (napr. priradili sme `zoznam2 = zoznam`), tak takáto **mutable** zmena jedného z nich zmení obe premenné
- väčšina doterajších typov `int`, `float`, `bool`, `str` a `tuple` sú **immutable** teda nemenné, s nimi tento problém nenastáva
- nami definované nové typy (triedy) sú vo všeobecnosti **mutable** - ak by sme chceli vytvoriť novú **immutable** triedu, treba ju definovať veľmi špeciálnym spôsobom a tiež s ňou úptpm treba pracovať veľmi opatrne

Ukážme si to na príklade:

```
>>> mato = fero
>>> vypis(mato)
volam sa Ferdinand Fyzik
```

Objekt `mato` nie je novým objektom ale referenciou na ten istý objekt ako `fero`. Zmenou niektorého atribútu sa zmení obsah oboch premenných:

```
>>> mato.meno = 'Martin'
>>> vypis(mato)
volam sa Martin Fyzik
>>> vypis(fero)
volam sa Martin Fyzik
```

Preto si treba dávať naozaj veľký pozor na priradenie **mutable** objektov.

14.1.3 Funkcie

Už sme definovali funkciu `vypis()`, ktorá vypisovala dva konkrétne atribúty parametra (objektu). Táto funkcia nemodifikovala žiaden atribút, ani žiadnu doteraz existujúcu premennú. Zapíšme funkciu `urob()`, ktorá dostane dva znakové reťazce a vytvorí z nich nový objekt typy `Student`, pričom tieto dva reťazce budú obsahom dvoch atribútov `meno` a `priezvisko`:

```
def urob(m, p):
    novy = Student()
    novy.meno = m
    novy.priezvisko = p
    return novy
```

Pomocou tejto funkcie vieme definovať nové objekty typu `Student`, ktoré už budú mať vytvorené oba atribúty `meno` a `priezvisko`, napr.

```
>>> fero = urob('Ferdinand', 'Fyzik')
>>> zuzka = urob('Zuzana', 'Matikova')
>>> mato = urob('Martin', 'Fyzik')
>>> vypis(fero)
volam sa Ferdinand Fyzik
>>> vypis(zuzka)
volam sa Zuzana Matikova
>>> vypis(mato)
volam sa Martin Fyzik
```

Ani funkcia `urob()` nemodifikuje žiaden svoj parameter ani iné premenné, len vytvára novú inštanciu (a modifikuje atribúty svojej lokálnej premennej) a tú potom vracia ako výsledok funkcie. Funkcie, ktoré majú túto vlastnosť (nič nemodifikujú, len vytvárajú niečo nové) nazývame **pravé funkcie** (po anglicky **pure function**). Pravou funkciou bude aj funkcia `kopia`, ktorá na základe jedného objektu vyrobí nový, ktorý je jeho kópiou. Predpokladáme, že robíme kópiu inštancie `Student`, ktorá má atribúty `meno` a `priezvisko`:

```
def kopia(iny):
    novy = Student()
    novy.meno = iny.meno
    novy.priezvisko = iny.priezvisko
    return novy
```

Ak má `zuzka` sestru Evu, môžeme ju vytvoriť takto:

```
>>> evka = kopia(zuzka)
>>> evka.meno = 'Eva'
>>> vypis(evka)
volam sa Eva Matikova
>>> vypis(zuzka)
volam sa Zuzana Matikova
```

Obe inštancie sú teraz dva rôzne kontajnery, teda obe majú svoje vlastné súkromné premenné `meno` a `priezvisko`.

Okrem pravých funkcií existujú tzv. **modifikátory** (po anglicky **modifier**). Je to funkcia, ktorá niečo zmení, najčastejšie atribút nejakého objektu. Funkcia `nastav_hoby()` nastaví danému objektu atribút `hoby` a vypíše o tom text:

```
def nastav_hoby(st, text):
    st.hoby = text
    print(st.meno, st. priezvisko, 'ma hoby', st.hoby)
```

```
>>> nastav_hoby(fero, 'gitara')
Ferdinand Fyzik ma hoby gitara
>>> nastav_hoby(evka, 'cyklistika')
Eva Matikova ma hoby cyklistika
```

Oba objekty `fero` aj `evka` majú teraz už 3 atribúty, pričom `mato` a `zuzka` majú len po dvoch.

Keďže vlastnosť funkcie **modifikátor** je pre všetky **mutable** objekty veľmi dôležitá, pri písaní nových funkcií si vždy musíme uvedomiť, či je to modifikátor alebo pravá funkcia a často túto informáciu zapisujeme aj do dokumentácie.

Všimnite si, že

```
def zmen(st):
    meno = st.meno
    meno = meno[::-1]
    print(meno)
```

nie je modifikátor, lebo hoci funkcia mení obsah premennej `meno`, táto je len lokálnou premennou funkcie `zmen` a nemá žiaden vplyv ani na parameter `st` ani na žiadnu inú premennú.

14.2 Metódy

Všetky doteraz vytvárané funkcie dostávali ako jeden z parametrov objekt typu `Student` (inštanciu triedy) alebo takýto objekt vracali ako výsledok funkcie. Lenže v objektovom programovaní platí:

- **objekt** je kontajner údajov, ktoré sú vlastne súkromnými premennými objektu (**atribúty**)

- **trieda** je kontajner funkcií, ktoré vedia pracovať s objektmi (aj týmto funkciám niekedy hovoríme **atribúty**, ale častejšie ich voláme **metódy**)

Takže funkcie nemusíme vytvárať tak ako doteraz globálne v hlavnom mennom priestore (tzv. `__main__`), ale priamo ich môžeme definovať v triede. Pripomeňme si, ako vyzerá definícia triedy:

```
class Student:
    pass
```

Príkaz `pass` sme tu uviedli preto, lebo sme chceli vytvoriť prázdne telo triedy (podobne ako pre `def` ale aj `while` a `if`). Namiesto `pass` ale môžeme zadefinovať funkcie, ktoré sa stanú súkromné pre túto triedu. Takýmto funkciám hovoríme **metóda**. Platí tu ale jedno veľmi dôležité pravidlo: prvý parameter metódy musí byť premenná, v ktorej metóda dostane inštanciu tejto triedy a s ňou sa bude ďalej pracovať. Zapíšme funkcie `vypis()` a `nastav_hoby()` ako **metódy** (t.j. funkcie definované vo vnútri triedy, teda sú to atribúty triedy):

```
class Student:

    def vypis(self):
        print('volam sa', self.meno, self. priezvisko)

    def nastav_hoby(self, text):
        self.hoby = text
        print(self.meno, self. priezvisko, 'ma hoby', self.hoby)
```

Čo sa zmenilo:

- obe funkcie sú **vnorené** do definície triedy a preto sú odsunuté vpravo
- obom funkciám sme zmenili prvý parameter `st` na `self` - toto sme robiť nemuseli, ale je to **dohoda** medzi pythonistami, že prvý parameter metódy sa bude vždy volať **self** bez ohľadu pre akú triedu túto metódu definujeme (obe funkcie by fungovali korektne aj bez premenovania tohto parametra)

Keďže `vypis()` už teraz nie je globálna funkcia ale metóda, nemôžeme ju volať tak ako doteraz `vypis(fero)`, ale k menu uvedieme aj meno kontajnera (meno triedy), kde sa táto funkcia nachádza, teda `Student.vypis(fero)`:

```
>>> fero = urob('Ferdinand', 'Fyzik')
>>> zuzka = urob('Zuzana', 'Matikova')
>>> mato = urob('Martin', 'Fyzik')
>>> Student.vypis(fero)
volam sa Ferdinand Fyzik
>>> Student.vypis(zuzka)
volam sa Zuzana Matikova
>>> Student.vypis(mato)
volam sa Martin Fyzik
```

Takýto spôsob volania metód však nie je bežný:

```
Trieda.metoda(instancia, parametre)
```

Namiesto neho sa používa trochu pozmenený, pričom sa vynecháva meno triedy. Budeme používať takéto poradie zápisu volania metódy:

```
instancia.metoda(parametre)
```

čo znamená:

```
>>> fero.vypis()
volam sa Ferdinand Fyzik
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> zuzka.vypis()
volam sa Zuzana Matikova
>>> mato.vypis()
volam sa Martin Fyzik
```

Podobne zapíšeme priradenie hoby dvom študentom. Namiesto zápisu:

```
>>> Student.nastav_hoby(fero, 'gitara')
Ferdinand Fyzik ma hoby gitara
>>> Student.nastav_hoby(evka, 'cyklistika')
Eva Matikova ma hoby cyklistika
```

si radšej zvykneme na:

```
>>> fero.nastav_hoby('gitara')
Ferdinand Fyzik ma hoby gitara
>>> evka.nastav_hoby('cyklistika')
Eva Matikova ma hoby cyklistika
```

S takýmto zápisom volania funkcií (teda metód) sme sa už stretli skôr, ale asi to bola pre nás doteraz veľká záhada, napr.

```
>>> zoznam = [2, 5, 7]
>>> zoznam.append(11)
>>> zoznam.pop(0)
2
>>> a = '12-34-56'.split('-')
```

znamená:

```
>>> zoznam = [2, 5, 7]
>>> list.append(zoznam, 11)
>>> list.pop(zoznam, 0)
2
>>> a = str.split('12-34-56', '-')
```

Teda `append()` je metóda triedy `list` (pythonovský zoznam), ktorá má dva parametre: `self` (samotný zoznam, ktorý sa bude modifikovať) a `hodnota`, ktorá sa bude do zoznamu pridávať na jeho koniec. Táto metóda je zrejme definovaná niekde v triede `list` a samotná jej deklarácia by mohla vyzerať nejako takto:

```
class list:
    ...
    def append(self, hodnota):
        ...
```

14.2.1 Magické metódy

Do novo vytvárenej triedy môžeme pridávať ľubovoľné množstvo metód (súkromných funkcií), pričom majú jediné obmedzenie: prvý parameter by mal mať meno `self`. Už vieme, že takúto metódu môžeme volať nielen:

```
Trieda.metoda(instancia, parametre)
```

ale radšej ako:

```
instancia.metoda(parametre)
```

Okrem tohto štandardného mechanizmu volania metód, existuje ešte niekoľko **špeciálnych metód**, pre ktoré má Python aj iné využitie. Pre tieto špeciálne (tzv. **magické**) metódy má Python aj špeciálne pravidlá. My sa s niektorými z týchto magických metód budeme zoznamovať priebežne na rôznych prednáškach, podľa toho, ako ich budeme potrebovať.

Magické metódy majú definíciu úplne rovnakú ako bežné metódy. Python ich rozpozná podľa ich mena: ich meno začína aj končí dvojicou podčiarkovníkov ___. Pre Python je tento znak bežná súčasť identifikátorov, ale využíva ich aj na tento špeciálny účel. Ako prvé sa zoznámime s magickou metódou `__init__()`, ktorá je jednou z najužitočnejších a najčastejšie definovaných magických metód.

metóda `__init__()`

Je magická metóda, ktorá slúži na **inicializovanie atribútov** daného objektu. Má tvar:

```
def __init__(self, parametre):
    ...
```

Metóda môže (ale nemusí) mať ďalšie parametre za `self`. Metóda nič nevracia, ale najčastejšie obsahuje len niekoľko priradení.

Túto metódu (ak existuje) Python zavolá, v tom momente, keď sa vytvára nová inštancia.

Keď zapíšeme `instancia = Trieda(parametre)`, tak Python postupne:

1. vytvorí nový objekt typu `Trieda` - zatiaľ je to **prázdny kontajner**
   - vytvorí sa pomocná referencia na tento nový `objekt`
2. ak existuje metóda `__init__()`, zavolá ju s príslušnými parametrami: `Trieda.__init__(objekt, parametre)`
   - keď Python zavolá našu metódu `__init__()`, znamená to, že samotný objekt už existuje (dostaneme ho v parametri `self`), ale zatiaľ je to prázdny kontajner bez atribútov premenných - tie vzniknú až priraďovacím príkazom do týchto atribútov
3. do premennej `instancia` priradí práve vytvorený `objekt`
   - v tejto premennej už máme hotový objekt, ktorý prešiel inicializáciou v `__init__()`

Hovoríme, že metóda `__init__()` **inicializuje** objekt (niekedy sa hovorí aj, že **konštruuje**, resp. že je to **konštruktor**). Najčastejšie sa v tejto metóde priradzujú hodnoty do atribútov, napr.

```
class Student:

    def __init__(self, meno, priezvisko, hoby=''):
        self.meno = meno
        self.priezvisko = priezvisko
        self.hoby = hoby

    def vypis(self):
        print('volam sa', self.meno, self. priezvisko)

    def nastav_hoby(self, text):
        self.hoby = text
        print(self.meno, self. priezvisko, 'ma hoby', self.hoby)
```

Vďaka tomu už nepotrebujeme funkciu `urob()`, ale inštanciu aj s atribútmi vyrobíme pomocou konštruktora:

```
>>> fero = Student('Ferdinand', 'Fyzik')
>>> fero.nastav_hoby('gitara')
Ferdinand Fyzik ma hoby gitara
>>> evka = Student('Eva', 'Matikova, 'cyklistika')
```

14.2.2 Štandardná funkcia dir()

Funkcia `dir()` vráti postupnosť (zoznam) všetkých atribútov triedy alebo inštancie. Pozrime najprv nejakú prázdnu triedu:

```
>>> class Test: pass

>>> dir(Test)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__']
```

Vidíme, že napriek tomu, že sme zatiaľ pre túto triedu nič nedefinovali, v triede sa nachádza veľa rôznych atribútov. Jednu z nich už poznáme: `__init__` je magická metóda. Vždy keď zadefinujeme nový atribút alebo metódu, objaví sa aj v tomto zozname `dir()`:

```
>>> t = Test()
>>> t.x = 100
>>> t.y = 200
>>> dir(t)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'x', 'y']
```

Na konci tohto zoznamu sú dva nové atribúty `x` a `y`.

14.2.3 Príklad s grafikou

Zadefinujeme novú triedu `Kruh(r, x, y)`, ktorá bude mať 3 atribúty pre kruh v grafickej ploche: polomer a súradnice stredu:

```
class Kruh:

    def __init__(self, r, x, y):
        self.r = r
        self.x = x
        self.y = y
```

Teraz, keď máme triedu, môžeme vytvárať nové inštancie (objekty), napr.

```
>>> a = Kruh(70, 200, 100)
>>> b = Kruh(10, 180, 80)
>>> c = Kruh(10, 220, 80)
```

Tieto objekty sú zatiaľ len „kontajnery“ pre atribúty.

Do takejto triedy môžeme v inicializácii pridať aj ďalšie atribúty, ktoré nie sú v parametroch inicializácie napr.

```
class Kruh:

    def __init__(self, r, x, y):
        self.r = r
        self.x = x
        self.y = y
        self.farba = 'blue'
```

Znamená, že vždy keď vytvoríme nový objekt, okrem 3 apribútov `r`, `x` a `y` sa vytvorí aj atribút `farba` s hodnotou `'blue'`.

Teraz zadefinujeme pomocnú funkciu `kresli_kruh(kruh)`, ktorá očakáva parameter typu `Kruh` a tento kruh potom nakreslí do grafickej plochy (predpokladáme, že grafická plocha je už vytvorená a prístupná pomocou premennej `canvas`):

```
def kresli_kruh(kruh):
    canvas.create_oval(kruh.x-kruh.r, kruh.y-kruh.r, kruh.x+kruh.r, kruh.y+kruh.r,␣
↪fill=kruh.farba)
```

Otestujeme:

```
import tkinter

a = Kruh(70, 200, 100)
a.farba = 'yellow'
b = Kruh(10, 180, 80)
c = Kruh(10, 220, 80)

canvas = tkinter.Canvas()
canvas.pack()
kresli_kruh(a)
kresli_kruh(b)
kresli_kruh(c)
```

Takéto objekty kruhy môžeme uložiť aj do zoznamu a potom aj ich nakreslenie môže vyzerať takto:

```
zoznam = [a, b, c]
for k in zoznam:
    kresli_kruh(k)
```

Ak teraz zadáme:

```
>>> zoznam
[<__main__.Kruh object>, <__main__.Kruh object>, <__main__.Kruh object>]
```

vidíme len to, že zoznam obsahuje nejaké tri objekty typu `Kruh`. Zadefinujme preto metódu `výpis()`, ktorá vypíše detaily konkrétneho objektu. Do triedy `Kruh` dopíšeme túto metódu a do konštruktora pridáme aj štvrtý parameter `farba`. Tiež funkciu `kresli_kruh()` prepíšeme na metódu `kresli()`:

```
import tkinter

class Kruh:

    def __init__(self, r, x, y, farba='blue'):
        self.r = r
        self.x = x
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        self.y = y
        self.farba = farba

    def vypis(self):
        return f'Kruh({self.r}, {self.x}, {self.y}, {self.farba!r})'

    def kresli(self):
        canvas.create_oval(self.x-self.r, self.y-self.r, self.x+self.r, self.y+self.r,
↪ fill=self.farba)

canvas = tkinter.Canvas()
canvas.pack()

a = Kruh(70, 200, 100, 'yellow')
b = Kruh(10, 180, 80)
c = Kruh(10, 220, 80)

zoznam = [a, b, c]
for k in zoznam:
    k.kresli()
for k in zoznam:
    k.vypis()
```

Tento program teraz vypíše:

```
Kruh(70, 200, 100, 'yellow')
Kruh(10, 180, 80, 'blue')
Kruh(10, 220, 80, 'blue')
```

Pozrime ešte, čo nám vrátia volania funkcie `dir()` pre triedu `Kruh` aj inštanciu `a`:

```
>>> dir(Kruh)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'kresli', 'vypis']
>>> dir(a)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'farba', 'kresli', 'vypis', 'r', 'x', 'y']
```

Všimnite si, že v triede `Kruh` pribudli dva atribúty, ktoré nie sú magickými metódami: `kresli` a `vypis`, v inštancii `a` okrem týchto metód pribudli 4 atribúty: `farba`, `r`, `x` a `y`.

14.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Zadefinujte triedu `Cas`, ktorá bude mať dva celočíselné atribúty `hodiny` a `minuty`. Aj inicializácia (metóda `__init__()`) bude mať dva parametre `hodiny` a `minuty`. Metóda `vypis()` vypíše nastavený čas v tvare `čas je 9:17`.

   - trieda `Cas`:

   ```
   class Cas:
       ...
   ```

   - otestujte, napr.

   ```
   >>> c = Cas(9, 17)
   >>> c.vypis()
   čas je 9:17
   >>> d = Cas(10, 5)
   >>> d.vypis()
   čas je 10:05
   ```

   - zamyslite sa, čo sa stane pre volanie `Cas.vypis(c)`, čím sa to líši od `c.vypis()`

2. Do triedy `Cas` z úlohy (1) pridajte metódu `str()`, ktorá nič nevypisuje, ale namiesto toho vráti (`return`) znakový reťazec s hodinami a minútami v tvare `'9:17'`

   - napr.

   ```
   >>> c = Cas(9, 1)
   >>> print('teraz je', c.str())
   teraz je 9:01
   ```

3. Do triedy `Cas` z (2) úlohy dopíšte metódu `pridaj()`, ktorá bude mať 2 parametre `hodiny` a `minuty`. Metóda pridá k uloženému času zadané hodiny a minúty.

   - napr.

   ```
   >>> cas = Cas(17, 40)
   >>> print('teraz je', cas.str())
   teraz je 17:40
   >>> cas.pridaj(1, 35)
   >>> print('neskôr', cas.str())
   neskôr 19:15
   ```

4. Máme danú inštanciu `c` triedy `Cas` (z (3) úlohy). Vytvorte novú inštanciu, ktorá je od času `c` posunutá o zadaný počet hodín a minút. Využite metódu `pridaj()`. Nemusíte na to vytvárať ani novú metódu ani funkciu,

   - napr.

   ```
   >>> c = Cas(17, 40)
   >>> d = ...           # vyrob kópiu času c
   >>> ...               # posuň d o 2 hodiny a 55 minút
   >>> print(c.str())
   17:40
   >>> print(d.str())
   20:35
   ```

5. Vytvorte pätnásť-prvkový zoznam inštancií triedy `Cas`, v ktorom prvý prvok reprezentuje `8:10` a každý ďalší je posunutý o 50 minút. Ďalšie časy v zozname vytvárajte v cykle, využite metódu `pridaj()`.

   - napr.

```
>>> zoznam = [Cas(8, 10), ...]    # vyrob 15-prvkové zoznam časov
>>> for c in zoznam:
        print(c.str(), end=' ')
8:10 9:00 9:50 ... 19:50
```

6. Zapíšte definíciu triedy `Zlomok`, ktorá v inicializácii vytvorí dva atribúty `citatel` a `menovatel`. Metóda `vypis()` pomocou `print()` tento zlomok vypíše v tvare `zlomok je 3/8`.
   - napr.

```
>>> z1 = Zlomok(3, 8)
>>> z2 = Zlomok(2, 4)
>>> z1.vypis()
zlomok je 3/8
>>> z2.vypis()
zlomok je 2/4
```

7. Pridajte do triedy `Zlomok` z úlohy (6) dve metódy:
   - `str()` vráti (nič nevypisuje) reťazec v tvare `3/8`
   - `float()` vráti (nič nevypisuje) desatinné číslo, ktoré reprezentuje daný zlomok
   - napr.

```
>>> z = Zlomok(3, 8)
>>> print('z je', z.str())
z je 3/8
>>> print('z je', z.float())
z je 0.375
>>> w = Zlomok(2, 4)
>>> print('w je', w.str())
w je 2/4
>>> print('w je', w.float())
w je 0.5
```

8. Zadefinujte triedu `Body`, ktorá si bude uchovávať momentálny stav bodov (napr. získané body v nejakej hre). Trieda bude mať tieto metódy:
   - `pridaj()` k momentálnemu stavu pridá 1 bod
   - `uber()` od momentálneho stavu odoberie 1 bod
   - `kolko()` vráti celé číslo - momentálny bodový stav
   - napr.

```
>>> b = Body()
>>> for i in range(10):
        b.pridaj()
>>> b.uber()
>>> b.uber()
>>> print('body =', b.kolko())
body = 8
```

9. Zadefinujte triedu `Subor` s metódami:
   - `__init__(meno_suboru)` vytvorí nový prázdny súbor
   - `pripis(text)` na koniec súboru pridá nový riadok so zadaným textom; použite `open(..., 'a')`
   - `vypis()` vypíše momentálny obsah súboru

- napr.

```
>>> s = Subor('text.txt')
>>> s.pripis('prvy riadok')
>>> s.pripis('druhy riadok')
>>> s.vypis()
prvy riadok
druhy riadok
>>> s.pripis('posledny riadok')
>>> s.vypis()
prvy riadok
druhy riadok
posledny riadok
```

10. Zadefinujte triedu `Zoznam`, pomocou ktorej si budeme vedieť udržiavať zoznam svojich záväzkov (sľubov, povinností, ...). Tieto budete uchovávať v atribúte `zoznam` typu `list`. Trieda obsahuje tieto metódy:
    - `pridaj(prvok)`, ak sa tam takýto záväzok ešte nenachádza, pridá ho na koniec
    - `vyhod(prvok)`, ak sa tam takýto záväzok nachádzal, vyhodí ho
    - `je_v_zozname(prvok)` vráti `True` alebo `False` podľa toho, či sa tam tento záväzok nachádza
    - `vypis()` vypíše všetky záväzky v tvare `zoznam: záväzok, záväzok, záväzok`
    - napr.

```
moj = Zoznam()
moj.pridaj('upratat')
moj.pridaj('behat')
moj.pridaj('ucit sa')
if moj.je_v_zozname('behat'):
    print('musis behat')
else:
    print('nebehaj')
moj.pridaj('upratat')
moj.vyhod('spievat')
moj.vypis()
```

    vypíše

```
musis behat
zoznam: upratat, behat, ucit sa
```

11. Zadefinujte triedu `TelefonnyZoznam`, ktorá bude udržiavať informácie o telefónnych číslach (ako zoznam `list` dvojíc `tuple`). Trieda bude mať tieto metódy:
    - `pridaj(meno, telefon)` pridá do zoznamu dvojicu `(meno, telefon)`; ak takéto `meno` v zozname už existovalo, nepridáva novú dvojicu, ale nahradí len telefónne číslo
    - `vypis()` vypíše celý telefónny zoznam
    - napr.

```
tz = TelefonnyZoznam()
tz.pridaj('Jana', '0901020304')
tz.pridaj('Juro', '0911111111')
tz.pridaj('Jozo', '0212345678')
tz.pridaj('Jana', '0999020304')
tz.vypis()
```

vypíše

```
Jana 0999020304
Juro 0911111111
Jozo 0212345678
```

12. Zadefinujte triedu `Okno`, ktorá otvorí grafické okno a do stredu vypíše zadaný text. Výška otvoreného okna nech je 100. Vypísaný text nech je v strede okna fontom veľkosti 50. Inicializácia (metóda `__init__()`) vytvorí nový `canvas` (výšky 100) a do jeho stredu vypíše zadaný text. Zrejme si v svojich atribútoch zapamätá `canvas` aj identifikačný kód pre `create_text()`. Ďalšie dve metódy menia vypísaný text:
    - `zmen(text)` zmení vypísaný text (zrejme na to použijete `itemconfig()`)
    - `farba(farba)` zmení farbu vypísaného textu (zrejme na to použijete `itemconfig()`)
    - napr.

```
import tkinter
okno = Okno('ahoj')
okno.farba('red')
okno.zmen('Python')
```

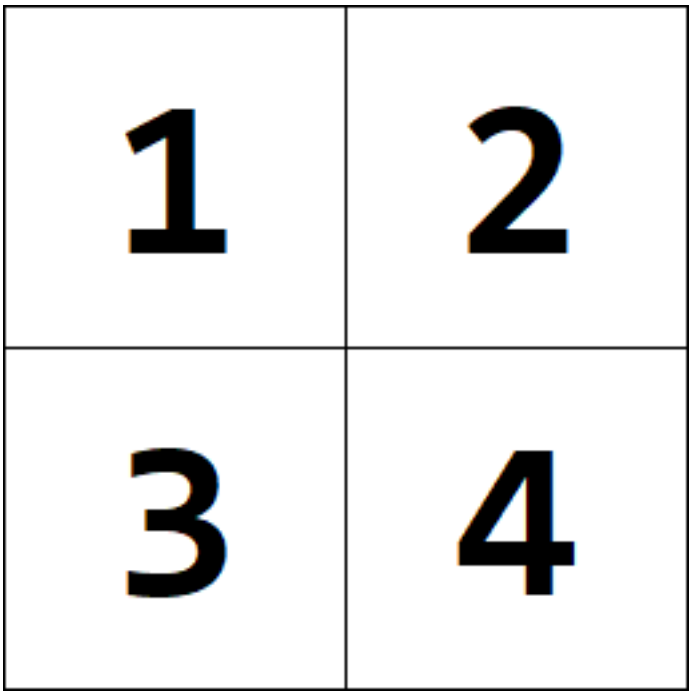
    - vyskúšajte vytvoriť dve inštancie `Okno`

14.4 6. Domáce zadanie

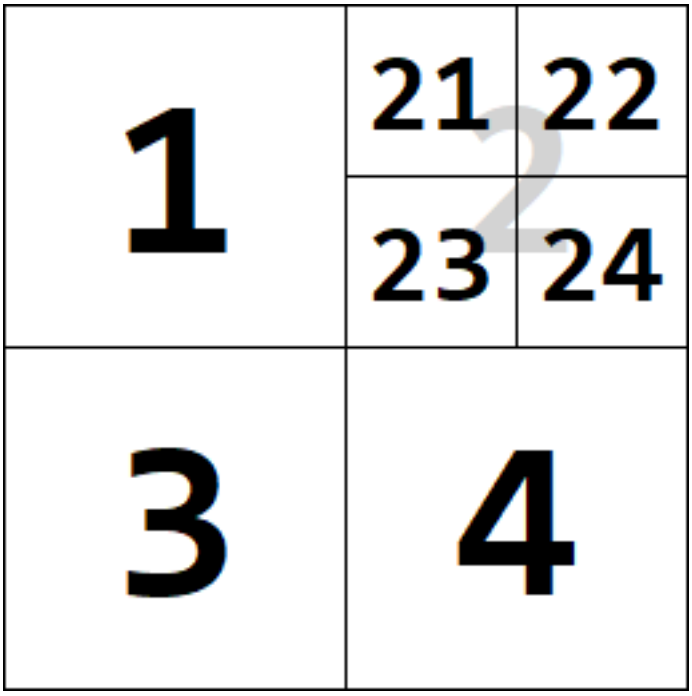
L.I.S.T.

- riešenie odovzdávajte na úlohový server https://list.fmph.uniba.sk/

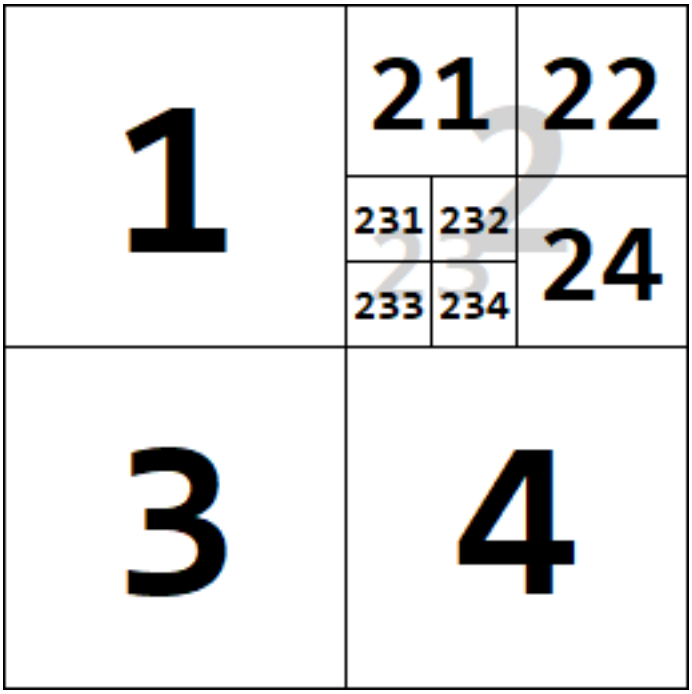
Veľký štvorec môžeme rozdeliť na 4 kvadranty takto:



Úplne rovnako môžeme každý kvadrant rozdeliť na menšie kvadranty, napr. 2. kvadrant takto:



Vidíte, že tieto menšie kvadranty v `2` sú očíslované ako `21`, `22`, `23`, `24`. Ľubovoľný kvadrant môžeme rozdeliť na 4 menšie. Napr. rozdelením kvadrantu `23`, dostávame:



Takýmto delením môžeme ísť do nejakej danej hĺbky `n`, kým neprídeme na elementárny štvorček, ktorý sa už deliť nedá. Napr. hĺbka `n=1` znamená, že plocha sa skladá z 2x2 elementárnych štvorčekov a deliť ju môžeme maximálne raz na 4 základné kvadranty. Hĺbka `n=4` označuje 16x16 elementárnych štvorčekov, ktorú môžeme deliť maximálne 4-krát. Potom napr. kvadrant `2` je veľký 8x8 elementárnych štvorčekov, kvadrant `23` zaberá 4x4 štvorčekov, kvadrant `234` zaberá 2x2 štvorčeky a zrejme `2343` už len jeden. Teda pre dané `n` poradové číslo kvadrantu má zmysel s

maximálnym počtom cifier `n`. Čím má číselné označenie kvadrantu menej cifier, tým je kvadrant väčší (napr. 0-ciferné číslo kvadrantu označuje celý štvorec).

Napíšte pythonovský modul, ktorý bude obsahovať jedinú triedu `Stvorce` a žiadne iné globálne premenné:

```
class Stvorce:
    def __init__(self, n):
        ...

    def urob(self, index):
        ...

    def vypis(self):
        ...
```

Metódy majú fungovať takto:

- inicializácia `__init__(self, n)` vyhradí takú dvojrozmernú tabuľku, aby sa dali deliť kvadranty do hĺbky `n`; každý elementárny štvorček môže obsahovať `0` alebo `1`, pri inicializácii budú všade `0`
- metóda `urob(self, index)` dostáva číslo kvadrantu ako znakový reťazec (môže byť aj prázdny) a pre zadaný kvadrant vyznačí všetky elementárne štvorčeky tak, že hodnoty `0` nahradí `1` a hodnoty `1` nahradí `0`
- metóda `vypis(self)` vypíše (pomocou `print()`) momentálny obsah dvojrozmernej tabuľky, pričom namiesto `0` použije znak `'-'` a namiesto `1` znak `'X'`

Napr.

```
>>> stv = Stvorce(2)
>>> stv.urob('23')
>>> stv.vypis()
----
--X-
----
----
>>> stv.urob('2')
>>> stv.urob('3')
>>> stv.vypis()
--XX
---X
XX--
XX--
```

Pre inšpiráciu môžete otestovať tieto postupnosti indexov:

```
['1', '11', '131', '1314', '143', '12', '1233', '1234', '1243', '1244', '23', '234',
'2133', '2134', '2143', '2144', '24', '242', '2423', '31', '313', '3132', '32', '324',
'3232', '3411', '3412', '3421', '3422', '4', '44', '424', '4241', '413', '4141', '43',
'4311', '4312', '4321', '4322']
```

```
----------------
----------------
----------------
----XXXXXXXX----
--XXXXXXXXXXXX--
-XXXXXXXXXXXXXX-
XXXX--XXXX--XXXX
XXXX--XXXX--XXXX
XXXXXXXXXXXXXXXX
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
XXXXXXXXXXXXXXXX
-XXXX------XXXX-
--XXXX----XXXX--
----XXXXXXXX----
----------------
----------------
----------------
```

```
['14', '141','1412','1431','124', '1241', '2', '21', '22', '24', '213', '2113',
'2143', '2324', '2433', '3', '31', '3142', '3211', '343', '3441', '3443', '33',
'332', '3321', '41', '4213', '4231', '4232', '43', '434', '441', '4423']
```

```
----------------
--------X-------
-------XXX------
------XXXXX-----
-----XXXXXXX----
------XXXXX-----
-----XXXXXXX----
----XXXXXXXXX---
-----XXXXXXX----
----XXXXXXXXX---
---XXXXXXXXXXX--
----XXXXXXXX----
---XXXXXXXXXXX--
--XXXXXXXXXXXXX-
-------XXX------
-------XXX------
```

```
['2', '23224', '2411', '24123', '24131', '24132', '2422', '24223', '2444', '24441',
'212', '21412', '21421', '21422', '211', '2113', '21132', '21143', '22', '2233',
'2234', '22342', '14', '14111', '14113', '1433', '14332', '132', '1322', '13223',
'134', '1344', '13441', '1312', '13142', '13144', '13322', '13324', '1334', '1134',
'11341', '11333', '11334', '11433', '1224', '12243', '12344', '124', '1241', '12421',
'12423', '12431', '3112', '31123', '31111', '31112', '31211', '32122', '322', '3223',
'32213', '3224', '32242', '411', '412', '4124', '41241', '41234', '4211', '42121',
'42122', '42211', '4131', '413114', '41321']
```

```
--------------------------------
--------------------------------
--------------XXX---------------
---------------XXXX-------------
---------------XXXXXX-----------
---------------XXXXXXXXX--------
---X---------XXXXXXXXXXXXXX-----
XXXXX------XXXXXXXXXXXXXXXXX----
--XXXX---XXXXXXXXXXXXXXX--XXXX--
--XXXXX--XXXXXXXXXXXXXX----XXXX-
---XXXXXXXXXXXXXXXXXXXX--XXXXXX
---XXXXXXXXXXXXXXXXXXXXXXXXXXXX
---XXXXXXXXXXXXXXXXXXXXXXXXXXXX
---XXXXXXXXXXXXXXXXXXXXXXXXXXXX
--XXXXX--XXXXXXXXXXXXXXXXXXXXX-
--XXXX----XXXXXXXXXXXXXXXXXXX--
XXXXX------XXXXXXXXXXXXXXXXX---
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
---X---------XXXXXXXXXXXX------
--------------XXXXXXXX---------
----------------XXXXX-----------
----------------XXX-------------
----------------XX--------------
--------------------------------
--------------------------------
--------------------------------
--------------------------------
--------------------------------
--------------------------------
--------------------------------
--------------------------------
--------------------------------
--------------------------------
```

Váš odovzdaný program s menom `riesenie6.py` musí začínať tromi riadkami komentárov:

```
# 6. zadanie: stvorce
# autor: Janko Hraško
# datum: 22.12.2017
```

Projekt `riesenie6.py` odovzdávajte na úlohový server https://list.fmph.uniba.sk/ najneskôr do 23:00 **5. januára**, kde ho môžete nechať otestovať. Testovač bude spúšťať vašu funkciu s rôznymi vstupmi. Odovzdať projekt aj ho testovať môžete ľubovoľný počet krát. Môžete zaň získať **10 bodov**.

KAPITOLA 15

15. Triedy a metódy

Zhrňme, čo už vieme o triedach a objektoch

Novú triedu najčastejšie definujeme takto:

```
class Meno_triedy:

    def __init__(self, parametre):
        ...

    def metoda1(self, parametre):
        ...

    def metoda2(self, parametre):
        ...

    def metoda3(self, parametre):
        ...
```

Objekty (inštancie triedy) vytvárame a používame takto:

```
>>> premenna = Meno_triedy(...)    # skonštruovanie objektu
>>> premenna.atribut = hodnota     # vytvorenie nového atribútu/zmena hodnoty atribútu
>>> premenna.metoda(parametre)     # zavolanie metódy
```

Metóda musí mať pri definovaní prvý parameter `self`, ktorý reprezentuje samotnú inštanciu: Python sem pri volaní metódy automaticky dosadí samotný objekt, nasledovné dvojice volaní robia to isté:

```
>>> premenna.metoda(parametre)
>>> Meno_triedy.metoda(premenna, parametre)

>>> 'mama ma emu'.count('ma')
3
>>> str.count('mama ma emu', 'ma')
3
```

Metódy sú súkromné funkcie definované v triede. Pozrime sa na príklad z cvičení, v ktorom sme definovali triedu `Cas` s dvoma atribútmi `hodiny` a `minuty`:

```
class Cas:

    def __init__(self, hodiny, minuty):
        self.hodiny = hodiny
        self.minuty = minuty

    def vypis(self):
        print(f'cas je {self.hodiny}:{self.minuty:02}')

    def str(self):
        return f'{self.hodiny}:{self.minuty:02}'
```

Spomínali sme, že v Pythone všetky funkcie (a teda aj metódy) môžeme rozdeliť do dvoch základných typov:

- **modifikátor** - mení niektorú hodnotu atribútu (alebo aj viacerých), napr.

```
class Cas:
    ...
    def pridaj(self, hodiny, minuty):
        self.hodiny += hodiny + (self.minuty + minuty) // 60
        self.minuty = (self.minuty + minuty) % 60
```

- **pravá funkcia** - nemení atribúty a nemá ani žiadne vedľajšie účinky (nemení globálne premenné); najčastejšie vráti nejakú hodnotu - môže to byť aj nový objekt, napr.

```
class Cas:
    ...
    def kopia(self):
        return Cas(self.hodiny, self.minuty)
```

- uvedomte si, že nemeniteľné typy (**immutable**) obsahujú iba pravé funkcie (zrejme okrem inicializácie `__init__()`, ktorá bude veľmi často modifikátor)

15.1 Magická metóda __str__

Niektoré metódy sú špeciálne (tzv. **magické**) - nimi definujeme špeciálne správanie (nová trieda sa lepšie prispôsobí filozofii Pythonu). Zatiaľ sme sa zoznámili len s jednou magickou metódou `__init__()`, ktorá inicializuje atribúty - automaticky sa vyvolá hneď po vytvorení (skonštruovaní) objektu.

Ďalšou veľmi často definovanou magickou metódou je `__str__()`. Jej využitie ukážeme na príklade triedy `Cas`, ktorú sme doteraz používali takto:

```
>>> c = Cas(9, 17)
>>> c.vypis()
cas je 9:17
>>> print('teraz je', c.str())
teraz je 9:17
```

Už sme si trochu zvykli, že keď priamo vypisujeme inštanciu `c`, dostaneme:

```
>>> c
<__main__.Cas object at 0x03220F10>
>>> print(c)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
<__main__.Cas object at 0x03220F10>
>>> str(c)
'<__main__.Cas object at 0x03220F10>'
```

Čo je ale veľmi nezaujímavú informácia. Asi by bolo veľmi užitočné, keby Python nejako pochopil, že reťazcová reprezentácia nášho objektu by mohla byť výsledkom nejakej našej metódy a automaticky by ju použil, napr. v `print()` alebo aj v `str()`.

Naozaj presne toto funguje: keď zadefinujeme magickú metódu `__str__()` a Python na niektorých miestach bude potrebovať reťazcovú reprezentáciu objektu, tak zavolá túto našu vlastnú metódu. Zrejme výsledkom tejto metódy **musí byť** znakový reťazec, inak by Python protestoval. Opravme triedu `Cas` so zadefinovaním magickej metódy:

```
class Cas:

    def __init__(self, hodiny, minuty):
        self.hodiny = hodiny
        self.minuty = minuty

    def __str__(self):
        return f'{self.hodiny}:{self.minuty:02}'

    def vypis(self):
        print('cas je', self)
```

Všimnite si, ako sme mohli vďaka tomuto opraviť metódu `vypis()`: keďže `print()` po vypísaní `'cas je'` chce vypísať aj `self` a zrejme táto inštancia nie je znakový reťazec (je to predsa inštancia triedy `Cas`), Python pohľadá, či táto trieda nemá náhodou definovanú metódu `__str__()` a keďže áno, zavolá ju a má reťazcovú reprezentáciu premennej `self`. Otestujeme:

```
>>> c = Cas(9, 17)
>>> c.vypis()
cas je 9:17
>>> print('teraz je', c.__str__())
teraz je 9:17
>>> print('teraz je', c)
teraz je 9:17
>>> c
<__main__.Cas object at 0x03220F10>
>>> print(c)
9:17
>>> str(c)
'9:17'
```

Všimnite si dosť škaredý zápis volania `c.__str__()`. Vôbec to nemusíme takto zapisovať: v príkaze `print()` stačí písať len meno premennej `c`, prípadne, ak potrebujeme naozaj reťazec, krajší je zápis `str(c)`. Mohli by ste si predstaviť štandardnú funkciu `str()`, že je definovaná takto:

```
def str(objekt=''):
    return objekt.__str__()
```

Pričom predpokladáme, že v každej triede Pythonu je zadefinovaná magická metóda `__str__()`.

15.1.1 Volanie metódy z inej metódy

Zatiaľ sme v našich jednoduchých príkladoch, v ktorých sme definovali nejaké triedy, nepotrebovali riešiť situácie, v ktorých v jednej metóde voláme nejakú inú metódu tej istej triedy. Doplňme do triedy `Cas` aj metódu `pridaj()`:

```
class Cas:

    def __init__(self, hodiny, minuty):
        self.hodiny = hodiny
        self.minuty = minuty

    def __str__(self):
        return f'{self.hodiny}:{self.minuty:02}'

    def vypis(self):
        print('cas je', self)

    def kopia(self):
        return Cas(self.hodiny, self.minuty)

    def pridaj(self, hodiny, minuty):
        self.hodiny += hodiny + (self.minuty + minuty) // 60
        self.minuty = (self.minuty + minuty) % 60
```

Pridajme teraz ďalšiu metódu `kopia_a_pridaj()`, ktorá vyrobí kópiu objektu a zároveň v tejto kópii posunie hodiny aj minúty:

```
class Cas:
    ...
    def kopia_a_pridaj(self, hodiny, minuty):
        novy = Cas(self.hodiny, self.minuty)
        novy.pridaj(hodiny, minuty)
        return novy
```

Vidíme, že novovytvorený objekt `novy` zavolal svoju metódu `pridaj()`, preto sme to museli zapísať `novy.pridaj(...)`. Prvý riadok tejto metódy je priradenie `novy = Cas(self.hodiny, self.minuty)`, ktoré vytvorí kópiu objektu `self`. Ale na toto už máme hotovú metódu `kopia()`, takže môžeme to zapísať aj takto:

```
class Cas:
    ...
    def kopia_a_pridaj(self, hodiny, minuty):
        novy = self.kopia()
        novy.pridaj(hodiny, minuty)
        return novy
```

Tu vidíme, že keď potrebujeme, aby objekt `self` zavolal niektorú svoju metódu, musíme pred meno metódy pripísať `self` aj s bodkou, tak ako je to v tejto metóde, teda `self.kopia()`. Uvedomte si, že bez tohto `self.` by toto označovalo volanie obyčajnej funkcie (nie metódy), ktorá je buď globálna alebo niekde lokálne zadefinovaná.

Ďalej uvedieme niekoľko príkladov, v ktorých sa stretneme s doteraz vysvetľovanými pojmami.

15.1.2 Príklad s nemeniteľnou triedou čas

Vylepšíme triedu `Cas`: bude mať 3 atribúty: `hod`, `min`, `sek` (pre hodiny, minúty, sekundy). Všetky metódy vytvoríme ako **pravé funkcie**, vďaka čomu sa bude táto trieda správať ako **immutable** (nemeniteľný typ):

```
class Cas:

    def __init__(self, hodiny=0, minuty=0, sekundy=0):
        self.hod = hodiny
        self.min = minuty
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        self.sek = sekundy

    def __str__(self):
        return f'{self.hod}:{self.min:02}:{self.sek:02}'

    def sucet(self, iny):
        return Cas(self.hod+iny.hod, self.min+iny.min, self.sek+iny.sek)

    def vacsi(self, iny):
        return (self.hod > iny.hod or
                self.hod == iny.hod and self.min > iny.min or
                self.hod == iny.hod and self.min == iny.min and self.sek > iny.sek)
```

Zadefinovali sme dve nové metódy `sucet()` a `vacsi()`, ktoré buď vytvoria novú inštanciu alebo zistia, či je čas väčší ako nejaký iný.

Otestujme:

```
cas1 = Cas(10, 22, 30)
cas2 = Cas(10, 8)
print('cas1 =', cas1)
print('cas2 =', cas2)
print('sucet =', cas1.sucet(cas2))
print('cas1 > cas2 =', cas1.vacsi(cas2))
print('cas2 > cas1 =', cas2.vacsi(cas1))
```

Výpis:

```
cas1 = 10:22:30
cas2 = 10:08:00
sucet = 20:30:30
cas1 > cas2 = True
cas2 > cas1 = False
```

Vidíme, že metóda `vacsi()`, ktorá porovnáva dva časy, je dosť prekomplikovaná, lebo treba porovnávať tri atribúty v jednom aj druhom objekte.

Pomocná metóda

Predchádzajúce riešenie má viac problémov:

- pomocou metódy `sucet()` môžeme vytvoriť čas, v ktorej minúty alebo sekundy majú hodnotu väčšiu ako 59
- dva časy sa porovnávajú dosť komplikovane

Vytvorme pomocnú funkciu (teda metódu), ktorá z daného času vypočíta celkový počet sekúnd. Zároveň opravíme aj inicializáciu `__init__()`:

```
class Cas:

    def __init__(self, hodiny=0, minuty=0, sekundy=0):
        cas = abs(3600*hodiny + 60*minuty + sekundy)
        self.hod = cas // 3600
        self.min = cas // 60 % 60
        self.sek = cas % 60
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def __str__(self):
        return f'{self.hod}:{self.min:02}:{self.sek:02}'

    def sucet(self, iny):
        return Cas(self.hod+iny.hod, self.min+iny.min, self.sek+iny.sek)

    def rozdiel(self, iny):
        return Cas(sekundy = self.pocet_sekund() - iny.pocet_sekund())

    def pocet_sekund(self):
        return 3600 * self.hod + 60 * self.min + self.sek

    def vacsi(self, iny):
        return self.pocet_sekund() > iny.pocet_sekund()

cas1 = Cas(10, 22, 30)
cas2 = Cas(9, 58, 45)
print('cas1 =', cas1)
print('cas2 =', cas2)
print('sucet =', cas1.sucet(cas2))
print('cas1 > cas2 =', cas1.vacsi(cas2))
print('cas2 > cas1 =', cas2.vacsi(cas1))
print('cas1 - cas2 =', cas1.rozdiel(cas2))
print('cas2 - cas1 =', cas2.rozdiel(cas1))
```

Pomocnú funkciu `pocet_sekund()` sme využili nielen v porovnávaní dvoch časov (metóda `vacsi()`) ale aj v novej metóde `rozdiel()`.

Celá trieda sa dá ešte viac zjednodušiť, ak by samotný objekt nemal 3 atribúty `hod`, `min` a `sek`, ale len jeden atribút `sek` pre celkový počet sekúnd. Vďaka tomu by sme nemuseli pri každej operácii čas prepočítavať na sekundy: len pri výpise by sme museli sekundy previesť na hodiny a minúty. Napr.

```
class Cas:

    def __init__(self, hodiny=0, minuty=0, sekundy=0):
        self.sek = abs(3600*hodiny + 60*minuty + sekundy)

    def __str__(self):
        return f'{self.sek//3600}:{self.sek//60%60:02}:{self.sek%60:02}'

    def sucet(self, iny):
        return Cas(sekundy=self.sek+iny.sek)

    def rozdiel(self, iny):
        return Cas(sekundy=self.sek-iny.sek)

    def vacsi(self, iny):
        return self.sek > iny.sek
```

Ak budeme teraz potrebovať vytvoriť zoznam časov, pričom prvý z nich je `8:10` a každý ďalší je o 50 minút posunutý, môžeme to zapísať napr. takto:

```
zoznam = [Cas(8, 10)]

for i in range(14):
    zoznam.append(zoznam[-1].sucet(Cas(0, 50)))
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
for cas in zoznam:
    print(cas, end=' ')
```

Zápis `zoznam[-1].sucet(Cas(0, 50))` označuje, že k času, ktorý je momentálne posledným prvkom v zozname pripočítame 50 minút (teda čas, ktorý je 0 hodín a 50 minút). Ak by sme vedeli zabezpečiť sčitovanie časov rovnakým zápisom ako je napr. sčitovanie čísel alebo zreťazovanie reťazcov, tento zápis by vyzeral `zoznam[-1] + Cas(0, 50)`, čo už vyzerá zaujímavo, ale žiaľ nefunguje.

15.2 Triedne a inštančné atribúty

Už vieme, že

- triedy sú kontajnery na súkromné funkcie (metódy)
- inštancie sú kontajnery na súkromné premenné (atribúty)

Napr.

```
>>> class Test: pass
>>> t.x = 100                # nový atribút v inštancii
>>> t.y = 200
```

Lenže atribúty ako premenné môžeme definovať aj v triede, vtedy sú to tzv. **triedne atribúty** (atribúty na úrovni inštancií sú **inštančné atribúty**). Ak teda definujeme triedny atribút:

```
>>> Test.z = 300             # nový atribút v triede
```

tak tento atribút automaticky získavajú (vidia) aj všetky inštancie tejto triedy (tak ako všetky inštancie vidia všetky metódy triedy):

```
>>> print(t.x, t.y, t.z)
100 200 300
```

Aj novovytvorená inštancia získava (teda vidí) tento triedny atribút:

```
>>> t2 = Test()
>>> t2.z
300
```

Lenže tento atribút sa zatiaľ nachádza ina v kontajneri triedy `Test` a v kontajneroch inštancií atribúty s takýmto menom nie sú. Inštancie tento triedny atribút vidia (môžu ho čítať), ale tento sa v ich kontajneri nenachádza.

Ak ho chceme mať ako súkromnú premennú v inštancii (inštančný atribút), musíme mu **v inštancii** priradiť hodnotu:

```
>>> t2.z = 400
>>> Test.z
300
>>> t.z
300
>>> t2.z
400
```

Kým do inštancie nepriradíme tento atribút, inštancia „vidí“ hodnotu triedy, keď už vyrobíme vlastný atribút, tak vidí túto hodnotu. Uvedomte si, že momentálne existuje **triedny atribút** `Test.z` a s rovnakým menom aj inštančný atribút `t2.z`. Inštancia `t2` teraz po zapísaní `t2.z` vidí už len tento svoj súkromný atribút.

Triedne atribúty môžeme vytvoriť už pri definovaní triedy, napr.

```
class Test:
    z = 300

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f'test {self.x},{self.y},{self.z}'
```

```
>>> t1 = Test(100, 200)
>>> print(t1)
test 100 200 300
>>> t2 = Test(10, 20)
>>> t2.z = 30
>>> print(t2)
test 10 20 30
```

Triedny atribút `z` má stále hodnotu 300, hoci inštancia `t2` má už svoju vlastnú verziu inštančného atribútu `z` s hodnotou 30 a preto pri výpise `t2.z` vidí len svoj atribút `z`. Keď zmeníme obsah triedneho atribútu, dostaneme:

```
>>> Test.z = 9
>>> print(t1)
test 100 200 9
>>> print(t2)
test 10 20 30
```

Ukážme si takéto využitie triedneho atribútu:

```
import tkinter
import random

class Bodka:

    def __init__(self, canvas, x, y):
        self.id = canvas.create_oval(x-5, y-5, x+5, y+5)
        self.canvas = canvas

    def prefarbi(self):
        if random.randrange(2):
            farba = 'red'
        else:
            farba = 'blue'
        self.canvas.itemconfig(self.id, fill=farba)

canvas = tkinter.Canvas()
canvas.pack()
bodky = []
for i in range(100):
    bodky.append(Bodka(canvas, random.randint(10, 300), random.randint(10, 250)))
for b in bodky:
    b.prefarbi()
```

V programe sme vytvorili 100-prvkový zoznam bodiek (inštancií triedy `Bodka`), tieto sa už pri inicializácii nakreslili ako malé nezafarbené krúžky. Na záver sme všetky bodky náhodne prefarbili na modro alebo červeno.

Keďže sme v metódach triedy nechceli pracovať s globálnou premennou `canvas`, poslali sme `canvas` ako parameter do inicializácie. Tu sa `canvas` zapamätal ako atribút každej jednej inštancie.

Ak by sme teraz dostali úlohu na záver vypísať počet modrých a červených, zdá sa, že bez globálnej premennej to bude veľmi ťažké.

Tu nám pomôžu triedne atribúty:

- `canvas` nemusíme posielať zvlášť každému objektu (takto v každom objekte vzniká inštančný atribút `canvas`, pričom všetky objekty triedy `Bodka` majú rovnakú hodnotu tohto atribútu), môžeme vytvoriť jediný triedny atribút, ktorý budú vidieť všetky inštancie
- pridáme ďalšie dva triedne atribúty pre počítanie počtu modrých a červených, pričom v metóde `prefarbi()` budeme tieto dve počítadla zvyšovať

Dostávame takúto verziu programu:

```
import tkinter
import random

class Bodka:
    canvas = None
    pocet_modrych = pocet_cervenych = 0

    def __init__(self, x, y):
        self.id = self.canvas.create_oval(x-5, y-5, x+5, y+5)

    def prefarbi(self):
        if random.randrange(2):
            farba = 'red'
            Bodka.pocet_cervenych += 1
        else:
            farba = 'blue'
            Bodka.pocet_modrych += 1
        self.canvas.itemconfig(self.id, fill=farba)

Bodka.canvas = tkinter.Canvas()
Bodka.canvas.pack()
bodky = []
for i in range(100):
    bodky.append(Bodka(random.randint(10, 300), random.randint(10, 250)))
for b in bodky:
    b.prefarbi()
print('pocet modrych =', Bodka.pocet_modrych)
print('pocet cervenych =', Bodka.pocet_cervenych)
```

Zamyslite sa nad tým čo sa stane, keď v metóde `prefarbi()` zmeníme `Bodka.pocet_cervenych += 1` na `self.pocet_cervenych += 1`.

15.2.1 Príklad s grafickými objektmi

Postupne zadefinujeme niekoľko tried, ktoré pomocou `tkinter` pracujú s rôznymi objektmi v grafickej ploche.

Objekt Kruh

Zadefinujeme:

```
import tkinter

class Kruh:
    def __init__(self, x, y, r, farba='red'):
        self.x = x
        self.y = y
        self.r = r
        self.farba = farba
        self.canvas.create_oval(
            self.x - self.r, self.y - self.r,
            self.x + self.r, self.y + self.r,
            fill=self.farba)

Kruh.canvas = tkinter.Canvas(bg='white')
Kruh.canvas.pack()
k1 = Kruh(50, 50, 30, 'blue')
k2 = Kruh(150, 100, 80)
```

Aby sme mohli nakreslený objekt posunúť alebo zmeniť jeho veľkosť alebo farbu, musíme si zapamätať jeho identifikačné číslo, ktoré vráti funkcia `create_oval()`. Využijeme už známy mechanizmus metód objektu `canvas`, ktoré menia už nakreslený útvar:

- `canvas.move(id, dx, dy)` - posúva ľubovoľný útvar
- `canvas.itemconfig(id, nastavenie=hodnota, ...)` - zmení ľubovoľné nastavenie (napr. farbu, hrúbku, ...)
- `canvas.coords(id, x1, y1, x2, y2, ...)` - zmení súradnice útvaru

Zapíšme novú verziu triedy `Kruh`:

```
import tkinter

class Kruh:
    canvas = None

    def __init__(self, x, y, r, farba='red'):
        self.x = x
        self.y = y
        self.r = r
        self.farba = farba
        self.id = self.canvas.create_oval(
            self.x - self.r, self.y - self.r,
            self.x + self.r, self.y + self.r,
            fill=self.farba)

    def posun(self, dx=0, dy=0):
        self.x += dx
        self.y += dy
        self.canvas.move(self.id, dx, dy)

    def zmen(self, r):
        self.r = r
        self.canvas.coords(self.id,
            self.x - self.r, self.y - self.r,
            self.x + self.r, self.y + self.r)

    def prefarbi(self, farba):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        self.farba = farba
        self.canvas.itemconfig(self.id, fill=farba)

Kruh.canvas = tkinter.Canvas(bg='white')
Kruh.canvas.pack()
k1 = Kruh(50, 50, 30, 'blue')
k2 = Kruh(150, 100, 80)

k1.posun(30,10)
k2.zmen(50)
k1.prefarbi('green')
```

Na začiatok definície triedy `Kruh` sme pridali vytvorenie triedneho atribútu `canvas` zatiaľ s hodnotou `None`. Robiť sme to nemuseli - funguje to rovnako dobre aj bez tohto, ale čitateľ nášho programu bude vidieť, že na tomto mieste počítame s triednym atribútom.

Trieda Obdlznik

Skopírujeme triedu `Kruh` a zmeníme na `Obdlznik`:

```
import tkinter

class Obdlznik:
    canvas = None

    def __init__(self, x, y, sirka, vyska, farba='red'):
        self.x = x
        self.y = y
        self.sirka = sirka
        self.vyska = vyska
        self.farba = farba
        self.id = self.canvas.create_rectangle(
            self.x, self.y,
            self.x + self.sirka, self.y + self.vyska,
            fill=self.farba)

    def posun(self, dx=0, dy=0):
        self.x += dx
        self.y += dy
        self.canvas.move(self.id, dx, dy)

    def zmen(self, sirka, vyska):
        self.sirka = sirka
        self.vyska = vyska
        self.canvas.coords(self.id,
            self.x, self.y,
            self.x + self.sirka, self.y + self.vyska)

    def prefarbi(self, farba):
        self.farba = farba
        self.canvas.itemconfig(self.id, fill=farba)

Obdlznik.canvas = tkinter.Canvas(bg='white')
Obdlznik.canvas.pack()
r1 = Obdlznik(50, 50, 50, 30, 'blue')
r2 = Obdlznik(150, 100, 80, 80)
```

Trieda Skupina

Vyrobíme triedu `Skupina`, pomocou ktorej budeme ukladať rôzne útvary do jednej štruktúry:

```
import tkinter

class Skupina:
    def __init__(self):
        self.zoznam = []

    def pridaj(self, utvar):
        self.zoznam.append(utvar)

canvas = tkinter.Canvas(bg='white')
canvas.pack()
Kruh.canvas = Obdlznik.canvas = canvas

sk = Skupina()
sk.pridaj(Kruh(50, 50, 30, 'blue'))
sk.pridaj(Obdlznik(100, 20, 100, 50))
sk.zoznam[0].prefarbi('green')
sk.zoznam[1].posun(50)
```

Vidíme, ako môžeme meniť už nakreslené útvary.

Ak budeme potrebovať meniť viac útvarov, použijeme cyklus:

```
for i in range(len(sk.zoznam)):
    sk.zoznam[i].prefarbi('orange')
```

alebo krajšie:

```
for utvar in sk.zoznam:
    utvar.posun(dy=15)
```

Do triedy `Skupina` môžeme doplniť metódy, ktoré pracujú so všetkými útvarmi v skupine, napr.

```
class Skupina:
    ...

    def prefarbi(self, farba):
        for utvar in self.zoznam:
            utvar.prefarbi(farba)

    def posun(self, dx=0, dy=0):
        for utvar in self.zoznam:
            utvar.posun(dx, dy)
```

Môžeme navrhnúť metódy, ktoré nebudú pracovať so všetkými útvarmi, ale len s útvarmi nejakého konkrétneho typu (napr. len s kruhmi). Preto do tried `Kruh` aj `Obdlznik` doplníme ďalší atribút:

```
class Kruh:
    canvas = None
    typ = 'kruh'

    def __init__(self, x, y, r, farba='red'):
        ...
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
class Obdlznik:
    canvas = None
    typ = 'obdlznik'

    def __init__(self, x, y, sirka, vyska, farba='red'):
        ...

class Skupina:
    ...
    def posun_typ(self, typ, dx=0, dy=0):
        for utvar in self.zoznam:
            if utvar.typ == typ:
                utvar.posun(dx, dy)

    def prefarbi_typ(self, typ, farba):
        for utvar in self.zoznam:
            if utvar.typ == typ:
                utvar.prefarbi(farba)
```

Môžeme vygenerovať skupinu 20 náhodných útvarov - kruhov a obdĺžnikov:

```
import tkinter
import random

canvas = Kruh.canvas = Obdlznik.canvas = tkinter.Canvas(bg='white')
canvas.pack()

sk = Skupina()

for i in range(20):
    if random.randrange(2) == 0:
        sk.pridaj(Kruh(random.randint(50, 200), random.randint(50, 200), 30, 'blue'))
    else:
        sk.pridaj(Obdlznik(random.randint(50, 200), random.randint(50, 200), 40, 40))

sk.prefarbi_typ('kruh', 'yellow')
sk.posun_typ('obdlznik', -10, -25)
```

Metóda __str__()

Do oboch tried `Kruh` aj `Obdlznik` pridáme magickú metódu `__str__()` a vďaka tomu môžeme veľmi elegantne vypísať všetky útvary v skupine:

```
class Kruh:
    ...
    def __str__(self):
        return f'Kruh({self.x},{self.y},{self.r},{self.farba!r})'

class Obdlznik:
    ...
    def __str__(self):
        return f'Obdlznik({self.x},{self.y},{self.sirka},{self.vyska},{self.farba!r})'

...
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
for utvar in sk.zoznam:
    print(utvar)
```

a dostávame niečo takéto:

```
Obdlznik(185,50,40,40,'red')
Kruh(95,115,30,'blue')
Obdlznik(63,173,40,40,'red')
Kruh(138,176,30,'blue')
Obdlznik(92,50,40,40,'red')
Obdlznik(180,80,40,40,'red')
...
```

15.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Zadefinujeme triedu, pomocou ktorej budeme vedieť reprezentovať obdĺžniky. Pri obdĺžnikoch nás budú zaujímať len veľkosti strán a na základe toho budeme vedieť vypočítať obsah aj obvod.

   - dopíšte všetky metódy:

   ```
   class Obdlznik:

       def __init__(self, a, b):
           # inicializuje
           ...

       def __str__(self):
           # vráti ret'azec v tvare 'Obdlznik(100, 70)'
           ...

       def obsah(self):
           # vráti obsah
           ...

       def obvod(self):
           # vráti obvod
           ...

       def zmen_velkost(self, pomer):
           # vynásobí obe vel'kosti pomerom
           ...

       def kopia(self):
           # vyrobí kópiu samého seba
           ...
   ```

   - otestujte, napr.

```
>>> obd1 = Obdlznik(20, 7)
>>> print('obvod =', obd1.obvod())
obvod = 54
>>> print(obd1)
Obdlznik(20, 7)
>>> obd2 = obd1.kopia()
>>> obd2.zmen_velkost(2)
>>> print(obd2)
Obdlznik(40, 14)
```

2. Zoberte riešenie (11) úlohy z predchádzajúceho cvičenia: trieda `TelefonnyZoznam`, ktorá udržiava informácie o telefónnych číslach (ako zoznam `list` dvojíc `tuple`). Trieda mala tieto metódy:

   - `pridaj(meno, telefon)` pridá do zoznamu dvojicu `(meno, telefon)`; ak takéto `meno` v zozname už existovalo, nepridáva novú dvojicu, ale nahradí len telefónne číslo
   - `vypis()` vypíše celý telefónny zoznam
   - malo by fungovať:

```
tz = TelefonnyZoznam()
tz.pridaj('Jana', '0901020304')
tz.pridaj('Juro', '0911111111')
tz.pridaj('Jozo', '0212345678')
tz.pridaj('Jana', '0999020304')
tz.vypis()
```

   - doplňte túto triedu tak, aby fungovalo aj zapisovanie aj čítanie s textovým súborom:
     - metóda `__init__(meno_suboru)` si zapamätá meno súboru (nič ešte nezapisuje ani nečíta)
     - metóda `zapis()` obsah telefónneho zoznamu zapíše do súboru: v každom riadku bude jedna dvojica meno a číslo, pričom budú navzájom oddelené znakom `';'`
     - metóda `citaj()` prečíta zadaný súbor a vyrobí z neho zoznam dvojíc (`list` s prvkami `tuple`) - starý obsah zoznamu v pamäti sa zruší a nahradí novým
   - malo by fungovať napr.

```
tz = TelefonnyZoznam('tel.txt')
tz.pridaj('Jana', '0901020304')
tz.pridaj(...
...
tz.zapis()      # zapísal do súboru
t2 = TelefonnyZoznam('tel.txt')
t2.citaj()
t2.vypis()      # pôvodny obsah
```

3. Zadefinujte triedu `VyrobPolygon`, ktorá bude fungovať takto:

   - metóda `__init__(meno_suboru)` si zapamätá meno súboru (nič ešte nezapisuje ani nečíta), vytvorí grafickú plochu (`self.canvas`) a v nej jeden polygón s jediným bodom (0, 0), s čiernym obrysom a bielym vnútrom; zároveň zviaže (`bind`) dve metódy `self.klik` a `self.enter` na udalosti kliknutia a stlačenie klávesu Enter (udalosť `'<Return>'`)
   - metóda `klik(event)` pridá do zoznamu `self.zoznam` kliknuté súradnice (nie ako dvojicu, ale dve celé čísla) a pomocou `self.canvas.coords(...)` zmení vykresľovaný polygón na obsah zoznamu
   - metóda `enter(event)` zapíše obsah zoznamu (súradnice polygónu) do súboru tak, že v každom riadku bude jedna dvojica súradníc (len 2 čísla)

- keď bude táto trieda hotová, program sa naštartuje pomocou:

```
VyrobPolygon('poly.txt')     # týmto sa zavolá konštruktor __init__()
```

4. Zadefinujte triedu `CitajPolygon`, ktorá vykreslí polygón uložený do súboru z úlohy (3). Zapíšte 2 metódy:
   - metóda `__init__(meno_suboru)` vytvorí grafickú plochu, prečíta súradnice z daného súboru a vykreslí polygón s čiernym obrysom a bielym vnútrom; zároveň zviaže metódu `self.prefarbi` s udalosťou kliknutia
   - metóda `prefarbi(event)` zmení vnútro (`fill`) nakresleného polygónu na náhodnú farbu
   - keď bude táto trieda hotová, program sa naštartuje pomocou:

```
CitajPolygon('poly.txt')     # týmto sa zavolá konštruktor __init__()
```

5. Zadefinujte triedu `MojaGrafika` s týmito metódami:
   - metóda `__init__()` vytvorí grafickú plochu veľkosti 400x300 (atribút `self.canvas`)
   - metóda `kruh(r, x, y, farba=None)` nakreslí kruh s polomerom `r` so stredom `(x, y)` s danou výplňou (`None` označuje náhodnú farbu)
   - metóda `stvorec(a, x, y, farba=None)` nakreslí štvorec so stranou `a` so stredom `(x, y)` s danou výplňou (`None` označuje náhodnú farbu)
   - metóda `text(text, x, y, farba=None)` vypíše daný text na súradnice `(x, y)` s danou farbou (`None` označuje náhodnú farbu)
   - metóda `zapis(meno_suboru)` zapíše všetky nakreslené útvary do textového súboru: každý do samostatného riadka v tvare, napr. `kruh 40 100 150 red` alebo `text Python 100 50 #12ff3a`, ...
   - metóda `citaj(meno_suboru)` zruší všetky nakreslené objekty (`self.canvas.delete('all')`), prečíta súbor a nakreslí všetky v ňom zapísané útvary
   - napr.

```
g = MojaGrafika()
g.stvorec(280, 200, 150, 'yellow')
for x in range(20, 400, 40):
    g.kruh(20, x, 100)
g.text(200, 150, 'Python', 'red')
g.zapis('grafika.txt')               # vytvorí súbor
```

```
g = MojaGrafika()
g.citaj('grafika.txt')               # znovu ho prečíta a vykreslí
```

KAPITOLA 16

16. Triedy a dedičnosť

Čo už vieme o triedach a ich inštanciách:

- triedy sú kontajnery atribútov:
  - atribúty sú väčšinou funkcie, hovoríme im metódy
  - niekedy sú to premenné, hovoríme im triedne atribúty
  - niektoré metódy sú „magické“: začínajú aj končia dvomi znakmi ‚__‘ a každá z nich má pre Python svoje špeciálne využitie
- triedy sú vzormi na vytvárame inštancií (niečo ako formičky na vyrábanie nejakých výrobkov)
- aj inštancie sú kontajnery atribútov:
  - väčšinou sú to súkromné premenné inštancií
- ak nejaký atribút nie je v inštancii definovaný, tak Python zabezpečí, že sa použije atribút z triedy (inštancia automaticky „vidí“ triedne atribúty) - samozrejme, len ak tam existuje, inak sa o tom vyhlási chyba

16.1 Objektové programovanie

je v programovacom jazyku charakterizované týmito tromi vlastnosťami:

1. Zapuzdrenie

Zapuzdrenie (enkapsulácia, encapsulation) označuje:

- v objekte sa nachádzajú premenné aj metódy, ktoré s týmito premennými pracujú (hovoríme, že údaje a funkcie sú zapuzdrené v jednom celku)
- vďaka metódam môžeme premenné v objekte ukryť, takže zvonku sa s údajmi pracuje len pomocou týchto metód

- pripomeňme si triedu `Cas`: v atribútoch `hodiny` a `minuty` (prípadne `sekundy`) sa tieto hodnoty nachádzajú vždy v správnom tvare, t.j. sú to kladné čísla, pričom atribút `minuty` je vždy menší ako 60, predpokladáme, že s týmito atribútmi nepracujeme priamo, ale len pomocou metód `__init__()`, `__str__()`, `sucet()`, ...
- z tohto dôvodu, by sme niekedy potrebovali dáta skryť a preto doplniť funkcie, tzv. **getter** a **setter** pre tie atribúty, ktoré chceme nejako ochrániť, neskôr uvidíme ďalší pojem, ktorý s týmto súvisí, tzv. **vlastnosť** (property), napr.

```
class Cas:

    def __init__(self, hodiny=0, minuty=0, sekundy=0):
        cas = abs(3600*hodiny + 60*minuty + sekundy)
        self.hod = cas // 3600
        self.min = cas // 60 % 60
        self.sek = cas % 60

    def __str__(self):
        return f'{self.hod}:{self.min:02}:{self.sek:02}'

    def sucet(self, iny):
        return Cas(self.hod+iny.hod, self.min+iny.min, self.sek+iny.sek)

    def rozdiel(self, iny):
        return Cas(sekundy = self.pocet_sekund() - iny.pocet_sekund())

    def pocet_sekund(self):
        return 3600 * self.hod + 60 * self.min + self.sek

    def vacsi(self, iny):
        return self.pocet_sekund() > iny.pocet_sekund()

    def hodiny(self):                       # getter
        return self.hod

    def zmen_hodiny(self, hodiny):          # setter
        self.hod = abs(hodiny)

    def minuty(self):                       # getter
        return self.min

    def zmen_minuty(self, minuty):          # setter
        self.min = minuty % 60
        self.hod += minuty // 60

    ...
```

- podobne by sme zrealizovali `sekundy()` aj `zmen_sekundy()`
- vieme to zapísať aj pre vylepšenú verziu s jediným apribútom premennou `sek`:

```
class Cas:

    def __init__(self, hodiny=0, minuty=0, sekundy=0):
        self.sek = abs(3600*hodiny + 60*minuty + sekundy)

    def __str__(self):
        return f'{self.sek//3600}:{self.sek//60%60:02}:{self.sek%60:02}'
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def sucet(self, iny):
        return Cas(sekundy=self.sek+iny.sek)

    def rozdiel(self, iny):
        return Cas(sekundy=self.sek-iny.sek)

    def vacsi(self, iny):
        return self.sek > iny.sek

    def hodiny(self):                    # getter
        return self.sek // 3600

    def zmen_hodiny(self, hodiny):       # setter
        self.sek = 3600 * abs(hodiny) + self.sek % 3600

    ...
```

2. Dedičnosť

Dedičnosť (inheritance) označuje, že

- novú triedu nevytvárame z nuly, ale využijeme už existujúcu triedu
- tejto vlastnosti sa budeme venovať v tejto prednáške

3. Polymorfizmus

Tejto vlastnosti objektového programovania sa budeme venovať v ďalších prednáškach.

16.2 Dedičnosť

Začneme definíciou jednoduchej triedy:

```
class Bod:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __str__(self):
        return f'Bod({self.x},{self.y})'

    def posun(self, dx=0, dy=0):
        self.x += dx
        self.y += dy

bod = Bod(100, 50)
bod.posun(-10, 40)
print('bod =', bod)
```

Toto by nemalo byť pre nás nič nové. Tiež sme sa už stretli s tým, že:

```
>>> dir(Bod)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',
 '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'posun']
```

V tomto výpise všetkých atribútov triedy `Bod` vidíme nielen nami definované tri metódy: `__init__()`, `__str__()` a `posun()`, ale aj veľké množstvo neznámych identifikátorov, o ktorých asi netušíme odkiaľ sa tu nabrali a na čo slúžia.

V Pythone, keď vytvárame novú triedu, tak sa táto „nenarodí“ úplne prázdna, ale získava niektoré dôležité atribúty od základnej Pythonovskej triedy `object`. Keď pri definovaní triedy zapíšeme:

```
class Bod:
    ...
```

v skutočnosti to znamená:

```
class Bod(object):
    ...
```

Do okrúhlych zátvoriek píšeme triedu (v tomto prípade triedu `object`), z ktorej sa vytvára naša nová trieda `Bod`. Vďaka tomuto naša nová trieda už pri „narodení“ pozná základnú množinu atribútov a my našimi definíciami metód tieto atribúty buď prepisujeme alebo pridávame nové. Tomuto mechanizmu sa hovorí **dedičnosť** a znamená to, že z jednej triedy vytvárame nejakú inú:

- triede, z ktorej vytvárame nejakú novú, sa hovorí **základná trieda**, alebo **bázová trieda**, alebo **super trieda** (base class, super class)
- triede, ktorá vznikne dedením z inej triedy, hovoríme **odvodená trieda**, alebo **podtrieda** (derived class, subclass)

Niekedy sa vzťahu základná trieda a odvodená trieda hovorí aj terminológiou **rodič** a **potomok** (potomok zdedil nejaké vlastnosti od svojho rodiča).

16.2.1 Odvodená trieda

Vytvorme nový typ (triedu) z triedy, ktorú sme definovali my, napr. z triedy `Bod` vytvoríme novú triedu `FarebnyBod`:

```
class FarebnyBod(Bod):
    def zmen_farbu(self, farba):
        self.farba = farba
```

Vďaka takémuto zápisu trieda `FarebnyBod` získava už pri narodení metódy `__init__()`, `__str__()` a `posun()`, pritom metódu `zmen_farbu()` sme jej dodefinovali teraz. Teda môžeme využívať všetko z definície triedy, z ktorej sme **odvodili** novú triedu (t.j. všetky atribúty, ktoré sme **zdedili**). Môžeme teda zapísať:

```
fbod = FarebnyBod(200, 50)          # volá __init__() z triedy Bod
fbod.zmen_farbu('red')              # volá zmen_farbu() z triedy FarebnyBod
fbod.posun(dy=50)                   # volá posun() z triedy Bod
print('fbod =', fbod)               # volá __str__() z triedy Bod
```

Zdedené metódy môžeme v novej triede nielen využívať, ale aj predefinovať - napr. môžeme zmeniť inicializáciu `__init__()`:

```
class FarebnyBod(Bod):
    def __init__(self, x, y, farba='black'):
        self.x = x
        self.y = y
        self.farba = farba

    def zmen_farbu(self, farba):
        self.farba = farba

fbod = FarebnyBod(200, 50, 'green')
fbod.posun(dy=50)
print('fbod =', fbod)
```

Pôvodná verzia inicializačnej metódy `__init__()` z triedy `Bod` sa teraz prekryla novou verziou tejto metódy, ktorá má teraz už tri parametre. Ak by sme v metóde `__init__()` chceli využiť pôvodnú verziu tejto metódy zo základnej triedy `Bod`, môžeme ju z tejto metódy zavolať, ale **nesmieme** to urobiť takto:

```
class FarebnyBod(Bod):
    def __init__(self, x, y, farba='black'):
        self.__init__(x, y)
        self.farba = farba
    ...
```

Toto je totiž **rekurzívne volanie**, ktoré spôsobí spadnutie programu `RecursionError: maximum recursion depth exceeded`. Musíme to zapísať takto:

```
class FarebnyBod(Bod):
    def __init__(self, x, y, farba='black'):
        Bod.__init__(self, x, y)              # inicializácia zo základnej triedy
        self.farba = farba
    ...
```

T.j. pri inicializácii inštancie triedy `FarebnyBod` najprv použi inicializáciu ako keby to bola inicializácia základnej triedy `Bod` (inicializuje atribúty `x` a `y`) a potom ešte inicializuj niečo navyše - t.j. atribút `farba`. Dá sa to zapísať ešte univerzálnejšie:

```
class FarebnyBod(Bod):
    def __init__(self, x, y, farba='black'):
        super().__init__(x, y)
        self.farba = farba
    ...
```

Štandardná funkcia `super()` na tomto mieste označuje: urob tu presne to, čo by na tomto mieste urobil môj rodič (t.j. moja super trieda). Tento zápis uvidíme aj v ďalších ukážkach.

16.2.2 Grafické objekty

Trochu sme upravili grafické objekty `Kruh`, `Obdlznik` a `Skupina` z prednášky: *15. Triedy a metódy*:

```
import tkinter

class Kruh:
    canvas = None
    typ = 'kruh'
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def __init__(self, x, y, r, farba='red'):
        self.x, self.y, self.r = x, y, r
        self.farba = farba
        self.id = self.canvas.create_oval(
            self.x - self.r, self.y - self.r,
            self.x + self.r, self.y + self.r,
            fill=self.farba)

    def __str__(self):
        return f'Kruh({self.x},{self.y},{self.r},{self.farba!r})'

    def posun(self, dx=0, dy=0):
        self.x += dx
        self.y += dy
        self.canvas.move(self.id, dx, dy)

    def zmen(self, r):
        self.r = r
        self.canvas.coords(self.id,
            self.x - self.r, self.y - self.r,
            self.x + self.r, self.y + self.r)

    def prefarbi(self, farba):
        self.farba = farba
        self.canvas.itemconfig(self.id, fill=farba)

class Obdlznik:
    canvas = None
    typ = 'obdlznik'

    def __init__(self, x, y, sirka, vyska, farba='red'):
        self.x, self.y, self.sirka, self.vyska = x, y, sirka, vyska
        self.farba = farba
        self.id = self.canvas.create_rectangle(
             self.x, self.y,
             self.x + self.sirka, self.y + self.vyska,
             fill=self.farba)

    def __str__(self):
        return f'Obdlznik({self.x},{self.y},{self.sirka},{self.vyska},{self.farba!r})'

    def posun(self, dx=0, dy=0):
        self.x += dx
        self.y += dy
        self.canvas.move(self.id, dx, dy)

    def zmen(self, sirka, vyska):
        self.sirka, self.vyska = sirka, vyska
        self.canvas.coords(self.id,
            self.x, self.y,
            self.x + self.sirka, self.y + self.vyska)

    def prefarbi(self, farba):
        self.farba = farba
        self.canvas.itemconfig(self.id, fill=farba)

class Skupina:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def __init__(self):
        self.zoznam = []

    def pridaj(self, utvar):
        self.zoznam.append(utvar)

    def prefarbi(self, farba):
        for utvar in self.zoznam:
            utvar.prefarbi(farba)

    def posun(self, dx=0, dy=0):
        for utvar in self.zoznam:
            utvar.posun(dx, dy)

    def posun_typ(self, typ, dx=0, dy=0):
        for utvar in self.zoznam:
            if utvar.typ == typ:
                utvar.posun(dx, dy)

    def prefarbi_typ(self, typ, farba):
        for utvar in self.zoznam:
            if utvar.typ == typ:
                utvar.prefarbi(farba)

#----------------------------------------

c = Kruh.canvas = Obdlznik.canvas = tkinter.Canvas(bg='white')
c.pack()

k = Kruh(50, 50, 30, 'blue')
r = Obdlznik(100, 20, 100, 50)
k.prefarbi('green')
r.posun(50)
```

Všimnite si:

- obe triedy `Kruh` aj `Obdlznik` majú niektoré atribúty aj metódy úplne rovnaké (napr. `x`, `y`, `farba`, `posun`, `zmen`)
- ak by sme chceli využiť dedičnosť (jedna trieda zdedí nejaké atribúty a metódy od inej), nie je rozumné, aby `Kruh` niečo dedil z triedy `Obdlznik`, alebo naopak `Obdlznik` bol odvodený z triedy `Kruh`

Zadefinujeme novú triedu `Utvar`, ktorá bude predkom (rodičom, bude základnou triedou) oboch tried `Kruh` aj `Obdlznik` - táto trieda bude obsahovať všetky spoločné atribúty týchto tried, t.j. aj niektoré metódy:

```
import tkinter

class Utvar:
    canvas = None

    def __init__(self, x, y, farba='red'):
        self.x, self.y, self.farba = x, y, farba
        self.id = None

    def posun(self, dx=0, dy=0):
        self.x += dx
        self.y += dy
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        self.canvas.move(self.id, dx, dy)

    def prefarbi(self, farba):
        self.farba = farba
        self.canvas.itemconfig(self.id, fill=farba)

Utvar.canvas = tkinter.Canvas(width=400, height=400)
Utvar.canvas.pack()
```

Uvedomte si, že nemá zmysel vytvárať objekty tejto triedy, lebo okrem inicializácie zvyšné metódy nemajú zmysel. Teraz dopíšme triedy `Kruh` a `Obdlznik`:

```
class Kruh(Utvar):
    def __init__(self, x, y, r, farba='red'):
        super().__init__(x, y, farba)
        self.r = r
        self.id = self.canvas.create_oval(
             self.x - self.r, self.y - self.r,
             self.x + self.r, self.y + self.r,
             fill=self.farba)

    def zmen(self, r):
        self.r = r
        self.canvas.coords(self.id,
            self.x - self.r, self.y - self.r,
            self.x + self.r, self.y + self.r)

class Obdlznik(Utvar):
    def __init__(self, x, y, sirka, vyska, farba='red'):
        super().__init__(x, y, farba)
        self.sirka, self.vyska = sirka, vyska
        self.id = self.canvas.create_rectangle(
             self.x, self.y,
             self.x + self.sirka, self.y + self.vyska,
             fill=self.farba)

    def zmen(self, sirka, vyska):
        self.sirka, self.vyska = sirka, vyska
        self.canvas.coords(self.id,
            self.x, self.y,
            self.x + self.sirka, self.y + self.vyska)
```

Zrušili sme atribút `typ`, ktorý slúžil pre metódy `posun_typ` a `prefarbi_typ` triedy `Skupina`: vďaka atribútu `typ` mali tieto metódy vplyv len na inštancie príslušného typu. Uvidíme, že tento atribút naozaj nepotrebujeme.

16.2.3 Testovanie typu inštancie

Pomocou štandardnej funkcie `type()` vieme otestovať, či je inštancia konkrétneho typu, napr.

```
>>> t1 = Kruh(10, 20, 30)
>>> t2 = Obdlznik(40, 50, 60, 70)
>>> type(t1) == Kruh
True
>>> type(t2) == Kruh
False
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> type(t2) == Obdlznik
True
>>> type(t1) == Utvar
False
```

Okrem tohto testu môžeme použiť štandardnú funkciu `isinstance(i, t)`, ktorá zistí, či je inštancia `i` typu `t` alebo je typom niektorého jeho predka, preto budeme radšej písať:

```
>>> t1 = Kruh(10, 20, 30)
>>> t2 = Obdlznik(40, 50, 60, 70)
>>> isinstance(t1, Kruh)
True
>>> isinstance(t1, Utvar)
True
>>> isinstance(t2, Kruh)
False
>>> isinstance(t2, Utvar)
True
```

Môžeme teraz prepísať metódy `posun_typ` a `prefarbi_typ` triedy `Skupina` takto:

```
class Skupina:
    def __init__(self):
        self.zoznam = []

    def pridaj(self, utvar):
        self.zoznam.append(utvar)

    def prefarbi(self, farba):
        for utvar in self.zoznam:
            utvar.prefarbi(farba)

    def posun(self, dx=0, dy=0):
        for utvar in self.zoznam:
            utvar.posun(dx, dy)

    def posun_typ(self, typ, dx=0, dy=0):
        for utvar in self.zoznam:
            if isinstance(utvar, typ):
                utvar.posun(dx, dy)

    def prefarbi_typ(self, typ, farba):
        for utvar in self.zoznam:
            if isinstance(utvar, typ):
                utvar.prefarbi(farba)
```

a použiť takto:

```
import random

def ri(a, b):
    return random.randint(a, b)

sk = Skupina()
for i in range(20):
    if ri(0, 1):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        sk.pridaj(Kruh(ri(50, 350), ri(50, 350), ri(10, 25)))
    else:
        sk.pridaj(Obdlznik(ri(50, 350), ri(50, 350), ri(10, 50), ri(10, 50)))

sk.prefarbi_typ(Kruh, 'yellow')
sk.posun_typ(Obdlznik, -10, -25)
```

- volanie `prefarbi_typ` zmení farbu všetkých kruhov v skupine na žltú
- volanie `posun_typ` posunie len všetky obdĺžniky

Všimnite si pomocnú funkciu `ri()`, ktorú sme definovali len pre zjednodušenie zápisu volania funkcie `random.randint()` (náhodná hodnota z daného intervalu, na rozdiel od `randrange()` medzi náhodnými číslami sa vyskytne aj horná hranica intervalu). Ten istý efekt by sme dosiahli, keby sme namiesto `def ri(...): ...` zapísali:

```
import random

ri = random.randint
```

Takto sme vytvorili premennú `ri`, ktorá je referenciou na funkciu `randint` z modulu `random`. Keďže z tohto modulu v našom programe nevyužívame žiadne iné funkcie, môžeme takýto zápis funkcie `ri` ešte zapísať inak - samotný príkaz `import` to umožňuje urobiť takto:

```
from random import randint as ri
```

Môžeme to prečítať takto: z modulu `random` použijeme (importujeme) iba funkciu `randint` a pritom ju v našom programe chceme volať ako `ri`. Niekedy môžete vidieť aj takýto zápis:

```
from math import sin, cos, pi
```

Z modulu `math` importujeme len tieto tri funkcie. Pri ich používaní už nebudeme musieť písať predponu `math.`

16.2.4 Odvodená trieda od Turtle

Aj od triedy `Turtle` z prednášky: *11. Korytnačky (turtle)* môžeme odvádzať nové triedy, napr.

```
import turtle

class MojaTurtle(turtle.Turtle):
    def stvorec(self, velkost):
        for i in range(4):
            self.fd(velkost)
            self.rt(90)

t = MojaTurtle()
t.stvorec(100)
t.lt(30)
t.stvorec(200)
```

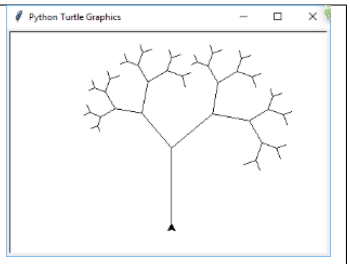
Zadefinovali sme novú triedu `MojaTurtle`, ktorá je odvodená od triedy `Turtle` (z modulu `turtle`, preto musíme písať `turtle.Turtle`) a oproti pôvodnej triede má dodefinovanú novú metódu `stvorec()`. Samozrejme, že túto metódu môžu volať len korytnačky typu `MojaTurtle`, obyčajné korytnačky pri takomto volaní metódy `stvorec()` hlásia chybu.

Môžeme definovať aj zložitejšie metódy, napr. aj rekurzívny strom:

```
import turtle

class MojaTurtle(turtle.Turtle):
    def strom(self, n, d):
        self.fd(d)
        if n > 0:
            self.lt(40)
            self.strom(n - 1, d * 0.6)
            self.rt(90)
            self.strom(n  -1, d * 0.7)
            self.lt(50)
        self.bk(d)

t = MojaTurtle()
t.lt(90)
t.strom(5, 100)
```



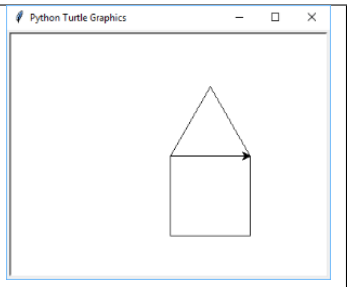
Niekedy nám môže chýbať to, že trieda `Turtle` neumožňuje vytvoriť korytnačku inde ako v strede plochy. Predefinujme inicializáciu našej novej korytnačky a zároveň sme tu zadefinujme metódu `domcek()`, ktorá nakreslí domček zadanej veľkosti:

```
import turtle

class MojaTurtle(turtle.Turtle):
    def __init__(self, x=0, y=0):
        super().__init__()
        self.speed(0)
        self.pu()
        self.setpos(x, y)
        self.pd()

    def domcek(self, dlzka):
        for uhol in 90, 90, 90, 30, 120, -60:
            self.fd(dlzka)
            self.rt(uhol)

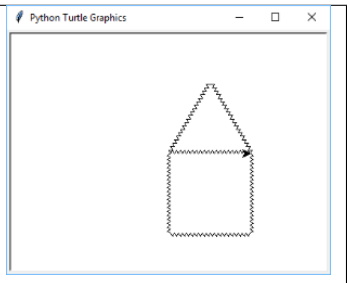
t = MojaTurtle(-200, 100)
t.domcek(100)
```



Vytvorme dve odvodené triedy od triedy `MojaTurtle`, v ktorých pozmeníme kreslenie rovnej čiary. Trieda `MojaTurtle1` prepíše korytnačiu metódu `fd()` tak, že namiesto rovnej čiary danej dĺžky, nakreslí cikcakovú čiaru, pričom skončí presne v tom mieste, kde by skončila aj pôvodná rovná čiara:

```
class MojaTurtle1(MojaTurtle):
    def fd(self, dlzka):
        while dlzka >= 5:
            self.lt(60)
            super().fd(5)
            self.rt(120)
            super().fd(5)
            self.lt(60)
            dlzka -= 5
        super().fd(dlzka)

MojaTurtle1().domcek(100)
```



Zapis `MojaTurtle1().domcek(100)` označuje, že najprv vytvoríme novú inštanciu `MojaTurtle1()`, ale

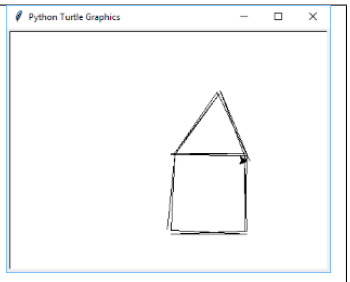
namiesto toho, aby sme ju priradili do nejakej premennej, napr. `t=MojaTurtle1()` a s ňou ďalej pracovali, napr. `t.domcek(100)`, tak sme priamo bez priradenia zavolali danú metódu. Toto sa zvykne robiť vtedy, keď inštanciu potrebujeme len na jedno zavolanie jej metódy a nepredpokladáme, že budeme potrebovať premennú na prístup k tejto inštancii.

Trieda `MojaTurtle2` namiesto jednej rovnej čiary danej dĺžky, nakreslí tri čiary tejto dĺžky, pričom sa zakaždým otočí o 180 stupňov plus nejaká malá náhodná odchýlka <-3, 3> stupne. Vďaka tejto odchýlke môže vzniknúť efekt, že kresba domčeka vznikla kreslením od ruky:

```
from random import randint as ri

class MojaTurtle2(MojaTurtle):
    def fd(self, dlzka):
        super().fd(dlzka)
        self.rt(180 - ri(-3, 3))
        super().fd(dlzka)
        self.rt(180 - ri(-3, 3))
        super().fd(dlzka)

MojaTurtle2().domcek(100)
```



16.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Zadefinujte triedu `Ucet` s metódami:
   - `__init__(meno, suma)` - meno účtu a počiatočná suma, napr. `Ucet('mbank', 100)` alebo `Ucet('jbanka')`
   - `__str__()` - reťazec v tvare `'ucet mbank -> 100 euro'` alebo `ucet jbanka -> 0 euro`
   - `stav()` - vráti momentálny stav účtu
   - `vklad(suma)` - danú sumu pripočíta k účtu
   - `vyber(suma)` - vyberie sumu z účtu (len ak je to kladné číslo), vráti vybranú sumu, ak je na účte menej ako požadovaná suma, vyberie len toľko koľko sa dá
   - otestujte napr.

```
mbank = Ucet('mbank')
csob = Ucet('csob', 100)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
tatra = Ucet('tatra', 17)
sporo = Ucet('sporo', 50)
mbank.vklad(sporo.vyber(30) + tatra.vyber(30))
csob.vyber(-5)
spolu = 0
for ucet in mbank, csob, tatra, sporo:
    print(ucet)
    spolu += ucet.stav()
print('spolu = ', spolu)
```

- vypíše:

```
ucet mbank -> 47 euro
ucet csob -> 100 euro
ucet tatra -> 0 euro
ucet sporo -> 20 euro
spolu =  167
```

2. Zadefinujte triedu `UcetHeslo`, ktorá je odvodená z triedy `Ucet` a má takto zmenené správanie:
   - `__init__(meno, heslo, suma)` - k účtu si zapamätá aj heslo
   - `vklad(suma)` - si najprv vypýta heslo a až keď je správne, zrealizuje vklad
   - `vyber(suma)` - si najprv vypýta heslo a až keď je správne, zrealizuje výber, inak vráti `None`
   - pri definovaní týchto metód využite volania ich pôvodných verzií z triedy `Ucet`
   - otestujte napr.

```
mbank = UcetHeslo('mbank', 'gigi')
csob = Ucet('csob', 100)
tatra = UcetHeslo('tatra', 'gogo', 17)
sporo = Ucet('sporo', 50)
mbank.vklad(sporo.vyber(30) + tatra.vyber(30))
csob.vyber(-5)
spolu = 0
for ucet in mbank, csob, tatra, sporo:
    print(ucet)
    spolu += ucet.stav()
print('spolu = ', spolu)
```

   - tento program si najprv dvakrát vypýta heslo:

```
zadaj heslo uctu tatra: gogo
zadaj heslo uctu mbank: gigi
```

   - a až potom (po správnom zadaní hesiel) vypíše to isté, ako predtým
   - zistite, čo sa stane s účtami, keď pre `'mbank'` určíme chybné heslo
3. Na prednáške sa kreslil domček pomocou korytnačky, ktorá malá pozmenenú metódu `fd()`. Zadefinujte triedu `MojaTurtle3`, ktorá bude odvodená od `MojaTurtle` s metódou `domcek()`. Táto nová trieda `MojaTurtle3` bude mať dodefinovanú jedinú metódu:
   - metóda `rt()` sa bude pri otáčaní náhodne mýliť, t.j. k uhlu pripočíta náhodné číslo z `<-3, 3>`:

```
class MojaTurtle3(MojaTurtle):
    def rt(self, uhol):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        ...

MojaTurtle3().domcek(100)
```

- zistite, ako sa zmení kresba domčeka, keď triedu `MojaTurtle3` odvodíme z `MojaTurtle1` (s cikcakovým `fd()`) alebo `MojaTurtle2` (s `fd()`, ktorý každú čiaru prejde trikrát)

4. Zadefinujte dve nové triedy `Turtle1` a `Turtle2`, obidve odvodené od `Turtle`, pričom obe majú zadefinovanú metódu `otoc()`

   - metóda `otoc(uhol)` v triede `Turtle1` otočí korytnačku o zadaný uhol **vľavo**, v triede `Turtle2` ju otočí **vpravo**:

```
from turtle import Turtle
from random import randrange as rr

class Turtle1(Turtle):
    ...

class Turtle2(Turtle):
    ...
```

   - teraz naprogramujte takýto test týchto dvoch tried:

     – na x-ovej osi rovnomerne rozložte 20 korytnačiek s rozostupmi 20 krokov, všetky budú otočené na východ - náhodným generátorom rozhodnite, ktorá z nich bude inštancia triedy `Turtle1` a ktorá `Turtle2` - korytnačky uložte do zoznamu

     – teraz postupne prejdete všetky korytnačky z tohto zoznamu a zmeníte im farbu pera na červenú (pre `Turtle1`) alebo na modrú (pre `Turtle2`)

     – na záver štyrikrát zopakujete: každá korytnačka prejde 20 krokov a otočí sa pomocou `otoc()` o 90 stupňov

5. Naprogramujte triedu `Pero`, pomocou ktorej budeme vedieť kresliť do grafickej plochy. Trieda má tieto metódy:

   - `__init__(x=0, y=0)`, ak ešte nebol vytvorený `canvas`, vytvorí ho s danou šírkou a výškou, zapamätá si súradnice pera a to, že pero je spustené dolu (bude kresliť)
   - `pu()` zdvihne pero, odteraz pohyb pera nekreslí
   - `pd()` spustí pero, pohyb bude zanechávať čiaru
   - `setpos(x, y)` presunie pero na novú pozíciu, ak je pero spustené, zanecháva čiernu čiaru hrúbky 1:

```
import tkinter
from math import sin, cos, radians

class Pero:
    canvas = None
    sirka, vyska = 400, 300

    def __init__(...):
        ...
    ...
```

   - otestujte vytvorením dvoch inštancií pera, ktoré nakreslia napr. dva štvorce:

```
p1 = Pero(100, 200)
p2 = Pero(200, 150)
...
```

6. Zadefinujte novú triedu `Turtle`, ktorá bude odvodená od triedy `Pero` z úlohy (5):
   - metóda `__init__()` vytvorí pero v strede plochy a do nového atribútu `uhol` nastaví 0 (teda otočenie smerom na východ)
   - metódy `lt(uhol)` a `rt(uhol)` zmenšia, resp. zväčšia atribút `uhol` o zadanú hodnotu, uhly počítajte v stupňoch
   - metóda `fd(dlzka)` presunie pero (zavolá metódu `setpos()`) o zadanú dĺžku, ktorá je v momentálnom smere natočenia
     - asi použijete nejaký takýto vzorec pre nové `x` a `y`: `x+dlzka*cos(uhol)`, `y+dlzka*sin(uhol)`
     - nezabudnite, že `sin()` a `cos()` fungujú v radiánoch, pričom náš atribút `uhol` pracuje v stupňoch
   - nepoužívajte modul `turtle`
   - otestujte napr.

```
class Turtle(Pero):
    ...

#---- test -------

t = Turtle()
for i in range(1, 200, 2):
    t.fd(i)
    t.lt(89)
```

7. Z triedy `Turtle` zo (6) úlohy odvoďte triedu `Turtle1`, do ktorej dopíšete metódu `strom(n, d)` (z prednášky)
   - potom otestujte, napr.

```
t = Turtle1()
t.lt(90)
t.strom(5, 60)
```

8. Otestujte, ako v tejto triede fungujú príklady z prednášky s kreslením domčeka rôznym typom čiar
   - napr.

```
class MojaTurtle(Turtle):
    def domcek(...):
        ...
class MojaTurtle1(MojaTurtle):
    def fd(...):
        ...
class MojaTurtle2(MojaTurtle):
    def fd(...):
        ...

MojaTurtle1().domcek(100)
...
```

KAPITOLA 17

17. Výnimky

Zapíšme funkciu, ktorá prečíta celé číslo zo vstupu:

```
def cislo():
    vstup = input('zadaj cislo: ')
    return int(vstup)
```

```
>>> cislo()
zadaj cislo: 234
234
```

Toto funguje, len ak zadáme korektný reťazec celého čísla. Spadne to s chybovou správou pri zle zadanom vstupe:

```
>>> cislo()
zadaj cislo: 234a
...
ValueError: invalid literal for int() with base 10: '234a'
```

Aby takýto prípad nenastal, vložíme pred volanie funkcie `int()` test, napr. takto

```
def test_cele_cislo(retazec):
    for znak in retazec:
        if znak not in '0123456789':
            return False
    return True

def cislo():
    vstup = input('zadaj cislo: ')
    if test_cele_cislo(vstup):
        return int(vstup)
    print('chybne zadane cele cislo')
```

Pripadne s opakovaným vstupom pri chybne zadanom čísle (hoci aj tento test `test_cele_cislo()` nie je dokonalý a niekedy prejde, aj keď nemá):

```
def cislo():
    while True:
        vstup = input('zadaj cislo: ')
        if test_cele_cislo(vstup):
            return int(vstup)
        print('*** chybne zadane cele cislo ***')
```

Takto ošetrený vstup už nespadne, ale oznámi chybu a pýta si nový vstup, napr.

```
>>> cislo()
zadaj cislo: 234a
*** chybne zadane cele cislo ***
zadaj cislo: 234 a
*** chybne zadane cele cislo ***
zadaj cislo: 234
234
>>>
```

17.1 try - except

Python umožňuje aj iný spôsob riešenia takýchto situácií: chybe sa nebudeme snažiť predísť, ale keď vznikne, tak ju „ošetríme“. Využijeme novú programovú konštrukciu `try` - `except`. Jej základný tvar je

```
try:
    '''blok príkazov'''
except MenoChyby:
    '''ošetrenie chyby'''
```

Konštrukcia sa skladá z dvoch častí:

- príkazy medzi `try` a `except`
- príkazmi za `except`

Blok príkazov medzi `try` a `except` bude teraz Python spúšťať „opatrnejšie“, t.j. ak pri ich vykonávaní nastane uvedená chyba (meno chyby za `except`), vykonávanie bloku príkazov sa okamžite ukončí a pokračuje sa príkazmi za `except`, pritom Python zruší chybový stav, v ktorom sa práve nachádzal. Ďalej sa pokračuje v príkazoch za touto konštrukciou.

Ak pri opatrnejšom vykonávaní bloku príkazov uvedená chyba nenastane, tak príkazy za `except` sa preskočia a normálne sa pokračuje v príkazoch za konštrukciou.

Ak pri opatrnejšom vykonávaní bloku príkazov nastane iná chyba ako je uvedená v riadku `except`, tak táto konštrukcia túto chybu nespracuje, ale pokračuje sa tak, ako sme boli zvyknutí doteraz, t.j. celý program spadne a IDLE o tom vypíše chybovú správu.

Ukážme to na predchádzajúcom príklade (pomocnú funkciu `test_cele_cislo()` teraz už nepotrebujeme):

```
def cislo():
    while True:
        vstup = input('zadaj cislo: ')
        try:
            return int(vstup)
        except ValueError:
            print('*** chybne zadane cele cislo ***')
```

To isté by sa dalo zapísať aj niekoľkými inými spôsobmi, napr.

```
def cislo():
    while True:
        try:
            return int(input('zadaj cislo: '))
        except ValueError:
            print('*** chybne zadane cele cislo ***')
```

```
def cislo():
    while True:
        try:
            vysledok = int(input('zadaj cislo: '))
            break
        except ValueError:
            print('*** chybne zadane cele cislo ***')
    return vysledok
```

```
def cislo():
    ok = False
    while not ok:
        try:
            vysledok = int(input('zadaj cislo: '))
            ok = True
        except ValueError:
            print('*** chybne zadane cele cislo ***')
    return vysledok
```

Na chybové prípady odteraz nemusíme pozerať ako na niečo zlé, ale ako na výnimočné prípady (výnimky, t.j. **exception**), ktoré vieme veľmi šikovne vyriešiť. Len neošetrená výnimka spôsobí spadnutie nášho programu. Často to bude znamenať, že sme niečo zle naprogramovali, alebo sme nedomysleli nejaký špeciálny prípad.

Na predchádzajúcom príklade sme videli, že odteraz bude pre nás dôležité meno chyby (napr. ako v predchádzajúcom príklade **ValueError**). Mená chýb nám prezradí Python, keď vyskúšame niektoré konkrétne situácie, napr.

```
>>> 1+'2'
...
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> 12/0
...
ZeroDivisionError: division by zero
>>> x+1
...
NameError: name 'x' is not defined
>>> open('')
...
FileNotFoundError: [Errno 2] No such file or directory: ''
>>> [1,2,3][10]
...
IndexError: list index out of range
>>> 5()
...
TypeError: 'int' object is not callable
>>> ''.x
...
AttributeError: 'str' object has no attribute 'x'
>>> 2**10000/1.
...
OverflowError: int too large to convert to float
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> import x
...
ImportError: No module named 'x'
>>> def t(): x += 1
>>> t()
...
UnboundLocalError: local variable 'x' referenced before assignment
```

Všimnite si, že Python za meno chyby vypisuje aj nejaký komentár, ktorý nám môže pomôcť pri pochopení dôvodu chyby, resp. pri ladení. Tento text je ale mimo mena chyby, v `except` riadku ho nepíšeme. Takže, ak chceme odchytiť a spracovať nejakú konkrétnu chybu, jej meno si najjednoduchšie zistíme v dialógovom režime v IDLE.

17.1.1 Spracovanie viacerých výnimiek

Pomocou `try` a `except` môžeme zachytiť aj viac chýb ako jednu. V ďalšom príklade si funkcia vyžiada celé číslo, ktoré bude indexom do nejakého zoznamu. Funkcia potom vypíše hodnotu prvku s týmto indexom. Môžu tu nastať dve rôzne výnimky:

- **ValueError** pre zle zadané celé číslo indexu
- **IndexError** pre index mimo rozsah zoznamu

Zapíšme funkciu:

```
def zisti(zoznam):
    while True:
        try:
            vstup = input('zadaj index: ')
            index = int(vstup)
            print('prvok zoznamu =', zoznam[index])
            break
        except ValueError:
            print('*** chybne zadane cele cislo ***')
        except IndexError:
            print('*** index mimo rozsah zoznamu ***')
```

otestujeme:

```
>>> zisti(['prvy', 'druhy', 'treti', 'stvrty'])
zadaj index: 22
*** index mimo rozsah zoznamu ***
zadaj index: 2.
*** chybne zadane cele cislo ***
zadaj index: 2
prvok zoznamu = treti
>>>
```

To isté by sme dosiahli aj vtedy, keby sme to zapísali pomocou dvoch vnorených príkazov `try`:

```
def zisti(zoznam):
    while True:
        try:
            try:
                vstup = input('zadaj index: ')
                index = int(vstup)
                print('prvok zoznamu =', zoznam[index])
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
                break
            except ValueError:
                print('*** chybne zadane cele cislo ***')
        except IndexError:
            print('*** index mimo rozsah zoznamu ***')
```

17.1.2 Zlúčenie výnimiek

Niekedy sa môže hodiť, keď máme pre rôzne výnimky spoločný kód. Za `except` môžeme uviesť aj viac rôznych mien výnimiek, ale musíme ich uzavrieť do zátvoriek (urobiť z nich `tuple`), napr.

```
def zisti(zoznam):
    while True:
        try:
            print('prvok zoznamu =', zoznam[int(input('zadaj index: '))])
            break
        except (ValueError, IndexError):
            print('*** chybne zadany index zoznamu ***')
```

```
>>> zisti(['prvy', 'druhy', 'treti', 'stvrty'])
zadaj index: 22
*** chybne zadany index zoznamu ***
zadaj index: 2.
*** chybne zadany index zoznamu ***
zadaj index: 2
prvok zoznamu = treti
>>>
```

Uvedomte si, že pri takomto zlučovaní výnimiek môžeme stratiť detailnejšiu informáciu o tom, čo sa v skutočnosti udialo.

Príkaz `try - except` môžeme použiť aj bez uvedenia mena chyby: vtedy to označuje zachytenie všetkých typov chýb, napr.

```
def zisti(zoznam):
    while True:
        try:
            print('prvok zoznamu =', zoznam[int(input('zadaj index: '))])
            break
        except:
            print('*** chybne zadany index zoznamu ***')
```

Takýto spôsob použitia `try - except` sa ale **neodporúča**, skúste sa ho vyvarovať! Jeho používaním môžete ušetriť zopár minút pri hľadaní všetkých typov chýb, ktoré môžu vzniknúť pri vykonávaní daného bloku príkazov. Ale skúsenosti ukazujú, že môžete zase stratiť niekoľko hodín pri hľadaní chýb v takýchto programoch. Veľmi často sú takéto bloky `try - except` bez uvedenia výnimky zdrojom veľmi nečakaných chýb.

17.1.3 Ako funguje mechanizmus výnimiek

Kým sme nepoznali výnimky, ak nastala nejaká chyba v našej funkcii, dostali sme nejaký takýto výpis:

```
>>> fun1()
Traceback (most recent call last):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
  File "<pyshell#9>", line 1, in <module>
    fun1()
  File "p.py", line 13, in fun1
    fun2()
  File "p.py", line 16, in fun2
    fun3()
  File "p.py", line 19, in fun3
    fun4()
  File "p.py", line 22, in fun4
    int('x')
ValueError: invalid literal for int() with base 10: 'x'
```

Python pri výskyte chyby (t.j. nejakej výnimky) hneď túto chybu nevypisuje, ale zisťuje, či sa nevyskytla v bloku príkazu `try - except`. Ak áno a meno chyby zodpovedá menu v `except`, vykoná definované príkazy pre túto výnimku.

Ak na danom mieste neexistuje obsluha tejto výnimky, vyskočí z momentálnej funkcie a zisťuje, či jej volanie v nadradenej funkcii nebolo chránené príkazom `try - except`. Ak áno, vykoná čo treba a na tomto mieste pokračuje ďalej, akoby sa žiadna chyba nevyskytla.

Ak ale ani v tejto funkcii nie je kontrola pomocou `try - except`, vyskakuje o úroveň vyššie a vyššie, až kým nepríde na najvyššiu úroveň, teda do dialógu IDLE. Keďže nik doteraz nezachytil túto výnimku, IDLE ju vypíše v nám známom tvare. V tomto výpise vidíme, ako sa Python „vynáral“ vyššie a vyššie.

17.1.4 Práca so súbormi

Pri práci so súbormi výnimky vznikajú veľmi často a nie je jednoduché ošetriť všetky situácie pomocou podmienených príkazov. Najčastejšou chybou je neexistujúci súbor:

```
try:
    with open('x.txt') as subor:
        cislo = int(subor.readline())
except FileNotFoundError:
    print('*** neexistujuci subor ***')
    cislo = 0
except (ValueError, TypeError):
    print('*** prvy riadok suboru neobsahuje cele cislo ***')
    cislo = 10
```

Prípadne sa môže hodiť pomocná funkcia, ktorá zistí, či súbor s daným menom existuje:

```
def existuje(meno_suboru):
    try:
        with open(meno_suboru):
            return True
    except (TypeError, OSError, FileNotFoundError):
        return False
```

Uvedomte si ale, že táto funkcia, aby zistila, či súbor existuje, ho otvorí a okamžite aj zatvorí. Z tohto dôvodu nasledovný kód nie je veľmi efektívny:

```
if existuje('x.txt'):
    with open('x.txt') as t:
        obsah = t.read()
else:
    obsah = ''
```

17.2 Vyvolanie výnimky

V niektorých situáciách sa nám môže hodiť vyvolanie vzniknutej chyby aj napriek tomu, že ju vieme zachytiť príkazom `try - except`. Slúži na to nový príkaz `raise`, ktorý má niekoľko variantov. Prvý z nich môžete vidieť v upravenej verzii funkcie `cislo()`. Funkcia sa najprv 3-krát pokúsi prečítať číslo zo vstupu, a ak sa jej to napriek tomu nepodarí, rezignuje a vyvolá známu chybu `ValueError`:

```
def cislo():
    pokus = 0
    while True:
        try:
            return int(input('zadaj cislo: '))
        except ValueError:
            pokus += 1
            if pokus >= 3:
                raise
            print('*** chybne zadane cele cislo ***')
```

```
>>> cislo()
zadaj cislo: jeden
*** chybne zadane cele cislo ***
zadaj cislo: dva
*** chybne zadane cele cislo ***
zadaj cislo: tri
...
ValueError: invalid literal for int() with base 10: 'tri'
```

Pomocou príkazu `raise` môžeme vyvolať nielen práve zachytenú výnimku, ale môžeme vyvolať ľubovoľnú inú chybu aj s vlastným komentárom, ktorý sa pri nej vypíše, napr.

```
raise ValueError('chybne zadane cele cislo')
raise ZeroDivisionError('delenie nulou')
raise TypeError('dnes sa ti vobec nedari')
```

17.2.1 Príklad s metódou index()

Poznáme už metódu `index()`, ktorá v zozname (typ `list`, `tuple` alebo `str`) nájde prvý výskyt nejakej hodnoty. Metóda je zaujímavá tým, že vyvolá výnimku `ValueError`, ak sa táto hodnota v zozname nenachádza. Kým sme nepoznali odchytávanie výnimiek, väčšinou sme museli najprv kontrolovať, či sa tam príslušný prvok nachádza a až potom, keď sa nachádza, volali sme `index()`, napr.

```
def nahrad(zoznam, h1, h2):
    if h1 in zoznam:
        i = zoznam.index(h1)
        zoznam[i] = h2
```

Pomocou `try - except` to vyriešime výrazne efektívnejšie:

```
def nahrad(zoznam, h1, h2):
    try:
        i = zoznam.index(h1)
        zoznam[i] = h2
    except ValueError:
        pass
```

Táto metóda `index()`, ktorá funguje pre jednorozmerné zoznamy, nás môže inšpirovať aj na úlohu, v ktorej budeme hľadať indexy do dvojrozmernej tabuľky. Napíšme funkciu `hladaj(zoznam, hodnota)`, ktorá hľadá prvý výskyt danej hodnoty v dvojrozmernom zozname a ak taký prvok nájde, vráti jeho číslo riadku a číslo stĺpca. Ak sa tam taký prvok nenachádza, funkcia by mala vyvolať rovnakú výnimku, ako to robila pôvodná metóda `index()`, t.j. `ValueError: hodnota is not in list`. Zapíšme riešenie dvoma vnorenými cyklami:

```
def hladaj(zoznam, hodnota):
    for r in range(len(zoznam)):
        for s in range(len(zoznam[r])):
            if zoznam[r][s] == hodnota:
                return r, s
    raise ValueError(f'{hodnota!r} is not in list')
```

alebo to isté zápis pomocou štandardnej funkcie `enumerate()`:

```
def hladaj(zoznam, hodnota):
    for r, riadok in enumerate(zoznam):
        for s, prvok in enumerate(riadok):
            if prvok == hodnota:
                return r, s
    raise ValueError(f'{hodnota!r} is not in list')
```

Hoci je toto správne riešenie, vieme ho zapísať aj efektívnejšie pomocou volania metódy `index()`:

```
def hladaj(zoznam, hodnota):
    for r, riadok in enumerate(zoznam):
        try:
            s = riadok.index(hodnota)
            return r, s
        except ValueError:
            pass
    raise ValueError(f'{hodnota!r} is not in list')
```

Ak si ale uvedomíme, že neúspešné hľadanie prvku v `r`-tom riadku zoznamu pomocou `index()` vyvolá presne tú istú chybu, ktorú sme zachytili a potom znovu vyvolali, môžeme to celé skrátiť takto:

```
def hladaj(zoznam, hodnota):
    for r, riadok in enumerate(zoznam):
        try:
            s = riadok.index(hodnota)
            return r, s
        except ValueError:
            if r == len(zoznam)-1:
                raise
```

Teda zachytená chyba `ValueError` v poslednom riadku dvojrozmerného zoznamu označuje, že sa hodnota nenachádza v žiadnom riadku zoznamu a teda sa opätovne vyvolá zachytená chyba. (Zamyslite sa, ako bude toto riešenie fungovať pre prázdny dvojrozmerný zoznam, teda zoznam, ktorý neobsahuje ani jeden riadok).

17.2.2 Vytváranie vlastných výnimiek

Keď chceme vytvoriť vlastný typ výnimky, musíme vytvoriť novú triedu odvodenú od základnej triedy `Exception`. Môže to vyzerať napr. takto:

```
class MojaChyba(Exception): pass
```

Príkaz `pass` tu znamená, že nedefinujeme nič nové oproti základnej triede `Exception`. Použiť to môžeme napr. takto:

```
def podiel(p1, p2):
    if not isinstance(p1, int):
        raise MojaChyba('prvy parameter nie je cele cislo')
    if not isinstance(p2, int):
        raise MojaChyba('druhy parameter nie je cele cislo')
    if p2 == 0:
        raise MojaChyba('neda sa delit nulou')
    return p1 // p2
```

```
>>> podiel(22, 3)
7
>>> podiel(2.2, 3)
...
MojaChyba: prvy parameter nie je cele cislo
>>> podiel(22, 3.3)
...
MojaChyba: druhy parameter nie je cele cislo
>>> podiel(22, 0)
...
MojaChyba: neda sa delit nulou
>>>
```

17.2.3 Kontrola pomocou assert

Ďalší nový príkaz `assert` slúži na kontrolu nejakej podmienky: ak táto podmienka nie je splnená, vyvolá sa výnimka `AssertionError` aj s uvedeným komentárom. Tvar tohto príkazu je:

```
assert podmienka, 'komentár'
```

Toto sa často používa pri ladení, keď potrebujeme mať istotu, že je splnená nejaká konkrétna podmienka (vlastnosť) a v prípade, že nie je, chceme radšej aby program spadol, ako pokračoval ďalej. Prepíšme funkciu `podiel()` tak, že namiesto `if` a `raise` zapíšeme volanie `assert`:

```
def podiel(p1, p2):
    assert isinstance(p1, int), 'prvy parameter nie je cele cislo'
    assert isinstance(p2, int), 'druhy parameter nie je cele cislo'
    assert p2 != 0, 'neda sa delit nulou'
    return p1 // p2
```

```
>>> podiel(2.2, 3)
...
AssertionError: prvy parameter nie je cele cislo
>>> podiel(22, 3.3)
...
AssertionError: druhy parameter nie je cele cislo
>>> podiel(22, 0)
...
AssertionError: neda sa delit nulou
```

17.2.4 Príklad s triedou Ucet

Na minulých cvičeniach ste riešili príklad, v ktorom ste definovali triedu `Ucet`, resp. `UcetHeslo` aj jej metódy `vklad()` a `vyber()`. Ukážme riešenie trochu zjednodušeného zadania len s jednou triedou, zatiaľ bez ošetrovania výnimiek:

```
class UcetHeslo:
    def __init__(self, meno, heslo, suma=0):
        self.meno, self.heslo, self.suma = meno, heslo, suma

    def __str__(self):
        return f'ucet {self.meno} -> {self.suma} euro'

    def vklad(self, suma):
        if self.heslo == input(f'heslo pre {self.meno}? '):
            self.suma += suma
        else:
            print('chybne heslo')

    def vyber(self, suma):
        if self.heslo == input(f'heslo pre {self.meno}? '):
            if suma <= 0:
                return 0
            suma = min(self.suma, suma)
            self.suma -= suma
            return suma
        else:
            print('chybne heslo')

#---- test ----

mbank = UcetHeslo('mbank', 'heslo1', 100)
csob = UcetHeslo('csob', 'heslo2', 30)
print('*** stav uctov:', mbank, csob, sep='\n')
mbank.vklad(csob.vyber(55))
print('*** stav uctov:', mbank, csob, sep='\n')
```

Po spustení a zadaní správnych hesiel, dostávame takýto výpis:

```
*** stav uctov:
ucet mbank -> 100 euro
ucet csob -> 30 euro
heslo pre csob? heslo2
heslo pre mbank? heslo1
*** stav uctov:
ucet mbank -> 130 euro
ucet csob -> 0 euro
```

Zdá sa, že test prebehol v poriadku.

Prepíšme riešenie s využitím výnimiek. Zadefinujeme pritom vlastnú výnimku `ChybnaTransakcia` a budeme sa snažiť ošetriť všetky možné chybové situácie:

```
class ChybnaTransakcia(Exception): pass

class UcetHeslo:
    def __init__(self, meno, heslo, suma=0):
        self.meno, self.heslo, self.suma = meno, heslo, suma
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def __str__(self):
        return f'ucet {self.meno} -> {self.suma} euro'

    def kontrola(self):
        if self.heslo and self.heslo != input(f'heslo pre {self.meno}? '):
            raise ChybnaTransakcia('chybne heslo pre pristup k uctu ' + self.meno)

    def vklad(self, suma):
        self.kontrola()
        try:
            if suma <= 0:
                raise TypeError
            self.suma += suma
        except TypeError:
            raise ChybnaTransakcia('chybne zadana suma pre vklad na ucet ' + self.
↪meno)

    def vyber(self, suma):
        self.kontrola()
        try:
            if suma <= 0:
                raise TypeError
            suma = min(self.suma, suma)
            self.suma -= suma
            return suma
        except TypeError:
            raise ChybnaTransakcia('chybne zadana suma pre vyber z uctu ' + self.meno)

def prevod(ucet1, ucet2, suma):
    '''prevedie sumu z ucet1 na ucet2'''
    suma = ucet1.vyber(suma)
    try:
        ucet2.vklad(suma)
    except ChybnaTransakcia:
        ucet1.suma += suma    # vrati vybratu sumu na ucet1
        raise

#---- test ----

mbank = UcetHeslo('mbank', 'heslo1', 100)
csob = UcetHeslo('csob', 'heslo2', 30)
print('*** stav uctov:', mbank, csob, sep='\n')
prevod(csob, mbank, 55)
print('*** stav uctov:', mbank, csob, sep='\n')
```

Všimnite si, že pomocná funkcia `prevod()` sa snaží pri neúspešnej transakcii vrátiť vybratú sumu peňazí - hoci to asi nerobí úplne korektne ...

17.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Nájdite čo najviac rôznych štandardných chybových správ, ktoré môžu vzniknúť pri práci so znakovými reťazcami (hľadajte aj chybové správy s volaniami funkcií a metód). Väčšinou to budú rôzne chyby typu `TypeError`.
   - napr.

```
>>> 'abc'[3]
IndexError: string index out of range
>>> 'abc'[3] = 'a'
TypeError: 'str' object does not support item assignment
>>> ...
```

2. Napíšte funkcie `cele(hodnota)` a `desatinne(hodnota)`, ktoré prevedú danú hodnotu na celé číslo (funkciou `int()`), resp. na desatinné (funkciou `float()`) a ak sa to nedá, funkcie vrátia nulu.
   - napr.

```
>>> cele(None)
0
>>> cele(3.14)
3
>>> desatinne('3.14')
3.14
>>> desatinne('3,14')
0.0
>>> desatinne(abs)
0.0
```

3. Napíšte funkciu `priemer(zoznam)`, ktorá vypočíta priemer hodnôt v danom zozname. Využite štandardnú funkciu `sum()` a to či je zoznam prázdny netestujte príkazom `if` ale odchyťte pomocou `try - except`. Ak je zoznam neprázdny a nedá sa vypočítať súčet prvkov, funkcia `priemer()` vráti 0, ak je zoznam prázdny, funkcia vráti `None`.
   - napr.

```
>>> priemer([1.5, 2, 3.14, 4])
2.66
>>> priemer([1, 2, [3], 4])
0
>>> print(priemer([]))
None
```

4. Napíšte funkciu `suma(zoznam)`, ktorá zistí číselný súčet prvkov zoznamu (bez štandardnej funkcie `sum()`). Zrejme bude postupne pripočítavať prvky zoznamu a ak sa nejaký pripočítať nedá (neprejde operácia +), tento prvok vynechá.
   - napr.

```
>>> suma([1.5, 2, 3.14, 4])
10.64
>>> suma([1, 2, [3], 4])
7
>>> print(suma([]))
0
```

5. Napíšte funkciu `suma2(zoznam)`, ktorá zistí číselný súčet prvkov zoznamu alebo n-tice. Ak je niektorý z

prvkov zoznamu opäť zoznam alebo n-tica, funkcia sa rekurzívne zavolá na tomto prvku a jeho súčet pripočíta k výsledku. Ak sa nejaký prvok pripočítať nedá (neprejde operácia +), tento prvok vynechá.

- napr.

```
>>> suma2([1, 2, '3', 4])
7
>>> suma2([1, 2, [3], 4])
10
>>> suma2([[], [1], 2, (3, 4)])
10
>>> suma2([[[[42]]]])
42
>>> suma2(((((5.5, 6.5),),),))
12.0
```

6. Napíšte funkciu `pocet_riadkov(meno_suboru)`, ktorá vráti počet riadkov zadaného súboru. Ak daný súbor neexistuje (nepodaril sa `open()`), funkcia vráti `-1`.

   - napr.

```
>>> pocet_riadkov('cvicenie.py')
104
>>> pocet_riadkov('x.x')     # pre neexistujuci subor
-1
```

   - nezabudnite zatvoriť otvorený súbor

7. Napíšte funkciu `daj_cislo(meno_suboru, index, inak=0)`, ktorá predpokladá, že daný súbor má v každom riadku po jednom celom čísle. Funkcia vráti číselnú hodnotu z tohto riadka a ak sa to nedá (napr. súbor neexistuje, nemá dosť riadkov, nie je v tomto riadku iba jediná celočíselná hodnota), vráti hodnotu tretieho parametra `inak`.

   - napr.

```
>>> daj_cislo('cvicenie.py', 17, 'neviem')
'neviem'
```

   - nezabudnite zatvoriť otvorený súbor

8. Napíšte funkciu `sustavy(retazec)`, ktorá sa pokúsi daný reťazec previesť na číslo v rôznych číselných sústavách. Využite to, že štandardná funkcia `int()` môže byť zavolaná aj s druhým parametrom - číselnou sústavou, napr. `int('ff', 16)` vráti 255, t.j. `'ff'` je v 16-ovej sústave číslo 255. Funkcia `sustavy()` vráti 17-prvkový zoznam, pričom i-ty prvok zoznamu obsahuje prevod daného reťazca na číslo v i-sústave. Ak sa to pre nejakú sústavu urobiť nedá, prvok zoznamu na danom mieste bude mať hodnotu `None`.

   - napr.

```
>>> sustavy('11')
[11, None, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
>>> sustavy('1a1')
[None, None, None, None, None, None, None, None, None, None, 232, 265,␣
↪300, 337, 376, 417]
>>> sustavy('FF')
[None, None, None, None, None, None, None, None, None, None, None, None,
↪ None, None, None, 255]
>>> sustavy('x')
[None, None, None, None, None, None, None, None, None, None, None, None,
↪ None, None, None, None]
```

9. Napíšte funkciu `ako(hodnota1, hodnota2)`, ktorá najprv zistí typ prvého parametra `hodnota1` a potom sa pokúsi pretypovať na tento typ druhý parameter `hodnota2` (zrejme volaním `typ(hodnota2)`). Ak sa toto pretypovanie úspešne podarí, funkcia vráti túto pretypovanú hodnotu, inak vráti `None`.

   - napr.

```
>>> ako('a', 123)
'123'
>>> ako(3.1, 123)
123.0
>>> ako((), '123')
('1', '2', '3')
>>> print(ako((), 123))
None
```

10. Napíšte funkciu `sucet(zoznam)`, ktorá spočíta všetky prvky daného zoznamu. Pracovať bude tak, že najprv zoberie prvý prvok zoznamu a ten priradíme do `vysl` ako momentálny výsledok - k nemu budete postupne pripočítavať (použijete operáciu +) ďalšie prvky zoznamu. Lenže tieto ďalšie prvky v zozname nemusia byť rovnakého typu ako prvý prvok a preto ich budete postupne pretypovávať na rovnaký typ a až pri úspešnom pretypovaní ich pripočítame k výsledku `vysl`, inak ich idignorujete.

    - napr.

```
>>> sucet([1, 2, 3.1, '4'])
10
>>> sucet([1., 2, 3.1, '4'])
10.1
>>> sucet([('p', 'y'), 'tho', ['n']])
('p', 'y', 't', 'h', 'o', 'n')
>>> sucet([[1], (2, 3), 4])
[1, 2, 3]
```

KAPITOLA 18

18. Polymorfizmus

Ešte raz zopakujme, aké sú najdôležitejšie vlastnosti objektového programovania:

- **zapuzdrenie** (encapsulation)
- **dedičnosť** (inheritance)
- **polymorfizmus** (polymorphism)

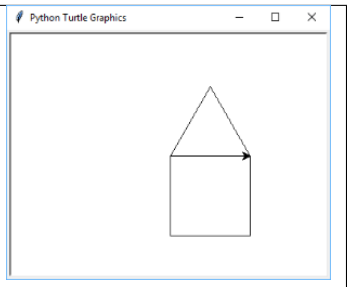
V dnešnej téme sa sústredíme na poslednú vlastnosť **polymorfizmus**.

Vráťme sa k príkladu, v ktorom korytnačka kreslila domček:

```
from turtle import Turtle

class MojaTurtle(Turtle):
    def domcek(self, dlzka):
        for uhol in 90, 90, 90, 30, 120, -60:
            self.fd(dlzka)        # fd z triedy Turtle
            self.rt(uhol)         # rt z triedy Turtle

t = MojaTurtle()
t.domcek(100)
```

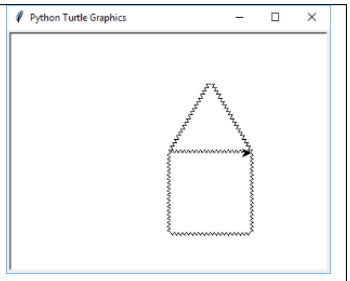


V metóde `domcek()` predpokladáme, že pri kreslení domčeka inštancia `t` triedy `MojaTurtle` použije zdedenú metódu `fd()` (z triedy `Turtle`) a tiež zdedenú metódu `rt()` z triedy `Turtle`.

Z triedy `MojaTurtle` sme **odvodili** novú triedu `MojaTurtle1`. Táto nová trieda teda zdedila od svojej základnej triedy `MojaTurtle` všetko okrem metódy `fd()`, ktorú **prekryla** (override) svojou vlastnou verziou tejto metódy (tento `fd()` kreslí cikcakové čiary):

```
class MojaTurtle1(MojaTurtle):
    def fd(self, dlzka):
        while dlzka >= 5:
            self.lt(60)
            super().fd(5)
            self.rt(120)
            super().fd(5)
            self.lt(60)
            dlzka -= 5
        super().fd(dlzka)

t = MojaTurtle1()
t.speed(0)
t.domcek(100)
```



Takže teraz:

- inštancia triedy `MojaTurtle` pomocou metódy `domcek()` nakreslí domček zo 6 riadnych úsečiek
- inštancia triedy `MojaTurtle1` pomocou metódy `fd()` kreslí cikcakové čiary
- táto inštancia triedy `MojaTurtle1` bude takýmito cikcakovými čiarami kresliť aj domček (volaním metódy `domcek()`)

Ako je to možné? Veď predsa v metóde `domcek()`, keď sme ju definovali, sme jasne zapísali, že kreslenie čiar `fd()` sa bude robiť tak, ako bolo definované v základnej triede `Turtle`. Nikde sme tu nijako nezaznačovali (ani nás to vtedy nenapadlo), že tento `fd()` niekto v budúcnosti možno nahradí svojou vlastnou verziou metódy (napr. cikcak). Tak práve tomuto mechanizmu sa hovorí **polymorfizmus** a označuje:

- keď Python vykonáva nejakú metódu (napr. `domcek()`), tak sa toto vykonávanie **prispôsobí** (adaptuje) tej inštancii, ktorá túto metódu zavolala
- vždy sa použijú aktuálne verzie metód objektu, pre ktorý sa niečo vykonáva
- funguje to aj spätne, teda vo všetkých zdedených metódach: ak sa v nich nachádza volanie niečoho, čo sme práve prekryli svojou novou verziou, tak sa to naozaj uplatní

Uvedomte si, čo by sa stalo, keby tu nefungoval polymorfizmus:

- metóda `domcek()` by vždy kreslila úplne rovnaký domček z rovných čiar bez ohľadu na to, kto túto metódu zavolal (kto bol `self`)
- keby sme potrebovali domček z cikcakových čiar aj pre objekt typu `MojaTurtle1`, museli by sme túto metódu skopírovať aj do tejto triedy, hoci dedičnosť nám hovorí, že by sme to nemali robiť

Pridajme k týmto dvom triedam aj `MojaTurtle2`, pomocou ktorej korytnačka po každej kreslenej čiare prešla trikrát:

```
from turtle import Turtle, delay
from random import randint as ri

class MojaTurtle(Turtle):
    def domcek(self, dlzka):
        for uhol in 90, 90, 90, 30, 120, -60:
            self.fd(dlzka)        # fd z triedy Turtle
            self.rt(uhol)         # rt z triedy Turtle

class MojaTurtle1(MojaTurtle):
    def fd(self, dlzka):
        while dlzka >= 5:
            self.lt(60)
            super().fd(5)
            self.rt(120)
            super().fd(5)
            self.lt(60)
            dlzka -= 5
        super().fd(dlzka)

class MojaTurtle2(MojaTurtle):
    def fd(self, dlzka):
        super().fd(dlzka)
        self.rt(180 - ri(-3, 3))
        super().fd(dlzka)
        self.rt(180 - ri(-3, 3))
        super().fd(dlzka)
```

Ďalej vytvoríme 100-prvkový zoznam korytnačiek, pričom pre každú z nich sa náhodne rozhodneme, aký typ vyberieme:

```
delay(0)
zoz = []
moja = [MojaTurtle, MojaTurtle1, MojaTurtle2]
for i in range(100):
    t = moja[ri(0, 2)]()
    t.ht()
    t.speed(0)
    t.pu()
    t.setpos(ri(-300, 250), ri(-250, 250))
    t.pd()
    zoz.append(t)

for t in zoz:
    t.domcek(50)
```

Vytvorili sme tu zoznam korytnačiek troch rôznych typov. Každá z korytnačiek dokáže nakresliť `domcek()` ale každá to robí po svojom. Keďže tento zoznam obsahuje inštancie rôznych typov, niekedy hovoríme, že je to tzv. **polymorfný zoznam** (prípadne polymorfné pole).

V Pythone ale nie je problém so zoznamami, resp. poľami, ktorých prvky sú rôznych typov. Toto ale nie je bežné v iných programovacích jazykoch (Pascal, C++, ...), kde väčšinou určujeme nejaký jeden konkrétny typ ako typ všetkých prvkov poľa (napr. pole celých čísel, pole reťazcov, ...). Aj v týchto jazykoch sa dá vytvárať polymorfné pole, ale už to nebude také jednoduché ako v Pythone.

Toto ale nie sú jediné významy polymorfizmu - tento pojem sa objavuje na mnohých miestach aj v situáciách, s ktorými sme sa zoznámili dávnejšie a už sme sa zmierili s takýmto správaním Pythonu. Pripomeňme si triedu `Cas` z 15. prednášky:

```
class Cas:

    def __init__(self, hodiny=0, minuty=0, sekundy=0):
        self.sek = abs(3600*hodiny + 60*minuty + sekundy)

    def __str__(self):
        return '{}:{:02}:{:02}'.format(self.sek//3600, self.sek//60%60, self.sek%60)

    def sucet(self, iny):
        return Cas(sekundy=self.sek+iny.sek)

    def rozdiel(self, iny):
        return Cas(sekundy=self.sek-iny.sek)

    def mensi(self, iny):
        return self.sek < iny.sek

    def rovny(self, iny):
        return self.sek == iny.sek
```

Tu vidíme použitie aj magickej metódy `__str__()`:

```
>>> c1 = Cas(10, 22, 30)
>>> c1.__str__()
'10:22:30'
>>> c2 = Cas(4, 55, 18)
>>> str(c2)
'4:55:18'
>>> print('sucet =', c1.sucet(c2))
sucet = 15:17:48
```

Už vieme, že `c1.__str__()` priamo zavolá metódu `__str__()`, teda vráti reťazcovú reprezentáciu hodnoty čas. Volanie `str(c2)` tiež zavolá `__str__()`, ale neurobí sa to priamo, ale cez nejaký „magický“ mechanizmus:

- štandardná funkcia `str()` má za úlohu ľubovoľnú Pythonovskú hodnotu (napr. číslo, zoznam, n-ticu, ...) vyjadriť ako znakový reťazec
- keďže túto štandardnú funkciu naprogramovali vo firme „Python“ pred veľa rokmi, nemohli vtedy myslieť aj na to, že v roku 2017 niekto zadefinuje vlastný typ `Cas` a bude ho potrebovať pomocou `str(c2)` previesť na znakový reťazec
- preto má táto štandardná funkcia v sebe skrytý mechanizmus, pomocou ktorého veľmi jednoducho zistí reťazcovú reprezentáciu ľubovoľného typu: namiesto toho aby sama vyrábala znakový reťazec, zavolá metódu `__str__()` danej hodnoty; pritom každá trieda má vždy zadefinovanú náhradnú verziu tejto metódy, ktorá (keď ju neprekryjeme vlastnou metódou) vypisuje známe `'<__main__.Cas object at 0x035B92D0>'`

Štandardná funkcia `print()`, ktorá má za úlohu vypísať všetky svoje parametre, najprv všetky neznakové parametre prevedie na znakové reťazce pomocou štandardnej funkcie `str()` z nich vyrobí reťazce a tieto vypíše.

Takže aj prevod hodnoty typu `Cas` na znakový reťazec pomocou štandardnej funkcie `str()` funguje vďaka **polymorfizmu**: aj táto funkcia sa prispôsobí (adaptuje) k zadanému typu a snaží sa z neho získať reťazec volaním jeho metódy `__str__()`.

18.1 Operátorový polymorfizmus

Už máme skúsenosti s tým, že napr. operácia + funguje nielen s číslami ale aj s reťazcami a zoznamami:

```
>>> 12 + 34
46
>>> 'Pyt' + 'hon'
Python
>>> [1, 2] + [3, 4, 5]
[1, 2, 3, 4, 5]
>>> 12 + '34'
...
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Hovoríme tomu **operátorový polymorfizmus**, lebo táto operácia funguje rôzne pre rôzne typy. Python sa v tomto prípade nemusí pre každú dvojicu typov rozhodovať, či ich súčet je realizovateľný alebo je to chyba `TypeError`. Jednoducho prvému operandu oznámi, aby pripočítal druhý operand, t.j. zavolá nejakú jeho metódu a pošle mu druhý operand. Tou metódou je samozrejme magická metóda `__add__()` a preto pri vyhodnocovaní súčtu Python vlastne volá magickú metódu:

```
>>> 12 + 34
46
>>> (12).__add__(34)    # 12 tu musí byť v zátvorkách
46
>>> 'Pyt' + 'hon'
Python
>>> 'Pyt'.__add__('hon')
Python
>>> (12).__add__('34')
NotImplemented
```

Tiež si uvedomte, že `a.__add__(b)` pre `a` napr. celé číslo je to isté ako `int.__add__(a, b)`. Práve táto metóda je zodpovedná za to, či a ako sa dá k celému číslu pripočítať hodnota nejakého iného typu.

Teraz už vieme, že keď v Pythone zapíšeme `a + b`, v skutočnosti sa volá metóda `a.__add__(b)` a preto aj pre našu triedu `Cas` stačí dodefinovať túto metódu, teda vlastne stačí len premenovať `sucet()` na `__add__()`. Vyskúšajme:

```
class Cas:

    ...

    def __add__(self, iny):
        return Cas(sekundy=self.sek+iny.sek)

    ...

c1 = Cas(10, 22, 30)
c2 = Cas(4, 55, 18)
print('sucet =', c1 + c2)
```

a vidíme, že to naozaj funguje. Zrejme na rovnakom princípe fungujú nielen všetky aritmetické operácie ale aj relačné operátory:

Tabuľka 1: aritmetické operácie

metóda	operácia	
`x.__add__(y)`	`x + y`	
`x.__sub__(y)`	`x - y`	
`x.__mul__(y)`	`x * y`	
`x.__truediv__(y)`	`x / y`	
`x.__floordiv__(y)`	`x // y`	
`x.__mod__(y)`	`x % y`	
`x.__pow__(y)`	`x ** y`	
`x.__neg__()`	`- x`	

Tomuto sa hovorí **preťažovenie operátorov** (operator overloading): existujúca operácia dostáva pre našu triedu nový význam, t.j. prekryli sme štandardné správanie Pythonu, keď niektoré operácie pre neznáme operandy hlásia chybu. Stretnete sa s tým aj v iných programovacích jazykoch.

Tabuľka 2: relačné operácie

metóda	relácia	
`x.__eq__(y)`	`x == y`	
`x.__ne__(y)`	`x != y`	
`x.__lt__(y)`	`x < y`	
`x.__le__(y)`	`x <= y`	
`x.__gt__(y)`	`x > y`	
`x.__ge__(y)`	`x >= y`	

Teraz môžeme vylepšiť kompletnú triedu `Cas`:

```
class Cas:

    def __init__(self, hodiny=0, minuty=0, sekundy=0):
        self.sek = abs(3600*hodiny + 60*minuty + sekundy)

    def __str__(self):
        return '{}:{:02}:{:02}'.format(self.sek//3600, self.sek//60%60, self.sek%60)

    def __add__(self, iny):
        return Cas(sekundy=self.sek+iny.sek)

    def __sub__(self, iny):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        return Cas(sekundy=self.sek-iny.sek)

    def __lt__(self, iny):
        return self.sek < iny.sek

    def __eq__(self, iny):
        return self.sek == iny.sek
```

Vďaka tomuto môžeme časy nielen sčitovať ale aj odčitovať a porovnávať relačnými operátormi.

Pozrime si ešte takúto funkciu:

```
def sucet(a, b):
   return a + b
```

Zrejme táto funkcia bude dávať správne výsledky pre rôzne typy parametrov, môžeme im hovoriť **polymorfné parametre** a niekedy sa stretnete aj s pojmom **parametrický polymorfizmus**.

18.1.1 Trieda Zlomok

Na 14. cvičeniach ste riešili aj úlohu, v ktorej ste definovali triedu `Zlomok` aj s metódami `str()` a `float()`. My toto riešenie trochu vylepšíme:

```
class Zlomok:
    def __init__(self, citatel=0, menovatel=1):
        self.cit = citatel
        self.men = menovatel

    def __str__(self):
        return '{}/{}'.format(self.cit, self.men)

    def __int__(self):
        return self.cit // self.men

    def __float__(self):
        return self.cit / self.men
```

a jednoduchý test:

```
>>> z1 = Zlomok(3, 8)
>>> z2 = Zlomok(2, 4)
>>> print('desatinne cislo z', z1, 'je', float(z1))
desatinne cislo z 3/8 je 0.375
>>> print('cela cast', z2, 'je', int(z2))
cela cast 2/4 je 0
```

Magické metódy `__int__()` a `__float__()` slúžia na to, aby sme objekt typu `Zlomok` mohli poslať do konvertovacích funkcií `int()` a `float()`.

Tento dátový typ by mohol byť naozaj užitočný, keby obsahoval aj nejaké operácie. S týmto už máme nejaké skúsenosti z definovania triedy `Cas`. Tiež by bolo veľmi vhodné, keby sa v tejto triede zlomok automaticky upravil na základný tvar. Túto úpravu budeme robiť v inicializácii `__init__()`: z matematiky na základnej škole vieme, že na to potrebujeme zistiť **najväčší spoločný deliteľ**. Použijeme známy Euklidov algoritmus:

```
def nsd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

Ak budeme túto funkciu potrebovať len v metóde `__init__()`, nemusíme ju definovať ako globálnu funkciu, ale ju prenesieme do tela inicializačnej funkcie, čím z nej urobíme lokálnu funkciu (vidí ju len samotná metóda `__init__()`). Všimnite si, že sme sem doplnili niekoľko zatiaľ neznámych magických metód:

```
class Zlomok:

    def __init__(self, citatel=0, menovatel=1):

        def nsd(a, b):
            while b != 0:
                a, b = b, a % b
            return a

        if menovatel == 0:
            menovatel = 1
        delitel = nsd(citatel, menovatel)
        self.cit = citatel // delitel
        self.men = menovatel // delitel

    def __str__(self):
        return '{}/{}'.format(self.cit, self.men)

    __repr__ = __str__

    def __add__(self, iny):
        if isinstance(iny, int):
            c, m = iny, 1
        else:
            c, m = iny.cit, iny.men
        return Zlomok(self.cit*m+self.men*c, self.men*m)

    __radd__ = __add__

    def __sub__(self, iny):
        if isinstance(iny, int):
            c, m = iny, 1
        else:
            c, m = iny.cit, iny.men
        return Zlomok(self.cit*m-self.men*c, self.men*m)

    def __rsub__(self, iny):
        if isinstance(iny, int):
            c, m = iny, 1
        else:
            c, m = iny.cit, iny.men
        return Zlomok(self.men*c-self.cit*m, self.men*m)


    def __mul__(self, iny):
        if isinstance(iny, int):
            c, m = iny, 1
        else:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            c, m = iny.cit, iny.men
        return Zlomok(self.cit*c, self.men*m)

    __rmul__ = __mul__

    def __abs__(self):
        return Zlomok(abs(self.cit), self.men)

    def __int__(self):
        return self.cit // self.men

    def __float__(self):
        return self.cit / self.men

    def __lt__(self, iny):
        return self.cit*iny.men < self.men*iny.cit

    def __eq__(self, iny):
        return self.men==iny.men and self.cit==iny.cit
```

Niekoľko noviniek v tomto kóde:

- atribút `__repr__` je tu definovaný pomocou priradenia `__repr__ = __str__` a znamená:
  - aj `__repr__` bude metódou triedy `Zlomok` a týmto sme ju definovali ako identickú k `__str__` (triedny atribút `__repr__` obsahuje rovnakú referenciu ako `__str__`, teda obsahuje rovnakú definíciu metódy)
  - magická metóda `__repr__` špecifikuje, čo sa bude vypisovať, ak inštanciu zadáme priamo v shelli alebo sa objaví pri vypisovaní prvkov zoznamu, napr.

```
>>> z = Zlomok(1, 3)
>>> z
1/3
>>> zoznam = [Zlomok(1, 5), Zlomok(2, 5), Zlomok(3, 5), Zlomok(4, 5)]
>>> zoznam
[1/5, 2/5, 3/5, 4/5]
```

- magická metóda `__radd__` (jej definícia je identická s `__add__`) je potrebná v situáciách, keď chceme sčitovať celé číslo so zlomkom:
  - samotná `__add__` zvláda sčítať len zlomok s číslom (súčet `Zlomok(1, 3) + 1` označuje volanie `Zlomok(1, 3).__add__(1)`)
  - sčitovanie čísla so zlomkom `1 + Zlomok(1, 3)` označuje `(1).__add__(Zlomok(1, 3))`, čo by znamenalo, že metóda `__add__` triedy `int` by mala vedieť sčitovať aj zlomky (je nemožné predefinovať štandardnú metódu `int.__add__()` aby fungovala s nejakým divným typom)
  - preto pri sčitovaní `1 + Zlomok(1, 3)`, keď Python zistí, že nefunguje `(1).__add__(Zlomok(1, 3))`, vyskúša vymeniť operandy operácie a namiesto `__add__()` zavolať `__radd__()`
- podobne je definovaná aj metóda `__rmul__`, pričom odčitovanie `__rsub__` nemôže byť identická funkcia s metódou `__sub__`, preto je zadefinovaná zvlášť
- pridali sme magickú metódu `__abs__()`, vďaka ktorej bude fungovať aj štandardná funkcia `abs(zlomok)`

Uvedomte si, že všetky nami definované metódy triedy `Zlomok` (okrem `__init__()`) sú **pravé funkcie** a preto aj náš nový typ `Zlomok` môžeme považovať za nemeniteľný (immutable).

Vďaka relačným operátorom `__lt__()` a `__eq__()` a schopnosti sčitovať zlomky s číslami bude fungovať aj takáto ukážka:

```
>>> zoznam = []
>>> for m in range(2, 8):
        for c in range(1, m):
            zoznam.append(Zlomok(c, m))

>>> zoznam
[1/2, 1/3, 2/3, 1/4, 1/2, 3/4, 1/5, 2/5, 3/5, 4/5, 1/6, 1/3, 1/2, 2/3, 5/6, 1/7, 2/7,␣
↪3/7, 4/7, 5/7, 6/7]
>>> min(zoznam)
1/7
>>> max(zoznam)
6/7
>>> sum(zoznam)
21/2
>>> sorted(zoznam)
[1/7, 1/6, 1/5, 1/4, 2/7, 1/3, 1/3, 2/5, 3/7, 1/2, 1/2, 1/2, 4/7, 3/5, 2/3, 2/3, 5/7,␣
↪3/4, 4/5, 5/6, 6/7]
```

18.1.2 Typ množina

Na 14. cvičeniach ste riešili aj príklad s triedou `Zoznam`, pomocou ktorej sa uchovávali nejaké texty v zozname. Tu je možné riešenie:

```
class Zoznam:
    def __init__(self):
        self.zoznam = []

    def __str__(self):
        p = []
        for prvok in self.zoznam:
            p.append(str(prvok))
        return ', '.join(p)

    def pridaj(self, prvok):
        if prvok not in self.zoznam:
            self.zoznam.append(prvok)

    def vyhod(self, prvok):
        if prvok in self.zoznam:
            self.zoznam.remove(prvok)

    def je_v_zozname(self, prvok):
        return prvok in self.zoznam

    def pocet(self):
        return len(self.zoznam)
```

Jednoduchý test:

```
z = Zoznam()
z.pridaj('behat')
z.pridaj('upratat')
z.pridaj('ucit sa')
if z.je_v_zozname('behat'):
    print('musis behat')
else:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    print('nebehaj')
z.pridaj('upratat')
print('zoznam =', z)
z.vyhod('spievat')
print('pocet prvkov v zozname =', z.pocet())
```

```
musis behat
zoznam = behat, upratat, ucit sa
pocet prvkov v zozname = 3
```

V Pythone je zaužívané použiť operáciu `in` vtedy, keď potrebujeme zistiť, či sa v nejakej postupnosti hodnôt nachádza nejaká konkrétna hodnota, napr.

```
>>> 3 in [1, 2, 3, 4, 5]
True
>>> 'x' in 'Python'
False
```

Zrejme by bolo prirodzené, keby sme aj našu metódu `je_v_zozname()` vedeli prerobiť na pythonovský štýl (pythonic). Aj na toto existuje magická metóda `__contains__()` a predchádzajúce dva príklady sú vlastne krajšími zápismi (tzv. *syntactic sugar*) pre:

```
>>> [1, 2, 3, 4, 5].__contains__(3)
True
>>> 'Python'.__contains__('x')
False
```

Podobne aj štandardná funkcia `len()`, ktorá vie zistiť počet prvkov zoznamu alebo dĺžku reťazca (počet znakov v reťazci), využíva polymorfizmus, teda v skutočnosti, aby zistila počet prvkov nejakej štruktúry, sa jej na to opýta pomocou magickej metódy `__len__()`. Preto nasledovné trojice príkazov robia to isté:

```
>>> len([1, 2, 3, 4, 5])
5
>>> [1, 2, 3, 4, 5].__len__()
5
>>> list.__len__([1, 2, 3, 4, 5])
5

>>> len('Python')
6
>>> 'Python'.__len__()
6
>>> str.__len__('Python')
6
```

Upravme aj našu triedu `Zoznam`, pričom premenujeme aj metódy `pridaj()` a `vyhod()` na anglické ekvivalenty:

```
class Zoznam:
    def __init__(self):
        self.zoznam = []

    def __str__(self):
        p = []
        for prvok in self.zoznam:
            p.append(str(prvok))
        return ', '.join(p)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def __contains__(self, prvok):
        return prvok in self.zoznam

    def __len__(self):
        return len(self.zoznam)

    def add(self, prvok):
        if prvok not in self.zoznam:
            self.zoznam.append(prvok)

    def discard(self, prvok):
        if prvok in self.zoznam:
            self.zoznam.remove(prvok)
```

Otestujeme rovnako ako predtým:

```
z = Zoznam()
z.add('behat')
z.add('upratat')
z.add('ucit sa')
if 'behat' in z:
    print('musis behat')
else:
    print('nebehaj')
z.add('upratat')
print('zoznam =', z)
z.discard('spievat')
print('pocet prvkov v zozname =', len(z))
```

```
musis behat
zoznam = behat, upratat, ucit sa
pocet prvkov v zozname = 3
```

Uvedomte si, že do takéhoto zoznamu nemusíme vkladať len znakové reťazce, ale rovnako by fungoval aj pre ľubovoľné iné typy hodnôt. Tento typ je vlastne jednoduchá realizácia matematickej množiny hodnôt: každý prvok sa tu môže nachádzať maximálne raz.

18.2 Štandardný typ množina - set

Python má medzi štandardnými typmi aj typ **množina**, ktorý má v Pythone meno `set`. Podobne ako aj iné typy `str`, `list` a `tuple` aj tento množinový typ je postupnosťou hodnôt, ktorú môžeme prechádzať for-cyklom alebo ju poslať ako parameter pri konštruovaní iného typu. Napr.

```
>>> mnozina = {'behat', 'ucit sa', 'upratat'}
>>> mnozina
{'upratat', 'behat', 'ucit sa'}
>>> zoznam = list(mnozina)
>>> zoznam
['upratat', 'behat', 'ucit sa']
>>> ntica = tuple(mnozina)
>>> ntica
('upratat', 'behat', 'ucit sa')
>>> for prvok in mnozina:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        print(prvok, end=', ')

'upratat', 'behat', 'ucit sa',
```

Podobne ako vieme skonštruovať zoznam pomocou generátora postupnosti `range()`, vieme to urobiť aj s množinami:

```
>>> list(range(7))
[0, 1, 2, 3, 4, 5, 6]
>>> set(range(7))
{0, 1, 2, 3, 4, 5, 6}
```

alebo vytvorenie zoznamu a množiny zo znakového reťazca:

```
>>> list('mama ma emu')
['m', 'a', 'm', 'a', ' ', 'm', 'a', ' ', 'e', 'm', 'u']
>>> set('mama ma emu')
{' ', 'm', 'u', 'a', 'e'}
```

Štandardný Pythonovský typ **set** má kompletnú sadu množinových operácií a veľa užitočných metód. Pre prvky množiny ale platí, že to nemôžu byť ľubovoľné hodnoty, ale musia to byť nemenné typy (immutable), napr. čísla, reťazce, n-tice.

Predchádzajúci príklad, v ktorom sme definovali triedu `Zoznam` vieme prepísať l s použitím pythonovských množín napr. takto:

```
z = set()                   # prázdna pythonovská množina
z.add('behat')
z.add('upratat')
z.add('ucit sa')
if 'behat' in z:
    print('musis behat')
else:
    print('nebehaj')
z.add('upratat')
print('zoznam =', z)
z.discard('spievat')
print('pocet prvkov v zozname =', len(z))
```

```
musis behat
zoznam = {behat, upratat, ucit sa}
pocet prvkov v zozname = 3
```

18.2.1 Operácie a metódy s množinami

Štandardný typ množina (`set`) je meniteľný (**mutable**) a z toho vyplývajú všetky dôsledky podobne ako pre pythonovské zoznamy (`list`).

V ďalších tabuľkách predpokladáme, že `M`, `M1` a `M2` sú nejaké množiny:

Tabuľka 3: množinové operácie

	popis
`M1 | M2`	zjednotenie dvoch množín
`M1 & M2`	prienik dvoch množín
`M1 - M2`	rozdiel dvoch množín
`M1 ^ M2`	vylučovacie zjednotenie dvoch množín
`M1 == M2`	dve množiny majú rovnaké prvky
`M1 is M2`	dve množiny sú identické štruktúry v pamäti (je to tá istá hodnota)
`M1 < M2`	M1 je podmnožinou M2 (funguje aj pre zvyšné relačné operátory)
`prvok in M`	zistí, či prvok patrí do množiny
`prvok not in M`	zistí, či prvok nepatrí do množiny
`for prvok in M: ...`	cyklus, ktorý prechádza cez všetky prvky množiny

Tabuľka 4: štandardné funkcie

	popis
`len(M)`	počet prvkov
`min(M)`	minimálny prvok (ale všetky prvky sa musia dať navzájom porovnávať)
`max(M)`	maximálny prvok (ale všetky prvky sa musia dať navzájom porovnávať)
`list(M)`	vráti neusporiadaný zoznam prvkov z množiny
`sorted(M)`	vráti usporiadaný zoznam (ale všetky prvky sa musia dať navzájom porovnávať)

Tabuľka 5: niektoré metódy

	popis
`M.add(prvok)`	pridá prvok do množiny (ak už v množine bol, neurobí nič)
`M.remove(prvok)`	vyhodí daný prvok z množiny (ak neexistuje, vyhlási chybu)
`M.discard(prvok)`	vyhodí daný prvok z množiny (ak neexistuje, neurobí nič)
`M.pop()`	vyhodí nejaký neurčený prvok z množiny a vráti jeho hodnotu (ak je množina prázdna, vyhlási chybu)
`M.clear()`	vyčistí množinu

Všetky tieto metódy sú **mutable**, teda zmenia obsah premennej a ak je na ňu viac referencií, ovplyvní to všetky.

Metód, ktoré pracujú s množinami, je oveľa viac.

18.2.2 Vytvorenie množiny

Tabuľka 6: niektoré metódy

	popis
`M = set()`	vytvorí prázdnu množinu
`M = {hodnota, hodnota, ...}`	vytvorí neprázdnu množinu so zadanými prvkami
`M = set(zoznam)`	so zadaného zoznamu vytvorí množinu
`M = set(M1)`	vytvorí kópiu množiny `M1`

Uvedomte si, že niektoré situácie vieme riešiť rôznymi spôsobmi, napr.

- pridať `prvok` do množiny `mnoz`:

```
mnoz.add(prvok)          # mutable
```

alebo:

```
mnoz = mnoz | {prvok}    # immutable
```

čo je to isté ako:

```
mnoz |= {prvok}          # mutable
```

- vyhodiť jeden `prvok` z množiny `mnoz`:

```
mnoz.discard(prvok)      # mutable
```

alebo:

```
mnoz = mnoz - {prvok}    # immutable
```

čo je to isté ako:

```
mnoz -= {prvok}          # mutable
```

ak máme istotu, že prvok je v množine (inak to spadne na chybe):

```
mnoz.remove(prvok)       # mutable
```

- zistí, či je množina `mnoz` prázdna:

```
mnoz == set()
```

alebo:

```
len(mnoz) == 0
```

alebo veľmi nečitateľne:

```
not mnoz
```

18.2.3 Príklady s množinami

Napíšme funkciu, ktorá vráti počet rôznych samohlások v danom slove:

```
def pocet_samohlasok(slovo):
    return len(set(slovo) & set('aeiouy'))
```

Bez použitia množiny by sme ju zapísali asi takto:

```
def pocet_samohlasok(slovo):
    vysl = 0
    for znak in 'aeiouy':
        if znak in slovo:
            vysl += 1
    return vysl
```

Ďalšia funkcia skonštruuje množinu s takouto vlastnosťou:

- `1` patrí do množiny
- ak do množiny patrí nejaké `i`, tak tam patrí aj `2*i+1` aj `3*i+1`

Funkcia vytvorí všetky prvky množiny ktoré nie sú väčšie ako zadané `n`:

```
def urob(n):
    m = {1}
    for i in range(n // 2):
        if i in m:
            if 2 * i + 1 <= n:
                m.add(2 * i + 1)
            if 3 * i + 1 <= n:
                m.add(3 * i + 1)
    return m
```

Ďalšia funkcia je rekurzívna a zisťuje, či nejaké dané `i` je prvkom množiny z predchádzajúceho príkladu (bez toho, aby sme museli túto množinu najprv skonštruovať):

```
def test(i):
    if i == 0:
        return False
    if i == 1:
        return True
    if i % 2 == 1 and test(i // 2):
        return True
    if i % 3 == 1 and test(i // 3):
        return True
    return False
```

To isté sa dá zapísať trochu úspornejšie (a výrazne menej čitateľne):

```
def test(i):
    if i <= 1:
        return i == 1
    return i % 2 == 1 and test(i // 2) or i % 3 == 1 and test(i // 3)
```

Pomocou funkcie `test()` vieme zapísať aj funkciu `urob()`:

```
def urob(n):
    m = set()
    for i in range(n+1):
        if test(i):
            m.add(i)
    return m
```

Ďalšia funkcia skonštruuje množinu prvočísel pomocou algoritmu Eratostenovo sito:

- zoberieme zoznam všetkých celých čísel od 2 po nejaké zadané `n`
- prvé číslo v zozname `2` je prvočíslo, zo zoznamu odstránime všetky jeho násobky
- druhé číslo v tomto novom zozname `3` je tiež prvočíslo, zo zoznamu odstránime všetky jeho násobky
- aj tretie číslo v tomto novom zozname `5` je prvočíslo, zo zoznamu odstránime všetky jeho násobky
- takto to budeme opakovať, kým neprejdeme celý zoznam čísel

Zapíšme to ako funkciu::

```
def eratostenovo_sito(n):
    mnozina = set(range(2, n + 1))
    for i in range(n):
        if i in mnozina:
            mnozina -= set(range(i + i, n + 1, i))
    return mnozina
```

```
>>> eratostenovo_sito(100)
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
 73, 79, 83, 89, 97}
```

Python nezaručuje, že prvky v množine sú v rastúcej postupnosti. Preto niekedy množinu prevedieme na usporiadaný zoznam, napr.

```
>>> m1 = {1,5,10,30,100,1000}
>>> m2 = {2,6,11,31,101,1001}
>>> m1 | m2
{1, 2, 100, 5, 101, 6, 1000, 1001, 10, 11, 30, 31}
>>> sorted(m1 | m2)
[1, 2, 5, 6, 10, 11, 30, 31, 100, 101, 1000, 1001]
```

18.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Aritmetické operácie sú vnútorne realizované cez volania magických metód, napr. `x-2` ako `x.__sub__(2)`.
   - prepíšte tento výraz tak, aby sa všetky oparácie nahradili volaniami metód:

     ```
     (x * 7 - 8 ) / 4
     ```

     skontrolujte váš zápis na správnosť dosadením, napr.

```
>>> x = 5
>>> (x * 7 - 8 ) / 4
6.75
```

- prepíšte nasledovný výraz tak, aby v ňom neboli volania magických metód:

```
>>> (6).__sub__((5).__mul__(4)).__add__((3).__pow__(2))
-5
```

  po vyhodnotení by ste mali dostať rovnaký výsledok

2. Triedu `Cas` z prednášky doplňte tak, aby operácie sčitovania a odčitovania fungovali aj s celými číslami (pripočítava, resp. odpočítava sekundy) alebo aj s n-ticami (prvým prvkom sú hodiny, druhým minúty a tretím sekundy)

   - napr.

```
>>> c = Cas(8, 10, 34)
>>> print(c + 640)
8:21:14
>>> print(c + (1, 55))
10:05:34
>>> print(c - 100)
8:08:54
```

3. Pomocou modulu `time` a funkcie vieme zistiť momentálny čas v počítači.

   - napr.

```
>>> import time
>>> time.localtime()
time.struct_time(tm_year=2017, tm_mon=11, tm_mday=22, tm_hour=8, tm_min=26,␣
↪tm_sec=12,
tm_wday=1, tm_yday=327, tm_isdst=0)
>>> time.localtime()[3:6]
(8, 26, 24)
```

   Napíšte funkciu `Teraz()`, ktorá vráti inštanciu triedy `Cas` s momentálnym časom.

   - napr.

```
>>> c = Teraz()
>>> print(type(c), c)
<class '__main__.Cas'> 8:34:07
```

4. Vytvorte zoznam rôznych časov (napr. ich generujte náhodným generátorom). Otestujte, či funguje triedenie pomocou štandardnej funkcie `sorted()`.

   - napr.

```
>>> zoznam = [Cas(20, 15), Cas(7), ...]
>>> zoznam1 = sorted(zoznam)
>>> # vypíš zoznam1
```

18.3.1 Množiny - set

5. Napíšte funkciu `neparne(n)`, ktorá vráti množinu všetkých nepárnych čísel menších ako `n`.

- napr.

```
>>> m = neparne(20)
>>> type(m)
<class 'set'>
>>> m
{1, 3, ...
```

6. Napíšte funkciu `samohlasky(veta)`, ktorá vráti množinu samohlások v danej vete.

   - napr.

```
>>> samohlasky('mama ma emu')
{'e', 'u', 'a'}
>>> samohlasky('strc prst skrz krk')
set()
```

7. Napíšte funkciu `slova(meno_suboru)`, ktorá z textového súboru vytvorí množinu slov (slová sú navzájom oddelené medzerami). Funkcia túto množinu vráti ako svoj výsledok.

   - napr.

```
>>> mn = slova('text1.txt')
>>> mn
{...}
```

8. Napíšte funkciu `vyrob(n)`, ktorá vytvorí (a vráti) množinu celých čísel, ktoré sú menšie ako `n`, nie sú deliteľné 7 a zvyšok po delení 5 je 2 alebo 3

   - napr.

```
>>> a = vyrob(20)
>>> a
{2, 3, 8, 12, 13, 17, 18}
```

   - vytvorenie množiny vo funkcii zapíšte najprv pomocou for-cyklu:

```
def vyrob(n):
    vysl = set()
    for ...
        if ...
            ...
    return vysl
```

   - potom sa túto funkciu zapíšte tak, aby obsahovala len jeden riadok:

```
def vyrob(n):
    return ...
```

9. Napíšte funkciu `bez(mnoz, k)`, ktorá z danej množiny `mnoz` vyhodí všetky prvky, ktoré sú deliteľné číslom `k`. Funkcia nič nevracia ani nevypisuje.

   - napr.

```
>>> m = {1, 2, 3, 4, '5', 6}
>>> bez(m, 3)
>>> m
{1, 2, 4, '5'}
>>> m1 = set(range(0, 100, 5))
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> bez(m1, 10)
>>> m1
{65, 35, 5, 75, 45, 15, 85, 55, 25, 95}
```

10. Napíšte funkciu `viac(zoznam)`, ktorá vráti množinu tých prvkov zoznamu `zoznam`, ktoré sa v ňom vyskytujú viac ako raz.

    - napr.

```
>>> p = ['prvy', 2, (3, 4), 'dva', 3, 4, 'prvy', 3]
>>> v = viac(p)
>>> v
{3, 'prvy'}
```

11. Napíšte funkciu `len_cisla(mnozina)`, ktorá z danej množiny vyrobí novú ale len z tých prvkov, ktoré sú čísla (`int` alebo `float`).

    - napr.

```
>>> a = {'1', 2.2, (3, 4), 5}
>>> b = len_cisla(a)
>>> b
{2.2, 5}
```

12. Napíšte funkciu `vsetky(a, b, c, d)`, ktorá vráti zoznam všetkých dvojprvkových množín z prvkov `a`, `b`, `c`, `d`. Predpokladajte, že všetky hodnoty parametrov sú navzájom rôzne.

    - napr.

```
>>> vsetky(1, 2, 3, 4)
[{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}]
```

13. Vylepšite funkciu z predchádzajúceho príkladu `vsetky(postupnost)`, tak aby generovala všetky dvojprvkové množiny z prvkov zadanej postupnosti (zoznam, reťazec, `range()`, ...). Dajte pozor, ak sa nejaké hodnoty v postupnosti opakujú.

    - napr.

```
>>> vsetky('java')
[{'j', 'v'}, {'a', 'v'}, {'j', 'a'}]
>>> vsetky(range(5))
[{0, 1}, {0, 2}, {0, 3}, {0, 4}, {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3,␣
↪4}]
>>> vsetky((3, 1, 'x', 4, 1, 2))
[{1, 2}, {1, 'x'}, {1, 3}, {1, 4}, {2, 'x'}, {2, 3}, {2, 4}, {3, 'x'}, {'x',␣
↪4}, {3, 4}]
```

18.4 7. Domáce zadanie

L.I.S.T.

- riešenie odovzdávajte na úlohový server https://list.fmph.uniba.sk/

Iste ste sa už stretli s logickou hrou Sudoku: úlohou hráča je zaplniť voľné políčka štvorcovej siete 9x9 (dvojrozmerná tabuľka) číslami od 1 do 9, tak aby boli splnené podmienky:

- v každom riadku tabuľky bolo každé číslo práve raz
- v každom stĺpci tabuľky bolo každé číslo práve raz
- v každom vyznačenom štvorci 3x3 tabuľky bolo každé číslo práve raz

Zadanie hry môže vyzerať, napr. takto:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Váš program sa bude snažiť riešiť túto hru veľmi zjednodušeným spôsobom: postupne prejde všetky voľné políčka a pre každé z nich zistí množinu kandidátov, t.j. všetkých takých čísel, ktoré by sme na toto políčko mohli položiť a nevznikla by kolízia z už položemými číslami v riadku, v stĺpci a ani vo príslušnom štvorci 3x3. Zrejme, ak je niektorá z týchto množín prázdna, táto hra už nemá riešenie a netreba sa ju ďalej snažiť riešiť. Ak na niektorom voľnom políčku je táto množina jednoprvková, znamená to, že práve toto číslo je jediné, ktoré sem môžeme (musíme) zapísať. Ak toto urobíme so všetkými jednoprvkovými množinami, trochu sa priblížime k celkovému riešeniu tohto Sudoku. Riešenie pre vás teda bude znamenať toto:

1. všetky voľné políčka nahradiť množinami kandidátov
2. ak je niektorá z množín prázdna, úloha nemá riešenie, bude treba skončiť
3. ak tam nie je žiadna množina (nie je tam voľné políčko), úloha je zrejme vyriešená a bude treba skončiť
4. ak majú všetky množiny viac ako 1 prvok, bude treba skončiť, lebo tento algoritmus už viac robiť nevie
5. ak je tam niekoľko jednoprvkových množín, všetky sa nahradia týmto svojim jediným prvkom, všetky ostatné množiny sa nahradia znakom `'.'`
6. ďalej sa pokračuje v 1. kroku

Napíšte pythonovský modul, ktorý bude obsahovať jedinú triedu `Sudoku` a žiadne iné globálne premenné:

```
class Sudoku:
    def __init__(self, meno_suboru):
        self.tab = []
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        ...

    def __str__(self):
        ...

    def urob(self):
        ...

    def nahrad(self):
        ...

    def ries(self):
        ...
```

Metódy majú fungovať takto:

- inicializácia `__init__(self, meno_suboru)` prečíta textový súbor s počiatočným zaplnením čísel, voľné políčka sú vyznačené znakmi `','`
  - súbor obsahuje 9 riadkov, v každom je 9 číslic alebo bodiek oddelených medzerou
  - inicializácia zaplní dvojrozmernú tabuľku `self.tab` (prvkami musia byť celé čísla `int` a znaky `'.'`)
- metóda `__str__(self)` vyrobí stringovú reprezentáciu hracej plochy: reťazec bude obsahovať 9 riadkov v každom po 9 hodnôt (čísel alebo bodiek) v rovnakom formáte, ako bol zadaný vstupný súbor
- metóda `urob(self)` všetky voľné políčka (v tabuľke sú tam `'.'`) nahradí množinami kandidátov, t.j. čísel, ktoré by sa na tejto pozícii mohli náchadzať
  - funkcia vráti `-1`, ak sa medzi týmito množinami objavila **prázdna** množina, inak funkcia vráti celkový počet **jednoprvkových** množín
- metóda `nahrad(self)` všetky políčka s jednoprvkovými množinami sa nahradia priamo hodnotou v tejto množine, ostatné políčka s množinami sa nahradia znakom `'.'`
- metóda `ries(self)` bude postupne volať metódy `urob()` a `nahrad()`, kým bude metóda `urob()` vracať číslo väčšie ako `0` (zrejme bude vykonávať hore uvedený cyklus)
  - funkcia vráti `-1`, ak táto hra nemá riešenie (metóda `urob()` vrátila `-1`)
  - funkcia vráti počet voľných políčok (zrejme `0` označuje, že úloha je vyriešená)
  - po skončení metódy `ries(self)` by dvojrozmerná tabuľka `self.tab` mala obsahovať len čísla a znaky `'.'`

Napr. pre vstupný súbor `'subor1.txt'`:

```
. . . . . 9 . . .
. . 7 . 8 6 . . .
6 . . 3 . . . . .
. 4 . . . 7 . . 8
. . . . . . . 3 2
. . 3 6 . 5 1 . .
. 6 . 7 . . . 8 .
3 . 2 . . . 4 9 .
. 5 4 8 . . . . 3
```

Tento test nám bude postupne vypisovať:

```
>>> s = Sudoku('subor1.txt')
>>> print(s)
. . . . . 9 . . .
. . 7 . 8 6 . . .
6 . . 3 . . . . .
. 4 . . . 7 . . 8
. . . . . . . 3 2
. . 3 6 . 5 1 . .
. 6 . 7 . . . 8 .
3 . 2 . . . 4 9 .
. 5 4 8 . . . . 3
>>> s.urob()
1
>>> for r in s.tab:
        print(r)

[{1, 2, 4, 5, 8}, {1, 2, 3, 8}, {1, 5, 8}, {1, 2, 4, 5}, {1, 2, 4, 5, 7}, 9, {2, 3, 5,
↪ 6, 7, 8}, {1, 2, 4, 5, 6, 7}, {1, 4, 5, 6, 7}]
[{1, 2, 4, 5, 9}, {1, 2, 3, 9}, 7, {1, 2, 4, 5}, 8, 6, {2, 3, 5, 9}, {1, 2, 4, 5}, {1,
↪ 4, 5, 9}]
[6, {1, 2, 8, 9}, {1, 5, 8, 9}, 3, {1, 2, 4, 5, 7}, {1, 2, 4}, {2, 5, 7, 8, 9}, {1, 2,
↪ 4, 5, 7}, {1, 4, 5, 7, 9}]
[{1, 2, 5, 9}, 4, {1, 5, 6, 9}, {1, 2, 9}, {1, 2, 3, 9}, 7, {5, 6, 9}, {5, 6}, 8]
[{1, 5, 7, 8, 9}, {1, 7, 8, 9}, {1, 5, 6, 8, 9}, {1, 4, 9}, {1, 4, 9}, {1, 4, 8}, {5,␣
↪6, 7, 9}, 3, 2]
[{2, 7, 8, 9}, {2, 7, 8, 9}, 3, 6, {2, 4, 9}, 5, 1, {4, 7}, {4, 7, 9}]
[{1, 9}, 6, {1, 9}, 7, {1, 2, 3, 4, 5, 9}, {1, 2, 3, 4}, {2, 5}, 8, {1, 5}]
[3, {1, 7, 8}, 2, {1, 5}, {1, 5, 6}, {1}, 4, 9, {1, 5, 6, 7}]
[{1, 7, 9}, 5, 4, 8, {1, 2, 6, 9}, {1, 2}, {2, 6, 7}, {1, 2, 6, 7}, 3]
>>> s.nahrad()
>>> for r in s.tab:
        print(r)

['.', '.', '.', '.', '.', 9, '.', '.', '.']
['.', '.', 7, '.', 8, 6, '.', '.', '.']
[6, '.', '.', 3, '.', '.', '.', '.', '.']
['.', 4, '.', '.', '.', 7, '.', '.', 8]
['.', '.', '.', '.', '.', '.', '.', 3, 2]
['.', '.', 3, 6, '.', 5, 1, '.', '.']
['.', 6, '.', 7, '.', '.', '.', 8, '.']
[3, '.', 2, '.', '.', 1, 4, 9, '.']
['.', 5, 4, 8, '.', '.', '.', '.', 3]
>>> s.ries()
28
>>> print(s)
. . . . . 9 . . .
. . 7 . 8 6 . . .
6 . . 3 . 4 . . .
2 4 1 9 3 7 5 6 8
. . . 4 1 8 . 3 2
. . 3 6 2 5 1 . .
1 6 9 7 4 3 2 8 5
3 8 2 5 6 1 4 9 7
7 5 4 8 9 2 6 1 3
```

Váš odovzdaný program s menom `riesenie7.py` musí začínať tromi riadkami komentárov:

```
# 7. zadanie: sudoku
# autor: Janko Hraško
# datum: 15.12.2017
```

Projekt `riesenie7.py` odovzdávajte na úlohový server https://list.fmph.uniba.sk/ najneskôr do 23:00 **5. januára**, kde ho môžete nechať otestovať. Testovač bude spúšťať vašu funkciu s rôznymi vstupmi. Odovzdať projekt aj ho testovať môžete ľubovoľný počet krát. Môžete zaň získať **10 bodov**.

KAPITOLA 19

19. Slovníky (dict)

Na 14. cvičení ste riešili úlohu, v ktorej ste zostavovali metódy pre triedu telefónny zoznam. Riešenie úlohy by mohlo vyzerať asi takto:

```
class TelefonnyZoznam:
    def __init__(self):
        self.zoznam = []

    def __str__(self):
        vysl = []
        for meno, telefon in self.zoznam:
            vysl.append(f'{meno}: {telefon}')
        return '\n'.join(vysl)

    def pridaj(self, meno, telefon):
        for i in range(len(self.zoznam)):
            if self.zoznam[i][0] == meno:
                self.zoznam[i] = meno, telefon
                return
        self.zoznam.append((meno, telefon))
```

Vidíte, že do zoznamu ukladáme dvojice `(meno, telefon)`.

Jednoduchý test:

```
tz = TelefonnyZoznam()
tz.pridaj('Jana', '0901020304')
tz.pridaj('Juro', '0911111111')
tz.pridaj('Jozo', '0212345678')
tz.pridaj('Jana', '0999020304')
print(tz)
```

```
Jana: 0999020304
Juro: 0911111111
Jozo: 0212345678
```

19.1 Hľadanie údajov

Do tejto triedy pridajme ešte metódy `__contains__()` a `zisti()`, vďaka ktorým budeme vedieť zistiť, či sa dané meno nachádza v zozname, prípadne pre dané meno zistíme jeho telefónne číslo:

```
class TelefonnyZoznam:
    ...

    def __contains__(self, hladaj_meno):
        for meno, telefon in self.zoznam:
            if meno == hladaj_meno:
                return True
        return False

    def zisti(self, hladaj_meno):
        for meno, telefon in self.zoznam:
            if meno == hladaj_meno:
                return telefon
        raise ValueError('zadane meno nie je v zozname')
```

Opäť otestujeme:

```
tz = TelefonnyZoznam()
tz.pridaj('Jana', '0901020304')
tz.pridaj('Juro', '0911111111')
tz.pridaj('Jozo', '0212345678')
tz.pridaj('Jana', '0999020304')
print(tz)
print('Juro ma telefon', tz.zisti('Juro'))
print('Fero je v zozname:', 'Fero' in tz)       # volanie tz.__contains__('Fero')
print('Fero ma telefon', tz.zisti('Fero'))
```

```
Jana: 0999020304
Juro: 0911111111
Jozo: 0212345678
Juro ma telefon 0911111111
Fero je v zozname: False
...
ValueError: zadane meno nie je v zozname
```

Zamerajme sa teraz na spôsob hľadania v tomto telefónnom zozname: všetky tri metódy `pridaj()`, `__contains__()` aj `zisti()` postupne pomocou for-cyklu prechádzajú celý zoznam a kontrolujú, či pritom nenašli hľadané meno. Zrejme pre veľký telefónny zoznam (napr. telefónny zoznam New Yorku môže obsahovať viac ako 8 miliónov dvojíc `(meno, číslo)`) takéto **sekvenčné** prehľadávanie môže trvať už dosť dlho.

Naučme sa jednoduchý spôsob, akým môžeme odmerať čas behu nejakého programu. Hoci takéto meranie vôbec nie je presné, môže nám to dať nejaký obraz o tom, ako dlho trvajú niektoré časti programu. Na meranie použijeme takúto šablónu:

```
import time

start = time.time()                  # začiatok merania času

# nejaký výpočet

print(round(time.time()-start, 3))   # koniec merania času
```

Volanie funkcie `time.time()` vráti momentálny stav nejakého počítadla sekúnd. Keď si tento stav zapamätáme (v premennej `start`) a o nejaký čas opäť zistíme stav počítadla, rozdiel týchto dvoch hodnôt nám vráti približný počet sekúnd koľko ubehlo medzi týmito dvomi volaniami `time.time()`. Túto hodnotu sa dozvedáme ako desatinné číslo, takže vidíme aj milisekundy (výsledný rozdiel zaokrúhľujeme na 3 desatinné miesta).

Začnime s meraním tohto jednoduchého programu na výpočet súčtu `n` prirodzených čísel:

```
import time

n = int(input('zadaj n: '))
start = time.time()                             # začiatok merania času
sucet = 0
for i in range(1, n+1):
    sucet += i
print('cas =', round(time.time()-start, 3))     # koniec merania času
print('sucet =', sucet)
```

Tento program spustíme niekoľkokrát s rôzne veľkým `n`:

```
zadaj n: 10000
cas = 0.002
sucet = 50005000
```

```
zadaj n: 100000
cas = 0.021
sucet = 5000050000
```

```
zadaj n: 1000000
cas = 0.235
sucet = 500000500000
```

Môžete si všimnúť istú závislosť trvania programu od veľkosti `n`: keď zadáme 10-krát väčšiu vstupnú hodnotu, výpočet bude trvať približne 10-krát tak dlho. Zrejme je to nejaká lineárna závislosť: čas behu programu závisí od veľkosti `n` lineárne.

Teraz namiesto výpočtu sumy budeme merať približný čas hľadania údajov v nejakej tabuľke, pričom použijeme **sekvenčné** prehľadávanie (postupne budeme prechádzať celý zoznam pomocou for-cyklu). Pre jednoduchosť testovania budeme pracovať len so zoznamom celých čísel, ktoré najprv vygenerujeme náhodne:

```
import time
from random import randrange as rr

def hladaj(hodnota):
    for prvok in zoznam:
        if prvok == hodnota:
            return True
    return False

n = int(input('zadaj n: '))
zoznam = []
for i in range(n):
    zoznam.append(rr(2*n))
start = time.time()             # začiatok merania času
for i in range(0, 2*n, 2):
    hladaj(i)
cas = time.time()-start         # koniec merania času
print('cas =', round(cas/n*1000, 3))
```

Najprv sme vygenerovali n-prvkový zoznam náhodných čísel z intervalu <0, 2n-1>. Pre toto generovanie zoznamu sme ešte nemerali čas trvania. Samotný odmeriavaný test teraz n-krát hľadá v tomto zozname nejakú hodnotu z intervalu <0, 2n-1> (zrejme sa poskúša hľadať len párne hodnoty). Na záver vypíše čas ale vydelený počtom hľadaní (počet volaní funkcie hladaj()) a aby to neboli príliš malé čísla, čas ešte vynásobí 1000, teda nedostávame sekundy, ale milisekundy.

Tento test niekoľkokrát spustíme pre rôzne n:

```
zadaj n: 100
cas = 0.005
```

```
zadaj n: 1000
cas = 0.05
```

```
zadaj n: 10000
cas = 0.496
```

```
zadaj n: 100000
cas = 5.921
```

čo môžeme vyčítať z týchto výsledkov:

- pre 100-prvkový zoznam trvalo jedno hľadanie približne 0.005 milisekúnd (t.j. 5 milióntin sekundy)
- pre 1000-prvkový zoznam trvalo jedno hľadanie približne 0.05 milisekúnd, to znamená, že pre 10-krát väčší zoznam aj hľadanie trvá asi 10-krát tak dlho
- pre 10000-prvkový zoznam trvalo jedno hľadanie približne 0.5 milisekundy, čo asi potvrdzuje našu hypotézu, že čas hľadania je lineárne závislý od veľkosti zoznamu
- pre 10000-prvkový zoznam jedno hľadanie trvá skoro 6 milisekúnd a môžeme si zatipovať, že v 1000000-prvkovom zozname by sme hľadali 100-krát dlhšie, teda asi 600 milisekúnd, čo je viac ako pol sekundy

Asi vidíme, že telefónny zoznam New Yorku s 8 miliónmi mien bude náš program priemerne prehľadávať 5 sekúnd - a to už je dosť veľa.

Zrejme múdre programy na prehľadávanie telefónnych zoznamov používajú nejaké lepšie stratégie. V prvom rade sú takéto zoznamy usporiadané podľa mien, vďaka čomu sa bude dať hľadať oveľa rýchlejšie.

19.1.1 Binárne vyhľadávanie

Použijeme podobnú ideu, akou sme na 4. prednáške riešili úlohu zisťovanie druhej odmocniny nejakého čísla. V tomto prípade využijeme ten fakt, že v reálnom telefónnom zozname sú všetky mená usporiadané podľa abecedy. Vďaka tomu, ak by sme si vybrali úplne náhodnú pozíciu v takomto zozname, na základe príslušnej hodnoty v tabuľke, sa vieme jednoznačne rozhodnúť, či ďalej budeme pokračovať v hľadaní v časti zoznamu pred touto pozíciou alebo za. Nebudeme ale túto pozíciu voliť náhodne, vyberieme pozíciu v strede zoznamu.

Ukážme to na príklade: máme telefónny zoznam, ktorý je usporiadaný podľa mien. Pre lepšie znázornenie použijeme len dvojpísmenové mená, napr. tu je utriedený 13-prvkový zoznam a ideme v ňom hľadať meno 'hj' (možno Hraško Ján):

0	1	2	3	4	5	6	7	8	9	10	11	12	
ab	**ad**	**dd**	**fm**	**hj**	**jj**	**ka**	**kz**	**mo**	**pa**	**rd**	**tz**	**zz**	
zac						stred						kon	

Označili sme tu začiatok a koniec intervalu zoznamu, v ktorom práve hľadáme: na začiatku je to kompletný zoznam od indexu 0 až po 12. Vypočítame stred (ako (zac+kon)//2) - to je pozícia, kam sa pozrieme úplne na začiatku.

Keďže v strede zoznamu je meno `'ka'`, tak zrejme všetky mená v zozname za týmto stredom sú v abecede vyššie a teda `'hj'` sa medzi nimi určite nenachádza (platí `'hj' < 'ka'`). Preto odteraz bude stačiť hľadať len v prvej polovici zoznamu teda v intervale od 0 do 5. Opravíme koncovú hranicu intervalu a vypočítame nový `stred`:

0	1	2	3	4	5	6	7	8	9	10	11	12	
ab	**ad**	**dd**	**fm**	**hj**	**jj**	ka	kz	mo	pa	rd	tz	zz	
zac		stred			kon								

Stredná pozícia v tomto prípade padla na meno `'dd'`. Keďže `'hj'` je v abecede až za `'dd'` (platí `'dd' < 'hj'`), opäť prepočítame hranice sledovaného intervalu a tiež nový stred:

0	1	2	3	4	5	6	7	8	9	10	11	12	
ab	ad	dd	**fm**	**hj**	**jj**	ka	kz	mo	pa	rd	tz	zz	
			zac	stred	kon								

Už pri tomto treťom kroku algoritmu sa nám podarilo objaviť pozíciu hľadaného mena v zozname: `stred` teraz odkazuje presne na naše hľadané slovo.

Ak by sa hľadané slovo v tomto telefónnom zozname nenachádzalo (napr. namiesto `'hj'` by sme hľadali `'gr'`), tak by algoritmus pokračoval ďalšími krokmi: opäť porovnáme stredný prvok s našim hľadaným (`'gr' < 'hj'`) a keďže je v abecede pred ním, zmeníme interval tak, že `zac` == `kon` == `stred`:

0	1	2	3	4	5	6	7	8	9	10	11	12	
ab	ad	dd	**fm**	hj	jj	ka	kz	mo	pa	rd	tz	zz	
			stred										

A to je teda už definitívny koniec algoritmu (interval je 1-prvkový): keďže tento jediný prvok je rôzny od hľadaného `'gr'`, je nám jasné, že sme skončili so správou „nenašli“.

Zapíšme tento algoritmus do Pythonu:

```
def hladaj(hodnota):
    zac = 0                             # zaciatok intervalu
    kon = len(zoznam) - 1               # koniec intervalu
    while zac <= kon:
        stred = (zac + kon) // 2        # stred intervalu
        if zoznam[stred] < hodnota:
            zac = stred + 1
        elif zoznam[stred] > hodnota:
            kon = stred - 1
        else:
            return True
    return False
```

Ak by sme teraz spustili rovnaké testy ako pred tým so sekvenčným vyhľadávaním, boli by sme milo prekvapení: čas na hľadanie v 1000-prvkovom zozname a 1000000-prvkovom zozname je skoro stále rovnaký a sú to len tisíciny milisekúnd.

19.2 Štandardný typ dict

Typ `dict` **slovník** (informatici to volajú **asociatívne pole**) je taká dátová štruktúra, v ktorej k prvkom neprichádzame cez poradové číslo (index) tak ako pri zoznamoch, n-ticiach a reťazcoch, ale k prvkom prichádzame pomocou **kľúča**.

Hovoríme, že k danému kľúču je **asociovaná** nejaká hodnota (niekedy hovoríme, že hodnotu **mapujeme** na daný kľúč).

Zapisujeme to takto:

```
kľúč : hodnota
```

Samotný slovník zapisujeme ako takéto dvojice (`kľúč : hodnota`) a celé je to uzavreté v `'{'` a `'}'` zátvorkách. Slovník si môžeme predstaviť ako zoznam dvojíc (kľúč, hodnota), pričom v takomto zozname nemôžu byť dve dvojice s rovnakým kľúčom. Napr.

```
>>> vek = {'Jan':17, 'Maria':15, 'Ema':18}
```

Takto sme vytvorili slovník (asociatívne pole, teda pythonovský typ `dict`), ktorý má tri prvky: `'Jan'` má 17 rokov, `'Maria'` má 15 a `'Ema'` má 18. V priamom režime vidíme, ako ho vypisuje Python a tiež to, že pre Python to má 3 prvky (3 dvojice):

```
>>> vek
{'Jan': 17, 'Maria': 15, 'Ema': 18}
>>> len(vek)
3
```

Teraz zápisom, ktorý sa podobá indexovaniu zoznamu:

```
>>> vek['Jan']
17
```

získame asociovanú hodnotu pre kľúč `'Jan'`.

Ak sa pokúsime zistiť hodnotu pre neexistujúci kľúč:

```
>>> vek['Juraj']
...
KeyError: 'Juraj'
```

Python nám tu oznámi `KeyError`, čo znamená, že tento slovník nemá definovaný kľúč `'Juraj'`.

Rovnako, ako priraďujeme hodnotu do zoznamu, môžeme vytvoriť novú hodnotu pre nový kľúč:

```
>>> vek['Hana'] = 15
>>> vek
{'Jan': 17, 'Hana': 15, 'Maria': 15, 'Ema': 18}
```

alebo môžeme zmeniť existujúcu hodnotu:

```
>>> vek['Maria'] = 16
>>> vek
{'Jan': 17, 'Hana': 15, 'Maria': 16, 'Ema': 18}
```

Všimnite si, že poradie dvojíc `kľúč:hodnota` v samotnom slovníku je v nejakom neznámom poradí, dokonca, ak spustíme program viackrát, môžeme dostať rôzne poradia prvkov. Podobnú skúsenosť určite máte aj s typom `set` z predchádzajúcej prednášky.

Pripomeňme si, ako funguje operácia `in` s typom zoznam:

```
>>> zoznam = [17, 15, 16, 18]
>>> 15 in zoznam
True
```

Operácia `in` s takýmto zoznamom prehľadáva hodnoty a ak takú nájde, vráti `True`.

So slovníkom to funguje trochu inak: operácia `in` neprehľadáva hodnoty ale kľúče:

```
>>> 17 in vek
False
>>> 'Hana' in vek
True
```

Vďaka tomuto vieme zabrániť, aby program spadol pri neznámom kľúči:

```
>>> if 'Juraj' in vek:
        print('Juraj ma', vek['Juraj'], 'rokov')
    else:
        print('nepoznám Jurajov vek')
```

Keďže operácia `in` so slovníkom prehľadáva kľúče, tak aj for-cyklus bude fungovať na rovnakom princípe:

```
>>> for i in zoznam:
        print(i)
17
15
16
18
>>> for i in vek:
        print(i)
Jan
Hana
Maria
Ema
```

Z tohto istého dôvodu funkcia `list()` s parametrom slovník vytvorí zoznam kľúčov a nie zoznam hodnôt:

```
>>> list(vek)
['Jan', 'Hana', 'Maria', 'Ema']
```

Keď chceme zo slovníka vypísať všetky kľúče aj s ich hodnotami, zapíšeme:

```
>>> for kluc in vek:
        print(kluc, vek[kluc])

Jan 17
Hana 15
Maria 16
Ema 18
```

Zrejme slovník rovnako ako zoznam je **meniteľný** typ (**mutable**), keďže môžeme do neho pridávať nové prvky, resp. meniť hodnoty existujúcich prvkov.

Napr. funguje takýto zápis:

```
>>> vek['Jan'] = vek['Jan'] + 1
>>> vek
{'Jan': 18, 'Hana': 15, 'Maria': 16, 'Ema': 18}
```

a môžeme to využiť aj v takejto funkcii:

```
def o_1_starsi(vek):
    for kluc in vek:
        vek[kluc] = vek[kluc] + 1
```

Funkcia zvýši hodnotu v každej dvojici slovníka o 1:

```
>>> o_1_starsi(vek)
>>> vek
{'Jan': 19, 'Hana': 16, 'Maria': 17, 'Ema': 19}
```

Pythonovské funkcie teda môžu meniť (ako vedľajší účinok) nielen zoznam ale aj slovník.

Ak by sme chceli prejsť kľúče v utriedenom poradí, musíme zoznam kľúčov najprv utriediť:

```
>>> for kluc in sorted(vek):
        print(kluc, vek[kluc])
Ema 19
Hana 16
Jan 19
Maria 17
```

Slovníky majú definovaných viac zaujímavých metód, my si najprv ukážeme len 3 z nich. Táto skupina troch metód vráti nejaké špeciálne „postupnosti“:

- `keys()` - postupnosť kľúčov
- `values()` - postupnosť hodnôt
- `items()` - postupnosť dvojíc kľúč a hodnota

Napríklad:

```
>>> vek.keys()
dict_keys(['Jan', 'Hana', 'Maria', 'Ema'])
>>> vek.values()
dict_values([19, 16, 17, 19])
>>> vek.items()
dict_items([('Jan', 19), ('Hana', 16), ('Maria', 17), ('Ema', 19)])
```

Tieto postupnosti môžeme použiť napr. vo for-cykle alebo môžu byť parametrami rôznych funkcií, napr. `list()`, `max()` alebo `sorted()`:

```
>>> list(vek.values())
[19, 16, 17, 19]
>>> list(vek.items())
[('Jan', 19), ('Hana', 16), ('Maria', 17), ('Ema', 19)]
```

Metódu `items()` najčastejšie využijeme vo for-cykle:

```
>>> for prvok in vek.items():
        kluc, hodnota = prvok
              print(kluc, hodnota)

Jan 19
Hana 16
Maria 17
Ema 19
```

Alebo krajšie dvojicou premenných for-cyklu:

```
>>> for kluc, hodnota in vek.items():
            print(kluc, hodnota)

Jan 19
Hana 16
Maria 17
Ema 19
```

Jednou z najpoužívanejších metód je `get()`.

metóda `get()`

Táto metóda vráti asociovanú hodnotu k danému kľúču, ale v prípade, že daný kľúč neexistuje, nespadne na chybe, ale vráti nejakú náhradnú hodnotu. Metódu môžeme volať s jedným alebo aj dvoma parametrami:

```
slovnik.get(kluc)
slovnik.get(kluc, nahrada)
```

V prvom prípade, ak daný kľúč neexistuje, funkcia vráti `None`, ak v druhom prípade neexistuje kľúč, tak funkcia vráti hodnotu `nahrada`.

Napr.

```
>>> print(vek.get('Maria'))
17
>>> print(vek.get('Mario'))
None
>>> print(vek.get('Maria', 20))
17
>>> print(vek.get('Mario', 20))
20
```

Príkaz `del` funguje nielen so zoznamom, ale aj so slovníkom:

```
>>> zoznam = [17, 15, 16, 18]
>>> del zoznam[1]
>>> zoznam
[17, 16, 18]
```

príkaz `del` so slovníkom

Príkaz `del` vyhodí zo slovníka príslušný kľúč aj s jeho hodnotou:

```
del slovnik[kluc]
```

Ak daný kľúč v slovníku neexistuje, príkaz vyhlási `KeyError`

Napríklad:

```
>>> vek
{'Jan': 19, 'Hana': 16, 'Maria': 17, 'Ema': 19}
>>> del vek['Jan']
>>> vek
{'Hana': 16, 'Maria': 17, 'Ema': 19}
```

Zhrňme všetko, čo sme sa doteraz naučili pre dátovú štruktúru slovník:

- `slovník = {}`
  - prázdny slovník
- `slovník = {kľúč:hodnota, ...}`
  - priame priradenie celého asociatívneho poľa
- `kľúč in slovník`
  - zistí, či v slovníku existuje daný `kľúč` (vráti `True` alebo `False`)
- `len(slovník)`
  - zistí počet prvkov (dvojíc `kľúč:hodnota`) v slovníku
- `slovník[kľúč]`
  - vráti príslušnú hodnotu alebo príkaz spadne na chybe `KeyError`, ak neexistuje
- `slovník[kľúč] = hodnota`
  - vytvorí novú asociáciu `kľúč:hodnota` alebo zmení existujúcu
- `del slovník[kľúč]`
  - zruší dvojicu `kľúč:hodnota` alebo príkaz spadne na chybe `KeyError`, ak neexistuje
- `for kľúč in slovník: ...`
  - prechádzanie všetkých kľúčov
- `for kľúč in sorted(slovník): ...`
  - prechádzanie všetkých kľúčov slovníka v utriedenom poradí
- `for kľúč in slovník.values(): ...`
  - prechádzanie všetkých hodnôt
- `for kľúč, hodnota in slovník.items(): ...`
  - prechádzanie všetkých dvojíc `kľúč:hodnota`
- `slovník.get(kľúč)`
  - vráti príslušnú hodnotu alebo `None`, ak kľúč neexistuje
- `slovnik.get(kľúč, náhradná)`
  - vráti príslušnú hodnotu alebo vráti hodnotu parametra `náhradná`, ak kľúč neexistuje

---

Predstavte si teraz, že máme daný nejaký zoznam dvojíc a chceme z neho urobiť slovník. Môžeme to urobiť, napr. for-cyklom:

```
>>> zoznam_dvojic = [('one', 1), ('two', 2), ('three', 3)]
>>> slovnik = {}
>>> for k, h in zoznam_dvojic:
        slovnik[k] = h
>>> slovnik
{'one': 1, 'two': 2, 'three': 3}
```

alebo priamo použitím štandardnej funkcie `dict()`, ktorá takto funguje ako konverzná funkcia:

```
>>> slovnik = dict(zoznam_dvojic)
>>> slovnik
{'one': 1, 'two': 2, 'three': 3}
```

Takýto zoznam dvojíc vieme vytvoriť aj z nejakého existujúceho slovníka pomocou metódy `items()`:

```
>>> list(slovnik.items())
[('one', 1), ('two', 2), ('three', 3)]
```

Funkciu `dict()` môžeme zavolať aj takto:

```
>>> slovnik = dict(one=1, two=2, three=3)
```

Vďaka tomuto zápisu nemusíme kľúče uzatvárať do apostrofov. Toto ale funguje len pre kľúče, ktoré sú znakové reťazce a majú správny formát pre identifikátory pomenovaných parametrov.

19.2.1 Asociatívne pole ako slovník (dictionary)

Anglický názov tohto typu `dict` je zo slova **dictionary**, teda slovník. Naozaj je „obyčajný“ slovník príkladom pekného využitia tohto typu. Napr.

```
slovnik = {'cat':'macka', 'dog':'pes', 'my':'moj', 'good':'dobry', 'is':'je'}

def preklad(veta):
    vysl = []
    for slovo in veta.lower().split():
        vysl.append(slovnik.get(slovo, slovo))
    return ' '.join(vysl)
```

```
>>> preklad('my dog is very good')
'moj pes je very dobry'
```

Zrejme, keby sme mali kompletnejší slovník, napr. anglicko-slovenský s tisícami dvojíc slov, vedeli by sme veľmi jednoducho realizovať takýto „kuchársky“ preklad. V skutočnosti by sme asi mali mať slovník, v ktorom jednému anglickému slovu zodpovedá niekoľko (napr. n-tica) slovenských slov.

19.2.2 Slovník ako frekvenčná tabuľka

Frekvenčnou tabuľkou nazývame takú tabuľku, ktorá pre rôzne hodnoty obsahuje informácie o počte výskytov v nejakom zozname hodnôt (napr. súbor, reťazec, zoznam, ...).

Ukážeme to na zisťovaní počtu výskytov napr. písmen v nejakom texte. Budeme konštruovať slovník, v ktorom každému prvku vstupného zoznamu (kľúču) bude zodpovedať jedno celé číslo - počítadlo výskytov tohto prvku:

```
def pocty_vyskytov(zoznam):
    vysl = {}
    for prvok in zoznam:
        vysl[prvok] = vysl.get(prvok, 0) + 1
    return vysl

pocet = pocty_vyskytov(list('anicka dusicka nekasli, aby ma pri tebe nenasli.'))
for k, h in pocet.items():
    print(k, h)
```

Všimnite si použitie metódy `get()`, vďaka ktorej nepotrebujeme pomocou podmieneného príkazu `if` zisťovať, či sa v slovníku príslušný kľúč už nachádza. Zápis `vysl.get(prvok, 0)` pre spracovávaný `prvok` vráti momentálny počet jeho výskytov, alebo 0, ak sa tento `prvok` vyskytol prvýkrát. Hoci my by sme vedeli toto isté zapísať aj pomocou spracovania výnimky:

```
def pocty_vyskytov(zoznam):
    vysl = {}
    for prvok in zoznam:
        try:
            vysl[prvok] += 1
        except KeyError:
            vysl[prvok] = 1
    return vysl
```

Podobne môžeme spracovať aj nejaký celý textový súbor (dobs.txt alebo twain.txt):

```
with open('dobs.txt') as subor:
    pocet = pocty_vyskytov(subor.read())
for k, h in pocet.items():
    print(k, h)
```

Vidíme, že pri väčších súboroch sú aj zistené počty výskytov dosť vysoké. Napr. všetkých spracovaných znakov bolo:

```
>>> sum(pocet.values())
```

Ak by sme potrebovali zisťovať nie počty písmen, ale počty slov, stačí zapísať:

```
with open('dobs.txt') as subor:
    pocet = pocty_vyskytov(subor.read().split())
```

Táto konkrétna štruktúra slovník je už dosť veľká a nemá zmysel ju vypisovať celú, veď všetkých rôznych slov a teda veľkosť slovníka je

```
>>> len(pocet)
```

Ukážeme, ako môžeme zistiť 10 najčastejších slov vo veľkom texte. Najprv z frekvenčnej tabuľky `pocet` vytvoríme zoznam dvojíc `(hodnota, kľúč)` (prehodíme poradie v dvojiciach `(kľúč, hodnota)`). Potom tento zoznam utriedime. Robíme to preto, lebo Python triedi n-tice tak, že najprv porovnáva prvé prvky (teda počty výskytov) a potom až pri zhode porovná druhé prvky (teda samotné slová). Všimnite si, že sme tu použili metódu `sort()` pre zoznamy a pridali sme jeden pomenovaný parameter `reverse=True`, aby sa zoznam utriedil zostupne, teda od najväčších po najmenšie:

```
zoznam = []
for k, h in pocet.items():
    zoznam.append((h, k))

zoznam.sort(reverse=True)

print(zoznam[:10])
```

Ešte si uvedomte, že kľúčmi nemusia byť len slová, resp. písmená. Kľúčmi môže byť ľubovoľný nemeniteľný (**immutable**) typ, teda okrem `str` aj `int`, `float` a `tuple`. Teda by sme zvládli zisťovať aj počet výskytov prvkov číselných zoznamov, alebo hoci dvojíc čísel (súradníc bodov v rovine) a pod.

19.2.3 Zoznam slovníkov

Slovník môže byť aj hodnotou v inom slovníku. Zapíšme:

```
student1 = {
    'meno': 'Janko Hrasko',
    'adresa': {'ulica': 'Strukova',
               'cislo': 13,
               'obec': 'Fazulovo'},
    'narodeny': {'datum': {'den': 1, 'mesiac': 5, 'rok': 1999},
                 'obec': 'Korytovce'}
}
student2 = {
    'meno': 'Juraj Janosik',
    'adresa': {'ulica': 'Pod sibenicou',
               'cislo': 1,
               'obec': 'Liptovsky Mikulas'},
    'narodeny': {'datum': {'den': 25, 'mesiac': 1, 'rok': 1688},
                 'obec': 'Terchova'}
}
student3 = {
    'meno': 'Margita Figuli',
    'adresa': {'ulica': 'Sturova',
               'cislo': 4,
               'obec': 'Bratislava'},
    'narodeny': {'datum': {'den': 2, 'mesiac': 10, 'rok': 1909},
                 'obec': 'Vysny Kubin'}
}
student4 = {
    'meno': 'Ludovit Stur',
    'adresa': {'ulica': 'Slovenska',
               'cislo': 12,
               'obec': 'Modra'},
    'narodeny': {'datum': {'den': 28, 'mesiac': 10, 'rok': 1815},
                 'obec': 'Uhrovec'}
}

skola = [student1, student2, student3, student4]
```

Vytvorili sme 4-prvkový zoznam, v ktorom je každý prvok typu slovník. V týchto slovníkoch sú po 3 kľúče `'meno'`, `'adresa'`, `'narodeny'`, pričom dva z nich majú hodnoty opäť slovníky. Môžeme zapísať, napr.

```
>>> for st in skola:
        print(st['meno'], 'narodeny v', st['narodeny']['obec'])

Janko Hrasko narodeny v Korytovce
Juraj Janosik narodeny v Terchova
Margita Figuli narodeny v Vysny Kubin
Ludovit Stur narodeny v Uhrovec
```

Získali sme nielen mená všetkých študentov v tomto zozname ale aj ich miesto narodenia.

19.3 Textové súbory JSON

Ak pythonovská štruktúra obsahuje iba:

- zoznamy `list`
- slovníky `dict` s kľúčmi, ktoré sú reťazce
- znakové reťazce `str`

- celé alebo desatinné čísla `int` a `float`
- logické hodnoty `True` alebo `False`

môžeme túto štruktúru (napr. zoznam `skola`) zapísať do špeciálneho súboru:

```
import json

with open('subor.txt', 'w') as subor:
    json.dump(skola, subor)
```

Takto vytvorený súbor je dosť nečitateľný, čo môžeme zmeniť parametrom `indent`:

```
with open('subor.txt', 'w') as subor:
    json.dump(skola, subor, indent=2)
```

Prečítať takýto súbor a zrekonštruovať z neho celú štruktúru je potom veľmi jednoduché:

```
>>> precitane = json.load(open('subor.txt'))
>>> print(skola == precitane)
True
```

Vytvorila sa nová štruktúra `precitane`, ktorá má presne rovnaký obsah, ako zapísaný zoznam `skola`.

19.4 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Odmerajte čas, koľko trvá vygenerovať `n`-prvkové náhodný zoznam čísel. Testujte pre rôzne `n`: 1000, 10000, 100000, 1000000.
2. Funkcia `nahodne_utriedene(n)` vygeneruje `n`-prvkový zoznam náhodných hodnôt od 0 do `2*n-1`, pričom tento zoznam bude vzostupne usporiadaný a nebude obsahovať viac rovnakých prvkov. Výsledkom (`return`) funkcie je tento zoznam. Odmerajte čas, koľko trvá vygenerovať rôzne veľké zoznamy, napr. pre `n` 100, 1000, 10000, 100000.
3. Vygenerujte 20-prvkový náhodný zoznam čísel od 10 do 99, utrieďte ho a spustite na ňom algoritmus **binárneho vyhľadávania** pre nejaké konkrétne hodnoty. Doplňte túto jeho funkciu `hladaj()` o kontrolné výpisy:
   - na začiatok `while`-cyklu za výpočet hodnoty `stred`, vložte výpis celého zoznamu do jedného riadka a pod neho vyznačte pozície `z`, `s` a `k` pre momentálny začiatok, stred a koniec intervalu, napr. pri hľadaní čísla 50, to môže vyzerať takto:

```
10 25 30 32 43 45 51 53 58 59 63 65 70 81 82 85 90 97 99 99
 z                             s                             k
10 25 30 32 43 45 51 53 58 59 63 65 70 81 82 85 90 97 99 99
 z           s           k
10 25 30 32 43 45 51 53 58 59 63 65 70 81 82 85 90 97 99 99
                  z  s     k
10 25 30 32 43 45 51 53 58 59 63 65 70 81 82 85 90 97 99 99
                  z  s  k
nenasiel
```

4. Odmerajte čas, koľko priemerne bude trvať vyhľadanie nejakej hodnoty pomocou **binárneho vyhľadávania**. Použite podobný test, ako sa robil pre sekvenčné hľadanie. Využite funkciu `nahodne_utriedene(n)` z úlohy (2).
5. Zadefinujte slovník `vysky` tak, že kľúčmi budú mená vašich 10 kolegov (môžete si vymyslieť) a hodnotami ich približné výšky v cm.
   - napr.

```
>>> vysky = {'Igor':197, 'Mariana':160, 'Adam':171, ...}
>>> vysky
```

6. Vypíšte váš slovník z úlohy (5) tak, že v každom riadku bude jedno meno a príslušná výška, pričom výpis bude usporiadaný podľa abecedy.
7. Podobne ako v úlohe (6) vypíšete slovník `vysky`, ale výpis bude usporiadaný podľa výšok od najmenšej po najväčšiu.
8. Napíšte funkciu `priemer(vysky)`, ktorá vypočíta priemer všetkých výšok zo slovníka `vysky` z úlohy (5).
9. Napíšte funkciu `vypis(vysky)`, ktorá z daného slovníka vypíše všetky dvojice (kľúč, hodnota) - každú do jedného riadka, pričom do každého riadka pripíše jedno zo slov `'priemer'`, `'podpriemer'` alebo `'nadpriemer'`, podľa toho či sa daná výška rovná priemernej, alebo je menšia ako priemer, alebo je väčšia.
10. Napíšte funkciu `len_od_do(vysky, od, do)`, ktorá z daného slovníka vytvorí nový (pôvodný nechá bez zmeny), v ktorom budú len tie dvojice (kľúč, hodnota), ktorých výška je z intervalu `<od, do>`. Napr. `len_od_do(vysky, 170, 179)` vytvorí nový slovník, ktorý bude obsahovať len tých kolegov z pôvodného slovníka `vysky`, ktorých výška nie je menšia ako 170 a nie je väčšia ako 179 cm.
11. Napíšte funkciu `prevrat(vysky)`, ktorá z daného slovníka `vysky` skonštruuje nový slovník. Kľúčmi v tomto novom slovníku budú výšky a hodnotami budú zoznamy (alebo množiny, rozhodnite sa) všetkých tých kolegov zo slovníka `vysky`, ktorí majú túto výšku.
12. Napíšte funkciu `dve_kocky(n)`, ktorá bude simulovať hody dvoch hracích kociek (s číslami od 1 do 6) a evidovať si, koľkokrát padol ktorý súčet. Zrejme súčty budú čísla od 2 do 12. Funkcia bude simulovať `n` takýchto hodov dvomi kockami a vráti frekvenčnú tabuľku (`return`). Funkcia nemusí nič vypisovať.
13. Napíšte funkciu `farba(retʼazec)`, ktorá z daného reťazca - mena farby v slovenčine vráti správny názov farby pre `tkinter`. Ak danú farbu nerozpozná, vráti farbu `'pink'`. Funkcia by mala akceptovať tieto slovenské mená farieb: `'biela'`, `'cierna'`, `'cervena'`, `'modra'`, `'zlta'`, `'zelena'` (môžete si podľa vlastného uváženia doplniť ďalšie). Vo funkcii nepoužite príkaz `if`.
14. Napíšte funkciu `pretypuj(nazov, hodnota)`, ktorá na základe názvu typu pretypuje danú hodnotu. Vo funkcii nepoužite príkaz `if`. Ak sa dané pretypovanie urobiť nedá, funkcia by mala spadnúť na zodpovedajúcej chybe. Funkcia by mala akceptovať tieto názvy typov: `'int'`, `'float'`, `'list'`, `'tuple'`, `'str'`, `'set'`. Pre neznámy názov typu by funkcia mala spadnúť na chybe `KeyError`.
   - napr.

```
>>> pretypuj('str', 3.14)
'3.14'
>>> pretypuj('set', 'Python')
{'t', 'y', 'P', 'h', 'o', 'n'}
>>> pretypuj('float', '1e5')
100000.0
```

15. Napíšte funkciu `opakuju(meno_suboru)`, ktorá vypíše všetky tie riadky daného textového súboru, ktoré sa v tomto súbore vyskytujú aspoň trikrát. Každý takýto opakujúci sa riadok vypíšte len raz.
16. Na prednáške sa zisťovala frekvenčná tabuľka písmen, resp. slov vo vete. Napíšte funkciu `dvojice(meno_suboru)`, ktorá z daného súboru slov zistí počet výskytov všetkých za sebou idú-

cich **dvojíc písmen**, napr. pre slova ‚laska' bude akceptovať tieto štyri dvojice ‚la', ‚as', ‚sk' a ‚ka'. Funkcia vráti frekvenčnú tabuľku ako slovník. Otestujte na súboroch dobs.txt alebo twain.txt a zistite 10 najčastejších dvojíc písmen.

17. Uložte slovník, ktorý je výsledkom úlohy (11) - prevrátený slovník s výškami kolegov do textového súboru vo formáte `json`. Potom tento súbor otvorte v nejakom textovom editore, cez clipboard preneste jeho obsah do Pythonu a priraďte hodnotu do nejakej premennej. Porovnajte jej obsah.

    - napr.

```
>>> a = prevrat(vysky)
>>> ... ulož do json súboru ...
>>> b = ... prenesený obsah json súboru ...
>>> a == b
???
```

KAPITOLA 20

20. Funkcie a parametre

20.1 Parametre funkcií

Zapíšme funkciu, ktorá vypočíta súčin niekoľkých čísel. Aby sme ju mohli volať s rôznym počtom parametrov, využijeme náhradné hodnoty:

```
def sucin(a=1, b=1, c=1, d=1, e=1):
    return a * b * c * d * e
```

Túto funkciu môžeme volať aj bez parametrov, ale nefunguje viac ako 5 parametrov:

```
>>> sucin(3, 7)
21
>>> sucin(2, 3, 4)
24
>>> sucin(2, 3, 4, 5, 6)
720
>>> sucin(2, 3, 4, 5, 6, 7)
...
TypeError: sucin() takes from 0 to 5 positional arguments but 6 were given
>>> sucin()
1
>>> sucin(13)
13
```

Ak chceme použiť aj väčší počet parametrov, môžeme využiť zoznam:

```
def sucin(zoznam):
    vysl = 1
    for prvok in zoznam:
        vysl *= prvok
    return vysl
```

Teraz to funguje pre ľubovoľný počet čísel, ale musíme ich uzavrieť do hranatých (alebo okrúhlych) zátvoriek:

```
>>> sucin([3, 7])
21
>>> sucin([2, 3, 4, 5, 6])
720
>>> sucin((2, 3, 4, 5, 6, 7))
5040
>>> sucin(range(2, 8))
5040
>>> sucin(range(2, 41))
815915283247897734345611269596115894272000000000
```

Namiesto zoznamu môžeme ako parameter poslať aj `range(2, 8)`, t.j. ľubovoľnú štruktúru, ktorá sa dá rozobrať pomocou for-cyklu.

20.1.1 Zbalené a rozbalené parametre

Predchádzajúce riešenie stále nerieši náš problém: funkciu s ľubovoľným počtom parametrov. Na toto slúži tzv. **zbalený parameter** (po anglicky packing):

- pred menom parametra v hlavičke funkcie píšeme znak `*` (zvyčajne je to posledný parameter)
- pri volaní funkcie sa všetky zvyšné parametre **zbalia** do jednej n-tice (typ `tuple`)

Otestujme:

```
def test(prvy, *zvysne):
    print('prvy =', prvy)
    print('zvysne =', zvysne)
```

po spustení:

```
>>> test('jeden', 'dva', 'tri')
prvy = jeden
zvysne = ('dva', 'tri')
>>> test('jeden')
prvy = jeden
zvysne = ()
```

Funkcia sa môže volať s jedným alebo aj viac parametrami. Prepíšme funkciu `sucin()` s použitím jedného zbaleného parametra:

```
def sucin(*ntica):              # zbalený parameter
    vysl = 1
    for prvok in ntica:
        vysl *= prvok
    return vysl
```

Uvedomte si, že teraz jeden parameter `ntica` zastupuje **ľubovoľný počet parametrov** a Python nám do tohto parametra automaticky zbalí všetky skutočné parametre ako jednu n-ticu (`tuple`). Otestujeme:

```
>>> sucin()
1
>>> sucin(3, 7)
21
>>> sucin(2, 3, 4, 5, 6, 7)
5040
>>> sucin(2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
479001600
>>> sucin(range(2, 13))
...
TypeError: unsupported operand type(s) for *=: 'int' and 'range'
```

V poslednom príklade vidíte, že `range(...)` tu nefunguje: Python tento jeden parameter zbalí do jednoprvkovej n-tice a potom sa s týmto `range()` bude chcieť násobiť, čo samozrejme nefunguje.

Ešte ukážme druhý podobný prípad, ktorý sa môže vyskytnúť pri práci s parametrami funkcií. Napíšme funkciu, ktorá dostáva dva alebo tri parametre a nejako ich vypíše:

```
def pis(meno, priezvisko, rok=2015):
    print(f'volam sa {meno} {priezvisko} a narodil som sa v {rok}')
```

Napr.

```
>>> pis('Janko', 'Hrasko', 2014)
volam sa Janko Hrasko a narodil som sa v 2014
>>> pis('Juraj', 'Janosik')
volam sa Juraj Janosik a narodil som sa v 2015
```

Malá nepríjemnosť nastáva vtedy, keď máme takéto hodnoty pripravené v nejakej štruktúre:

```
>>> p1 = ['Janko', 'Hrasko', 2014]
>>> p2 = ['Juraj', 'Janosik']
>>> p3 = ['Monty', 'Python', 1968]
>>> pis(p1)
...
TypeError: pis() missing 1 required positional argument: 'priezvisko'
```

Túto funkciu nemôžeme volať s trojprvkovým zoznamom, ale musíme prvky tohto zoznamu **rozbaliť**, aby sa priradili do príslušných parametrov, napr.

```
>>> pis(p1[0], p1[1], p1[2])
volam sa Janko Hrasko a narodil som sa v 2014
>>> pis(p2[0], p2[1])
volam sa Juraj Janosik a narodil som sa v 2015
```

Takáto situácia sa pri programovaní stáva dosť často: v nejakej štruktúre (napr. v zozname) máme pripravené parametre pre danú funkciu a my potrebujeme túto funkciu zavolať s rozbalenými prvkami štruktúry. Na toto slúži **rozbaľovací operátor**, pomocou ktorého môžeme ľubovoľnú štruktúru poslať ako skupinu parametrov, pričom sa automaticky rozbalia (a teda prvky sa priradia do formálnych parametrov). Rozbaľovací operátor pre parametre je opäť znak `*` a používa sa takto:

```
>>> pis(*p1)              # je to isté ako pis(p1[0], p1[1], p1[2])
volam sa Janko Hrasko a narodil som sa v 2014
>>> pis(*p2)              # je to isté ako pis(p2[0], p2[1])
volam sa Juraj Janosik a narodil som sa v 2015
```

Takže, všade tam, kde sa očakáva nie jedna štruktúra ako parameter, ale veľa parametrov, ktoré sú prvkami tejto štruktúry, môžeme použiť tento rozbaľovací operátor (po anglicky *unpacking argument lists*).

Tento operátor môžeme využiť napr. aj v takýchto situáciách:

```
>>> print(range(10))
range(0, 10)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> print(*range(10))
0 1 2 3 4 5 6 7 8 9
>>> print(*range(10), sep='...')
0...1...2...3...4...5...6...7...8...9
>>> param = (3, 20, 4)
>>> print(*range(*param))
3 7 11 15 19
>>> dvenasto = 2**100
>>> print(dvenasto)
1267650600228229401496703205376
>>> print(*str(dvenasto))
1 2 6 7 6 5 0 6 0 0 2 2 8 2 2 9 4 0 1 4 9 6 7 0 3 2 0 5 3 7 6
>>> print(*str(dvenasto), sep='-')
1-2-6-7-6-5-0-6-0-0-2-2-8-2-2-9-4-0-1-4-9-6-7-0-3-2-0-5-3-7-6
>>> p = [17, 18, 19, 20, 21]
>>> [*p[3:], *range(5), *p]
[20, 21, 0, 1, 2, 3, 4, 17, 18, 19, 20, 21]
```

Pripomeňme si funkciu `sucin()`, ktorá počítala súčin ľubovoľného počtu čísel - tieto sa spracovali jedným zbaleným parametrom. Teda funkcia očakáva veľa parametrov a niečo z nich vypočíta. Ak ale máme jednu štruktúru, ktorá obsahuje tieto čísla, musíme použiť rozbaľovací operátor:

```
>>> cisla = [7, 11, 13]
>>> sucin(cisla)              # zoznam [7, 11, 13] sa násobí 1
[7, 11, 13]
>>> sucin(*cisla)
1001
>>> sucin(*range(2, 11))
3628800
```

20.1.2 Parameter s meniteľnou hodnotou

Teraz trochu odbočíme od zbalených a rozbalených parametrov. Ukážme veľký problém, ktorý nás môže zaskočiť v situácii, keď náhradnou hodnotou parametra je meniteľný typ (mutable). Pozrime na túto nevinne vyzerajúcu funkciu:

```
def pokus(a=1, b=[]):
    b.append(a)
    return b
```

Očakávame, že ak neuvedieme druhý parameter, výsledkom funkcie bude jednoprvkový zoznam s prvkom prvého parametra. Skôr, ako to otestujeme, vypíšme, ako túto našu funkciu vidí `help()`:

```
>>> help(pokus)
Help on function pokus in module __main__:

pokus(a=1, b=[])
```

a teraz test:

```
>>> pokus(2)
[2]
```

Zatiaľ je všetko v poriadku. Ale po druhom spustení:

```
>>> pokus(7)
[2, 7]
```

Vidíme, že Python si tu nejako pamätá aj naše prvé spustenie tejto funkcie. Znovu pozrime `help()`:

```
>>> help(pokus)
Help on function pokus in module __main__:

pokus(a=1, b=[2, 7])
```

A vidíme, že sa dokonca zmenila hlavička našej funkcie `pokus()`. Mali by sme teda rozumieť, čo sa tu vlastne deje:

- Python si pre každú funkciu pamätá zoznam všetkých náhradných hodnôt pre formálne parametre funkcie, tak ako sme ich zadefinovali v hlavičke (môžete si pozrieť premennú `pokus.__defaults__`)
- ak sú v tomto zozname len nemeniteľné hodnoty (immutable), nevzniká žiaden problém
- problémom sú meniteľné hodnoty (mutable) v tomto zozname: pri volaní funkcie, keď treba použiť náhradnú hodnotu, Python použije hodnotu z tohto zoznamu (použije referenciu na túto štruktúru) - keď tomuto parametru ale v tele funkcie zmeníme obsah, zmení sa tým aj hodnota v zozname náhradných hodnôt (`pokus.__defaults__`)

Z tohto pre nás vyplýva, že radšej nikdy nebudeme definovať náhradnú hodnotu parametra ako meniteľný objekt. Funkciu `pokus` by sme mali radšej zapísať takto:

```
def pokus(a=1, b=None):
    if b is None:
        b = []
    b.append(a)
    return b
```

A všetko by fungovalo tak, ako sme očakávali.

Skúsení programátori vedia túto vlastnosť využiť veľmi zaujímavo. Napr. do funkcie posielame nejaké hodnoty a funkcia nám oznamuje, či už sa taká vyskytla, alebo ešte nie:

```
def kontrola(hodnota, bola=set()):
    if hodnota in bola:
        print(hodnota, 'uz bola')
    else:
        bola.add(hodnota)
        print(hodnota, 'OK')
```

a test:

```
>>> kontrola(7)
7 OK
>>> kontrola(17)
17 OK
>>> kontrola(-7)
-7 OK
>>> kontrola(17)
17 uz bola
>>> kontrola(7)
7 uz bola
```

Veľmi pekným využitím tejto nečakanej vlastnosti parametra s meniteľnou náhradnou hodnotou je zrýchlenie výpočtu fibonacciho postupnosti. Už sme sa stretli s rekurzívnou verziou, ktorá je pre väčšie hodnoty nepoužiteľne pomalá:

```
def fib(n):
    if n < 2:
        return n
    return fib(n-2) + fib(n-1)
```

Vyskúšajte napr. `fib(40)`.

Tu by mohol pomôcť jeden parameter navyše, vďaka ktorému by si funkcia mohla pamätať všetky doteraz vypočítané hodnoty. Zapíšme:

```
def fib(n, pamat={}):
    if n in pamat:
        return pamat[n]
    if n < 2:
        vysl = n
    else:
        vysl = fib(n-2) + fib(n-1)
    pamat[n] = vysl
    return vysl
```

Aj táto funkcia je rekurzívna, len si vie zapamätať niečo navyše. Takému spôsobu riešenia úlohy, pri ktorom vieme využiť až predtým vypočítané a zapamätané medzivýsledky, hovoríme **memoizácia** a budete sa to učiť vo vyšších ročníkoch.

20.1.3 Zbalené pomenované parametre

Pozrime sa na túto funkciu:

```
def vypis(meno, vek, vyska, vaha, bydlisko):
    print('volam sa', meno)
    print('    vek =', vek)
    print('    vyska =', vyska)
    print('    vaha =', vaha)
    print('    bydlisko =', bydlisko)
```

otestujeme:

```
>>> vypis('Janko Hrasko', vek=5, vyska=7, vaha=0.3, bydlisko='Pri poli')
volam sa Janko Hrasko
    vek = 5
    vyska = 7
    vaha = 0.3
    bydlisko = Pri poli
```

Radi by sme aj tu dosiahli podobnú vlastnosť parametrov, ako to bolo pri zbalenom parametri, ktorý do jedného parametra dostal ľubovoľný počet skutočných parametrov. V tomto prípade by sme ale chceli, aby sa takto zbalili všetky vlastnosti vypisovanej osoby ale aj s príslušnými menami týchto vlastností. V tomto prípade nám pomôžu **zbalené pomenované parametre** (keyword argument packing): namiesto viacerých pozičných parametrov, uvedieme jeden s dvomi hviezdičkami `**`:

```
def vypis(meno, **vlastnosti):
    print('volam sa', meno)
    for k, h in vlastnosti.items():
        print('    ', k, '=', h)
```

Tento zápis označuje, že ľubovoľný počet pomenovaných parametrov sa zbalí do jedného parametra a ten vo vnútri funkcie bude typu **slovník** (asociatívne pole `dict`). Uvedomte si ale, že v slovníku sa nezachováva poradie dvojíc:

```
>>> vypis('Janko Hrasko', vek=5, vyska=7, vaha=0.3, bydlisko='Pri poli')
volam sa Janko Hrasko
     vyska = 7
     vaha = 0.3
     bydlisko = Pri poli
     vek = 5
```

Ďalší príklad tiež ilustruje takýýto zbalený slovník:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()

def kruh(r, x, y):
    canvas.create_oval(x-r, y-r, x+r, y+r)

kruh(50, 100, 100)
```

Funkcia `kruh()` definuje nakreslenie kruhu s daným polomerom a stredom, ale nijako nevyužíva ďalšie parametre na definovanie farieb a obrysu kruhu. Doplňme do funkcie zbalené pomenované parametre:

```
def kruh(r, x, y, **param):
    canvas.create_oval(x-r, y-r, x+r, y+r)
```

Toto označuje, že `kruh()` môžeme zavolať s ľubovoľnými ďalšími pomenovanými parametrami, napr. `kruh(... , fill='red', width=7)`. Tieto parametre ale chceme ďalej poslať do funkcie `create_oval()`. Určite sem nemôžeme poslať `param`, lebo toto je premenná typu `dict` a `create_oval()` s tým pracovať nevie. Tu by sa nám zišlo premennú `param` rozbaliť do viacerých pomenovaných parametrov: Rozbaľovací operátor pre pomenované parametre sú dve hviezdičky `**`, teda zapíšeme:

```
def kruh(r, x, y, **param):
    canvas.create_oval(x-r, y-r, x+r, y+r, **param)
```

a teraz funguje aj

```
kruh(50, 100, 100)
kruh(30, 150, 100, fill='red')
kruh(100, 200, 200, width=10, outline='green')
```

Takýto rozbaľovací parameter by sme vedeli využiť aj v predchádzajúcom príklade s funkciou `vypis()`:

```
>>> p1 = {'meno':'Janko Hrasko', 'vek':5, 'vyska':7, 'vaha':0.3, 'bydlisko':'Pri poli
↪'}
>>> vypis(**p1)
volam sa Janko Hrasko
     vaha = 0.3
     vek = 5
     vyska = 7
     bydlisko = Pri poli
>>> p2 = {'vek':25, 'narodeny':'Terchova', 'popraveny':'Liptovsky Mikulas'}
>>> vypis('Juraj Janosik', **p2)
volam sa Juraj Janosik
     popraveny = Liptovsky Mikulas
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    vek = 25
    narodeny = Terchova
```

20.2 Funkcia ako hodnota

v Pythone sú aj funkcie objektami a môžeme ich priradiť do premennej, napr.

```
>>> def fun1(x): return x*x
>>> fun1(7)
49
>>> cojaviem = fun1
>>> cojaviem(8)
64
```

Funkcie môžu byť prvkami zoznamu, napr.

```
>>> def fun2(x): return 2*x+1
>>> def fun3(x): return x//2
>>> zoznam = [fun1, fun2, fun3]
>>> for f in zoznam:
        print(f(10))
100
21
5
```

Funkciu môžeme poslať ako parameter do inej funkcie, napr.

```
>>> def urob(fun, x):
        return fun(x)
>>> urob(fun2, 3.14)
7.28
```

Funkcia (teda referencia na funkciu) môže byť aj prvkom slovníka. Pekne to ilustruje príklad s korytnačkou:

```
def vykonaj():
    t = turtle.Turtle()
    p = {'fd': t.fd, 'rt': t.rt, 'lt': t.lt}
    while True:
        prikaz, parameter = input('> ').split()
        p[prikaz](int(parameter))
```

a funguje napr.

```
>>> vykonaj()
> fd 100
> lt 90
> fd 50
> rt 60
> fd 100
```

20.2.1 Anonymné funkcie

Často sa namiesto jednoriadkovej funkcie, ktorá počíta jednoduchý výraz a tento vráti ako výsledok (`return`) používa špeciálna konštrukcia `lambda`. Tá vygeneruje tzv. anonymnú funkciu, ktorú môžeme priradiť do premennej alebo poslať ako parameter do funkcie, napr.

```
>>> urob(lambda x: 2*x+1, 3.14)
7.28
```

Tvar konštrukcie `lambda` je nasledovný:

```
lambda parametre: výraz
```

Tento zápis nahrádza definovanie funkcie:

```
def anonymne_meno(parametre):
    return vyraz
```

Môžeme zapísať napr.

```
lambda x: x % 2==0              # funkcia vráti True pre párne číslo
lambda x, y: x ** y             # vypočíta príslušnú mocnimu čísla
lambda x: isinstance(x, int)    # vráti True pre celé číslo
```

20.2.2 Mapovacie funkcie

Ideu funkcie ako parametra najlepšie ilustruje takáto funkcia `mapuj()`:

```
def mapuj(fun, zoznam):
    vysl = []
    for prvok in zoznam:
        vysl.append(fun(prvok))
    return vysl
```

Funkcia aplikuje danú funkciu (prvý parameter) na všetky prvky zoznamu a z výsledkov poskladá nový zoznam, napr.

```
>>> mapuj(fun1, (2, 3, 7))
[4, 9, 49]
>>> mapuj(list, 'Python'))
[['P'], ['y'], ['t'], ['h'], ['o'], ['n']]
>>> mapuj(lambda x: [x] * x, range(1, 6))
[[1], [2, 2], [3, 3, 3], [4, 4, 4, 4], [5, 5, 5, 5, 5]]
```

V Pythone existuje štandardná funkcia `map()`, ktorá robí skoro to isté ako naša funkcia `mapuj()` ale s tým rozdielom, že `map()` nevracia zoznam, ale niečo ako generátorový objekt, ktorý môžeme použiť ako prechádzanú postupnosť vo for-cykle, alebo napr. pomocou `list()` ho previesť na zoznam, napr.

```
>>> list(map(int, str(2 ** 30)))
[1, 0, 7, 3, 7, 4, 1, 8, 2, 4]
```

Vráti zoznam cifier čísla `2**30`.

Podobná funkcii `mapuj()` je aj funkcia `filtruj()`, ktorá z daného zoznamu vyrobí nový nový zoznam, ale nechá v ňom len tie prvky, ktoré spĺňanú nejakú podmienku. Podmienka je definovaná funkciou, ktorá je prvým parametrom:

```
def filtruj(fun, zoznam):
    vysl = []
    for prvok in zoznam:
        if fun(prvok):
            vysl.append(prvok)
    return vysl
```

Napr.

```
>>> def podm(x): return x % 2==0      # zistí, či je číslo párne
>>> list(range(1, 20, 3))
[1, 4, 7, 10, 13, 16, 19]
>>> mapuj(podm, range(1, 20, 3))
[False, True, False, True, False, True, False]
>>> filtruj(podm, range(1, 20, 3))
[4, 10, 16]
```

Podobne ako pre `mapuj()` existuje štandardná funkcia `map()`, aj pre `filtruj()` existuje štandardná funkcia `filter()` - tieto dve funkcie ale nevracajú zoznam (`list`) ale postupnosť, ktorá sa dá prechádzať for-cyklom alebo poslať ako parameter do funkcie, kde sa očakáva postupnosť.

Ukážkovým využitím funkcie `map()` je funkcia, ktorá počíta ciferný súčet nejakého čísla:

```
def cs(cislo):
    return sum(map(int, str(cislo)))
```

```
>>> cs(1234)
10
```

20.3 Generátorová notácia

Veľmi podobná funkcii `map()` je generátorová notácia (po anglicky **list comprehension**):

- je to spôsob, ako môžeme elegantne vygenerovať nejaký zoznam pomocou for-cyklu a nejakého výrazu
- do `[...]` nezapíšeme prvky zoznamu, ale predpis, akým sa majú vytvoriť
- základný tvar je tohto zápisu:

```
[vyraz for i in postupnost]
```

- kde `výraz` najčastejšie obsahuje premennú cyklu a `postupnost'` je ľubovoľná štruktúra, ktorá sa dá prechádzať for-cyklom (napr. `list`, `set`, `str`, `range()`, riadky otvoreného súboru, ale aj výsledok `map()` a `filter()` a pod.
- táto notácia môže používať aj vnorené cykly ale aj podmienku `if`, vtedy je to v takomto tvare:

```
[vyraz for i in postupnost if podmienka]
```

alebo

```
[vyraz for i in postupnost for j in postupnost]
```

alebo

```
[vyraz for i in postupnost for j in postupnost if podmienka]
```

a podobne

- generátorová notácia s podmienkou nechá vo výsledku len tie prvky, ktoré spĺňajú danú podmienku

Niekoľko príkladov:

```
>>> [i**2 for i in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> [i*j for i in range(1, 5) for j in range(1, 5)]
[1, 2, 3, 4, 2, 4, 6, 8, 3, 6, 9, 12, 4, 8, 12, 16]
>>> [[i*j for i in range(1, 5)] for j in range(1, 5)]
[[1, 2, 3, 4], [2, 4, 6, 8], [3, 6, 9, 12], [4, 8, 12, 16]]
>>> [[i*j] for i in range(1, 5) for j in range(1, 5)]
[[1], [2], [3], [4], [2], [4], [6], [8], [3], [6], [9], [12], [4], [8], [12], [16]]
>>> [i for i in range(100) if cs(i)==5]      # cs() vypočíta ciferný súčet
[5, 14, 23, 32, 41, 50]
>>> ''.join(2*znak for znak in 'python')
'ppyytthhoonn'
```

Pomocou tejto konštrukcie by sme vedeli zapísať aj mapovacie funkcie:

```
def mapuj(fun, zoznam):
    return [fun(prvok) for prvok in zoznam]

def filtruj(fun, zoznam):
    return [prvok for prvok in zoznam if fun(prvok)]
```

Všimnite si, že funkcia `filtruj()` využíva `if`, ktorý je vo vnútri generátorovej notácie.

20.4 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/
- pozrite si *Riešenie 20. cvičenia*

20.4.1 Zbalené a rozbalené parametre

1. Napíšte funkciu `ntica`, ktorá bude mať ľubovoľný počet parametrov a vráti n-ticu z týchto parametrov
   - napr.

   ```
   >>> ntica(5, 'x', 7)
   (5, 'x', 7)
   >>> p = ntica(123, ntica(37))
   >>> p
   (123, (37,))
   ```

2. Napíšte funkciu `min` s ľubovoľným počtom parametrov, ktorá vráti najmenšiu hodnotu medzi parametrami.
   - napr.

```
def min(...):
    ...
>>> min(5, 3.14, 7)
3.14
```

3. Napíšte funkciu `zisti` s ľubovoľným počtom parametrov, ktorá zistí (vráti `True`), či aspoň jeden z parametrov je n-tica (typ `tuple`).

   - napr.

```
>>> zisti(1, 2, 3)
False
>>> t = zisti(())
>>> t
True
```

4. Napíšte funkciu `zlep` s ľubovoľným počtom parametrov, pričom všetky sú typu `list`. Výsledkom funkcie je zreťazenie všetkých týchto parametrov.

   - napr.

```
>>> zlep(['a', 1], [], [('b', 2)])
['a', 1, ('b', 2)]
>>> zlep()
[]
```

5. Napíšte funkciu `vypis(zoznam)`, ktorá pomocou `print` vypíše všetky prvky zoznamu do jedného riadka. Nepoužite for-cyklus.

   - napr.

```
>>>vypis([123, 'ahoj', (50, 120), 3.14])
123 ahoj (50, 120) 3.14
```

20.4.2 Funkcie ako parametre

6. Napíšte funkciu `retazec(zoznam)`. Funkcia vráti znakový reťazec, ktorý reprezentuje prvky zoznamu. Prvky zoznamu budú v reťazci oddelené znakom bodkočiarka. Nepoužite žiadne cykly, ale namiesto toho štandardnú funkciu `map` a metódu `join`.

   - napr.

```
>>> r = retazec([123, 'ahoj', (50, 120), 3.14])
>>> r
"123; 'ahoj'; (50, 120); 3.14"
```

7. Napíšte funkciu `aplikuj`, ktorej parametrami sú nejaké funkcie, okrem posledného parametra, ktorým je nejaká hodnota. Funkcia postupne zavolá všetky tieto funkcie s danou hodnotou, pričom každú ďalšiu funkciu aplikuje na predchádzajúci výsledok. Napr. `aplikuj(f1, f2, f3, x)` vypočíta `f3(f2(f1(x)))`. Funkcia by mala pracovať pre ľubovoľný nenulový počet parametrov.

   - napr.

```
>>> def rev(x): return x[::-1]
>>> aplikuj(str, rev, int, 1074)
4701
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> aplikuj(abs, lambda x: x+1, -17)
18
```

8. Napíšte funkciu `urob(k)`. Funkcia ako svoj výsledok **vráti funkciu** s jedným parametrom, ktorá bude počítať `k`-tu mocninu parametra.

   - napr.

```
>>> g = urob(3)    # g je funkcia, ktora pocita 3 mocninu
>>> g(4)
64
>>> urob(3)(4)     # tu sa pocita to iste
64
>>> f = urob(7)
>>> print(f(2), f(3), f(4))
127 2187 16384
```

   - funkcia v svojom tele zadefinuje vnorenú pomocnú funkciu, ktorá počíta `k`-tu mocninu svojho poarametra a túto vnorenú funkciu vráti ako výsledok funkcie `urob()`

20.4.3 Generátorová notácia

9. Napíšte funkciu `mocniny(n)`, ktorá vráti zoznam druhých mocnín čísel od 1 do `n`.

   - napr.

```
>>> mocniny(4)
[1, 4, 9, 16]
```

10. Napíšte funkciu `zistí(veta)`, ktorá zistí dĺžku najdlhšieho slova vo vete.

    - napr.

```
>>> zisti('isiel macek do malacek')
7
```

11. Napíšte funkciu `prevrat_slova(veta)`, ktorá vráti zadanú vetu tak, že každé slovo v nej bude otočené.

    - napr.

```
>>> prevrat_slova('isiel macek do malacek')
'leisi kecam od kecalam'
```

    - pokúste sa celú funkciu zapísať len do jedného riadka (len jeden `return`)

12. Napíšte funkciu `najdlhsie_slovo(veta)`, ktorá vráti najdlhšie slovo vo vete. Riešte to takto:

    - funkcia najprv z danej vety vytvorí postupnosť dvojíc (dĺžka slova, slovo)
    - pomocou `sorted()` túto postupnosť dvojíc utriedi
    - funkcia vráti slovo z poslednej dvojice (je to najdlhšie slovo) utriedenej postupnosti
    - napr.

```
>>> najdlhsie_slovo('isiel macek do malacek')
'malacek'
```

    - pokúste sa celú funkciu zapísať do jedného riadka (len jeden `return`)

13. Napíšte funkciu `zoznam2(m, n, hodnota=None)`, ktorá vygeneruje dvojrozmerný zoznam veľkosti `m` x `n` pričom všetky prvky majú zadanú hodnotu

    - napr.

```
>>> zoznam2(3, 2, 1)            # 3 riadky po 2 prvky s hodnotami 1
[[1, 1], [1, 1], [1, 1]]
```

14. Predpokladáme, že textový súbor v každom riadku obsahuje niekoľko celých čísel. Napíšte funkciu `citaj_zoznam(meno_suboru)`, ktorá z neho vytvorí dvojrozmerný zoznam čísel.

    - napr. pre súbor:

```
1 2
3 4 5 6

7 8 9
```

    - vytvorí:

```
>>> citaj_zoznam('subor.txt')
[[1, 2], [3, 4, 5, 6], [], [7, 8, 9]]
```

15. Napíšte funkciu `rozdel(zoznam, x)`, ktorá ako výsledok vráti dva zoznamy: prvý obsahuje všetky menšie prvky ako `x` a druhý všetky zvyšné.

    - napr.

```
>>> p1, p2 = rozdel([6, 8, 4, 7, 11, 9], 7)
>>> p1
[6, 4]
>>> p2
[8, 7, 11, 9]
```

16. Už poznáme **štandardnú funkciu** `enumerate()`, ktorá z danej postupnosti vracia postupnosť dvojíc. Napíšte vlastnú funkciu `enumerate(postupnost)`, ktorá vytvorí takýto zoznam dvojíc (`list` s prvkami `tuple`): prvým prvkom bude poradové číslo dvojice a druhým prvkom prvok zo vstupnej postupnosti.

    - napr.

```
def enumerate(postupnost):
    return ...

>>> enumerate('python')
[(0, 'p'), (1, 'y'), (2, 't'), (3, 'h'), (4, 'o'), (5, 'n')]
```

17. Napíšte funkciu `zip(p1, p2)`, ktorá z dvoch postupností rovnakých dĺžok vytvorí zoznam zodpovedajúcich dvojíc, t.j. zoznam v ktorom prvým prvkom bude dvojica prvých prvkov postupností, druhým prvkom dvojica druhých prvkov, ...

    - napr.

```
>>> zip('python', [2, 3, 5, 7, 11, 13])
[('p', 2), ('y', 3), ('t', 5), ('h', 7), ('o', 11), ('n', 13)]
```

    - pokúste sa to zapísať tak, aby to fungovala aj pre postupnosti rôznych dĺžok: vtedy vytvorí len toľko dvojíc, koľko je prvkov v kratšej z týchto postupností, napr.

```
>>> zip('python', [2, 3, 5, 7, 11])
[('p', 2), ('y', 3), ('t', 5), ('h', 7), ('o', 11)]
```

- pokúste sa to zapísať tak, aby funkcia fungovala pre ľubovoľný počet ľubovoľne dlhých postupností, napr.

```
>>> zip('python', [2, 3, 5, 7, 11], 'abcd')
[('p', 2, 'a'), ('y', 3, 'b'), ('t', 5, 'c'), ('h', 7, 'd')]
```

- v Pythone existuje **štandardná funkcia** `zip()`, ktorá funguje skoro presne ako táto posledná verzia funkcie, len jej výsledkom nie je zoznam, ale opäť postupnosť (dá sa prechádzať for-cyklom, alebo vypísať pomocou `print(*zip(...))`, alebo previesť na zoznam pomocou `list(zip(...))`)
- funkciu `zip()` (štandardnú alebo tú vašu) môžeme použiť aj vo for-cykle, napr.

```
>>> ''.join(x*y for x, y in zip('python', [2, 3, 2, 1, 3, 2]))
'ppyyytthooonn'
```

KAPITOLA 21

21. Práca s obrázkami

21.1 Python Imaging Library

Aby ste mohli pracovať s knižnicou **PIL** (Python Imaging Library), musíte mať nainštalovaný modul **Pillow**. Základné info o inštalácii nájdete na stránke Inštalácia Pillow. Pod operačným systémom Windows (podobne aj v iných OS) treba v príkazovom okne `cmd` spustiť príkaz:

```
> pip install Pillow
```

Je možné, že bude od vás požadovať administrátorské práva, resp. spustiť `cmd` ako správca.

Po úspešnej inštalácii musíme v samotnom Pythone na začiatok našich programov zadať:

```
from PIL import Image
```

Teraz je Python pripravený pracovať s obrázkovými objektmi. Základné informácie o tomto module môžete nájsť na webe Pillow.

21.1.1 Vytvorenie obrázkového objektu

Knižnica `Image` umožňuje pracovať s rastrovými obrázkami priamo v pamäti. Obrázkové objekty (inštancie triedy `Image.Image`) sa vytvárajú buď prečítaním obrázkového súboru, alebo vytvorením nového obrázka (jednofarebný obdĺžnik) alebo ako výsledok niektorých operácií s nejakými už existujúcimi obrázkami.

Obrázkový objekt vytvoríme prečítaním obrázkového súboru z disku pomocou príkazu:

```
>>> obr = Image.open('meno súboru')
```

Obrázkový súbor môže mať skoro ľubovoľný grafický typ, napr. **png**, **bmp**, **jpg**, **gif**, ... Aby tento súbor príkaz `Image.open()` mal šancu nájsť, buď sa bude nachádza v tom istom priečinku na disku ako náš program, alebo mu zadáme relatívnu alebo absolútnu cestu, napr.

```
>>> obr1 = Image.open('tiger.bmp')
>>> obr2 = Image.open('obrazky/slon.jpg')
>>> obr3 = Image.open('d:/user/data/python.png')
```

Po prečítaní súboru z danej inštancie môžeme zistiť niektoré jeho parametre, napr.

```
>>> obr1
<PIL.BmpImagePlugin.BmpImageFile image mode=RGB size=384x256 at 0x2C4F710>
>>> obr1.size
(384, 256)
>>> obr1.width
384
>>> obr1.height
256
>>> obr1.mode
'RGB'
```

Obrázky môžu byť uložené v niekoľkých rôznych **módoch**: pre nás budú zaujímavé len dva z nich `'RGB'` a `'RGBA'` pre „obyčajné“ **rgb** a **rgb**, ktoré si pamätá aj priesvitnosť, tzv. **alfa-kanál**, teda **rgba**.

Veľmi často používaným príkazom tri testovaní a ladení bude:

```
>>> obr1.show()
```

ktorý zobrazí momentálny obsah nami pripravovaného obrázku v nejakom externom programe - toto závisí od nastavení vo vašom operačnom systéme.

Obrázkový objekt môžeme vytvoriť aj pomocou funkcie `new()`, ktorej zadáme „mód“ (t.j. `'RGB'` alebo `'RGBA'`), veľkosť obrázka v pixeloch ako dvojicu `(šírka, výška)` a prípadne farbu, ktorou sa tento obrázok zafarbí (inak bude čierny):

```
>>> obr4 = Image.new('RGB', (300, 200), 'pink')
```

Vytvorí nový objekt `obr4` - obrázok veľkosti 300x200 pixelov (šírka 300, výška 200), ktorý bude celý ružový. Ako farbu pixelov tu môžeme okrem mena farby uviesť aj reťazec známy z `tkinter`, ktorý obsahuje cifry v 16-ovej sústave, napr. `'#12a4ff'`, alebo trojicu **rgb** (teda `tuple`) v tvare, napr. `(120, 198, 255)`. Neskôr tu uvidíme aj štvoricu pre **rgba**.

Ak by sme chceli vytvoriť nový obrázok rovnakej veľkosti akú má niektorý už existujúci, môžeme využiť jeho atribút `size`, napr.

```
>>> obr4 = Image.new('RGB', obr1.size, '#ffffff')
```

vytvorí obrázkový objekt eľkosti 384x256, pričom všetky jeho pixely budú biele.

21.1.2 Uloženie obrázka do súboru

Metóda `save('meno súboru')` uloží obrázok do súboru s ľubovoľnou príponou. Takto môžeme veľmi ľahko zmeniť grafický formát obrázku. Napr.

```
>>> obr1.save('tiger.png')
>>> obr4.save('temp/prazdny.jpg')
```

Obrázok `obr1`, ktorý mal pôvodne bitmapový formát (prípona `.bmp`) sa zapísal do `.png` súboru. Obrázok `obr4`, ktorý sme vytvorili pomocou metódy sme zapísali do `.jpg` súboru.

Uvedomte si, že ak nejaký obrázok chceme len prečítať a hneď ho zapísať do súboru (možno s inou príponou), môžeme to priamo zapísať:

```
>>> Image.open('obrazok.jpg').save('obrazok.png')
```

Vytvorila sa kópia pôvodného obrázka s novou prípomnou.

21.1.3 Oblasť v obrázku

Z obrázka môžeme odkopírovať ľubovoľnú obdĺžnikovú oblasť a uložiť ju do iného obrázka. Oblasť definujeme ako štvoricu (`tuple`) štyroch celých čísel `(x, y, x+šírka, y+výška)`, kde `(x, y)` je poloha ľavého horného pixla oblasti (riadky aj stĺpce indexujeme od 0) a `šírka` a `výška` sú rozmery oblasti v pixeloch, t.j. počet stĺpcov a riadkov pixelov.

Na vykopírovanie oblasti slúži príkaz:

```
>>> novy_obr = povodny_obr.crop(oblast')
```

Touto metódou vzniká nový obrázok zadaných rozmerov (podľa oblasti) s vykopírovaným obsahom. Pôvodný obrázok ostáva bez zmeny. Ak je časť oblasti mimo pôvodného obrázka, v novom obrázku tu budú doplnené čierne (resp. pre **rgba** priesvitné) pixely.

Napr.

```
>>> obr1a = obr1.crop((60, 50, 220, 200))
>>> obr1a.size
(160, 150)
```

Ak vieme pozíciu `(x, y)` ľavého horného pixelu oblasti a jej šírku `sir` a výšku `vys`, môžeme kopírovanie zapísať aj takto:

```
>>> x, y, sir, vys = 60, 50, 160, 150
>>> obr1a = obr1.crop((x, y, x+sir, y+vys))
```

niekedy môžete vidieť aj takýto zápis:

```
>>> oblast = 60, 50, 220, 200
>>> obr1a = obr1.crop(oblast)
```

Pomocou metódy `crop()` sme vytvorili nový obrázkový objekt `obr1a`. Na nasledovnom obrázku vidíme pôvodný obrázok aj nový s kópiou jeho časti (oblasti):

Niečo ako opačným príkazom pre vykopírovanie oblasti je vloženie obrázka na nejaké miesto iného obrázka. Zabezpečí to metóda `paste()`:

```
>>> obr1.paste(obr2, kam)
```

Tento príkaz **modifikuje** pôvodný obsah obrázka `obr1`. Obrázok `obr2` sa „opečiatkuje“ do `obr1`, pričom parameter `kam` určí pozíciu. Najvhodnejšie je sem písať dvojicu `(x, y)` t.j. pozícia v `obr1`, kam sa umiestni `obr2`, t.j. jeho ľavý horný pixel (inak 4-prvkový parameter `kam` musí určovať rozmer, ktorý je identický s rozmerom `obr2`). Uvedomte si, že táto metóda nevracia žiadnu hodnotu (teda vráti `None`) a preto nemá zmysel jej výsledok niekam priraďovať.

Pozor na „lazy“ vyhodnocovanie

Príkaz `crop()`, ktorý vykopíruje časť obrázka a vytvorí z neho nový obrázok má tzv. **lazy vyhodnocovanie**. To znamená, že hoci samotná operácia ešte neskončila, Python začne vyhodnocovať ďalší príkaz programu. Väčšinou to nevadí, ale niekedy, ak v ďalšom príkaze potrebujeme pracovať s obrázkovou premennou s vykopírovaným obsahom (napr. v `paste()`), samotný `crop()` ešte nemusel dobehnúť. Z tohto dôvodu sa odporúča ešte pred `paste()` presvedčiť ešte nedobehnutý `crop()`, aby už skončil. Na to slúži príkaz `obr.load()`, napr.

```
maly = velky.crop(oblast)
maly.load()                 # tu sa počká na dokončenie crop()
velky.paste(maly, (x, y))
```

Príkaz `paste()` má aj iné využitie:

```
>>> obr1.paste(pixel, oblast')
```

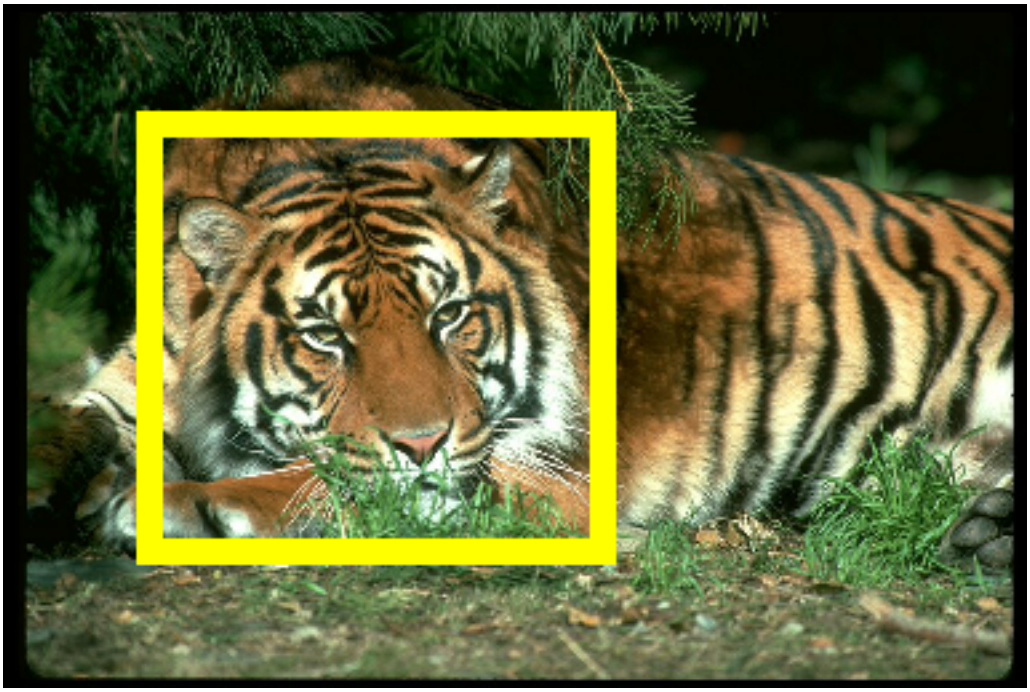
V tomto prípade je `pixel` nejakou farbou a táto sa vyleje do špecifikovanej oblasti (nakreslí zafarbený obdĺžnik). Napr.

```
img = Image.new('RGB', (300, 200), 'gray')
img.paste('red', (20, 20, 80, 180))
img.paste((0, 0, 255), (100, 20, 160, 180))
img.paste('#bf00bf', (180, 20, 240, 180))
```

Keď budete s príkazom `paste()` experimentovať, môžete si pomocou príkazu `copy()` vytvárať kópiu pôvodného obrázka, keďže `paste()` ho modifikuje. Napr.

```
>>> zaloha = obr1.copy()
>>> obr1.paste('yellow', (50, 40, 230, 210))
>>> obr1.paste(obr1a, (60, 50))
>>> obr1.show()      # alebo obr1.save(...)
>>> obr1 = zaloha
```

Zobrazí sa tento obrázok:



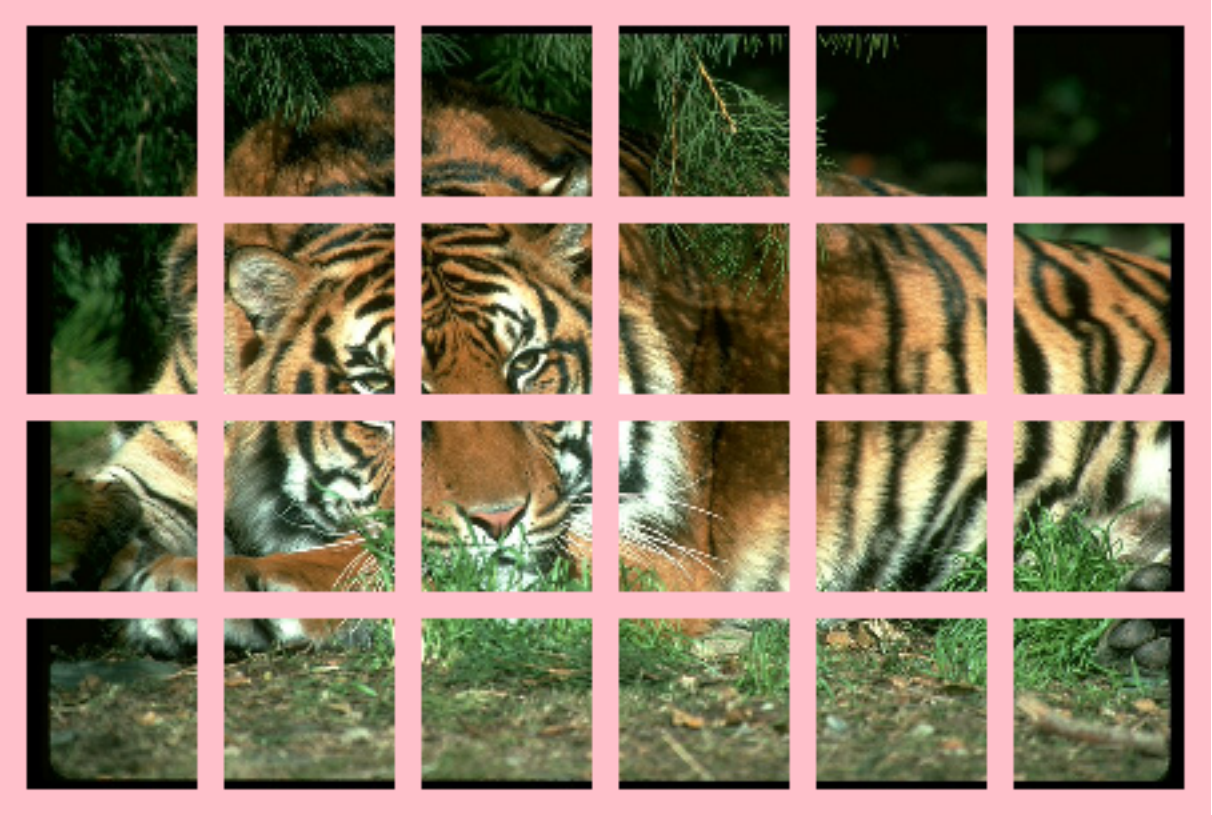
A premenná `obr1` bude stále obsahovať nepokazený obrázok tigra.

Premysleným strihaním nejakého obrázka a potom skladaním týchto rozstrihaných častí:

```
from PIL import Image

obr1 = Image.open('tiger.bmp')
d = 10
novy = Image.new('RGB', (384+7*d, 256+5*d), 'pink')
for i in range(6):
    for j in range(4):
        x, y = i*64, j*64
        novy.paste(obr1.crop((x, y, x+64, y+64)), (x+i*d+d, y+j*d+d))
novy.show()
```

môžeme dostať:



21.1.4 Zmeny obrázka

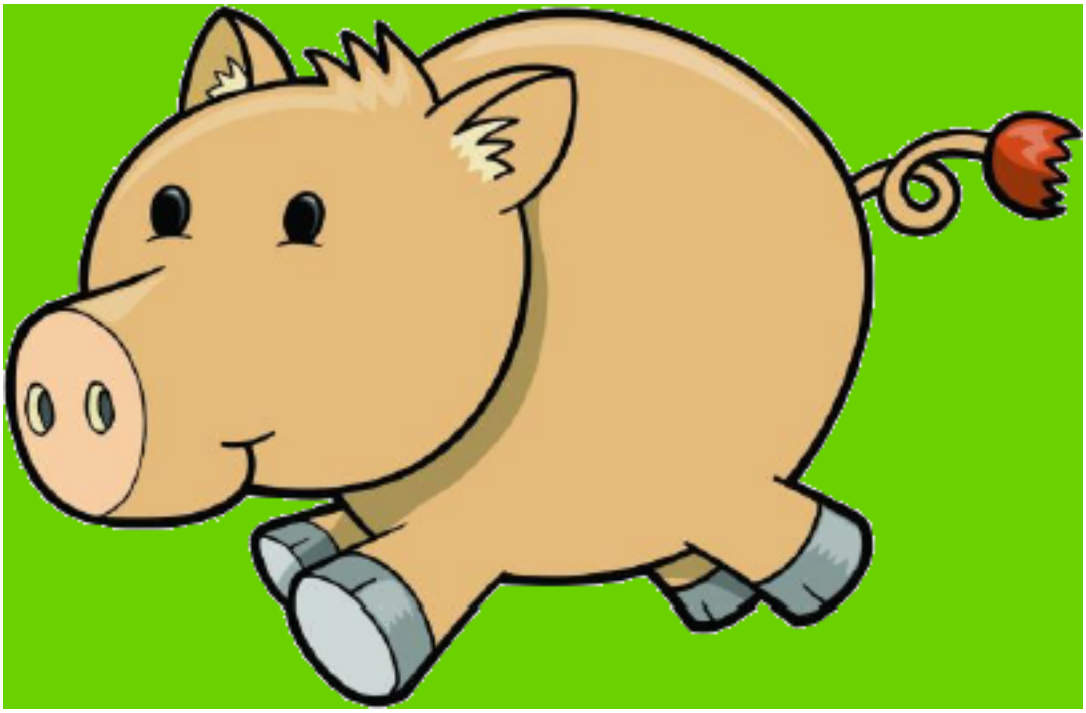
Obrázku vieme zmeniť veľkosť, napr.

```
>>> obr2 = obr1.resize((nova_sirka, nova_vyska))
```

Vytvorí sa nový obrázok so zadanými rozmermi, pôvodný ostáva bez zmeny. Napr.

```
>>> obr3 = obr1.resize((obr1.width*3, obr1.height*3))
>>> obr1 = obr1.resize((obr1.width//2, obr1.height//2))
```

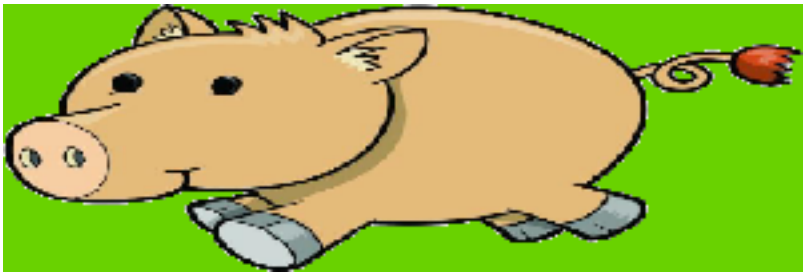
Obrázok `obr3` má trojnásobné rozmery pôvodného obrázka, pričom `obr1` sa zmenší na polovičné rozmery. Napríklad, ak máme takýto obrázok `'prasiatko.png'`:



keď mu zmeníme veľkosť bez zachovania pomeru strán:

```
obr1 = Image.open('prasiatko.png')
obr2 = obr1.resize((300, 100))
```

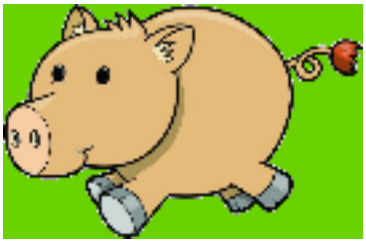
dostávame:



ale zmensenie oboch strán na tretinu:

```
sirka, vyska = obr1.size
obr3 = obr1.resize((sirka//3, vyska//3))
```

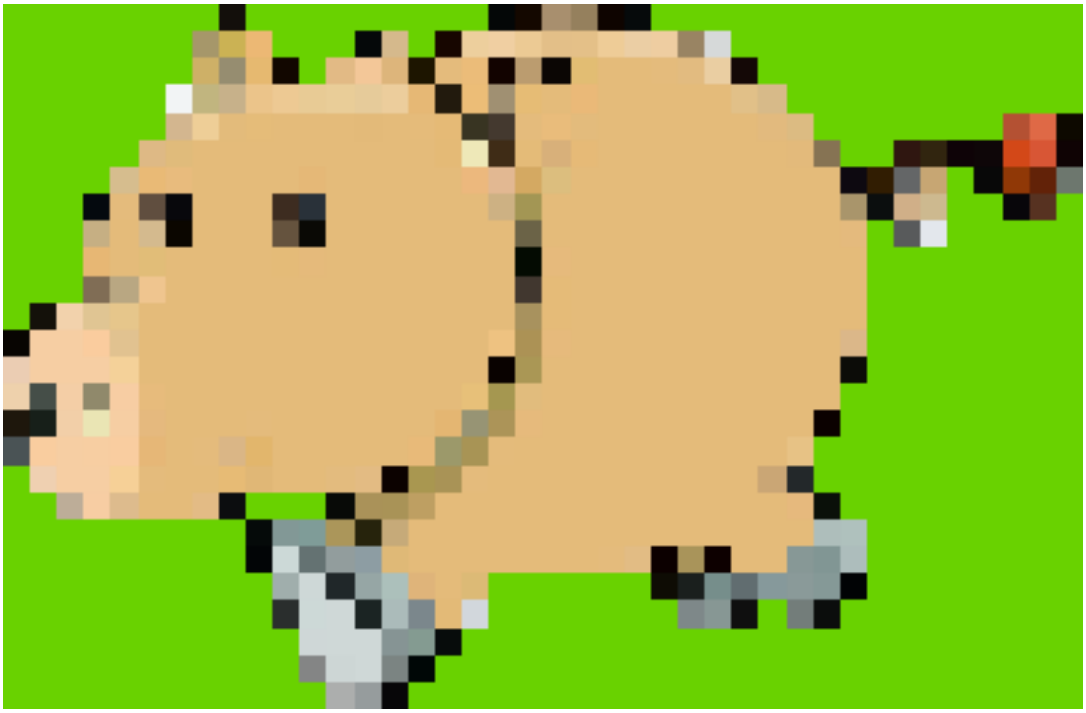
vyzerá takto:



Uvedomte si, že zmenšovaním obrázka sa nejaká informácia stráca, takže, keď mu po zmenšení vrátime pôvodnú veľkosť:

```
obr4 = obr1.resize((sirka//10, vyska//10)).resize(obr1.size)
```

dostávame:



Obrázok môžeme otáčať, resp. preklápať pomocou `transpose()`:

```
>>> novy_obr = obr1.transpose(Image.FLIP_LEFT_RIGHT)      # preklopí
>>> novy_obr = obr1.transpose(Image.FLIP_TOP_BOTTOM)      # preklopí
>>> novy_obr = obr1.transpose(Image.ROTATE_90)            # otočí
>>> novy_obr = obr1.transpose(Image.ROTATE_180)           # otočí
>>> novy_obr = obr1.transpose(Image.ROTATE_270)           # otočí
```

Zrejme pri tomto sa niekedy zmenia rozmery výsledného obrázka.

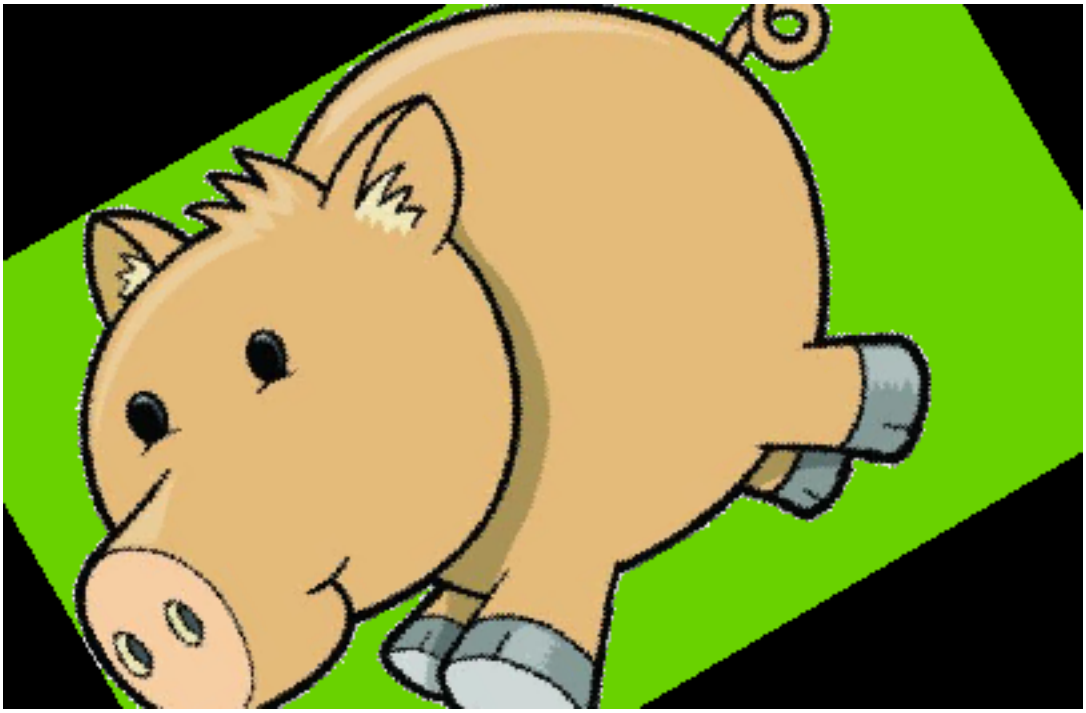
Otáčať môžeme o ľubovoľný uhol:

```
>>> novy_obr = obr1.rotate(uhol)
```

Uhol zadávame v stupňoch v protismere otáčania hodinových ručičiek. Ak otočíme obrázok prasiatka `obr1` napr. o 30 stupňov:

```
obr5 = obr1.rotate(30)
```

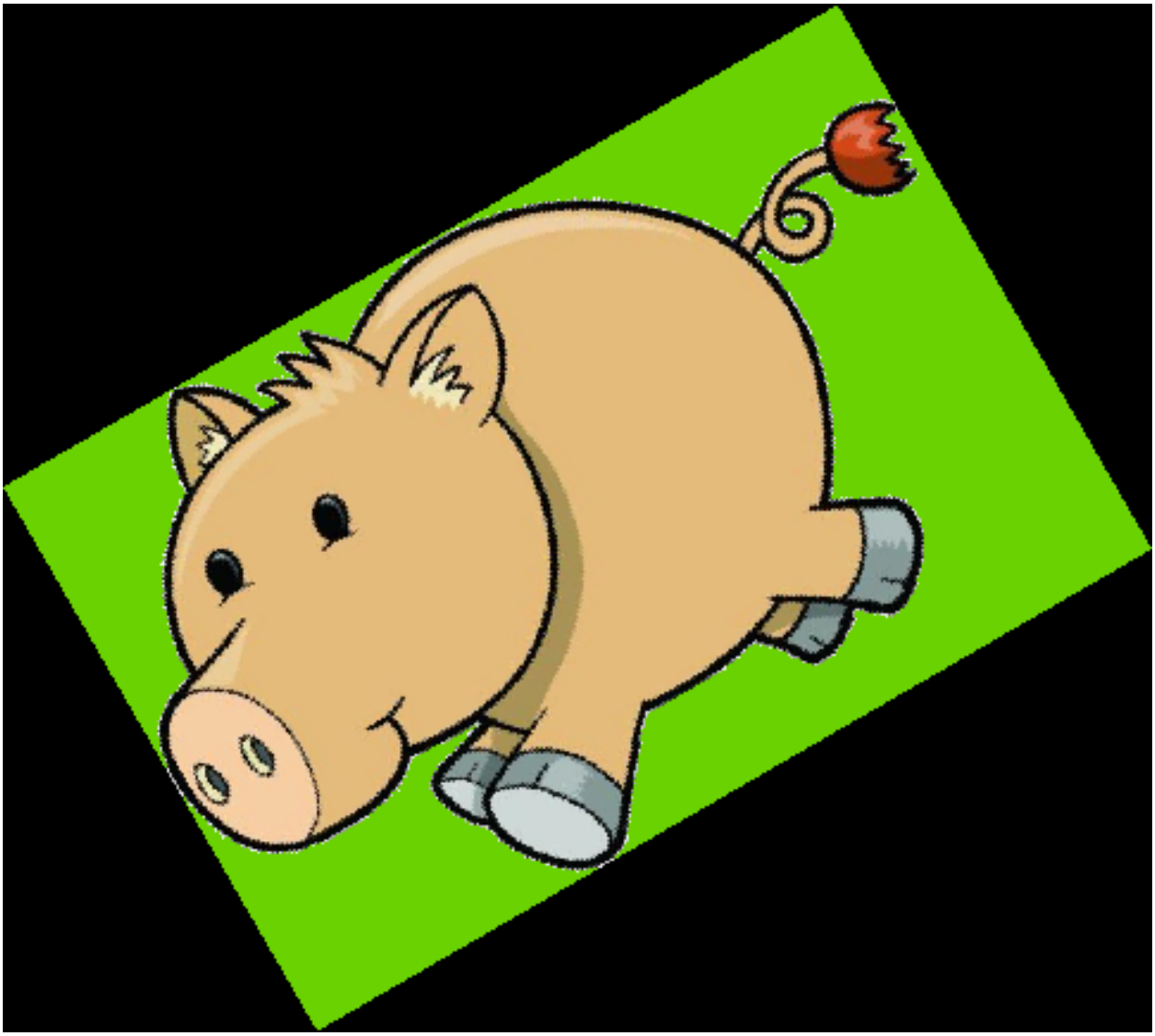
dostávame:



Výsledný obrázok bude mať pôvodné rozmery `obr1`, teda nejaké otočené časti sa pritom stratia a ešte pribudli aj nejaké čierne oblasti. Ak ale zadáme:

```
obr6 = obr1.rotate(30, expand=True)
```

výsledný obrázok sa zväčší tak, aby sa teraz do neho zmestil celý otočený pôvodný obrázok. Teraz ešte lepšie vidieť nové čierne oblasti:



21.1.5 Práca s jednotlivými pixelmi

Metóda `getpixel()` vráti farbu konkrétneho pixelu, napr.

```
>>> obr1.getpixel((108, 154))
(228, 187, 122)
```

Metóda `putpixel()` zmení konkrétny pixel v obrázku. Napr.

```
>>> obr1.putpixel((108, 154), (255, 0, 0))
```

Zafarbí daný pixel na červeno. V tomto prípade musí byť farba zadaná ako trojica (resp. pre **rgba** ako štvorica) čísel od 0 do 255. Ak v obrázku prasiatka zafarbíme v nejakej oblasti 500 náhodných bodiek:

```
from random import randrange as rr

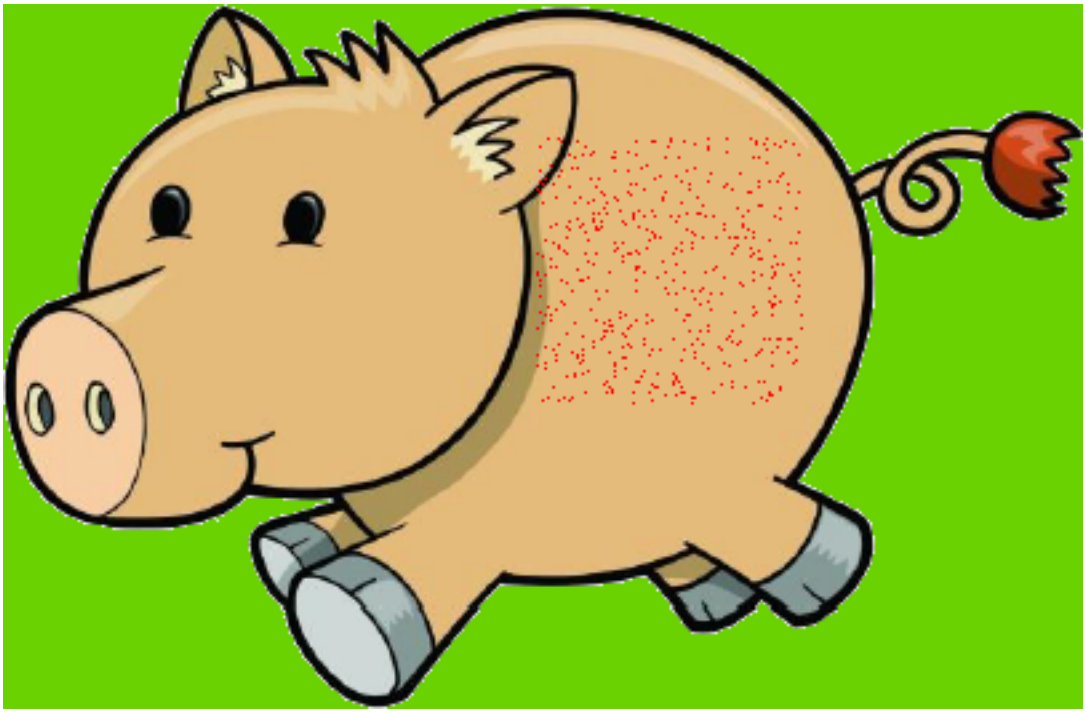
for i in range(500):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    xy = rr(200, 300), rr(50, 150)
    farba = (255, 0, 0)
    obr1.putpixel(xy, farba)
```

vyzerá to nejako takto:



Takýto zápis zistí **množinu** všetkých farieb v obrázku:

```
>>> mn = {obr1.getpixel((x, y)) for x in range(obr1.width) for y in range(obr1.
↪height)}
>>> len(mn)
17653
```

21.1.6 Priesvitnosť

Obrázok musí byť v móde `'RGBA'`, t.j. každý pixel obsahuje ešte jednu číselnú informáciu o priesvitnosti (tzv. alfa-kanál). Pre toto číslo **255** označuje **nepriesvitný** pixel, **0** označuje úplne **priesvitný** pixel (vtedy na farbe **rgb** nezáleží), a hodnoty medzi tým označujú mieru priesvitnosti.

Pomocou metódy `convert()` môžeme prekonvertovať mód obrázka, napr.

```
>>> im = Image.open('obrazok.bmp')
>>> im1 = im.convert('RGBA')
>>> im2 = Image.open('obrazok2.png').convert('RGBA')
```

Obrázok `im` má zachovaný mód zo súboru, obrázky `im1` aj `im2` majú zmenený mód na `'RGBA'`. Od teraz musíme pre tieto obrázky pixely zadávať ako štvorice (r, g, b, a), operácie `crop()` aj `rotate()` môžu vytvoriť priesvitné pixely za hranicou pôvodných obrázkov. Operácia `paste()` ale správne nezlučuje priesvitné pixely s pôvodným obrázkom, tak ako by sme očakávali. Na to potrebujeme iný mechanizmus:

- funkcia `alpha_composite()` dokáže na seba položiť dva obrázky, ktoré majú priesvitnosť (polopriesvitnosť)
- jej formát je `Image.alpha_composite(obr1, obr2)`, kde oba obrázky musia mať rovnaké rozmery a mód `'RGBA'`, výsledkom je nový obrázok, v ktorom na `obr1` je položený `obr2`, pričom cez priesvitné časti `obr2` sú vidieť pixely z `obr1`

Tento mechanizmus môžete vidieť použitý v tejto funkcii `poloz()`:

```
def poloz(obr_kam, obr_co, xy):
    w, h = obr_co.size
    x, y = xy
    box = (x, y, x+w, y+h)
    obr_kam.paste(Image.alpha_composite(obr_kam.crop(box), obr_co), box)
```

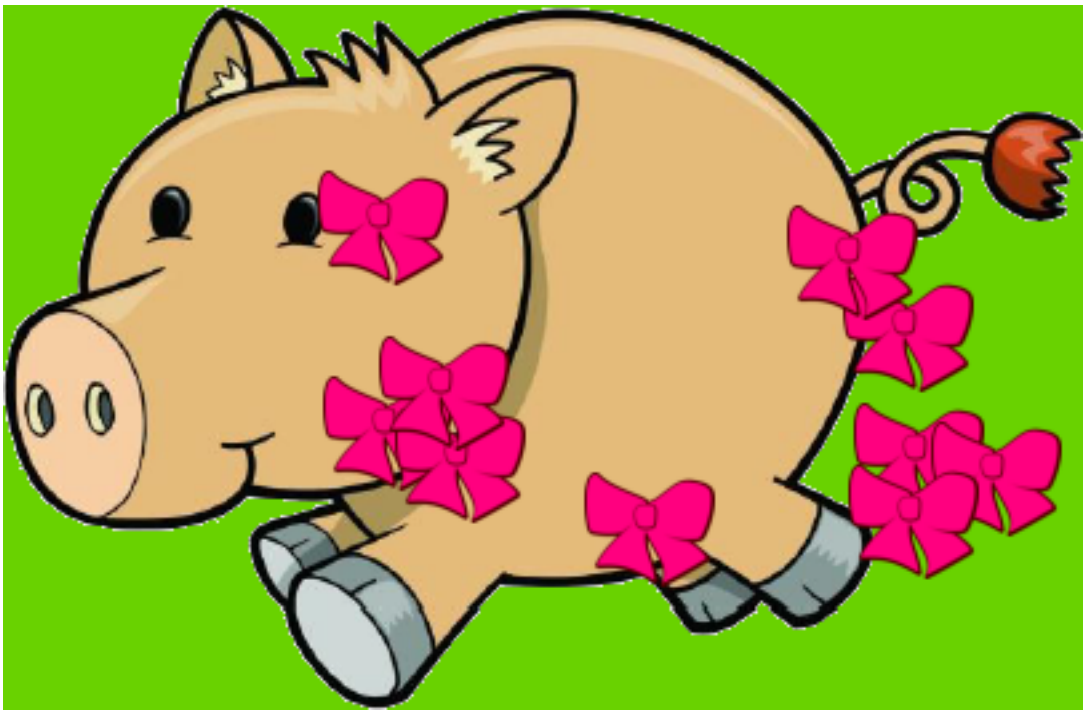
Funkcia do obrázka `obr_kam` položí `obr_co` na pozíciu `xy` (čo je dvojica `(x, y)`), `xy` je pozícia, kam sa dostane ľavý horný pixel `obr2` v `obr1`. Táto pozícia `xy` sa môže nachádzať aj mimo `obr1`.

Napr. tento kód s prasiatkom a obrázkom mašličky:

```
from random import randrange as rr

obr1 = Image.open('prasiatko.png').convert('RGBA')
masla = Image.open('masla.png')              # mala by byt RGBA
for i in range(10):
    xy = rr(50, 350), rr(50, 200)
    poloz(obr1, masla, xy)
```

vytvorí takýto obrázok:



21.1.7 Rozoberanie animovaných gif

Animovaný **gif** súbor sa skladá zo série za sebou idúcich obrázkov. Načítanie takéhoto súboru pomocou `Image.open()` nám automaticky sprístupní prvú fázu (má poradové číslo 0). Ak potrebujeme pracovať s `i`-tou fázou animácie (`i`-tym obrázkom série), použijeme metódu `obr.seek(i)`.

Nasledujúca časť programu otvorí obrázkový súbor s animáciou napr. `'vtak.gif'`, ktorý sa skladá z neznámeho počtu obrázkov. Postupne všetky tieto fázy uloží do samostatných obrázkových súborov vo formáte `'png'`:

```
gif = Image.open('vtak.gif')
i = 0
while True:
    gif.save(f'vtak/vtak{i}.png')
    try:
        i += 1
        gif.seek(i)
    except EOFError:
        break
```

Pre tento súbor vtak.gif s animovaným obrázkom:

v priečinku `vtak` dostávame túto postupnosť súborov:

21.1.8 Vypĺňanie oblasti farbou

Na vypĺňanie farbou nejakej oblasti, ktorá je ohraničená nejakým obrysom, slúži funkcia `floodfill()` z modulu `ImageDraw`. Funkcia má tieto parametre (dva varianty volania):

```
from PIL import ImageDraw

ImageDraw.floodfill(obr, xy, farba)
ImageDraw.floodfill(obr, xy, farba, hraničná_farba)
```

kde

- `obr` je obrázok, v ktorom sa bude vyfarbovať nejaká oblasť
- `xy` je dvojica `(x, y)` pozície v obrázku, kde sa naštartuje „vylievanie“ farby
- `farba` je tá farba, ktorou sa bude vyfarbovať

- `hraničná_farba` - ak nie je zadaná, tak sa farba vylieva do celej súvislej oblasti, ktorá má rovnakú farbu ako bod na pozícii `xy`; ak je tento 4 parameter zadaný, tak hranicu súvislej oblasti určujú pixely tejto farby

Nasledovná ukážka demonštruje použitie tejto funkcie:

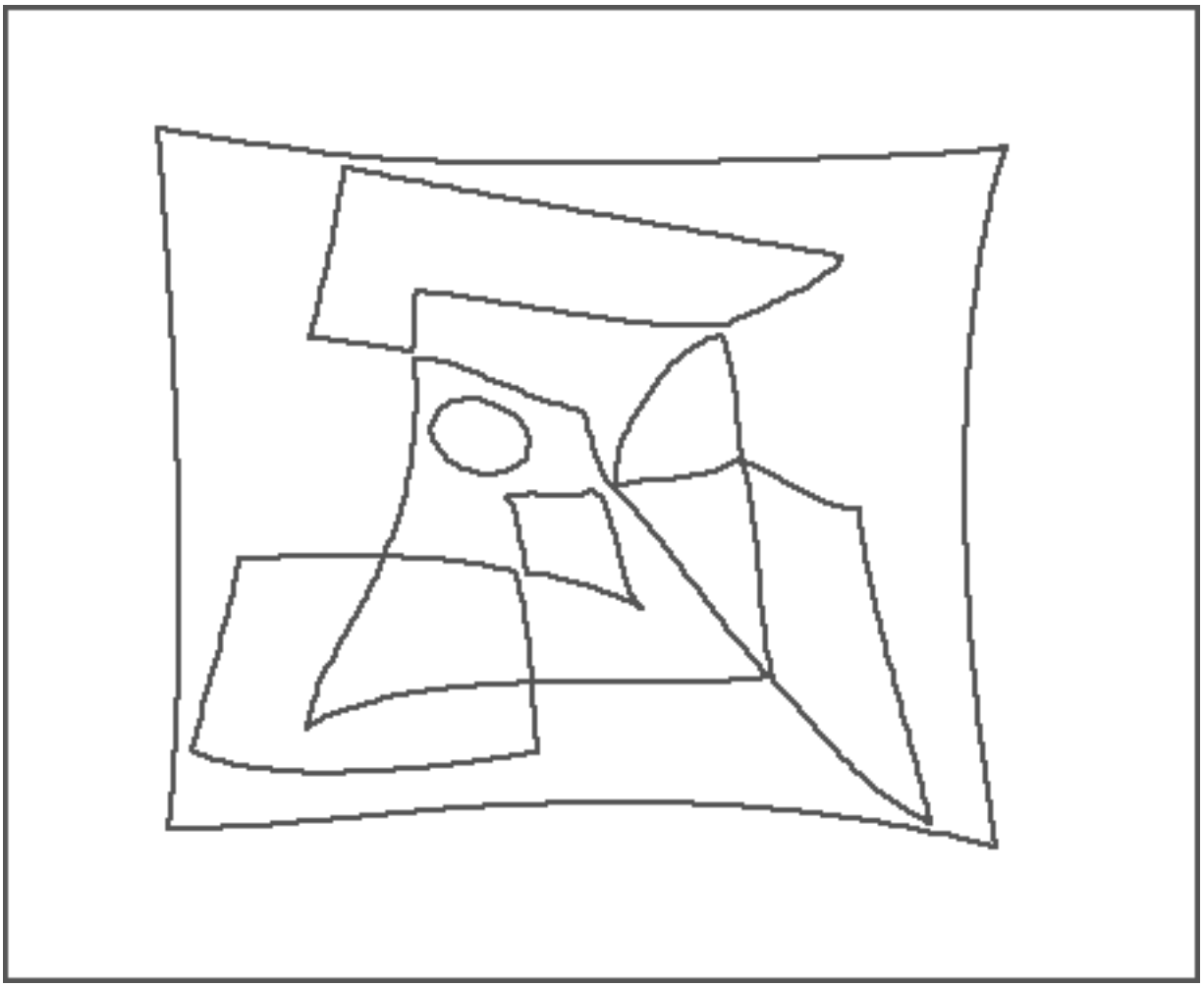
```
from PIL import Image, ImageDraw
from random import randrange as rr

f = Image.open('fill.png')
for i in range(100):
    xy = rr(f.width), rr(f.height)
    if f.getpixel(xy) == (255, 255, 255):
        ImageDraw.floodfill(f, xy, (rr(256), rr(256), rr(256)))

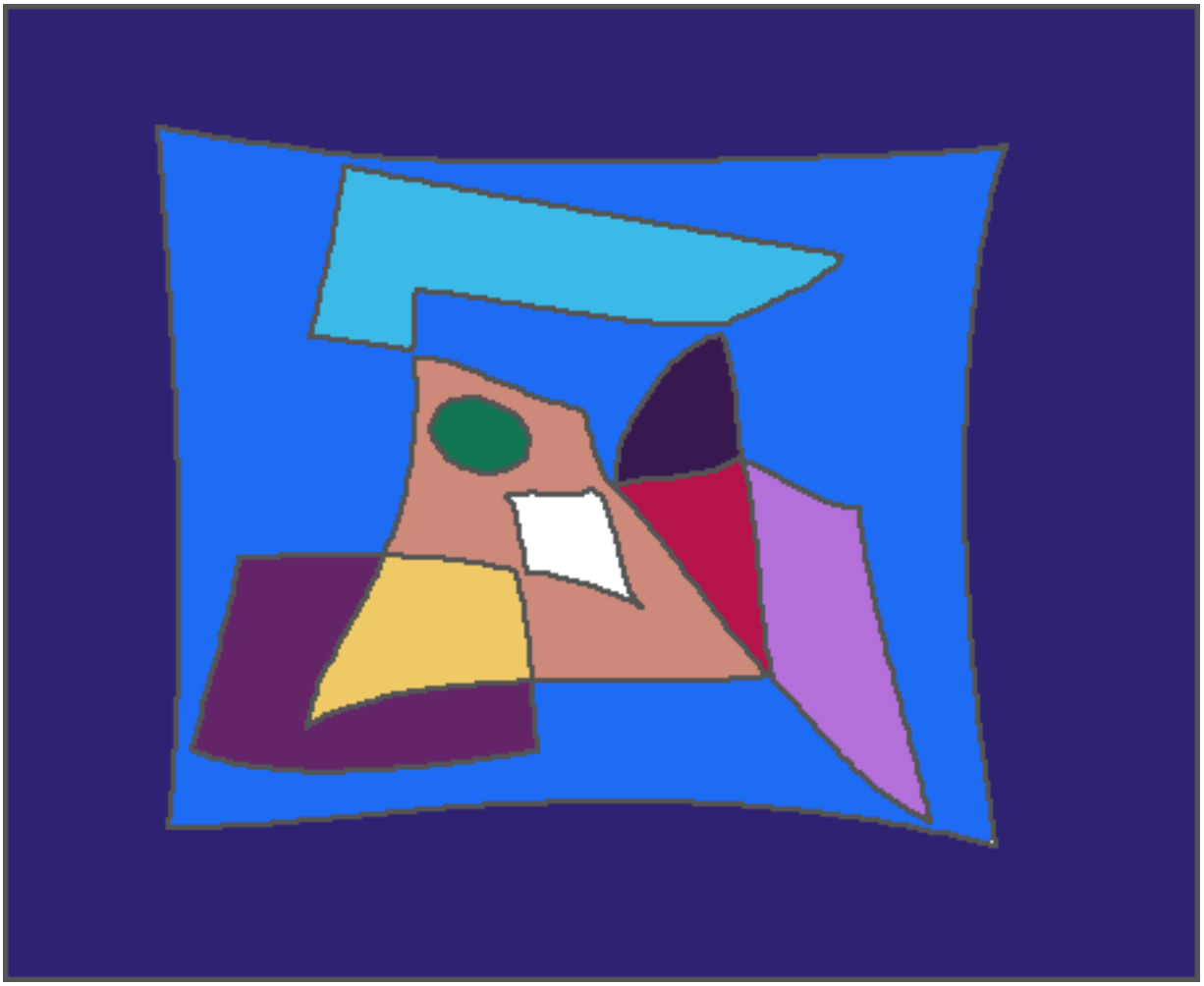
f.show()
```

V pôvodnom obrázku `'fill.png'` sú niektoré plochy biele. Tento program stokrát náhodne zvolí nejakú pozíciu a ak je tento pixel biely, tak vyplní celú súvislú oblasť s náhodnou farbou.

Ak bol súbor `'fill.png'` takýto:



môžeme dostať napríklad tento obrázok:



21.2 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Napíšte funkciu `novy(sir, vys, meno_suboru=None)`, ktorá vytvorí biely obrázok veľkosti `sir``x``vys` a ak je zadané aj `meno_suboru`, uloží ho do tohto súboru, inak vráti obrázok ako výsledok funkcie. Pomocou tejto funkcie potom vytvorte súbor `'biely.bmp'` s bitmapou veľkosti 100x100. Skontrolujte to na disku.

   - napr.

   ```
   >>> novy(600, 100).show()
   ```

2. Napíšte funkciu `konvertuj(meno_suboru1, meno_suboru2)`, ktorá prekonvertuje súbor `meno_suboru1` na súbor `meno_suboru2`. Nájdite na internete obrázok vo formáte `'.bmp'`, uložte ho na disk a potom pomocou funkcie `konvertuj()` ho prekonvertujte na `'.png'` formát.

   - napr.

```
>>> konvertuj('obrazok.bmp', 'obrazok.png')
```

3. Vytvorte biely obrázok veľkosti 380x380, do ktorého vložíte 16 farebných štvorcov veľkosti 80x80 (medzi štvorcami je medzera 20 pixelov). Farby štvorcov si zvoľte ľubovoľne (napr. všetky sú rovnaké, alebo sa nejaké striedajú, alebo sú náhodné). Obrázok uložte do súboru.
4. Napíšte funkciu `zmensi(obrazok)`, ktorá zmenší daný obrázok tak, že jeho šírka bude 128 a výška sa prepočíta tak, aby bol zachovaný pomer strán. Funkcia ako výsledok vráti tento zmenšený obrázok. Prečítajte nejaký obrázok zo súboru a pomocou `zmensi()` ho zmenšite a uložte do súboru s pozmeneným menom. Skontrolujte výsledný súbor.
   - napr.

```
>>> zmensi(Image.open('subor1.png')).save('subor1x.png')
```

5. Napíšte funkciu `vymen(obrázok)`, ktorá navzájom v obrázku vymení ľavú a pravú polovicu obrázku. Funkcia nemodifikuje pôvodný obrázok, ale vráti nový s vymenenými polovicami. Prečítajte nejaký obrázok zo súboru a pomocou `vymen()` vytvorte nový a ten uložte do súboru. Výsledok skontrolujte.
   - napr.

```
>>> vymen(Image.open('subor2.png')).save('subor2x.png')
```

6. Napíšte funkciu `kopia(obrazok)`, ktorá vyrobí kópiu pôvodného obrázka, ale ho kopíruje po jednom pixeli. Zrejme si najprv vytvoríte prázdny obrázok rovnakých rozmerov a sem budete kopírovať pixely (pomocou `getpixel()` a `putpixel()`). Funkcia vráti tento nový obrázok ako svoj výsledok. Teraz prečítajte nejaký malý obrázok zo súboru a vyrobte z neho pomocou `kopia()` kópiu. Výsledok uložte do súboru a skontrolujte.
   - napr.

```
>>> kopia(obr1).show()
```

7. Napíšte funkciu `prevrat(obrazok)`, ktorá vyrobí **prevrátenú** kópiu pôvodného obrázka (obrázok je hore nohami), ale ho kopíruje po jednom pixeli. Funkcia vráti tento nový obrázok ako svoj výsledok. Teraz prečítajte nejaký malý obrázok zo súboru a vyrobte z neho nový pomocou `prevrat()`. Výsledok uložte do súboru a skontrolujte.
   - napr.

```
>>> prevrat(obr1).show()
```

8. Napíšte funkciu `sedy(obrazok)`, ktorá vyrobí **čierno-bielu** kópiu pôvodného obrázka (vypočítate priemer zložiek `(r, g, b)` (nech je to `p`) a z toho vznikne nová farba `(p, p, p)`). Funkcia vráti tento nový obrázok ako svoj výsledok. Teraz prečítajte nejaký malý obrázok zo súboru a vyrobte z neho čiernobiely pomocou `sedy()`. Výsledok uložte do súboru a skontrolujte.
   - napr.

```
>>> sedy(obr1).show()
```

9. Napíšte funkciu `strihaj1(obr, n)`, ktorá rozstrihá zadaný obrázok na `n` rovnako-širokých častí (po stĺpcoch) a všetky takto rozstrihané časti vráti ako zoznam obrázkov.
10. Napíšte funkciu `strihaj2(obr, n)`, ktorá rozstrihá zadaný obrázok na `n` rovnako-vysokých častí (po riadkoch) a všetky takto rozstrihané časti vráti ako zoznam obrázkov.
11. Napíšte funkciu `zlep1(zoznam)`, ktorá zlepí vedľa seba obrázky zadané v zozname (napr. sú výsledkom `strihaj1()`). Výsledný obrázok vráti ako výsledok funkcie.

- otestujte:

```
>>> zlep1(strihaj1(obr1, 5)[::-1]).show()
```

12. Napíšte funkciu `zlep2(zoznam)`, ktorá zlepí pod seba obrázky zadané v zozname (napr. sú výsledkom `strihaj2()`). Výsledný obrázok vráti ako výsledok funkcie.
13. Napíšte funkciu `zapis(zoznam, meno, pripona)`, ktorá v parametri `zoznam` dostáva postupnosť obrázkov a všetky tieto obrázky uloží do súborov s menami ‚meno0.pripona', ‚meno1.pripona', ‚meno2.pripona', ...
14. Napíšte funkciu `citaj(*mena_suborov)`, ktorá ako parameter dostáva postupnosť mien grafických súborov, tieto súbory prečíta, uloží do zoznamu a tento nový zoznam vráti ako výsledok funkcie.
    - napr.

```
>>> zoznam = citaj('tiger.bmp', 'image0.png', 'image1.png')
```

    - vytvorí trojprvkový zoznam prečítaných obrázkov
    - potom pre funkciu `zapis()` z úlohy (13)

```
>>> zapis(citaj('tiger.bmp', 'image0.png', 'image1.png'), 'temp/obrazok', 'jpg
↪')
```

    - vytvorí kópie 3 zadaných súborov s novými menami (v priečinku `'temp'`) vo formáte `'jpg'`

KAPITOLA 22

22. Animované obrázky

Pripomeňme si, ako sme kreslili obrázky v `tkinter`:

```
import tkinter

canvas = tkinter.Canvas(bg='navy')
canvas.pack()
tkim = tkinter.PhotoImage(file='python-logo.png')
canvas.create_image(200, 150, image=tkim)
```

Zvolili sme si obrázok python-logo.png, ktorý má niektoré časti priesvitné.

Je veľmi dôležité si uvedomiť, že `tkinter` si príkazom `create_image()` zapamätá referenciu na tento obrázok, ale Pythonu o tom „nedá vedieť“. Lenže Python je priveľmi usilovný v upratovaní nepoužívanej pamäti a ak premennej `tkim` zmeníme obsah, alebo ju zrušíme, Python z pamäte obrázok vyhodí, lebo za každú cenu chce upratať nepotrebné informácie. Môžete vyskúšať po spustení predchádzajúceho kódu zmeniť obsah premennej:

```
>>> tkim = 0
```

Väčšinou obrázok zmizne okamžite, niekedy treba ešte niečo nakresliť a až potom sa obrázok stratí, napr.

```
>>> tkim = 0
>>> canvas.create_line(0, 0, 300, 300)
```

Podobný efekt dosiahneme aj vtedy, keď tento program zapíšeme bez pomocnej premennej `tkim`:

```
import tkinter

canvas = tkinter.Canvas(bg='navy')
canvas.pack()
canvas.create_image(200, 150, image=tkinter.PhotoImage(file='python-logo.png'))
```

Python aj v tomto prípade obrázok najprv prečíta vo formáte `tkinter.PhotoImage()`, pošle ho ako skutočný parameter do `create_image()`, lenže, keďže naňho nikto neodkazuje (žiadna premenná neobsahuje referenciu), obrázok okamžite uvoľní. Z tohto dôvodu tento program nezobrazí žiaden obrázok.

22.1 ImageTk

Obrázky vieme načítať a vykresliť aj v `PIL.Image`, ale tieto dva formáty sú navzájom nekompatibilné. Napr.

```
from PIL import Image

im = Image.open('python-logo.png')
im.show()
```

Ak budeme chcieť obrázky vytvorené alebo prečítané v `PIL` potom v grafickej ploche nielen vykresľovať, ale potom ich aj pomocou `tkinter` meniť a posúvať, budeme musieť použiť nejakú konverziu z `PIL` do `tkinter`. V tomto prípade využijeme z knižnice `PIL` ďalší podmodul `ImageTk`. Môžeme to zapísať takto:

```
from PIL import Image, ImageTk
import tkinter

canvas = tkinter.Canvas(bg='navy')
canvas.pack()
#########################################################
tkim = ImageTk.PhotoImage(Image.open('python-logo.png'))
#########################################################
canvas.create_image(200, 150, image=tkim)
```

Funkcia `PhotoImage()` z knižnice `ImageTk` má jeden parameter typu obrázok z `Image` a prerobí ho na obrázkový objekt pre `tkinter`. Tento objekt môžeme ešte pred prekonvertovaní pre `tkinter` upraviť najrozličnejšími obrázkovými metódami, s ktorými sme sa naučili pracovať na minulej prednáške. Môžeme napr. zapísať:

```
from PIL import Image, ImageTk
import tkinter

canvas = tkinter.Canvas(bg='navy')
canvas.pack()
#########################################################
img = Image.open('python-logo.png')
img1 = img.rotate(45, expand=True)
tkim = ImageTk.PhotoImage(img1)
#########################################################
canvas.create_image(200, 150, image=tkim)
```

Často uvidíte aj veľmi kompaktný zápis, napr. takto:

```
#########################################################
tkim = ImageTk.PhotoImage(Image.open('python-logo.png').rotate(45, expand=True))
#########################################################
```

22.1.1 Otáčanie obrázka

Využime tento kompaktný zápis na pomalé otáčanie celého obrázka:

```
from PIL import Image, ImageTk
import tkinter

canvas = tkinter.Canvas(bg='navy')
canvas.pack()
a = canvas.create_image(200, 150)     # zatial prazdny obrazok
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
uhol = 0
while True:
    tkim = ImageTk.PhotoImage(Image.open('python-logo.png').rotate(uhol, expand=True))
    canvas.itemconfig(a, image=tkim)
    uhol += 10
    canvas.update()
    canvas.after(100)
```

Všimnite si, že v tomto nekonečnom cykle stále čítame a otáčame ten istý súbor, pričom súbor by sme mohli prečítať len raz a potom ho už len otáčame:

```
from PIL import Image, ImageTk
import tkinter

canvas = tkinter.Canvas(bg='navy')
canvas.pack()
a = canvas.create_image(200, 150)
img = Image.open('python-logo.png')
uhol = 0
while True:
    tkim = ImageTk.PhotoImage(img.rotate(uhol, expand=True))
    canvas.itemconfig(a, image=tkim)
    uhol += 10
    canvas.update()
    canvas.after(100)
```

V tomto nekonečnom cykle sa po 36 prechodoch znovu opakujú tie isté obrázky. Môžeme to prepísať tak, že tieto obrázky vypočítame len raz ešte pred samotným cyklom a uložíme ich do poľa. V cykle sa už bude len odvolávať na prvky tohto poľa:

```
from PIL import Image, ImageTk
import tkinter

canvas = tkinter.Canvas(bg='navy')
canvas.pack()
a = canvas.create_image(200, 150)
img = Image.open('python-logo.png')
#########################################################
pole = [ImageTk.PhotoImage(img.rotate(uhol, expand=True)) for uhol in range(0, 360,␣
↪10)]
#########################################################
i = 0
while True:
    canvas.itemconfig(a, image=pole[i])
    i = (i + 1) % len(pole)
    canvas.update()
    canvas.after(100)
```

Tento malý testovací program by mohol fungovať aj pre inú postupnosť obrázkov. Do podadresára `a1` uložíme týchto 8 obrázkových súborov

Tieto súbory prečítame do poľa a otestujeme:

```
from PIL import Image, ImageTk
import tkinter

canvas = tkinter.Canvas(bg='navy')
canvas.pack()
a = canvas.create_image(200, 150)
#########################################################
pole = [tkinter.PhotoImage(file=f'a1/vtak{i}.png') for i in range(8)]
#########################################################
i = 0
while True:
    canvas.itemconfig(a, image=pole[i])
    i = (i + 1) % len(pole)
    canvas.update()
    canvas.after(100)
```

22.2 Grafická aplikácia

Na základe týchto skúseností postupne vytvoríme aplikáciu, v ktorej sa bude naraz animovať viac objektov. Začneme obrázkom, ktorý bude pozadím canvasu. My sme si zvolili obrázok jazero.png. Našim cieľom bude vytvoriť `canvas`, ktorý bude presne rovnakých rozmerov ako obrázkový súbor. Zapíšme:

```
import tkinter

bg = tkinter.PhotoImage(file='jazero.png')
canvas = tkinter.Canvas(width=bg.width(), height=bg.height())
canvas.pack()
canvas.create_image(0, 0, image=bg)
```

Žiaľ tento program nefunguje a padne na takejto chybe:

```
RuntimeError: Too early to create image
```

Táto chyba označuje, že sa snažíme vytvoriť objekt `PhotoImage` ešte skôr, ako vzniklo grafické okno, v ktorom bude canvas. Teda nemali by sme volať `PhotoImage()` skôr ako vytvoríme `Canvas()` - toto nám trochu skomplikuje vytvorenie canvasu správneho rozmeru. Ale dá sa to aj inak: keď vytvárame canvas, `tkinter` automaticky najprv vytvorí **grafické okno**. A až potom v tomto okne vytvorí canvas. Pridáme na úplný začiatok príkaz na vytvorenie okna:

```
import tkinter

#########################
win = tkinter.Tk()
#########################
bg = tkinter.PhotoImage(file='jazero.png')
canvas = tkinter.Canvas(width=bg.width(), height=bg.height())
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
canvas.pack()
canvas.create_image(0, 0, image=bg)
```

Teraz už vytvorenie grafického okna správnych rozmerov funguje, len samotný obrázok nepokrýva celý canvas, ale len jeho jednu štvrtinu. Samozrejme, že je to tak: v príkaze `create_image()` súradnice umiestnenia obrázka určujú, kde sa má umiestniť jeho stred. Správne sme mali zapísať:

```
canvas.create_image(bg.width()//2, bg.height()//2, image=bg)
```

Alebo lepšie riešenie bude využiť ďalší parameter príkazu `create_image()`, ktorým sa dá nastaviť iné určenie umiestnenia obrázka. Parameter `anchor='center'` znamená, že `(x, y)` je v strede, `anchor='n'` označuje, že je v strede hornej strany obrázka (tzv. „sever“), `anchor='w'` označuje stred ľavej strany (tzv. „západ“) a `anchor='nw'` je ľavý horný roh obrázka, t.j. „severozápad“, atď. Takže vykreslenie obrázka ako pozadia grafickej plochy bude teraz vyzerať takto:

```
canvas.create_image(0, 0, image=bg, anchor='nw')
```

Pridáme animovanú sériu obrázkov vtáčika:

```
import tkinter

win = tkinter.Tk()
bg = tkinter.PhotoImage(file='jazero.png')
canvas = tkinter.Canvas(width=bg.width(), height=bg.height())
canvas.pack()
canvas.create_image(0, 0, image=bg, anchor='nw')

a = canvas.create_image(200, 150)
pole = [tkinter.PhotoImage(file=f'a1/vtak{i}.png') for i in range(8)]
i = 0
while True:
    canvas.itemconfig(a, image=pole[i])
    i = (i + 1) % len(pole)
    canvas.update()
    canvas.after(100)
```

22.2.1 Objekt Anim

Aby sa nám lepšie manipulovalo s animovaným obrázkom, zapuzdrime to do triedy `Anim`:

```
import tkinter

win = tkinter.Tk()
bg = tkinter.PhotoImage(file='jazero.png')
canvas = tkinter.Canvas(width=bg.width(), height=bg.height())
canvas.pack()
canvas.create_image(0, 0, image=bg, anchor='nw')

class Anim:
    canvas = None
    def __init__(self, x, y, pole):
        self.id = self.canvas.create_image(x, y)
        self.pole = pole
        self.faza = 0
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def dalsia_faza(self):
        self.canvas.itemconfig(self.id, image=self.pole[self.faza])
        self.faza = (self.faza + 1) % len(self.pole)

Anim.canvas = canvas
pole = [tkinter.PhotoImage(file=f'a1/vtak{i}.png') for i in range(8)]

a1 = Anim(200, 150, pole)
while True:
    a1.dalsia_faza()
    canvas.update()
    canvas.after(100)
```

Keďže sme túto animačnú triedu vymysleli týmto spôsobom, môžeme veľmi jednoducho pridať niekoľko ďalších rovnakých objektov na rôznych pozíciách:

```
...

a1 = Anim(200, 150, pole)
a2 = Anim(300, 250, pole)
a3 = Anim(400, 200, pole)
while True:
    a1.dalsia_faza()
    a2.dalsia_faza()
    a3.dalsia_faza()
    canvas.update()
    canvas.after(100)
```

Prípadne môžeme všetky animované objekty uložiť do poľa:

```
...

apole = [Anim(200, 150, pole), Anim(300, 250, pole), Anim(400, 200, pole)]

while True:
    for a in apole:
        a.dalsia_faza()
    canvas.update()
    canvas.after(100)
```

22.2.2 Udalosti

Ďalším krokom vylepšovania aplikácie sú udalosti: časovač a klikanie myšou. Časovačom `timer()` nahradíme nekonečný while-cyklus. Klikaním myšou budeme definovať nové objekty triedy `Anim`:

```
...

apole = []

def klik(event):
    apole.append(Anim(event.x, event.y, pole))

def timer():
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    for a in apole:
        a.dalsia_faza()
    canvas.after(100, timer)

canvas.bind('<Button-1>', klik)
timer()
```

22.2.3 Trieda Plocha

Zapuzdrime všetky príkazy okrem vytvorenia poľa animovaných obrázkov do triedy `Plocha`:

```
import tkinter

class Plocha:
    def __init__(self, subor, pole):
        win = tkinter.Tk()
        self.bg = tkinter.PhotoImage(file=subor)
        self.canvas = tkinter.Canvas(width=self.bg.width(), height=self.bg.height())
        self.canvas.pack()
        self.canvas.create_image(0, 0, image=self.bg, anchor='nw')
        Anim.canvas = self.canvas
        self.apole = []
        self.pole = pole
        self.canvas.bind('<Button-1>', self.klik)
        self.timer()

    def klik(self, event):
        self.apole.append(Anim(event.x, event.y, self.pole))

    def timer(self):
        for a in self.apole:
            a.dalsia_faza()
        self.canvas.after(100, self.timer)

class Anim:
    canvas = None
    def __init__(self, x, y, pole):
        self.id = self.canvas.create_image(x, y)
        self.pole = pole
        self.faza = 0

    def dalsia_faza(self):
        self.canvas.itemconfig(self.id, image=self.pole[self.faza])
        self.faza = (self.faza + 1) % len(self.pole)

pole = [tkinter.PhotoImage(file=f'a1/vtak{i}.png') for i in range(8)]

Plocha('jazero.png', pole)
```

Opäť sa objavuje známa chyba:

```
RuntimeError: Too early to create image
```

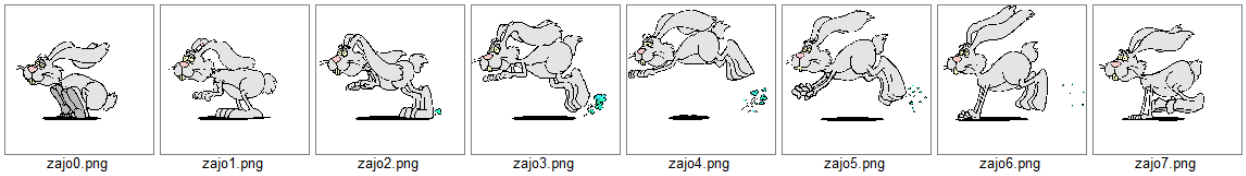
Volanie `PhotoImage()` je tu skôr ako vzniklo grafické okno pomocou `tkinter.Tk()`. Preto vytvorenie okna presunieme von z triedy pre vytvorením poľa `pole`:

```
...

win = tkinter.Tk()
pole = [tkinter.PhotoImage(file=f'a1/vtak{i}.png') for i in range(8)]

Plocha('jazero.png', pole)
```

Teraz je to už funkčné. Potrebujeme pridať ďalšie typy animovaných obrázkov. Ďalšia sada obrázkov animuje skákajúceho zajaca:



Týchto 8 obrázkov prenesieme do podadresára `a2` a pridajme tieto príkazy:

```
import tkinter
from random import randrange as rr

class Plocha:
    def __init__(self, subor, *pole):
        self.bg = tkinter.PhotoImage(file=subor)
        self.canvas = tkinter.Canvas(width=self.bg.width(), height=self.bg.height())
        self.canvas.pack()
        self.canvas.create_image(0, 0, image=self.bg, anchor='nw')
        Anim.canvas = self.canvas
        self.apole = []   # pole animovanych objektov
        self.pole = pole  # pole animovanych serii obrazkov
        self.canvas.bind('<Button-1>', self.klik)
        self.timer()

    def klik(self, event):
        self.apole.append(Anim(event.x, event.y, self.pole[rr(len(self.pole))]))

    def timer(self):
        for a in self.apole:
            a.dalsia_faza()
        self.canvas.after(100, self.timer)

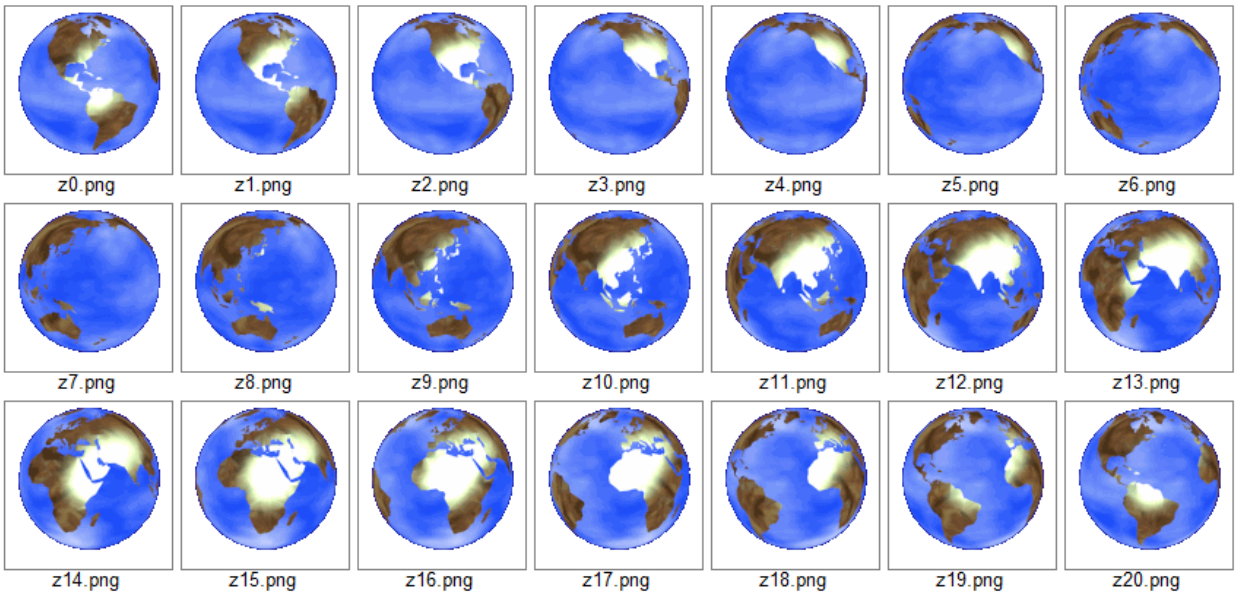
class Anim:
    canvas = None
    def __init__(self, x, y, pole):
        self.id = self.canvas.create_image(x, y)
        self.pole = pole
        self.faza = 0

    def dalsia_faza(self):
        self.canvas.itemconfig(self.id, image=self.pole[self.faza])
        self.faza = (self.faza + 1) % len(self.pole)

win = tkinter.Tk()
pole1 = [tkinter.PhotoImage(file=f'a1/vtak{i}.png') for i in range(8)]
pole2 = [tkinter.PhotoImage(file=f'a2/zajo{i}.png') for i in range(8)]

Plocha('jazero.png', pole1, pole2)
```

Podobne môžeme pridať týchto 21 obrázkov zemegule (presunieme ich do podadresára `a3`):



Okrem týchto troch animovaných sérií môžeme pridať preklopené vtáčiky a zajaze:

```
from PIL import Image, ImageTk

...

win = tkinter.Tk()
pole1 = [tkinter.PhotoImage(file=f'a1/vtak{i}.png') for i in range(8)]
pole1a = [ImageTk.PhotoImage(Image.open(f'a1/vtak{i}.png').transpose(Image.FLIP_LEFT_
↪RIGHT)) for i in range(8)]
pole2 = [tkinter.PhotoImage(file=f'a2/zajo{i}.png') for i in range(8)]
pole2a = [ImageTk.PhotoImage(Image.open(f'a2/zajo{i}.png').transpose(Image.FLIP_LEFT_
↪RIGHT)) for i in range(8)]
pole3 = [tkinter.PhotoImage(file=f'a3/z{i}.png') for i in range(21)]

Plocha('jazero.png', pole1, pole1a, pole2, pole2a, pole3)
```

22.2.4 Trieda Program

Posledným krokom pri vytváraní grafickej aplikácie bude vytvorenie triedy Program, pričom všetky globálne akcie (vytváranie polí s animáciami, volanie `Plocha()`) presunieme do inicializácie tejto triedy. Zároveň pozmeníme veľkosť animovanej zemegule:

```
...

class Program:
    def __init__(self):

        def resize(img, pomer):
            return img.resize((int(img.width*pomer), int(img.height*pomer)))

        win = tkinter.Tk()
        win.title('moja animovana aplikacia')
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        pole1 = [tkinter.PhotoImage(file=f'a1/vtak{i}.png') for i in range(8)]
        pole1a = [ImageTk.PhotoImage(Image.open(f'a1/vtak{i}.png').transpose(Image.
↪FLIP_LEFT_RIGHT)) for i in range(8)]
        pole2 = [tkinter.PhotoImage(file=f'a2/zajo{i}.png') for i in range(8)]
        pole2a = [ImageTk.PhotoImage(Image.open(f'a2/zajo{i}.png').transpose(Image.
↪FLIP_LEFT_RIGHT)) for i in range(8)]
        pole3 = [ImageTk.PhotoImage(resize(Image.open(f'a3/z{i}.png'), 2)) for i in␣
↪range(21)]
        img = resize(Image.open('python-logo.png'), 0.7)
        pole4 = [ImageTk.PhotoImage(img.rotate(uhol, expand=True)) for uhol in␣
↪range(0, 360, 10)]

        Plocha('jazero.png', pole1, pole1a, pole2, pole2a, pole3, pole4)

Program()
```

22.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Spojazdnite kompletnú aplikáciu z prednášky.
2. Vymeňte v aplikácii pozadie: nájdite na internete vhodný obrázok rozmerov aspoň 800x600, najlepšie vo formáte `.jpg` a nahraďte ním `jazero.png`.
3. Ako pozadie aplikácie zvoľte nejaký menší obrázok, ktorý rozkopírujete vedľa seba a pod seba tak, aby sa zaplnil `canvas` veľkosti napr. 800x600. Môžete použiť jednu z bitmáp: pozadie.zip
   - pomocou `Image` vytvorte jeden veľký obrázok požadovaných rozmerov a do neho príslušný počet krát opečiatkujte jednu z bitmáp a tento výsledok použite ako pozadie canvasu grafickej aplikácie
4. Všetky obrázky v obrazky.zip sú vo formáte `.bmp` a preto nemajú priesvitné časti. Prečítajte ich a pomocou `Image` z nich vyrobte obrázky v móde `'RGBA'` a farbu v pixeli na súradnici (0, 0) nahraďte v týchto obrázkoch priesvitnými pixelmi.
   - v grafickú aplikáciu zmeňte tak, aby sa namiesto animovaných obrázkov klikaním pridávali upravené bitmapy z tejto skupiny
   - pre každú z týchto bitmáp môžete pripraviť 2 fázy animácie: 1. je pôvodný obrázok, 2. je obrázok zmenšený na 90%
5. Všetky obrázky v súbore animacie.zip obsahujú viac fáz. Treba ich správne rozstrihať a vytvoriť z nich polia obrázkov pre `tkinter` tak, aby sa dali použiť v našom animačnom programe.
   - všetky tieto obrázky už majú dobre nastavené priesvitné pixely
6. V úlohe (5) ste rozstrihali 3 väčšie obrázky na fázy animácie. Pre dve z nich `potvorka1.png` a `potvorka2.png` treba pripraviť aj otočené fázy o 90, 180 a 270 stupňov, t.j. každej vyrobíte ďalšie tri animované série obrázkov.
   - otestujte ich vo vašej grafickej aplikácii

KAPITOLA 23

23. Pohybujúce sa obrázky

Obrázkom v nejakej väčšej scéne, ktoré sa animujú, hýbu, spolupracujú navzájom, resp. sa dajú modifikovať prostredníctvom myši alebo klávesnice, sa zvykne hovoriť sprite. V predchádzajúcej prednáške sme do grafickej plochy umiestňovali nejaké animované objekty a už toto boli najjednoduchšie verzie sprite (škriatkov).

23.1 Grafická aplikácia so spritami

V dnešnej prednáške sa budeme znovu zaoberať „spritami“, ale pridáme im ďalšiu funkčnosť:

- okrem toho, že sa grafické objekty v ploche budú nejako animovať, môžu sa automaticky po scéne pohybovať, resp. meniť smer pohybu na okraji plochy
- objektmi nemusia byť len obrázky vo formáte `PhotoImage`, ale aj nakreslené útvary, napr. farebné obdĺžniky
- objekty budeme môcť pohybovať pomocou myši
- v niektorých sitáciách budeme riešiť aj vzájomnú polohu viacerých objektov

23.1.1 Automatický pohyb

Predpokladajme, že všetky grafické objekty môžu mať určený aj svoj smer (vektor) pohybu, t.j. dvojicu `(dx, dy)`, ktorá pri tikaní časovača mení polohu objektu `(x, y)` (t.j. posúva objekt). Zároveň zadefinujeme správanie týchto objektov na okraji plochy: objekt sa odrazí od okrajov ako sa odráža guličníková guľa od okrajov hracieho stola.

```
import tkinter
from random import randrange as rr

class Zaklad:
    canvas = None
    sirka = 0
    vyska = 0
    def __init__(self, x, y, dx=0, dy=0, sirka=0, vyska=0):
        self.x, self.y = x, y
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        self.dx, self.dy = dx, dy
        self.w2, self.h2 = sirka//2, vyska//2
        self.id = None

    def timer(self):
        if self.dx!=0 or self.dy!=0:
            if self.x+self.dx < self.w2: self.dx = abs(self.dx)
            if self.y+self.dy < self.h2: self.dy = abs(self.dy)
            if self.x+self.dx > self.sirka - self.w2: self.dx = -abs(self.dx)
            if self.y+self.dy > self.vyska - self.h2: self.dy = -abs(self.dy)
            self.x += self.dx
            self.y += self.dy
            if self.id is not None:
                self.canvas.move(self.id, self.dx, self.dy)

#------------------------------------------------------------------------

class Ramik(Zaklad):
    def __init__(self, x, y, dx=0, dy=0, sirka=30, vyska=30, farba=''):
        super().__init__(x, y, dx, dy, sirka, vyska)
        if farba == 'random':
            farba = f'#{rr(256**3):06x}'
        self.id = self.canvas.create_rectangle(
            x-self.w2, y-self.h2, x+self.w2, y+self.h2, fill=farba)

#------------------------------------------------------------------------

class Obrazok(Zaklad):
    def __init__(self, obr, x, y, dx=0, dy=0):
        if not isinstance(obr, list):
            obr = [obr]
        self.pole = obr
        self.faza = 0
        super().__init__(x, y, dx, dy, obr[0].width(), obr[0].height())
        self.id = self.canvas.create_image(x, y, image=obr[0])

    def timer(self):
        if len(self.pole) > 1:
            self.canvas.itemconfig(self.id, image=self.pole[self.faza])
            self.faza = (self.faza + 1) % len(self.pole)
        super().timer()

################################################

class Plocha(tkinter.Canvas):
    def __init__(self, *param, **pparam):
        super().__init__(*param, **pparam)
        self.pack()
        Zaklad.canvas = self
        Zaklad.sirka = int(self['width'])
        Zaklad.vyska = int(self['height'])
        self.pole = []
        self.timer()

    def timer(self):
        for obj in self.pole:
            obj.timer()
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        self.after(100, self.timer)

    def pridaj(self, obj):
        self.pole.append(obj)
```

Otestujeme:

```
p = Plocha(width=800, height=500, bg='green')
for i in range(10):
    p.pridaj(Ramik(rr(0,800), rr(0, 500), rr(-2, 2), rr(-2, 2), 60, 40, farba='random
↪'))
zemegula = [tkinter.PhotoImage(file=f'a3/z{i}.png') for i in range(21)]
for i in range(5):
    p.pridaj(Obrazok(zemegula, rr(0,800), rr(0, 500), rr(-2, 2), rr(-2, 2)))
```

23.1.2 Ťahanie objektov myšou

Pridáme metódy:

```
import tkinter
from random import randrange as rr

class Zaklad:
    canvas = None
    sirka = 0
    vyska = 0
    def __init__(self, x, y, dx=0, dy=0, sirka=0, vyska=0):
        self.x, self.y = x, y
        self.dx, self.dy = dx, dy
        self.w2, self.h2 = sirka//2, vyska//2
        self.id = None

    def vnutri(self, x, y):
        return abs(self.x-x) <= self.w2 and abs(self.y-y) <= self.h2

    def mouse_move(self, x, y):
        if self.id is not None:
            self.canvas.move(self.id, x-self.x, y-self.y)
        self.x, self.y = x, y

    def mouse_down(self):
        pass

    def mouse_up(self):
        pass

    def timer(self):
        if self.dx!=0 or self.dy!=0:
            if self.x+self.dx < self.w2: self.dx = abs(self.dx)
            if self.y+self.dy < self.h2: self.dy = abs(self.dy)
            if self.x+self.dx > self.sirka - self.w2: self.dx = -abs(self.dx)
            if self.y+self.dy > self.vyska - self.h2: self.dy = -abs(self.dy)
            self.x += self.dx
            self.y += self.dy
            if self.id is not None:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
                self.canvas.move(self.id, self.dx, self.dy)

#------------------------------------------------------------------------

class Ramik(Zaklad):
    def __init__(self, x, y, dx=0, dy=0, sirka=30, vyska=30, farba=''):
        super().__init__(x, y, dx, dy, sirka, vyska)
        if farba == 'random':
            farba = f'#{rr(256**3):06x}'
        self.id = self.canvas.create_rectangle(
            x-self.w2, y-self.h2, x+self.w2, y+self.h2, fill=farba)

    def mouse_down(self):
        self.canvas.itemconfig(self.id, width=3, outline='red')

    def mouse_up(self):
        self.canvas.itemconfig(self.id, width=1, outline='black')


#------------------------------------------------------------------------

class Obrazok(Zaklad):
    def __init__(self, obr, x, y, dx=0, dy=0):
        if not isinstance(obr, list):
            obr = [obr]
        self.pole = obr
        self.faza = 0
        super().__init__(x, y, dx, dy, obr[0].width(), obr[0].height())
        self.id = self.canvas.create_image(x, y, image=obr[0])

    def timer(self):
        if len(self.pole) > 1:
            self.canvas.itemconfig(self.id, image=self.pole[self.faza])
            self.faza = (self.faza + 1) % len(self.pole)
        super().timer()

################################################

class Plocha(tkinter.Canvas):
    def __init__(self, *param, **pparam):
        super().__init__(*param, **pparam)
        self.pack()
        Zaklad.canvas = self
        Zaklad.sirka = int(self['width'])
        Zaklad.vyska = int(self['height'])
        self.bind('<Button-1>', self.mouse_down)
        self.bind('<B1-Motion>', self.mouse_move)
        self.bind('<ButtonRelease-1>', self.mouse_up)
        self.pole = []
        self.tahany = None
        self.timer()

    def timer(self):
        for obj in self.pole:
            obj.timer()
        self.after(100, self.timer)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def mouse_down(self, event):
        for obj in reversed(self.pole):
            if obj.vnutri(event.x, event.y):
                self.tahany = obj
                self.dx, self.dy = event.x - obj.x, event.y - obj.y
                obj.mouse_down()
                return
        self.tahany = None

    def mouse_move(self, event):
        if self.tahany is not None:
            self.tahany.mouse_move(event.x-self.dx, event.y-self.dy)

    def mouse_up(self, event):
        if self.tahany is not None:
            self.tahany.mouse_up()
            self.tahany = None

    def pridaj(self, obj):
        self.pole.append(obj)
```

Otestujeme:

```
p = Plocha(width=800, height=500, bg='green')
for i in range(10):
    p.pridaj(Ramik(rr(0,800), rr(0, 500), rr(-2, 2), rr(-2, 2), 60, 40, farba='random
↪'))
zemegula = [tkinter.PhotoImage(file=f'a3/z{i}.png') for i in range(21)]
for i in range(5):
    p.pridaj(Obrazok(zemegula, rr(0,800), rr(0, 500), rr(-2, 2), rr(-2, 2)))
```

23.1.3 Akcie pri pustení myši

Budeme riešiť takúto úlohu: ...

```
import tkinter
from random import randrange as rr

class Zaklad:
    canvas = None

    def __init__(self, x, y, sirka=0, vyska=0):
        self.x, self.y = self.x0, self.y0 = x, y
        self.w2, self.h2 = sirka//2, vyska//2
        self.id = None
        self.moze_tahat = True
        self.vnom = None

    def vnutri(self, x, y):
        return abs(self.x-x) <= self.w2 and abs(self.y-y) <= self.h2

    def mouse_move(self, x, y):
        if self.id is not None:
            self.canvas.move(self.id, x-self.x, y-self.y)
        self.x, self.y = x, y
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def mouse_down(self):
        pass

    def mouse_up(self):
        pass

    def timer(self):
        pass

    def domov(self):
        self.mouse_move(self.x0, self.y0)

#------------------------------------------------------------------------

class Ramik(Zaklad):
    def __init__(self, x, y, sirka=30, vyska=30, farba=''):
        super().__init__(x, y, sirka, vyska)
        if farba == 'random':
            farba = f'#{rr(256**3):06x}'
        self.id = self.canvas.create_rectangle(
            x-self.w2, y-self.h2, x+self.w2, y+self.h2, fill=farba)

    def mouse_down(self):
        self.canvas.itemconfig(self.id, width=3, outline='red')

    def mouse_up(self):
        self.canvas.itemconfig(self.id, width=1, outline='black')


#------------------------------------------------------------------------

class Obrazok(Zaklad):
    def __init__(self, obr, x, y, dx=0, dy=0):
        if not isinstance(obr, list):
            obr = [obr]
        self.pole = obr
        self.faza = 0
        super().__init__(x, y, obr[0].width(), obr[0].height())
        self.id = self.canvas.create_image(x, y, image=obr[0])

    def timer(self):
        if len(self.pole) > 1:
            self.canvas.itemconfig(self.id, image=self.pole[self.faza])
            self.faza = (self.faza + 1) % len(self.pole)

################################################

class Plocha(tkinter.Canvas):
    def __init__(self, *param, **pparam):
        super().__init__(*param, **pparam)
        self.pack()
        Zaklad.canvas = self
        Zaklad.sirka = int(self['width'])
        Zaklad.vyska = int(self['height'])
        self.bind('<Button-1>', self.mouse_down)
        self.bind('<B1-Motion>', self.mouse_move)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        self.bind('<ButtonRelease-1>', self.mouse_up)
        self.pole = []
        self.tahany = None
        self.ciele = []
        self.timer()

    def timer(self):
        for obj in self.pole:
            obj.timer()
        self.after(100, self.timer)

    def mouse_down(self, event):
        for obj in reversed(self.pole):
            if obj.moze_tahat and obj.vnutri(event.x, event.y):
                self.tahany = obj
                self.dx, self.dy = event.x - obj.x, event.y - obj.y
                obj.mouse_down()
                return
        self.tahany = None

    def mouse_move(self, event):
        if self.tahany is not None:
            self.tahany.mouse_move(event.x-self.dx, event.y-self.dy)

    def mouse_up(self, event):
        if self.tahany is not None:
            kto = self.tahany
            kto.mouse_up()

            for ciel in self.ciele:
                if ciel.vnom is kto:
                    ciel.vnom = None
            for ciel in self.ciele:
                if ciel.vnutri(kto.x, kto.y):
                    if ciel.vnom is not None:
                        ciel.vnom.domov()
                    ciel.vnom = kto
                    kto.mouse_move(ciel.x, ciel.y)
                    break
            else:
                kto.domov()


            self.tahany = None

    def pridaj(self, obj):
        self.pole.append(obj)
```

Pri testovaní použijeme napr. tieto obrázky číslic:

```
p = Plocha(width=800, height=500, bg='green')
for i in range(8):
    r = Ramik(i*80+50, 200, 70, 70)
    r.moze_tahat = False
    p.pridaj(r)
    p.ciele.append(r)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
for i in range(10):
    p.pridaj(Obrazok(tkinter.PhotoImage(file=f'number{i}.png'), i*50+50, 100))
```

23.2 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

Dnešné cvičenie je venované príprave na záverečný test. Tento test bude prebiehať 12.12. v posluchárňach A a B. Pokúste sa vyriešiť čo najviac z týchto vybraných úloh, pričom sa snažte nepoužívať ako pomôcku počítač. Tieto tréningové úlohy sú mierne upravené úlohy z testov v rokoch 2014 a 2015:

1. Zistite, čo vypočíta táto rekurzívna funkcia:

```
def pocitaj(n):
    if n < 2:
        return n
    return pocitaj(n-2) + pocitaj(n-1)

for i in 5, 5.5, 6, 6.5:
    print(i, pocitaj(i))
```

2. Dopíšte funkciu `vyrob(dlzky)`, ktorá vytvorí dvojrozmerné pole celých čísel tak, že ak parameter `dlzky` je pole celých čísel, tak tieto označujú dĺžky riadkov vytváraného dvojrozmerného poľa. Počet prvkov poľa `dlzky` označuje počet riadkov vytváraného poľa. Prvky vytváraného poľa pritom postupne zaplňte hodnotami od 1.

```
def vyrob(dlzky):
    pole = []
    _________________
    for d in dlzky:
        _________________
        _________________

    return pole
```

```
>>> pole = vyrob([3, 2, 4])
>>> print(*pole, sep='\n')
[1, 2, 3]
[4, 5]
[6, 7, 8, 9]
```

   Pri dopisovaní kódu do funkcie nemusíte dodržať naznačený počet riadkov programu

3. Textový súbor `'subor.txt'` v každom riadku obsahuje niekoľko slov oddelených medzerami. Nasledovná funkcia by mala vytvoriť dvojrozmerné pole znakových reťazcov, ktoré bude v každom riadku obsahovať ako prvky slová z príslušného riadku súboru:

```
def urob(meno):
    with open(meno) as subor:
        vysledok = []
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        while vysledok:
            riadok = subor.read()
            if riadok:
                return vysledok
            riadok = riadok.strip()
            riadok.append(vysledok)
```

Opravte všetky chyby. Nedopisujte nové príkazy - opravte len chybné zápisy.

4. V triede `Kniha` si ukladáme informácie o knihách:

```
class Kniha:
    def __init__(self, autor, titul, cena):
        self.pole = [autor, titul, cena]

    def autor(self, zmen=None):
        ...
```

Dopíšte metódu `autor()` tak, aby volanie bez parametrov vrátilo autora knihy a volanie s jedným parametrom zmenilo autora, napr.

```
>>> k = Kniha('Doyle', 'Sherlock Holmes', 11.5)
>>> k.autor()
'Doyle'
>>> k.autor('sir Arthur Conan Doyle')
>>> k.pole
['sir Arthur Conan Doyle', 'Sherlock Holmes', 11.5]
```

5. Zistite, čo sa vypíše:

```
>>> pole1 = [3, 'sedem', 3.14]
>>> pole2 = ['dog', 'cat', 'mouse', 'duck']
>>> pole3 = {'sedem':[3]*3, 3.14: pole2, 'cat': pole1}
>>> pole3[pole3[pole2[1]][2]][2]
```

6. V triede `Mnozina` je 5 chýb. Opravte ich! Neopravujte kód, ktorý nie je chybný.

```
class Mnozina:
    def __init__(self):
        self.pole = []

    def __str__(self):
        print(tuple(sorted(self.pole)))

    def add(self, cislo):
        if cislo in self:
            self.pole.append(cislo)

    def discard(self, cislo):
        if cislo in self:
            self.pole.pop(cislo)

    def __contains__(self, cislo):
        return self.pole.index(cislo) >= 0

    def __len__(self):
        return len(self.pole)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def zjednotenie(self, ina):
        for i in ina.pole:
            self.append(i)
```

7. Dopíšte funkciu `nechaj_float(pole)`, ktorá v poli znakových reťazcov ponechá len tie, ktoré reprezentujú desatinné čísla (dajú sa previesť konverznou funkciou `float()`). Dopisuje len medzi riadky `while` a `del`:

```
def nechaj_float(pole):
    i = 0
    while i < len(pole):




        del pole[i]
```

```
>>> pole = ['3..', '1e1', '7', '' , '1e1e', '.7.']
>>> nechaj_float(pole)
>>> pole
['1e1', '7']
```

8. Dané sú dve polia `pole1` a `pole2`, ktoré majú rovnaký počet prvkov. Vytvorte funkciu, ktorá z takýchto dvoch polí vytvorí a vráti nové asociatívne pole. V tomto asociatívnom poli sú kľúčmi prvky z prvého poľa a hodnotami sú príslušné prvky druhého poľa. V tele tejto funkcie môžete použiť štandardnú funkciu `zip()`.

```
def urob(pole1, pole2):


    return ...
```

```
>>> a = urob(['druh', 'vaha', 'vek'],['slon', 1000, 10])
>>> a
{'druh': 'slon', 'vek': 10, 'vaha': 1000}
```

9. Funkcia `vsetky()` nejako spracováva pole množín:

```
def vsetky(pole_mnozin):
    vysl, b = set(), True
    for m in pole_mnozin:
        if b:
            vysl |= m
        else:
            vysl -= m
        b = not b
    return vysl
```

Zistite, čo treba dosadiť do premennej `x`, aby sme dostali tento výsledok:

```
>>> x = ________________________
>>> vsetky([set(range(3, 10, 2)), x, set(range(5, 15, 3))])
{5, 7, 8, 11, 14}
```

10. Zapíšte funkciu `urob(n)`, ktorá vytvorí pole tretích mocnín čísel od 1 do `n`. Zapíšte ju pomocou generátorovej notácie:

```
def urob(n):

    return [...]
```

```
>>> urob(5)
[1, 8, 27, 64, 125]
```

11. Máme daný slovník `d = {'prvy' :7, 'druhy' :3, 'treti' :5, 'stvrty' :6, 'piaty' :2}`. Zistite, čo vráti tento kód:

```
'.'.join(b for a, b in sorted((d[x], x) for x in d))
```

KAPITOLA 24

24. Úvodná prednáška v letnom semestri

24.1 Priebeh letného semestra

Pravidlá predmetu v letnom semestri sú skoro indentické so zimným semestrom. Rozdiely sú len tieto:

skúška

sa skladá z dvoch častí:

1. jeden písomný test (v pondelok **14.5.** o 14:00 v posluchárni A) - max. 40 bodov
2. praktická skúška pri počítači (v skúškovom období) - max. 60 bodov

hodnotenie skúšky

spočítajú sa body z písomného testu, praktickej skúšky, príp. bodov za semestrálny projekt:

- **:-)** známka **A** 88 bodov
- **:-)** známka **B** 81 bodov
- **:-)** známka **C** 74 bodov
- **:-)** známka **D** 67 bodov
- **:-)** známka **E** 60 bodov
- **:-(** známka **Fx** menej ako 60 bodov

24.2 Turingov stroj

24.2.1 Turingov stroj

Alan Turing (1912-1954) sa považuje za zakladateľa modernej informatiky - rok 2012 sa na celom svete oslavoval ako Turingov rok.

Turingov stroj je veľmi zjednodušený model počítača, vďaka ktorému sa informatika dokáže zaoberať zložitosťou problémov - Turing ho publikoval v roku 1936 (keď mal 24 rokov).

Základná idea takéhoto stroja je nasledovná:

- pracuje na nekonečnej páske - postupnosť políčok, na každom môže byť nejaký symbol (my si to zjednodušíme obyčajnými znakmi, t.j. jednoznakovými reťazcami) - táto páska je nekonečná oboma smermi
- nad páskou sa pohybuje čítacia/zapisovacia hlava, ktorá vie prečítať symbol na páske a prepísať ho iným symbolom
- samotný stroj (riadiaca jednotka) sa stále nachádza v jednom zo svojich stavov: na základe momentálneho stavu a prečítaného symbolu na páske sa riadiaca jednotka rozhoduje, čo bude robiť
- na začiatku sa na políčkach pásky nachádzajú znaky nejakého vstupného reťazca (postupnosť symbolov), stroj sa nachádza v počiatočnom stave (stavy pomenujeme ľubovoľnými reťazcami znakov, napr. `'s1'` alebo `'end'`), celý zvyšok nekonečnej pásky obsahuje prázdne symboly (my sa dohodneme, že to budeme označovať symbolom `'_'`)
- činnosť stroja (program) sa definuje špeciálnou dvojrozmernou tabuľkou, tzv. **pravidlami**
- každé pravidlo je takáto pätica: (`stav1`, `znak1`, `znak2`, `smer`, `stav2`) a popisuje:
  - keď je stroj v stave `stav1` a na páske (pod čítacou hlavou) je práve symbol `znak1`, tak ho stroj prepíše týmto novým symbolom `znak2`, posunie sa na páske podľa zadaného smeru `smer` o (-1, 0, 1) políčko a zmení svoj stav na `stav2`
  - napr. pravidlo `(s0, a, b, >, s1)` označuje: v stave `'s0'` so symbolom `'a'` na páske ho zmení na `'b'`, posunie sa o jedno políčko vpravo a zmení svoj stav na `'s1'`
  - pre lepšiu čitateľnosť budeme smery na posúvanie hlavy označovať pomocou znakov `'<'` (vľavo), `'>'` (vpravo), `'='` (bez posunu)
  - takéto pätice sa niekedy označujú aj v tomto tvare: (`stav1`, `znak1`) -> (`znak2`, `smer`, `stav2`), t.j. pre danú dvojicu stav a znak, sa zmení znak
- program má väčšinou viac stavov, z ktorých niektoré sú špeciálne, tzv. koncové a majú takýto význam:
  - keď riadiaca jednotka prejde do koncového stavu, výpočet stroja sa zastaví a stroj oznámi, že **akceptoval** vstupné slovo, vtedy sa môžeme pozrieť na obsah pásky a táto informácia môže byť výsledkom výpočtu
- stroj sa zastaví aj v prípade, že neexistuje pravidlo, pomocou ktorého by sa dalo pokračovať (stroj skončil v nekoncovom stave), vtedy
  - hovoríme, že stroj oznámil, že **zamietol** (neakceptoval) vstupné slovo
- zrejme sa takýto stroj môže niekedy aj zacykliť a neskončí nikdy (pre informatikov je aj toto veľmi dôležitá informácia)

Na internete sa dá nájsť veľa rôznych zábavných stránok, ktoré sa venujú Turingovmu stroju, napr.

- Alan Turing's 100th Birthday
- LEGO Turing Machine
- Turing Machine-IA Lego Mindstorms

- A Turing Machine - Overview

Zostavme náš prvý program pre Turingov stroj:

(0, a)	->	(A, >, 0)	
(0, _)	->	(_, =, end)	

Vidíme, že pre počiatočný stav sú tu dve pravidlá: buď je na páske symbol `'a'` alebo symbol `'_'`, ktorý označuje prázdne políčko. Ak teda čítacia hlava má pod sebou na páske symbol `'a'`, tak ho prepíše na symbol `'A'` a posunie hlavu na políčko vpravo. Riadiaca jednotka pritom stále ostáva v stave `0`. Vďaka tomuto pravidlu sa postupne všetky písmená `'a'` nahradia `'A'`. Ak sa pritom narazí na iný symbol, prvé pravidlo sa už nebude dať použiť a stroj sa pozrie, či nemá pre tento prípad iné pravidlo. Ak je na páske už len prázdny symbol, stroj sa zastaví a oznámi radostnú správu, že vstupné slovo **akceptoval** a vyrobil z neho nový reťazec. Ak ale bude na páske za postupnosťou `'a'` aj nejaké iné písmeno, stroj nenájde pravidlo, ktoré by mohol použiť a preto sa zastaví. Oznámi pritom správu, že takéto vstupné slovo **zamietol**.

Priebeh výpočtu pre vstupné slovo ‚aaa' by sme si mohli znázorniť napr. takto:

```
aaa
^ 0
Aaa
 ^ 0
AAa
  ^ 0
AAA_
   ^ 0
AAA_
   ^ end
True
```

`True` zmamená, že stroj sa úspešne zastavil v koncovom stave, teda stroj **akceptoval** vstupné slovo

Všimnite si, že v našej vizualizácii sa na páske automaticky objavujú prázdne symboly, keďže páska je nekonečná a okrem vstupného slova obsahuje práve tieto znaky.

Ak by sme zadali napr. slovo ‚aba', tak by výpočet prebiehal takto:

```
aba
^ 0
Aba
 ^ 0
False
```

`False` tu znamená, že stroj sa zastavil v inom ako koncovom stave, teda zamietol vstup

24.2.2 Návrh interprétra

Aby sme mohli s takýmto strojom lepšie experimentovať a mať možnosť si uvedomiť, ako sa pomocou neho riešenia úlohy, naprogramujeme si veľmi jednoduchý interpreter. Začneme tým, že navrhneme triedu `Paska`, ktorá bude popisovať pásku Turingovho stroja a jej metódy:

```
class Paska:
    def __init__(self, obsah=''):
        self.paska = list(obsah or '_')
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        self.poz = 0

    def symbol(self):
        return self.paska[self.poz]

    def zmen_symbol(self, znak):
        self.paska[self.poz] = znak

    def __repr__(self):
        return ''.join(self.paska) + '\n' + ' '*self.poz + '^'

    def vpravo(self):
        self.poz += 1
        if self.poz == len(self.paska):
            self.paska.append('_')

    def vlavo(self):
        if self.poz > 0:
            self.poz -= 1
        else:
            self.paska.insert(0, '_')

    def text(self):
        return ''.join(self.paska).strip('_')
```

Atribúty sú asi zrejmé:

- `paska` - zoznam (typu `list`), ktorého prvky sú jednoznakové reťazce - tieto reprezentujú políčka pásky
  - pásku Turingovho stroja by sme mohli namiesto zoznamu reprezentovať aj pomocou znakového reťazca, zvolili sme zoznam, nakoľko v ňom sa jednoduchšie mení hodnota jedného prvku
  - všimnite so zapis `self.paska = list(obsah or '_')` a konkrétne použitie operácie `or`, v tomto prípade má takýto význam: ak je premenná `obsah` prázdna (Python ju interpretuje ako `False`), výsledkom operácie je druhý operand, teda `'_'`, inak je výsledkom samotný reťazec `obsah`
- `poz` - pozícia čítacej/zapisovacej hlavy, pomocou tejto pozície budeme indexovať zoznam `paska`
  - metódami `vpravo()` a `vlavo()` sa páska automaticky nafukuje, ak by hlava prišla mimo momentálne prvky zoznamu

Túto triedu využije trieda `Turing`, v ktorej sa okrem pásky bude uchovávať aj samotný program, teda množina prepisovacích pravidiel:

```
class Turing:
    def __init__(self, program, obsah='', start=None, koniec={'end', 'stop'}):
        self.prog = {}
        self.stav = start
        for riadok in program.split('\n'):
            riadok = riadok.split()
            if len(riadok) == 5:
                stav1, znak1, znak2, smer, stav2 = riadok
                self.prog[stav1, znak1] = znak2, smer, stav2
                if self.stav is None:
                    self.stav = stav1
        self.paska = Paska(obsah)
        self.koniec = koniec
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def __repr__(self):
        return repr(self.paska) + ' ' + self.stav

    def krok(self):
        stav1, znak1 = self.stav, self.paska.symbol()
        try:
            znak2, smer, stav2 = self.prog[stav1, znak1]
        except KeyError:
            return False
        self.paska.zmen_symbol(znak2)
        self.stav = stav2
        if smer == '>':
            self.paska.vpravo()
        elif smer == '<':
            self.paska.vlavo()
        return True

    def rob(self, vypis=True):
        if vypis:
            print(self)
        while self.stav not in self.koniec:
            if not self.krok():
                return False
            if vypis:
                print(self)
        return True
```

Atribúty tejto triedy:

- `prog` je tabuľka pravidiel - tieto pravidlá tu môžu byť uvedené v ľubovoľnom poradí a riadiaca jednotka si vyhľadá to správne
  - každé pravidlo sa skladá z piatich prvkov: v akom sme stave, aký je symbol na páske, na aký symbol sa prepíše, ako sa posunie hlava (buď `'<'` alebo `'>'`) a do akého stavu sa prejde
  - pravidlá budeme ukladať do slovníka - asociatívneho poľa (typ `dict`) tak, že kľúčom bude dvojica **(stav, znak)** a hodnotou pre tento kľúč bude trojica **(nový_znak, smer_posunu, nový_stav)**
- `paska` bude „nekonečná“ postupnosť symbolov, využili sme tu našu triedu `Paska`
- `stav` označuje, momentálne meno stavu (na začiatku bude napr. `'s1'` alebo `'0'`), v ktorom sa nachádza Turingov stroj
- `koniec` označuje, množinu koncových stavov

Metódy

- `__init__()` inicializuje atribúty, má tieto parametre:
  - `program` zoznam pravidiel: zadáva sa ako jeden dlhý reťazec, v ktorom sú pravidlá oddelené znakom `'\n'`, každé pravidlo sa skladá z 5 prvkov, pravidlo, ktoré neobsahuje práve 5 prvkov, sa ignoruje (môžeme použiť, napr. ako komentár)
  - `obsah` počiatočný obsah pásky, hlava sa nastaví na prvý znak tohto reťazca
  - `start` počiatočný stav, v ktorom štartuje Turingov stroj, ak ho nezadáme, ako počiatočný sa vyberie stav z prvého pravidla
  - `koniec` je množina koncových stavov, predpokladáme že najčastejšie to bude `{'end', 'stop'}`

- `krok()` Turingov stroj vykoná jeden krok programu; vráti `False`, keď sa krok nepodarí vykonať (teda pre daný `stav` a `znak` na páske neexistuje pravidlo v slovníku - asociatívnom poli `prog`), inak vráti `True`
- `rob()` riadiaca jednotka bude postupne vykonávať kroky programu, kým sa stroj nezastaví
  - metóda vráti `True`, ak program akceptoval symboly na páske, inak vráti `False`

Do metódy `rob()` sme pridali kontrolný výpis, ktorý môžeme vypnúť parametrom `vypis`.

Turingov stroj otestujeme jediným pravidlom:

```
t = Turing('0 a A > 0', 'aaa')
print(t.rob())
```

Dostávame tento výpis:

```
aaa
^ 0
Aaa
 ^ 0
AAa
  ^ 0
AAA_
   ^ 0
False
```

Zrejme to nemohlo dopadnúť inak ako `False`, lebo náš program neobsahuje pravidlo, ktorého výsledkom by bol koncový stav.

Doplňme druhé pravidlo:

```
t = Turing('0 a A > 0\n0 _ _ = end', 'aaa')
print(t.rob())
```

```
aaa
^ 0
Aaa
 ^ 0
AAa
  ^ 0
AAA_
   ^ 0
AAA_
   ^ end
True
```

Ďalší turingov program bude akceptovať vstup len vtedy, keď obsahuje jediné slovo `'ahoj'`:

```
print(Turing('''
1 a a > 2
2 h h > 3
3 o o > 4
4 j j > 5
5 _ _ = stop
''', 'ahoj').rob())
```

Všimnite si, že stavy tohto programu sú **1**, **2**, **3**, **4**, **5** a **stop**, pričom **1** je počiatočný stav a **stop** je koncový stav.

Program zadaný vstup `'ahoj'` akceptuje, ale napr. `'hello'` nie. Doplňme program tak, aby akceptoval obe slová `'ahoj'` aj `'hello'` (na páske bude buď `'ahoj'` alebo `'hello'`):

```
prog = '''
1 a a > 2
1 h h > 6
2 h h > 3
3 o o > 4
4 j j > 5
5 _ _ = stop
6 e e > 7
7 l l > 8
8 l l > 9
9 o o > 5
'''
t = Turing(prog, 'hello')
print(t.rob())
```

Podobný tomuto je program, ktorý akceptuje vstup len vtedy, keď sa skladá z ľubovoľného počtu dvojíc znakov `'ok'`:

```
prog = '''
1 o o > 2
1 _ _ = end
2 k k > 1
'''
t = Turing(prog, 'ok'*100)
print(t.rob())
```

Zostavme ešte takýto program:

- predpokladáme, že na páske je postupnosť `0` a `1`, ktorá reprezentuje nejaké dvojkové číslo, napr. `'1011'` označuje desiatkové číslo `11`
- čítacia hlava je na začiatku nastavená na prvej číslici
- program pripočíta k tomuto dvojkovému číslu `1`, t.j. v prípade `'1011'` bude výsledok na páske `'1100'` teda číslo `12`
- program bude postupovať takto:
  - najprv nájde koniec vstupného reťazca, teda prázdny znak `'_'` za poslednou cifrou (toto sa zabezpečuje v stave `'s1'`)
  - potom predchádza od konca a všetky `'1'` nahrádza `'0'`
  - keď pritom príde na `'0'`, túto nahradí `'1'` a skončí
  - ak sa v čísle nachádzajú iba `'1'` a žiadna `'0'`, tak namiesto prázdneho znaku `'_'` pred číslom dá `'1'` a skončí

Program pre turingov stroj:

(s1, 0)	->	(0, >, s1)	
(s1, 1)	->	(1, >, s1)	
(s1, _)	->	(_, <, s2)	
(s2, 1)	->	(0, <, s2)	
(s2, 0)	->	(1, =, end)	
(s2, _)	->	(1, =, end)	

Otestujeme naším programom:

```
prog = '''
s1 0 0 > s1
s1 1 1 > s1
s1 _ _ < s2

s2 1 0 < s2
s2 0 1 = end
s2 _ 1 = end
'''
t = Turing(prog, '1011')
print(t.rob())
```

po spustení:

```
1011
^ s1
1011
 ^ s1
1011
  ^ s1
1011
   ^ s1
1011_
    ^ s1
1011_
   ^ s2
1010_
  ^ s2
1000_
 ^ s2
1100_
 ^ end
True
```

Záverečná ukážka demonštruje zložitejší Turingov stroj: zistí, či je vstup zložený len z písmen `'a'` a `'b'` a či je to palindrom:

```
palindrom = '''
s1 a _ > sa
s1 b _ > sb
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
s1 _ _ = end
sa a a > sa
sa b b > sa
sa _ _ < saa
saa a _ < s2
saa _ _ < end
sb a a > sb
sb b b > sb
sb _ _ < sbb
sbb b _ < s2
sbb _ _ < end
s2 a a < s2
s2 b b < s2
s2 _ _ > s1
'''
t = Turing(palindrom, 'aba')
print(t.rob())
```

```
aba
^ s1
_ba
 ^ sa
_ba
  ^ sa
_ba_
   ^ sa
_ba_
  ^ saa
_b__
 ^ s2
_b__
^ s2
_b__
 ^ s1
____
  ^ sb
____
 ^ sbb
____
^ end
True
```

24.3 Cvičenia

1. Pomocou programu `Turing` z prednášky otestujte tieto pravidlá

`(stav, x)`	`->`	`(y, >, stav)`	
`(stav, _)`	`->`	`(z, =, end)`	

```
prog = '''


'''
t = Turing(prog, ...)
print(t.rob())
```

Zvoľte takú vstupnú pásku, aby Turingov stroj akceptoval vstup (metóda `rob()` vráti `True`). Otestujte s rôzne veľkými páskami.

2. Zistite, aké reťazce bude akceptovať nasledovný Turingov stroj:

```
start q q > jedna
jedna q q > dva
dva q q > start
jedna _ _ = stop
```

Nájdite aspoň 3 rôzne dlhé reťazce aspoň dĺžky 8, ktoré budú akceptované.

3. Otestujte Turingov stroj z prednášky, ktorý pripočítaval 1 k dvojkovému zápisu pre nejaké väčšie číslo, napr. takto:

```
cislo = 1000000
retazec = ... #preved' cislo do dvojkovej sústavy (napr. funkciou bin())
t = Turing(...) #zavolaj Turingov stroj s daným ret'azcom
t.rob()
vysledok = t.paska.text()
# skontroluj, či je výsledok o 1 viac ako pôvodné číslo
```

   - Otestujte ešte pre väčšie číslo, napr. `2**100-1`. Metódu `rob()` nastavte tak, aby nerobila priebežný výpis.
   - Zistite, koľko pravidiel sa pritom vykonalo (metóda `krok()` bola úspešná) - zmodifikujte metódu `rob()` tak, aby zvyšovala počítadlo `self.pocet`

4. Zistite, čo robí nasledovný Turingov stroj:

```
q00 a a > q10
q00 b b > q01
q10 a a > q00
q10 b b > q11
q01 a a > q11
q01 b b > q00
q01 _ _ = end
q11 a a > q01
q11 b b > q10
```

Nájdite reťazce dĺžky 5, 6, 7, 8, ktoré budú akceptované týmto Turingovým strojom

5. Navrhnite Turingov stroj, ktorý zo vstupného reťazca `'matfyz'` vytvorí na páske slovo `'python'`. Pravidlá by mali mať jediný stav (okrem koncového). Otestujte napr.

```
>>> t = Turing(prog, 'matfyz')
>>> t.rob(False)
True
>>> t.paska.text()
'python'
```

6. Upravte pravidlá Turingovho stroja z 3. úlohy tak, aby na páske vzniklo dvojkové číslo o 1 menšie. Napr.

```
>>> t = Turing(prog, '1000')
>>> t.rob(False)
True
>>> t.paska.text()
'111'
```

7. Navrhinte Turingov stroj, ktorý predpokladá, že na páske je postupnosť znakov `'1'`. Po skončení práce stroja (metóda `rob()` vráti `True`) bude na páske celá táto postupnosť jednotiek skopírovaná za seba, napr.

```
>>> t = Turing(prog, '1111')
>>> t.rob(False)
True
>>> t.paska.text()
'1111_1111'
```

8. Zapíšte pravidlá pre takýto Turingov stroj:
   - predpokladá, že na páske je len postupnosť `'1'`, napr. `'11111'`
   - pred túto postupnosť (o jedno políčko vľavo) vloží znak `'0'`, napr. `'0 11111'` - na páske budeme vytvárať dvojkové číslo (`'0'` pred prázdnym políčkom) z postupnosti `'1'` za prázdnym políčkom
   - teraz bude opakovať túto činnosť:
     - kým je postupnosť jednotiek neprázdna, tak
     - z nej odoberie poslednú jednotku
     - k dvojkovému číslu pripočíta 1 (algoritmom z 3. úlohy)

   Na páske sa postupne budú objavovať tieto reťazce:

```
0 11111
1 1111
10 111
11 11
100 1
101
```

   Na koniec vznikne dvojková reprezentácia čísla, ktoré bolo zapísané pomocou 1. Otestujte napr.

```
>>> t = Turing(prog, '1'*13)
>>> t.rob(False)
True
>>> t.paska.text()
'1101'
```

KAPITOLA 25

25. Zásobníky a rady

Zoznámime sa s dvoma novými jednoduchými dátovými štruktúrami, ktoré sú veľmi dôležité pri realizácii mnohých algoritmov.

Obe tieto dátové štruktúry uchovávajú nejakú skupinu údajov (tzv. kolekciu), pričom si udržujú informáciu o tom, v akom poradí boli tieto údaje vkladané do kolekcie. Pre oba typy dátovej štruktúry sa definujú operácie:

- na vkladanie ďalšieho údaja do štruktúry
- na vyberanie jedného údaja zo štruktúry
- zistenie hodnoty údaja, ešte pred jeho vybratím
- zistenie, či je kolekcia prázdna

Zrejme, ak by sme chceli vybrať údaj z prázdnej kolekcie, spôsobilo by to vyvolanie výnimky „prázdna dátová štruktúra“.

25.1 Zásobník

Dátová štruktúra zásobník (budeme používať aj anglické slovo **stack**) je charakteristická tým, že vyberané údaje prichádzajú v opačnom poradí, ako sa do zásobníka vkladali. Môžeme si to predstaviť ako umelohmotnú rúrku, ktorá je na jednom konci uzavretá a do ktorej vkladáme šumivé tablety. Vyberať ich zrejme môžeme len z vrchu, teda najskôr tú, ktorú sme vložili ako poslednú. Tomuto princípu hovoríme **LIFO** z anglického **last-in-first-out**.

Na operácie, ktoré manipulujú so zásobníkom, využijeme zaužívané anglické slová:

- **push** vloží do zásobníka nový údaj - hovoríme, že vkladáme **na vrch** zásobníka
- **pop** vyberie zo zásobníka naposledy vložený údaj - hovoríme, že vyberáme **z vrchu** zásobníka; táto hodnota je potom výsledkom operácie **pop**; ak je ale zásobník prázdny, nie je čo vybrať, operácia vyvolá výnimku `EmptyError`
- **top** vráti hodnotu z vrchu zásobníka (naposledy vkladaný údaj), ale zásobník pritom nemení; ak je ale zásobník prázdny, nie je čo vrátiť, operácia vyvolá výnimku `EmptyError`
- **is_empty** zistí, či je zásobník prázdny; operácia vráti `True` alebo `False`

Zásobník budeme v Pythone realizovať pomocou nejakého už existujúceho typu. Keďže si potrebujeme pamätať poradie vkladaných hodnôt, asi najlepšie sa bude hodiť typ zoznam (`list`). Ak si zvolíme **vrch zásobníka** ako koniec zoznamu, potom môžeme pre operácie:

- **push** použiť metódu `append()`, ktorá vkladá novú hodnotu na koniec zoznamu
- **pop** použiť metódu `pop()`, ktorá odoberá posledný prvok zoznamu a vracia jeho hodnotu, hoci budeme musieť zabezpečiť, aby sa namiesto výnimky `IndexError` pre prázdny zoznam vyvolala výnimka `EmptyError`
- **top** použiť indexovanie posledného prvku zoznamu, teda s indexom `-1`; aj tu budeme musieť zabezpečiť vyvolanie výnimky `EmptyError`
- **is_empty** len odkontrolovať, či je zoznam prázdny

Zapíšme túto realizáciu pomocou zoznamu do definície triedy `Stack`. Všimnite si, že sme pridali aj novú definíciu výnimky `EmptyError`:

```
class EmptyError(Exception): pass

class Stack:

    def __init__(self):
        '''inicializuje zoznam'''
        self._prvky = []

    def push(self, data):
        '''na vrch zasobnika vlozi novu hodnotu'''
        self._prvky.append(data)

    def pop(self):
        '''z vrchu zasobnika vyberie hodnotu, alebo vyvola EmptyError'''
        if self.is_empty():
            raise EmptyError('prazdny zasobnik')
        return self._prvky.pop()

    def top(self):
        '''z vrchu zasobnika vrati hodnotu, alebo vyvola EmptyError'''
        if self.is_empty():
            raise EmptyError('prazdny zasobnik')
        return self._prvky[-1]

    def is_empty(self):
        '''zisti, ci je zasobnik prazdny'''
        return self._prvky == []
```

V tejto definícii triedy si všimnite, že atribút `prvky` sme zapísali aj s podčiarkovníkom navyše, teda ako `_prvky`. Takýmto zápisom zvyknú pythonisti označovať, že práve tento atribút je **súkromný** a mimo metód triedy by sme s týmto atribútom nemali pracovať. Neznamená to, že sa to nedá:

```
>>> s = Stack()
>>> s.push(37)
>>> s.push('x')
>>> s._prvky
[37, 'x']
```

Hoci takto vidíme prvky *súkromného* zoznamu nejakej triedy, môžeme to využiť nanajvýš počas ladenia, ale žiadny *slušný* programátor to v reálnych programoch nepoužije. V niektorých iných programovacích jazykoch existujú zápisy, pomocou ktorých môžeme určiť, či je nejaký atribút naozaj súkromný (zvykne sa to označovať ako **private**). Vtedy sa k týmto atribútom programátor naozaj nijako nedostane, ale Python túto vlastnosť ponechal len na slušnosti a

skúsenosti programátora: dobrý programátor sa tým, že je atribút označený ako súkromný, naozaj riadi, ale ak to vo výnimočnej situácii bude chcieť použiť, nik mu v tom nebude brániť.

25.1.1 Otestujme zásobník

Do zásobníka najprv vložíme niekoľko slov a potom ich pomocou `pop()` postupe všetky vyberieme a vypíšeme. V tomto výpise by sa mali všetky prvky vyskytnúť v opačnom poradí, ako boli do zásobníka vkladané.

```
s = Stack()
for slovo in 'Anicka dusicka kde si bola'.split():
    s.push(slovo)
print('na vrchu zasobnika:', s.top())
while not s.is_empty():
    print(s.pop())
print('zasobnik je prazdny:', s.is_empty())
print('vyberame:', s.pop())
```

Na záver tohto testovacieho programu sme vložili ešte jedno zavolanie `pop()`, teda úmyselne sa pokúsime vyberať z prázdneho zásobníka. Dostávame takýto výpis:

```
na vrchu zasobnika: bola
bola
si
kde
dusicka
Anicka
zasobnik je prazdny: True
...
EmptyError: prazdny zasobnik
```

Ďalší príklad využije zásobník na otestovanie, či prvky nejakej postupnosti tvoria palindrom (dostávame rovnaké poradie prvkov, či postupnosť čítame odpredu alebo odzadu). Použijeme takýto algoritmus: najprv všetky prvky pstupnosti zapíšeme do pomocného zásobníka, potom znovu prejdeme všetky prvky postupnosti a každý porovnáme s hodnotou, ktorú vyberieme z vrchu zásobníka. Keďže v zásobníku sú prvky postupnosti uložené tak, že sa vyberajú v opačnom poradí, takéto porovnanie označuje, že porovnávame najprv prvý prvok s posledným, potom druhý prvok s predposledným atď. Zrejme, ak je postupnosť palindrom, všetky tieto testy by mali byť splnené. Program:

```
def palindrom(post):
    stack = Stack()
    for prvok in post:
        stack.push(prvok)
    for prvok in post:
        if prvok != stack.pop():
            return False
    return True
```

Otestujeme:

```
>>> palindrom([17, 'xy', 3.14, 'xy', 17])
True
>>> palindrom(['hello'])
True
>>> palindrom('hello')
False
```

Uvedomte si, že pri takomto porovnávaní prvkov, či tvoria palindrom, každú dvojicu prvkov kontrolujeme dvakrát, napr. prvý a posledný aj posledný a prvý, druhý a predposledný aj predposledný a druhý, atď. Porozmýšľajte, ako by

ste vylepšili túto funkciu, aby pomocou zásobníka testovala každú dvojicu prvkov maximálne raz.

Skôr ako prejdeme na ďalšie príklady so zásobníkom, zamyslime sa nad tým, ako môžeme organizovať definíciu novej triedy, ktorú budeme potrebovať pri riešení našej úlohy. Teda, kde treba umiestniť definíciu triedy `Stack` tak, aby sme ju mohli použiť vo funkcii palindrom. Doteraz sme v našich programoch predpokladali, že jej definícia sa v kóde nachádza niekde pred definíciou samotnej funkcie, preto to často vyzeralo takto:

```
class EmptyError(Exception): pass

class Stack:
    ...

def palindrom(post):
    ...

print(palindrom('tahat'))
```

Keďže je predpoklad, že táto naša nová dátová štruktúra je tak užitočná, že ju využijeme aj v ďalších programoch, uložíme definíciu triedy `Stack` (aj s triedou `EmptyError`) do samostatného súboru, tzv. modulu, napr. s menom `struktury.py`. Do takéhoto modulu môžeme neskôr vkladať aj ďalšie definície našich tried a aj tie potom používať v ďalších programoch. Teraz náš program môžeme zapísať takto:

```
from struktury import Stack

def palindrom(post):
    ...

print(palindrom('tahat'))
```

V tomto prípade príkaz `from struktury import Stack` označuje, že Python na tomto mieste prečíta celý súbor `struktury.py` a do nášho programu vloží definíciu triedy `Stack`. Od tohto momentu môžeme používať triedu `Stack` ako keby bola definovaná v našom súbore s programom.

Niekedy však môžu vzniknúť situácie, keď našu triedu (napr. triedu `Stack`) chceme použiť len v jednom prípade, napr. v jednej funkcii `palindrom()` (alebo len v jednej triede) a nechceme, aby táto naša pomocná trieda bola „videná“ aj mimo tejto funkcie. Vtedy môžeme definíciu triedy vnoriť napr. do funkcie alebo do inej triedy a takto vytvoríme **lokálnu definíciu triedy**. Pozrime sa, ako to môžeme zapísať:

```
def palindrom(post):

    # vnorena definicia tried EmptyError a Stack

    class EmptyError(Exception): pass

    class Stack:

        def __init__(self):
            self._prvky = []

        def push(self, data):
            self._prvky.append(data)

        def pop(self):
            if self.is_empty():
                raise EmptyError('prazdny zasobnik')
            return self._prvky.pop()

        def top(self):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            if self.is_empty():
                raise EmptyError('prazdny zasobnik')
            return self._prvky[-1]

        def is_empty(self):
            return self._prvky == []

    # koniec vnorenej definicie

    # tu pokracuje funkcia palindrom

    stack = Stack()           # pouzivanie vnorenej definicie
    for prvok in post:
        stack.push(prvok)
    for prvok in post:
        if prvok != stack.pop():
            return False
    return True
```

Toto je zrejme extrémny prípad, v ktorom vnorená definícia je príliš komplexná vzhľadom na telo funkcie, v ktorej sa používa. Neskôr uvidíme situácie, v ktorých bude takýto postup výrazným vylepšením kódu.

Podobné situácie sa dajú riešiť aj inak: namiesto vnorenej definície triedy a volaní jej príslušných metód, priamo v tele funkcie zapíšeme kód týchto metód a v komentári zaznačíme, že sa vlastne jedná o akcie príslušných metód. Funkciu `palindrom()` by sme mohli zapísať aj takto:

```
def palindrom(post):
    stack = []                    # Stack()
    for prvok in post:
        stack.append(prvok)       # push(prvok)
    for prvok in post:
        if prvok != stack.pop():  # pop()
            return False
    return True
```

Takýto zápis označuje, že čitateľ tohto programu vie, čo je to **stack** a vie, ako sa s tým pracuje, a teda bude rozumieť princípu fungovania algoritmu.

Lenže takýto spôsob používania, napr. štruktúry zásobník, **vo všeobecnosti neodporúčame**, lebo

- pri nahrádzaní volania metódy priamo kódom môžeme do kódu vložiť chyby, ktoré sa môžu ťažko odladiť
- do funkcie sme vložili kód konkrétnej realizácie zásobníka, ale v skutočnosti môžu byť tieto metódy realizované inak a možno výrazne efektívnejšie - mali by sme namiesto toho radšej používať volania metód (neskôr uvidíme aj inú realizáciu zásobníka)
- je to zlý obraz o našom spôsobe programovania a naozaj ho použime len pri rýchlom otestovaní nejakého algoritmu, ale nie v odovzdanom kóde

25.1.2 Počet prvkov v zásobníku

V ďalšom príklade napíšeme funkciu, ktorá zistí počet prvkov v zásobníku. Zrejme by sa to dalo zapísať takto jednoducho:

```
def pocet_prvkov(zasobnik):
    return len(zasobnik._prvky)
```

My už vieme, že autor tejto realizácie dátovej štruktúry `Stack` označil atribút `_prvky` so znakom podčiarkovník a teda nepredpokladá, že by sme si to niekedy dovolili použiť. Preto to vyriešme správne. Najprv nie najlepšia verzia:

```
def pocet_prvkov(zasobnik):
    pocet = 0
    while not zasobnik.is_empty():
        zasobnik.pop()
        pocet += 1
    return pocet
```

Hoci táto verzia dáva správny výsledok, popri tom nám ale vymaže celý obsah zásobníka. Takéto správanie funkcie sa nám bude hodiť veľmi zriedka. Častejšie budeme očakávať, že sa pôvodný obsah zachová. Niekoho by možno napadlo riešenie, v ktorom si zo zásobníka urobíme kópiu, zistíme koľko má prvkov táto kópia a tým sa nám pôvodný zásobník uchová. Lenže ani táto verzia nemá šancu fungovať správne:

```
def pocet_prvkov(zasobnik):
    pocet = 0
    kopia = zasobnik
    while not kopia.is_empty():
        kopia.pop()
        pocet += 1
    return pocet
```

Žiaľ, aj táto verzia funkcie vymaže pôvodný zásobník. Priradenie `kopia = zasobnik` nevytvorí naozaj kópiu, ale do premennej `kopia` priradí referenciu na pôvodný zásobník. A operácia `kopia.pop()` v skutočnosti číta pôvodnú dátovú štruktúru.

Správne riešenie bude vyžadovať, aby sme celý obsah zásobníka prečítali, niekam si ho popritom ukladali a zároveň počítali počet prvkov. Potom sa všetky prvky presunú späť do pôvodného zásobníka. V nasledovnom riešení sme ako pomocnú štruktúru na uloženie prvkov použili opäť zásobník:

```
def pocet_prvkov(zasobnik):
    pocet = 0
    kopia = Stack()                 # pomocny zasobnik
    while not zasobnik.is_empty():
        kopia.push(zasobnik.pop())
        pocet += 1
    while not kopia.is_empty():        # vrati povodny obsah zasobnika
        zasobnik.push(kopia.pop())
    return pocet
```

Uvedomte si, že v premennej `kopia` bude po prvom while-cykle kópia obsahu pôvodného zásobníka, ale prvky v ňom budú v opačnom poradí. Za tým nasledujúci while-cyklus vráti tento obsah späť ale už v správnom poradí.

25.2 Aritmetické výrazy

Aby sme videli krásu dátovej štruktúry zásobník, ukážeme si jej použitie pri práci s aritmetickými výrazmi. Pritom sa musíme zoznámiť so základnou terminológiou:

- aritmetické výrazy sa skladajú z operandov (napr. čísla a premenné ako `127` a `pocet`) a operátorov (napr. pre operácie súčet `+`, súčin `*`, ...)
- ďalej budeme uvažovať len s binárnymi operáciami, pri ktorých každému operátoru prislúchajú práve dva operandy (napr. `pocet + 1` alebo `22 % 6`)
- niektoré operátory majú pri vyhodnocovaní vyššiu prioritu (precedenciu) ako iné (napr. vo výraze `2 + 3 * 4` sa najprv vyhodnotí súčin a až potom súčet, lebo operátor `*` má vyššiu prioritu ako `+`)

  - ak by vás zaujímalo, môžete si pozrieť kompletnú tabuľku priorít operátorov v Pythone

- časti aritmetického výrazu môžeme uzavrieť do okrúhlych zátvoriek a tým zmeníme poradie vyhodnocovania výrazu (napr. vo výraze `(2 + 3) * 4` sa najprv vyhodnotí súčet a až jeho výsledok sa vynásobí operandom `4`)

Pravdepodobne ste všetky tieto pojmy už poznali predtým, alebo intuitívne s týmto pracujete už dlhšie.

25.2.1 Infixový zápis

Takýto zápis výrazov je v matematike najbežnejší a hovorí sa mu infixový zápis. Takéto pomenovanie dostal preto, lebo operátor sa nachádza medzi (teda **in**) operandami. My sa budeme zaoberať najmä s číselnými výrazmi a preto nás budú zaujímať len operácie s číslami. V nasledovnej tabuľke uvádzame skupiny priorít operátorov, pričom v jednej skupine sú operátory s rovnakou prioritou:

`( )`	zátvorky	
`**`	umocňovanie	
`* / % //`	násobenie, delenie	
`+ −`	sčitovanie, odčitovanie	

Najvyššiu prioritu majú zátvorky, potom nasleduje umocňovanie (jediné sa vyhodnocuje sprava doľava, všetky ostatné zľava doprava), nasleduje násobenie a delenie a najnižšiu prioritu majú sčitovanie a odčitovanie.

Kvôli komplikovaným pravidlám priorít operátorov a tiež zátvorkám, sa takéto aritmetické výrazy vyhodnocujú nejakým programom trochu ťažšie. Skúste sa zamyslieť, ako by ste naprogramovali funkciu, ktorá by vyhodnocovala aritmetické výrazy zadané v znakovom reťazci, napr. `'5+(71-13*4)//(5+22%7)'`. Nie je to až tak ťažké, existuje na to niekoľko elegantných algoritmov, ktoré sa budete možno učiť niekedy neskôr, ale my si ukážeme iné spôsoby zápisov aritmetických výrazov, ktoré sa vyhodnocujú výrazne jednoduchšie a práve pri nich veľmi elegantne využijeme typ **zásobník**.

25.2.2 Prefixový zápis

Prefixový zápis, niekedy sa mu hovorí poľský zápis (na počesť poľského matematika, ktorý zaviedol tento zápis), je charakteristický tým, že operátory sa nezapisujú medzi svoje operandy, ale **pred** operandy. Teda namiesto `3 * 4` zapíšeme

```
* 3 4
```

Funguje to takto aj so zložitejšími výrazmi, napr. namiesto `2 + 3 * 4` zapíšeme

```
+ 2 * 3 4
```

čo prečítame ako „sčítaj 2 a výsledok súčinu 3 a 4“. Zaujímavo vyzerajú výrazy so zátvorkami, napr. `(2 + 3) * 4` v prefixovom zápise vyzerá

```
* (+ 2 3) 4
```

a označuje „vynásob výsledok súčtu 2 a 3 číslom 4“. Lenže v prefixovom zápise sa zátvorky nepíšu a preto bez zátvoriek to vyzerá takto

```
* + 2 3 4
```

a má to úplne rovnaký význam ako predchádzajúci zápis so zátvorkami. Prefix má oproti infixu tieto dve užitočné vlastnosti:

- na vyhodnocovanie výrazu nepotrebujeme poznať prioritu operátorov - všetky operátory majú rovnakú prioritu a výraz sa vyhodnocuje zľava doprava
- zátvorkovanie výrazu nemá zmysel, lebo ten sa vyhodnocuje jednoznačne

Uveďme niekoľko príkladov:

infix	prefix	
`2 + 3 * 4 - 5`	`- + 2 * 3 4 5`	
`(2 + 3) * 4 - 5`	`- * + 2 3 4 5`	
`2 + 3 * (4 - 5)`	`+ 2 * 3 - 4 5`	
`(2 + 3) * (4 - 5)`	`* + 2 3 - 4 5`	

Všimnite si, že poradie operandov je v infixovom aj prefixovom rovnaké, tieto zápisy sa líšia len umiestnením operátorov.

Takéto prefixové zápisy sa vyhodnocujú veľmi jednoducho: každý operátor má za sebou dva operandy, pričom každý z nich je buď nejaká hodnota (napr. číslo) alebo ďalší prefixový zápis. Keď to budeme vyhodnocovať ručne, môžeme si pomôcť zátvorkami, napr. `- + 2 * 3 4 5` ozátvorkujeme takto `(- (+ 2 (* 3 4)) 5)` z čoho vidíme, že najprv sa vyhodnotí `(* 3 4)`, výsledok tohto výrazu sa dosadí do súčtu `(+ 2 12)` a hodnota tohto výrazu sa vloží do rozdielu `(- 14 5)`, teda výsledkom je `9`. Na vyhodnocovanie prefixových výrazov programom zátvorky potrebovať nebudeme - neskôr uvidíte algoritmus.

25.2.3 Postfixový zápis

Postfixový zápis niekedy sa mu hovorí prevrátený poľský zápis, má podobný princíp ako infix a prefix, len sa líši v umiestnení operátora ku svojim dvom operandom: operátor píšeme **za** svoje operandy. Uvedieme podobné ukážky, ako sme videli pri prefixe a porovnáme ich s týmto prefixom:

infix	prefix	postfix	
`3 * 4`	`* 3 4`	`3 4 *`	
`2 + 3 * 4`	`+ 2 * 3 4`	`2 3 4 * +`	
`(2 + 3) * 4`	`* + 2 3 4`	`2 3 + 4 *`	
`2 + 3 * 4 - 5`	`- + 2 * 3 4 5`	`2 3 4 * + 5 -`	
`(2 + 3) * 4 - 5`	`- * + 2 3 4 5`	`2 3 + 4 * 5 -`	
`2 + 3 * (4 - 5)`	`+ 2 * 3 - 4 5`	`2 3 4 5 - * +`	
`(2 + 3) * (4 - 5)`	`* + 2 3 - 4 5`	`2 3 + 4 5 - *`	

Všimnite si, že aj pri postfixe je zachované poradie všetkých operandov, len operátory sa presťahovali na iné pozície. V porovnaní s prefixom sú tieto operátory v opačnom poradí, len sú na iných miestach. Keď sa budete cvičiť v ručnom prevádzaní medzi týmito zápismi, prípadne budete ich ručne vyhodnocovať, niekedy pomôže ich najprv ozátvorkovať, vykonať prevod (alebo vyhodnotenie) do iného zápisu a zbytočné zátvorky vyhodiť.

Na postfixovom zápise si ukážeme algoritmus vyhodnocovania takýchto aritmetických výrazov. Princíp je nasledovný:

- postupne sa prechádza postfixový výraz zľava doprava

- keď je v postupnosti operand (najčastejšie celé číslo), vloží sa do zásobníka
- keď spracovávame operátor, z vrhu zásobníka sa vyberú dve hodnoty a tie sa stanú operandami spracovávanej operácie: príslušná operácia sa vykoná a jej výsledok sa vloží do zásobníka (namiesto dvoch práve vybraných operandov)
- keď sme spracovali celý výraz, na vrchu zásobníka ostala jediná hodnota a to je vyhodnotením aritmetického výrazu

Postup najprv ukážeme na konkrétnom príklade s výrazom `2 3 4 * + 5 -`:

zásobník	spracovávame	výraz v postfixe	čo robíme
		`2 3 4 * + 5 -`	na začiatku je zásobník prázdny
	`2`	`3 4 * + 5 -`	zoberieme prvý prvok
`2`		`3 4 * + 5 -`	vložíme ho na vrch zásobníka
`2`	`3`	`4 * + 5 -`	zoberieme ďalší prvok
`2 3`		`4 * + 5 -`	vložíme ho na vrch zásobníka
`2 3`	`4`	`* + 5 -`	zoberieme ďalší prvok
`2 3 4`		`* + 5 -`	vložíme ho na vrch zásobníka
`2 3 4`	`*`	`+ 5 -`	zoberieme ďalší prvok
`2`	`3*4`	`+ 5 -`	vykonáme operáciu s dvoma vrchnými prvkami zásobníka
`2 12`		`+ 5 -`	výsledok vložíme na vrch zásobníka
`2 12`	`+`	`5 -`	zoberieme ďalší prvok
	`2+12`	`5 -`	vykonáme operáciu s dvoma vrchnými prvkami zásobníka
`14`		`5 -`	výsledok vložíme na vrch zásobníka
`14`	`5`	`-`	zoberieme ďalší prvok
`14 5`		`-`	vložíme ho na vrch zásobníka
`14 5`	`-`		zoberieme ďalší prvok
	`14-5`		vykonáme operáciu s dvoma vrchnými prvkami zásobníka
`9`			výsledok vložíme na vrch zásobníka = výsledok výrazu

Naprogramujme:

```
def pocitaj(vyraz):
    s = Stack()
    for prvok in vyraz.split():
        if prvok == '+':
            s.push(s.pop() + s.pop())
        elif prvok == '-':
            s.push(-s.pop() + s.pop())
        elif prvok == '*':
            s.push(s.pop() * s.pop())
        elif prvok == '/':
            op2 = s.pop()              # mozeme zapisat aj: op2, op1 = s.pop(), s.pop()
            op1 = s.pop()
            s.push(op1 // op2)
        else:
            s.push(int(prvok))
    return s.pop()
```

Je veľmi dôležité si pre jednotlivé operácie uvedomiť, v akom poradí vyberáme operandy zo zásobníka. Keďže operácie súčtu a násobenia čísel sú komutatívne, nemusíme sa starať o to, v akom poradí boli operandy v zásobníku. Pri rozdiele a podiele je to už iné: tu musíme presne rozlišovať, ktorý z operandov je na vrchu zásobníka a ktorý je v

zásobníku pod ním. Keď vykonávame operáciu rozdiel `7 2 -`, označuje to, že od prvého operandu odčitujeme druhý. Keďže do zásobníka sa našim algoritmom najprv dostáva prvý operand a až potom druhý a vyberajú sa v opačnom poradí, musíme to zohľadniť aj pri výpočte, preto to zapíšeme ako `-s.pop() + s.pop()`. Teda to, čo bolo na vrchu zásobníka dostáva opačné znamienko a odčíta sa od druhého operandu v zásobníku pod ním. Podobne je to aj s podielom (v našom prípade celočíselnom), ale tu sme to kvôli čitateľnosti najprv priradili do dvoch premenných a až potom sme vykonali operáciu delenia.

Otestujme:

```
>>> pocitaj('2 3 4 * + 5 -')
9
>>> pocitaj('2 3 + 4 * 5 -')
15
>>> pocitaj('2 3 4 5 - * +')
-1
>>> pocitaj('2 3 + 4 5 - *')
-5
```

Pravdepodobne by sme mali do kódu pridať ešte niekoľko testov, aby to nepadalo s nepochopiteľnými hláškami výnimiek pri chybových situáciách. Napr.

- po skončení for-cyklu v zásobníku nie je nič, alebo je tam viac ako jedna hodnota
- delenie nulou
- namiesto očakávaného čísla (príkaz za `else`, v ktorom sa prevádza reťazec na číslo) prišiel chybný reťazec, napr. pri `22 7 %`

Program by mohol v takých situáciách buď vyvolať nejakú svoju výnimku (napr. `ExpressionError('delenie nulou')`) alebo vrátiť nejakú špeciálnu hodnotu (napr. znakový reťazec s popisom chyby). Toto budete riešiť na cvičeniach.

Funkcia `pocitaj()` funguje len pre **postfixový zápis** a pre prefix asi rýchlo spadne na chybe. Ak budeme ale v tomto algoritme vstupnú postupnosť prvkov výrazu brať v opačnom poradí od konca, po malej úprave to bude fungovať práve pre **prefixový zápis**:

```
def pocitaj_prefix(vyraz):
    s = Stack()
    for prvok in reversed(vyraz.split()):
        if prvok == '+':
            s.push(s.pop() + s.pop())
        elif prvok == '-':
            s.push(s.pop() - s.pop())
        elif prvok == '*':
            s.push(s.pop() * s.pop())
        elif prvok == '/':
            s.push(s.pop() // s.pop())
        else:
            s.push(int(prvok))
    return s.pop()
```

Všimnite si tieto dôležité zmeny oproti verzii pre postfix:

- aby sme for-cyklom prechádzali prvky vstupnej postupnosti výrazu od konca, použili sme štandardnú funkciu `reversed()` - táto funkcia bude z postupnosti prvkov dávať do for-cyklu prvky od konca; mohli sme to zapísať aj ako `vyraz.split()[::-1]` ale odporúča sa to pomocou `reversed()` lebo toto je čitateľnejšie
- pre všetky naše operácie je poradie operandov v správnom poradí, teda v zásobníku je najprv prvý operand a pod ním druhý; preto aj vyhodnocovanie operácií je teraz jednoduchšie a čitateľnejšie

Podobne, ako sme odkrokovali vyhodnocovanie postfixového zápisu, zapíšme aj postup pre prefix pre výraz `- + 2 * 3 4 5`:

zásobník	spracovávame	výraz v postfixe	čo robíme
		`- + 2 * 3 4 5`	na začiatku je zásobník prázdny
	`5`	`- + 2 * 3 4`	zoberieme posledný prvok výrazu
`5`		`- + 2 * 3 4`	vložíme ho na vrch zásobníka
`5`	`4`	`- + 2 * 3`	zoberieme ďalší prvok
`5 4`		`- + 2 * 3`	vložíme ho na vrch zásobníka
`5 4`	`3`	`- + 2 *`	zoberieme ďalší prvok
`5 4 3`		`- + 2 *`	vložíme ho na vrch zásobníka
`5 4 3`	`*`	`- + 2`	zoberieme ďalší prvok
`5`	`3*4`	`- + 2`	vykonáme operáciu s dvoma vrchnými prvkami zásobníka
`5 12`		`- + 2`	výsledok vložíme na vrch zásobníka
`5 12`	`2`	`- +`	zoberieme ďalší prvok
`5 12 2`		`- +`	vložíme ho na vrch zásobníka
`5 12 2`	`+`	`-`	zoberieme ďalší prvok
`5`	`2+12`	`-`	vykonáme operáciu s dvoma vrchnými prvkami zásobníka
`5 14`		`-`	výsledok vložíme na vrch zásobníka
`5 14`	`-`		zoberieme ďalší prvok
	`14-5`		vykonáme operáciu s dvoma vrchnými prvkami zásobníka
`9`			výsledok vložíme na vrch zásobníka = výsledok výrazu

Funkciu otestujeme na tých istých výrazoch ako predtým, ale v prefixovej verzii zápisu:

```
>>> pocitaj_prefix('- + 2 * 3 4 5')
9
>>> pocitaj_prefix('- * + 2 3 4 5')
15
>>> pocitaj_prefix('+ 2 * 3 - 4 5')
-1
>>> pocitaj_prefix('* + 2 3 - 4 5')
-5
```

Môžete vidieť, že takýto algoritmus je dostatočne jednoduchý na to, aby sa rozšíril o ďalšie operácie, napr. zvyšok po delení `%`, resp. prerobil na čísla s desatinnou čiarkou `float`.

S infixovým zápisom to nie je až tak jednoduché ako s prefixom alebo postfixom. V rámci domáceho zadania si precvičíte jeden konkrétny nie veľmi náročný algoritmus na prepis aritmetického výrazu v infixovom zápise do prefixového tvaru. Tento už potom dokážete veľmi jednoducho aj vyhodnocovať.

25.3 Náhrada rekurzie

Dátová štruktúra zásobník ma ešte jedno veľmi užitočné využitie: pomocou nej vieme niektoré **rekurzívne funkcie** relatívne jednoducho previesť na nerekurzívny výpočet. Začnime najprv triviálnou verziou rekurzie:

```
def vypis(zoz):
    if len(zoz) == 0:
        print()
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    else:
        print(zoz[0], end=' ')
        vypis(zoz[1:])
```

ktorá vypíše prvky zoznamu do jedného riadku, napr.

```
>>> vypis([2, 3, 5, 7, 11, 13, 17])
2 3 5 7 11 13 17
```

Naše doterajšie skúsenosti s rekurziou nám zrejme rýchlo naznačia, že sa jedná o chvostovú rekurziu, ktorá sa dá bezbolestne prerobiť na while-cyklus (nezabudneme použiť pomocný zoznam, aby sme nepokazili obsah parametra `zoz`, hoci to isté sa dalo urobiť aj obyčajným for-cyklom bez pomocného zoznamu):

```
def vypis(zoz):
    pom = zoz[:]
    while len(pom) != 0:
        print(pom[0], end=' ')
        pom = pom[1:]
    print()
```

Takéto nerekurzívne riešenie má oproti rekurzii obrovskú výhodu najmä v tom, že dĺžka zoznamu nie je obmedzená hĺbkou vnorenia rekurzie, čo je okolo 1000. Teda rekurzívna verzia by spadla na pretečení rekurzie už pri 1000-prvkovom zozname. Pravdepodobne podobným postupom (zatiaľ bez zásobníka) vieme nahradiť rekurziu while-cyklom aj pri trochu komplikovanejších verziách funkcie `výpis`, napr.:

```
def vypis1(zoz):
    if len(zoz) != 0:
        vypis1(zoz[1:])
        print(zoz[0], end=' ')

def vypis2(zoz):
    if len(zoz) != 0:
        print(zoz[0], end=' ')
        vypis2(zoz[1:])
        print(zoz[0], end=' ')
```

My sa ale budeme zaoberať takými rekurzívnymi funkciami, na ktoré nám takýto while-cyklus stačiť nebude. V 12. prednáške (*12. Rekurzia*) sme sa prvýkrát zoznámili s tým, že programovacie jazyky na zabezpečenie fungovania volania funkcií a teda aj rekurzie používajú **zásobník**: v Pythone si to najlepšie predstavíme tak, že na vrchu zásobníka sa pri každom volaní funkcie vytvorí menný priestor danej funkcie a ňom sa nachádzajú všetky lokálne premenné a teda aj parametre. Zopakujme si tento **mechanizmus volania funkcií**:

- zapamätá sa návratová adresa
- vytvorí sa nový menný priestor funkcie (v ňom sa vytvárajú lokálne premenné aj parametre)
- vykoná sa telo funkcie
- zruší sa menný priestor (aj so všetkými premennými)
- vykonávanie programu sa vráti na návratovú adresu

Ukážeme, ako si zjednodušíme túto predstavu volania funkcií tak, aby sme vedeli odsimulovať rekurziu len pomocou nejakých cyklov. Skúsme to predviesť na takejto jednoduchej rekurzívnej funkcii:

```
def rekurzia(n):
    if n == 0:
        print('.', end=' ')     # trivialny pripad
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    else:
        rekurzia(n - 1)
        print(n, end=' ')
        rekurzia(n - 1)

rekurzia(3)
print('koniec')
```

Po spustení:

```
. 1 . 2 . 1 . 3 . 1 . 2 . 1 . koniec
```

Menný priestor v tomto prípade tvorí jediná premenná `n`, ktorá má hodnotu 3. Takže na vrchu zásobníka sa vytvorí informácia o tejto premennej, ale okrem toho sa musí pre každé volanie funkcie zabezpečiť „zapamätanie návratovej adresy“. **Návratová adresa** je to miesto v programe, od ktorého sa bude pokračovať po úspešnom ukončení vykonania funkcie. Zaznačme do nášho ukážkového programu všetky návratové miesta:

```
def rekurzia(n):
    if n == 0:
        print('.', end=' ')     # trivialny pripad
    else:
        rekurzia(n - 1)       # <--- volanie funkcie
        # navratove miesto
        print(n, end=' ')
        rekurzia(n - 1)       # <--- volanie funkcie
        # navratove miesto

rekurzia(3)                   # <--- volanie funkcie
# navratove miesto
print('koniec')
```

Keďže samotná rekurzia zakaždým **opakuje** nejaké výpočty ale so zmenenými premennými (menným priestorom), nahrádzať ju budeme **while-cyklom** a na zapamätávanie návratových adries a menného priestoru použijeme **zásobník**. V našom jednoduchom príklade si bude treba na zásobníku vedieť zapamätať dve hodnoty: adresu a premennú `n`.

Samotnú funkciu `rekurzia()` „rozsekneme“ na časti presne tam, kde sa nachádzajú rekurzívne volania a návratové miesta:

```
def rekurzia(n):

# prva cast'
    if n == 0:
        print('.', end=' ')     # trivialny pripad
    else:
        rekurzia(n - 1)       # <--- volanie funkcie

# druha cast'
        # navratove miesto
        print(n, end=' ')
        rekurzia(n - 1)       # <--- volanie funkcie

# tretia cast'
        # navratove miesto
```

Tieto tri časti sa budú nejako opakovať pomocou nášho nového while-cyklu a pre každú z nich využijeme zásobník s návratovou adresou a premennou `n` (menným priestorom). Ktorá z týchto častí sa bude opakovať, to nám oznámi **adresa** tejto časti a aká bude vtedy hodnota premennej `n` sa tiež dozvieme z vrchu zásobníka.

V našej funkcii sa nachádzajú dve volania funkcie, ktoré musíme nahradiť novým mechanizmom. Tento zabezpečí, že sa namiesto každého volania urobí niečo so zásobníkom:

1. zapamätáme si (operácia `push()`), kam sa bude treba vrátiť a aká bude pritom hodnota `n`
2. nastavíme skok na úplný začiatok funkcie s novým `n` (zmenšeným o 1) - toto môžeme urobiť takýmto malým vylepšením: na vrch zásobníka vložíme informáciu (operácia `push()`), že chceme pokračovať od `# prva cast` s novým `n`

Teda namiesto každého volania `rekurzia(n - 1)` zapíšeme tieto dve vkladania do zásobníka:

```
stack.push((adresa, n))    # adresa je návratové miesto 2 alebo 3
stack.push((1, n - 1))     # 1 je adresa prvej časti, teda začiatku funkcie
```

Teraz môžeme celú rekurzívnu funkciu prepísať:

```
def rekurzia(n):

# inicializacia zasobnika
    stack = Stack()
    stack.push((1, n))

    while not stack.is_empty():
        adresa, n = stack.pop()

        # prva cast'
        if adresa == 1:
            if n == 0:
                print('.', end=' ')     # trivialny pripad
            else:
                #rekurzia(n - 1)       # <--- volanie funkcie
                stack.push((2, n))
                stack.push((1, n - 1))

        # druha cast'
        elif adresa == 2:
                # navratove miesto
                print(n, end=' ')
                #rekurzia(n - 1)       # <--- volanie funkcie
                stack.push((3, n))
                stack.push((1, n - 1))

        # tretia cast'
        elif adresa == 3:
                # navratove miesto
                pass
```

Po otestovaní dostávame ten istý výsledok, ako pôvodná rekurzívna verzia. Odstráňme komentáre a tiež tretiu časť, ktorú ako vidíme, nerobí nič a preto môžeme odstrániť aj `stack.push((3, n))`, ktorý spôsobuje návrat na tretiu časť po skončení druhej:

```
def rekurzia(n):
    stack = Stack()
    stack.push((1, n))
    while not stack.is_empty():
        adresa, n = stack.pop()
        if adresa == 1:
            if n == 0:
                print('.', end=' ')     # trivialny pripad
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            else:
                stack.push((2, n))
                stack.push((1, n - 1))
        elif adresa == 2:
                print(n, end=' ')
                #stack.push((3, n))
                stack.push((1, n - 1))

rekurzia(3)
print('koniec')
```

A toto je kompletná nerekurzívna verzia tejto funkcie. Takýto proces **nahrádzania rekurzie** pomocou while-cyklu sa dá zapísať veľa rôznymi spôsobmi, bude to závisieť hlavne od vašich programátorských skúseností. Pozrite si napríklad takýto prepis, ktorý robí presne to isté ako naše predchádzajúce riešenie:

```
def rekurzia(n):
    stack = Stack()
    adresa = 1
    while True:
        if adresa == 1:
            if n == 0:
                print('.', end=' ')     # trivialny pripad
                if stack.is_empty():
                    break               # alebo return
                adresa, n = stack.pop()
            else:
                stack.push((2, n))
                adresa, n = 1, n - 1
        elif adresa == 2:
                print(n, end=' ')
                #stack.push((3, n))
                adresa, n = 1, n - 1
```

Rekurzívne funkcie, ktoré vracajú nejakú hodnotu, je „odrekurzívňovať“ trochu náročnejšie. Nám by mohlo postačiť také riešenie, v ktorom najprv túto rekurzívnu funkciu prepíšeme na funkciu, ktorá nevracia hodnotu, ale mení nejakú globálnu premennú. Takúto zmenenú rekurzívnu funkciu by sme už mali vedieť nahradiť while-cyklom.

Rekurziu sme doteraz najviac využívali v korytnačej grafike. Pripomeňme si jednu z najznámejších rekurzívnych funkcií s korytnačou grafikou - kreslenie binárneho stromu:

```
import turtle

def strom(n, d):
    t.fd(d)
    if n > 0:
        t.lt(40)
        strom(n - 1, d * 0.7)
        t.rt(75)
        strom(n - 1, d * .6)
        t.lt(35)
    t.bk(d)

t = turtle.Turtle()
t.lt(90)
strom(5, 80)
```

Na jej odrekurzívnenie použijeme rovnaký postup, ako pri funkcii `rekurzia()`. „Rozsekneme“ funkciu na časti v

tých miestach, kde sa nachádza rekurzívne volanie. Musíme si pritom dať veľký pozor na triviálny prípad, teda čo sa naozaj vykoná v triviálnom prípade (teda, keď neplatí `n > 0`):

```
def strom(n, d):

# prva cast
    t.fd(d)
    if n <= 0:
        t.bk(d)          # trivialny pripad
    else:
        t.lt(40)
        strom(n - 1, d * 0.7)

# druha cast
        t.rt(75)
        strom(n - 1, d * .6)

# tretia cast
        t.lt(35)
        t.bk(d)
```

Zatiaľ je to rekurzívne, ale pripravené na nahradenie while-cyklom so zásobníkom:

```
def strom(n, d):
    stack = Stack()
    stack.push((1, n, d))
    while not stack.is_empty():
        adresa, n, d = stack.pop()
    # prva cast
        if adresa == 1:
            t.fd(d)
            if n <= 0:
                t.bk(d)          # trivialny pripad
            else:
                t.lt(40)
                #strom(n - 1, d * 0.7)
                stack.push((2, n, d))
                stack.push((1, n - 1, d * 0.7))

    # druha cast
        elif adresa == 2:
            t.rt(75)
            #strom(n - 1, d * .6)
            stack.push((3, n, d))
            stack.push((1, n - 1, d * 0.7))

    # tretia cast
        elif adresa == 3:
            t.lt(35)
            t.bk(d)
```

Dostali sme nerekurzívnu verziu kreslenia binárneho stromu. Bude veľmi užitočné, keď si potrénujete takéto nahrádzanie rekurzie, lebo neskoršie témy v tomto semestri budú plné rekurzívnych funkcií, ktoré bude niekedy treba vykonať aj pre veľmi veľký počet rekurzívnych vnorení.

25.4 Rad

Dátová štruktúra **rad** (budeme používať aj **front** alebo anglické **queue**, ale nie *rada* ani *fronta*) je veľmi podobná dátovej štruktúre zásobník. Na rozdiel od zásobníka ale nevyberá prvky z vrchu (od konca, teda naposledy vložený prvok) ale zo začiatku štruktúry. Tomuto princípu hovoríme **FIFO** z anglického **first-in-first-out**. Na tomto princípe funguje aj náš bežný rad v obchode alebo v jedálni. Keď sa v takomto rade nepredbiehame a obslúžený sme presne v tom poradí, ako sme prišli, hovoríme tomu **spravodlivý rad** (na rozdiel od zásobníka, ktorý je nespravodlivý rad). Počas tohto semestra uvidíme niekoľko veľmi užitočných aplikácií tejto dátovej štruktúry.

Opäť na operácie, ktoré manipulujú s radom, využijeme zaužívané anglické slová:

- **enqueue** vloží na koniec radu nový údaj
- **dequeue** vyberie prvý údaj z radu; táto hodnota je výsledkom operácie; ak je ale rad prázdny, nie je čo vybrať, operácia vyvolá výnimku `EmptyError`
- **front** vráti prvý prvok radu, ale rad pritom nemení; ak je ale rad prázdny, nie je čo vrátiť, operácia vyvolá výnimku `EmptyError`
- **is_empty** zistí, či je rad prázdny; operácia vráti `True` alebo `False`

Rad budeme v Pythone realizovať podobne ako zásobník pomocou typu zoznamu (`list`). Začiatok radu bude zodpovedať prvému prvku zoznamu (prvky[0]) a vkladať budeme na koniec zoznamu, t.j. operáciou `append()`. Túto realizáciu pomocou zoznamu zapíšeme do triedy `Queue` a opäť pridáme aj definíciu výnimky `EmptyError`:

```
class EmptyError(Exception): pass

class Queue:

    def __init__(self):
        '''inicializuje zoznam'''
        self._prvky = []

    def enqueue(self, data):
        '''na koniec radu vlozi novu hodnotu'''
        self._prvky.append(data)

    def dequeue(self):
        '''zo zaciatku radu vyberie prvu hodnotu, alebo vyvola EmptyError'''
        if self.is_empty():
            raise EmptyError('prazdny rad')
        return self._prvky.pop(0)

    def front(self):
        '''zo zaciatku radu vrati prvu hodnotu, alebo vyvola EmptyError'''
        if self.is_empty():
            raise EmptyError('prazdny rad')
        return self._prvky[0]

    def is_empty(self):
        '''zisti, ci je rad prazdny'''
        return self._prvky == []
```

Ak porovnáte túto realizáciu s definíciou triedy `Stack`, zistíte, že sú medzi nimi minimálne rozdiely. Uvidíte neskôr, že využitie týchto dvoch dátových štruktúr je veľmi rozdielne a často sa využívajú na úplne iné ciele.

Podobne ako sme definíciu triedy `Stack` uložili do súboru `struktury.py`, môžeme sem prekopírovať aj túto definíciu radu. Zrejme triedu `EmptyError` vtedy nemá zmysel kopírovať druhý krát - jej jedna definícia pokryje obe tieto štruktúry.

25.4.1 Otestujme rad

Otestujeme rovnakým testom, ako sme to robili so zásobníkom: do radu najprv vložíme niekoľko slov a potom ich pomocou `dequeue()` postupe všetky vyberieme a vypíšeme. V tomto výpise by sa mali všetky prvky vyskytnúť presne v poradí, ako boli vkladané do radu.

```
from struktury import Queue

queue = Queue()
for slovo in 'Anicka dusicka kde si bola'.split():
    queue.enqueue(slovo)
print('prvy v rade:', queue.front())
while not queue.is_empty():
    print(queue.dequeue())
print('rad je prazdny:', queue.is_empty())
print('vyberame:', queue.dequeue())
```

Po spustení vidíme, že to naozaj funguje:

```
prvy v rade: Anicka
Anicka
dusicka
kde
si
bola
rad je prazdny: True
...
struktury.EmptyError: prazdny rad
```

Ďalší príklad prečíta textový súbor a na jeho koniec vloží celú kópiu samého seba:

```
from struktury import Queue

def zdvoj_subor(meno_suboru):
    queue = Queue()
    with open(meno_suboru) as subor:
        for riadok in subor:
            queue.enqueue(riadok)
    with open(meno_suboru, 'a') as subor:
        while not queue.is_empty():
            subor.write(queue.dequeue())

zdvoj_subor('text.txt')
```

Toto isté by sme vedeli zapísať aj bez použitia radu (napríklad najprv celý súbor prečítať pomocou jediného `subor.read()` a potom ho jedným `subor.write()` celý zapísať), ale príklad ilustruje použitie dátovej štruktúry `Queue`.

S niektorými operáciami s dátovou štruktúrou rad sa budeme trápiť podobne, ako pri zásobníkoch. Napr. zisťovať počet prvkov v rade, hodnotu posledného prvku a pod. bez toho, aby sme rad poškodili, bude náročnejšie, ale potrénujeme to na cvičeniach.

25.5 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/
- pozrite si *Riešenie 25. cvičenia*

1. Zistite, čo by sa vypísalo.
   - najprv bez počítača a potom skontrolujte:

```
>>> s = Stack()
>>> s.push(7); s.push(10); s.push(13); s.pop(); s.push(9); s.pop(); s.pop()
>>> s.push(2); s.push(5); s.pop(); s.pop(); s.pop(); s.pop()
```

2. Zistite, čo by sa vypísalo.
   - najprv bez počítača a potom skontrolujte:

```
s = Stack()
for i in range(10):
    if i % 3 == 2:
        print(s.pop(), end=' ')
    else:
        s.push(i)
print()
while not s.is_empty():
    print(s.pop(), end=' ')
```

3. Zistite, čo by sa vypísalo.
   - najprv bez počítača a potom skontrolujte:

```
s = Stack()
for i in 'python':
    try:
        a = s.pop()
        s.push(i)
        s.push(a)
    except EmptyError:
        s.push(i)
while not s.is_empty():
    print(s.pop(), end=' ')
```

4. Vytvorte súbor `struktury.py` a prekopírujte do neho definíciu triedy `Stack` z prednášky. V ďalších úlohách budete používať `import` z tohto súboru. Do inicializácie triedy `Stack` ešte pridajte nepovinný parameter `seq`, ktorý označuje postupnosť hodnôt, ktorou sa inicializuje zásobník (použite metódu `push()`). Pridajte ešte novú metódu `get_list()`, ktorá pomocou volania `pop()` vráti zoznam všetkých prvkov zasobníka (zásobník sa pritom vyprázni).
   - dopíšte:

```
class Stack:
    def __init__(self, seq=None):
        ...
    ...
    def get_list(self):
        return ...
```

   - otestujte:

```
>>> from struktury import Stack
>>> Stack(range(5)).get_list()
[4, 3, 2, 1, 0]
```

5. Napíšte funkciu `dno(stack)`, ktorá z daného zásobníka vyberie a vráti prvok z dna tohto zásobníka. Zvyšné prvky v ňom zostanú nezmenené. Použite pomocný zásobník.
   - napr.

```
>>> s = Stack(range(1, 6))
>>> dno(s)
1
>>> s.get_list()
[5, 4, 3, 2]
```

6. V prednáške bola funkcia `palindrom()`, ktorá pomocou zásobníka kontrolovala, či je daná postupnosť palindrom. Lenže pritom každú dvojicu prvkov zoznamu kontrolovala dvakrát (prvý prvok s posledným, druhý s predposledným, ..., predposledný s druhým, posledný s prvým). Zrejme by na zisťovanie palindromu stačila polovica testov. Opravte túto funkciu tak, aby sa využil zásobník a pritom bolo vo funkcii maximálne polovica porovnaní. Vtedy aj do zásobníka stačí vložiť len polovicu prvkov zoznamu.
   - funkcia z prednášky:

```
def palindrom(post):
    stack = Stack()
    for prvok in post:
        stack.push(prvok)
    for prvok in post:
        if prvok != stack.pop():
            return False
    return True
```

7. Ručne prepíšte infixové zápisy do prefixu aj postfixu:
   - infix:

```
7 * 6 * 5 * 4 * 3 * 2
(1 + 2) * 3 / (4 + 5) * 6
```

8. Ručne vyhodnoťte tieto zápisy
   - prefix aj postfix:

```
+ 8 * / - 14 6 3 - 8 * 2 3
1 2 3 * + 4 5 * + 6 7 * +
```

   - svoje výsledky porovnajte s vyhodnotením pomocou `pocitaj()` a `pocitaj_prefix()`
9. Opravte funkciu `pocitaj()` z prednášky, ktorá vyhodnocuje postfix tak, aby správne reagovala na chyby. Na každú chybovú situáciu sa vyvolá výnimka `ExpressionError` s príslušným komentárom, napr.
   - delenie nulou
   - očakávalo sa celé číslo
   - málo operandov pre operáciu
   - málo operátorov pre toľko operandov
10. Odrekurzívnite rekurzívnu krivku Sierpiňského trojuholník z 12. prednášky v zimnom semestri.

- rekurzívna funkcia:

```
import turtle

def trojuholniky(n, a):
    if n > 0:
        for i in range(3):
            t.fd(a)
            t.rt(120)
            trojuholniky(n - 1, a / 2)

t = turtle.Turtle()
trojuholniky(4, 200)
```

11. Do súboru `struktury.py` pridajte aj definíciu triedy `Queue` z prednášky. Naprogramujte aj tieto dve funkcie s parametrom typu `Queue` tak, aby sa nepoškodil obsah radu:

    - `pocet(rad)` zistí počet prvkov v rade
    - `posledny(rad)` vráti hodnotu posledného prvku radu (posledne pridaného prvku)

    Pre obe funkcie to riešte najprv s pomocným radom (podobne, ako sme to riešili so zásobníkom) a potom aj bez pomocného radu (resp. inej štruktúry). Vyberané prvky totiž nemusíte ukladať niekam inam a potom ich vracať späť, ale ukladáte ich na koniec samotného radu.

12. Naprogramujte dve funkcie `otoc_zasobnik(stack)` a `otoc_rad(queue)`, ktoré obe otočia poradie svojej vstupnej dátovej štruktúry

    - na otáčanie zásobníka použite pomocný rad a naopak na otáčanie radu použite pomocný zásobník

13. Do inicializácie triedy `Stack` okrem nepovinnej vstupnej postupnosti pridajte aj parameter `maxlen`, ktorý obmedzí dĺžku, pri prekročení ktorej (pri volaní `push()`) sa najstarši prvok vyhodí - takýto zásobník bude mať maximálnu dĺžku `maxlen`. Ak má `maxlen` hodnotu `None`, zásobník pracuje normálne bez obmedzenia.

    - dopíšte:

```
class Stack:
    def __init__(self, seq=None, maxlen=None):
        ...
    ...
```

25.6 1. Domáce zadanie

L.I.S.T.

- riešenie odovzdávajte na úlohový server https://list.fmph.uniba.sk/

Napíšte modul s menom `riesenie1.py`, ktorý bude obsahovať jedinú triedu s ďalšou vnorenou podtriedou a týmito metódami:

```
class Vyraz:
    class Stack:
        def __init__(self):
            ...
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        def push(self, data):
            ...

        def pop(self):
            return None

        def top(self):
            return None

        def is_empty(self):
            return True

    def __init__(self):
        self.tabulka = {}

    def __repr__(self):
        return ''

    def prirad(self, premenna, vyraz):
        ...

    def vyhodnot(self, vyraz):
        return None

    def in2post(self, vyraz):
        return None
```

Cieľom týchto metód je udržiavať tabuľku premenných a ich hodnôt (v atribúte `self.tabulka`)a tiež vyhodnocovať aritmetické výrazy, ktoré obsahujú aj premenné. Tieto aritmetické výrazy môžu byť nielen v bežnom infixovom tvare, ale aj postfixovom alebo prefixovom. Súčasťou riešenia bude aj prevod z infixu do postfixu (metóda `in2post()`). Samotný zásobnik `Stack` realizujte ako **vnorený** v triede `Vyraz` (napr. v metóde `vyhodnot()` budete vytvárať nový zásobník pomocou `self.Stack()`). Ešte ho musíte mierne zmodifikovať tak, aby pri chybe nespôsobil výnimku, ale vrátil `None`.

Metódy triedy `Vyraz` by mali mať túto funkčnosť (môžete si dodefinovať aj ďalšie pomocné metódy):

- `prirad(premenna, vyraz)` do premennej priradí **hodnotu zadaného výrazu**, priradenie znamená, že sa do tabuľky (slovník, t.j. asociatívne pole `self.tabulka`) pre zadaný reťazec (meno premennej) priradí vypočítaná hodnota; oba parametre sú znakové reťazce, napr.

  ```
  >>> v = Vyraz()
  >>> v.prirad('a','123')
  >>> v.prirad('abc','a+1')
  >>> v.prirad('a','abc 4 /')
  >>> v.prirad('res22','/ 5 0')
  ```

  uvedomte si, že hodnotou výrazu je buď celé číslo alebo `None`

- `__repr__()` vráti reťazec, ktorý obsahuje momentálne obsahy premenných (funkcia nič nevypisuje), predchádzajúce priradenia vytvoria tieto 3 premenné:

  ```
  >>> v
  a = 31
  abc = 124
  res22 = None
  ```

- `vyhodnot(vyraz)`, kde `vyraz` je znakový reťazec, ktorý obsahuje aritmetický výraz v infixovom, postfixovom, alebo prefixovom zápise; funkcia tento výraz vyhodnotí a vráti celočíselnú hodnotu výsledku alebo `None`;

funkcia by mala:

  - rozlíšiť, o ktorý typ zápisu sa jedná, stačí jednoduchý test: ak je prvý znak reťazca operátor (jeden z `'+-*/%'`), jedná sa o **prefixový** zápis, ak je posledný znak reťazca operátor, bude to **postfixový** zápis, inak je to **infixivý** zápis
  - v prípade infixu, prevedie reťazec na postfix (metódou `in2post()`)
  - vyhodnotí postfixový, resp. prefixový výraz algoritmom z prednášky, pričom využije pomocný zásobník (trieda `self.Stack`)
  - pri vyhodnocovaní výrazu premenné (identifikátory) nahradí ich hodnotou z tabuľky premenných (`self.tabulka`)
  - výrazy sú celočíselné, teda všetky operácie vracajú celé číslo (operácia `'/'` zodpovedá pythonovskému `'//'`, operácia `'%'` počíta zvyšok po delení); hoci medzi operátormi a operandami nemusia byť medzery, funkcia aj tak zabezpečí správne rozdelenie výrazu na operátory a operandy (celé čísla a identifikátory premenných)
  - ak pri vyhodnocovaní vznikne nejaká chyba, napr. delenie nulou, neznámy obsah premennej, chybajúci operand alebo operátor vo výraze, atď. funkcia vráti `None`

- `in2post(vyraz)`, kde `vyraz` je znakový reťazec, ktorý obsahuje aritmetický výraz v infixovom tvare; funkcia prevedie tento výraz na postfixový zápis; tento zápis vráti ako znakový režazec; môžete použiť tento algoritmus:
  1. rozdelí reťazec na operátory, zátvorky, celočíselné premenné a identifikátory premenných, pripraví si pomocný **zásobník** (budú sa do neho vkladať operácie a zátvorky) a zatiaľ prázdny **výstup**
  2. postupne prechádza prvky vstupu zľava doprava
  3. ak je prvkom hodnota (celé číslo alebo premenná), pridá sa na výstup
  4. ak je prvkom operátor (jeden z `'+-*/%'`)
     - ak je zásobník prázdny, operátor sa dá na vrch zásobníka (`push()`)
     - ak zásobník nie je prázdny, postupne z neho vyberá (`pop()`) všetky operátory s vyššou alebo rovnakou prioritou (a tie sa pridávajú na výstup) a až potom sa vloží (`push()`) samotný operátor (ak je na zásobníku zátvorka, táto sa teraz zo zásobnika nevyberá)
  5. ak je prvkom ľavá zátvorka `'('`, vloží (`push()`) sa do zásobníka
  6. ak je prvkom pravá zátvorka `')'`, vyberú sa (`pop()`) všetky prvky, až kým nepríde `'('` - tieto prvky (okrem zátvorky) sa pridajú na výstup
  7. keď sa už takto spracoval celý vstup, všetky prvky zásobníka sa vyberú (`pop()`) a pridajú na výstup
  8. operátory `'+', '-'` majú nižšiu prioritu ako `'*', '/', '%'`
  9. ak vo vstupnom výraze nezodpovedajú zátvorky `'('` a `')'`, funkcia vráti prázdny reťazec

Prevod z **infixu** do **postfixu** môžete vidieť na prevode tohto výrazu `'2+(44+a3*222+1)/pocet'`. Tento výraz najprv rozdelíte na prvky: `'2','+','(','44','+','a3','*','222','+','1',')','/','pocet'` a potom spúšťame algoritmus:

výstup	zásobník	spracováva	prvky vstupu	komentár
			`2 + ( 44 + a3 * 222 + 1 ) / pocet`	na začiatku
		`2`	`+ ( 44 + a3 * 222 + 1 ) / pocet`	prvý prvok
`2`			`+ ( 44 + a3 * 222 + 1 ) / pocet`	-> výstup
`2`		`+`	`( 44 + a3 * 222 + 1 ) / pocet`	ďalší prvok
`2`	`+`		`( 44 + a3 * 222 + 1 ) / pocet`	-> zásobník
`2`	`+`	`(`	`44 + a3 * 222 + 1 ) / pocet`	ďalší prvok
`2`	`+ (`		`44 + a3 * 222 + 1 ) / pocet`	-> zásobník
`2`	`+ (`	`44`	`+ a3 * 222 + 1 ) / pocet`	ďalší prvok
`2 44`	`+ (`		`+ a3 * 222 + 1 ) / pocet`	-> výstup
`2 44`	`+ (`	`+`	`a3 * 222 + 1 ) / pocet`	ďalší prvok
`2 44`	`+ ( +`		`a3 * 222 + 1 ) / pocet`	-> zásobník
`2 44`	`+ ( +`	`a3`	`* 222 + 1 ) / pocet`	ďalší prvok
`2 44 a3`	`+ ( +`		`* 222 + 1 ) / pocet`	-> výstup
`2 44 a3`	`+ ( +`	`*`	`222 + 1 ) / pocet`	ďalší prvok
`2 44 a3`	`+ ( + *`		`222 + 1 ) / pocet`	-> zásobník
`2 44 a3`	`+ ( + *`	`222`	`+ 1 ) / pocet`	ďalší prvok
`2 44 a3 222`	`+ ( + *`		`+ 1 ) / pocet`	-> výstup
`2 44 a3 222`	`+ ( + *`	`+`	`1 ) / pocet`	ďalší prvok
`2 44 a3 222 * +`	`+ (`	`+`	`1 ) / pocet`	zasobník -> výstup
`2 44 a3 222 * +`	`+ ( +`		`1 ) / pocet`	-> zásobník
`2 44 a3 222 * +`	`+ ( +`	`1`	`) / pocet`	ďalší prvok
`2 44 a3 222 * + 1`	`+ ( +`		`) / pocet`	-> výstup
`2 44 a3 222 * + 1`	`+ ( +`	`)`	`/ pocet`	ďalší prvok
`2 44 a3 222 * + 1 +`	`+`		`/ pocet`	zasobník -> výstup
`2 44 a3 222 * + 1 +`	`+`	`/`	`pocet`	ďalší prvok
`2 44 a3 222 * + 1 +`	`+ /`		`pocet`	-> zásobník
`2 44 a3 222 * + 1 +`	`+ /`	`pocet`		ďalší prvok
`2 44 a3 222 * + 1 + pocet`	`+ /`			-> výstup
`2 44 a3 222 * + 1 + pocet / +`				zasobník -> výstup

Na výstupe je teraz **postfixový** zápis výrazu.

25.6.1 Obmedzenia

- vaše riešenie odovzdajte v súbore `riesenie1.py`, pričom sa v ňom bude nachádzať **len jedna definícia triedy** `Vyraz`, trieda `Stack` bude vnorená v triede `Vyraz`
- prvé dva riadky tohto súboru budú obsahovať:

```
# autor: Janko Hrasko
# uloha: 1. domace zadanie Vyraz
```

- zrejme ako autora uvediete svoje meno
- atribút `tabulka` v triede `Vyraz` bude obsahovať slovník (asociatívne pole) so všetkými premennými a ich hodnotami (hodnoty premenných sú buď celé čísla alebo `None`)
- váš program by nemal počas testovania testovačom nič vypisovať (žiadne vaše testovacie `print()`)

25.6.2 Testovanie

Keď budete spúšťať vaše riešenie na svojom počítači, môžete do súboru `riesenie1.py` pridať testovacie riadky, ktoré ale testovač vidieť nebude, napr.:

```
if __name__ == '__main__':
    v = Vyraz()
    print(v.in2post('2+(44+a3*222+1)/pocet'))
    v.prirad('x','13')
    print(v.vyhodnot('x%5'))
    print(v.vyhodnot('x 5%'))
    print(v.vyhodnot('%x 5'))
    print(v.vyhodnot('x 5'))
    v.prirad('a','123')
    v.prirad('abc','a+1')
    v.prirad('a','abc 4 /')
    v.prirad('res22','/ 5 0')
    print(v)
```

Tento test by vám mal vypísať:

```
2 44 a3 222 * + 1 + pocet / +
3
3
3
None
res22 = None
x = 13
a = 31
abc = 124
```

KAPITOLA 26

26. Spájané štruktúry

Doteraz sme pracovali s „predpripravenými“ dátovými štruktúrami jazyka Python (zjednodušene hovoríme, že štruktúra je typ, ktorý môže obsahovať viac prvkov), napr.

- `list` - zoznam (postupnosť) hodnôt, ktoré sú očíslované indexmi od 0 do počet prvkov-1
- `dict` - slovník (asociatívne pole), kde každému prvku zodpovedá kľúč
- `set` - množina rôznych hodnôt

Okrem toho už viete konštruovať vlastné dátové typy pomocou definovania tried. Inštancie tried obsahujú atribúty, ktoré sú buď stavovými premennými alebo metódami (napr. `Stack` alebo `Queue`). Takto vieme vytvárať vlastné typy, pričom využívame štruktúry Pythonu.

Referencie

Už vieme, že premenné v Pythone (aj atribúty objektov) sú vlastne pomenované **referencie** na nejaké hodnoty, napr. čísla, reťazce, zoznamy, funkcie, atď. Referencie (kreslili sme ich šípkou) sú niečo ako adresou do pamäte, kde sa príslušná hodnota nachádza. Takýmito referenciami sú aj prvky iných štruktúrovaných typov, napr.

- zoznam čísel `[1, 2, 5, 2]` je v skutočnosti štvorprvkový zoznam referencií na hodnoty `1`, `2`, `5` (posledný prvok obsahuje rovnakú referenciu ako druhý prvok zoznamu);
- slovník (asociatívne pole) uchováva dvojice (kľúč, hodnota) a každá z nich je opäť referenciou na príslušné hodnoty;
- množina hodnôt sa tiež uchováva ako množina referencií na hodnoty
- atribúty tried, ktoré sú stavové premenné obsahujú tiež „iba“ referencie na hodnoty.

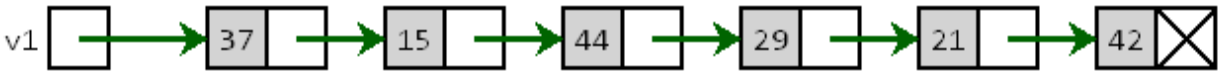
Naučíte sa využívať referencie (adresy rôznych hodnôt, teda adresy objektov) na vytváranie **spájaných štruktúr**.

26.1 Spájaný zoznam

Jednosmerný spájaný zoznam (singly linked list) je najjednoduchšia spájaná štruktúra, v ktorej každý prvok obsahuje referenciu (adresu, niekedy hovoríme aj *link*, *smerník*, *pointer*) na nasledovný prvok štruktúry. Prvkom štruktúry hovoríme **Vrchol** (alebo niekedy po anglicky **Node**). Spájané štruktúry budú vždy prístupné pomocou premennej, ktorá odkazuje (obsahuje referenciu) na prvý prvok (vrchol) zoznamu. Spájaný zoznam reprezentuje postupnosť nejakých hodnôt a v tejto postupnosti je jeden vrchol posledný, za ktorým už nie je nasledovný prvok. Tento jeden vrchol si teda namiesto nasledovníka bude pamätať informáciu, že nasledovník už nie je - najčastejšie na to využijeme hodnotu `None`.

Takúto štruktúru budeme kresliť takto:

- jedna premenná odkazuje (obsahuje referenciu) na prvý prvok (vrchol) zoznamu
- každý vrchol nakreslíme ako obdĺžnik s dvomi priečinkami: časť s údajom (pomenujeme to ako `data`) a časť s referenciou na nasledovníka (pomenujeme ako `next`)
- referencie budeme kresliť šípkami, pričom neexistujúcu referenciu (pre nasledovníka posledného vrcholu), t.j. hodnotu `None` môžeme značiť prečiarknutím políčka `next`



26.1.1 Reprezentácia vrcholu

Vrchol budeme definovať ako objekt s dvoma premennými `data` a `next`:

```
class Vrchol:
    def __init__(self, data, next=None):
        self.data = data
        self.next = next
```

Pozrite, čo sa definuje týmto zápisom:

```
>>> v1 = Vrchol(11)
```

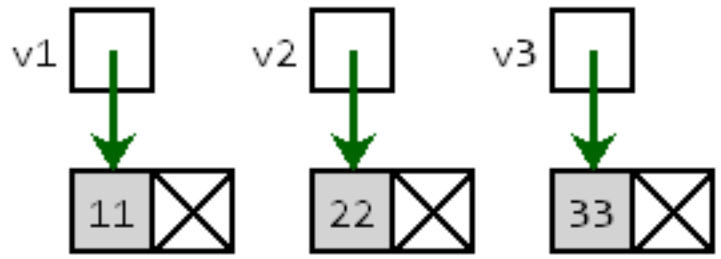
Vytvorili sme jeden vrchol, resp. jednovrcholový spájaný zoznam. Jeho nasledovníkom je `None`. Na tento vrchol odkazuje premenná `v1`. Jedine, čo zatiaľ môžeme s takýmto vrcholom robiť je to, že si vypíšeme jeho atribúty:

```
>>> print(v1.data)
11
>>> print(v1.next)
None
```

Podobne zapíšeme:

```
>>> v2 = Vrchol(22)
>>> v3 = Vrchol(33)
```

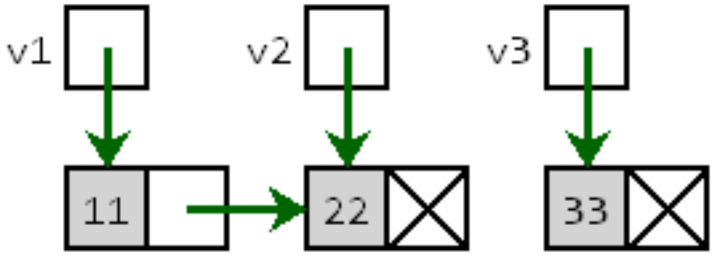
Teraz máme vytvorené 3 izolované vrcholy, ktoré sú prístupné pomocou troch premenných `v1`, `v2` a `v3`. Tieto tri vrcholy sa nachádzajú v rôznych častiach pamäti a nie sú nijako prepojené.



Vytvorme prvé prepojenie: ako nasledovníka `v1` nastavíme vrchol `v2`:

```
>>> v1.next = v2
>>> print(v1.next)
<__main__.Vrchol object at 0x01FAF0D0>
```

Vidíte, že nasledovníkom `v1` už nie je `None` ale nejaký objekt typu `Vrchol` - zrejme je to vrchol `v2`.



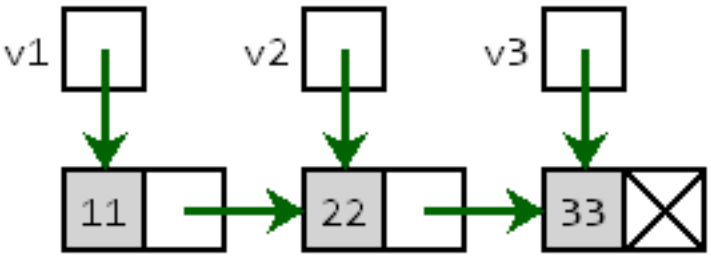
Môžete sa pozrieť na údaj nasledovníka `v1`:

```
>>> print(v1.next.data)
22
>>> print(v1.next.next)
None
```

Keďže jeho nasledovníkom je `None`, znamená to, že nasledovníka nemá. Premenná `v1` obsahuje referenciu na vrchol, ktorý má jedného nasledovníka, t.j. `v1` odkazuje na dvojprvkový spájaný zoznam. Pripojme teraz do tejto postupnosti aj vrchol `v3`:

```
>>> v2.next = v3
```

Takže prvým vrcholom v spájanom zozname je `v1` s hodnotou `11`, jeho nasledovníkom je `v2` s hodnotou `22` a nasledovníkom `v2` je `v3` s hodnotou `33`. Nasledovníkom tretieho vrcholu je stále `None`, teda tento vrchol nemá nasledovníka.



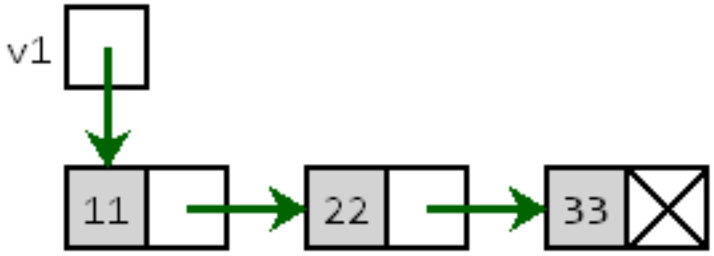
Vytvorili sme trojprvkový spájaný zoznam, ktorého vrcholy majú postupne tieto hodnoty:

```
>>> print(v1.data)
11
>>> print(v1.next.data)
22
>>> print(v1.next.next.data)
33
```

Vidíte, že pomocou referencie na prvý vrchol sa vieme dostať ku každému vrcholu, len treba dostatočný počet krát zapísať `next`. Premenné `v2` a `v3` teraz už nepotrebujete a mohli by ste ich hoci aj zrušiť, na vytvorený zoznam to už nemá žiaden vplyv:

```
>>> del v2, v3
```

Teraz to v pamäti vyzerá takto:



Webová stránka pythontutor.com

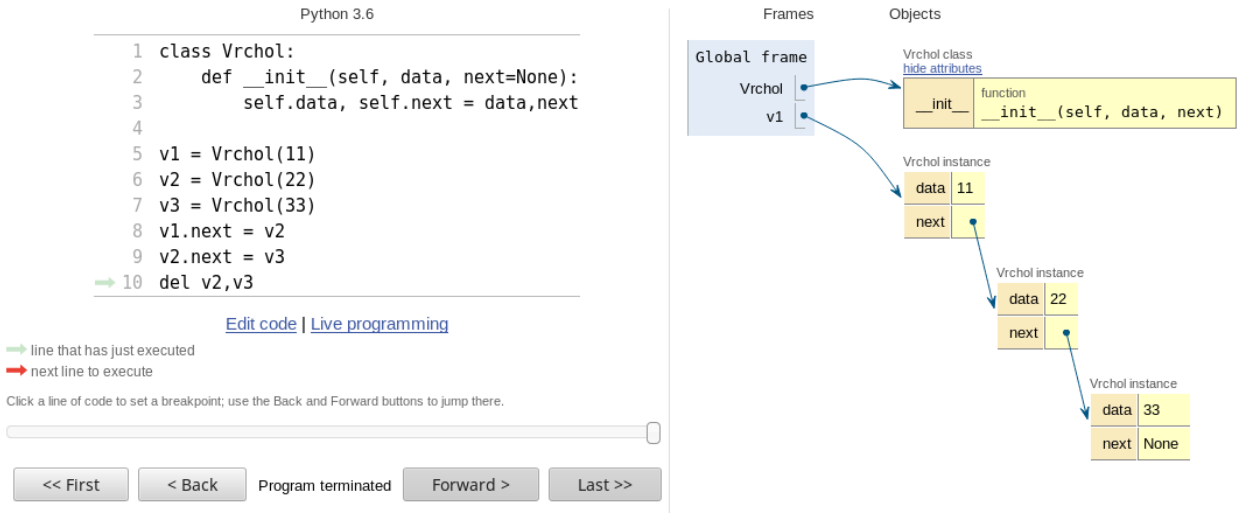
Zo zimného semestra poznáme http://pythontutor.com/visualize.html - veľmi užitočný nástroj na vizualizáciu pythonovských programov. Vieme, že sem môžeme preniesť skoro ľubovoľný algoritmus, ktorý sme robili doteraz (okrem grafiky) a odkrokovať ho. Môžete sem preniesť napr. tento program:

```
class Vrchol:
    def __init__(self, data, next=None):
        self.data, self.next = data, next

v1 = Vrchol(11)
v2 = Vrchol(22)
v3 = Vrchol(33)
v1.next = v2
v2.next = v3
del v2,v3
```

Po spustení vizualizácie môžete vidieť, že globálna premenná `v1` obsahuje referenciu na inštanciu triedy `Vrchol`, v ktorej atribút `data` má hodnotu 11 a atribút `next` je opäť referenciou na ďalšiu inštanciu triedy `Vrchol`, atď.

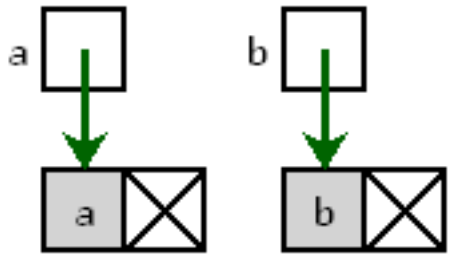
Tiež tu môžete vidieť, že globálna premenná `Vrchol` obsahuje referenciu na definíciu triedy `Vrchol`.



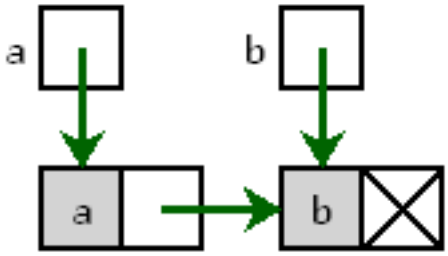
Spájané vytváranie vrcholov

Pozrite tento zápis:

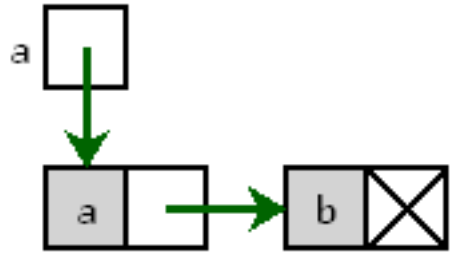
```
>>> a = Vrchol('a')
>>> b = Vrchol('b')
```




```
>>> a.next = b
```



```
>>> del b
```



Vytvorí dvojprvkový zoznam, pričom premenná `b` je len pomocná a hneď po priradení do `a.next` sa aj zruší. To isté môžete zapísať aj bez nej:

```
>>> a = Vrchol('a')
>>> a.next = Vrchol('b')
```

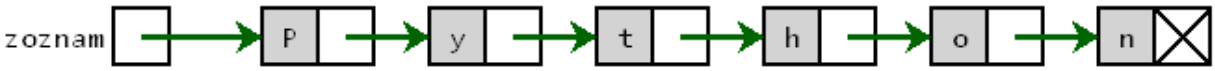
Tu si všimnite, že inicializačná metóda (`Vrchol.__init__()`) má druhý parameter, ktorým môžete definovať hodnotu `next` už pri vytváraní vrcholu. Preto môžete tieto dve priradenia zlúčiť do jedného:

```
>>> a = Vrchol('a', Vrchol('b'))
```

Hoci teraz je tu malý rozdiel a to v tom, že vrchol `Vrchol('b')` sa vytvorí skôr ako `Vrchol('a')`, čo ale vo väčšine prípadov nevadí. Podobne by sme vedeli jedným priradením vytvoriť nielen dvojprvkový, ale aj viacprvkový zoznam, napr.:

```
>>> zoznam = Vrchol('P', Vrchol('y', Vrchol('t', Vrchol('h', Vrchol('o', Vrchol('n
↪')))))
```

Vytvorí šesťprvkový zoznam, pričom každý prvok obsahuje jedno písmeno z reťazca ‚Python‘:

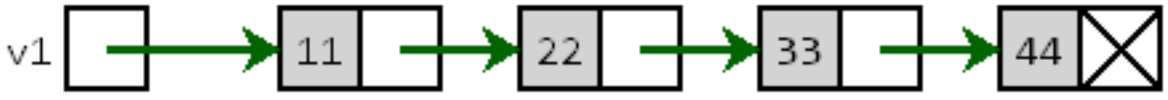


26.1.2 Výpis pomocou cyklu

Predpokladajte, že máte vytvorený nejaký, napr. štvorprvkový zoznam:

```
>>> v1 = Vrchol(11, Vrchol(22, Vrchol(33, Vrchol(44))))
```

V pamäti by ste ho mohli vidieť nejako takto:



Teraz treba vypísať všetky jeho hodnoty postupne od prvej po poslednú, môžete to urobiť napr. takto:

```
>>> print(v1.data)
11
>>> print(v1.next.data)
22
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> print(v1.next.next.data)
33
>>> print(v1.next.next.next.data)
44
```

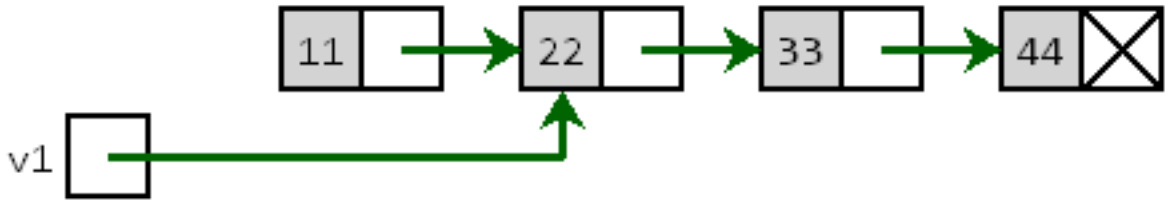
alebo v jednom riadku:

```
>>> print(v1.data, v1.next.data, v1.next.next.data, v1.next.next.next.data)
11 22 33 44
```

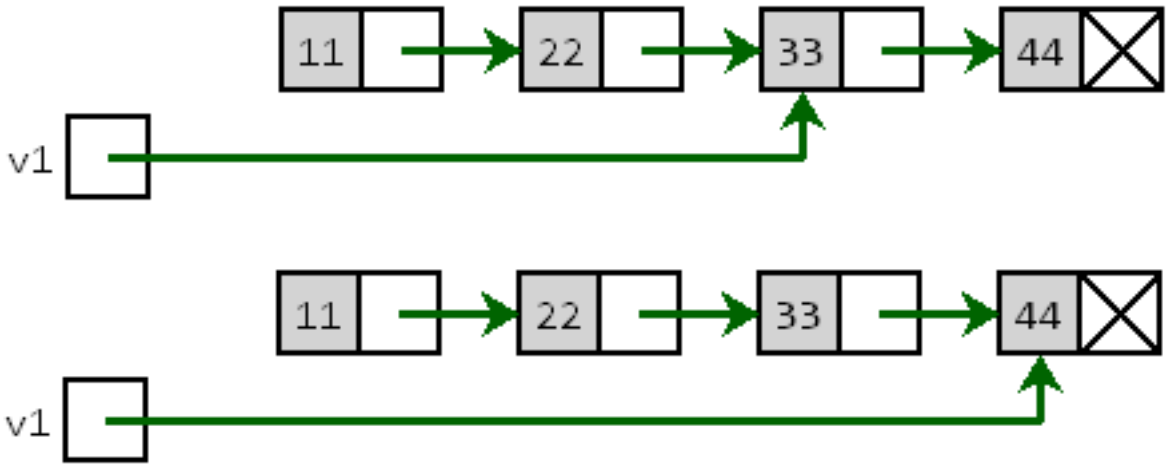
Zrejme pre zoznam ľubovoľnej dĺžky budeme musieť použiť nejaký cyklus, najskôr while-cyklus. Keď vypíšete prvú hodnotu, posuniete premennú `v1` na nasledovníka prvého vrcholu:

```
>>> print(v1.data)
>>> v1 = v1.next
```

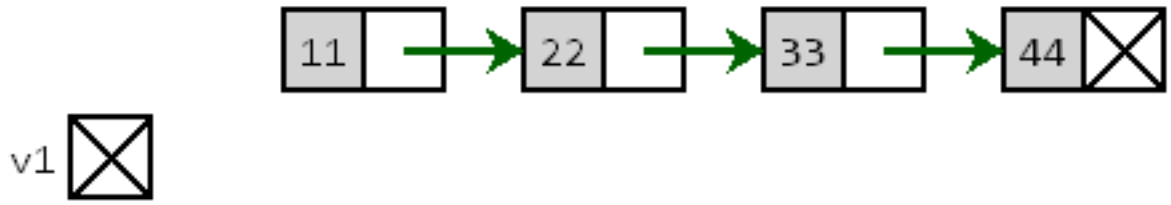
a môže sa to celé opakovať. Zápis `v1 = v1.next` je veľmi dôležitý a budeme ho v súvislosti so spájanými zoznamami používať veľmi často. Označuje, že do premennej `v1` sa namiesto referencie na nejaký vrchol dostáva referencia na jeho nasledovníka:



postupne dostávame:



Ak už tento vrchol nasledovníka nemá, do `v1` sa dostane hodnota `None`:



Preto kompletný výpis hodnôt zoznamu môžeme zapísať takto:

```
while v1 is not None:
    print(v1.data, end=' -> ')
    v1 = v1.next
print(None)
```

Pre názornosť sme tam medzi každé dve vypisované hodnoty pridali reťazec `' -> '`:

```
11 -> 22 -> 33 -> 44 -> None
```

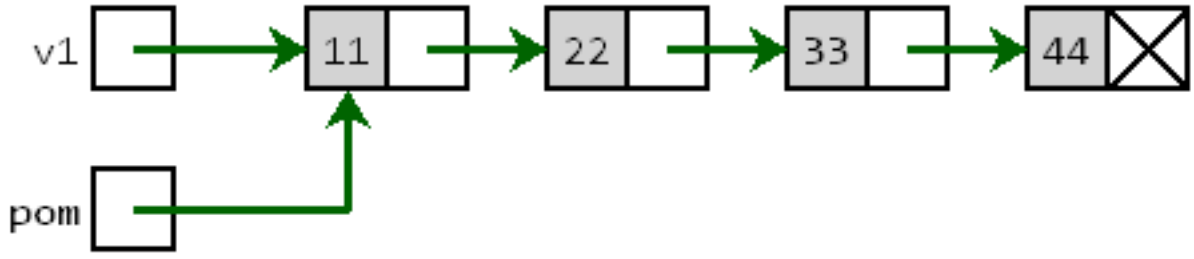
Hoci to vyzerá dobre a dostatočne jednoducho, má to jeden problém: po skončení vypisovania pomocou tohto while-cyklu je v premennej `v1` hodnota `None`:

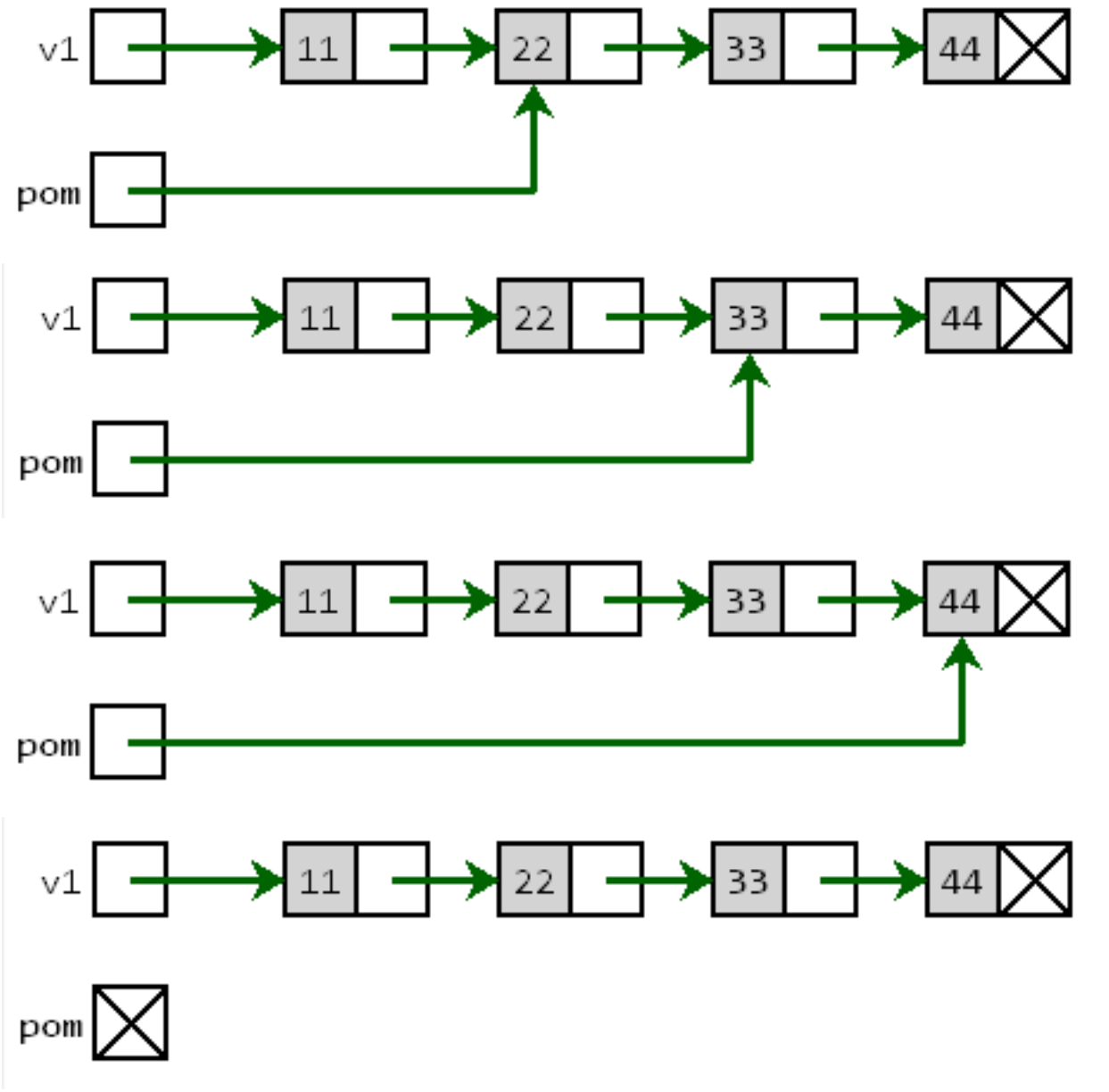
```
>>> print(v1)
None
```

Teda výpisom sme si zničili jedinú referenciu na prvý vrchol zoznamu a teda Python pochopil, že so zoznamom už pracovať ďalej nechceme a celú štruktúru z pamäti vyhodil (hovorí sa tomu *garbage collection*). Môžete to skontrolovať aj vo vizualizácii http://pythontutor.com/visualize.html. Tento príklad ukazuje to, že niekedy bude potrebné si uchovať referenciu na začiatok zoznamu, resp. v takomto cykle nebude pracovať priamo s premennou `v1`, ale s jej kópiou, napr. takto:

```
pom = v1
while pom is not None:
    print(pom.data, end=' -> ')
    pom = pom.next
print(None)
```

Po skončení tohto výpisu sa premenná `pom` vynuluje na `None`, ale začiatok zoznamu `v1` ostáva neporušený. Teraz je to už správne a v pamäti to postupne vyzerá takto:





Takýto výpis sa dá zapísať aj do funkcie, pričom tu pomocnú referenciu na začiatok zoznamu zastúpi parameter:

```
def vypis(zoznam):
    while zoznam is not None:
        print(zoznam.data, end=' -> ')
        zoznam = zoznam.next
    print(None)
```

Pri volaní funkcie sa do formálneho parametra `zoznam` priradí hodnota skutočného parametra (napr. obsah premennej `v1`) a teda referencia na začiatok zoznamu sa týmto volaním nepokazí.

Teraz môžete volať funkciu na výpis nielen so začiatkom zoznamu ale hoci napr. aj od druhého vrcholu:

```
>>> vypis(v1)
11 -> 22 -> 33 -> 44 -> None
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> vypis(v1.next)
22 -> 33 -> 44 -> None
```

Vidíte, že referencia na prvý vrchol v spájanom zozname má špeciálny význam a preto sa zvykne označovať nejakým dohodnutým menom, napr. `zoznam`, `zoz`, `zac`, `z` (ako začiatok zoznamu) alebo niekedy aj po anglicky `head` (hlavička zoznamu).

Postupné prechádzanie vrcholov zoznamu

Spôsob, akým sa prechádzajú všetky vrcholy zoznamu pomocou while-cyklu, bude užitočný aj na riešenie iných úloh. Často sa preto použije práve takáto schéma algoritmu:

```
pom = zoznam
while pom is not None:
    # spracuj vrchol s referenciou pom
    pom = pom.next
```

26.1.3 Vytvorenie zoznamu pomocou cyklu

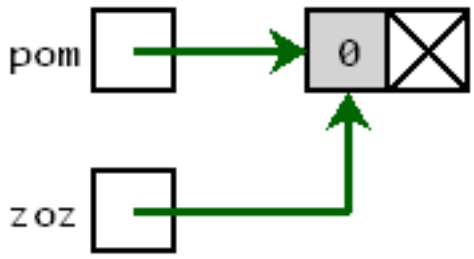
Zoznamy sa doteraz vytvárali sériou priradení a to bez cyklov. Častejšie sa ale budú vytvárať, možno aj dosť dlhé, zoznamy pomocou opakujúcich sa konštrukcií. Začneme vytváraním zoznamu pridávaním nového vrcholu na začiatok doterajšieho zoznamu, keďže toto je výrazne jednoduchšie.

Vytvoríme desaťprvkový zoznam s hodnotami 0, 1, 2, ... 9. Začneme s prázdnym zoznamom:

```
>>> zoz = None
```

Vytvoríme prvý vrchol s hodnotou 0 a dáme ho na začiatok:

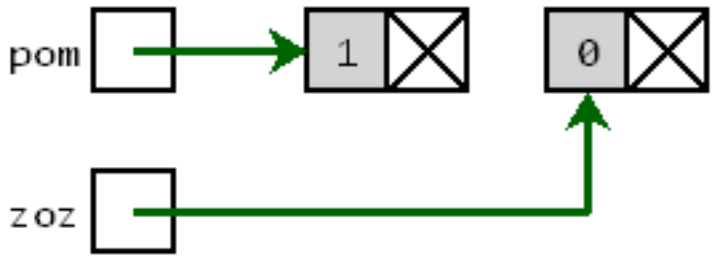
```
>>> pom = Vrchol(0)
>>> zoz = pom
```



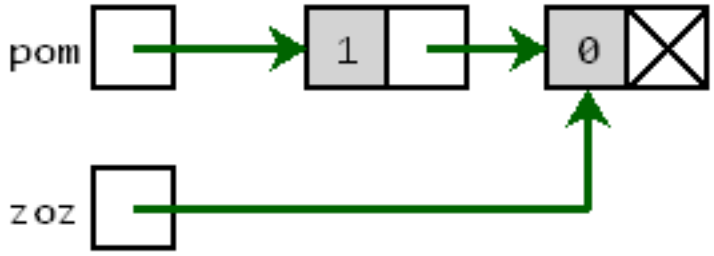
Keby sme to vypísali pomocou funkcie `vypis()`, dostali by sme: `0 -> None`

Vytvoríme druhý vrchol a dáme ho opäť na začiatok:

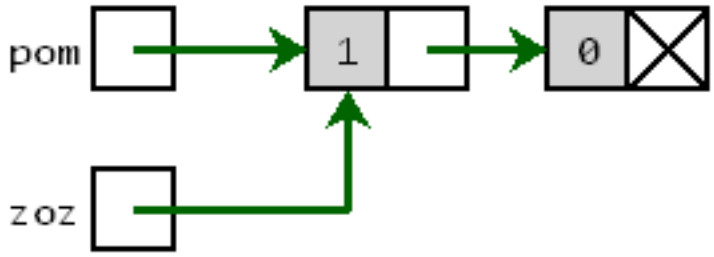
```
>>> pom = Vrchol(1)
```




```
>>> pom.next = zoz
```



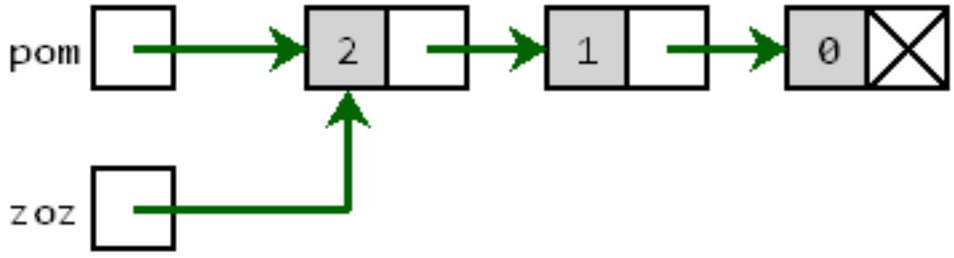

```
>>> zoz = pom
```



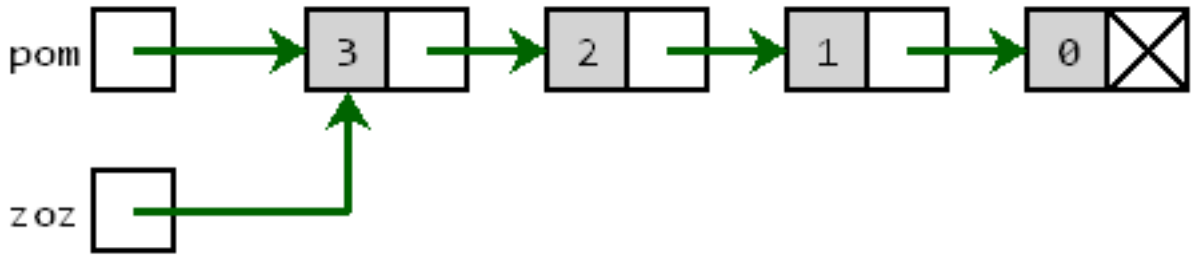
Po výpise by sme dostali: `1 -> 0 -> None`

Toto môžeme opakovať viackrát pre rôzne hodnoty - zakaždým sa na začiatok doterajšieho zoznamu pridá nový vrchol:

```
>>> pom = Vrchol(2)
>>> pom.next = zoz
>>> zoz = pom
```



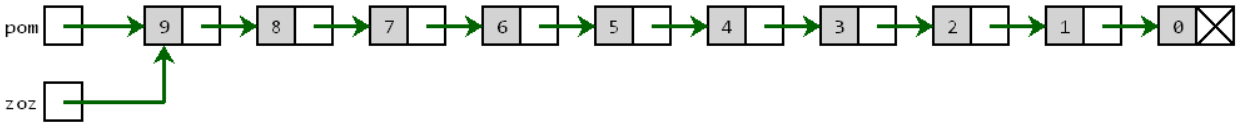
```
>>> pom = Vrchol(3)
>>> pom.next = zoz
>>> zoz = pom
```



Takto by sme mohli pokračovať až do 9. Teraz už vidíte, čo sa tu opakuje a čo treba dať do cyklu:

```
zoz = None                      # zatial este prazdny zoznam
for hodnota in range(10):
    pom = Vrchol(hodnota)
    pom.next = zoz
    zoz = pom
```

Týmto postupom sme dostali 10 prvkový zoznam hodnôt v poradí od 9 do 0




```
>>> vypis(zoz)
9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0 -> None
```

Opäť si všimnime zápis tela cyklu:

```
pom = Vrchol(hodnota)
pom.next = zoz
zoz = pom
```

Vytvorí sa tu nový vrchol najprv s danou hodnotou a nasledovníkom `None`. Potom sa tento nasledovník zmení na `pom.next = zoz` a na záver sa tento nový vrchol `pom` stáva novým začiatkom zoznamu, t.j. `zoz = pom`. Toto isté sa dá zapísať kompaktnejšie:

```
for hodnota in range(10):
    zoz = Vrchol(hodnota, zoz)
```

Pridanie nového vrcholu na začiatok zoznamu

Zapamätajte si, že zápis `zoz = Vrchol(hodnota, zoz)` pre `zoz`, ktorý referencuje na začiatok zoznamu, znamená pridanie nového vrcholu na začiatok zoznamu.

Takto by sme vedeli vytvoriť ľubovoľné zoznamy. Zapíšme tento algoritmus do funkcie:

```
def vyrob(postupnost):
    zoz = None
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    for hodnota in postupnost:
        zoz = Vrchol(hodnota, zoz)
    return zoz
```

Otestujme napr.

```
>>> zoz1 = vyrob(range(1000))
>>> vypis(zoz1)
999 -> 998 -> ... -> 1 -> 0 -> None
>>> zoz2 = vyrob('Python')
>>> vypis(zoz2)
n -> o -> h -> t -> y -> P -> None
```

Vytvorili sa dva zoznamy: prvý s 1000 vrcholmi a druhý so šiestimi vrcholmi s písmenami reťazca ‚Python'. Treba si pri tomto uvedomiť, že takto sa vytvárajú zoznamy s hodnotami v opačnom poradí, ako so do neho vkladali.

Častejšie budeme potrebovať vyrábať zoznamy, v ktorých budú prvky v tom poradí, v akom sme ich vkladali. Jednoduchým riešením môže byť prevrátenie vstupnej postupnosti pomocou `reversed()`:

```
def vyrob1(postupnost):
    zoz = None
    for hodnota in reversed(postupnost):
        zoz = Vrchol(hodnota, zoz)
    return zoz
```

Otestujeme:

```
>>> zoz2 = vyrob1('Python')
>>> vypis(zoz2)
P -> y -> t -> h -> o -> n -> None
```

Uvedomte si, že nie vždy môžeme takto jednoducho otočiť vstupnú postupnosť, z ktorej sa má vytvoriť spájaný zoznam. Napr. ak sú vstupnou postupnosťou riadky otvoreného textového súboru, tak takúto postupnosť neotočíme.

26.1.4 Zistenie počtu prvkov

Zapíšeme funkciu, ktorá spočíta počet prvkov zoznamu. Bude pracovať na rovnakom princípe ako funkcia `vypis()` len namiesto samotného vypisovania hodnoty funkcia zvýši počítadlo o 1:

```
def pocet(zoznam):
    vysl = 0
    while zoznam is not None:
        vysl += 1
        zoznam = zoznam.next
    return vysl
```

Otestujeme napr.

```
>>> zoz = vyrob('Python')
>>> pocet(zoz)
6
```

Malou úpravou túto funkciu vylepšíme:

```
def pocet(zoznam, hodnota=None):
    vysl = 0
    while zoznam is not None:
        if hodnota is None or zoznam.data == hodnota:
            vysl += 1
        zoznam = zoznam.next
    return vysl
```

Táto funkcia dokáže nielen zistiť počet prvkov zoznamu, ale aj počet výskytov nejakej konkrétnej hodnoty. Napr.

```
>>> zoz1 = vyrob('programujem v pythone')
>>> pocet(zoz1)
21
>>> pocet(zoz1, 'p')
2
```

26.1.5 Hľadanie vrcholu

Podobný cyklus, ako sme použili pri výpise a pri zisťovaní počtu prvkov, môžeme použiť pri zisťovaní, či sa daná hodnota nachádza v zozname. Napíšme funkciu, ktorá vráti `True`, ak nájde konkrétnu hodnotu, inak vráti `False`:

```
def zisti(zoznam, hodnota):
    while zoznam is not None:
        if zoznam.data == hodnota:
            return True
        zoznam = zoznam.next
    return False
```

Otestujeme:

```
>>> zoznam = vyrob('Python')
>>> zisti(zoznam, 'h')
True
>>> zisti(zoznam, 'g')
False
```

Táto funkcia skončila prechádzanie prvkov zoznamu už pri prvom výskyte hľadanej hodnoty.

26.1.6 Zmena hodnoty vo vrchole

Najprv jednoduchá funkcia, ktorá zmení všetky hodnoty v zozname:

```
def zmen(zoznam, hodnota):
    while zoznam is not None:
        zoznam.data = hodnota
        zoznam = zoznam.next
```

```
>>> zoznam = vyrob('Python')
>>> zmen(zoznam, 0)
>>> vypis(zoznam)
0 -> 0 -> 0 -> 0 -> 0 -> 0 -> None
```

Ak chceme zmeniť len vrcholy, ktoré obsahujú nejakú konkrétnu hodnotu, môžeme to zapísať takto:

```
def zmen(zoznam, hodnota, na_hodnotu):
    while zoznam is not None:
        if zoznam.data == hodnota:
            zoznam.data = na_hodnotu
            # return
        zoznam = zoznam.next
```

Príkaz `return` v tele cyklu spôsobí ukončenie funkcie už po prvom výskyte hľadanej hodnoty. Inak sa zmenia obsahy všetkých vrcholov s danou hodnotou.

Otestujeme so zakomentovaným `return`:

```
>>> zoz = vyrob((1, 2, 3) * 3)
>>> zmen(zoz, 2, 'xy')
>>> vypis(zoz)
1 -> xy -> 3 -> 1 -> xy -> 3 -> 1 -> xy -> 3 -> None
```

26.1.7 Vloženie vrcholu na koniec zoznamu

Chceme vyrobiť novú operáciu, ktorá vloží na koniec zoznamu nový vrchol s danou hodnotou. Už vieme, že pridávanie vrcholu na začiatok je takto jednoduché:

```
zoz = ...       # zoz je referencia na začiatok zoznamu
zoz = Vrchol(hodnota, zoz)
```

S pridávaním na koniec to bude zložitejšie: najprv bude treba nájsť posledný vrchol zoznamu a tomuto vrcholu zmeníme jeho atribút `next`, t.j. na záver urobíme

```
posledny.next = Vrchol(hodnota)
```

Hľadanie posledného vrcholu sa bude podobať na postupné prechádzanie všetkých vrcholov:

```
posledny = zoz                # posledny je pomocná referencia
while posledny is not None:
    posledny = posledny.next
```

Lenže toto nebude fungovať - po skončení while-cyklu nebude v premennej `posledny` referencia na posledný vrchol ale bude tam `None`. Treba to zapísať trochu zložitejšie - while neskončí až vtedy, keď v `posledny` bude `None`, ale keď jeho `next` bude `None`:

```
posledny = zoz
while posledny.next is not None:
    posledny = posledny.next
```

Teraz je to už naozaj dobre, ale toto bude fungovať len pre neprázdny zoznam. Pre prázdny zoznam hodnota premennej `posledny` bude `None` a preto `posledny.next` spadne na chybe. Tento špeciálny prípad musíme vyriešiť ešte pred cyklom. Teraz môžeme zapísať kompletnú funkciu, ktorá pridá na koniec zoznamu nový vrchol. Táto funkcia bude vracať ako svoj výsledok začiatok takto vytvoreného zoznamu:

```
def pridaj_koniec(zoz, hodnota):
    if zoz is None:
        return Vrchol(hodnota)
    posledny = zoz
    while posledny.next is not None:
        posledny = posledny.next
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    posledny.next = Vrchol(hodnota)
    return zoz
```

Môžete otestovať:

```
zoznam = None
for i in 'Python':
    zoznam = pridaj_koniec(zoznam, i)
vypis(zoznam)
```

Mali by ste dostať zoznam so 6 písmenami v správnom poradí. Zapamätajte si:

Hľadanie posledného vrcholu zoznamu

Aj práca s posledným vrcholom zoznamu sa môže vyskytnúť v našich programoch. Preto najčastejšie použijeme takýto zápis:

```
if zoz is None:
    '''spracuj prípad, ked' zoznam je prázdny'''
else:
    posledny = zoz
    while posledny.next is not None:
        posledny = posledny.next
    '''spracuj posledný vrchol'''
```

26.1.8 Vloženie nového vrcholu do zoznamu

Nový vrchol môžeme vložiť buď pred nejaký existujúci alebo za. Jednoduchšie to bude s vkladaním **za** nejaký existujúci. Zapíšme

```
def pridaj_za(zoznam, za_hodnotu, hodnota):
    while zoznam is not None and zoznam.data != za_hodnotu:
        zoznam = zoznam.next
    if zoznam is not None:
        zoznam.next = Vrchol(hodnota, zoznam.next)
```

To isté môžeme zapísať aj takto:

```
def pridaj_za(zoznam, za_hodnotu, hodnota):
    while zoznam is not None:
        if zoznam.data == za_hodnotu:
            zoznam.next = Vrchol(hodnota, zoznam.next)
            return
        zoznam = zoznam.next
```

Vkladanie **pred** vrchol bude trochu náročnejšie a bude sa trochu podobať hľadaniu posledného vrcholu v zozname:

```
def pridaj_pred(zoznam, pred_hodnotu, hodnota):
    if zoznam is None:
        return                          # nie je čo robit'
    if zoznam.data == pred_hodnotu:
        return Vrchol(hodnota, zoznam)  # pred prvý
    pom = zoznam
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    while pom.next is not None:
        if pom.next.data == pred_hodnotu:
            pom.next = Vrchol(hodnota, pom.next)
            break
        pom = pom.next
    return zoznam
```

Keďže v tomto prípade sa môže zmeniť začiatok zoznamu, táto funkcia vždy vráti začiatok takéhoto zoznamu.

Na tomto príklade sa dá ukázať ešte jedno programátorské vylepšenie prechádzania spájaného zoznamu. Okrem pomocnej referencie `pom` budeme mať ešte jednu referenciu `pred` na predchodcu `pom`:

```
def pridaj_pred(zoznam, pred_hodnotu, hodnota):
    if zoznam is None:
        return                          # nie je čo robiť
    pred, pom = None, zoznam
    while pom is not None and pom.data != pred_hodnotu:
        pred, pom = pom, pom.next
    if pred is None:
        zoznam = Vrchol(hodnota, zoznam)  # pred prvý
    elif pom is not None:
        pred.next = Vrchol(hodnota, pred.next)
    return zoznam
```

Všimnite si, že vo while-cykle sa paralelne menia obe referencie: `pom` na svojho nasledovníka a `pred` na predchodcu `pom`. Keď while-cyklus skončí a v `pom` je `None`, znamená to, že budeme pracovať s vrcholom, ktorý nemá predchodcu, čo je prvý vrchol v zozname (máme vložiť pred prvý). Ak po skončení while-cyklu je v `pom` hodnota `None`, znamená to, že sme prešli celý spájaný zoznam a nenašli sme vrchol, ktorého hodnota je zadané `pred_hodnotu`.

26.2 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/
- pozrite si *Riešenie 26. cvičenia*

1. Bez spúšťania na počítači zistite, čo urobia nasledovné programy.
   - predpokladajte tieto deklarácie a funkcie z prednášky:

```
class Vrchol:
    def __init__(self, data, next=None):
        self.data, self.next = data, next

def vypis(zoznam):
    while zoznam is not None:
        print(zoznam.data, end=' -> ')
        zoznam = zoznam.next
    print(None)

def vyrob(postupnost):
    zoz = None
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    for hodnota in reversed(postupnost):
        zoz = Vrchol(hodnota, zoz)
    return zoz
```

- prvý spájaný zoznam:

```
v1 = Vrchol('A')
v2 = Vrchol('B')
v3 = Vrchol('C')
v4 = Vrchol('D')
v3.next = v1
v1.next = v2
v2.next = v3
v1.next = v4
zoz = v2
vypis(zoz)
```

- druhý spájaný zoznam:

```
v1 = None
v2 = Vrchol('X', v1)
v2 = Vrchol('Y', v2)
v3 = Vrchol('Z', v1)
v3 = Vrchol('T', v3)
v2 = Vrchol('U', v2)
v3.next.next = v2
vypis(v3)
```

2. Napíšte funkciu `pripocitaj1(zoznam)`, ktorá ku každému prvku zoznamu pripočíta 1, ale len vtedy, ak sa dá (nevznikne pritom chyba), inak tento prvok nezmení a pokračuje na ďalších. Funkcia nič nevracia.

   - otestujte, napr.

```
>>> zoz = vyrob([11, 12, 13.5, 'a', (2, 3), 14])
>>> pripocitaj1(zoz)
>>> vypis(zoz)
12 -> 13 -> 14.5 -> a -> (2, 3) -> 15 -> None
```

3. Prerobte funkciu `vypis()` tak, aby najprv vytvorila zoznam reťazcov z jednotlivých prvkov spájaného zoznamu a až na záver pomocou `' -> '.join(zoznam)` z toho vyrobí reťazec, ktorý vypíše

   - otestujte tento nový výpis aj pre dlhý zoznam (najprv s pôvodnou verziou a potom s prerobenou):

```
>>> vypis(vyrob(range(10000)))
```

4. Bez spúšťania na počítači zistite, čo urobí:

   - tretí spájaný zoznam:

```
zoz = vyrob((1, 3, 5, 7, 9, 11, 13))
v = zoz.next.next
v1 = v.next.next
v.next.next = v1.next
v1.next = v.next
v.next = v1
vypis(zoz)
```

5. Napíšte rekurzívnu verziu funkcie `pocet(zoznam)`, t.j. funkcia prejde všetky prvky zoznamu bez použitia cyklu len pomocou rekurzie. Nepridávajte ďalšie parametre do definície funkcie.

   - otestujte

```
>>> zoz = vyrob(range(10, 20))
>>> pocet(zoz)
10
>>> pocet(zoz.next)
9
```

6. Napíšte funkciu `spoj(zoz1, zoz2)`, ktorá na koniec zoznamu `zoz1` pripojí zoznam `zoz2`. Funkcia ako výsledok vráti začiatok takéhoto nového zoznamu. Nepoužívajte žiadne pomocné zoznamy (napr. typu `list`).

   - otestujte napr.

```
>>> z1 = vyrob('ABC')
>>> z2 = vyrob('prst')
>>> z = spoj(z1, z2)
>>> vypis(z)
A -> B -> C -> p -> r -> s -> t -> None
>>> vypis(spoj(None, Vrchol(1234)))
1234 -> None
```

7. Napíšte funkciu `prevratena_kopia(zoznam)`, ktorá vytvorí a vráti z daného zoznamu nový zoznam. Tento bude mať všetky prvky z pôvodného v opačnom poradí. Pôvodný zoznam musí ostať bez zmeny. Nepoužívajte žiadne pomocné zoznamy (typ `list`).

   - otestujte

```
>>> z1 = vyrob('python')
>>> vypis(z1)
p -> y -> t -> h -> o -> n -> None
>>> z2 = prevratena_kopia(z1)
>>> vypis(z2)
n -> o -> h -> t -> y -> p -> None
>>> vypis(z1)
p -> y -> t -> h -> o -> n -> None
```

8. Napíšte funkciu `oprav(zoznam, funkcia)`, ktorá pre každý vrchol v danom zozname spustí zadanú funkciu s parametrom hodnota vo vrchole a ak to nespadne na chybe, zmení hodnotu vrcholu. Funkcia nič nevracia.

   - napr.

```
>>> zoz = vyrob((1, -2, 3, -4, 5, -6))
>>> oprav(zoz, abs)
>>> vypis(zoz)
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> None
```

9. Napíšte funkciu `vyhod_prvy(zoznam)`. Funkcia vráti pôvodný zoznam bez prvého prvku

   - napr.

```
>>> x = Vrchol(5, Vrchol(7))
>>> vypis(x)
5 -> 7 -> None
>>> x = vyhod_prvy(x)
>>> vypis(x)
7 -> None
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> x = vyhod_prvy(x)
>>> vypis(x)
None
```

10. Napíšte funkciu `vyhod_posledny(zoznam)`. Funkcia vráti pôvodný zoznam bez posledného prvku

    - napr.

```
>>> x = Vrchol(5, Vrchol(7))
>>> vypis(x)
5 -> 7 -> None
>>> x = vyhod_posledny(x)
>>> vypis(x)
5 -> None
>>> x = vyhod_posledny(x)
>>> vypis(x)
None
```

11. Napíšte funkciu `vyhod_kazdy_druhy(zoznam)`, ktorá zo zoznamu vyhodí každý druhý prvok. Funkcia nič nevracia.

    - napr.

```
>>> zoz = vyrob('abcdef')
>>> vypis(zoz)
a -> b -> c -> d -> e -> f -> None
>>> vyhod_kazdy_druhy(zoz)
>>> vypis(zoz)
a -> c -> e -> None
>>> vyhod_kazdy_druhy(zoz)
>>> vypis(zoz)
a -> e -> None
```

12. Napíšte funkciu `vyhod(zoznam, podmienka)`, ktorá vyhodí všetky prvky zo zoznamu, pre ktoré zavolanie parametra `podmienka` s hodnotou vo `data` vo vrchole vráti `True`.

    - napr.

```
>>> zoz = vyrob(range(5, 12))
>>> vypis(zoz)
5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> None
>>> zoz = vyhod(zoz, lambda x: x%3)
>>> vypis(zoz)
6 -> 9 -> None
```

26.3 2. Domáce zadanie

L.I.S.T.

- riešenie odovzdávajte na úlohový server https://list.fmph.uniba.sk/

Zapíšte metódy triedy `Turing` s týmito metódami:

```
class Turing:
    def __init__(self, program, obsah=''):
        self.prog = {}
        ...

    def restart(self, stav=None, obsah=None, n=None):
        # od noveho stavu (ak nie je None), s novou paskou (ak nie je None) a zavola␣
↪rob()
        return False, 0

    def rob(self, n=None):
        return False, 0

    def text(self):
        return ''
```

Tento Turingov stroj by sa mal správať veľmi podobne, ako ten z prednášky Turingov stroj, líšiť sa bude len v niekoľkých detailoch:

- inicializácia `__init__()` má dva parametre `program` a počiatočný stav pásky, pričom `program` sa zadáva inak ako bolo v prednáške (uvádzame nižšie)
- metódy `restart()` a `rob()` majú posledný parameter `n`, ktorý, ak je zadaný, určuje maximálny počet vykonávaných pravidiel, ak by sa mal tento počet presiahnuť, výpočet končí tak, akokeby neakceptoval vstup (Turingov stroj vtedy vykoná maximálne len `n` pravidiel, ale `n+1`-pravidlo už nie)
- obe tieto metódy vrátia dvojicu: `True`/`False` a počet vykonaných pravidiel
- metóda `restart()` môže mať tieto ďalšie 2 parametre, ktorým môžeme nastaviť počiatočný stav, resp. zmeniť obsah pásky (pozícia na páske sa vždy nastaví na 0), táto metóda okrem nastavovania stavu a pásky zavolá metódu `rob()`
- množina koncových stavov je `{'end', 'stop'}`
- metóda `text()` vráti momentálny stav pásky, ktorý je očistený od úvodných a záverečných prázdnych znakov `'_'`
- v triede `Turing` môžete dodefinovať ďalšie metódy aj atribúty

26.3.1 Formát zadávaného programu pre Turingov stroj

Doteraz sme definovali Turingov stroj ako zoznam pravidiel, pričom každé z nich bolo päticou:

```
stav1 znak1 znak2 smer stav2
```

Napr.

```
s1 0 0 > s1
s1 1 1 > s1
s1 _ _ < s2

s2 1 0 < s2
s2 0 1 = end
s2 _ 1 = end
```

Tento istý Turingov stroj môžeme definovať aj takouto tabuľkou:

	s1	s2	
0	0 > s1	1=end	
1	1 > s1	0 < s2	
_	_ < s2	1=end	

V tejto tabuľke sú v prvom riadku vymenované všetky stavy (okrem koncových), v prvom stĺpci sú všetky sledované symboly, pre ktoré existuje pravidlo. Každé vnútorné políčko tabuľky reprezentuje jedno pravidlo: `stav1` z príslušného stĺpca a `znak1` z príslušného riadka, samotné políčko obsahuje trojicu `znak2 smer stav2`. Ak pre nejakú dvojicu `stav1`, `znak1` neexistuje pravidlo, tak na príslušnom mieste tabuľky je namiesto trojice reťazcov jeden znak `'.'`.

Niekedy sa takáto tabuľka môže trochu sprehľadniť tým, že sa môžu vynechať časti `znak2`, resp. `stav2`, ak sa zhodujú so `znak1`, resp. `stav1`. Predchádzajúca tabuľka by mohla vyzerať aj takto a popisovala by ten istý Turingov stroj:

	s1	s2	
0	>	1=end	
1	>	0<	
_	<s2	1=end	

Všimnite si, že sme tu vynechali medzery v trojiciach vo vnútri tabuľky. Takúto tabuľku budeme do triedy `Turing` zadávať pri inicializácii, napr. takto:

```
prog = '''
    s1    s2
0    >    1=end
1    >    0<
_   <s2   1=end
'''
t = Turing(prog, '1011')
```

Uvedomte si, že ak má Turingov stroj `n` stavov (okrem koncových), tak prvý riadok súboru obsahuje `n` reťazcov - názvov stavov (prvý z nich bude štartový), ktoré sú navzájom oddelené aspoň jednou medzerou. Každý ďalší riadok (podľa počtu rôznych rozlišovaných znakov) obsahuje presne `n+1` reťazcov navzájom oddelených aspoň jednou medzerou.

Ďalší príklad ukazuje tabuľku aj s políčkami bez pravidiel:

	s1	s2	s3	s4	s5	
a	>s2	.	.	.	.	
h	.	>s3	.	.	.	
o	.	.	>s4	.	.	
j	.	.	.	>s5	.	
_	.	.	.	.	=end	

26.3.2 Obmedzenia

- vaše riešenie odovzdajte v súbore `riesenie2.py`, pričom sa v ňom bude nachádzať len jedna definícia triedy `Turing`
- atribút `prog` v triede `Turing` bude obsahovať asociatívne pole s pravidlami Turingovho stroja (vo formáte z prednášky Turingov stroj)
- prvé dva riadky tohto súboru budú obsahovať:

```
# autor: Janko Hrasko
# uloha: 2. domace zadanie Turing
```

- zrejme ako autora uvediete svoje meno
- váš program by nemal počas testovania testovačom nič vypisovať (žiadne testovacie `print()`)

26.3.3 Testovanie

Keď budete spúšťať vaše riešenie na svojom počítači, môžete do súboru `riesenie2.py` pridať testovacie riadky, ktoré takto zapísané testovač nebude vidieť, napr.:

```
if __name__ == '__main__':
    prog = '''
        s1    s2
    0    >    1=end
    1    >    0<
    _    <s2   1=end
    '''
    t = Turing(prog, '1011')
    print(t.prog)
    print(t.rob())
    print('vysledok =', t.text())
    print(t.restart('s1', '10102010'))
```

Tento test by vám mal vypísať:

```
{('s2', '_'): ('1', '=', 'end'), ('s1', '_'): ('_', '<', 's2'), ('s2', '1'): ('0', '<
↪', 's2'),
('s1', '0'): ('0', '>', 's1'), ('s1', '1'): ('1', '>', 's1'), ('s2', '0'): ('1', '=',
↪'end')}
(True, 8)
vysledok = 1100
(False, 4)
```

KAPITOLA 27

27. Spájané zoznamy

Predchádzajúca prednáška sa venovala spájanej dátovej štruktúre, v ktorej mal každý prvok svojho nasledovníka (okrem posledného). Takejto štruktúre sme hovorili **spájaný zoznam** (linked list).

Zhrňme z minulej prednášky najdôležitejšie funkcie, ktoré pracujú so spájanými zoznamami. Používali sme túto deklaráciu triedy `Vrchol`:

```
class Vrchol:
    def __init__(self, data, next=None):
        self.data, self.next = data, next
```

Takýto `Vrchol` využívali všetky ďalšie funkcie:

```
def vypis(zoznam):                      # vypíše prvky zoznamu
    while zoznam is not None:
        print(zoznam.data, end=' -> ')
        zoznam = zoznam.next
    print(None)

def vyrob(postupnost):                  # z prvkov danej postupnosti vytvorí nový␣
↪zoznam
    zoz = None
    for hodnota in reversed(postupnost):
        zoz = Vrchol(hodnota, zoz)
    return zoz

def pocet(zoznam):                      # zistí počet prvkov zoznamu
    vysl = 0
    while zoznam is not None:
        vysl += 1
        zoznam = zoznam.next
    return vysl

def zisti(zoznam, hodnota):             # zistí, či je prvok v zozname
    while zoznam is not None:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        if zoznam.data == hodnota:
            return True
        zoznam = zoznam.next
    return False

def pridaj_zaciatok(zoz, hodnota):      # pridaj prvok na začiatok zoznamu
    return Vrchol(hodnota, zoz)

def pridaj_koniec(zoz, hodnota):        # pridaj prvok na koniec zoznamu
    if zoz is None:
        return Vrchol(hodnota)
    posledny = zoz
    while posledny.next is not None:
        posledny = posledny.next
    posledny.next = Vrchol(hodnota)
    return zoz
```

V podobnom duchu boli aj ďalšie funkcie, ktoré ste programovali na cvičeniach. Keď sa vytvára nejaká nová dátová štruktúra, veľmi často sa všetky funkcie zapuzdria do jedného celku - do jednej triedy, pričom sa skryjú realizačné detaily a aj nejaké pomocné funkcie a atribúty (hovoríme tomu **enkapsulácia**).

27.1 Trieda spájaný zoznam

Navrhneme novú triedu `SpajanyZoznam` (v anglickej verzii by sme ju nazvali `LinkedList`), pre ktorú zadefinujeme najdôležitejšie metódy. Všimnite si, že definíciu **pomocnej triedy** `Vrchol` sme vnorili do triedy `SpajanyZoznam`. Tým, že sme definíciu tejto triedy vnorili, oznamujeme, že patrí do triedy `SpajanyZoznam` a zrejme sa bude využívať v metódach tejto triedy. Všetky funkcie, ktoré pracovali so spájaným zoznamom, mali prvý parameter referenciu na prvý prvok zoznamu. Teraz tento parameter zastúpi atribút `zac`, teda **začiatok zoznamu**. Už vieme, že si bude treba dať pozor, aby sme túto referenciu nepokazili. Tiež si všimnite, že funkciu `vyrob(postupnost)` sme zakomponovali priamo do inicializácie zoznamu:

```
class SpajanyZoznam:

    #----- vnorena trieda -----
    class Vrchol:
        def __init__(self, data, next=None):
            self.data, self.next = data, next

    #----- koniec vnorenej definicie triedy -----

    def __init__(self, postupnost=None):
        self.zac = None                 # začiatok zoznamu
        if postupnost is not None:
            for hodnota in reversed(postupnost):
                self.pridaj_zaciatok(hodnota)

    def vypis(self):                    # vypíše prvky zoznamu
        pom = self.zac
        while pom is not None:
            print(pom.data, end=' -> ')
            pom = pom.next
        print(None)

    def pocet(self):                    # zistí počet prvkov zoznamu
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        vysl = 0
        pom = self.zac
        while pom is not None:
            vysl += 1
            pom = pom.next
        return vysl

    def zisti(self, hodnota):              # zistí, či je prvok v zozname
        pom = self.zac
        while pom is not None:
            if pom.data == hodnota:
                return True
            pom = pom.next
        return False

    def pridaj_zaciatok(self, hodnota):    # pridaj prvok na začiatok zoznamu
        self.zac = self.Vrchol(hodnota, self.zac)

    def pridaj_koniec(self, hodnota):      # pridaj prvok na koniec zoznamu
        if self.zac is None:
            self.zac = self.Vrchol(hodnota)
        pom = self.zac
        while pom.next is not None:
            pom = pom.next
        pom.next = self.Vrchol(hodnota)
```

Skôr ako to budeme testovať, všimnite si, že metódy `pridaj_zaciatok()` a `pridaj_koniec()` už nie sú také funkcie, ktoré vždy vracali novú referenciu na začiatok zoznamu - teraz túto referenciu už nepotrebujeme ako výsledok funkcie. Samotné metódy zmenia referenciu na začiatok v atribúte `zac`. Tiež vidíte použite vnorenej triedy `Vrchol`: aby sme mohli vytvoriť nový vrchol, musíme zapísať `self.Vrchol(hodnota)`. Zapíšme nejaký jednoduchý test, aby sme si zvykli na prácu s touto dátovou štruktúrou:

```
>>> zoz = SpajanyZoznam()
>>> zoz.pridaj_zaciatok(7)
>>> zoz.pridaj_koniec('abc')
>>> zoz.pridaj_zaciatok((1, 2))
>>> zoz.vypis()
(1, 2) -> 7 -> abc -> None
>>> zoz.pocet()
3
>>> zoz.zisti('abc')
True
```

Funguje to podľa očakávania dobre. Len by to mohlo byť viac pythonovské:

- namiesto metódy `vypis()` by mohlo byť radšej `__repr__()` alebo `__str__()` a teda by fungovalo napr. `print(zoz)`
- namiesto `pocet()` radšej `__len__()` a teda by fungovalo `len(zoz)`
- namiesto `pridaj_koniec()` radšej `append()`, aby sa to podobalo pythonovskému pridávaniu na koniec pythonovského zoznamu
- namiesto `zisti()` radšej `__contains__()` a teda by fungovalo `hodnota in zoz`

Budeme sa snažiť aj ďalšie metódy zapisovať tak, aby sa so spájaným zoznamom pracovalo podobne ako s inými dátovými typmi. Prepíšme triedu `SpajanyZoznam`:

```
class SpajanyZoznam:
    class Vrchol:
        def __init__(self, data, next=None):
            self.data, self.next = data, next

    def __init__(self, postupnost=None):
        self.zac = None                    # začiatok zoznamu
        if postupnost is not None:
            for hodnota in reversed(postupnost):
                self.insert0(hodnota)

    def __repr__(self):
        vysl, pom = [], self.zac
        while pom is not None:
            vysl.append(repr(pom.data))
            pom = pom.next
        vysl.append('None')
        return ' -> '.join(vysl)

    def __len__(self):
        vysl, pom = 0, self.zac
        while pom is not None:
            vysl += 1
            pom = pom.next
        return vysl

    def __contains__(self, hodnota):
        pom = self.zac
        while pom is not None:
            if pom.data == hodnota:
                return True
            pom = pom.next
        return False

    def insert0(self, hodnota):
        self.zac = self.Vrchol(hodnota, self.zac)

    def append(self, hodnota):
        if self.zac is None:
            self.zac = self.Vrchol(hodnota)
        pom = self.zac
        while pom.next is not None:
            pom = pom.next
        pom.next = self.Vrchol(hodnota)
```

Pristavme sa na dvoch posledných metódach:

- metóda `insert0()`, ktorá pridáva nový prvok na začiatok zoznamu, je veľmi rýchla, lebo obsahuje len jedno priradenie a bude trvať rovnaký čas bez ohľadu na to, či je doterajší zoznam krátky alebo obsahuje už veľa prvkov
- metóda `append()`, ktorá pridáva nový prvok na koniec zoznamu, je v niektorých prípadoch **veľmi pomalá**: ak už doterajší zoznam obsahuje obrovské množstvo prvkov (napr. 100000), pridať nový prvok na koniec bude trvať už citeľne nejaký nezanedbateľný čas (napr. 0.01 sekundy); keď takéto pridávanie urobíme 1000 krát, už to môžu byť desiatky sekúnd.

Zrejme, čo v tejto metóde závisí od momentálnej veľkosti zoznamu, je vnútorný while-cyklus. Ak by sme sa ho dokázali zbaviť, aj metóda `append()` by mohla byť dostatočne rýchla. Tento cyklus nerobí nič iné, len hľadá

momentálne posledný vrchol v zozname. Ak by sme ale okrem referencie na začiatok zoznamu zabezpečili pamätanie aj referencie na posledný vrchol, všetko by sa vyriešilo.

Do inicializácie pridáme vytvorenie ďalšieho atribútu `kon`, v ktorom vždy bude referencia na posledný vrchol zoznamu. Pri všetkých metódach, ktoré nejako modifikujú samotný spájaný zoznam, bude treba zabezpečiť, aby sa správne nastavila aj táto koncové referencia. V našom programe sa okrem inicializácie a metódy `append()` musí opraviť aj metóda `insert0()`:

```
class SpajanyZoznam:
    class Vrchol:
        def __init__(self, data, next=None):
            self.data, self.next = data, next

    def __init__(self, postupnost=None):
        self.zac = self.kon = None                    # začiatok a koniec zoznamu
        if postupnost is not None:
            for hodnota in postupnost:
                self.append(hodnota)

    ...

    def insert0(self, hodnota):
        self.zac = self.Vrchol(hodnota, self.zac)
        if self.kon is None:
            self.kon = self.zac

    def append(self, hodnota):
        if self.zac is None:
            self.kon = self.zac = self.Vrchol(hodnota)
        else:
            self.kon.next = self.Vrchol(hodnota)
            self.kon = self.kon.next
```

V tejto novej verzii v metóde `append()` už nie je žiaden cyklus a obsahuje len jeden test a niekoľko priradení. Môžete otestovať rýchlosť tejto metódy napr. takto:

```
>>> zoz = SpajanyZoznam(range(100000))
>>> for i in range(100000):
        zoz.append(i)
>>> len(zoz)
200000
```

Doplňme do tejto implementácie triedy `SpajanyZoznam` aj metódy `pop0()` a `pop()`, ktoré **vyhodia** 1. prvok, resp. posledný prvok zoznamu a **vrátia** príslušnú hodnotu vyhadzovaného prvku (podobne, ako to robia metódy `list.pop(0)` a `list.pop()`):

```
class SpajanyZoznam:
    ...

    def pop0(self):
        if self.zac is None:
            raise EmptyError
        vysl = self.zac.data
        self.zac = self.zac.next
        if self.zac is None:
            self.kon = None
        return vysl                    # hodnota vyhodeného prvku
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def pop(self):
        if self.zac is None:
            raise EmptyError
        if self.zac.next is None:      # jednoprvkový zoznam
            vysl = self.zac.data
            self.zac = self.kon = None
            return vysl
        self.kon = self.zac
        while self.kon.next.next is not None:
            self.kon = self.kon.next
        vysl = self.kon.next.data
        self.kon.next = None
        return vysl                    # hodnota vyhodeného prvku
```

Vidíme, že vyhodenie prvého prvku zoznamu (metóda `pop0()`) je veľmi jednoduché a rýchle (nezávisí od momentálnej veľkosti zoznamu). Metóda na vyhodenie posledného prvku je už náročnejšia a obsahuje while-cyklus na nájdenie predposledného vrcholu zoznamu. Preto je táto metóda veľmi pomalá a už nám tu nepomôže ani *finta* s udržiavaním si referencie na posledný vrchol.

Otestujme:

```
>>> zoz = SpajanyZoznam(range(10000))
>>> for i in range(10000):
        x = zoz.pop()
```

Zhrňme základné metódy, ktoré pridávajú, resp. odoberajú prvok zo začiatku alebo konca zoznamu:

- metóda `insert0()` vloží na začiatok zoznamu - je veľmi rýchla
- metóda `append()` vloží na koniec zoznamu - je veľmi rýchla len ak využíva pomocnú referenciu na koniec zoznamu (inak je pomalá)
- metóda `pop0()` vyberie zo začiatku zoznamu - je veľmi rýchla
- metóda `pop()` vyberie z konca zoznamu - je veľmi pomalá a pre jednosmerný spájaný zoznam neexistuje spôsob, ako to urýchliť

Keď sa budete niekedy rozhodovať, ako budete pracovať s nejakým spájaným zoznamom, spomeňte si na toto porovnanie rýchlostí a ak sa bude dať, snažte sa vyhnúť odoberaniu prvku z konca zoznamu.

27.1.1 Mapovacie metódy

Zapíšme metódu, ktorá postupne prejde všetky vrcholy spájaného zoznamu a každému zmení hodnotu podľa zadanej funkcie:

```
class SpajanyZoznam:
    ...

    def mapuj(self, funkcia):
        pom = self.zac
        while pom is not None:
            pom.data = funkcia(pom.data)
            pom = pom.next
```

Otestujeme:

```
>>> def fun(x): return x * x
>>> zoz = SpajanyZoznam(range(5))
>>> zoz.mapuj(fun)
>>> zoz
0 -> 1 -> 4 -> 9 -> 16 -> None
```

Funguje to aj s `lambda` konštrukciou, napr.

```
>>> zoz = SpajanyZoznam('Python')
>>> zoz.mapuj(lambda x: x.upper())
>>> zoz
'P' -> 'Y' -> 'T' -> 'H' -> 'O' -> 'N' -> None
```

Parametrom funkcie môže byť aj nejaká podmienka, napr. takáto verzia `mapuj`:

```
class SpajanyZoznam:
    ...

    def mapuj(self, podmienka, funkcia):
        pom = self.zac
        while pom is not None:
            if podmienka(pom.data):
                pom.data = funkcia(pom.data)
            pom = pom.next
```

napr.

```
>>> zoz = SpajanyZoznam(range(1, 20, 2))
>>> zoz
1 -> 3 -> 5 -> 7 -> 9 -> 11 -> 13 -> 15 -> 17 -> 19 -> None
>>> zoz.mapuj(lambda x: x%3, lambda x: x*x)
>>> zoz
1 -> 3 -> 25 -> 49 -> 9 -> 121 -> 169 -> 15 -> 289 -> 361 -> None
```

Ďalšia metóda vytvorí obyčajný zoznam (typu `list`) prvkov z prvkov spájaného zoznamu, ktoré spĺňajú nejakú podmienku:

```
class SpajanyZoznam:
    ...

    def tolist(self, podmienka=None):
        lst = []
        pom = self.zac
        while pom is not None:
            if podmienka is None or podmienka(pom.data):
                lst.append(pom.data)
            pom = pom.next
        return lst
```

napr.

```
>>> zoz = SpajanyZoznam((1, 2, 'A', 4, 'B'))
>>> zoz.tolist(lambda x: isinstance(x, int))
[1, 2, 4]
>>> zoz.tolist()
[1, 2, 'A', 4, 'B']
>>> def podm(x): return type(x) == str
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> zoz.tolist(podm)
['A', 'B']
```

27.1.2 Spájaný zoznam a for-cyklus

Už sme si zvykli, že prvky spájaného zoznamu môžeme prechádzať while cyklom, napr.

```
class SpajanyZoznam:
    ...

    def sucet(self):
        pom = self.zac
        vysl = 0
        while pom is not None:
            vysl += pom.data
            pom = pom.next
        return vysl
```

Žiaľ, prechádzať prvky spájaného zoznamu tak, ako to vie Python so zoznamom, n-ticou, množinou, atď. sa zatiaľ nedá. Ak by sme vyskúšali:

```
>>> zoz = SpajanyZoznam((2, 3, 5, 7, 11))
>>> for prvok in zoz:
        print(prvok)
...
TypeError: 'SpajanyZoznam' object is not iterable
```

alebo

```
>>> for prvok in zoz.zac:
        print(prvok.data)
...
TypeError: 'Vrchol' object is not iterable
```

Python vyhlási chybu `TypeError`, lebo nevie, akým spôsobom by mal postupne prechádzať (iterovať) všetky prvky spájaného zoznamu. Uvidíme, že Python to dokáže, ale potrebuje na to, aby sme mu niečo o našej štruktúre prezradili.

27.1.3 Rekurzia v metóde

Na cvičeniach ste zostavovali rekurzívnu funkciu, ktorá zisťovala počet prvkov spájaného zoznamu. Vaše riešenie mohlo vyzerať napr. takto:

```
def pocet(zoz):
    if zoz is None:
        return 0
    return pocet(zoz.next) + 1
```

Ak by sme túto rekurziu chceli zapísať ako metódu triedy `SpajanyZoznam`, najlepšie to urobíme ako vnorenú rekurzívnu funkciu do príslušnej metódy:

```
class SpajanyZoznam:
    ...
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def pocet(self):
        #----- vnorená funkcia
        def pocet_rek(zoz):
            if zoz is None:
                return 0
            return pocet_rek(zoz.next) + 1
        #----- koniec vnorenej funkcie

        return pocet_rek(self.zac)
```

Hoci táto rekurzívna funkcia nemá žiaden praktický význam (funguje len pre zoznamy kratšie ako 1000), ideu vnorených a hlavne rekurzívnych funkcií budeme neskôr používať veľmi často.

27.2 Realizácia zásobníka a radu

V prednáške *25. Zásobníky a rady* sme sa zoznámili s dátovou štruktúrou zásobník. Využili sme ho hlavne pri spracovávaní aritmetických výrazov, ale aj ako mechanizmus, pomocou ktorého vieme nahradiť rekurziu zásobníkom.

Zásobník sme realizovali pomocou obyčajného zoznamu (typ `list`):

- operácia `push()` pridávala nový prvok na koniec zoznamu pomocou metódy `list.append()`
- operácia `pop()` odoberala zo zoznamu posledný prvok pomocou metódy `list.pop()`
- operácia `empty()` zisťovala, či je zoznam prázdny

Zásobník sa dá veľmi elegantne realizovať aj pomocou spájaného zoznamu:

- v tomto prípade bude vrch zásobníka na začiatku zoznamu
- operácia `push()` pridá nový prvok **na začiatok** spájaného zoznamu
- operácia `pop()` odoberie prvok **zo začiatku** spájaného zoznamu
- operácia `empty()` zistí, či je spájaný zoznam prázdny
- operácia `top()` bude veľmi jednoduchá
- opäť použijeme novú výnimku `EmptyError`

Zapíšme kompletný kód:

```
class EmptyError(Exception): pass

class Stack:
    class _Vrchol:
        def __init__(self, data, next):
            self.data, self.next = data, next

    def __init__(self):
        self._zac = None

    def push(self, hodnota):
        self._zac = self._Vrchol(hodnota, self._zac)

    def pop(self):
        if self.empty():
            raise EmptyError
        vysl = self._zac.data
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        self._zac = self._zac.next
        return vysl

    def top(self):
        if self.empty():
            raise EmptyError
        return self._zac.data

    def empty(self):
        return self._zac is None
```

Vidíte, že trieda `Stack` je veľmi zjednodušená verzia triedy `SpajanyZoznam`.

27.2.1 Realizácia radu

V prednáške *25. Zásobníky a rady* sme sa zoznámili aj s ďalšou dátovou štruktúrou rad. Aj rad sme realizovali pomocou obyčajného zoznamu (typ `list`):

- operácia `enqueue()` pridávala nový prvok na koniec zoznamu pomocou metódy `list.append()`
- operácia `dequeue()` odoberala zo zoznamu prvý prvok pomocou metódy `list.pop(0)`
- operácia `empty()` zisťovala, či je zoznam prázdny

Pri realizovaní radu pomocou spájaného zoznamu sa musíme zamyslieť nad tým, či:

1. bude začiatok radu na začiatku spájaného zoznamu
    - pridávať budeme na koniec zoznamu a odoberať budeme zo začiatku
2. bude začiatok radu na konci spájaného zoznamu
    - pridávať budeme na začiatok zoznamu a odoberať budeme z konca

My už vieme, že pridávanie, resp. odoberanie zo začiatku spájaného zoznamu sú rýchle operácie. Ale pridávanie na koniec je rýchle len s pomocnou referencie na koniec zoznamu a odoberanie z konca je vždy pomalé. Preto si zvolíme variant (a), pri ktorom vieme rýchlo pridávať na koniec a rýchlo odoberať zo začiatku zoznamu:

- operácia `enqueue()` pridá nový prvok **na koniec** spájaného zoznamu (použije referenciu na posledný vrchol zoznamu)
- operácia `dequeue()` odoberie prvok **zo začiatku** spájaného zoznamu
- operácia `empty()` zistí, či je spájaný zoznam prázdny
- operácia `front()` bude veľmi jednoduchá
- opäť použijeme výnimku `EmptyError`

Zapíšme kompletný kód:

```
class EmptyError(Exception): pass

class Queue:
    class _Vrchol:
        def __init__(self, data):
            self.data, self.next = data, None

    def __init__(self):
        self._zac = self._kon = None
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def enqueue(self, hodnota):
        novy = self._Vrchol(hodnota)
        if self._zac is None:
            self._zac = self._kon = novy
        else:
            self._kon.next = novy
            self._kon = novy

    def dequeue(self):
        if self.empty():
            raise EmptyError
        vysl = self._zac.data
        self._zac = self._zac.next
        if self._zac is None:
            self._kon = None
        return vysl

    def front(self):
        if self.empty():
            raise EmptyError
        return self._zac.data

    def empty(self):
        return self._zac is None
```

Vidíte, že aj trieda `Queue` je veľmi zjednodušená verzia triedy `SpajanyZoznam`.

27.3 Operátory indexovania

Pre pythonovský obyčajný zoznam (typ `list`) sú veľmi typické operácie indexovania, t.j. keď vieme zistiť hodnotu nejakého prvku podľa jeho indexu (pozície v zozname), resp. keď vieme zmeniť hodnotu prvku zadaného indexom. Zapíšme tieto dve operácie ako metódy triedy `SpajanyZoznam`:

```
class SpajanyZoznam:
    ...

    def _ity(self, index):
        if index < 0:
            raise IndexError
        pom = self.zac
        while pom is not None and index > 0:
            pom = pom.next
            index -= 1
        if pom is None:
            raise IndexError
        return pom

    def daj_ity(self, index):
        return self._ity(index).data

    def zmen_ity(self, index, hodnota):
        self._ity(index).data = hodnota
```

Vytvorili sme pomocnú metódu `_ity()`, ktorá vráti referenciu na príslušný vrchol (alebo spadne na chybe

`IndexError`). Opäť tu prvý znak mena _ označuje, že je to pomocná metóda, ktorú by sme radšej nemali používať mimo metód samotnej triedy.

Budeme to testovať takto - najprv zapíšeme pomocou obyčajného zoznamu:

```
>>> lst = list(range(1, 20, 2))
>>> for i in range(len(lst)):
        lst[i] = lst[i] ** 2
>>> lst
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
```

a prepíšeme to pre spájaný zoznam:

```
>>> zoz = SpajanyZoznam(range(1, 20, 2))
>>> for i in range(len(zoz)):
        zoz.zmen_ity(i, zoz.daj_ity(i) ** 2)
>>> zoz
1 -> 9 -> 25 -> 49 -> 81 -> 121 -> 169 -> 225 -> 289 -> 361 -> None
```

Vidíme, že v oboch prípadoch dostávame rovnakú postupnosť čísel, len zápis `lst[i] = lst[i] ** 2` je výrazne lepšie čitateľný ako `zoz.zmen_ity(i, zoz.daj_ity(i) ** 2)`.

Hodilo by sa nám, keby sme

- namiesto volania `zoz.daj_ity(i)` mohli **zapísať** `zoz[i]` a Python by to pochopil ako vyhľadanie i-teho prvku zoznamu a vrátil by príslušnú hodnotu
- namiesto volania `zoz.zmen_ity(i, hodnota)` mohli **zapísať** `zoz[i] = hodnota` a Python by to pochopil ako vyhľadanie i-teho prvku zoznamu a zmenu jeho hodnoty

Naozaj toto v Pythone funguje. Rovnako ako sme prekryli, teda **preťažili** (overload) operácie súčtu pomocou magickej metódy `__add__()`, ako sme preťažili operáciu `in` pomocou `__contains__()`, atď. môžeme preťažiť aj **operácie indexovania**. Funguje to takto:

- keď v Pythone zapíšeme napr. `lst[i]`, toto sa Pythonom prepíše na magické volanie `lst.__getitem__(i)` a až táto magická metóda vykoná výber hodnoty
- keď v Pythone zapíšeme napr. `lst[i] = hodnota`, toto sa prepíše na magické volanie `lst.__setitem__(i, hodnota)` a až táto magická metóda vykoná zmenu hodnoty
- keď v Pythone zapíšeme napr. `del lst[i]`, toto sa prepíše na magické volanie `lst.__delitem__(i)` a až táto magická metóda vykoná zrušenie hodnoty

Môžete to otestovať:

```
>>> lst = list(range(1, 20, 2))
>>> for i in range(len(lst)):
        lst.__setitem__(i, lst.__getitem__(i) ** 2)
```

Tomuto v programovacích jazykoch hovoríme syntaktický cukor a slúži len na uľahčenie zápisu a čitateľnosť kódu. Už sme sa s tým stretli pri aritmetických operáciách, keď `a+b` často označuje `a.__add__(b)`.

Takže prepíšme naše dve metódy `daj_ity()` a `zmen_ity()` na magické `__getitem__()` a `__setitem__()`:

```
class SpajanyZoznam:
    ...

    def _ity(self, index):                # pomocná metóda
        if index < 0:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            raise IndexError
        pom = self.zac
        while pom is not None and index > 0:
            pom = pom.next
            index -= 1
        if pom is None:
            raise IndexError
        return pom

    def __getitem__(self, index):
        return self._ity(index).data

    def __setitem__(self, index, hodnota):
        self._ity(index).data = hodnota
```

Samozrejme, že môžeme písať aj bez „syntaktického cukru“:

```
>>> zoz.__setitem__(i, zoz.__getitem__(i) ** 2)
```

ale určite čitateľnejšie to bude „osladené“:

```
>>> zoz[i] = zoz[i] ** 2
```

Treba si ale pri tomto zápise uvedomiť, že pre dlhé spájané zoznamy a pre veľký index toto nevinne vyzerajúce priradenie dvakrát prelieza spájaný zoznam, aby našiel i-ty prvok. Už by sme si mohli pamätať, že napr. hľadanie posledného prvku v zozname môže naozaj trvať veľmi dlho.

27.3.1 Spájaný zoznam a for-cyklus

Python je pre programátora veľmi ústretový a snaží sa mu čo najviac vychádzať v ústrety. Keď zadefinujeme magickú metódu `__getitem__()`, Python z vlastnej iniciatívy „pochopí“, že takáto štruktúra by sa mohla dať prechádzať aj for-cyklom (iterovať). Veď zrejme mu stačí postupne indexovať s indexom 0, potom 1, potom 2, atď. až kým to nespadne na chybe a vtedy ukončí aj for-cyklus. Otestujme:

```
>>> zoz = SpajanyZoznam(x**2 for x in range(1, 20, 2))
>>> for prvok in zoz:
        print(prvok, end=' -> ')
1 -> 9 -> 25 -> 49 -> 81 -> 121 -> 169 -> 225 -> 289 -> 361 ->
```

Teda to naozaj urobí presne to, čo sme očakávali. Ale **pozor**! Tento for-cyklus pre každý prvok zoznamu prelieza celý zoznam vždy od začiatku. Pre krátke zoznamy to asi vadiť nebude, ale pre dlhé to bude neprijateľne **pomalé**!

Na rovnakom princípe potom funguje aj rozbaľovací parameter, napr.

```
>>> print(*zoz)
1 9 25 49 81 121 169 225 289 361
```

A vy už teraz viete, že tento „syntaktický cukor“ za sebou skrýva **veľmi neefektívny** kód.

27.4 Dvojsmerný a cyklický spájaný zoznam

Zatiaľ sme sa zoznámili s tzv. **jednosmerným spájaným zoznamom** (Singly Linked List). Slovo „jednosmerný“ tu označuje, že v každom vrchole sa uchováva jedna referencia na **nasledovný** vrchol. Vďaka tejto vlastnosti, vždy keď

potrebujeme zistiť predchodcu nejakého vrcholu (napr. predchodcu posledného), musíme prejsť zoznam od začiatku až po hľadaný vrchol. O tomto už vieme, že je to **drahá operácia**.

V situáciách, keď naozaj budeme často potrebovať pre nejaké vrcholy zisťovať ich predchodcov, využijeme alternatívnu organizáciu spájaných zoznamov, tzv. **dvojsmerné spájané zoznamy** (Doubly Linked List). Každý vrchol v takomto zozname si okrem referencie na nasledovníka uchováva aj referenciu na svojho predchodcu. A zrejme prvý vrchol má svojho predchodcu `None`. Zapíšme niekoľko metód:

```
class DvojsmernyZoznam:
    class Vrchol:
        def __init__(self, data, prev=None, next=None):
            self.data = data
            self.prev = prev
            self.next = next

    def __init__(self, postupnost):
        self.zac = self.kon = None
        for prvok in postupnost:
            self.append(prvok)

    def __repr__(self):
        vysl, pom = [], self.zac
        while pom is not None:
            vysl.append(repr(pom.data))
            pom = pom.next
        vysl.append('None')
        return ' <-> '.join(vysl)

    def insert0(self, hodnota):
        self.zac = self.Vrchol(hodnota, None, self.zac)
        if self.kon is None:
            self.kon = self.zac
        else:
            self.zac.next.prev = self.zac

    def append(self, hodnota):
        if self.zac is None:
            self.insert0(hodnota)
        else:
            novy = self.Vrchol(hodnota, self.kon)
            self.kon.next = novy
            self.kon = novy
```

V rámci cvičení si vyskúšate naprogramovať aj ďalšie metódy.

27.4.1 Cyklický spájaný zoznam

Spomenieme ešte jeden variant spájaných zoznamov, ktorý sa reálne využíva v niektorých špeciálnych situáciách. **Cyklický spájaný zoznam** (Circular Linked List) môže byť jednosmerný aj dvojsmerný - my sa tu budeme zaoberať len jednosmerným zoznamom. Označuje, že posledný vrchol v zozname nemá svojho nasledovníka označeného ako `None` ale nejaký iný vrchol v zozname. Najčastejšie to býva práve prvý vrchol zoznamu. Pri cyklickom zozname si musíte dať veľmi dobrý pozor na to, aby ste nevytvorili nekonečný cyklus. Zapíšme na ukážku niekoľko metód:

```
class CyklickyZoznam:
    class Vrchol:
        def __init__(self, data, next=None):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            self.data, self.next = data, next

    def __init__(self, postupnost=None):
        self.zac = self.kon = None
        if postupnost is not None:
            for hodnota in postupnost:
                self.append(hodnota)

    def __repr__(self):
        vysl, pom = [], self.zac
        while pom is not None:
            vysl.append(repr(pom.data))
            pom = pom.next
            if pom == self.zac:
                break
        vysl.append('...')
        return ' -> '.join(vysl)

    def __len__(self):
        vysl, pom = 0, self.zac
        while pom is not None:
            vysl += 1
            pom = pom.next
            if pom == self.zac:
                break
        return vysl

    def insert0(self, hodnota):
        if self.zac is None:
            self.zac = self.kon = self.Vrchol(hodnota)
            self.zac.next = self.zac
        else:
            self.kon.next = self.Vrchol(hodnota, self.zac)
            self.zac = self.kon.next

    def append(self, hodnota):
        if self.zac is None:
            self.insert0(hodnota)
        else:
            self.kon.next = self.Vrchol(hodnota, self.zac)
            self.kon = self.kon.next
```

27.5 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Poskladajte triedu `SpajanyZoznam` z prednášky s týmito metódami:
   - `__repr__()`, `__len__()`, `__contains__(hodnota)`, `insert0(hodnota)`, `append(hodnota)` (rýchla verzia), `pop0()` a `pop()`
   - triedu `SpajanyZoznam` uložte do súboru `zoznam.py` a otestujte:

```
with open('zoznam.py') as subor:
    zoz = SpajanyZoznam(subor)
for i in range(len(zoz)):
    print(repr(zoz.pop0()))
```

2. Odmerajte, koľkokrát je rýchlejšia metóda `pop0()` v porovnaní s `pop()` pre rôzne veľké zoznamy.
   - napr.

```
>>> zoz = SpajanyZoznam(range(n))  # pre n = 1000, 2000, 4000, 8000
>>> for i in range(n):
        x = zoz.pop()
```

   - na meranie času použite funkciu `time.time()`, napr. takto:

```
import time
start = time.time()
# meraný algoritmus
cas = time.time() - start
print(round(cas, 3))
```

   - vaše testovanie môže dať napr. takýto výstup:

```
pre n=1000   pop()=0.158   pop0()=0.001
pre n=2000   pop()=0.619   pop0()=0.001
pre n=4000   pop()=2.487   pop0()=0.003
pre n=8000   pop()=9.837   pop0()=0.006
```

3. Do triedy `SpajanyZoznam` zapíšte metódu `count(hodnota)`, ktorá zistí počet výskytov nejakej hodnoty v zozname
   - napr.

```
>>> zoz = SpajanyZoznam((2,5,8,6,4,5,7,6,5,2))
>>> zoz.count(5)
3
>>> zoz.count(3)
0
```

4. Zistite, čo robí táto metóda.
   - otestujte ju pre rôzne dlhé spájané zoznamy:

```
class SpajanyZoznam:
    ...

    def co_robi(self):
        pom = self.zac
        while pom and pom.next:
            pom.next = pom.next.next
            pom = pom.next
```

   - zamyslite sa nad tým, či by sa v prípade, že udržiavame referenciu na koniec `self.kon`, malo niečo v tomto kóde upraviť (aby aj po skončení tejto metódy atribút `self.kon` odkazoval na posledný prvok zoznamu)

5. Zapíšte metódu `vkladaj()`, ktorá pre spájané zoznamy čísel, ktoré majú aspoň dva prvky, pracuje takto:
   - medzi každé dva susedné vrcholy vloží nový, ktorého hodnotou je súčet týchto dvoch vrcholov, medzi ktoré sa vkladá

- otestujte napr.

```
>>> z = SpajanyZoznam(range(7))
>>> z.vkladaj()
>>> z
0 -> 1 -> 1 -> 3 -> 2 -> 5 -> 3 -> 7 -> 4 -> 9 -> 5 -> 11 -> 6 -> None
```

6. Do triedy `SpajanyZoznam` doplňte metódy z prednášky: `mapuj1()` (metóda má jeden parameter `funkcia`), `mapuj2()` (metóda má dva parametre `podmienka` aj `funkcia`) a `tolist()`
   - doplňte `...???...` a otestujte:

```
>>> zoz = SpajanyZoznam('nejaky text')
>>> zoz.mapuj1(...???...)          # prerobi samohlasky na velke ostatne na␣
↪male pismena
>>> ''.join(zoz.tolist())
'nEjAkY tExt'
>>> zoz = SpajanyZoznam(...???...)
>>> zoz.mapuj2(lambda x: x%2, lambda x: 2*x+1)
>>> zoz
8 -> 7 -> 6 -> 4 -> 3 -> 2 -> None
```

7. Do triedy `SpajanyZoznam` zadefinujte dvojicu metód bez parametrov `start()` a `dalsi()`, ktoré umožnia postupne prechádzať prvky zoznamu takto:
   - metóda `start()` inicializuje prechádzanie, t.j. umožní, aby nasledovné volanie `dalsi()` vrátilo hodnotu v prvom vrchole
   - metóda `dalsi()` pri každom svojom volaní vráti buď hodnotu vo vrchole alebo `None`, keď sa už prešli všetky vrcholy, po každom zavolaní si nastaví, že ďalšie jej volanie bude už s nasledovným vrcholom
   - pomocou týchto dvoch metód zapíšte funkcie, ktorá vypočítajú počet prvkov zoznamu, súčet prvkov, minimálnu hodnotu - parametrom týchto funkcií bude inštancia triedy `SpajanyZoznam`

```
def pocet(zoz):
    ...

def sucet(zoz):
    ...

def minimum(zoz):
    ...
```

8. Do triedy `SpajanyZoznam` dopíšte tieto metódy:
   - `reverse()` prevráti poradie prvkov v zozname (nevytvára nové vrcholy `Vrchol()`, ale len existujúcim vrcholom mení atribút `next`)
   - `min()` a `max()` vrátia príslušnú hodnotu
   - `vyber_max()` odstráni zo zoznamu maximálnu hodnotu (prvý výskyt) a vráti referenciu na tento odstránený vrchol
9. Do triedy `SpajanyZoznam` dopíšte metódu `sort()`, ktorá preusporiada zoznam od najmenšieho po najväčší - využije metódu `vyber_max()` zo (7) úlohy. Bude pracovať takto:
   - bude vytvárať nový zoznam - na začiatku bude prázdny `zoz = None`
   - kým je pôvodný zoznam neprázdny, vyberie z neho maximálny prvok (pomocou metódy `vyber_max()`) a zaradí ho na začiatok nového vytváraného zoznamu

- novovytvorený zoznam je už usporiadaný pôvodný zoznam - zapíše si ho ako svoj aktuálny (t.j. `self.zac = zoz`)

10. Doplňte metódy `__getitem__()` a `__setitem__()` z prednášky.
    - odmerajte čas pre rôzne veľké zoznamy a pre obe verzie zvyšovania hodnôt prvkov zoznamu o 1:

```
>>> zoz1 = SpajanyZoznam(range(10000))
>>> zoz1.mapuj(lambda x: x+1)

>>> zoz2 = SpajanyZoznam(range(10000))
>>> for i in range(len(zoz2)):
        zoz2[i] = zoz2[i] + 1
```

    - zamyslite sa, prečo `list(SpajanyZoznam(range(10000)))` je pomalé a `SpajanyZoznam(range(10000)).tolist()` je rýchle?
11. Do triedy `DvojsmernyZoznam` dopíšte metódy `pop()`, `pop0()`, `__delitem__(index)` a `kontrola()`
    - metóda `kontrola()` prejde celý spájaný zoznam a zistí, či sú všetky referencie `prev` a `next` korektné (t.j. napr. či platí `pom.next.prev==pom`)
    - otestujte správnosť fungovania týchto nových metód
12. Implementujte metódy triedy `Queue` pomocou cyklického spájaného zoznamu: udržiavajte jedinú referenciu a to na posledný vrchol (jeho nasledovníkom je začiatok zoznamu)
    - otestujte rýchlosť práce s takýmto radom v porovnaní s realizáciou pomocou obyčajného zoznamu (typ `list`)
    - najprv vytvoríte `n` prvkový rad a potom postupne z neho všetky prvky vyberáte - testujte napr. pre veľké `n` = 100000

27.6 3. Domáce zadanie

L.I.S.T.

- riešenie odovzdávajte na úlohový server https://list.fmph.uniba.sk/

Napíšte modul s menom `riesenie3.py`, ktorý bude obsahovať jedinú triedu s ďalšou vnorenou podtriedou a týmito metódami:

```
class SpajanyZoznam2:
    class Vrchol:
        def __init__(self, data, next=None):
            self.data, self.next = data, next

    def __init__(self, postupnost=None):
        self.zac = self.kon = None
        ...

    def append(self, hodnota):
        ...

    def count(self, hodnota):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        ...
        return 0

    def tolist(self):
        ...
        return []

    def totuple(self):
        ...
        return ()
```

Trieda `SpajanyZoznam2` implementuje **jednosmerný spájaný zoznam**. Oproti podobnej triede z prednášky táto bude tie prvky, ktoré sú typu `list` alebo `tuple`, automaticky prerábať opäť na spájaný zoznam, t.j. na typ `SpajanyZoznam2`. Takže prvkami spájaného zoznamu môžu byť hodnoty ľubovoľných typov okrem `list` a `tuple`, pričom tieto budú nahradené spájanými zoznamami.

Metódy triedy `SpajanyZoznam2` by mali mať takúto funkčnosť (môžete si dodefinovať aj ďalšie pomocné metódy):

- metóda `__init__()` okrem inicializácie atribútov `zac` a `kon` pre referencie na začiatok a koniec zoznamu, bude inicializovať samotný zoznam hodnotami vstupnej postupnosti, použije na to metódu `append()`;
- metóda `append()` pridá na koniec zoznamu nový prvok, ak je táto hodnota typu `list` alebo `tuple`, tak z nej najprv urobí zoznam (typu `SpajanyZoznam2`) a až túto prerobenú hodnotu vloží na koniec zoznamu;
- metóda `count()` zistí počet výskytov zadanej hodnoty v celom zozname, pričom, ak sú nejakými prvkami opäť spájané zoznamy, tak prvky počíta aj v týchto zoznamoch;
- metóda `tolist()` vráti všetky prvky zoznamu ako obyčajný zoznam (typu `list`), ak je niektorým prvkom spájaný zoznam (typu `SpajanyZoznam2`), tak aj tento sa do výstupného zoznamu prerobí na typ `list`;
- metóda `totuple()` vráti všetky prvky zoznamu ako pythonovskú n-ticu typu `tuple`, ak je niektorým prvkom spájaný zoznam, tak aj tento sa do výstupnej n-tice prerobí na typ `tuple`.

27.6.1 Obmedzenia

- vaše riešenie odovzdajte v súbore `riesenie3.py`, pričom sa v ňom bude nachádzať **len jedna definícia triedy** `SpajanyZoznam2`, trieda `Vrchol` bude vnorená v triede `SpajanyZoznam2`
- prvé dva riadky tohto súboru budú obsahovať:

```
# autor: Janko Hrasko
# uloha: 3. domace zadanie SpajanyZoznam2
```

- zrejme ako autora uvediete svoje meno
- atribúty `zac` a `kon` v triede `SpajanyZoznam2` musia obsahovať referencie na začiatok a koniec zoznamu
- váš program by nemal počas testovania testovačom nič vypisovať (žiadne vaše testovacie `print()`)

27.6.2 Testovanie

Keď budete spúšťať vaše riešenie na svojom počítači, môžete do súboru `riesenie3.py` pridať testovacie riadky, ktoré ale testovač vidieť nebude, napr.:

```
if __name__ == '__main__':
    z1 = SpajanyZoznam2('abc')
    print(z1.totuple())
    data = [2, z1, 'xy', [0, 1], 1]
    z = SpajanyZoznam2(data)
    print(z.tolist())
    print(z.totuple())
    print(z.count(0), z.count(1), z.count(3))
```

Tento test by vám mal vypísať:

```
('a', 'b', 'c')
[2, ['a', 'b', 'c'], 'xy', [0, 1], 1]
(2, ('a', 'b', 'c'), 'xy', (0, 1), 1)
1 2 0
```

KAPITOLA 28

28. Binárne stromy

Pripomeňme si, ako sme doteraz pracovali so spájaným zoznamom:

- spájaný zoznam sa skladal z vrcholov, ktorá boli navzájom pospájané smerníkmi (referenciami, pointermi)
- zoznam bol prístupný cez referenciu na prvý vrchol
- každý vrchol, okrem posledného mal svojho jediného nasledovníka (atribút `next`)
- (pre dvojsmerný spájaný zoznam) každý vrchol okrem prvého má v zozname svojho predchodcu - už vieme, že keď chceme zrušiť zo zoznamu nejaký vrchol, musíme meniť nasledovníka predchodcu ...

Používali sme takúto deklaráciu vrcholu:

```
class Vrchol:
    def __init__(self, data, next=None):
        self.data = data
        self.next = next
```

Na postupné prechádzanie všetkých vrcholov zoznamu nám stačil takýto jednoduchý while-cyklus:

```
pom = zac               # začiatok zoznamu
while pom is not None:
    print(pom.data)     # spracuje vrchol
    pom = pom.next
```

Túto schému prechádzania prvkov vieme využiť aj na zisťovanie počtu prvkov, na zisťovanie, či sa nejaká hodnota nachádza v zozname, na vyhadzovanie niektorých existujúcich, resp. pridávanie nových vrcholov.

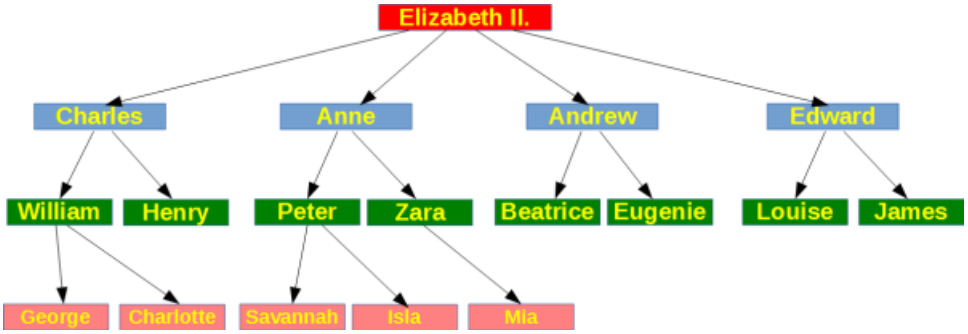
Spájané zoznamy sú najjednoduchším prípadom **spájaných štruktúr**, pre ktoré sú prvky spájané referenciami (smerníkmi).

Ďalšou spájanou dátovou štruktúrou je **strom** (tree), o ktorej sa zvykne hovoriť, že je to hierarchická dátová štruktúra.

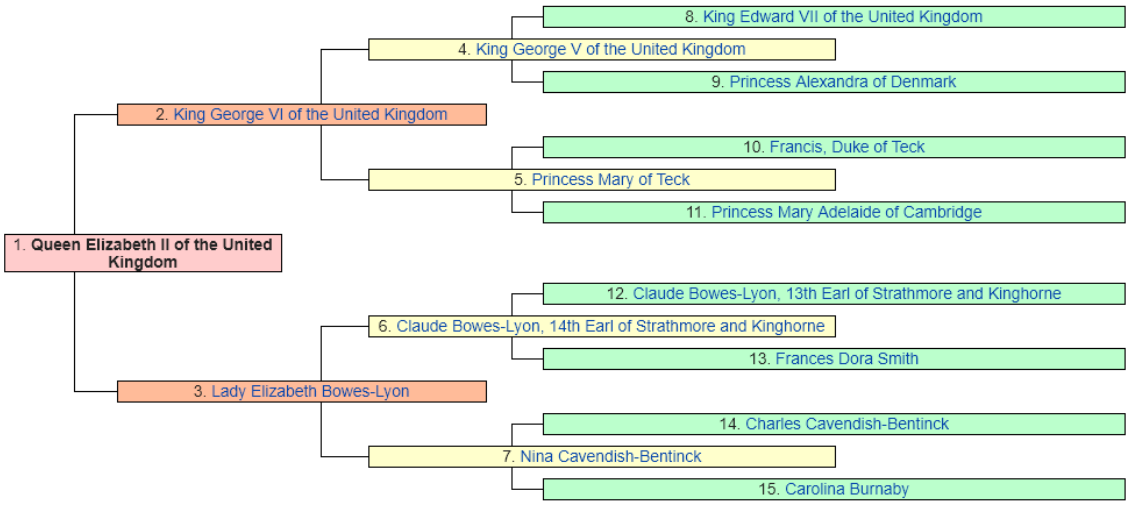
Kde všade môžeme vidieť stromovú dátovú štruktúru:

- rodokmeň, napr. potomkovia anglickej kráľovnej Alžbety II:

– z tabuľky potomkov na wikipédii môžeme pripraviť takéto grafické zobrazenie:



– veľmi zaujímavý je aj graf jej predkov na wikipédii



v tomto grafe má každá osoba (v strede vľavo je anglická kráľovná Alžbeta II.) práve dvoch rodičov: otec je horná osoba a matka je dolná

- štruktúra adresárov na disku: nejakú adresár je domovský a ten môže mať pod sebou niekoľko podadresárov, niektoré z nich môžu opäť obsahovať ďalšie podadresáre, ...
- štruktúra fakulty: naša fakulta sa skladá z troch sekcií (matematickej, fyzikálnej a informatickej) a ďalšími pracoviskami, každá sekcia sa skladá z niekoľkých katedier a väčšina katedier sa ešte delí na oddelenia
- štruktúra učebnice, manuálu: materiály sa skladajú z kapitol, tie z podkapitol a rôznych podčastí
- hierarchia tried: z bázovej triedy môžeme odvodiť niekoľko ďalších a aj z týchto odvodených tried môžeme vytvárať ďalšie triedy, napr. v prednáške *16. Triedy a dedičnosť* sme zo základnej triedy `turtle.Turtle` vytvorili triedu `MojaTurtle` a z nej sme postupne vytvárali triedy `MojaTurtle1`, `MojaTurtle2` a `MojaTurtle3`

Aj všetky tieto ďalšie štruktúry by sme vedeli zakresliť do tvaru **stromovej štruktúry**, v ktorá má nejaký počiatočný vrchol (napr. kráľovná, domovský adresár, fakulta, kniha, bázová trieda, ...) a z tohto vrcholu vychádzajú ďalšie a ďalšie vrcholy (potomkovia, podadresáre, sekcie, kapitoly, ...).

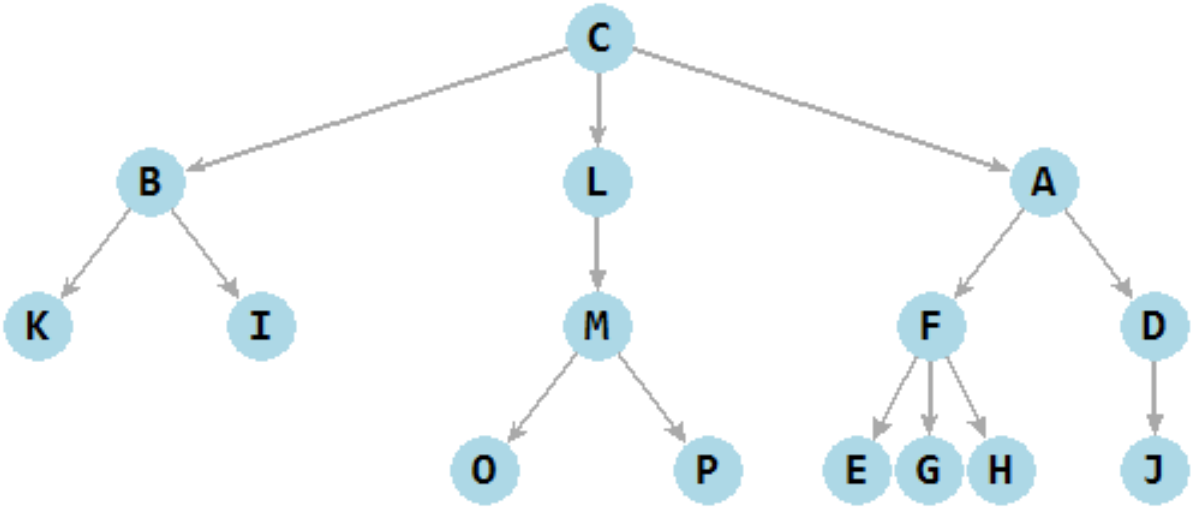
28.1 Všeobecný strom

Spájaná dátová štruktúra **strom** je zovšeobecnením spájaného zoznamu:

- skladá sa z množiny vrcholov, v každom vrchole sa nachádza nejaká informácia (atribút `data`)

- vrcholy sú navzájom pospájané hranami:
  - jeden vrchol má špeciálne postavenie: je prvý a od neho sa vieme dostať ku všetkým zvyšným vrcholom
  - každý vrchol okrem prvého má práve jedného **predchodcu**
  - každý vrchol môže mať ľubovoľný počet **nasledovníkov** (nie iba jeden ako pri spájaných zoznamoch)

Napríklad, v takomto strome:



jediný vrchol `C` nemá predchodcu, všetky ostatné vrcholy majú práve jedného predchodcu. Niektoré vrcholy (`D` a `L`) majú len jedného nasledovníka, niektoré (`A`, `B` a `M`) majú dvoch nasledovníkov, dva vrcholy `C` a `F` majú až troch nasledovníkov. Všetky zvyšné vrcholy nemajú ani jedného nasledovníka.

Zvykne sa používať takáto terminológia z **botaniky**:

- **koreň** (root) je špeciálny prvý vrchol - jediný nemá svojho predchodcu (vrchol `C`)
- **vetva** (branch) je hrana (budeme ju kresliť šípkou) označujúca nasledovníka
- **list** (leaf) označuje vrchol, ktorý už nemá žiadneho nasledovníka (listami tohto stromu sú vrcholy `E`, `G`, `H`, `I`, `J`, `K`, `O` a `P`)

Okrem botanickej terminológie sa pri stromoch používa aj takáto terminológia **rodinných vzťahov**:

- predchodca každého vrcholu (okrem koreňa) sa nazýva **otec**, resp. **rodič**, **predok** (ancestor, parent)
- nasledovník vrcholu sa nazýva **syn**, resp. **dieťa** alebo **potomok** (child, descendant)
- vrcholy, ktoré majú spoločného predchodcu (otca) sa nazývajú **súrodenci** (siblings), napr. `A`, `B`, `L` sú súrodenci, ale `I` a `M` nie sú

Ďalej sú dôležité ešte aj tieto pojmy:

- okrem listov sú v strome aj **vnútorné vrcholy** (interior nodes), to sú tie, ktoré majú aspoň jedného potomka (v našom príklade sú to `A`, `B`, `C`, `D`, `F`, `L`, `M`)
- medzi dvoma vrcholmi môžeme vytvárať **cesty**, t.j. postupnosti hrán, ktoré spájajú vrcholy - samozrejme, že len v smere od predkov k potomkom (v smere šípok) - **dĺžkou cesty** je počet týchto hrán (medzi otcom a synom je cesta dĺžky 1), napr. cesta `C - A - F - G` má dĺžku **3**
- **úroveň**, alebo **hĺbka** vrcholu (level, depth) je dĺžka cesty od koreňa k tomuto vrcholu, teda koreň má hĺbku 0, jeho synovia majú hĺbku 1, ich synovia (vnuci) ležia na úrovni 2, atď. (na 2. úrovni stromu ležia vrcholy `D`, `F`, `I`, `K`, `M`)

- **výška** stromu (height) je maximálna úroveň v strome, t.j. dĺžka najdlhšej cesty od koreňa k listom (strom v našom príklade má výšku **3**)
- **podstrom** (subtree) je časť stromu, ktorá začína v nejakom vrchole a obsahuje **všetkých** jeho potomkov (nielen synov, ale aj ich potomkov, ...), napr. strom s vrcholmi `L`, `M`, `O`, `P` je podstrom výšky 2 a podstrom s vrcholmi `E`, `F`, `G`, `H` má výšku 1
- **šírka** stromu (width) je počet vrcholov v úrovni, v ktorej je najviac vrcholov (strom v našom príklade má šírku **6**)

Niekedy sa všeobecný strom definuje aj takto rekurzívne:

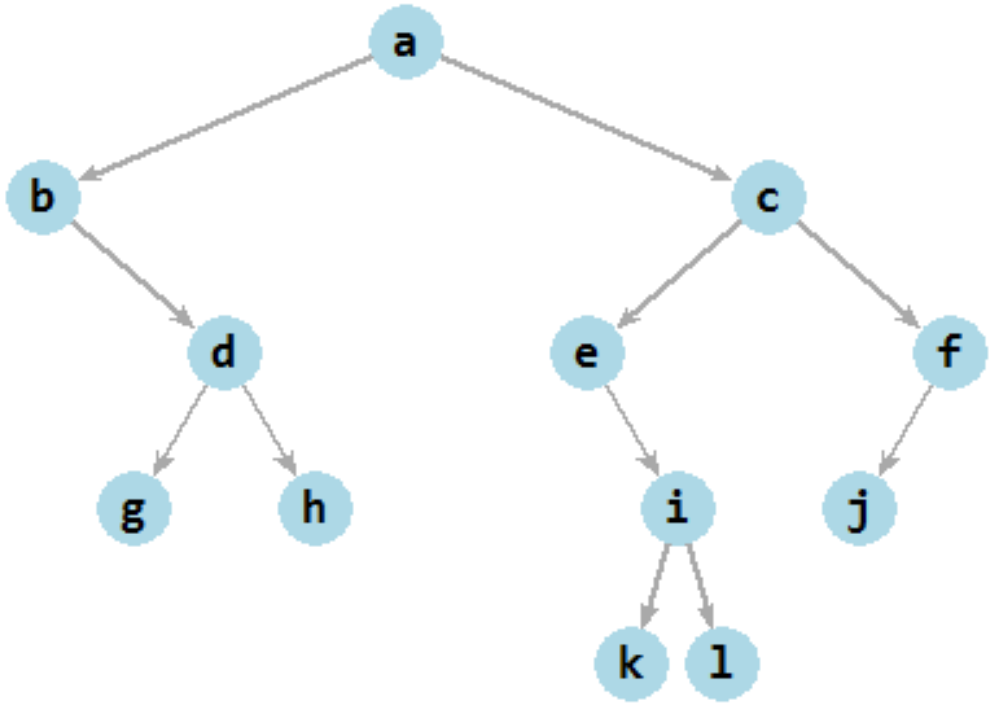
- strom je buď prázdny, alebo
- obsahuje koreň a množinu podstromov, ktoré vychádzajú z tohto koreňa, pričom každý podstrom je opäť všeobecný strom (má svoj koreň a svoje podstromy)

28.2 Binárny strom

má dve extra vlastnosti:

- každý vrchol má maximálne dvoch potomkov
- rozlišuje medzi ľavým a pravým synom - budeme to kresliť buď šípkou šikmo vľavo dole alebo šikmo vpravo dole

Napríklad, takýto binárny strom:



Vrchol `b` nemá ľavého syna, len pravého. Rovnako je na tom aj vrchol `e`. Vrchol `f` má len ľavého syna a nemá pravého. Všetky ostatné vrcholy sú buď listami (nemajú žiadneho syna) alebo majú oboch synov, ľavého aj pravého.

Aj binárny strom môžeme definovať rekurzívne:

- je buď prázdny,

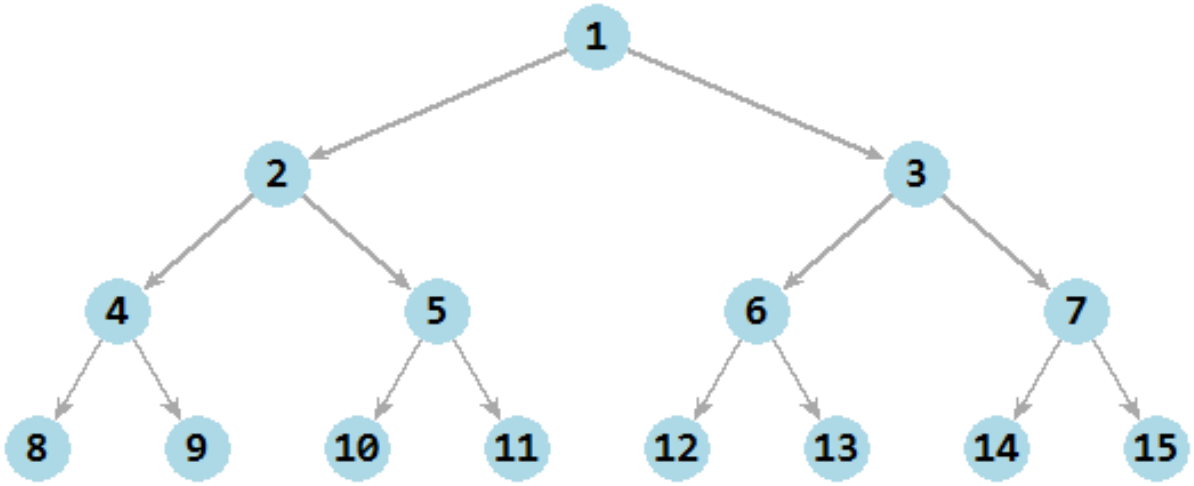
- alebo sa skladá z koreňa a z dvoch jeho podstromov: z ľavého podstromu a pravého podstromu, tieto sú opäť binárne stromy

Všetko, čo platí pre všeobecné stromy, platí aj pre binárne, t.j. má koreň, má otcov a synov, má listy aj vnútorné vrcholy, má cesty aj úrovne, hovoríme o hĺbke vrcholov aj výške stromu.

Zamyslime sa nad maximálnym počtom vrcholov v jednotlivých úrovniach:

- v 0. úrovni môže byť len koreň, teda len **1 vrchol**
- koreň môže mať dvoch synov, teda v 1. úrovni môžu byť maximálne **2 vrcholy**
- v 2. úrovni môžu byť len synovia predchádzajúcich dvoch vrcholov v 1. úrovni, teda maximálne **4 vrcholy**
- v každej ďalšej úrovni môže byť maximálne dvojnásobok predchádzajúcej, teda v i-tej úrovni môže byť maximálne **2**i** vrcholov
- strom, ktorý má výšku **h** môže mať maximálne **1 + 2 + 4 + 8 + … + 2**h** vrcholov, teda spolu **2**(h+1)-1** vrcholov
  - strom s maximálnym počtom vrcholov s výškou **h** sa nazýva **úplný strom**

Napríklad:



Tento strom má výšku **3** a preto počet všetkých vrcholov je `1 + 2 + 4 + 8 = 15`, čo je naozaj `2**4-1`

28.2.1 Realizácia spájanou štruktúrou

Binárny strom budeme realizovať podobne ako spájaný zoznam, t.j. najprv definujeme triedu `Vrchol`, v ktorej definujeme atribúty každého prvku binárneho stromu:

```
class Vrchol:
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right
```

Na rozdiel od spájaného zoznamu, tento vrchol binárneho stromu má namiesto jedného nasledovníka `next` dvoch nasledovníkov: ľavého `left` a pravého `right`. Vytvorme najprv 3 izolované vrcholy:

```
>>> a = Vrchol('A')
>>> b = Vrchol('B')
>>> c = Vrchol('C')
```

Tieto tri vrcholy môžeme chápať ako 3 jednovrcholové stromy. Pomocou priradení môžeme pripojiť stromy `b` a `c` ako ľavý a pravý podstrom `a`:

```
>>> a.left = b
>>> a.right = c
```

Takto vytvorený binárny strom má 3 vrcholy, z toho 2 sú listy a jeden (koreň) je vnútorný. Výška stromu je 1 a šírka 2. V 0. úrovni je jeden vrchol ‚A', v 1. úrovni je ‚B' a ‚C'.

Takýto istý strom vieme vytvoriť jediným priradením:

```
>>> a = Vrchol('A', Vrchol('B'), Vrchol('C'))
```

Ak chceme teraz vypísať hodnoty vo vrcholoch tohto stromu, môžeme to urobiť, napr. takto:

```
>>> print(a.data, a.left.data, a.right.data)
A B C
```

Ak by bol strom trochu komplikovanejší, výpis vrcholov by bolo dobre zautomatizovať. Napr. pre takýto strom:

```
>>> a = Vrchol('A',Vrchol('B',Vrchol(1),Vrchol(2)),Vrchol('C',None,Vrchol(3)))
```

Tento strom so 6 vrcholmi má už výšku 2, pričom v druhej úrovni sú tri vrcholy: 1, 2 a 3.

28.2.2 Nakreslenie stromu

Strom nakreslíme rekurzívne. Prvá verzia bez kreslenia čiar:

```
import tkinter
canvas = tkinter.Canvas(width=400, height=400)
canvas.pack()

def kresli(v, sir, x, y):
    if v is None:
        return
    canvas.create_text(x, y, text=v.data)
    kresli(v.left, sir/2, x-sir/2, y+40)
    kresli(v.right, sir/2, x+sir/2, y+40)

kresli(strom, 200, 200, 40)
```

Prvým parametrom funkcie je koreň stromu (teda referencia na najvrchnejší vrchol stromu). Druhým parametrom je polovičná šírka grafickej plochy. Ďalšie dva parametre sú súradnicami, kde sa nakreslí koreň stromu. Všimnite si, že v rekurzívnom volaní:

- smerom vľavo budeme vykresľovať ľavý podstrom, preto `x`-ovú súradnicu znížime o polovičnú šírku plochy, `y`-ovú zvýšime o nejakú konštantu, napr. 40
- smerom vpravo budeme kresliť pravý podstrom a teda `x`-ovú súradnicu zväčšíme tak, aby bola v polovici šírky oblasti

Ak chceme kresliť aj hrany stromu (spojnice medzi vrcholmi), zapíšeme to napr. takto (táto druhá verzia má ešte chyby):

```
import tkinter
canvas = tkinter.Canvas(width=400, height=400)
canvas.pack()

def kresli(v, sir, x, y):
    canvas.create_text(x, y, text=v.data)
    if v.left is not None:
        kresli(v.left, sir/2, x-sir/2, y+40)
        canvas.create_line(x, y, x-sir/2, y+40)
    if v.right is not None:
        kresli(v.right, sir/2, x+sir/2, y+40)
        canvas.create_line(x, y, x+sir/2, y+40)
```

Takéto vykresľovanie stromu má ale tú chybu, že nakreslené hrany (čiary) prechádzajú cez vypísané hodnoty vo vrcholoch. Opravíme to tak, že nakreslenie samotného vrcholu (bude to teraz farebný krúžok s textom) presťahujeme až na koniec funkcie, kreslenie čiar bude naopak ako prvé, aby nakreslené krúžky tieto čiary prekryli:

```
import tkinter
canvas = tkinter.Canvas(width=400, height=400)
canvas.pack()

def kresli(v, sir, x, y):
    if v.left is not None:
        canvas.create_line(x, y, x-sir/2, y+40)
        kresli(v.left, sir/2, x-sir/2, y+40)
    if v.right is not None:
        canvas.create_line(x, y, x+sir/2, y+40)
        kresli(v.right, sir/2, x+sir/2, y+40)
    canvas.create_oval(x-15, y-15, x+15, y+15, fill='white')
    canvas.create_text(x, y, text=v.data, font='consolas 12 bold')
```

Zrejme neskôr si budete túto verziu prispôsobovať:

- kresliť strom na iný rozmer plochy
- niektoré farebné krúžky môžete prefarbiť podľa obsahu, resp. polohy vrcholu
- hrany sa môžu kresliť ako šípky a nie ako úsečky

28.2.3 Generovanie náhodného stromu

Využijeme náhodný generátor `random.randrange`. Princíp tejto funkcie je nasledovný:

- pridávať budeme len do neprázdneho stromu (strom musí obsahovať aspoň koreň)
- funkcia sa najprv rozhodne, či bude pridávať do ľavého alebo do pravého podstromu (test `random.randrange(2)` testuje, či má náhodné číslo z množiny {0, 1} hodnotu 1, teda je to pravdepodobnosť 50%)
- ďalej zistí, či daný podstrom existuje (napr. `vrch.left is None` označuje, že neexistuje), ak nie, tak na jeho mieste vytvorí nový vrchol s danou hodnotou
- ak podstrom na tomto mieste už existuje, znamená to, že tu nemôžeme „zavesiť“ nový vrchol, ale musíme preň hľadať nové miesto, preto sa toto hľadanie opakuje už od nového vrcholu (nižšie uvidíme aj rekurzívnu verziu, pri ktorej sa pridávanie vrcholu do tohto podstromu zrealizuje rekurzívnym volaním):

```
import random

def pridaj_vrchol(vrch, hodnota):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    while True:
        if random.randrange(2):
            if vrch.left is None:
                vrch.left = Vrchol(hodnota)
                return
            vrch = vrch.left
        else:
            if vrch.right is None:
                vrch.right = Vrchol(hodnota)
                return
            vrch = vrch.right
```

alebo rekurzívna verzia:

```
def pridaj_vrchol(vrch, hodnota):
    if random.randrange(2):
        if vrch.left is None:
            vrch.left = Vrchol(hodnota)
        else:
            pridaj_vrchol(vrch.left, hodnota)
    else:
        if vrch.right is None:
            vrch.right = Vrchol(hodnota)
        else:
            pridaj_vrchol(vrch.right, hodnota)
```

Môžeme otestovať:

```
koren = Vrchol(random.randrange(20))
for h in range(20):
    pridaj_vrchol(koren, random.randrange(20))
```

28.2.4 Ďalšie rekurzívne funkcie

Tieto funkcie postupne (rekurzívne) prechádzajú všetky vrcholy v strome: najprv v ľavom podstrome a potom aj v pravom.

Uvádzame dve verzie súčtu hodnôt vo vrcholoch. Prvá verzia, keď zistí, že je strom neprázdny, vypočíta súčet najprv v ľavom podstrome a potom aj v pravom. Výsledok celej funkcie je potom súčet hodnoty v koreni stromu + súčet všetkých hodnôt v ľavom podstrome + súčet všetkých hodnôt v pravom podstrome:

```
def sucet(vrch):
    if vrch is None:
        return 0
    vlavo = sucet(vrch.left)
    vpravo = sucet(vrch.right)
    return vrch.data + vlavo + vpravo

print('sucet =', sucet(koren))
```

Skúsenejší programátori to zapisujú aj takto „úspornejšie“:

```
def sucet(vrch):
    if vrch is None:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        return 0
    return vrch.data + sucet(vrch.left) + sucet(vrch.right)
```

Na veľmi podobnom princípe pracuje aj zisťovanie celkového počtu vrcholov v strome:

```
def pocet(vrch):
    if vrch is None:
        return 0
    return 1 + pocet(vrch.left) + pocet(vrch.right)

print('pocet =', pocet(koren))
```

Ďalej budeme zisťovať výšku stromu (opäť bude niekoľko verzií). Najprv verzia, ktorá ešte nie je správna:

```
def vyska(vrch):
    if vrch is None:
        return 0
    vyska_vlavo = vyska(vrch.left)
    vyska_vpravo = vyska(vrch.right)
    return 1 + max(vyska_vlavo, vyska_vpravo)
```

Funkcia najprv rieši situáciu, keď je strom prázdny: vtedy dáva výsledok **0**. Inak sa vypočíta výška ľavého podstromu, potom výška pravého a na záver sa z týchto dvoch výšok vyberie ta väčšia a k tomu sa ešte pripočíta **1**, lebo celkový strom okrem dvoch podstromov obsahuje aj koreň, ktorý je o úroveň vyššie.

Táto funkcia ale nedáva správne výsledky. Keď ju otestujete, zistíte, že dostávate o **1** väčšiu hodnotu, ako je skutočná výška. Napr.

```
>>> strom = Vrchol(1)
>>> vyska(strom)
1
>>> strom = Vrchol(1, Vrchol(2), Vrchol(3))
>>> vyska(strom)
2
```

Pripomíname, že výška je počet hrán na ceste od koreňa k najnižšiemu vrcholu (k nejakému listu). Strom, ktorý obsahuje len koreň, by mal mať výšku 0 a strom, ktorý má koreň a 2 jeho synov, má dve takéto cesty (od koreňa k listom) a obe majú dĺžku 1 (len 1 hranu). Problémom v tomto riešení je výška prázdneho stromu: tá nemôže by **0**, lebo **0** je pre strom s 1 vrcholom. Dohodneme sa, že výškou prázdneho stromu bude **-1** a potom to už bude celé fungovať správne:

```
def vyska(vrch):
    if vrch is None:
        return -1
    vyska_vlavo = vyska(vrch.left)
    vyska_vpravo = vyska(vrch.right)
    return 1 + max(vyska_vlavo, vyska_vpravo)
```

alebo druhá verzia, ktorá si výšky jednotlivých podstromov nepočíta do pomocných premenných, ale priamo ich pošle do štandardnej funkcie `max()`:

```
def vyska(vrch):
    if vrch is None:
        return -1              # pre prázdny strom
    return 1 + max(vyska(vrch.left), vyska(vrch.right))

print('vyska =', vyska(koren))
```

Výpis hodnôt vo vrcholoch v nejakom poradí:

```
def vypis(vrch):
    if vrch is None:
        return
    print(vrch.data, end=' ')
    vypis(vrch.left)
    vypis(vrch.right)

vypis(koren)
```

Vo výpise sa objavia všetky hodnoty vo vrchol v nejakom poradí.

28.2.5 Metóda __repr__

Veľmi zaujímavo sa správa metóda `__repr__()` v triede `Vrchol`:

```
class Vrchol:
    def __init__(self, data, left=None, right=None):
        self.data, self.left, self.right = data, left, right

    def __repr__(self):
        return f'Vrchol({self.data}, {self.left}, {self.right})'
```

Otestujeme:

```
>>> strom = Vrchol(1, Vrchol(2, Vrchol(4)), Vrchol(3))
>>> strom
Vrchol(1, Vrchol(2, Vrchol(4, None, None), None), Vrchol(3, None, None))
```

Vidíte, že táto metóda je rekurzívna, pričom Python tu zvláda aj triviálny prípad.

28.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Do premennej `strom` sme priradili binárny strom
   - nakreslite ho na papier (bez počítača)

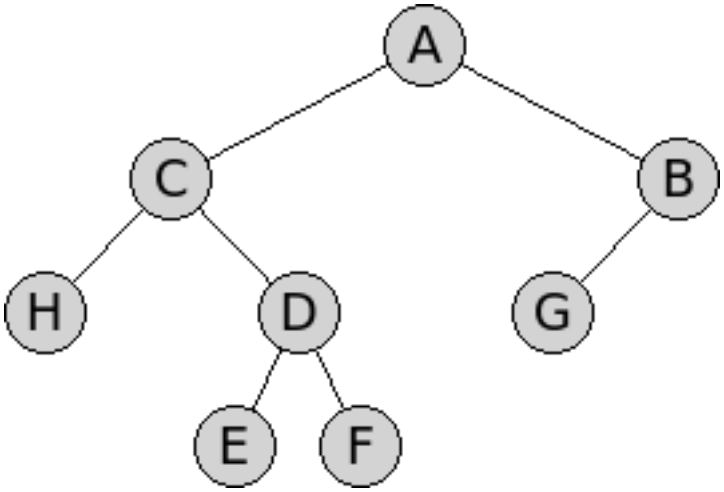
     ```
     class Vrchol:
         def __init__(self, data, left=None, right=None):
             self.data = data
             self.left = left
             self.right = right
     ```

     ```
     >>> strom = Vrchol(7,Vrchol(13,None,Vrchol(5,Vrchol(8))),Vrchol(2,Vrchol(11,
     ↪Vrchol(19)),Vrchol(3)))
     ```

   - skontrolujte vašu kresbu s vykreslením pomocou `kresli()`, napr. pomocou tejto verzie funkcie:

```
def kresli(v, sir, x, y):
    if v.left is not None:
        canvas.create_line(x, y, x-sir/2, y+40)
        kresli(v.left, sir/2, x-sir/2, y+40)
    if v.right is not None:
        canvas.create_line(x, y, x+sir/2, y+40)
        kresli(v.right, sir/2, x+sir/2, y+40)
    canvas.create_oval(x-15, y-15, x+15, y+15, fill='white')
    canvas.create_text(x, y, text=v.data, font='consolas 12')
```

2. Do premennej `strom` priraď binárny strom, ktorý po vykreslení vyzerá takto:



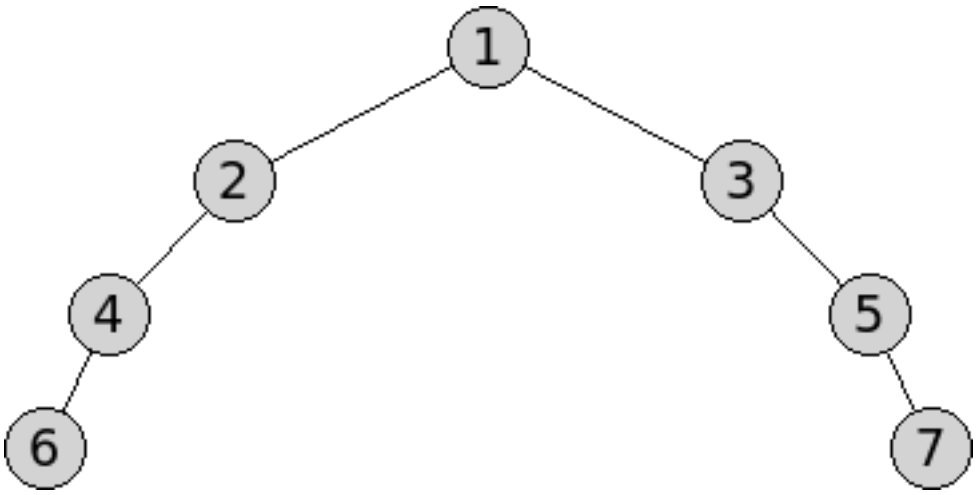
- napr.

```
strom = ...
kresli(strom, ...)
```

- Najprv bez počítača odpovedzte, aká je výška tohto stromu? Potom to skotrolujte pomocou:

```
def vyska(vrch):
    if vrch is None:
        return -1              # pre prázdny strom
    return 1 + max(vyska(vrch.left), vyska(vrch.right))
```

3. Rovnako ako predchádzajúca úloha, ale pre takýto strom:



4. Na papier nakreslite taký binárny strom, ktorý má 6 listov a v týchto listoch sú postupne zľava doprava písmená zo slova `'Python'`, zvyšné vnútorné vrcholy stromu obsahujú znak bodku `'.'`.

   - priraďte tento strom do premennej a skontrolujte:

   ```
   strom = ...
   kresli(strom, ...)
   ```

   - navrhnite taký strom, ktorého všetky listy majú rovnakú vzdialenosť od koreňa (tzv. **hĺbka** vrcholu)
   - navrhnite taký strom, ktorého všetky listy majú navzájom rôznu vzdialenosť od koreňa (napr. vrchol `'P'` bude mať vzdialenosť 1, vrchol `'Y'` vzdialenosť 2, vrchol `'T'` vzdialenosť 3, ...)

5. Napíšte funkciu `vyrob_strom(postupnost)`, ktorá vytvorí náhodný binárny strom z prvkov zadanej postupnosti. Koreňom stromu bude prvý vrchol postupnosti. Funkcia vráti referenciu na koreň stromu.

   - využite ideu tejto funkcie z prednášky:

   ```
   def pridaj_vrchol(vrch, hodnota):
       while True:
           if random.randrange(2):
               if vrch.left is None:
                   vrch.left = Vrchol(hodnota)
                   return
               vrch = vrch.left
           else:
               if vrch.right is None:
                   vrch.right = Vrchol(hodnota)
                   return
               vrch = vrch.right
   ```

   - napr. volanie `vyrob_strom('Python')` vytvorí náhodný strom so 6 vrcholmi - tento strom vykreslite a vypíšte výšku stromu aj počet vrcholov pomocou funkcie `pocet`:

   ```
   def pocet(vrch):
       if vrch is None:
           return 0
       return 1 + pocet(vrch.left) + pocet(vrch.right)
   ```

   - skontrolujte volanie `vyrob_strom(range(1000))` a vypíšte výšku tohto stromu aj počet vrcholov

6. Upravte **rekurzívnu** metódu `__repr__()` z prednášky tak, aby sa listy stromu nevypisovali v tvare `'Vrchol(hodnota, None, None)'`, ale ako `'Vrchol(hodnota)'`. Tiež opravte túto metódu tak,

aby sa reťazce ako hodnoty vypísali aj s apostrofmi

- napr.

```
>>> s = Vrchol(11, Vrchol(22, Vrchol(44)), Vrchol(33, right=Vrchol(55)))
>>> s
Vrchol(11, Vrchol(22, Vrchol(44), None), Vrchol(33, None, Vrchol(55)))
>>> Vrchol('P', Vrchol('y'), Vrchol('t'))
Vrchol('P', Vrchol('y'), Vrchol('t'))
```

7. Na papier nakreslite všetky rôzne binárne stromy, ktoré majú 3 vrcholy s hodnotami vo vrcholoch `'x'`:
   - malo by ich byť 5
   - nakreslite ich všetky vedľa seba (pomocou `kresli()`) do jednej grafickej plochy
8. Ručne zistite, čo sa vypíše
   - upravená funkcia `vypis`:

```
def vypis(vrch, k):
    if vrch is None:
        print('.'*k, None)
        return
    print('.'*k, vrch.data)
    vypis(vrch.left, k+2)
    vypis(vrch.right, k+2)

vypis(Vrchol('koren',Vrchol('lavy'),Vrchol('pravy')), 0)
```

   - skúste aj

```
vypis(Vrchol('A', Vrchol('B', None, Vrchol('D')), Vrchol('C', Vrchol('E'))),␣
↪0)
```

9. Napíšte funkciu `pocet_listov(vrch)`, ktorá vráti počet listov v strome
   - funkcia bude rekurzívna a bude riešiť tieto prípady:
     - pre prázdny strom bude výsledkom 0
     - pre vrchol, ktorý je listom (ľavý aj pravý podstrom je `None`), bude výsledkom 1
     - inak zistí počet listov v ľavom podstrome a aj počet listov v pravom podstrome, tieto dva výsledky spočíta a tento súčet je výsledným počtom listov v celom strome
   - funkciu otestujte, napr.

```
>>> strom = Vrchol('X',Vrchol('Y'),Vrchol('Z'))
>>> pocet_listov(strom)
2
>>> pocet_listov(Vrchol('X',Vrchol('Y',Vrchol('Z',Vrchol('U')))))
1
```

10. Napíšte funkciu `pocet2(vrch)`, ktorá vráti počet vrcholov v strome, ktoré majú práve 2 synov
    - funkciu otestujte aj na väčšom strome, ktorý vygenerujete náhodne

```
def pocet2(vrch):
    ...
```

11. Pozmeňte funkciu `kresli(...)` tak, aby bol koreň pri vykreslení zafarbený na červeno, všetky listy na modro a zvyšné vrcholy ostanú biele

- do funkcie môžete pridať ďalší parameter, napr.

```
def kresli(vrch, sir, x, y, koren=True):
    ...
```

12. Napíšte funkciu `nachadza_sa(vrch, hodnota)`, ktorá zistí (vráti, `True` alebo `False`), či sa v strome nachádza daná hodnota
    - riešite zrejme rekurzívne:

```
def nachadza_sa(vrch, hodnota):
    ...
```

13. Napíšte funkciu `pocet_vyskytov(vrch, hodnota)`, ktorá zistí počet výskytov danej hodnoty v strome
    - riešite zrejme rekurzívne:

```
def pocet_vyskytov(vrch, hodnota):
    ...
```

14. Napíšte funkciu `min_max(vrch)`, ktorá vráti dvojicu (`tuple`) s minimálnou a maximálnou hodnotou v strome
    - napr.

```
def min_max(vrch):
    ...

print(min_max(Vrchol(5, Vrchol(7), Vrchol(2))))

(2, 7)
```

15. Napíšte funkciu `vytvor_uplny(n)`, ktorá vráti vygenerovaný úplný strom: jeho najvyššia úroveň je `n` a hodnoty vo všetkých vrcholoch nech sú 0:
    - zrejme použijete rekurziu: úplný strom úrovne `n` sa skladá z ľavého úplného stromu úrovne `n-1`, z pravého úplného stromu tiež úrovne `n-1` a z koreňa, ktorý má tieto dva podstromy:

```
def vytvor_uplny(n):
    ...
```

16. Napíšte funkciu `mapuj(vrch, funkcia)`, ktorá zmení hodnoty vo všetkých vrcholoch stromu aplikovaním danej funkcie, t.j. pre každý vrchol zoberie hodnotu, zavolá s ňou danú funkciu a túto novú vypočítanú hodnotu vráti do vrcholu
    -

```
def mapuj(vrch, funkcia):
    ...
```

17. Napíšte funkcie `daj_pole(vrch)` a `daj_mnozinu(vrch)`, ktoré vrátia buď pole, alebo množinu všetkých hodnôt v strome
    -

```
def daj_pole(vrch):
    ...

def daj_mnozinu(vrch):
    ...
```

28.4 4. Domáce zadanie

L.I.S.T.

- riešenie odovzdávajte na úlohový server https://list.fmph.uniba.sk/

Vašou úlohou je vytvoriť súbor `riesenie4.py`, ktorý implementuje funkcie `pridaj_vrchol()`, `vytvor_strom()`, `zapis_strom_do_suboru()` a `pocet()`:

```
class Vrchol:
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right

def pridaj_vrchol(koren, riadok):
    ...

def vytvor_strom(meno_suboru):
    ...

def zapis_strom_do_suboru(koren, meno_suboru):
    ...

def pocet(koren):
    ...
```

Triedu `Vrchol` nemodifikujte a nedefinujte žiadne iné triedy.

28.4.1 Funkcie

pridaj_vrchol()

Táto funkcia dostane ako vstup koreň stromu a reťazec.

- *koreň* je buď existujúca inštancia triedy `Vrchol`, alebo `None`
- *reťazec* je ľubovoľný **string** vo formáte `'meno:cesta'`

Funkcia má rozšíriť binárny strom zadaný koreňom o nový vrchol a potom vrátiť tento strom ako výsledok funkcie. Hodnotou nového vrcholu (atribút `data`) je prvé slovo daného reťazca. Miesto, kde sa má nový vrchol pripojiť, je definované postupnosťou (reťazcom `cesta`), z ktorej nás budú zaujímať len písmená `'l'` a `'r'` (môžu byť aj veľké `'L'` a `'R'`). Začína sa od koreňa stromu. Pri písmene `'l'` sa pokračuje doľava a pri písmene `'r'` sa pokračuje doprava. Iné znaky ako písmená `'l'` a `'r'` sa ignorujú. Napr. reťazec `'meno: vpravo a VLAVO'` označuje meno vrcholu `'meno'` a z cesty sa použijú len znaky najprv `'r'` a potom `'L'`, ostatné sa odignorujú.

Cesta teda vedie

- buď k už existujúcemu vrcholu - vtedy sa hodnota tohto existujúceho vrcholu nahradí novým menom, nič iné v strome sa nezmení
- alebo k novému vrcholu, ktorý sa vytvorí aj s novou hodnotou vo vrchole; ak na ceste k tomuto vrcholu nejaký vrchol ešte neexistoval, tak sa vytvorí s hodnotou `None`

Ak cesta neexistuje (parameter `riadok` neobsahuje `':'` alebo znak `':'` je posledným znakom reťazca), nastavuje sa nová hodnota do koreňa stromu.

Funkcia vždy vráti referenciu na koreň stromu (ak parameter `koren` nebol pri volaní funkcie `None`, návratovou hodnotou bude tento pôvodný koreň stromu).

Príklad:

```
vrchol = None
vrchol = pridaj_vrchol(vrchol, 'KING') # Vrchol('KING')
vrchol = pridaj_vrchol(vrchol, 'QUEEN:L') # Vrchol('KING', Vrchol('QUEEN'))
vrchol = pridaj_vrchol(vrchol, 'PRINCE: L L L') # Vrchol('KING', Vrchol('QUEEN',␣
↪Vrchol(None, Vrchol('PRINCE'))))
vrchol = pridaj_vrchol(vrchol, 'FROG:w h y a m i f r o g') # Vrchol('KING', Vrchol(
↪'QUEEN', Vrchol(None, Vrchol('PRINCE'))), Vrchol('FROG'))
```

vytvor_strom()

Táto funkcia dostane ako vstup *meno súboru* (teda reťazec s cestou k súboru). Súbor má riadky vo formáte ako bol reťazec `riadok` z funkcie `pridaj_vrchol()`. Vašou úlohou je tento súbor spracovať a podľa riadkov v ňom vytvoriť a vrátiť nový strom (teda vráti koreň takto vytvoreného binárneho stromu).

zapis_strom_do_suboru()

Táto funkcia robí presný opak oproti funkcii `vytvor_strom()`. Na vstupe dostane *koreň* (inštancia triedy `Vrchol`) a *meno súboru* (reťazec s cestou k súboru). Úlohou funkcie je vytvoriť súbor, ktorý reprezentuje daný strom zadaný referenciou na koreň. (Pomocou takto zapísaného súboru by sa mal dať vytvoriť identický strom volaním funkcie `vytvor_strom()`)

pocet()

Táto funkcia dostane ako parameter `koren` referenciu na koreň stromu. Funkcia vráti dvojicu čísel (`tuple`): počet vrcholov stromu, ktoré nemajú v atribúte `data` hodnotu `None` a počet všetkých vrcholov stromu.

28.4.2 Obmedzenia

- vaše riešenie odovzdajte v súbore `riesenie4.py`, pričom sa v ňom bude nachádzať **len jedna definícia triedy** `Vrchol` a 4 funkcie `pridaj_vrchol`, `zapis_strom_do_suboru`, `vytvor_strom` a `pocet`
- prvé dva riadky tohto súboru budú obsahovať:

```
# autor: Janko Hrasko
# uloha: 4. domace zadanie binarny strom
```

- zrejme ako autora uvediete svoje meno
- váš program by nemal počas testovania testovačom nič vypisovať (žiadne vaše testovacie `print()`)

28.4.3 Testovanie

Keď budete spúšťať vaše riešenie na svojom počítači, môžete do súboru `riesenie4.py` pridať testovacie riadky, ktoré ale testovač vidieť nebude, napr.:

```
if __name__ == '__main__':
    s = pridaj_vrchol(None, 'a:rxl l')
    s = pridaj_vrchol(s, 'b:rl')
    s = pridaj_vrchol(s, 'c:LL')
    zapis_strom_do_suboru(s, 'strom1.txt')
    t = vytvor_strom('strom1.txt')
    print(pocet(t))
    t = pridaj_vrchol(t, 'x')
    t = pridaj_vrchol(t, 'd:' + 'lr'*10)
    zapis_strom_do_suboru(t, 'strom2.txt')
    print('subor strom1.txt')
    print(open('strom1.txt').read())
    print('subor strom2.txt')
    print(open('strom2.txt').read())
```

Tento test by vám mal vypísať:

```
(3, 6)
subor strom1.txt
c:ll
b:rl
a:rll

subor strom2.txt
a:rll
d:lrlrlrlrlrlrlrlrlrlr
c:ll
x
b:rl
```

Uvedomte si, že poradie riadkov v týchto súboroch môže byť ľubovoľne iné.

KAPITOLA 29

29. Trieda BinarnyStrom

Doteraz sme pracovali so stromom tak, že sme zadefinovali triedu `Vrchol` pre jeden vrchol stromu a k tomu sme zadefinovali niekoľko funkcií (väčšinou rekurzívnych), ktoré s pracovali s celým stromom. Napr.

```
def vypis(vrch):
    if vrch is None:
        return
    print(vrch.data, end=' ')
    vypis(vrch.left)
    vypis(vrch.right)

koren = Vrchol(...)
vypis(koren)
```

Vypíše všetky hodnoty vo vrchol stromu v nejakom poradí.

Prirodzenejšie a programátorsky správnejšie by bolo zadefinovanie triedy **Strom**, ktorá okrem definície jedného vrcholu bude obsahovať aj všetky užitočné funkcie ako svoje metódy. Teda **zapuzdrime** všetky funkcie spolu s dátovou štruktúrou do jedného „obalu“. Pre prístup ku stromu (k jeho vrcholom) si musíme pamätať referenciu na jeho koreň - to bude atribút `self.root`, ktorý bude mať pri inicializácii hodnotu `None`. Zapíšme základ:

```
class BinarnyStrom:

    #------ vnorena trieda ------

    class Vrchol:
        def __init__(self, data, left=None, right=None):
            self.data = data
            self.left = left
            self.right = right

    #------ koniec definicie vnorenej triedy ------

    def __init__(self):
        self.root = None
```

Takto sme na minulej prednáške pridávali nový vrchol na náhodnú pozíciu v strome:

```
import random

def pridaj_vrchol(vrch, hodnota):
    while True:
        if random.randrange(2):
            if vrch.left is None:
                vrch.left = Vrchol(hodnota)
                return
            vrch = vrch.left
        else:
            if vrch.right is None:
                vrch.right = Vrchol(hodnota)
                return
            vrch = vrch.right
```

Museli sme pritom predpokladať, že koreň stromu už existuje a jeho referencia je prvým parametrom funkcie. Z tejto funkcie urobíme metódu, ktorá bude fungovať aj pre prázdny strom. Zároveň doplníme inicializáciu `__init__()` o jeden parameter, vďaka čomu sa môže už pri inicializácii inštancie vytvoriť strom s náhodným umiestnením vrcholov s danými hodnotami:

```
import random

class BinarnyStrom:

    #------ vnorena trieda ------

    class Vrchol:
        def __init__(self, data, left=None, right=None):
            self.data = data
            self.left = left
            self.right = right

    #------ koniec definicie vnorenej triedy ------

    def __init__(self, postupnost=None):
        self.root = None
        if pole is not None:
            for hodnota in postupnost:
                self.pridaj_vrchol(hodnota)

    def pridaj_vrchol(self, hodnota):
        if self.root is None:
            self.root = self.Vrchol(hodnota)
        else:
            vrch = self.root
            while True:
                if random.randrange(2):
                    if vrch.left is None:
                        vrch.left = self.Vrchol(hodnota)
                        return
                    vrch = vrch.left
                else:
                    if vrch.right is None:
                        vrch.right = self.Vrchol(hodnota)
                        return
                    vrch = vrch.right
```

Takto vieme vytvárať náhodné binárne stromy s rôznym počtom vrcholov, ale zatiaľ s takýmito stromami nevieme robiť vôbec nič. Napr.

```
s1 = BinarnyStrom('Python')
s2 = BinarnyStrom([1] * 100)
s3 = BinarnyStrom(x ** 2 for x in range(10))
```

Ďalšou veľmi užitočnou funkciou pre binárny strom bola `kresli()`:

```
import tkinter
canvas = tkinter.Canvas(width=800, height=400)
canvas.pack()

def kresli(vrch, sir, x, y):
    if vrch.left is not None:
        canvas.create_line(x, y, x-sir/2, y+50)
        kresli(vrch.left, sir/2, x-sir/2, y+50)
    if vrch.right is not None:
        canvas.create_line(x, y, x+sir/2, y+50)
        kresli(vrch.right, sir/2, x+sir/2, y+50)
    canvas.create_oval(x-15, y-15, x+15, y+15, fill='white')
    canvas.create_text(x, y, text=vrch.data, font='consolas 12 bold')

strom = Vrchol(...)
kresli(strom, 200, 200, 40)
```

V tomto prípade bude treba riešiť niekoľko komplikácií:

1. funkcia `kresli()` je rekurzívna, takže sa nebude dať priamo prerobiť na metódu tak, ako sme to urobili pre `pridaj_vrchol()`
2. `canvas` je tu globálna premenná, hoci nám by sa asi viac hodil triedny atribút, ktorý by bol spoločný pre všetky inštancie
3. `canvas` by bolo vhodné inicializovať až pri prvom zavolaní metódy `kresli()` (zrejme to nesmieme inicializovať ako `self.canvas=tkinter.Canvas(...)`)
4. strom sa bude automaticky kresliť hore v strede grafickej plochy, preto bude treba predchádzajúcu kresbu v grafickej ploche vymazať, napr. pomocou `canvas.delete('all')`

Teraz to už môžeme zapísať kompletné:

```
import random
import tkinter

class BinarnyStrom:

    canvas = None
    sirka0, vyska0 = 800, 400

    #------ vnorena trieda ------

    class Vrchol:
        def __init__(self, data, left=None, right=None):
            self.data = data
            self.left = left
            self.right = right

    #------ koniec definicie vnorenej triedy ------
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def __init__(self, postupnost=None):
        self.root = None
        if pole is not None:
            for hodnota in postupnost:
                self.pridaj_vrchol(hodnota)

    def pridaj_vrchol(self, hodnota):
        if self.root is None:
            self.root = self.Vrchol(hodnota)
        else:
            vrch = self.root
            while True:
                if random.randrange(2):
                    if vrch.left is None:
                        vrch.left = self.Vrchol(hodnota)
                        return
                    vrch = vrch.left
                else:
                    if vrch.right is None:
                        vrch.right = self.Vrchol(hodnota)
                        return
                    vrch = vrch.right

    def kresli(self):

        #---- vnorena rekurzivna funkcia ----

        def kresli_rek(vrch, sir, x, y):
            if vrch.left is not None:
                self.canvas.create_line(x, y, x-sir/2, y+50)
                kresli_rek(vrch.left, sir/2, x-sir/2, y+50)
            if vrch.right is not None:
                self.canvas.create_line(x, y, x+sir/2, y+50)
                kresli_rek(vrch.right, sir/2, x+sir/2, y+50)
            self.canvas.create_oval(x-15, y-15, x+15, y+15, fill='white')
            self.canvas.create_text(x, y, text=vrch.data, font='consolas 12 bold')

        #----

        if self.canvas is None:
            BinarnyStrom.canvas = tkinter.Canvas(width=self.sirka0, height=self.
↪vyska0)
            self.canvas.pack()
        self.canvas.delete('all')
        kresli_rek(self.root, self.sirka0/2-20, self.sirka0/2, 40)
```

Všimnite si, ako sme to vyriešili:

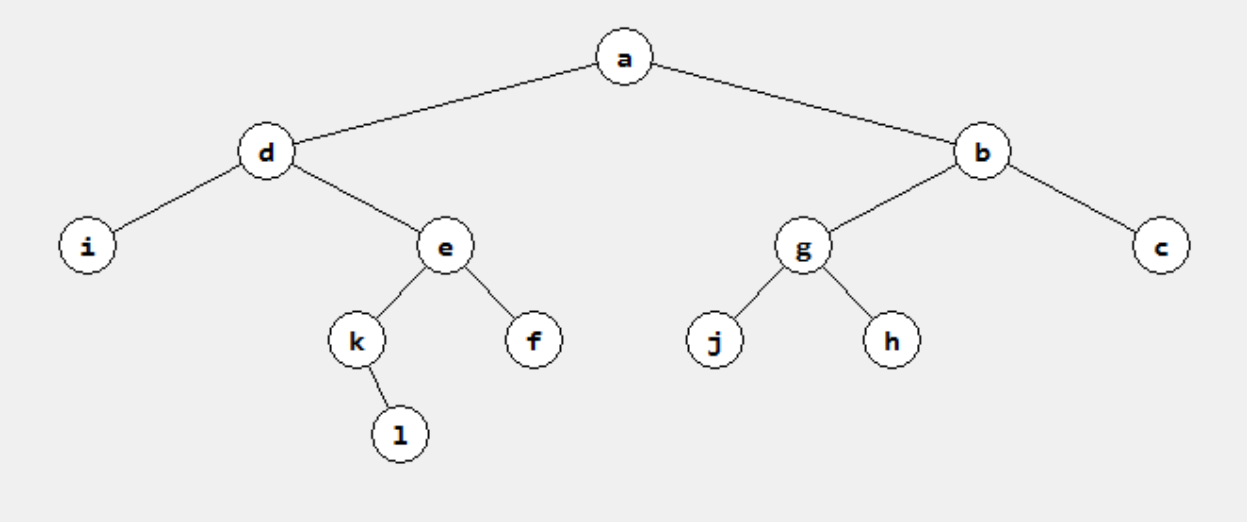
1. pôvodnú rekurzívnu funkciu sme premenovali na `kresli_rek()` a vnorili sme ju do metódy `kresli()` (tá je teraz bez parametrov)
2. `canvas` je teraz triedny atribút, inicializovaný je hodnotou `None`, okrem toho sme zadefinovali ešte dva takéto atribúty `sirka0` a `vyska0`, v ktorých je nastavená veľkosť grafickej plochy
3. pri prvom zavolaní metódy `kresli()` má atribút `canvas` zatiaľ hodnotu `None`, preto treba vytvoriť grafickú plochu `BinarnyStrom.canvas = tkinter.Canvas(...)`

4. pre každým vykresľovaním do grafickej plochy sa najprv táto vymaže pomocou `canvas.delete('all')`

Môžeme otestovať:

```
>>> s = BinarnyStrom('abcdefghijkl')
>>> s.kresli()
```

Dostávame napr.:



Na minulej prednáške sme zostavili niekoľko ďalších funkcií, napr. aj tú, ktorá počíta počet prvkov (vrcholov) stromu:

```
def pocet(vrch):
    if vrch is None:
        return 0
    return 1 + pocet(vrch.left) + pocet(vrch.right)
```

Keďže táto funkcia očakáva ako vstup vrchol, od ktorého sa počíta počet vrcholov (podstromu), musíme aj metódu, ktorá zisťuje počet prvkov, napr. `__len__()`, zapísať inak. Môžeme túto globálnu funkciu volať z metódy v triede:

```
class BinarnyStrom:

    ...

    def __len__(self):
        return pocet(self.root)
```

Toto je ale veľmi nevhodný spôsob, lebo okrem triedy sa musíme starať aj o nejakú globálnu funkciu - prípadne ich potom neskôr bude aj viac takýchto podobných. Všetky pomocné funkcie by mali byť tiež zapuzdrené v triede, najlepšie ako vnorené funkcie tam, kde sa budú používať. Opäť to urobíme podobne ako s metódou `kresli()`:

```
class BinarnyStrom:

    ...

    def __len__(self):
        #---- vnorena rekurzivna funkcia ----
        def pocet(vrch):
            if vrch is None:
                return 0
            return 1 + pocet(vrch.left) + pocet(vrch.right)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        #----
        return pocet(self.root)
```

Takýmto zápisom sa funkcia `pocet()` stáva lokálnou (vnorenou) do metódy `__len__()` a nik okrem nej ju nemôže používať. Takto to budeme najčastejšie používať pri všetkých rekurzívnych funkciách. Prepíšme niektoré ďalšie funkcie ako metódy tejto triedy:

```
def sucet(vrch):
    if vrch is None:
        return 0
    return vrch.data + sucet(vrch.left) + sucet(vrch.right)

def vyska(vrch):
    if vrch is None:
        return -1              # pre prázdny strom
    return 1 + max(vyska(vrch.left), vyska(vrch.right))

def nachadza_sa(vrch, hodnota):
    if vrch is None:
        return False
    if vrch.data == hodnota:
        return True
    if nachadza_sa(vrch.left, hodnota):
        return True
    if nachadza_sa(vrch.right, hodnota):
        return True
    return False
```

Ako metódy:

```
class BinarnyStrom:

    ...

    def sucet(self):
        #---- vnorena rekurzivna funkcia ----
        def sucet_rek(vrch):
            if vrch is None:
                return 0
            return vrch.data + sucet_rek(vrch.left) + sucet_rek(vrch.right)
        #----
        return sucet_rek(self.root)

    def vyska(self):
        #---- vnorena rekurzivna funkcia ----
        def vyska_rek(vrch):
            if vrch is None:
                return -1
            return 1 + max(vyska_rek(vrch.left), vyska_rek(vrch.right))
        #----
        return vyska_rek(self.root)

    def __contains__(self, hodnota):
        #---- vnorena rekurzivna funkcia ----
        def nachadza_sa(vrch, hodnota):
            if vrch is None:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
                return False
            return vrch.data==hodnota or nachadza_sa(vrch.left, hodnota) or nachadza_
↪sa(vrch.right, hodnota)
        #----
        return nachadza_sa(self.root, hodnota)
```

Všimnite si ako sme zjednodušili poslednú z metód, ktorá zisťuje, či sa nejaká hodnota nachádza niekde v strome:

- namiesto troch za sebou idúcich if-príkazov, sme zapísali jeden logický výraz, v ktorom sú podvýrazy spojené logickou operáciou `or` - toto označuje, že takýto výraz sa bude vyhodnocovať zľava doprava:
  - najprv `vrch.data == hodnota`: ak je to `True`, ďalšie podvýrazy sa nevyhodnocujú a celkový výsledok je `True`, inak sa spustí vyhodnocovanie `nachadza_sa(vrch.left, hodnota)`
  - `nachadza_sa(vrch.left, hodnota)` spustí rekurzívne zisťovanie, či sa daná hodnota nachádza v ľavom podstrome: ak áno celkový výsledok je `True` inak sa ešte spustí posledný podvýraz `nachadza_sa(vrch.right, hodnota)`
  - `nachadza_sa(vrch.right, hodnota)` spustí rekurzívne zisťovanie, či sa daná hodnota nachádza v pravom podstrome: ak áno celkový výsledok je `True` inak `False`

Pozrime sa ešte na parameter `hodnota` vo funkcii `nachadza_sa()`:

- do tohto parametra sme priradili hodnotu parametra metódy `__contains__()` s rovnakým menom
- pričom, keby vnorená funkcia `nachadza_sa()` nemala `hodnota` ako svoj parameter, tak by **videla** na parameter `hodnota` nadradenej funkcie `__contains__()`
- takže poučenie: keď vnorená funkcia chce používať nejakú premennú z nadradenej funkcie (kde je vnorená), nemusí ju dostať ako parameter, ale môže ju priamo používať (teda môže pracovať s jej hodnotou, ale meniť ju takto nevieme)

Zapíšme trochu vylepšenú verziu metódy `__contains__()`:

```
class BinarnyStrom:

    ...

    def __contains__(self, hodnota):
        #---- vnorena rekurzivna funkcia ----
        def nachadza_sa(vrch):
            if vrch is None:
                return False
            return vrch.data == hodnota or nachadza_sa(vrch.left) or nachadza_sa(vrch.
↪right)
        #----
        return nachadza_sa(self.root)
```

Metódu sme nazvali `__contains__()`, vďaka čomu ju môžeme volať nielen takto:

```
>>> strom.__contains__(123)
True
```

ale aj

```
>>> 123 in strom
True
```

29.1 Prechádzanie vrcholov stromu

Globálnu funkciu `vypis()`, ktorá v nejakom poradí vypisuje hodnoty vo všetkých vrcholoch:

```
def vypis(vrch):
    if vrch is None:
        return
    print(vrch.data, end=' ')
    vypis(vrch.left)
    vypis(vrch.right)
```

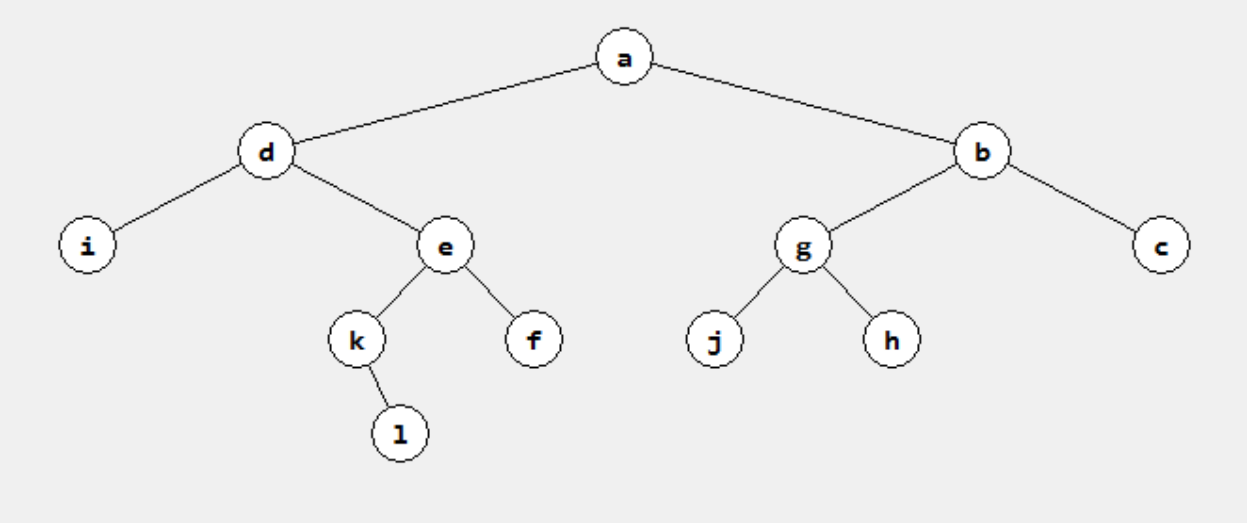
prepíšme ako metódu stromu a otestujme:

```
class BinarnyStrom:
    ...

    def vypis(self):
        #---- vnorena rekurzivna funkcia ----
        def vypis_rek(vrch):
            if vrch is None:
                return
            print(vrch.data, end=' ')
            vypis_rek(vrch.left)
            vypis_rek(vrch.right)
        #----
        vypis_rek(self.root)
        print()
```

Vytvorili sme novú metódu `vypis()`, ktorá v svojom tele opäť obsahuje len tri príkazy: definíciu vnorenej (lokálnej) funkcie `vypis_rek()`, jej zavolanie s referenciou na koreň stromu a ukončenie vypisovaného riadka. Táto metóda (podobne ako skoro všetky doterajšie) obíde všetky vrcholy v strome tak, že najprv spracuje koreň stromu (vypíše ho príkazom `print()`), potom rekurzívne celý ľavý podstrom a na záver celý pravý podstrom.

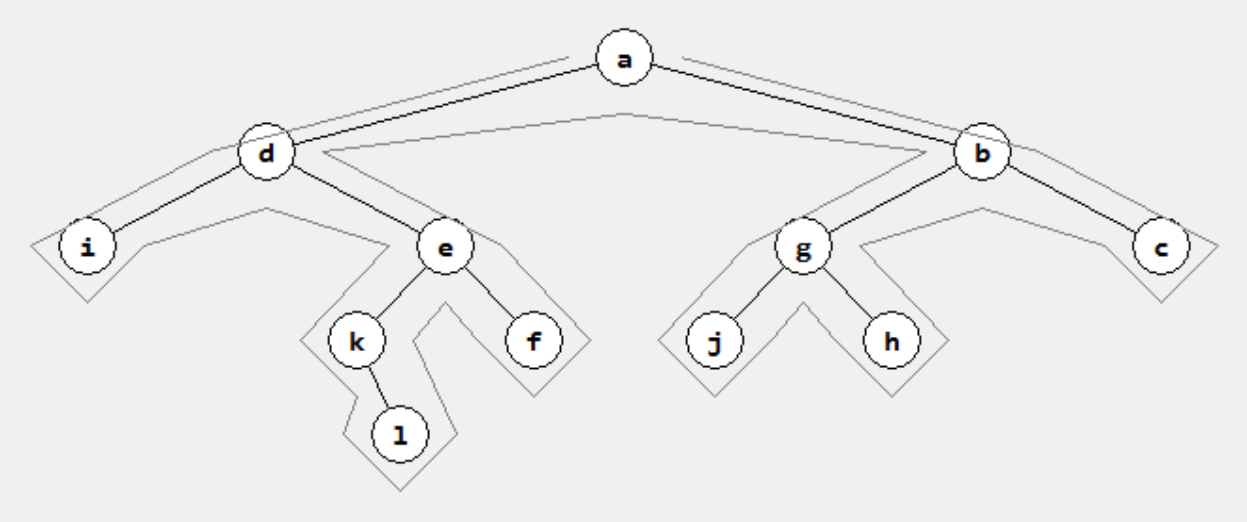
Pozrime, akú postupnosť dostávame z náhodného stromu:



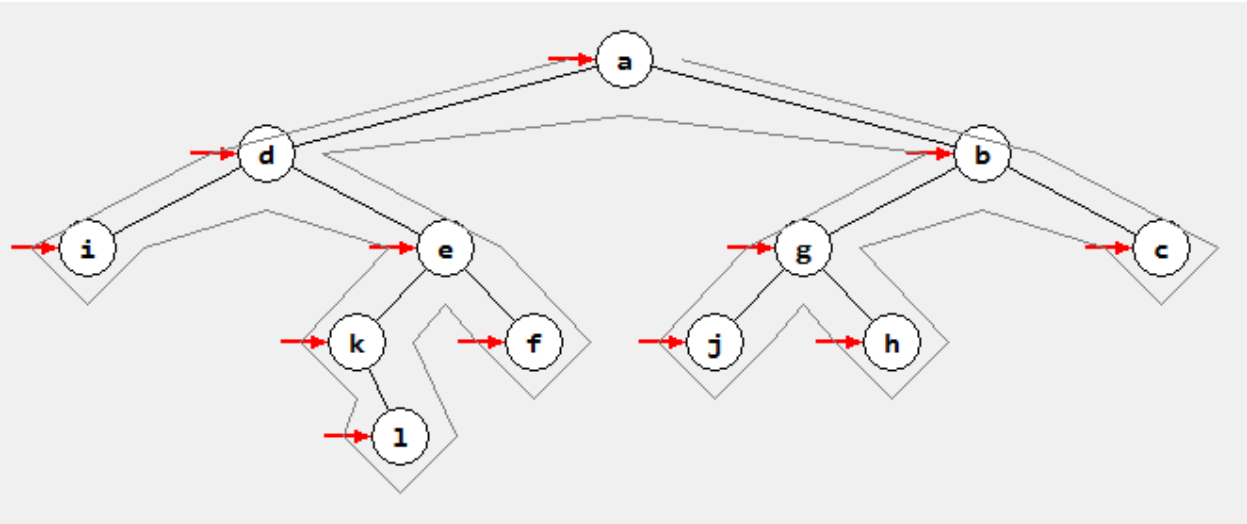

```
>>> s = BinarnyStrom('abcdefghijkl')
>>> s.kresli()
>>> s.vypis()
a d i e k l f b g j h c
```

Naozaj sa najprv vypísal koreň stromu `'a'` a pokračovalo sa vo vypisovaní celého ľavého podstromu. Tento ľavý podstrom má koreň `'d'` (vypísalo sa druhé písmeno) a opäť sa rekurzívne pokračovalo v ľavom podstrome ľavého podstromu teda s koreňom `'i'` (vypísalo sa tretie písmeno). Keďže tento podstrom už nemá žiadne ďalšie vrcholy, toto rekurzívne volanie končí a pokračuje sa s pravým podstromom stromu s koreňom `'d'`, teda so stromom s `'e'` (štvrté písmeno). Takto sa rekurzívne prejde celý strom.

Mohli by sme si graficky znázorniť trasu po ktorej sme takto prešli:



Obchádzame celý strom ale **z vonkajšej strany** pričom vždy, keď sa nachádzame **naľavo** od nejakého vrcholu, tak vypíšeme jeho hodnotu. Zakreslime si tieto prípady červenými šípkami:



Tomuto budeme hovoriť poradie spracovania **preorder**, keďže v rekurzívnej funkcii sa najprv spracuje hodnota vo vrchole a až potom sa spracovávajú ľavý a pravý podstrom. Vo všeobecnosti algoritmus **preorder poradia** zapíšeme:

```
class BinarnyStrom:
    ...

    def preorder(self):
        #---- vnorena rekurzivna funkcia ----
        def preorder_rek(vrch):
            if vrch is None:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
                return
            # spracuj samotný vrchol vrch
            preorder_rek(vrch.left)
            preorder_rek(vrch.right)
        #----
        preorder_rek(self.root)
```

alebo to isté zapísané aj trochu inak:

```
class BinarnyStrom:
    ...

    def preorder(self):
        #---- vnorena rekurzivna funkcia ----
        def preorder_rek(vrch):
            # spracuj samotný vrchol vrch
            if vrch.left is not None:
                preorder_rek(vrch.left)
            if vrch.right is not None:
                preorder_rek(vrch.right)
        #----
        if self.root is not None:
            preorder_rek(self.root)
```

Závisí od riešeného problému, ktorú z týchto verzií použijete. Všimnite si, že aj metódy `__len__()`, `vyska()`, `__contains__()`, `sucet()` boli tvaru **preorder**.

Pre nás je zaujímavý výpis z tohto preorder poradia: tento výpis závisí od tvaru binárneho stromu a bude užitočné, keď to budete vedieť nasimulovať aj ručne.

Okrem poradia **preorder** poznáme ešte tieto základné poradia obchádzania vrcholov stromu:

- **inorder** - najprv ľavý podstrom, potom samotný vrchol a na záver pravý podstrom
- **postorder** - najprv ľavý podstrom, potom pravý podstrom a na záver samotný vrchol
- **po úrovniach** - vrcholy sa spracovávajú v poradí ako sú vzdialené od koreňa, t.j. najprv koreň, potom obaja jeho synovia, potom všetci vnuci, atď.

Zapíšme prvé dve metódy:

```
class BinarnyStrom:
    ...

    def inorder_vypis(self):
        #---- vnorena rekurzivna funkcia ----
        def vypis_rek(vrch):
            if vrch is None:
                return
            vypis_rek(vrch.left)
            # spracuj samotný vrchol vrch
            print(vrch.data, end=' ')
            vypis_rek(vrch.right)
        #----
        vypis_rek(self.root)
        print()

    def postorder_vypis(self):
```

(pokračuje na ďalšej strane)

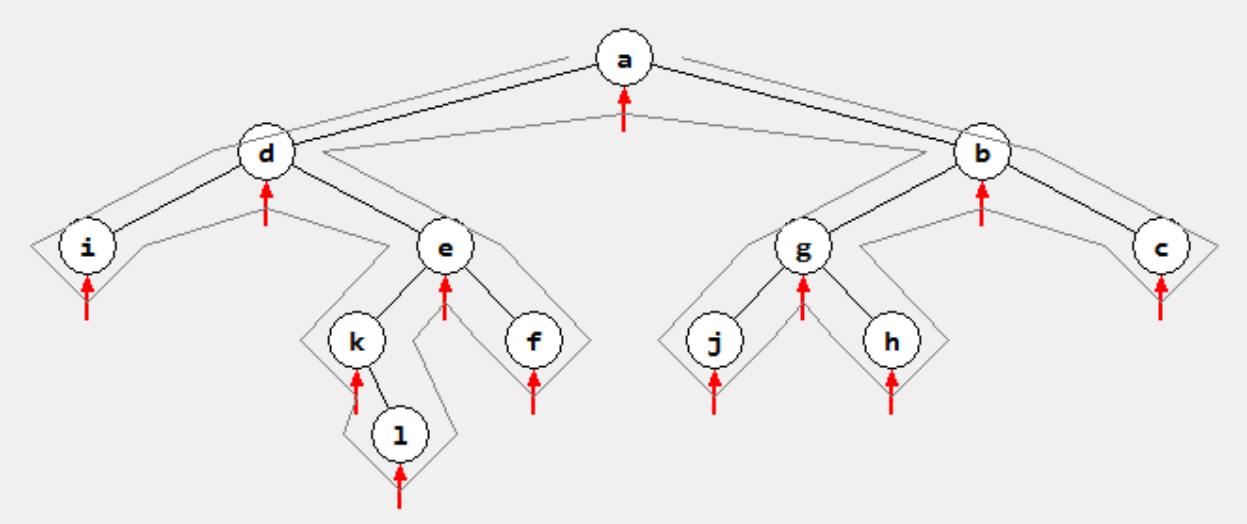
(pokračovanie z predošlej strany)

```
        #---- vnorena rekurzivna funkcia ----
        def vypis_rek(vrch):
            if vrch is None:
                return
            vypis_rek(vrch.left)
            vypis_rek(vrch.right)
            # spracuj samotný vrchol vrch
            print(vrch.data, end=' ')
        #----
        vypis_rek(self.root)
        print()
```

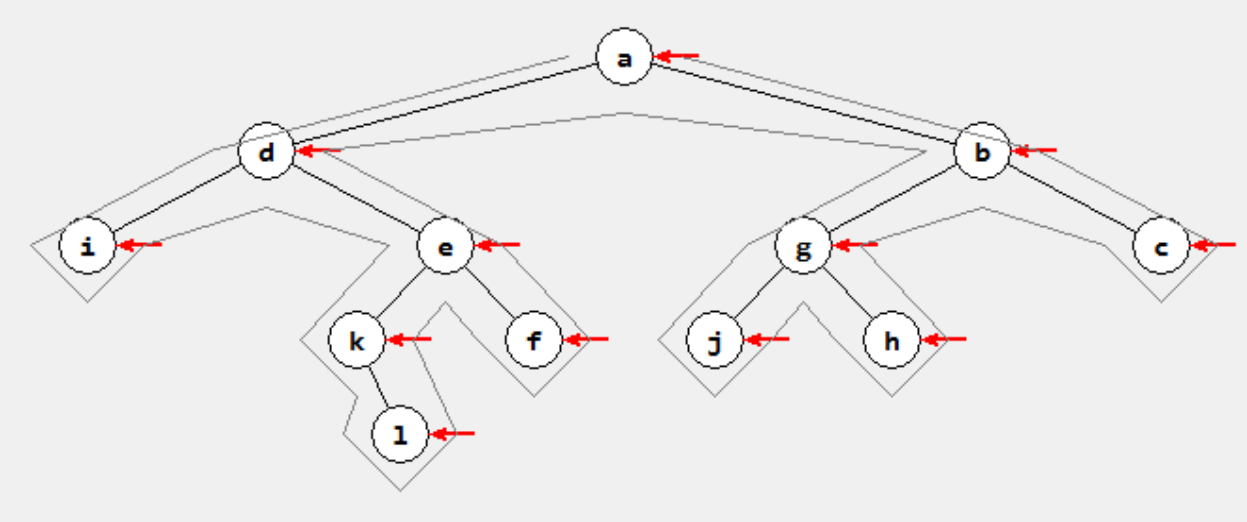
Ak otestujeme tieto výpisy pre náš náhodne generovaný strom, dostávame:

```
>>> s.inorder_vypis()
i d k l e f a j g h b c
>>> s.postorder_vypis()
i l k f e d j h g c b a
```

Všimnite si, že keď v predchádzajúcom obrázku s obchádzaním vrcholov a s červenými šípkami vľavo od vrcholov presunieme šípky pod každý vrchol, dostávame **inorder poradie**:



Podobne to bude, keď šípky presunieme vpravo od vrcholov dostaneme **postorder poradie**:



Tieto metódy by sa dali prepísať ako funkcie, ktoré vrátia znakový reťazec alebo zoznam, napr.

```
class BinarnyStrom:
    ...

    def preorder_str(self):
        #---- vnorena rekurzivna funkcia ----
        def retazec(vrch):
            if vrch is None:
                return ''
            return str(vrch.data) + ' ' + retazec(vrch.left) + retazec(vrch.right)
        #----
        return retazec(self.root)

    def preorder_list(self):
        #---- vnorena rekurzivna funkcia ----
        def urob_zoznam(vrch):
            if vrch is None:
                return []
            return [vrch.data] + urob_zoznam(vrch.left) + urob_zoznam(vrch.right)
        #----
        return urob_zoznam(self.root)
```

Tieto algoritmy postupného prechádzania všetkých vrcholov v nejakom poradí sa môžu využiť napr. na postupnú zmenu hodnôt vo vrcholoch stromu. Použijeme počítadlo, ktorého hodnotu budeme pri prechádzaní stromu priraďovať a za každým ho budeme zvyšovať o 1. Toto počítadlo ale nemôže byť obyčajná lokálna premenná, lebo ju chceme meniť počas behu rekurzie a chceme, aby sa táto zmenená hodnota zapamätala. Preto môžeme počítadlo vyrobiť ako atribút triedy, s ktorým už tento problém nebude.

Nasledovná metóda ilustruje spôsob očíslovania vrcholov v strome v poradí **preorder**, koreň bude mať hodnotu 1:

```
class BinarnyStrom:
    ...

    def ocisluj(self):
        #---- vnorena rekurzivna funkcia ----
        def ocisluj_rek(vrch):
            if vrch is None:
                return
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            vrch.data = self.cislo
            self.cislo += 1
            ocisluj_rek(vrch.left)
            ocisluj_rek(vrch.right)
        #----
        self.cislo = 1
        ocisluj_rek(self.root)
```

Pri podobných úlohách veľmi často existuje niekoľko veľmi rozdielnych spôsobov riešení. Toto isté môžeme dosiahnuť aj inak, napr. tak, že počítadlom nebude atribút (stavová premenná inštancie) ale ďalší parameter rekurzívnej vnorenej funkcie. V tomto prípade, ale budeme od tejto pomocnej funkcie vyžadovať, aby nám oznámila, koľko čísel z počítadla v danom podstrome už minula. Inými slovami, funkcia bude vracať hodnotu počítadla, na ktorom skončila pri číslovaní vrcholov v podstrome:

```
class BinarnyStrom:
    ...

    def ocisluj(self):
        #---- vnorena rekurzivna funkcia ----
        def ocisluj_rek(vrch, cislo):
            if vrch is None:
                return cislo
            vrch.data = cislo
            cislo = ocisluj_rek(vrch.left, cislo+1)
            return ocisluj_rek(vrch.right, cislo)
        #----
        ocisluj_rek(self.root, 1)
```

Takže vnorená funkcia najprv očísluje vrchol, v ktorom sa nachádza (lebo je to preorder), potom očísluje vrcholy v ľavom podstrome a dozvie sa (ako výsledok funkcie) s akým číslom sa bude pokračovať v pravom podstrome. Na záver vráti novú hodnotu počítadla ako výsledok funkcie.

29.2 Prechádzanie po úrovniach

Tento algoritmus bude používať dátovú štruktúru **queue** (rad, front) takto:

- na začiaku vyrobíme prázdny **rad** a vložíme do neho koreň stromu (čo je referencia na vrchol)
- potom sa v cykle bude robiť nasledovné:
  - z radu sa vyberie prvý čakajúci vrchol (**dequeue**)
  - spracuje sa tento vrchol, napr. sa pomocou `print()` vypíše jeho hodnota
  - na koniec frontu sa vložia (**enqueue**) obaja synovia spracovávaného vrcholu
- po skončení cyklu (práve boli spracované všetky vrcholy) sa môže urobiť nejaký záverečný úkon, napr. ukončiť rozpísaný riadok s výpisom vrcholov

Metóda, ktorá vypisuje hodnoty vo vrcholoch, ale prechádza strom po úrovniach, môže vyzerať takto:

```
class BinarnyStrom:
    ...

    def vypis_po_urovniach(self):
        if self.root is None:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            return
        q = [self.root]                 # q = Queue(); q.enqueue(self.root)
        vypis = 0
        while q != []:                  # while not q.is_empty():
            vrch = q.pop(0)             # vrch = q.dequeue()
            #spracuj
            print(vrch.data, end=' ')
            if vrch.left is not None:
                q.append(vrch.left)     # q.enqueue(vrch.left)
            if vrch.right is not None:
                q.append(vrch.right)    # q.enqueue(vrch.right)
        print()
```

V zápise funkcie môžete vidieť, aké operácie so štruktúrou rad sme nahradili operáciami s obyčajným zoznamom.

Po otestovaní, pre náš náhodný strom dostávame takéto poradie:

```
>>> s.vypis_po_urovniach()
a d b i e g c k f j h l
```

Ďalšia verzia pridáva do výpisu informáciu o konkrétnej úrovni - vrcholy, ktoré sú v rovnakej úrovni sa vypisujú do riadku a na nový riadok sa prejde vtedy, keď spracovávame vrchol z vyššej úrovne ako doteraz. Idea v tejto funkcii je taká, že do radu okrem samotného vrcholu z predchádzajúcej verzie budeme vkladať aj číslo úrovne. Preto, pri vyberaní vrcholu z radu, získavame aj jeho úroveň. Tiež musíme túto úroveň ukladať do radu, keď tam vkladáme nové vrcholy (synov momentálneho vrcholu):

```
class BinarnyStrom:
    ...

    def vypis_po_urovniach(self):
        if self.root is None:
            return
        q = [(self.root, 0)]
        vypis = 0
        while q != []:
            vrch, uroven = q.pop(0)
            #spracuj
            if vypis != uroven:
                print()
            print(vrch.data, end=' ')
            vypis = uroven
            if vrch.left is not None:
                q.append((vrch.left, uroven+1))
            if vrch.right is not None:
                q.append((vrch.right, uroven+1))
        print()
```

Obe tieto metódy sú nerekurzívne. Toto je veľmi dôležitá vlastnosť tohto algoritmu (hovorí sa mu aj **algoritmus do šírky**). Pomocou idey týchto algoritmov vieme riešiť veľkú skupinu úloh so stromami, napr.

- zisťovanie, v ktorej úrovni sa nachádza ten ktorý vrchol
- zisťovanie **šírky** stromu - čo je maximum z počtu vrcholov v jednotlivých úrovniach
- zisťovanie všetkých vrcholov, ktoré sú v jednej úrovni
- úlohy, ktoré sa dajú riešiť rekurzívne (napr. výška, počet vrcholov, apod.) ale ich nerekurzívne riešenie zabezpečí to, že takáto funkcia nespadne na pretečení rekurzie (čo je do 1000).

29.3 Cvičenia

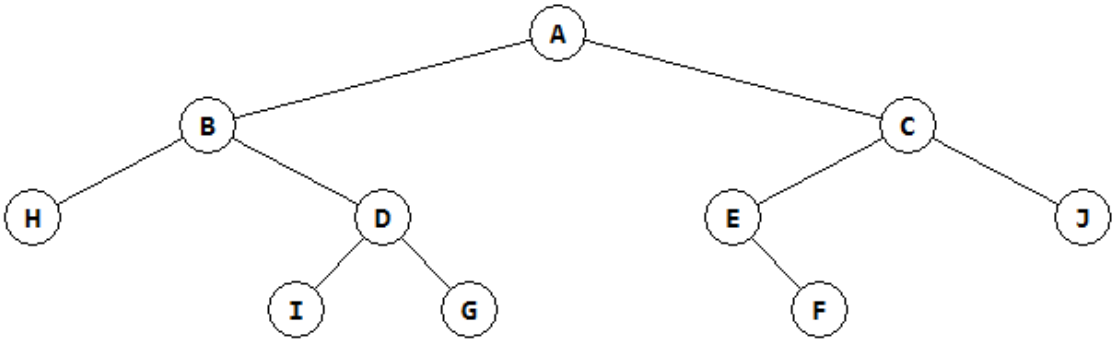
L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Zostavte triedu `BinarnyStrom` z prednášky aj so všetkými tam uvedenými metódami.
   - otestujte náhodné generovanie stromu so 16 vrcholmi s hodnotami `'x'`
   - otestujte všetky typy výpisov (preorder, inorder, postorder, po úrovniach) pre strom s 10 vrcholmi s číslami 0 až 9
   - naprogramujte metódu `count(hodnota)` - metóda vráti počet vyskytov nejakej hodnoty, napr.

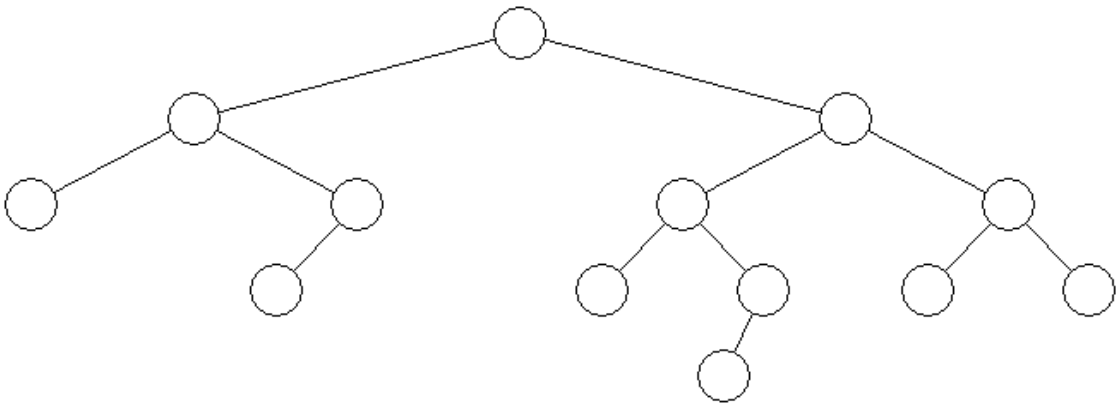
```
>>> strom = BinarnyStrom('mama ma emu a ema ma mamu')
>>> strom.count('m')
8
>>> 'mama ma emu a ema ma mamu'.count('m')
8
```

2. Pre daný strom



   - ručne vypíšte postupnosti:
     (a) **preorder**
     (b) **inorder**
     (c) **postorder**
     (d) **po úrovniach**

3. Do daného stromu



- ručne **vpíšte** hodnoty tak, aby vypisom pre postupnosť:
  (a) **preorder** bolo `'programovanie'`
  (b) **inorder** bolo `'programovanie'`
  (c) **postorder** bolo `'programovanie'`
  (d) **po úrovniach** bolo `'programovanie'`

4. Nevieme, ako vyzerá nejaký binárny strom, ale poznáme jeho inorder a postorder.
   - dané poradia:

```
inorder =  2 8 3 5 9 1 7 4 10 6
postorder =  8 2 5 3 7 1 10 6 4 9
```

   - ručne zostavte preorder
5. Ručne nakreslite všetky binárne stromy, ktoré majú 3 vrcholy a sú očíslované hodnotami 1, 2, 3 a to **po úrovniach**.
   - ku každému nakreslenému stromu vypíšte jeho **inorder**
6. Do triedy `BinarnyStrom` dopíšte metódy:
   - `mapuj(funkcia)` - zmení hodnotu v každom vrchole aplikovaním danej funkcie

```
class BinarnyStrom:

    ...

    def mapuj(self, funkcia):
        ...
        vrch.data = funkcia(vrch.data)
        ...
```

```
>>> strom = BinarnyStrom(...)
>>> strom.mapuj(lambda x: x*11)
```

   - `inorder_ocisluj(start=0, krok=1)` - očísluje všetky vrcholy celými číslami od hodnoty `start` s krokom `krok` tak, že `inorder_výpis()` vypíše usporiadanú postupost' čísel začínajúcu od zadaného štartu s daným krokom

```
class BinarnyStrom:

    ...

    def inorder_ocisluj(self, start=0, krok=1):
        ...
```

```
>>> strom = BinarnyStrom('Python')
>>> strom.inorder_ocisluj(3, 5)
>>> strom.inorder_vypis()
3 8 13 18 23 28
```

- `prirad(postupnost)` - postupne vyberá prvky postupnosti (napr. `list`, `tuple`, `str`, ...) a priraďuje ich do vrcholov stromu v poradí **preorder**; ak je prvkov v postupnosti menej ako vrcholov stromu, zvyšné vrcholy ostanú bezo zmeny

```
class BinarnyStrom:

    ...

    def prirad(self, postupnost):
        ...
```

```
>>> strom = BinarnyStrom('x'*10)
>>> strom.prirad([2, 3, 5, 7, 11, 13])
>>> strom.vypis()                          # preorder_vypis()
2 3 5 7 11 13 x x x x
```

- `brat(vrchol)` - funkcia vráti referenciu na brata zadaného vrcholu, resp. `None`, ak taký neexistuje

```
class BinarnyStrom:

    ...

    def brat(self, vrchol):
        ...
```

```
>>> strom = BinarnyStrom('python')
>>> strom.kresli()
>>> b = strom.brat(strom.root.right)
>>> b.data
???
```

- `ity(i)` - vráti hodnotu v `i`-tom vrchole (ak by sme ho prechádzali **preorderom**), nemali by ste konštruovať celú preorder postupnosť, ale treba zavolať preorder-algoritmus a ukončiť ho pri `i`-tom vrchole

```
class BinarnyStrom:

    ...

    def ity(self, i):
        ...
```

```
>>> strom = BinarnyStrom(range(10))
>>> strom.prorder_vypis()
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
...
>>> for i in range(len(strom)):
        print(strom.ity(i), end=' ')
...
```

oba výpisy vypíšu rovnakú postupnosť

- `kopia()` - vráti kópiu celého stromu, t.j. novú inštanciu triedy `BinarnyStrom`, v ktorej sa vyrobila kópia každého vrcholu pôvodného stromu

```
class BinarnyStrom:

    ...

    def kopia(self):
        ...
```

```
>>> strom = BinarnyStrom('programovanie')
>>> kopia = strom.kopia()
>>> strom.inorder_vypis()
...
>>> kopia.inorder_vypis()
...
```

oba výpisy vypíšu rovnakú postupnosť hodnôt

7. Využitím algoritmu prechádzania po úrovniach (`vypis_po_urovniach()`) naprogramujte tieto metódy:
   - `ocisluj_po_urovniach(start=0, krok=1)` - očísluje všetky vrcholy celými číslami od hodnoty `start` s krokom `krok`
   - `v_urovni(k)` - vráti zoznam (typu `list`) vrcholov (ich hodnôt) v danej úrovni, pre `k=0` zoznam obsahuje jedinú hodnotu v koreni stromu
   - `sirka()` - zistí šírku stromu

```
class BinarnyStrom:

    ...

    def ocisluj_po_urovniach(self, start=0, krok=1):
        ...

    def v_urovni(self, k):
        ...

    def sirka(self):
        ...
```

```
>>> strom = BinarnyStrom(...)
>>> strom.ocisluj_po_urovniach(1)
>>> strom.kresli()
>>> for k in range(strom.vyska()):
        print('v urovni', k, '=', strom.v_urovni(k))
v urovni 0 = [...]
v urovni 1 = [...]
v urovni 2 = [...]
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
...
>>> print('sirka =', strom.sirka())
sirka = ...
```

8. Nasledovné rekurzívne metódy prepíšte pomocou algoritmu po úrovniach na ich nerekurzívne verzie:
   - `vyska()`
   - `__len__()`
   - `sucet()`

```
class BinarnyStrom:

    ...

    def vyska_nerek(self):
        ...

    def pocet_nerek(self):
        ...

    def sucet_nerek(self):
        ...
```

```
>>> strom = BinarnyStrom(range(100))
>>> print('vyska =', strom.vyska(), strom.vyska_nerek())
vysk = ??? ???
>>> print('pocet =', strom.__len__(), strom.pocet_nerek())
pocet = 100 100
>>> print('sucet =', strom.sucet(), strom.sucet_nerek())
sucet = 4950 4950
```

29.4 5. Domáce zadanie

L.I.S.T.

- riešenie odovzdávajte na úlohový server https://list.fmph.uniba.sk/

Napíšte modul s menom `riesenie5.py`, ktorý bude obsahovať jedinú triedu s ďalšími dvomi vnorenými podtriedami a týmito metódami:

```
class FamilyTree:

    class Node:
        def __init__(self, data, left=None, right=None):
            self.data = data
            self.left = left
            self.right = right

    class Queue:
        def __init__(self):
            ...
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        def is_empty(self):
            ...

        def enqueue(self, data):
            ...

        def dequeue(self):
            ...

    # ----------------------------

    def __init__(self, meno_suboru):
        self.root = ...
        ...

    def __len__(self):
        ...

    def depth(self, data):
        ...

    def height(self):
        ...

    def width(self):
        ...

    def subtree_num(self, data):
        ...

    def descendant(self, data1, data2):
        ...

    def level_set(self, k):
        ...

    def leaves_num(self):
        ...
```

Trieda bude riešiť takúto úlohu:

- budete zostavovať rodokmeň panovníckej rodiny vymysleného kráľovstva
- keďže každý člen tejto rodiny mal maximálne dvoch potomkov, na reprezentáciu rodokmeňa použijeme binárny strom
- informácie o rodinných vzťahoch sú uložené v textovom súbore: v každom riadku je dvojica mien rodič-potomok (oddelené sú znakom ‚-‘), súbor môže vyzerať napr. takto:

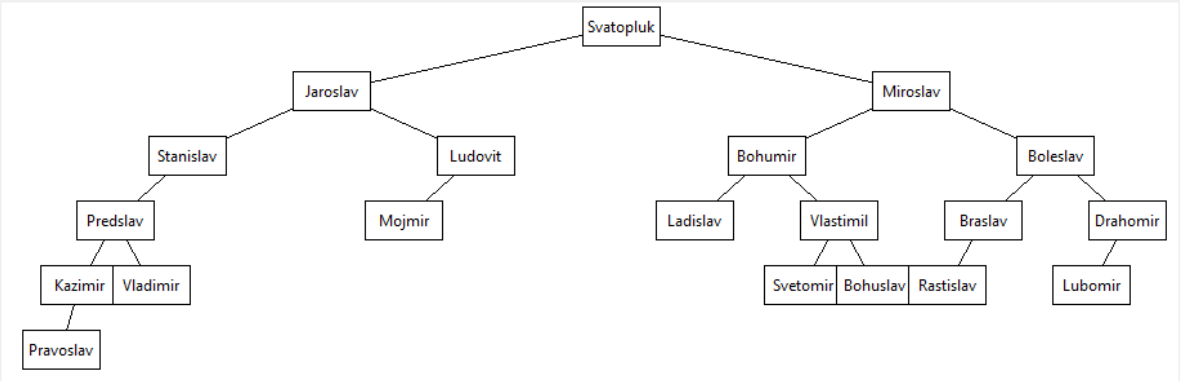
```
Predslav-Vladimir
Miroslav-Boleslav
Predslav-Kazimir
Braslav-Rastislav
Drahomir-Lubomir
Ludovit-Mojmir
Svatopluk-Miroslav
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
Stanislav-Predslav
Jaroslav-Stanislav
Kazimir-Pravoslav
Svatopluk-Jaroslav
Vlastimil-Bohuslav
Jaroslav-Ludovit
Bohumir-Ladislav
Vlastimil-Svetomir
Bohumir-Vlastimil
Miroslav-Bohumir
Boleslav-Drahomir
Boleslav-Braslav
```

- tento súbor (je to prvý testovací súbor `'subor1.txt'`) reprezentuje takýto rodokmeň (uvedomte si, že v samotnom súbore nie je informácia o tom, ktorý z potomkov je ľavý syn a ktorý pravý syn, preto správny rodokmeň je ľubovoľný strom, ktorý zodpovedá týmto dvojiciam zo súboru):



- všimnite si, že riadkov v súbore je presne o jeden menej, ako je vrcholov v strome - totiž každý vrchol okrem koreňa je niekoho potomkom a teda sa nachádza práve v jednom riadku súboru ako potomok (na druhom mieste dvojice)
- **koreňom stromu** je zrejme ten jediný vrchol, ktorý medzi potomkami chýba

Metódy triedy `FamilyTree` by mali mať túto funkčnosť (môžete si dodefinovať aj ďalšie pomocné metódy):

__init__(meno_suboru)

- prečíta zadaný textový súbor a vytvorí z neho binárny strom s koreňom v `self.root` (všetky vrcholy musia byť typu `self.Node`)
- môžete predpokladať, že súbor je zadaný korektne:
  - každý riadok obsahuje dvojicu mien oddelených znakom `'-'`
  - prvé meno v dvojici je niektorý rodič, druhé meno je jeho potomok
  - keďže tieto dvojice sú v súbore v náhodnom poradí, musíte vymyslieť nejakú ideu, ako celý strom skonštruujete (môžete využiť napr. typy `dict` a `set`)

__len__()

- vráti počet vrcholov v strome

subtree_num(data)

- nájde vrchol so zadaným údajom a vráti počet vrcholov v podstrome, ktorý začína od tohto vrcholu

  - ak sa zavolá s údajom v koreni stromu, tak vráti počet všetkých vrcholov v strome (to isté ako `len(strom)`)
  - ak sa zavolá s údajom v liste stromu, tak vráti 1 (podstrom má len tento jeden vrchol)
  - ak sa v strome zadaný údaj nenachádza, funkcia vráti 0

height()

- vráti výšku stromu

depth(data)

- vráti hĺbku vrcholu (ako ďaleko je to k vrcholu od koreňa stromu)
- ak vrchol so zadaným `data` neexistuje, metóda vráti `None`

width()

- vráti šírku stromu

descendant(data1, data2)

- zistí, či `data2` je jeden z potomkov pre vrchol `data1`
- metóda vráti `True`, ak áno, inak vráti `False`

level_set(k)

- vráti množinu všetkých údajov na zadanej úrovni:
  - pre `k==0` vráti jednoprvkovú množinu s koreňom stromu
  - pre `k==1` vráti dvojprvkovú množinu potomkov koreňa
  - ak je `k` väčšie ako počet úrovní stromu (výška), funkcia vráti prázdnu množinu

leaves_num()

- funkcia vráti počet všetkých listov stromu

Keď budete testovať vaše riešenie, môžete na koniec modulu pridať napr. takýto kód:

```
if __name__ == '__main__':
    f = FamilyTree('subor1.txt')
    print('pocet vrcholov =', len(f))
    print('podstrom pre Bohumir =', f.subtree_num('Bohumir'))
    print('podstrom pre Robert =', f.subtree_num('Robert'))
    print('vyska =', f.height())
    print('sirka =', f.width())
    print('hlbka vrcholu Vlastimil =', f.depth('Vlastimil'))
    print('Miroslav ma potomka Bohuslav =', f.descendant('Miroslav','Bohuslav'))
    print('Jaroslav ma potomka Svatopluk =', f.descendant('Jaroslav','Svatopluk'))
    print('vrcholy na urovni 2 =', f.level_set(2))
    print('vrcholy na urovni 10 =', f.level_set(10))
    print('pocet listov =', f.leaves_num())
```

Tento konkrétny test by mal vypísať:

```
pocet vrcholov = 20
podstrom pre Bohumir = 5
podstrom pre Robert = 0
vyska = 5
sirka = 6
hlbka vrcholu Vlastimil = 3
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
Miroslav ma potomka Bohuslav = True
Jaroslav ma potomka Svatopluk = False
vrcholy na urovni 2 = {'Boleslav', 'Bohumir', 'Ludovit', 'Stanislav'}
vrcholy na urovni 10 = set()
pocet listov = 8
```

Váš program sa bude testovať so 4 rôznymi súbormi:

1. súbor má 20 vrcholov
2. súbor má 63 vrcholov
3. súbor má skoro 1000 vrcholov
4. súbor má skoro 100000 vrcholov

Binárny strom pre 4. testový súbor je tak veľký, že na ňom nepobeží rekurzia. Preto bude treba všetky metódy prepísať bez použitia rekurzie (môžete použiť ideu nerekurzívnej metódy `vypis_po_urovnach()`).

V moduloch nič nevypisujte a tiež nevolajte žiadne ďalšie príkazy mimo definícií triedy.

Váš odovzdaný modul s menom `riesenie5.py` by mal začínať dvomi riadkami komentárov (s vašim menom):

```
# autor: Janko Hrasko
# uloha: 5. domace zadanie Rodokmen
```

KAPITOLA 30

30. Použitie stromov

Na minulej prednáške sme naprogramovali triedu `BinarnyStrom`, ktorá vďaka niekoľkým šikovným metódam zvláda zostrojovať binárne stromy aj s veľa tisícmi vrcholmi. Pritom v každom vrchole môže byť ľubovoľná hodnota, nielen číselná, ale aj reťazcová, prípadne by tu mohli byť aj zložitejšie dáta. Napr.

```
import random

class BinarnyStrom:
    class Vrchol:
        def __init__(self, data, left=None, right=None):
            self.data = data
            self.left = left
            self.right = right

    def __init__(self, postupnost=None):
        self.root = None
        if postupnost is not None:
            for hodnota in postupnost:
                self.pridaj_vrchol(hodnota)

    def pridaj_vrchol(self, hodnota):
        if self.root is None:
            self.root = self.Vrchol(hodnota)
        else:
            vrch = self.root
            while True:
                if random.randrange(2):
                    if vrch.left is None:
                        vrch.left = self.Vrchol(hodnota)
                        return
                    vrch = vrch.left
                else:
                    if vrch.right is None:
                        vrch.right = self.Vrchol(hodnota)
                        return
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
                    vrch = vrch.right

    def __len__(self):
        def pocet(vrch):
            if vrch is None:
                return 0
            return 1 + pocet(vrch.left) + pocet(vrch.right)
        return pocet(self.root)

    def vyska(self):
        def vyska_rek(vrch):
            if vrch is None:
                return -1
            return 1 + max(vyska_rek(vrch.left), vyska_rek(vrch.right))
        return vyska_rek(self.root)

    def __contains__(self, hodnota):
        def nachadza_sa(vrch):
            if vrch is None:
                return False
            return vrch.data == hodnota or nachadza_sa(vrch.left) or nachadza_sa(vrch.
↪right)
        return nachadza_sa(self.root)
```

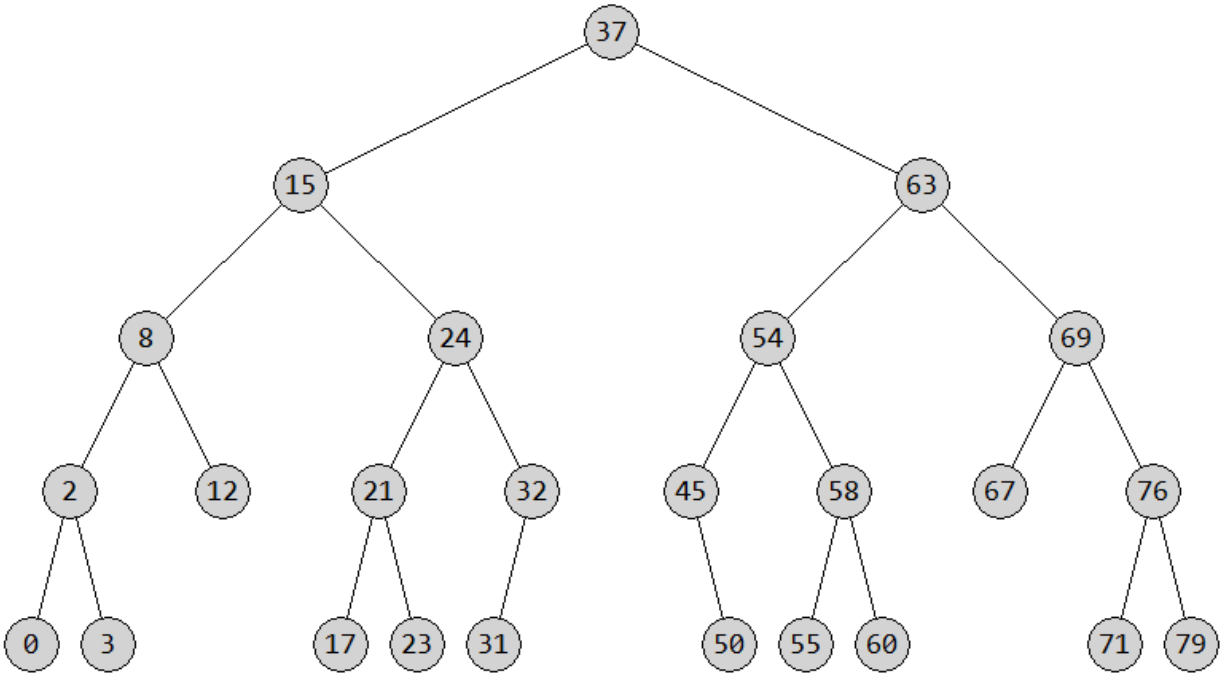
Pomocou takto definovanej dátovej štruktúry môžeme teda vytvoriť ľubovoľne veľký strom, napr.

```
>>> s = BinarnyStrom(random.randrange(10000) for i in range(10000))
>>> s.vyska()
17
>>> 5000 in s
False
```

V tomto strome sa nachádzajú náhodné čísla z intervalu <0, 9999> a aby sme zistili, či sa v ňom nachádza nejaká konkrétna hodnota, musíme postupne (najskôr rekurzívne) preliezť celý strom, teda všetkých 10000 vrcholov.

30.1 Binárny vyhľadávací strom

Pozrime sa ale na takýto binárny strom, ktorý obsahuje len 25 vrcholov a jeho hĺbka je len 4:



Skúsme popísať asi akú vlastnosť majú hodnoty vo vrcholoch tohoto stromu:

- vo všetkých vrcholoch sú rôzne hodnoty - sú to nejaké celé čísla
- v koreni stromu je číslo `37`
- v ľavom podstrome sú všetky hodnoty menšie ako `37`
- v pravom podstrome sú všetky hodnoty väčšie ako `37`
- teda, ak budeme v tomto strome hľadať nejakú hodnotu, nemusíme pritom prechádzať celý strom, ale len polovicu, podľa toho, či je hľadaná hodnota menšia ako číslo v koreni alebo väčšia

Predpokladajme, že hľadáme číslo `21` - zrejme stačí pozerať ľavý podstrom (tu je jeho koreň `15`). Ak by aj tento náš podstrom mal túto istú vlastnosť (vľavo sú len menšie a vpravo len väčšie), nemuseli by sme preliezať celý tento podstrom, ale stačilo by len tú časť, ktorú zistíme po porovnaní s koreňom: keďže hľadané je `21` a to je väčšie ako koreň podstromu `15`, stačí túto hodnotu hľadať v pravom podstrome od vrcholu `15`.

Keďže tento ukážkový strom je skonštruovaný tak, že vlastnosť „naľavo menšie a napravo väčšie“ platí pre **každý vrchol stromu**, môžeme hľadanie v takomto strome zapísať takto:

1. do `vrch` daj koreň stromu
2. ak je hodnota vo `vrch` zhodná s hľadanou hodnotou, môžeš skončiť (**našiel**)
3. ak je hodnota vo `vrch` väčšia ako hľadaná, bude treba pokračovať **v ľavom podstrome**, teda zmeň `vrch` na ľavý podstrom
4. inak sa hľadaná hodnota nachádza v pravom podstrome, teda zmeň `vrch` **na pravý podstrom**
5. ak `vrch` nie je `None` pokračuj od (2), inak skonči (**nenašiel**)

Tento algoritmus je zaujímavý tým, že

- nie je rekurzívny
- opakuje sa v ňom nejaká časť: kým nenájdeme hľadanú hodnotu, alebo neprídeme do listu stromu

- teda netreba hľadanú hodnotu porovnávať so všetkými hodnotami stromu, ale len s hodnotami na nejakej ceste smerom k listu stromu
- maximálny počet porovnaní (prechodov cyklu) bude teda závisieť od **výšky stromu**, čo je v našom prípade 4

Binárny strom, ktorý má vlastnosť, že **pre každý vrchol** stromu platí

- všetky vrcholy v ľavom podstrome sú menšie ako hodnota v koreni
- všetky vrcholy v pravom podstrome sú väčšie ako hodnota v koreni

sa nazýva binárny vyhľadávací strom.

Zapíšme základ definície triedy `BVS`, ktorá bude mať zatiaľ jedinú metódu na hľadanie hodnoty v strome:

```
class BVS:
    class Vrchol:
        def __init__(self, data, left=None, right=None):
            self.data = data
            self.left = left
            self.right = right

    def __init__(self):
        self.root = None

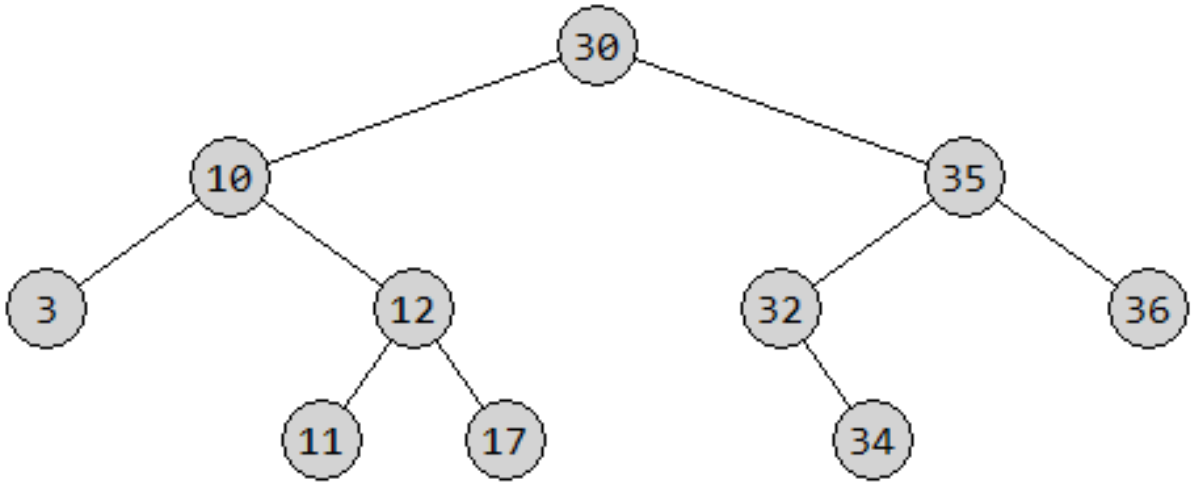
    def hladaj(self, hodnota):
        vrch = self.root
        while vrch is not None:
            if vrch.data == hodnota:
                return True
            if vrch.data > hodnota:
                vrch = vrch.left
            else:
                vrch = vrch.right
        return False
```

Môžete vidieť, že táto nerekurzívna metóda `hladaj()` je veľmi jednoduchá a preto dobre čitateľná. Horšie je, že zatiaľ to nevieme otestovať, lebo nevieme nejako jednoducho skonštruovať binárny strom, ktorý by bol správny **BVS**.

Vytvorme najprv BVS, priamym priradením vrcholov do `root`, napr. takto:

```
v = BVS.Vrchol
s = BVS()
s.root = v(30, v(10, v(3), v(12, v(11), v(17))), v(35, v(32, None, v(34)), v(36)))
```

Zápis `v = BVS.Vrchol` tu označuje to, že si vyrobíme skratku `v` pre volanie `BVS.Vrchol`. Takto vytvorený strom má tento tvar:



Vidíme, že naozaj každý vrchol stromu spĺňa podmienku **BVS**. Môžeme otestovať:

```
>>> s.hladaj(15)
False
>>> s.hladaj(17)
True
>>> s.hladaj(30)
True
>>> s.hladaj(35)
True
>>> s.hladaj(33)
False
```

Teraz sa zamyslime nad tým, ako môžeme zautomatizovať vytváranie BVS. Predpokladajme, že už máme nejaký BVS a chceme do neho pridať nový vrchol, ale tak, aby aj tento nový strom bol BVS - teda chceme, aby sa nepokazilo pravidlo BVS. Ak by sme chceli do nášho malého ukážkového stromu pridať, napr. hodnotu `7` a všetky existujúce vrcholy chceme pritom nechať na svojom mieste, budeme hľadať (podobne ako metóda `hladaj()`) miesto, kde by sme očakávali umiestnenie takejto hodnoty:

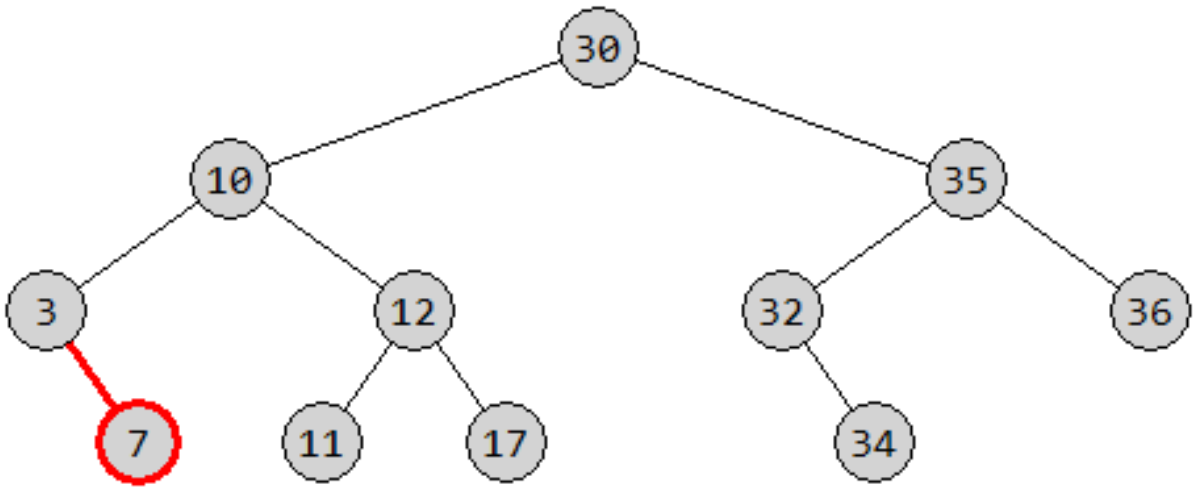
- keďže `7` nie je v koreni stromu, budeme pokračovať vľavo, lebo `7` je menšie ako `30`
- tu má ľavý podstrom svoj koreň `10`, čo je opäť viac ako `7`, preto sa opäť presúvame vľavo
- ľavý podstrom vrcholu `10` je strom s koreňom `3` (ten je už bez synov) - keďže `3` je menej ako `7`, pridávaná hodnota bude ležať v pravom podstrome `3`
- keďže pravý podstrom vrcholu `3` neexistuje, tu je presne to miesto, kde by sme mohli pripojiť nový vrchol s hodnotou `7`

Zapíšme tento postup do algoritmu:

1. ak koreň neexistuje, vyrobí sa nový koreň s vkladanou hodnotou
2. inak označme koreň stromu ako `vrch` a hľadajme miesto na pridanie nového vrcholu
3. skontrolujme, či pridávaná hodnota sa nenachádza v koreni, teda `vrch.data`, ak áno, skončíme (nebolo treba pridávať)
4. ak je pridávaná hodnota menšia ako hodnota v koreni, budeme kontrolovať ľavý smer:
    - ak ľavý syn vrcholu `vrch` neexistuje, na toto miesto vytvoríme nový vrchol a skončíme
    - inak zmeníme `vrch` na ľavý podstrom a pokračujeme v (3)

5. pridávaná hodnota je väčšia ako hodnota v koreni, preto budeme kontrolovať pravý smer:
   - ak pravý syn vrcholu `vrch` neexistuje, na toto miesto vytvoríme nový vrchol a skončíme
   - inak zmeníme `vrch` na pravý podstrom a pokračujeme v (3)

V strome to bude vyzerať takto:



Podobne ako metóda `hladaj()` aj táto je **nerekurzívna**:

```
class BVS:
    ...

    def vloz(self, hodnota):
        if self.root is None:
            self.root = self.Vrchol(hodnota)
        else:
            vrch = self.root
            while vrch.data != hodnota:
                if vrch.data > hodnota:
                    if vrch.left is None:
                        vrch.left = self.Vrchol(hodnota)
                        break
                    vrch = vrch.left
                else:
                    if vrch.right is None:
                        vrch.right = self.Vrchol(hodnota)
                        break
                    vrch = vrch.right
```

Pomocou tejto metódy môžeme BVS z predchádzajúceho príkladu vytvoriť napr. takto:

```
s = BVS()
for i in 30, 10, 35, 3, 12, 32, 36, 11, 17, 34:
    s.vloz(i)
```

30.1.1 Užitočné vlastnosti BVS

- **inorder** postupnosť je vždy **vzostupne utriedená** postupnosť všetkých hodnôt v strome

- **preorder** postupnosť jednoznačne určuje tvar BVS - ak vytvoríme nový strom s vkladaním prvkov v poradí preorder postupnosti, dostávame identický strom
- **minimálny prvok** je najľavejší v celom strome, t.j. ak pôjdeme od koreňa po ľavých synoch, tak posledný v trase je minimálny prvok
- **maximálny prvok** je najpravejší v celom strome, t.j. ak pôjdeme od koreňa po pravých synoch, tak posledný v trase je maximálny prvok
- ľubovoľný prvok v strome sa dá nájsť za maximálne toľko krokov, ako je **výška stromu**

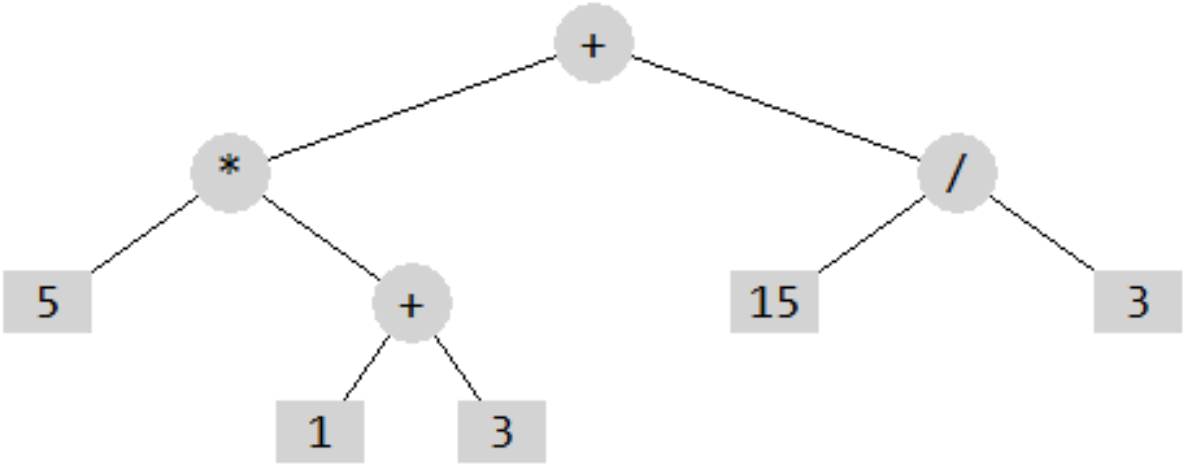
30.1.2 Degenerovaný BVS

Ak by sme vytvárali BVS **z utriedenej postupnosti** (postupným volaním metódy `vloz()`), dostávame strom, ktorý obsahuje, napr. len pravých synov. Strom sa zdegeneroval na spájaný zoznam - výška stromu je vlastne počet prvkov mínus 1 - v takomto strome potom hľadanie prvku prechádza celý strom. Degenerovaným stromom pomenujeme aj také BVS, ktoré majú veľkú výšku, napr. `n/2` kde `n` je počet prvkov stromu.

Ideálne by bolo, keby bol BVS **skoro úplný**, v ktorom väčšina listov má rovnakú hĺbku (rovnajúcu sa výške stromu). Vtedy je vyhľadávanie prvku veľmi rýchle a netreba na neho viac porovnaní ako je výška stromu, čo je pre skoro úplný strom približne `log2 n`. Vyrobiť takýto skoro úplný strom je ale už náročnejšia práca.

30.2 Aritmetický strom

Takýto binárny strom by pre nás mohol byť dobre čitateľný:



V tomto strome sú vo všetkých vnútorných vrcholoch nejaké **aritmetické operácie** a v listoch sú nejaké čísla (zrejme **operandy**). Strom na obrázku by mohol zodpovedať takémuto aritmetickému výrazu:

```
5 * (1 + 3) + 15 / 3
```

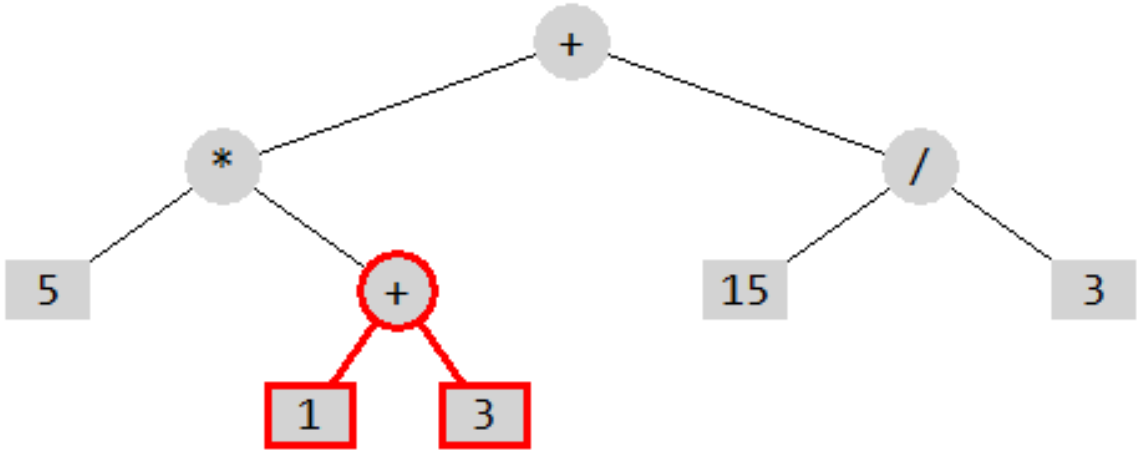
Hoci vo výraze, ktorý je uložený v tomto strome, nie sú žiadne zátvorky, vy si ich viete na správnych miestach domyslieť. Takýmto stromom budeme hovoriť **aritmetické stromy**. Môžete ich nájsť aj pod názvom binary expression tree (ale aj inde), čiastočne aj ako parse tree.

Môžeme zhrnúť základné vlastnosti takýchto stromov:

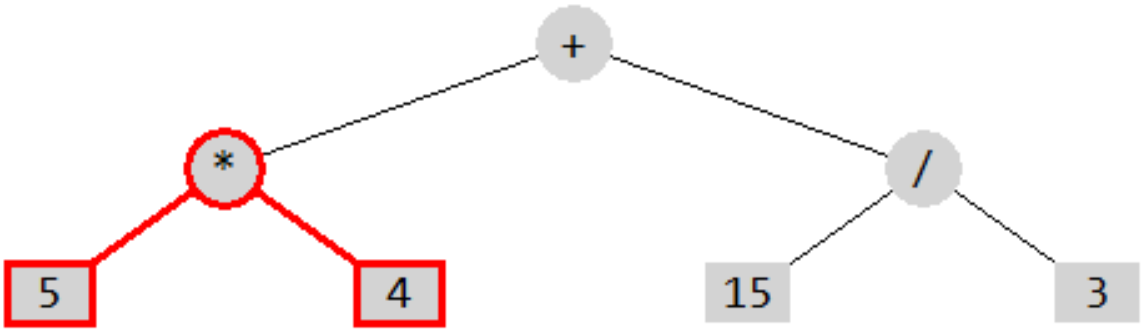
- **vo vnútorných vrcholoch** sa nachádzajú reťazce binárnych operácií, napr. `'+'`, `'-'`, `'*'`, `'/'`, ale môže tu byť aj `'**'` alebo `'and'`, ... ak vieme popísať ich spôsob vyhodnocovania
- **v listoch** sú operandy binárnych operácií, najčastejšie sú to celé alebo desatinné čísla, ale mohli by tu byť aj znakové reťazce, ktoré by mohli reprezentovať mená premenných

Takéto aritmetické stromy vieme veľmi jednoducho vyhodnocovať:

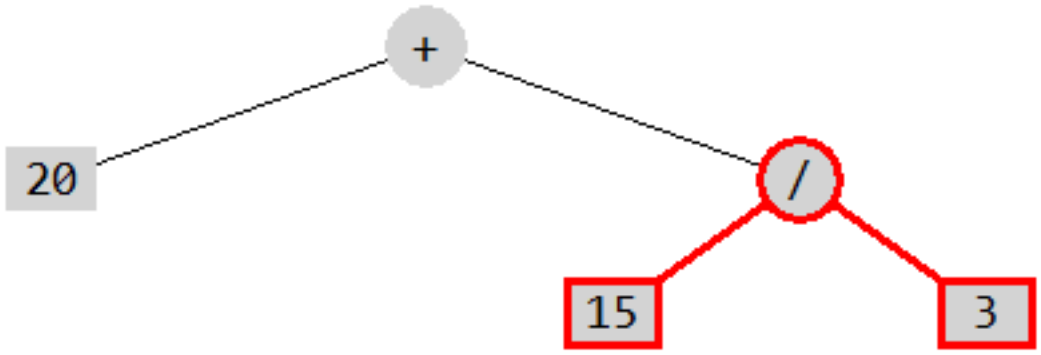
- nájdeme operáciu (vnútorný vrchol), ktorej oba synovia sú už vyhodnotené operandy (listy stromu alebo už vyhodnotené podstromy), v našom príklade je to podstrom `1 + 3`



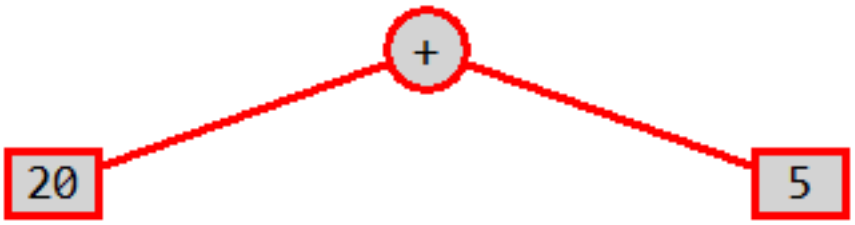
- vyhodnotí sa táto operácia a celý tento podstrom dostáva hodnotu, ktorá je výsledkom tejto operácie, teda číslo `4`



- teraz sa môže vyhodnocovať aj operácia `'*'`, lebo ľavým operandom je číslo `5` a pravým je hodnota pravého podstromu `4` - výsledkom je teda číslo `20`



- ďalšou operáciou je `/`, teda podstrom `15 / 3` - táto operácia sa vyhodnotí a výsledkom podstromu je hodnota `5` (tu to môžeme chápať ako celočíselné delenie)



- na záver sa vyhodnotí operácia `'+'`, ktorá je v koreni celého stromu, lebo už poznáme hodnoty oboch jej podstromov, teda `20` z ľavého podstromu a `5` z pravého - vyhodnotením dostávame výsledok pre celý strom, teda `25`

Uvedomte si, že v skutočnosti sa takýmto vyhodnocovaním aritmetický strom nemodifikuje - ten ostáva bez zmeny. Takto len ilustrujeme postup vyhodnocovania.

Celý tento postup môžeme zapísať rekurzívnou metódou `vyhodnot()` do definície triedy `AritmetickyStrom`:

```
class AritmetickyStrom:
    class Vrchol:
        def __init__(self, data, left=None, right=None):
            self.data = data
            self.left = left
            self.right = right

    def __init__(self):
        self.root = None

    def vyhodnot(self):

        def hodnota(vrch):
            if vrch is None:
                raise SyntaxError
            if vrch.left is None and vrch.right is None:
                return vrch.data
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            hodnota_left = hodnota(vrch.left)
            hodnota_right = hodnota(vrch.right)
            if vrch.data == '+':
                return hodnota_left + hodnota_right
            elif vrch.data == '-':
                return hodnota_left - hodnota_right
            elif vrch.data == '*':
                return hodnota_left * hodnota_right
            elif vrch.data == '/':
                return hodnota_left // hodnota_right
            else:
                raise SyntaxError

        return hodnota(self.root)
```

Keďže zatiaľ nevieme vytvoriť aritmetický strom inak ako priamym priradením vrcholov do koreňa stromu (`root`), musíme to otestovať takto:

```
>>> v = AritmetickyStrom.Vrchol
>>> s = AritmetickyStrom()
>>> s.root = v('+', v('*', v(5), v('+', v(1), v(3))), v('/', v(15), v(3)))
>>> print('vyhodnotenie =', s.vyhodnot())
vyhodnotenie = 25
```

Zaujímavý výsledok dostávame, keď z aritmetického stromu vypíšeme jeho **preorder**, **inorder** a **postorder** postupnosti (skopírujeme ich a mierne zmodifikujeme z predchádzajúcej prednášky):

```
>>> s.preorder()
'+ * 5 + 1 3 / 15 3'
>>> s.inorder()
'5 * 1 + 3 + 15 / 3'
>>> s.postorder()
'5 1 3 + * 15 3 / +'
```

Dostávame nám známe zápisy **prefix**, **infix** a **postfix**. Hoci asi nie je problém upraviť metódu `inorder()` tak, aby každý podvýraz (podstrom) ozátvorkovala, napr. takto:

```
>>> s.inorder()
'((5 * (1 + 3)) + (15 / 3))'
```

Tento reťazec by sme prekopírovaním (copy-paste) vedeli nechať vyhodnotiť aj Pythonu:

```
>>> ((5 * (1 + 3)) + (15 / 3))
25.0
```

30.3 Všeobecný strom

Už vieme, že všeobecný strom je dátová štruktúra podobná binárnym stromom, ale na rozdiel od nich môže mať každý vrchol ľubovoľný počet svojich synov. Zapíšme definíciu triedy aj s niekoľkými užitočnými metódami:

```
import random
import tkinter
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
class VseobecnyStrom:
    canvas = None
    class Vrchol:
        def __init__(self, data):
            self.data = data
            self.child = []

    def __init__(self):
        self.root = None

    def __len__(self):

        def pocet_rek(vrch):
            if vrch is None:
                return 0
            vysl = 1
            for child in vrch.child:
                vysl += pocet_rek(child)
            return vysl

        return pocet_rek(self.root)

    def __repr__(self):

        def repr_rek(vrch):
            if vrch is None:
                return '()'
            vysl = [repr(vrch.data)]
            for child in vrch.child:
                vysl.append(repr_rek(child))
            return '(' + ','.join(vysl) + ')'

        return repr_rek(self.root)

    def pridaj_nahodne(self, *postupnost):

        def pridaj_vrchol(vrch):
            n = len(vrch.child)
            i = random.randrange(n+1)
            if i == n:
                vrch.child.append(self.Vrchol(hodnota))
            else:
                pridaj_vrchol(vrch.child[i])

        for hodnota in postupnost:
            if self.root is None:
                self.root = self.Vrchol(hodnota)
            else:
                pridaj_vrchol(self.root)

    def kresli(self):

        def kresli_rek(vrch, sir, x, y):
            n = len(vrch.child)
            if n != 0:
                sir0 = 2 * sir // n
            for i in range(n):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

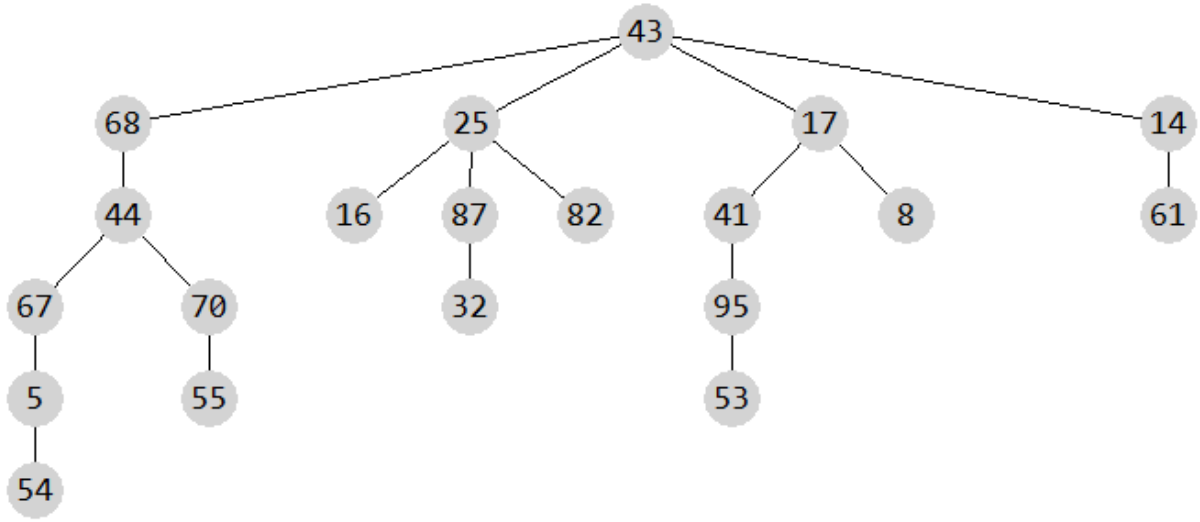
```
                x1, y1 = x - sir + i*sir0 + sir0//2, y + 50
                self.canvas.create_line(x, y, x1, y1)
                kresli_rek(vrch.child[i], sir0//2, x1, y1)
            self.canvas.create_oval(x-15, y-15, x+15, y+15, fill='lightgray', outline=
↪'')
            self.canvas.create_text(x, y, text=vrch.data, font='consolas 14')

        if self.canvas is None:
            VseobecnyStrom.canvas = tkinter.Canvas(bg='white', width=800, height=400)
            self.canvas.pack()
        self.canvas.delete('all')
        kresli_rek(self.root, 380, 400, 40)
```

Otestujeme:

```
strom = VseobecnyStrom()
strom.pridaj_nahodne(*(random.randrange(100) for i in range(20)))
strom.kresli()
print('pocet vrcholov v strome =', len(strom))
print('strom =', strom)
```

Náhodný strom môže vyzerať napr. takto:



a potom dostávame takýto výpis:

```
pocet vrcholov v strome = 20
strom = (43,(68,(44,(67,(5,(54))),(70,(55)))),(25,(16),(87,(32)),(82)),(17,(41,(95,
↪(53))),(8)),(14,(61)))
```

keby sme do `__repr__()` pridali 2 riadky:

```
class VseobecnyStrom:
    ...

    def __repr__(self):

        def repr_rek(vrch):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            if vrch is None:
                return '()'
            vysl = [repr(vrch.data)]
            for child in vrch.child:
                vysl.append(repr_rek(child))
            if len(vysl) == 1:                ##
                return vysl[0]                ##
            return '(' + ','.join(vysl) + ')'

        return repr_rek(self.root)
```

dostávame trochu úspornejší a možno čitateľnejší výpis:

```
>>> strom
(43,(68,(44,(67,(5,54)),(70,55))),(25,16,(87,32),82),(17,(41,(95,53)),8),(14,61))
```

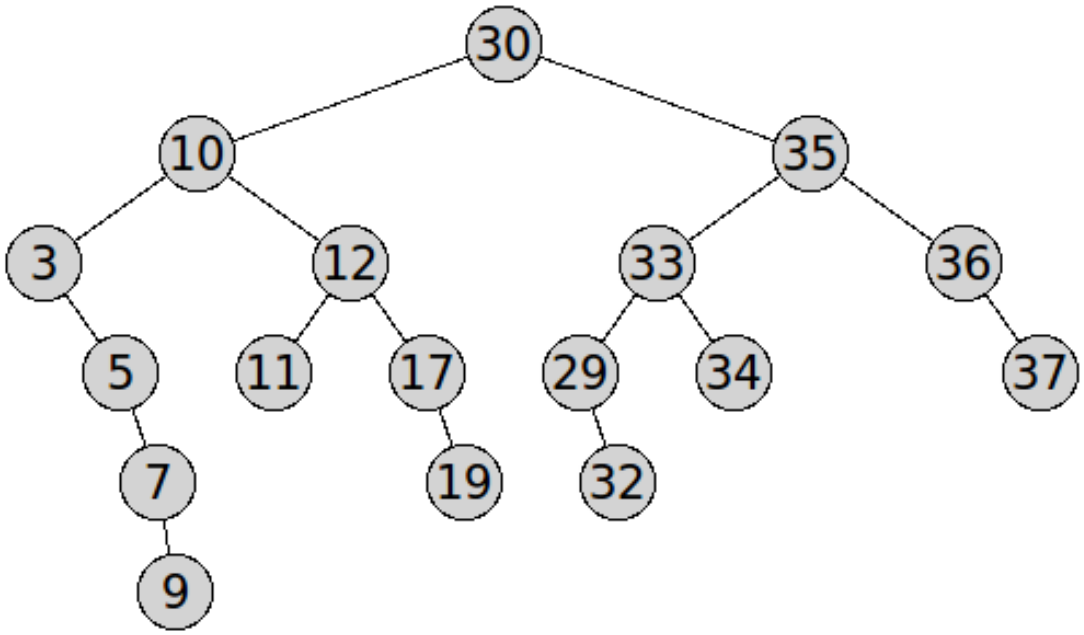
30.4 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

30.4.1 Vyhľadávacie stromy

1. Zistite, či tento strom spĺňa podmienky BVS

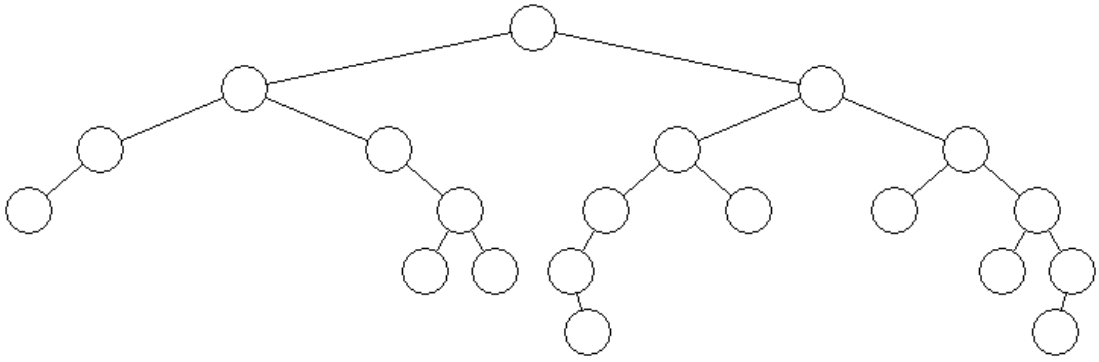


2. Ručne nakreslite strom, ktorý by vznikol postupným pridávaním do BVS

- týchto hodnôt:

```
'KE', 'BB', 'PO', 'PE', 'PK', 'BA', 'BS', 'NI', 'SC', 'BL', 'RK', 'BR'
```

3. Dopíšte do stromu na obrázku čísla z postupnosti `range(1, 21)` tak, aby vznikol BVS:



4. Spojazdnite triedu `BVS` z prednášky a pridajte do nej metódu na vykreslenie stromu z predchádzajúcej prednášky
   - preverte správnosť vášho riešenia pre strom:

```
s = BVS()
v = BVS.Vrchol
s.root = v(17, v(3, v(1, None, v(2)), v(5)), v(20, v(19, v(18), v(21)),␣
↪v(23)))
s.kresli()
```

5. Vylepšite inicializáciu `__init__()` triedy `BVS` tak, aby mohla mať nepovinný druhý parameter `postupnost` - zadaná postupnosť hodnôt, z ktorej sa vytvorí BVS (volaním metódy `vloz()`)
   - funkčnosť otestujte napr.

```
s = BVS('KE BB PO PE PK BA BS NI SC BL RK BR'.split())
s.kresli()
```

6. Do triedy `BVS` zapíšte metódy `min()` a `max()`. Tieto by mali vrátiť minimálnu, resp. maximálnu hodnotu
   - využite vlastnosť BVS, podľa ktorej je minimálny vrchol najnižší na ceste od koreňa vľavo a maximálny vpravo

```
>>> # strom z predchádzajúceho príkladu
>>> s.max()
'SC'
>>> s.min()
'BA'
```

7. Doplňte do triedy `BVS` metódy `inorder_list()` a `preorder_list()` (môžete využiť predchádzajúcu prednášku)
   - prekontrolujte, či **inorder** postupnosť vracia usporiadanú postupnosť všetkých hodnôt v strome, napr. takto

```
>>> zoznam = [... nejaká náhodná postupnost' ...]   # nemali by sa v nej␣
↪opakovat' žiadne hodnoty
>>> s = BVS(zoznam)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> s.inorder_list() == sorted(zoznam)
True
```

- prekontrolujte, či z **preorder** postupnosti sa dá skonštruovať BVS rovnakého tvaru

```
>>> s = BVS(... nejaká postupnost' ...)
>>> s.kresli()
>>> s1 = BVS(s.preorder_list())
>>> s1.kresli()
```

oba tieto stromy by mali vyzerať rovnako

8. (náročnejšia úloha) Danú postupnosť hodnôt preusporiadajte tak, aby BVS, ktorý vznikne postupným pridávaním prvkov tejto preusporiadanej postupnosti, bol skoro úplný. Idea by mohla byť takáto
   - usporiadajte danú postupnosť (napr. pomocou `sorted()`)
   - stredný prvok bude prvý (ďalší) vo výsledku
   - rekurzívne spracujte prvú polovicu (po stredný prvok)
   - rekurzívne spracujte druhú polovicu (od stredného prvku)

30.4.2 Aritmetické stromy

9. Ručne (na papieri) nakresliť strom pre aritmetický výraz
   - `6 * 5 + 7 / (4 - 2 * 5)`
10. Spojazdnite triedu `AritmetickyStrom` z prednášky a doplňte do nej metódu na vykresľovanie stromu
    - otestujte na príklade stromu z prednášky
11. Dopíšte metódy `preorder()`, `inorder()` a `postorder()` tak, aby vrátili znakový reťazec, pričom `inorder()` ozátvorkuje všetky podvýrazy (podstromy), ktoré obsahujú operácie
12. Do inicializácie `__init__()` dopíšte parameter `postfix` (znakový reťazec, ktorý obsahuje postfixový výraz - prvky sú v ňom oddelené medzerou) a metóda z tohto reťazca zkonštruuje aritmetický strom (algoritmus sa podobá vyhodnocovaniu postfixu):
    - vytvorí si pomocný zásobník
    - prerobí vstupný reťazec na zoznam prvkov
    - postupne prechádza prvky zoznamu:
      - ak je prvkom operácia, vyberie z vrchu zásobníka pravý aj ľavý operand, s danou operáciou poskladá malý podstrom (`Vrchol(operácia, l'avý, pravý)`) a vloží ho na vrch zásobníka
      - inak je prvkom číslo (zatiaľ je to reťazec, ktorý reprezentuje číslo), vyrobí z neho nový vrchol (`Vrchol(číslo)`) a vloží ho na vrch zásobníka
    - po spracovaní všetkých prvkov poľa je na vrchu zásobníka kompletný aritmetický strom - už ho iba priradí do atribútu `root`
    - otestujte napr.

```
>>> s = AritmetickyStrom('5 1 3 + * 15 3 / +')
```

30.4.3 Všeobecné stromy

13. Opravte definíciu triedy `VseobecnyStrom` tak, aby sa pri inicializácii vrcholov mohli uviesť aj referencie na podstromy.
    - napr.

```
>>> v = VseobecnyStrom.Vrchol
>>> s = VseobecnyStrom()
>>> s.root = v('a', v('b', v('c'), v('d')), v('e', v('f', v('g')), v('h'), v(
↪'i'), v('j')))
>>> s
('a',('b','c','d'),('e',('f','g'),'h','i','j'))
```

14. Do triedy `VseobecnyStrom` napíšte metódy:
    - `preorder()` - vráti postupnosť hodnôt v poli (typu `list`)
    - `vyska()` - zistí výšku stromu
    - `pocet_listov()` - zistí počet listov stromu

30.5 6. Domáce zadanie

L.I.S.T.

- riešenie odovzdávajte na úlohový server https://list.fmph.uniba.sk/

Napíšte modul s menom `riesenie6.py`, ktorý bude obsahovať jedinú triedu s ďalšou vnorenou podtriedou, tromi metódami a jednou funkciou:

```
class BVS:
    class Vrchol:
        def __init__(self, data, left=None, right=None):
            self.data = ...
            self.left = left
            self.right = right
            ...

    def __init__(self, postupnost=None):
        self.root = None
        ...

    def vloz(self, data):
        ...

    def inorder_list(self):
        return []

def tree_sort(zoznam):
    return BVS(zoznam).inorder_list()
```

Trieda `BVS` implementuje **binárny vyhľadávací strom** a pomocou neho algoritmus, ktorý triedi postupnosť údajov, tzv. **tree_soft**. Ak by sa mala nejaká hodnota (tzv. **kluc**) objavit v BVS viackrát, v strome bude tento kľúč len raz, ale

v príslušnom vrchole sa zapamätajú všetky hodnoty s týmto kľúčom presne v tom poradí, ako sa objavili vo vstupnej postupnosti (najlepšie v ďalšom atribúte vrcholu, ktorý bude typu `list`).

Metódy triedy `BVS` by mali mať túto funkčnosť (môžete si dodefinovať aj ďalšie pomocné metódy):

- metóda `__init__(postupnost)` z prvkov danej postupnosti vytvorí **binárny vyhľadávací strom**, jeho koreň bude v atribúte `root`, použite na to metódu `vloz()`;
- metóda `vloz(data)` pridá do **BVS** stromu nový prvok; ak je týmto pridávaným prvkom zoznam (`list` alebo `tuple`), tak sa ako kľúč použije len jeho prvý prvok, ale zapamätá sa jeho kompletná hodnota;
- metóda `inorder_list()` vráti **inorder** zoznam (`list`) všetkých prvkov v strome; zrejme tento výsledný zoznam bude obsahovať rovnaké prvky ako pôvodná postupnosť ale možno v inom poradí; pri testovaní sa môžu objaviť také údaje, pri ktorých môže spadnúť rekurzívna verzia tejto metódy.

30.5.1 Obmedzenia

- vaše riešenie odovzdajte v súbore `riesenie6.py`, pričom sa v ňom bude nachádzať **len jedna definícia triedy** `BVS`, trieda `Vrchol` bude vnorená v triede `BVS`
- prvé dva riadky tohto súboru budú obsahovať:

```
# autor: Janko Hrasko
# uloha: 6. domace zadanie tree_sort
```

- zrejme ako autora uviediete svoje meno
- atribút `root` v triede `BVS` musí obsahovať referenciu na koreň binárneho vyhľadávacieho stromu
- váš program by nemal počas testovania testovačom nič vypisovať (žiadne vaše testovacie `print()`)

30.5.2 Testovanie

Keď budete spúšťať vaše riešenie na svojom počítači, môžete do súboru `riesenie6.py` pridať testovacie riadky, ktoré ale testovač vidieť nebude, napr.:

```
if __name__ == '__main__':
    zoz = (10, 20, 30, 5, 15)
    print(tree_sort(zoz))
    zoz = [(10,), (20,), (30,), (5,), (15,)]
    print(tree_sort(zoz))
    zoz = [('b',6), ('d',7), ('e',8), ('a',9), ('b',10),
           ('b',1), ('d',2), ('e',3), ('a',4), ('b',5)]
    print(tree_sort(zoz))
    zoz = ['prvy', ('druhy',), ['treti'], ('stvrty', 1),
           ['piaty', 2], ('siesty', 'x'), ['siedmy', 'y']]
    print(tree_sort(zoz))
```

Tento test by vám mal vypísať:

```
[5, 10, 15, 20, 30]
[(5,), (10,), (15,), (20,), (30,)]
[('a', 9), ('a', 4), ('b', 6), ('b', 10), ('b', 1), ('b', 5), ('d', 7), ('d', 2), ('e
↪', 8), ('e', 3)]
[('druhy',), ['piaty', 2], 'prvy', ['siedmy', 'y'], ('siesty', 'x'), ('stvrty', 1), [
↪'treti']]
```

KAPITOLA 31

31. Triedenia

Triedením rozumieme taký algoritmus, ktorý **usporiada** vstupnú postupnosť prvkov (napr. zoznam, n-ticu, spájaný zoznam, ...) tak, aby ich **poradie** zodpovedalo nejakému usporiadaniu, napr. **vzostupne** alebo **zostupne**. Prvky, ktoré takto usporadúvame, sa môžu skladať z viacerých zložiek (častí alebo atribútov), pričom len nejaká ich časť slúži na porovnávanie usporiadania - časti prvku, podľa ktorého usporadúvame, hovoríme **kľúč**.

Mnohé z týchto triediacich algoritmov sú **in-place** (triedenie na mieste), čo znamená, že výsledná usporiadaná postupnosť sa nachádza na tom istom mieste ako bola vstupná. Takýmto triedením je aj pythonovská metóda `sort()` pre typ zoznam (`list`), napr.

```
>>> zoznam = [7, 16, 3, 7, 9, 5, 10]
>>> zoznam.sort()
>>> zoznam
[3, 5, 7, 7, 9, 10, 16]
```

Iný typ triediacich algoritmov vstupnú postupnosť nemení, ale vytvorí novú usporiadanú postupnosť (najčastejšie ako zoznam, t.j. typ `list`), ktorá obsahuje tie isté prvky, ako boli na vstupe, ale v správnom poradí. Napr. aj štandardná pythonovská funkcia `sorted()` pracuje na tomto princípe:

```
>>> ntica = (7, 16, 3, 7, 9, 5, 10)
>>> zoznam = sorted(ntica)
>>> zoznam
[3, 5, 7, 7, 9, 10, 16]
```

V ďalších častiach ukážeme niekoľko najznámejších jednoduchých triediacich algoritmov. Zameriame sa len na **in-place** triedenia, t.j. také, ktoré triedia `n`-prvkový zoznam (typu `list`) a výsledok usporiadania bude na pôvodnom mieste.

31.1 Bubble, min a insert sort

Predstavme si takýto problém:

Na stole máme v rade rozložených n kariet s nejakými číslami (napr. od 1 do n). Karty sú zamiešané a otočené tak, že nevidíme ich čísla. Karty treba usporiadať podľa čísel od najmenšieho po najväčšie. Pre preusporiadavanie môžeme použiť jedinú operáciu:

- vyberieme nejaké dve karty a otočíme ich číslom nahor
- porovnáme tieto dve hodnoty a rozhodneme sa, či ich navzájom vymeníme
- opäť otočíme karty číslom nadol

Toto môžeme opakovať ľubovoľný počet krát, ale asi najvhodnejšie by bolo to urobiť čo na najmenší počet týchto operácií.

Presne takto to „vidí“ aj triediaci algoritmus: mám k dispozícii nejaký zoznam čísel a ja teraz pomocou vzájomného porovnávania a prípadných výmen musím ich preusporiadať tak, aby boli na záver v rastúcom poradí.

Najjednoduchší algoritmus, ktorým dokážeme usporiadať prvky n-prvkového zoznamu bude postupovať takto:

- postupne bude porovnávať dvojice prvkov 0 <=> 1, 1 <=> 2, 2 <=> 3, ... n-3 <=> n-2, n-2 <=> n-1
- pre každú porovnávanú dvojicu, keď zistí že pravý prvok (druhý z dvojice) je menší ako ľavý (t.j. i-ty je väčší ako i+1-ty), tieto dva prvky navzájom vymení
- tento proces ešte neusporiada celý zoznam, treba ho opakovať n-krát

Ukážme to na takomto príklade:

Máme zoznam s týmito číslami:

```
7, 9, 5, 8, 2
```

- keďže karty sú otočené číslom nadol, vidíme len:

```
X X X X X
```

- otočíme prvé dve karty a porovnáme ich:

```
7 9 X X X
```

- keďže prvý prvok nie je väčší ako druhý, nerobíme nič, len otočíme karty a pokračujeme na ďalších dvoch kartách:

```
X 9 5 X X
```

- druhá karta z dvojice je menšia ako prvá, preto ich vymeníme:

```
X 5 9 X X
```

- pokračujeme z ďalšou dvojicou:

```
X X 9 8 X
```

- opäť ich vymeníme:

```
X X 8 9 X
```

- a pokračujeme na ďalšej dvojici:

```
X X X 9 2
```

- aj tieto dve musíme navzájom vymeniť

Týmto končí prvý prechod algoritmu a keby sme sa teraz pozreli do momentálneho obsahu zoznamu, videli by sme

```
7, 5, 8, 2, 9
```

Tento proces bude treba ešte niekoľkokrát zopakovať.

- postupne:

```
X X X X X
7 5 X X X
5 7 X X X   <-- výmena
X 7 8 X X
X X 8 2 X
X X 2 8 X   <-- výmena
X X X 8 9
```

Teraz zoznam obsahuje postupne:

```
5, 7, 2, 8, 9
```

Ďalší prechod algoritmu

- postupne:

```
X X X X X
5 7 X X X
X 7 2 X X
X 2 7 X X   <-- výmena
X X 7 8 X
X X X 8 9
```

V zozname teraz máme:

```
5 2 7 8 9
```

Ešte jeden prechod

- postupne:

```
X X X X X
5 2 X X X
2 5 X X X
X 5 7 X X
X X 7 8 X
X X X 8 9
```

A konečne dostávame utriedený zoznam:

```
2, 5, 7, 8, 9
```

Pozrime, ako vyzerá zoznam na začiatku a potom aj na konci každého prechodu:

```
7, 9, 5, 8, 2
-------------
7, 5, 8, 2, 9   <-- 9 je na svojom mieste
5, 7, 2, 8, 9   <-- 8, 9 sú na svojom mieste
5, 2, 7, 8, 9   <-- 7, 8, 9 sú na svojom mieste
2, 5, 7, 8, 9   <-- 5, 7, 8, 9 sú na svojom mieste
```

Všimnite si, po každom prechode „precestuje“ ďalší najväčší prvok na svoje miesto: najprv 9, potom 8, potom 7, atď. Hovoríme, že každým prechodom sa nejaké prvky presúvajú na koniec (veľké bublinky) a nejaké (malé bublinky) sa presunú smerom k začiatku. Hovoríme tomu **bublinkové triedenie**, lebo malé bublinky klesajú na začiatok a veľké stúpajú na koniec (ako bublinky v pohári vody).

31.1.1 Bubble_sort

Bublinkové triedenie postupne prechádza celý zoznam a porovnáva všetky vedľa seba ležiace dvojice prvkov: ak je prvý z nich väčší ako druhý, treba ich navzájom vymeniť. Tento postup sa opakuje dovtedy, kým nezostane celý zoznam utriedený, teda maximálne n-krát.

Zapíšme tento algoritmus, ale najprv si pripravíme pomocnú funkciu `vymen()`, pomocou ktorej budeme vymieňať prvky zoznamu:

```
def vymen(zoz, i, j):
    zoz[i], zoz[j] = zoz[j], zoz[i]

def bubble_sort(zoz):
    for i in range(len(zoz)):
        for j in range(len(zoz)-1):
            if zoz[j] > zoz[j+1]:
                vymen(zoz, j, j+1)
```

Všimnite si, že táto funkcia nevracia žiadnu hodnotu (teda vracia `None`) ani nič nevypisuje. Funkcia len modifikuje vstupný zoznam, ktorý dostala ako parameter.

Otestujme:

```
>>> zz = [7, 16, 3, 7, 9, 5, 10]
>>> bubble_sort(zz)
>>> zz
[3, 5, 7, 7, 9, 10, 16]
```

Skúsme si priebeh algoritmu vizualizovať tak, že vo vnútornom cykle ešte pred porovnaním vypíšeme obsah zoznamu a vyznačíme, ktoré dva prvky sa porovnávajú:

```
def bubble_sort(zoz):
    for i in range(len(zoz)):
        for j in range(len(zoz)-1):
            print(*zoz[:j], (zoz[j], zoz[j+1]), *zoz[j+2:])    # pridali sme výpis
            if zoz[j] > zoz[j+1]:
                vymen(zoz, j, j+1)
```

Po spustení rovnakého testu dostaneme dosť dlhý výpis, ale jeho začiatok je takýto:

```
(7, 16) 3 7 9 5 10
7 (16, 3) 7 9 5 10
7 3 (16, 7) 9 5 10
7 3 7 (16, 9) 5 10
7 3 7 9 (16, 5) 10
7 3 7 9 5 (16, 10)
(7, 3) 7 9 5 10 16
3 (7, 7) 9 5 10 16
3 7 (7, 9) 5 10 16
3 7 7 (9, 5) 10 16
3 7 7 5 (9, 10) 16
3 7 7 5 9 (10, 16)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
(3, 7) 7 5 9 10 16
...
```

Z tohto výpisu vidíme, že po prvom prechode algoritmu (po prvých šiestich riadkoch) sa maximálny prvok `16` dostáva na koniec zoznamu, teda na svoje cieľové miesto. Každý ďalší prechod porovnáva všetky susediace dvojice prvkov, teda bude zakaždým porovnávať aj predposledný s posledným - čo je zrejme zbytočné. Totiž po každom prechode sa dostáva na svoje miesto ďalší a ďalší prvok od konca, nielen posledný prvok po prvom prechode. Z tohto dôvodu sa zvykne vnútorný cyklus zakaždým skrátiť o jeden prvok od konca. Vhodnejším zápisom by bolo:

```
def bubble_sort(zoz):
    for i in range(len(zoz)):
        for j in range(len(zoz)-1-i):          # skracujeme vnutorný cyklus zakaždým o␣
↪1
            print(*zoz[:j], (zoz[j], zoz[j+1]), *zoz[j+2:])    # pridali sme výpis
            if zoz[j] > zoz[j+1]:
                vymen(zoz, j, j+1)
```

Na otestovanie bubliknkového triedenia môžeme použiť napr. generátor náhodných čísel:

```
import random

zoz = [random.randrange(100) for i in range(5000)]
zoz1 = sorted(zoz)           # kontrolne utriedený zoznam
bubble_sort(zoz)             # bublinkove triedenie
print(zoz == zoz1)
```

Tento program najprv vygeneruje 5000-prvkový zoznam náhodných čísel od 0 do 99. Potom do `zoz1` priradí jeho usporiadanú kópiu, ktorú sme vytvorili pomocou štandardnej funkcie `sorted()`. Ďalej usporiada náš zoznam pomocou algoritmu `bubble_sort()` (zrejme sme vyhodili kontrolný výpis) a na záver vypíše, či sú tieto dva zoznamy rovnaké. Zrejme, ak je naše bublinkové triedenie správne, program by mal vypísať `True`. Tento test môže bežať aj niekoľko sekúnd.

Ďalšie triedenie, tzv. **min sort**, funguje veľmi podobne, ako bublinkové triedenie.

31.1.2 Min sort

Triedenie s výberom minimálneho prvku pracuje na tomto princípe:

- v prvom prechode algoritmu presťahuje minimálny prvok na prvú pozíciu v zozname
- v každom ďalšom `i`-tom prechode presťahuje `i`-tu najmenšiu hodnotu na `i`-tu pozíciu - zrejme hľadá minimálnu od `i`-tej pozície zoznamu až do konca
- keď toto zopakuje `n`-krát, dostane usporiadaný celý zoznam
- `i`-ty minimálny prvok bude hľadať a presťahovávať nasledovným postupom:
  - porovná dvojice prvkov `i <=> 1`, `i <=> 2`, `i <=> 3`, ... `i <=> n-2`, `i <=> n-1`
  - pre každú porovnávanú dvojicu, keď zistí že `i`-ty prvok je väčší ako ten druhý (t.j. `i`-ty je väčší ako `j`-ty), tieto dva prvky navzájom vymení

Zapíšme tento algoritmu:

```
def min_sort(zoz):
    for i in range(len(zoz)-1):
        for j in range(i+1, len(zoz)):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            if zoz[i] > zoz[j]:
                vymen(zoz, i, j)
```

Všimnite si, ako sa táto verzia veľmi podobá na bublinkové triedenie. Môžeme aj sem pridať kontrolný výpis podobne, ako sme to robili v bublinkovom triedení. Teraz výpis vložíme až za vnútorný cyklus, ktorý hľadá a presťahuje `i`-ty minimálny prvok na svoje miesto:

```
def vymen(zoz, i, j):
    zoz[i], zoz[j] = zoz[j], zoz[i]

def min_sort(zoz):
    for i in range(len(zoz)-1):
        for j in range(i+1, len(zoz)):
            if zoz[i] > zoz[j]:
                vymen(zoz, i, j)
        print(*zoz[:i], [zoz[i]], *zoz[i+1:])

zz = [7, 16, 3, 7, 9, 5, 10]
print(*zz)
print('-----------------')
min_sort(zz)
print('vysledok =', zz)
```

V kontrolnom výpise môžete vidieť prvok zoznamu, ktorý sa práve zaradil ako `i`-ty minimálny:

```
7 16 3 7 9 5 10
-----------------
[3] 16 7 7 9 5 10
3 [5] 16 7 9 7 10
3 5 [7] 16 9 7 10
3 5 7 [7] 16 9 10
3 5 7 7 [9] 16 10
3 5 7 7 9 [10] 16
vysledok = [3, 5, 7, 7, 9, 10, 16]
```

Každý ďalší riadok kontrolného výpisu ukazuje obsah zoznamu po jednom prechode vonkajšieho cyklu, t.j. po zaradení i-teho najmenšieho na správne miesto: v prvom riadku je ešte zoznam netriedený, v druhom je jeho obsah po zaradení prvého minima na svoje miesto (číslo `3`), v ďalšom riadku je zaradené už aj druhé číslo `5`, atď. V poslednom riadku výpisu je utriedený celý zoznam.

Rovnaký test s náhodným zoznamom, ako sme robili s bublinkovým triedením, môžeme spustiť aj pre min sort (zrejme sme opäť vyhodili kontrolný výpis):

```
import random

zoz = [random.randrange(100) for i in range(5000)]
zoz1 = sorted(zoz)
min_sort(zoz)
print(zoz == zoz1)
```

31.1.3 Insert sort

Triedenie vkladaním pracuje na takomto princípe:

- predpokladajme, že istá časť zoznamu na začiatku je už utriedená

- počas triedenia sa najbližší ešte neutriedený prvok zaradí do utriedenej časti na svoje miesto a tým sa tento utriedený úsek predĺži o 1

Zapíšme algoritmus:

```
def insert_sort(zoz):
    for i in range(1, len(zoz)):
        j = i
        while j > 0 and zoz[j-1] > zoz[j]:
            vymen(zoz, j-1, j)
            j -= 1
```

Ako to funguje:

- v premennej `i` je hranica medzi utriedenými a neutriedenými časťami zoznamu
  - pri štarte predpokladáme, že prvý prvok je už jednoprvková utriedená časť a preto začíname od indexu 1 (teda druhým prvkom zoznamu)
- vo vnútornom cykle je v premennej `j` index zaraďovaného prvku do utriedenej časti (úsek zoznamu `zoz[0:i]`)
- kým je predchádzajúci prvok pred `j`-tym väčší ako zaraďovaný (`zoz[j]`), tento zaraďovaný ho predbehne (vymení ich navzájom) a zmenší `j`

Aby sme lepšie pochopili, ako to funguje, môžeme vložiť kontrolné výpisy a otestovať to:

```
def vymen(zoz, i, j):
    zoz[i], zoz[j] = zoz[j], zoz[i]

def insert_sort(zoz):
    print(zoz[0], '|', *zoz[1:])
    for i in range(1, len(zoz)):
        j = i
        while j > 0 and zoz[j-1] > zoz[j]:
            vymen(zoz, j-1, j)
            j -= 1
        print(*zoz[:i+1], '|', *zoz[i+1:])

zz = [7, 16, 3, 7, 9, 5, 10]
insert_sort(zz)
print('vysledok =', zz)
```

dostávame tento výpis:

```
7 | 16 3 7 9 5 10
7 16 | 3 7 9 5 10
3 7 16 | 7 9 5 10
3 7 7 16 | 9 5 10
3 7 7 9 16 | 5 10
3 5 7 7 9 16 | 10
3 5 7 7 9 10 16 |
vysledok = [3, 5, 7, 7, 9, 10, 16]
```

Znak `'|'` slúži na oddelenie utriedenej a ešte neutriedenej časti zoznamu.

Aj pre toto triedenie môžeme spustiť test s 5000-prvkovým náhodným zoznamom (zrejme sme opäť vyhodili kontrolné výpisy):

```
import random
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
zoz = [random.randrange(100) for i in range(5000)]
zoz1 = sorted(zoz)
insert_sort(zoz)
print(zoz == zoz1)
```

31.2 Vizualizácia triedenia

Rôznych algoritmov triedenia je dosť veľa, ale na základe kontrolných výpisov obsahu zoznamu sa zriedkavo dá zistiť, ako to funguje naozaj. Teraz ukážeme, ako využiť Python na vizualizáciu algoritmov, ktoré modifikujú nejaký zoznam. Zostavíme triedu, pomocou ktorej budeme môcť sledovať, čo sa deje s nejakým zoznamom:

```
class Vizualizuj:
    def __init__(self, zoz):
        self.zoz = zoz

    def __getitem__(self, index):
        return self.zoz[index]

    def __len__(self):
        return len(self.zoz)

    def __setitem__(self, index, hodnota):
        self.zoz[index] = hodnota
```

Táto nová trieda `Vizualizuj` bude sledovanie všetky základné operácie so zoznamom:

- ak máme nejaký zoznam, napr. `zoz = [2, 3, 5]` a zabalíme ho triedou `z = Vizualizuj(zoz)`, potom operácie indexovania `z[]` automaticky zavolajú metódy `__getitem__()` a `__setitem__()`
- do metódy `__setitem__()` môžeme pridať nejaký výpis, napr. `print(f'zoz[{index}] = {hodnota}')`
- teraz môžeme zavolať ľubovoľný algoritmus, ktorý manipuluje so zoznamom a my mu namiesto neho podstrčíme inštanciu triedy `Vizualizuj` (hovoríme tomu **wrap**)

```
class Vizualizuj:
    def __init__(self, zoz):
        self.zoz = zoz
        print('povodny zoznam =', zoz)

    def __getitem__(self, index):
        return self.zoz[index]

    def __len__(self):
        return len(self.zoz)

    def __setitem__(self, index, hodnota):
        self.zoz[index] = hodnota
        print(f'zoz[{index}] = {hodnota}')

def vymen(zoz, i, j):
    zoz[i], zoz[j] = zoz[j], zoz[i]

def min_sort(zoz):
    for i in range(len(zoz)-1):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        for j in range(i+1, len(zoz)):
            if zoz[i] > zoz[j]:
                vymen(zoz, i, j)

zz = [7, 16, 3, 9]
min_sort(Vizualizuj(zz))
print('vysledok =', zz)
```

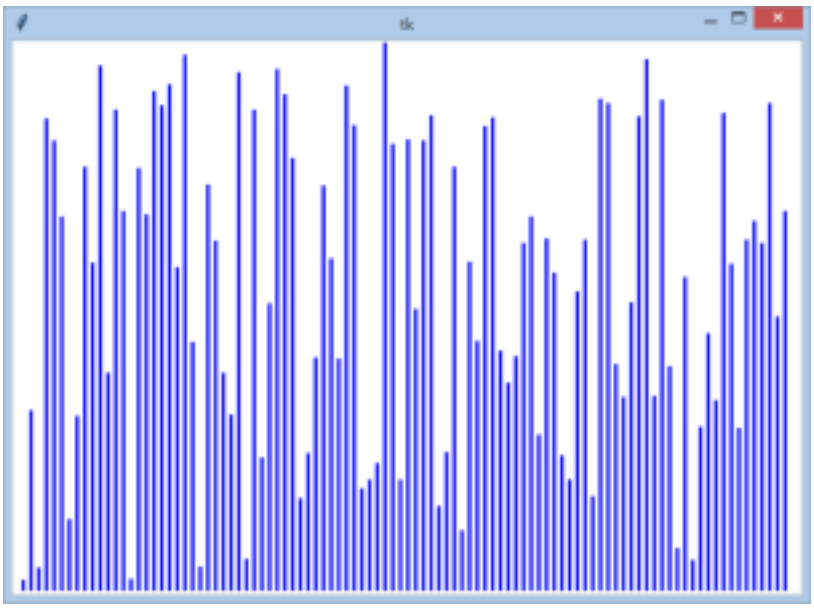
Všimnite si, že algoritmus `min_sort` netriedi obyčajný zoznam, ale zoznam, ktorý je obalený (teda **wrap**) triedou `Vizualizuj`. Vďaka tomu, hoci neobsahuje žiadne kontrolné výpisy, dostávame informácie o všetkých priradeniach do prvkov zoznamu. Okrem toho, už pri inicializácii, vypisujeme pôvodný zoznam:

```
povodny zoznam = [7, 16, 3, 9]
zoz[0] = 3
zoz[2] = 7
zoz[1] = 7
zoz[2] = 16
zoz[2] = 9
zoz[3] = 16
vysledok = [3, 7, 9, 16]
```

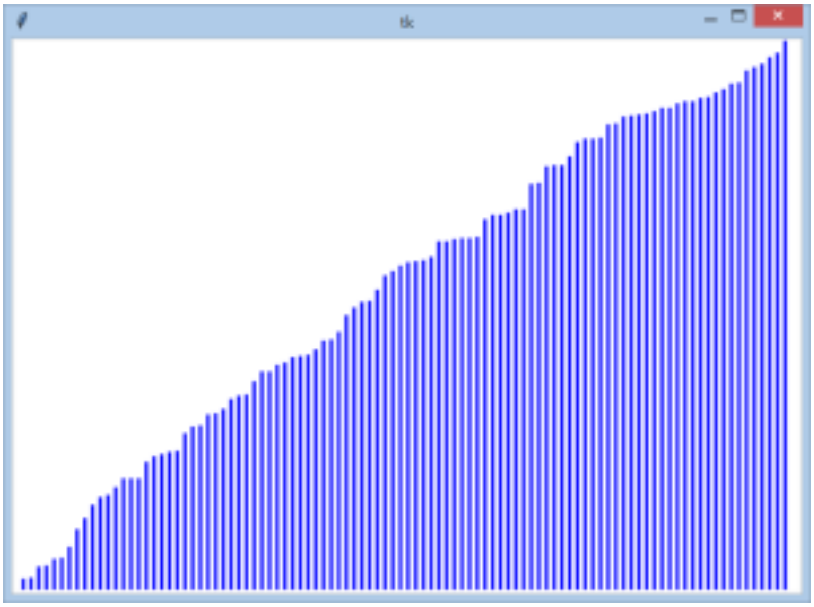
Takto by sme vedeli otestovať ľubovoľný algoritmus, v ktorom modifikujeme prvky nejakého zoznamu, napr. aj `bubble_sort` alebo `insert_sort`.

31.2.1 Vizualizácia v grafickom režime

Často sa používajú iné zobrazovania momentálneho obsahu zoznamu, napr. grafický, v ktorom sú rôzne veľké hodnoty zobrazené rôznou dĺžkou úsečiek. Napr. takto by sme mohli zobraziť nejaký 100-prvkový neutriedený zoznam:



V grafickej ploche je tu zobrazený obsah 100-prvkového zoznamu: každý prvok zodpovedá úsečke, ktorej dĺžka zodpovedá príslušnej hodnote v zozname. Uvedomte si, že keby sme takto zobrazili vzostupne usporiadaný zoznam, úsečky by mali tvoriť takýto rastúcu postupnosť:



Počas práce algoritmu by sa pri každej výmene dvoch prvkov zoznamu mohli vymeniť aj nakreslené úsečky - zrejme namiesto fyzického vymieňania úsečiek im len zmeníme dĺžky podľa aktuálneho obsahu príslušných prvkov zoznamu.

Trieda `Vizualizuj` preto musí zabezpečiť:

- pri inicializácii inštancie v metóde `__init__()` sa okrem zapamätania samotného zoznamu v atribúte `self.zoz` vykreslí celý zoznam do grafickej plochy ako postupnosť úsečiek
  - zrejme najprv vytvorí grafickú plochu (`canvas`)
  - identifikačné čísla nakreslených čiar uloží do zoznamu `self.id`, aby ich mohol neskôr meniť (v príkaze `canvas.coords`)
- pri zmene hodnoty prvku zoznamu `zoz[index]` metóda `__setitem__()` zrealizuje túto zmenu v zozname a tiež zmení dĺžku príslušnej úsečky
  - využije identifikačné číslo úsečky v zozname `self.id`
- pri zisťovaní hodnoty prvku zoznamu `zoz[index]` metóda `__getitem__()` vráti príslušnú hodnotu zo svojho atribútu
- pri zisťovaní dĺžky `len(zoz)` metóda `__len__()` vráti aktuálnu dĺžku zoznamu

Trieda `Vizualizuj` teraz kreslí čiary:

```
import tkinter

class Vizualizuj:
    def __init__(self, zoz):
        self.zoz = zoz
        self.canvas = tkinter.Canvas(width=800, height=600, bg='white')
        self.canvas.pack()
        self.dx = 800 / len(zoz)
        self.id = [None] * len(zoz)
        for i in range(len(zoz)):
            self.id[i] = self.canvas.create_line(i*self.dx, 600, i*self.dx, 600-
↪zoz[i])

    def __getitem__(self, index):
        return self.zoz[index]
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def __setitem__(self, index, hodnota):
        self.zoz[index] = hodnota
        self.canvas.coords(self.id[index], index*self.dx, 600, index*self.dx, 600-
↪hodnota)
        self.canvas.update()

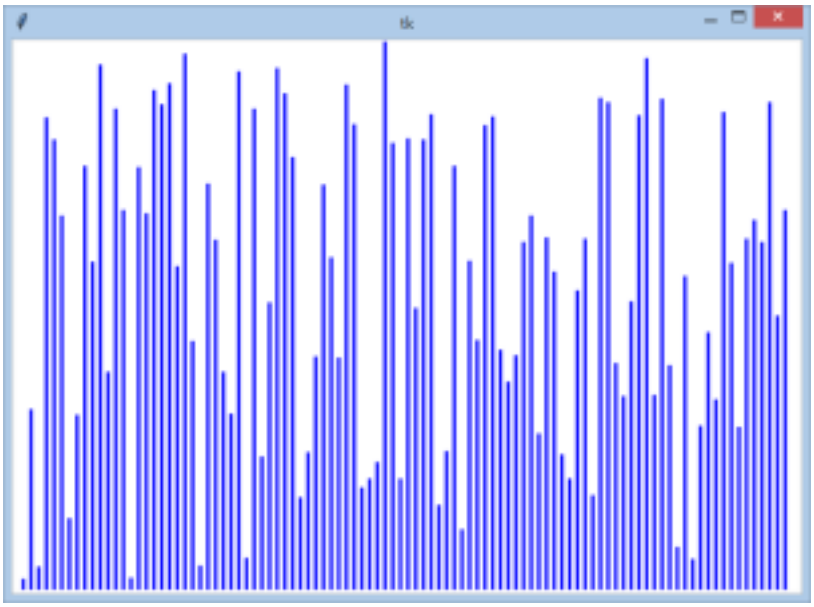
    def __len__(self):
        return len(self.zoz)
```

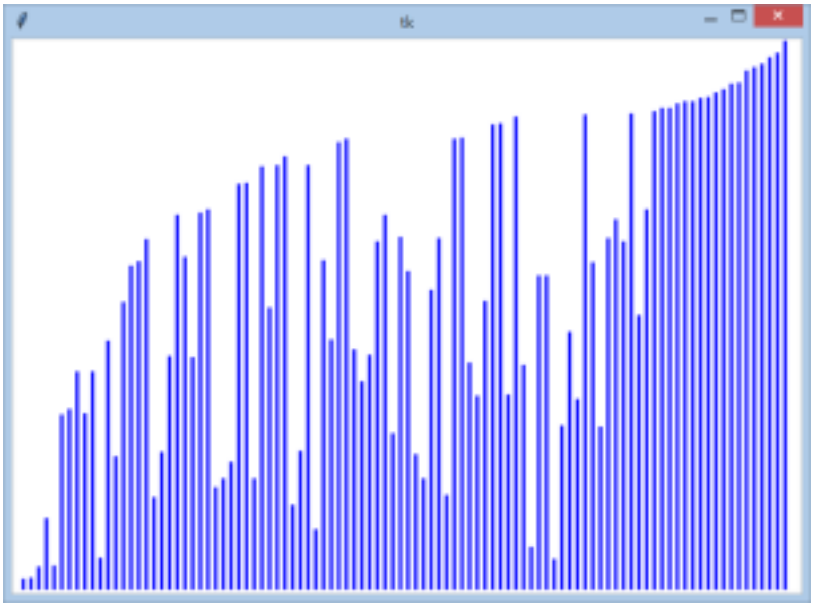
Teraz už môžeme otestovať ľubovoľné triedenie aj s vizualizáciou úsečkami. Vyskúšajme náhodný 100-prvkový zoznam a jeho utriedenie pomocou algoritmu `bubble_sort`:

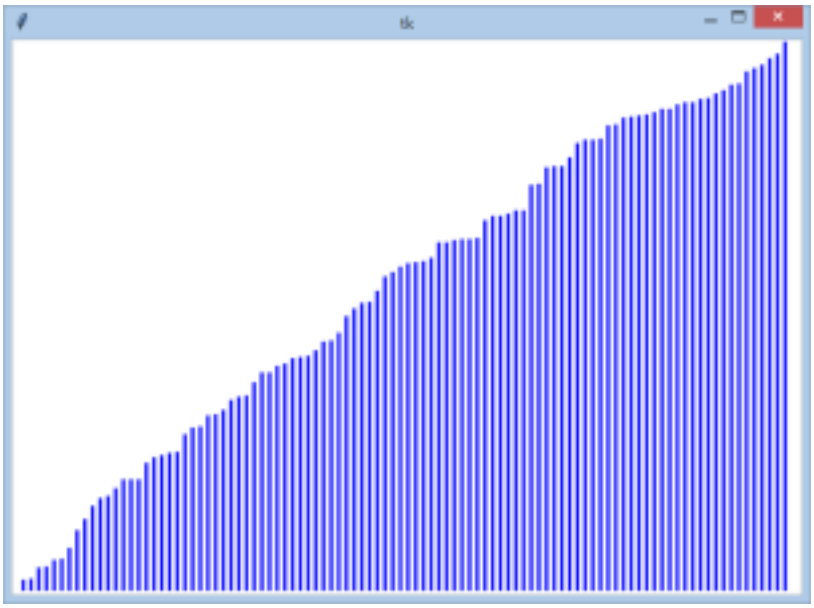
```
import random

zz = [random.randrange(500) for i in range(100)]
bubble_sort(Vizualizuj(zz))
print(zz)
```

Môžete postupne vidieť najprv náhodne vygenerovaný zoznam, potom priebežný stav počas triedenia (na konci zoznamu sa ukladajú maximálne prvky) a na záver utriedený zoznam:







Otestujte takto aj `min_sort()` aj `insert_sort`.

31.3 Quick sort

Je to najznámejší algoritmus rýchleho triedenia, ktorý v roku 1959 vymyslel Tony Hoare.

Algoritmus využíva **rekurziu**. Popíšme zjednodušený princíp:

- najprv zvolíme ľubovoľný prvok zoznamu ako tzv. **pivot** (pre jednoduchosť zvolíme prvý prvok zoznamu)
- ďalej všetky zvyšné prvky rozdelíme na tri kopy: na prvky, ktoré sú menšie ako **pivot**, na prvky, ktoré sa rovnajú **pivot** a na všetky zvyšné
- teraz rovnakým algoritmom (t.j. rekurzívnym volaním) utriedime dve kopy: kopu menších a kopu väčších a takto utriedené časti (spolu s kopou rovnakých) nakoniec spojíme do výsledného utriedeného zoznamu

Prvá verzia je rekurzívna funkcia, ktorá nemodifikuje svoj parameter, len vráti novo vytvorený zoznam, ktorý už teraz bude utriedený (zrejme tento algoritmus nie je **in-place**):

```
def quick_sort(zoz):
    if len(zoz) < 2:
        return zoz
    pivot = zoz[0]
    mensie = [prvok for prvok in zoz if prvok < pivot]
    rovne =  [prvok for prvok in zoz if prvok == pivot]
    vacsie = [prvok for prvok in zoz if prvok > pivot]
    return quick_sort(mensie) + rovne + quick_sort(vacsie)
```

Takéto riešenie **nemôžeme vizualizovať** v grafickej ploche našou triedou `Vizualizuj`, keďže výsledok sa postupne zlepuje z veľkého množstva malých kúskov pomocných zoznamov (my vieme vizualizovať len zmeny v pôvodnom zozname). Jedine, čo môžeme otestovať je vyskúšať triedenie aj väčšieho zoznamu, napr.

```
import random

zoz = [random.randrange(10000) for i in range(5000)]
zoz1 = sorted(zoz)
zoz2 = quick_sort(zoz)
print(zoz1 == zoz2)
```

Funguje správne. Môžete vyskúšať napr. aj `500000`-prvkový náhodný zoznam - aj tento by sa mal utriediť veľmi rýchlo.

Ďalej sa zameriame na verziu, ktorá **nebude** používať pomocné zoznamy, ale `quick_sort()` prebehne v samotnom zozname (bude **in-place**):

- na začiatku sa samotný zoznam (jeho časť od indexu `z` do indexu `k`) rozdelí na dve časti:
- zvolí sa tzv. **pivot** (napr. prvý prvok zoznamu) a zvyšné prvky sa postupne rozdelia do ľavej časti zoznamu (pred **pivot**) a pravej (za **pivot**) podľa toho či sú menšie alebo väčšie ako **pivot**
- v premennej `index` je pozícia **pivot** v takto upravenom zozname (ešte neutriedenom) - uvedomte si, že **pivot** je už na svojom mieste (podobne ako boli minimálne prvky v `min_sort()` a maximálne prvky v `bubble_sorte()`)
- ďalej nasleduje rekurzívne volanie pre ľavú časť zoznamu pred **pivot** a pre pravú časť zoznamu za **pivot**

Táto druhá verzia je teda už **in-place**, t.j. nevytvára sa nový zoznam, ale triedi sa priamo ten zoznam, ktorý je parametrom funkcie:

```
def quick_sort(zoz):
    def quick(z, k):
        if z < k:
            # rozdelenie na dve časti
            index = z
            pivot = zoz[index]
            for i in range(z+1, k+1):
                if zoz[i] < pivot:
                    index += 1
                    vymen(zoz, index, i)
            vymen(zoz, index, z)
            # v index je teraz pozícia pivota
            quick(z, index-1)
            quick(index+1, k)

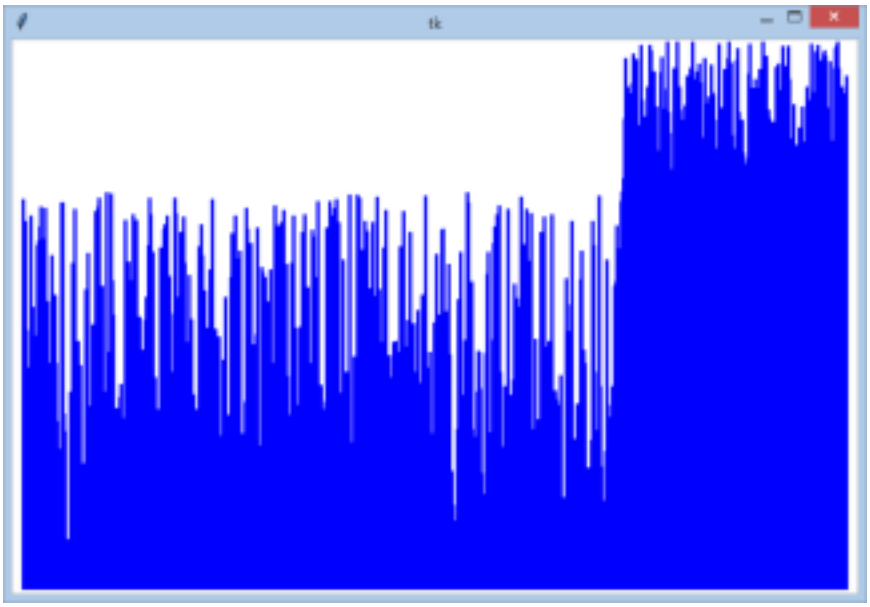
    quick(0, len(zoz)-1)
```

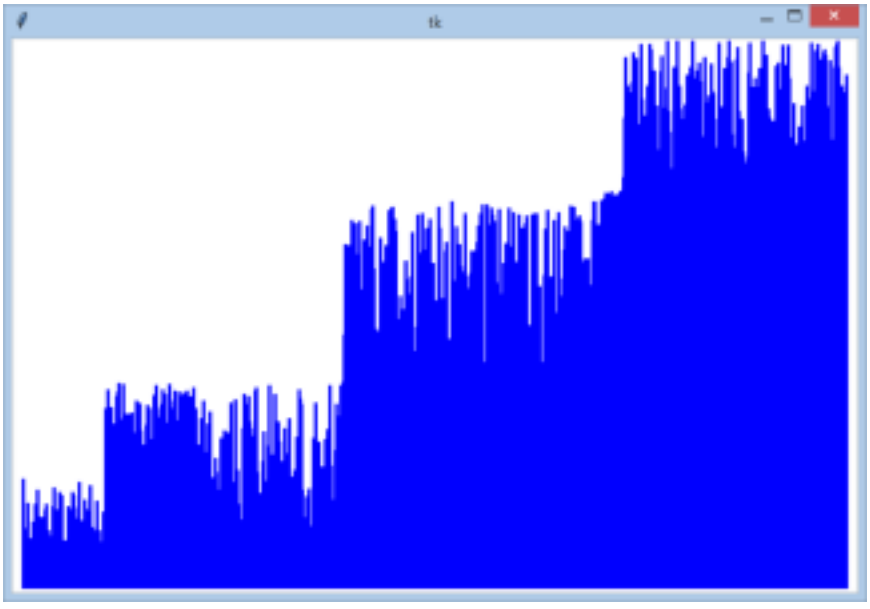
Otestujte to pomocou vizualizácie:

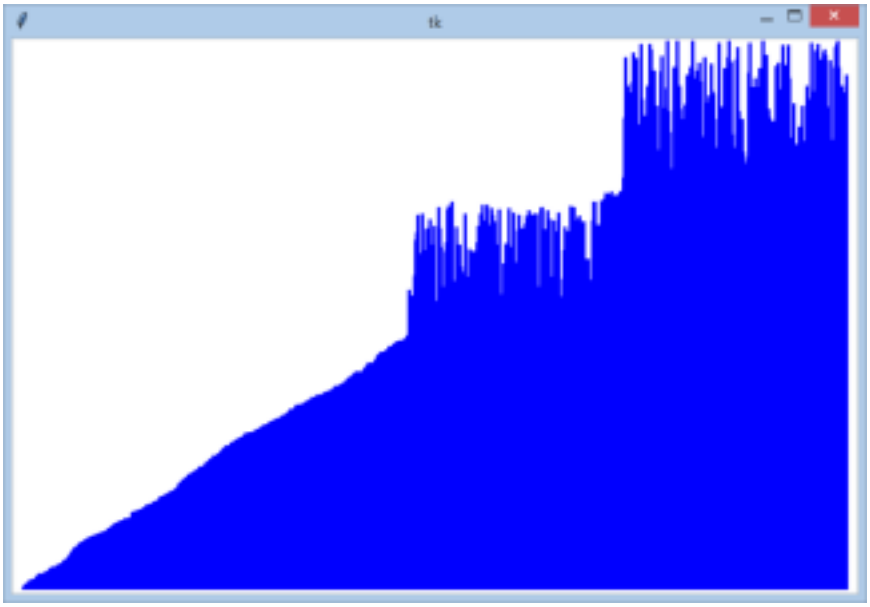
```
import random

zz = [random.randrange(600) for i in range(800)]
quick_sort(Vizualizuj(zz))
```

Môžete postupne vidieť, ako sa najprv náhodne vygenerovaný zoznam rozdelil podľa pivota na menšie a väčšie prvky, potom sa takto rekurzívne rozdelil aj prvý úsek a na poslednom zábere je už polovica zoznamu utriedená a triedi sa zvyšok zoznamu:







Niekedy sa triedenia zvyknú vizualizovať nie pomocou úsečiek ale len pomocou jedného koncového bodu úsečky - takto na začiatku vidíme náhodne rozhádzané bodky, ktoré sa postupne zoskupujú. Opravíme triedu `Vizualizuj`:

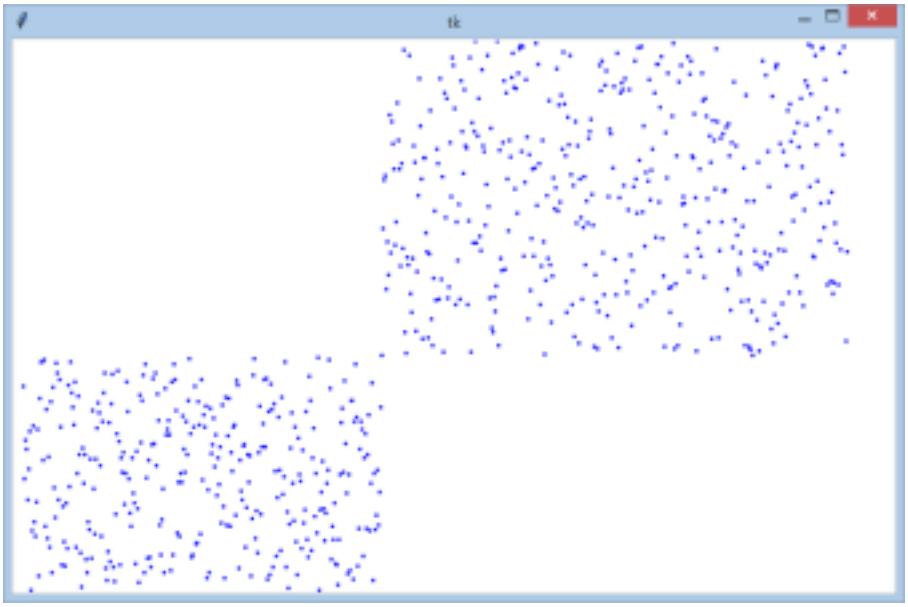
```
class Vizualizuj:
    def __init__(self, zoz):
        self.zoz = zoz
        self.canvas = tkinter.Canvas(width=800, height=600, bg='white')
        self.canvas.pack()
        self.dx = 800 / len(zoz)
        self.id = [None] * len(zoz)
        for i in range(len(zoz)):
            self.id[i] = self.canvas.create_line(i*self.dx, 600-zoz[i], i*self.dx,␣
↪601-zoz[i])

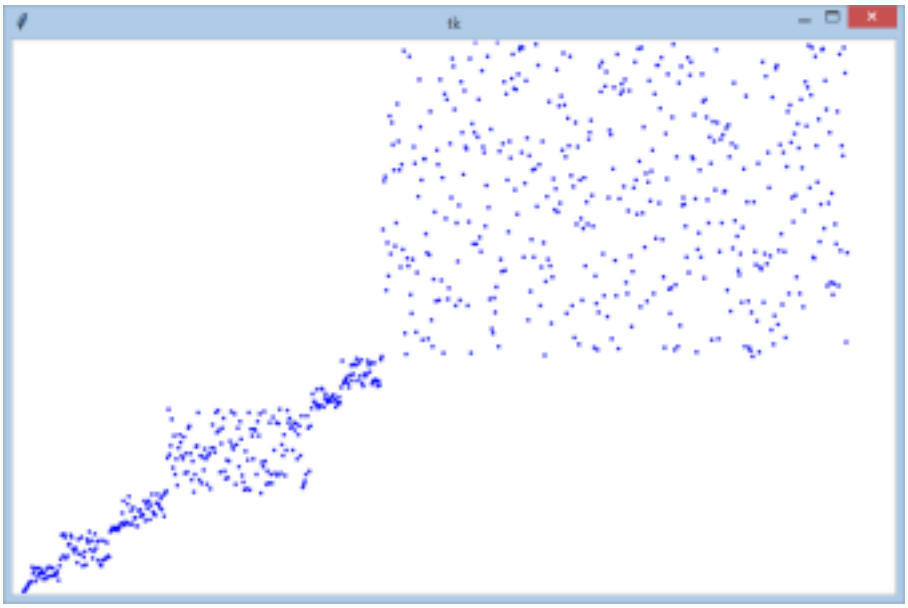
    def __getitem__(self, index):
        return self.zoz[index]

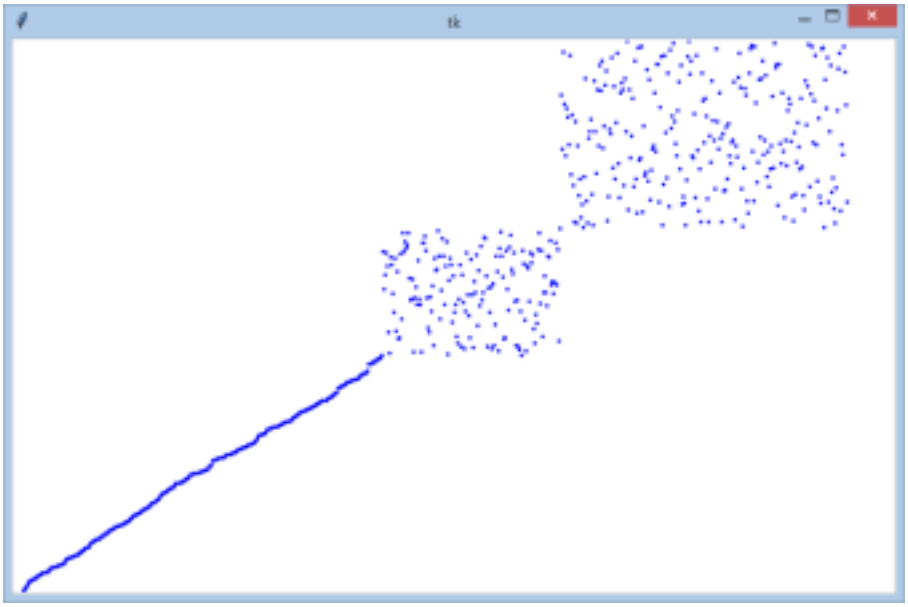
    def __setitem__(self, index, hodnota):
        self.zoz[index] = hodnota
        self.canvas.coords(self.id[index], index*self.dx, 600-hodnota, index*self.dx,␣
↪601-hodnota)
        self.canvas.update()

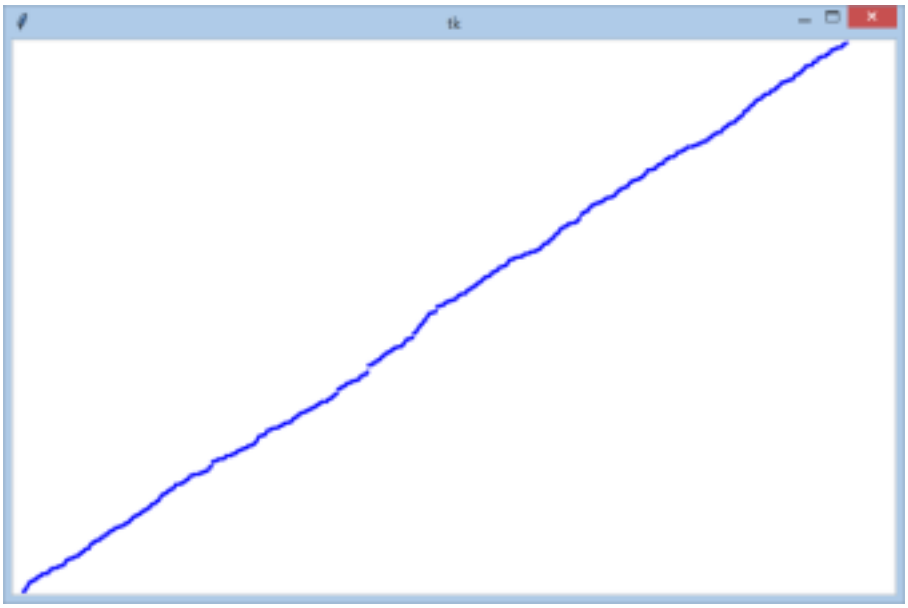
    def __len__(self):
        return len(self.zoz)
```

Po spustení vizualizácie vidíme:









31.3.1 Štandardné triedenie v Pythone

štandardná funkcia `sorted()` triedi, napr.

- postupnosť čísel, reťazcov
- znakový reťazec
- postupnosť n-tíc, napr. postupnosť súradníc
- postupnosť dvojíc (kľúč, hodnota), ktorá vznikne z asociatívneho poľa pomocou `slovnik.items()`

Výsledkom je vždy zoznam (typ `list`). Ak pri volaní uvedieme ďalší parameter `reverse=True`, zoznam sa utriedi zostupne od najväčších hodnôt po najmenšie.

Pozrime túto ukážku:

```
>>> import random
>>> zz = [random.randrange(10000) for i in range(15)]
>>> zz
[1372, 6518, 7580, 9941, 9518, 3211, 8428, 7624, 35, 9341, 572, 6218, 3819, 8943,␣
↪8527]
>>> sorted(zz)
[35, 572, 1372, 3211, 3819, 6218, 6518, 7580, 7624, 8428, 8527, 8943, 9341, 9518,␣
↪9941]
>>> sorted(zz, reverse=True)
[9941, 9518, 9341, 8943, 8527, 8428, 7624, 7580, 6518, 6218, 3819, 3211, 1372, 572,␣
↪35]
```

Vidíme vzostupne aj zostupne utriedený náhodne vygenerovaný zoznam. Ak by sme dostali úlohu tento zoznam utriediť len podľa posledných dvoch cifier každého čísla, bolo by to ťažšie. Vyrobme si pomocnú funkciu a nový zoznam, ktorý obsahuje len tieto posledné cifry:

```
>>> def posledne2(x):
        return x % 100

>>> zoz1 = [posledne2(p) for p in zoz]
>>> zoz1
[72, 18, 80, 41, 18, 11, 28, 24, 35, 41, 72, 18, 19, 43, 27]
>>> sorted(zoz1)
[11, 18, 18, 18, 19, 24, 27, 28, 35, 41, 41, 43, 72, 72, 80]
```

Ak chceme teraz vrátiť pôvodné čísla k týmto dvom posledným cifrám, musíme si ich pamätať, najlepšie vo dvojici spolu s poslednými 2 ciframi:

```
>>> zoz1 = [(posledne2(p), p) for p in zoz]
>>> zoz1
[(72, 1372), (18, 6518), (80, 7580), (41, 9941), (18, 9518), (11, 3211), (28, 8428),
 (24, 7624), (35, 35), (41, 9341), (72, 572), (18, 6218), (19, 3819), (43, 8943), (27,
↪ 8527)]
>>> zoz2 = sorted(zoz1)
>>> zoz2
[(11, 3211), (18, 6218), (18, 6518), (18, 9518), (19, 3819), (24, 7624), (27, 8527),
 (28, 8428), (35, 35), (41, 9341), (41, 9941), (43, 8943), (72, 572), (72, 1372), (80,
↪ 7580)]
```

Už je to skoro hotové: z týchto utriedených dvojíc zoberieme len pôvodné čísla, t.j. zoberieme druhé prvky dvojíc:

```
>>> [p[1] for p in zoz2]
[3211, 6218, 6518, 9518, 3819, 7624, 8527, 8428, 35, 9341, 9941, 8943, 572, 1372,␣
↪7580]
```

A máme naozaj to, čo sme na začiatku očakávali: utriedený pôvodný zoznam, ale triedime len podľa posledných dvoch cifier čísel v zozname.

Štandardná funkcia `sorted()` má okrem parametra `reverse` ešte jeden, ktorý slúži práve na toto: zadáme funkciu, ktorá určí, ako sa bude triedenie pri porovnávaní pozerať na každý prvok zoznamu - podľa tohto sa pôvodný zoznam bude triediť. Takže, komplikovaný postup s vytváraním dvojíc môžeme zjednodušiť použitím parametra `key`:

```
>>> sorted(zoz, key=posledne2)
[3211, 6518, 9518, 6218, 3819, 7624, 8527, 8428, 35, 9941, 9341, 8943, 1372, 572,␣
↪7580]
>>> sorted(zoz, key=lambda x: x%100)
[3211, 6518, 9518, 6218, 3819, 7624, 8527, 8428, 35, 9941, 9341, 8943, 1372, 572,␣
↪7580]
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

Vidíme, že tu môžeme použiť aj konštrukciu `lambda`. Takže, keď zadáme parameter `key`, tak triedenie nebude porovnávať prvky zoznamu, ale prerobené prvky zoznamu zadanou funkciou, napr. `posledne2`.

Zamyslite sa, v akom poradí sa teraz utriedi tento istý zoznam:

```
>>> sorted(zoz, key=str)
[1372, 3211, 35, 3819, 572, 6218, 6518, 7580, 7624, 8428, 8527, 8943, 9341, 9518,␣
↪9941]
```

Zhrňme použitie štandardnej funkcie `sorted()`:

štandardná funkcia `sorted()`

Funkcia `sorted()` z prvkov ľubovoľnej postupnosti (napr. zoznam, n-tica, reťazec, súbor, ...) vytvorí zoznam (typu `list`) týchto hodnôt vo vzostupne usporiadanom poradí. Funkcia môže mať ešte tieto dva nepovinné parametre:

- `reverse=True` spôsobí usporiadanie prvkov vo vzostupnom poradí (od najväčšieho po najmenšie)
- `key=funkcia`, kde `funkcia` je buď meno existujúcej funkcie alebo `lambda` konštrukcia - táto funkcia musí byť definovaná pre **jeden parameter** - potom vzájomné porovnávanie dvoch prvkov pri usporadúvaní bude používať práve túto funkciu

Je dôležité, aby sa všetky triedené prvky dali navzájom porovnávať, napr. nemôžeme usporiadať zoznam, ktorý obsahuje čísla aj reťazce. Môžeme ich pomocou parametra `key` previesť na také hodnoty, ktoré porovnávať vieme.

Nasledujúce ukážky ilustrujú najmä použitie parametra `key`.

- usporiadanie množiny mien (čo sú reťazce v tvare meno priezvisko) najprv podľa priezvisk, a ak sú dve priezviská rovnaké, tak podľa mien:

```
>>> m = {'Janko Hrasko', 'Eugen Suchon', 'Ludovit Stur', 'Andrej Sladkovic',
↪'Janko Stur'}
>>> sorted(m)
['Andrej Sladkovic', 'Eugen Suchon', 'Janko Hrasko', 'Janko Stur', 'Ludovit Stur']
>>> sorted(m, key=lambda x: x.split()[::-1])
['Janko Hrasko', 'Andrej Sladkovic', 'Janko Stur', 'Ludovit Stur', 'Eugen Suchon']
```

- usporiadanie telefónneho zoznamu podľa telefónnych čísel, keďže zoznam je tu definovaný ako asociatívne pole, vytvoríme z neho najprv zoznam dvojíc (kľúč, hodnota):

```
>>> tel = {'Betka':737373, 'Dusan':555444, 'Anka': 363636, 'Egon':210210,
'Cyril': 911111, 'Gaba':123456, 'Fero':288288}
>>> sorted(tel.items())
[('Anka', 363636), ('Betka', 737373), ('Cyril', 911111), ('Dusan', 555444),
 ('Egon', 210210), ('Fero', 288288), ('Gaba', 123456)]
>>> sorted(tel.items(), key=lambda x: x[1])
[('Gaba', 123456), ('Egon', 210210), ('Fero', 288288), ('Anka', 363636),
 ('Dusan', 555444), ('Betka', 737373), ('Cyril', 911111)]
```

- usporiadanie zoznamu reťazcov bez ohľadu na malé a veľké písmená:

```
>>> zoz = ('Prvy', 'DRUHY', 'druha', 'pRva', 'PRVE', 'druhI')
>>> sorted(zoz)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
['DRUHY', 'PRVE', 'Prvy', 'druhI', 'druha', 'pRva']
>>> sorted(zoz, key=str.lower)
['druha', 'druhI', 'DRUHY', 'pRva', 'PRVE', 'Prvy']
>>> sorted(zoz, key=lambda s: s.lower())
['druha', 'druhI', 'DRUHY', 'pRva', 'PRVE', 'Prvy']
```

všimnite si dva spôsoby použitia metódy `lower()`

- usporiadanie zoznamu bodov v rovine (zoznam dvojíc `(x, y)`) podľa toho, ako sú vzdialené od bodu `(0 0)`; predpokladáme, že máme danú funkciu `vzd(x, y)`, ktorá vypočíta vzdialenosť bodu od počiatku:

```
>>> def vzd(x, y):
        return (x**2 + y**2)**.5
>>> import random
>>> body = [(random.randint(-5, 5), random.randint(-5, 5)) for i in range(10)]
>>> body
[(-1, 5), (2, 3), (-1, 0), (-1, -2), (0, 1), (-4, 4), (-4, 4), (5, -1), (5, -2),␣
↪(-5, 0)]
>>> sorted(body, key=lambda b: vzd(*b))
[(-1, 0), (0, 1), (-1, -2), (2, 3), (-5, 0), (-1, 5), (5, -1), (5, -2), (-4, 4),␣
↪(-4, 4)]
```

všimnite si, že tu sme nemohli priamo zapísať `key=vzd`, lebo táto funkcia očakáva 2 parametre a my máme triediť dvojice

- v danom slove usporiadať znaky podľa abecedy:

```
>>> slovo = 'krasokorculovanie'
>>> ''.join(sorted(slovo))
'aaceikklnooorrsuv'
```

- v zozname sa vyskytujú reťazce aj čísla, treba to nejako usporiadať:

```
>>> zoz = ['abc', 3.14, 'def', 2.71, 'ghi', 333, 'jkl', 22]
>>> sorted(zoz)

TypeError: unorderable types: float() < str()
>>> sorted(zoz, key=str)
[2.71, 22, 3.14, 333, 'abc', 'def', 'ghi', 'jkl']
```

31.4 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Ručne bez počítača zistite, čo sa bude vypisovať (funkcia `vymen(zoz, i, j)` vymení obsahy `i`-teho a `j`-teho prvku daného zoznamu `zoz`):
   - bublinkové triedenie

```
def bubble_sort(zoz):
    for i in range(len(zoz)):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        for j in range(len(zoz)-1):
            if zoz[j] > zoz[j+1]:
                vymen(zoz, j, j+1)
        print(*zoz)

z = [13, 7, 11, 3, 5, 2]
bubble_sort(z)
```

- triedenie s výberom minimálneho prvku

```
def min_sort(zoz):
    for i in range(len(zoz)-1):
        for j in range(i+1, len(zoz)):
            if zoz[i] > zoz[j]:
                vymen(zoz, i, j)
        print(*zoz)

z = [13, 7, 11, 3, 5, 2]
min_sort(z)
```

- triedenie vkladaním

```
def insert_sort(zoz):
    for i in range(1, len(zoz)):
        j = i
        while j > 0 and zoz[j-1] > zoz[j]:
            vymen(zoz, j-1, j)
            j -= 1
        print(*zoz)

z = [13, 7, 11, 3, 5, 2]
insert_sort(z)
```

2. Zistite, čo budú vypisovať predchádzajúce tri triedenia, ak vstupný zoznam bude utriedený vzostupne, napr.
   - bublinkové triedenie

```
z = [1, 3, 5, 7, 9]
bubble_sort(z)
```

   - triedenie s výberom minimálneho prvku

```
z = [1, 3, 5, 7, 9]
min_sort(z)
```

   - triedenie vkladaním

```
z = [1, 3, 5, 7, 9]
insert_sort(z)
```

3. Spojazdnite vizualizáciu testov a otestujte, ako bude prebiehať triedenie 300-prvkového náhodného zoznamu pomocou `bubble_sort()`, `min_sort()`, `insert_sort()` aj `quick_sort()`.
4. Napíšte funkciu `zisti(postupnost, vzost=True)`, ktorá zistí (vráti `True` alebo `False`), či je zadaná postupnosť hodnôt usporiadaná vzostupne. Vstupnú postupnosť neukladajte do žiadneho pomocného zoznamu. Parameter `vzost` s hodnotou `False` označuje, že kontrolujete, či je postupnosť usporiadaná zostupne. Vzostupné usporiadanie označuje, že pre žiadne dva susedné prvky postupnosti nie je prvý z nich väčší ako druhý.

- malo by fungovať napr.

```
>>> zisti([1, 3, 3, 4])
True
>>> zisti(range(10))
True
>>> zisti(range(10), False)
False
```

5. Otestujte algoritmus `bubble_sort()` pre triedenie znakových reťazcov, ktoré sa skladajú z dvoch slov.
   - napr.

```
>>> zz = ['Janko Hrasko', 'Peter Botafogo', 'Juraj Janosik', 'Adam Sangala',
↪'Juraj Hrasko']
>>> bubble_sort(z)
>>> print(*z, sep='\n')
```

   - vytvorte novu verziu tohto triedenia, ktoré úsporiada takéto dvojslovné reťazce podľa druhého slova:

```
def bubble_sort2(zoz):
    ...
```

```
>>> z = z = ['Janko Hrasko', 'Peter Botafogo', 'Juraj Janosik', 'Adam Sangala
↪', 'Juraj Hrasko']
>>> bubble_sort2(z)
>>> print(*z, sep='\n')
Peter Botafogo
Janko Hrasko
Juraj Hrasko
Juraj Janosik
Adam Sangala
```

   - snažte sa v tejto novej verzii triedenia nepoužívať žiaden pomocný zoznam

6. Zapíšte verziu `min_sort_rev(zoz)`, ktorá na princípe triedenia s výberom minimálneho prvku utriedi vstupný zoznam v opačnom poradí.
   - otestujte, napr.

```
zz = [random.randrange(1000) for i in range(1000)]
szz = sorted(zz, reverse=True)
min_sort_rev(zz)
print(zz == szz)
```

7. Táto verzia triedenia vkladaním v niektorých situáciách vypíše momentálny obsah celého zoznamu.
   - ručne odtrasujte tento algoritmus a vypíšte tieto kontrolné výpisy:

```
def insert_sort1(zoz):
    for i in range(1, len(zoz)):
        j, t = i, zoz[i]
        while j > 0 and zoz[j-1] > t:
            zoz[j] = zoz[j-1]
            j -= 1
        if j < i:
            zoz[j] = t
            print(*zoz)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
z = [5, 9, 4, 3, 6, 10, 1, 8, 2, 7]
insert_sort1(z)
```

8. Nasledovná verzia triedenia vsúvaním `insert_sort2()` namiesto priraďovania do prvkov zoznamu používa metódy `pop()` a `insert()`.
   - vložte do tejto funkcie kontrolné výpisy tak, ako sa to robilo na prednáške, a skontrolujte jej spôsob triedenia

```
def insert_sort2(zoz):
    for i in range(1, len(zoz)):
        prvok = zoz.pop(i)
        j = i-1
        while j >= 0 and zoz[j] > prvok:
            j -= 1
        zoz.insert(j+1, prvok)
```

   - zamyslite sa, prečo pre túto verziu nebude fungovať vizualizácia pomocou `Vizualizuj`
   - odmerajte rýchlosť tohto triedenia v porovnaní s verziou z prednášky, vložte sem meranie času (`time.time()`):

```
zoz = [random.randrange(n) for i in range(5000)]
zoz1 = zoz[:]
insert_sort(zoz)
insert_sort2(zoz1)
print(zoz == zoz1)
```

9. Ručne bez počítača odkrokujte
   - `quick_sort()` z prednášky:

```
def quick_sort(zoz):
    def quick(z, k):
        if z < k:
            # rozdelenie na dve casti
            index = z
            pivot = zoz[z]
            for i in range(z+1, k+1):
                if zoz[i] < pivot:
                    index += 1
                    vymen(zoz, index, i)
            vymen(zoz, index, z)
            # v index je teraz pozicia pivota
            print(*zoz)                          # <== vypis
            quick(z, index-1)
            quick(index+1, k)
    print(*zoz)                                  # <== vypis
    quick(0, len(zoz)-1)

z = [32, 12, 66, 19, 75, 29, 50]
quick_sort(z)
```

   - program vypíše 6 riadkov čísel, pričom posledné dva sú rovnaké

10. Zapíšte tri funkcie `utried1(veta)`, `utried2(veta)` a `utried3(veta)`, ktoré nejako usporiadajú slová v danej vete. Využijete štandardnú funkciu `sorted()` a prípadne aj parametre `reverse` a `key`.
    - `utried1()` usporiada slová vo vete podľa abecedy

- `utried2()` usporiada slová vo vete podľa abecedy ale zostupne
- `utried3()` usporiada slová vo vete podľa dĺžky slov (najprv najkratšie) a až keď sú rovnako dlhé podľa abecedy

```
def utried1(veta):
    return ...

def utried2(veta):
    return ...

def utried3(veta):
    return ...
```

```
>>> utried1('kohutik jaraby nechod do zahrady')
'do jaraby kohutik nechod zahrady'
>>> utried2('kohutik jaraby nechod do zahrady')
'zahrady nechod kohutik jaraby do'
>>> utried3('jano ide z blavy do brna')
'z do ide brna jano blavy'
```

11. Do všetkých funkcií `bubble_sort()`, `min_sort()`, `insert_sort()` a `quick_sort()` z prednášky dorobte návratovú hodnotu, ktorou bude dvojica (počet porovnaní, počet výmen). Teda každá z funkcií vráti dve čísla: celkový počet porovnaní medzi prvkami zoznamu a celkový počet volaní funkcie `vymen()`.

    - otestujte napr. takto:

```
zoz0 = [random.randrange(1000) for i in range(5000)]

for sort in bubble_sort, min_sort, insert_sort, quick_sort:
    zoz = list(zoz0)
    tim = time.time()
    pp, pv = sort(zoz)
    tim = time.time() - tim
    print(pp, pv, tim)
```

    - mali by ste dostať štyri riadky výpisu - v každom je dvojica celých čísel a jedno desatinné číslo - zamyslite sa, čo by ste mohli z týchto čísel usúdiť

31.5 7. Domáce zadanie

L.I.S.T.

- riešenie odovzdávajte na úlohový server https://list.fmph.uniba.sk/

Naprogramujte triedenie **spájaného zoznamu** pomocou takéhoto min-sortu, ktorý ale zatiaľ pracuje len s obyčajným zoznamom (typu `list`):

```
def vymen(zoz, i, j):
    zoz[i], zoz[j] = zoz[j], zoz[i]

def prechod(zoz, prvy):
    for druhy in range(prvy+1, len(zoz)):
        if zoz[prvy] > zoz[druhy]:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            vymen(zoz, prvy, druhy)

def min_sort(zoz):
    for prvy in range(len(zoz)-1):
        prechod(zoz, prvy)
```

Toto triedenie zapíšte do metód tejto triedy:

```
class Zoznam:
    class Vrchol:
        def __init__(self, data, next=None):
            self.data = data
            self.next = next

    def __init__(self, postupnost):
        self.zac = self.Vrchol(None)
        ...

    def prechod(self, prvy, reverse):
        ...

    def min_sort(self, reverse):
        prvy = self.zac
        while prvy.next is not None:
            self.prechod(prvy, reverse)
            prvy = prvy.next

    def daj_zoznam(self):
        ...
        return [...]

def min_sort(postupnost, reverse=False):
    z = Zoznam(postupnost)
    z.min_sort(reverse)
    return z.daj_zoznam()
```

Vo vašom riešení má zmysel modifikovať len tieto metódy:

- `__init__()` (inicializácia v triede `Zoznam`) vytvorí z prvkov danej postupnosti spájaný zoznam (vrcholmi zoznamu budú inštancie vnorenej triedy `Vrchol`); všimnite si, že prvý prvok zoznamu sa pridá hneď na začiatku s hodnotou `None` - to je tzv. fiktívny vrchol, ktorého nasledovník (`next`) bude prvý prvok zoznamu; tento fiktívny vrchol nevyhadzujte
- `prechod()` zrealizuje jeden prechod triedenia min-sort
- `daj_zoznam()` vráti zoznam (typu `list`) všetkých prvkov (dátovej časti `data`) spájaného zoznamu
- do triedy `Zoznam` môžete pridávať ďalšie pomocné metódy

31.5.1 Obmedzenia

- vaše riešenie odovzdajte v súbore `riesenie7.py`, pričom sa v ňom bude nachádzať **len jedna definícia triedy** `Zoznam` a jedna globálna funkcia `min_sort()`.
- v metódach nepoužívajte žiadne atribúty typu `list`, `dict` ani `set` a tiež nevolajte štandardné funkcie `sort` a `sorted`

- nemá zmysel modifikovať podtriedu `Vrchol`, metódu ani funkciu `min_sort()` - testovač aj tak použije ich verzie z tohto zadania
- váš program by nemal vytvárať viac inštancií triedy `Vrchol` ako je prvkov zadanej postupnosti (plus fiktívny vrchol), tiež by nemal modifikovať hodnotu atribútu `data` v týchto vrcholoch, môže meniť len hodnotu atribútu `next`
- prvé dva riadky súboru `riesenie7.py` budú obsahovať:

```
# autor: Janko Hrasko
# uloha: 7. domace zadanie min_sort
```

- zrejme ako autora uvediete svoje meno
- atribút `zac` v triede `Zoznam` musí obsahovať aktuálny spájaný zoznam (s prvkami typu `Zoznam.Vrchol`)
- váš program by nemal počas testovania testovačom nič vypisovať (žiadne vaše testovacie `print()`)

31.5.2 Testovanie

Keď budete spúšťať vaše riešenie na svojom počítači, môžete do súboru `riesenie7.py` pridať testovacie riadky, ktoré ale testovač vidieť nebude, napr.:

```
if __name__ == '__main__':
    post = (4, 30, 8, 31, 48, 19)
    zoz = min_sort(post)
    print(zoz)

    post = 'kohutik jaraby nechod do zahrady'.split()
    zoz = min_sort(post, reverse=True)
    print(zoz)
```

Tento test by vám mal vypísať:

```
[4, 8, 19, 30, 31, 48]
['zahrady', 'nechod', 'kohutik', 'jaraby', 'do']
```

KAPITOLA 32

32. Grafy

Terminológia

Graf je dátová štruktúra, ktorá sa skladá

- z množiny vrcholov **V** = {V1, V2, . . . }
- z množiny hrán **H**, pričom každá hrana je dvojica (v, w), kde v, w `in` **V**
  - ak sú to neusporiadané dvojice, hovoríme tomu **neorientovaný** graf
  - ak sú to usporiadané dvojica, hovoríme tomu **orientovaný** graf

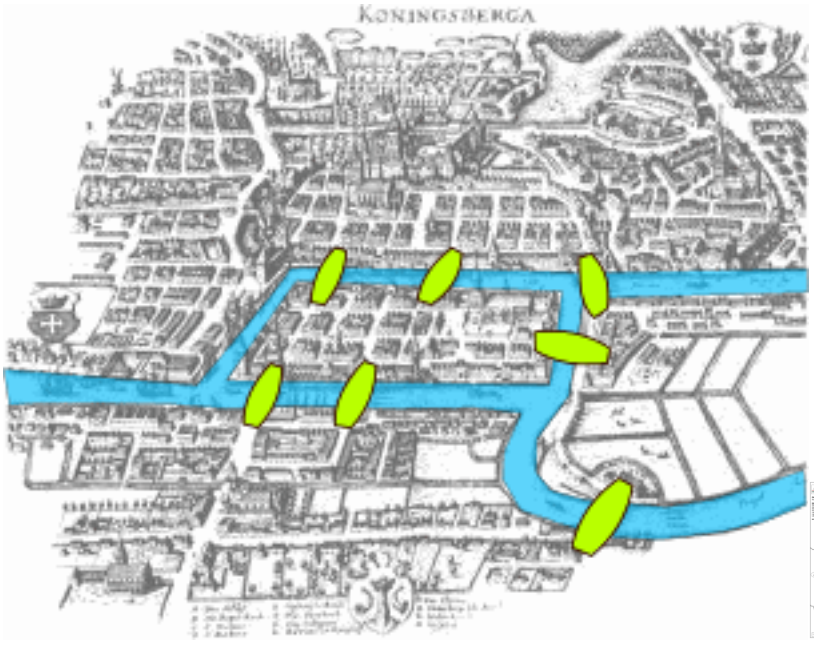
Graf budeme znázorňovať takto:

- vrcholy sú kolieska
- hrany sú spojovníky medzi vrcholmi, pričom, ak je graf orientovaný, tak spojovníkmi sú šípky

Graf najčastejšie používame, keď potrebujeme vyjadriť takéto situácie:

- mestá spojené cestami (najčastejšie je to neorientovaný graf: mestá sú vrcholy, hrany označujú cesty medzi mestami)
- križovatky a ulice v meste (križovatky sú vrcholy, hrany sú ulice medzi križovatkami)
- susediace štáty alebo nejaké oblasti (štáty sú vrcholy, hrany označujú, že nejaké štáty susedia)
- mestská verejná doprava (zastávky sú vrcholy, hrany označujú, že existuje linka, ktorá má tieto dve susediace zastávky)
- skupina ľudí, ktorí sa navzájom poznajú (ľudia sú vrcholy, hrany označujú, že nejaké konkrétne dve osoby sa poznajú)

Rôzne príklady využitia typu graf:



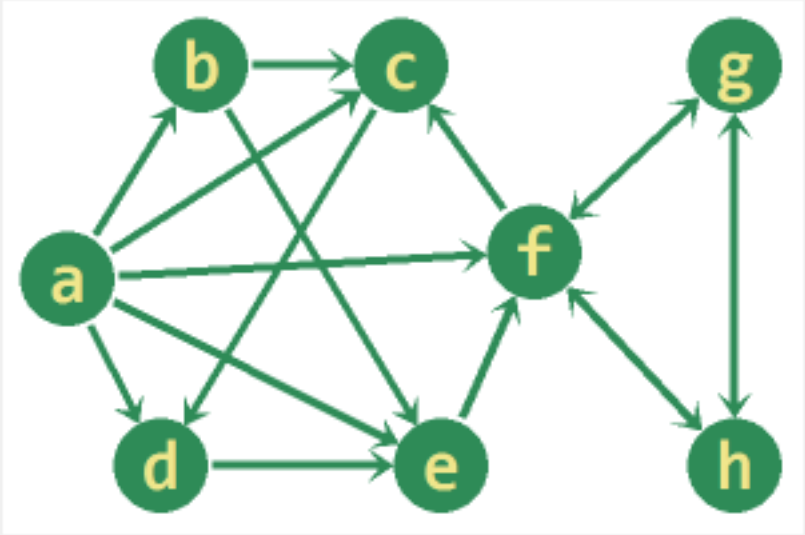
Dôležité pojmy

- ak existuje hrana (v, w) - tak hovoríme, že **v** a **w** sú susediace vrcholy
- v grafe môžeme mať uchované rôzne informácie:
  - v každom vrchole (podobne ako v strome) napr. meno, súradnice v ploche (na mape), ...
  - každá hrana môže mať nejakú hodnotu (tzv. **váha**), vtedy tomu hovoríme **ohodnotený** graf, napr. vzdialenosť dvoch miest, cena cestovného lístka, linky, ktoré spájajú dve zastávky, ...
- **cesta** je postupnosť vrcholov, ktoré sú spojené hranami
  - dĺžka cesty pre neohodnotený graf počet hrán na ceste, t.j. počet vrcholov cesty - 1
  - dĺžka cesty v ohodnotenom grafe je súčet váh na hranách cesty
- **cyklus** je taká cesta, pre ktorú prvý a posledný vrchol sú rovnaké
  - ak graf neobsahuje ani jeden cyklus, hovoríme že je **acyklický**
- hovoríme, že graf je **súvislý** (spojitý), ak pre každé dva vrcholy **v**, **w** `in` **V**, existuje cesta z **v** do **w**
- niekedy bude pre nás dôležité, keď nejaký graf bude **súvislý/nesúvislý** bez cyklov, ale aj **súvislý/nesúvislý** s cyklom
- hrane (v, v) hovoríme **slučka** (toto bude veľmi zriedkavé, ak sa to niekedy objaví, upozorníme na to)
- pojem **komponent** označuje súvislý podgraf, ktorý je disjunktný so zvyškom grafu (neexistuje hrana ku vrcholom vo zvyšku grafu)

Ďalej ukážeme najčastejšie spôsoby reprezentácie grafov. Konkrétna reprezentácia sa potom zvolí väčšinou podľa problémovej oblasti a rôznych obmedzení v zadaní.

32.1 Reprezentácie

Tento konkrétny graf postupne ukážeme v rôznych reprezentáciách



Na rozdiel od dátových štruktúr **spájaný zoznam** a **binárny strom**, pri ktorých sme si ukázali jedinú reprezentáciu pomocou smerníkov, pri dátovej štruktúre **graf** si väčšinou zvolíte tú reprezentáciu, s ktorou sa vám pri tej ktorej konkrétnej úlohe najlepšie pracuje. Nižšie uvedieme niekoľko bežných reprezentácií, v ktorých sa využívajú pythonovské typy `list`, `set` a `dict`. Neskôr uvidíte aj iné spôsoby reprezentácie grafov.

32.1.1 Zoznam množín susedností

Aby sme nemuseli pracovať s jednoznakovými reťazcami ‚a', ‚b', ..., očíslujeme vrcholy číslami 0, 1, 2, ...

Pre čitateľnosť zápisu najprv vytvoríme 8 konštánt `a=0`, `b=1`, `c=2`, ... a pomocou nich zadefinujeme celý graf ako zoznam množín, kde i-ta množina reprezentuje všetkých susedov i-teho vrcholu:

```
a,b,c,d,e,f,g,h = range(8)
graf = [{b,c,d,e,f},   #a
        {c,e},         #b
        {d},           #c
        {e},           #d
        {f},           #e
        {c,g,h},       #f
        {f,h},         #g
        {f,g},         #h
        ]
```

Pre túto štruktúru vieme zistiť, počet vrcholov, počet všetkých hrán, stupeň konkrétneho vrcholu a či medzi dvoma konkrétnymi vrcholmi je hrana:

```
>>> print('pocet vrcholov:', len(graf))
pocet vrcholov: 8
>>> print('pocet hran:', sum(map(len, graf)))
pocet hran: 17
>>> print('stupen vrcholu f:', len(graf[f]))
stupen vrcholu f: 3
>>> print('hrana medzi b, e:', e in graf[b])
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
hrana medzi b, e: True
>>> print('hrana medzi c, a:', a in graf[c])
hrana medzi c, a: False
```

Túto reprezentáciu môžeme „zabaliť“ do triedy napr. takto:

```
class Graf:
    def __init__(self, zoz=None):
        if zoz is None:
            self.zoz = []
        else:
            self.zoz = zoz

    def pridaj_hranu(self, v1, v2):
        while len(self.zoz)-1 < max(v1, v2):
            self.zoz.append(set())
        self.zoz[v1].add(v2)

    def je_hrana(self, v1, v2):
        return v2 in self.zoz[v1]

    def daj_vrcholy(self):
        return list(range(len(self.zoz)))

    def daj_hrany(self):
        return [(v1, v2) for v1 in range(len(self.zoz)) for v2 in self.zoz[v1]]

    def stupen(self, v=None):
        if v is not None:
            return len(self.zoz[v])
        return max(map(len, self.zoz))

    def __str__(self):
        return f'vrcholy: {self.daj_vrcholy()}\nhrany: {self.daj_hrany()}'

    def __repr__(self):
        return f'Graf({self.zoz})'
```

a graf môžeme teraz vytvoriť napr. takto:

```
>>> a,b,c,d,e,f,g,h = range(8)
>>> graf = Graf()
>>> graf.pridaj_hranu(a,b)
>>> graf.pridaj_hranu(b,e)
>>> graf.pridaj_hranu(f,h)
>>> graf.pridaj_hranu(g,f)
>>> graf
Graf([{1}, {4}, set(), set(), set(), {7}, {5}, set()])
>>> print(graf)
vrcholy: [0, 1, 2, 3, 4, 5, 6, 7]
hrany: [(0, 1), (1, 4), (5, 7), (6, 5)]
```

Všimnite si, že sme vyrobili dve rôzne metódy `__repr__()` a `__str__()`. Metóda `__repr__()` sa zavolá, napr. keď v dialógovom režime zapíšeme `>>> graf`. Metóda `__str__()` sa zavolá, napr. keď budeme graf vypisovať príkazom `print()`.

Tak ako sme túto triedu **Graf** definovali, je trochu komplikované vytvoriť izolovaný vrchol, t.j. taký, z ktorého nevy-

chádza žiadna hrana (asi by pomohla metóda `pridaj_vrchol()`).

Ak máme vyrobenú štruktúru grafu (ako zoznam množín susedov), vieme z nej priamo vytvoriť triedu graf:

```
>>> a,b,c,d,e,f,g,h = range(8)
>>> gg = [{b,c,d,e,f},{c,e},{d},{e},{f},{c,g,h},{f,h},{f,g}]
>>> graf = Graf(gg)
>>> print(graf)
vrcholy: [0, 1, 2, 3, 4, 5, 6, 7]
hrany: [(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (1, 2), (1, 4), (2, 3), (3, 4), (4,␣
↪5), (5, 2), (5, 6), (5, 7), (6, 5), (6, 7), (7, 5), (7, 6)]
>>> graf.je_hrana(b, e)
True
>>> graf.je_hrana(c, a)
False
>>> print('stupen vrcholu f:', graf.stupen(f))
stupen vrcholu f: 3
```

V tejto reprezentácii grafu triedou `Graf` o samotnom vrchole nemáme žiadnu špeciálnu informáciu okrem jeho poradového čísla a prislúchajúcej množiny susedov.

Zadefinujeme novú triedu `Graf`, ktorá bude uchovávať zoznam vrcholov, t.j. zoznam objektov typu `Vrchol`:

```
class Graf:

    class Vrchol:
        def __init__(self, meno):
            self.meno = meno
            self.sus = set()

    #-------

    def __init__(self):
        self.zoz = []

    def pridaj_vrchol(self, meno):      # vrati instanciu Vrchol
        for v in self.zoz:
            if v.meno == meno:
                return v                # uz existuje
        novy = self.Vrchol(meno)
        self.zoz.append(novy)
        return novy

    def hladaj_vrchol(self, meno):      # vrati instanciu Vrchol
        for v in self.zoz:
            if v.meno == meno:
                return v
        return None

    def pridaj_hranu(self, v1, v2):
        self.pridaj_vrchol(v1).sus.add(v2)

    def je_hrana(self, v1, v2):
        return v2 in self.hladaj_vrchol(v1).sus

    def daj_vrcholy(self):
        return [v.meno for v in self.zoz]

    def daj_hrany(self):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        return [(v.meno, v2) for v in self.zoz for v2 in v.sus]

    def stupen(self, v=None):
        if v is not None:
            return len(self.hladaj_vrchol(v).sus)
        return max(len(v.sus) for v in self.zoz)

    def __str__(self):
        return f'vrcholy: {self.daj_vrcholy()}\nhrany: {self.daj_hrany()}'
```

S vrcholmi teraz musíme pracovať prostredníctvom ich mien, t.j. jednoznakových reťazcov, napr.

```
>>> graf = Graf()
>>> for v1, v2 in 'ab ac ad ae af bc be cd de ef fc fg fh gf gh hf hg'.split():
        graf.pridaj_hranu(v1, v2)
>>> print(graf)
vrcholy: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
hrany: [('a', 'c'), ('a', 'b'), ('a', 'e'), ('a', 'd'), ('a', 'f'), ('b', 'c'),
('b', 'e'), ('c', 'd'), ('d', 'e'), ('e', 'f'), ('f', 'g'), ('f', 'c'), ('f', 'h'),
('g', 'h'), ('g', 'f'), ('h', 'g'), ('h', 'f')]
>>> print('stupen vrcholu f:', graf.stupen('f'))
stupen vrcholu f: 3
>>> print('stupen grafu:', graf.stupen())
stupen grafu: 5
>>> print('hrana medzi b, e:', graf.je_hrana('b', 'e'))    # alebo graf.je_hrana(*'be
↪')
hrana medzi b, e: True
>>> print('hrana medzi c, a:', graf.je_hrana('c', 'a'))
hrana medzi c, a: False
>>> print('pocet hran:', len(graf.daj_hrany()))
pocet hran: 17
```

Zrejme efektívnejšie by sa táto definícia realizovala nie zoznamom vrcholov (typ `list`), ale asociatívnym poľom (typ `dict`) vrcholov, v ktorom kľúčom by bol identifikátor vrcholu.

32.1.2 Asociatívne pole množín susedností

Pri realizácii reprezentácie **množiny susedností** pomocou tried `Graf` a `Vrchol` sa ukázalo nešikovné, keď sme mali všetky vrcholy (inštancie triedy `Vrchol`) uložené v zozname. Pri práci s takýmito vrcholmi sme museli veľakrát preliezať celý zoznam a hľadať vrchol s daným identifikátorom. Ak by sme namiesto toho použili asociatívne pole, výrazne by sa to zjednodušilo, napr.

```
graf = {'a': set('bcdef'),   # {'b','c','d','e','f'}
        'b': set('ce'),
        'c': set('d'),
        'd': set('e'),
        'e': set('f'),
        'f': set('cgh'),
        'g': set('fh'),
        'h': set('fg'),
       }
```

Zapíšeme to pomocou tried `Graf` a `Vrchol`:

```
class Graf:

    class Vrchol:
        def __init__(self, meno):
            self.meno = meno
            self.sus = set()

        def pridaj_hranu(self, v2):
            self.sus.add(v2)

        def __contains__(self, v2):
            return v2 in self.sus

    #-------

    def __init__(self):
        self.zoz = {}

    def pridaj_vrchol(self, meno):
        if meno not in self.zoz:
            self.zoz[meno] = self.Vrchol(meno)

    def pridaj_hranu(self, v1, v2):
        self.pridaj_vrchol(v1)
        self.zoz[v1].pridaj_hranu(v2)

    def je_hrana(self, v1, v2):
        return v2 in self.zoz[v1]

    def daj_vrcholy(self):
        return list(self.zoz.keys())

    def daj_hrany(self):
        return [(v1, v2) for v1, v in self.zoz.items() for v2 in v.sus]

    def stupen(self, v=None):
        if v  is not None:
            return len(self.zoz[v].sus)
        return max(len(v.sus) for v in self.zoz.values())

    def __str__(self):
        return f'vrcholy: {self.daj_vrcholy()}\nhrany: {self.daj_hrany()}'
```

Otestovať to môžeme rovnako ako sme testovali zoznam množín susedností:

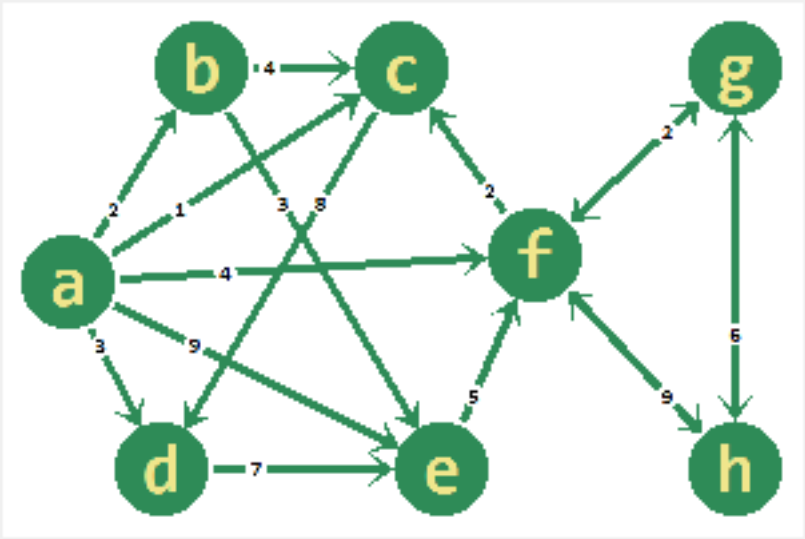
```
graf = Graf()
for v1, v2 in 'ab ac ad ae af bc be cd de ef fc fg fh gf gh hf hg'.split():
    graf.pridaj_hranu(v1, v2)
print(graf)
print('stupen vrcholu f:', graf.stupen('f'))
print('stupen grafu:', graf.stupen())
print('hrana medzi b, e:', graf.je_hrana('b', 'e'))
print('hrana medzi c, a:', graf.je_hrana('c', 'a'))
print('pocet hran:', len(graf.daj_hrany()))
```

a dostavame úplne rovnaké výsledky:

```
vrcholy: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
hrany: [('a', 'e'), ('a', 'f'), ('a', 'c'), ('a', 'd'), ('a', 'b'),
 ('b', 'c'), ('b', 'e'), ('c', 'd'), ('d', 'e'), ('e', 'f'), ('f', 'g'),
 ('f', 'c'), ('f', 'h'), ('g', 'f'), ('g', 'h'), ('h', 'g'), ('h', 'f')]
stupen vrcholu f: 3
stupen grafu: 5
hrana medzi b, e: True
hrana medzi c, a: False
pocet hran: 17
```

32.1.3 Zoznam asociatívnych polí susedností

Ak máme napr. takýto ohodnotený graf:



tak namiesto množiny susedností (prvá reprezentácia grafu) použijeme asociatívne pole (typ `dict`), v ktorom si pri každom susedovi budeme pamätať aj číslo na hrane, t.j. váhu:

```
a,b,c,d,e,f,g,h = range(8)

graf = [{b:2,c:1,d:3,e:9,f:4},      #a
        {c:4,e:3},                  #b
        {d:8},                      #c
        {e:7},                      #d
        {f:5},                      #e
        {c:2,g:2,h:9},              #f
        {f:2,h:6},                  #g
        {f:9,g:6},                  #h
        ]
```

Mohli by sme tu využiť aj reprezentáciu **asociatívne pole množín susedností**, v ktorej by sme množiny opäť nahradili asociatívnymi poľami, napr.

```
a,b,c,d,e,f,g,h = range(8)

graf = {a: {b:2,c:1,d:3,e:9,f:4},
        b: {c:4,e:3},
        c: {d:8},
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        d: {e:7},
        e: {f:5},
        f: {c:2,g:2,h:9},
        g: {f:2,h:6},
        h: {f:9,g:6},
        }
```

Zrejme túto reprezentáciu by sme mali volať **asociatívne pole asociatívnych polí susedností**. Aj tu môžeme čísla vrcholov nahradiť reťazcami:

```
graf = {'a': {'b':2,'c':1,'d':3,'e':9,'f':4},
        'b': {'c':4,'e':3},
        'c': {'d':8},
        'd': {'e':7},
        'e': {'f':5},
        'f': {'c':2,'g':2,'h':9},
        'g': {'f':2,'h':6},
        'h': {'f':9,'g':6},
        }
```

32.1.4 Matica susedností

Graf zapíšeme do dvojrozmernej tabuľky (zoznam zoznamov) veľkosti `n` x `n`, kde `n` je počet vrcholov:

```
graf = [[0,1,1,1,1,1,0,0],
        [0,0,1,0,1,0,0,0],
        [0,0,0,1,0,0,0,0],
        [0,0,0,0,1,0,0,0],
        [0,0,0,0,0,1,0,0],
        [0,0,1,0,0,0,1,1],
        [0,0,0,0,0,1,0,1],
        [0,0,0,0,0,1,1,0],
        ]
```

Často sa namiesto **0** a **1** píšu `False` a `True`.

Ukážme, ako sa zapisuje práca s takouto reprezentáciou:

```
>>> a,b,c,d,e,f,g,h = range(8)
>>> print('pocet vrcholov:', len(graf))
pocet vrcholov: 8
>>> print('pocet hran:', sum(map(sum,graf)))
pocet hran: 17
>>> print('stupen vrcholu f:', sum(graf[f]))
stupen vrcholu f: 3
>>> print('hrana medzi b, e:', graf[b][e]==1)
hrana medzi b, e: True
>>> print('hrana medzi c, a:', graf[c][a]==1)
hrana medzi c, a: False
```

32.1.5 Matica susedností s váhami

Namiesto **0** a **1** v dvojrozmernej tabuľke zapisujeme priamo váhy na príslušných hranách, pričom treba nejako označiť hodnotu, ktorá reprezentuje, že tu nie je hrana, napr.

```
a,b,c,d,e,f,g,h = range(8)
_ = -1        # oznacuje, ze tu nie je hrana, mohla tu byt aj hocijaka ina hodnota␣
↪napr. None

graf = [[_,2,1,3,9,4,_,_],
        [_,_,4,_,3,_,_,_],
        [_,_,_,8,_,_,_,_],
        [_,_,_,_,7,_,_,_],
        [_,_,_,_,_,5,_,_],
        [_,_,2,_,_,_,2,9],
        [_,_,_,_,_,2,_,6],
        [_,_,_,_,_,9,6,_],
        ]
```

Ukážme, ako sa zapisuje práca s takouto reprezentáciou:

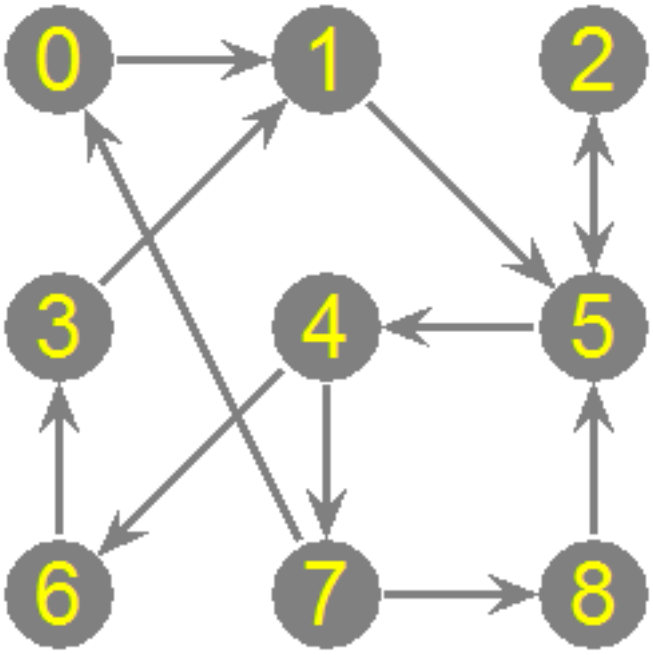
```
>>> print('pocet vrcholov:', len(graf))
pocet vrcholov: 8
>>> print('pocet hran:', sum(1 for r in graf for x in r if x!=_))
pocet hran: 17
>>> print('stupen vrcholu f:', sum(1 for x in graf[f] if x!=_))
stupen vrcholu f: 3
>>> print('hrana medzi b,e:', graf[b][e]!=_)
hrana medzi b,e: True
>>> print('hrana medzi c,a:', graf[c][a]!=_)
hrana medzi c,a: False
>>> print('vaha na hrane medzi b,e:', graf[b][e])
vaha na hrane medzi b,e: 3
```

32.2 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Pre tento daný graf (ručne) zapíšte množinu vrcholov a množinu hrán
   - graf



- odpovedzte áno/nie:
  - je tento graf orientovaný?
  - je tento graf súvislý?
  - je tento graf ohodnotený?
  - je tento graf acyklický?
- ručne zapíšte nejakú cestu (postupnosť vrcholov) z vrcholu **0** do vrcholu **3**

2. Zapíšte graf z úlohy (1) v zadanej reprezentácii
   - zoznam množín susedností

```
graf = [...]
```

   - asociatívne pole množín susedností

```
graf = {...}
```

3. Nakreslite graf (na papieri), ktorý zodpovedá tejto reprezentácii:
   - tabuľka susedností:

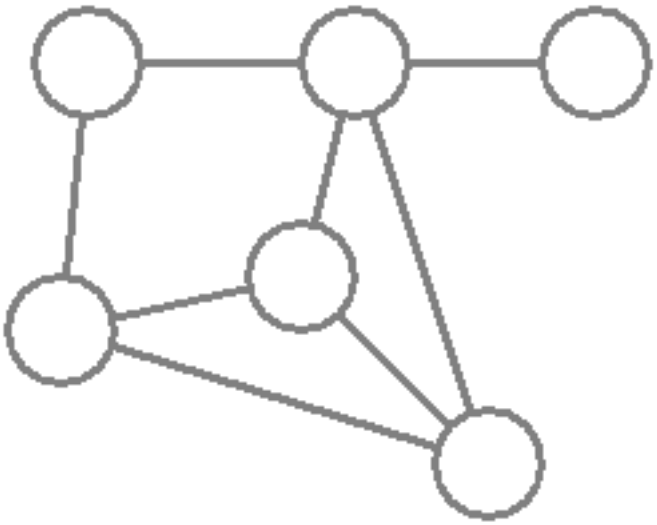
	A	B	C	D	E	F	
A		7	5			1	
B	2			7	3		
C		2				8	
D	1				2	4	
E	6			5			
F		1			8		

   - koľko vrcholov a koľko hrán má tento graf?

4. Zapíšte graf z úlohy (3) v reprezentácii
   - asociatívne pole asociatívnych polí susedností

```
graf = {...}
```

5. Označte vrcholy grafu na obrázku, ak poznáme jeho reprezentáciu
   - graf



   - reprezentácia

```
graf = [{2, 3, 4}, {3, 4}, {0, 3, 4}, {0, 1, 2}, {0, 1, 2, 5}, {4}]
```

   - odpovedzte áno/nie:
     - je tento graf orientovaný?
     - je tento graf súvislý?
     - je tento graf ohodnotený?
     - je tento graf acyklický?
6. Prepíšte graf z úlohy (5) do reprezentácie
   - matica susedností:

```
graf = [[...], ...]
```

7. Podľa vzoru triedy `Graf` pre jednu z reprezentácií (napr. „zoznam množín susedností“) zapíšte metódy pre reprezentáciu
   - matica susedností:

```
class Graf:

    def __init__(self, n):                  # n je pocet vrcholov, t.j. velkost␣
↪matice
        self.matica = [...]                 # zadefinuje prazdny graf (v matici␣
↪su same 0)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def pridaj_hranu(self, v1, v2):
        ...

    def je_hrana(self, v1, v2):
        return ...

    def daj_vrcholy(self):                # vrati mnozinu cisel vrcholov
        return ...

    def daj_hrany(self):                  # vrati mnozinu usporiadanych dvojic
        return ...

    def stupen(self, v=None):             # zisti stupen vrcholu alebo celeho
↪grafu
        if v is not None:
            return ...
        return ...

    def __str__(self):
        return 'vrcholy: {}\nhrany: {}'.format(self.daj_vrcholy(), self.daj_
↪hrany())
```

8. Zadefinujte inštanciu triedy z úlohy (7) tak, aby zodpovedala grafu z úloh (5) a (6)

   - napr.

```
graf = Graf(...)
graf.pridaj_hranu(...)
...
for i in range(len(graf.daj_vrcholy())):
    print('stupen vrcholu', i, 'je', graf.stupen(i))
```

9. Do triedy `Graf` z úlohy (7) dopíšte zadané metódy a otestujte ich na grafe z úlohy (8)

   - `je_neorientovany()` zistí, či je graf orientovaný (vtedy vráti `True`):

```
class Graf:
    ...

    def je_neorientovany(self):
        return ...
```

   - `trojuholniky()` vypíše všetky také trojice vrcholov, ktoré sú navzájom spojené hranami oboma smermi každý s každým

```
class Graf:
    ...

    def trojuholniky(self):
        print(...)
```

10. Do triedy `Graf` z prednášky, ktorá definuje reprezentáciu **asociatívne pole množín susedností** dodefinujte metódy na vykreslenie grafu

    - napr.

```
class Graf:
    canvas = None

    class Vrchol:
        def __init__(self, meno, x, y):
            self.meno = meno
            self.sus = set()
            self.x, self.y = x, y

        ...

        def kresli(self):            # vykresli vrchol ako kruh s textom
            ...

        def kresli_hranu(self, vrchol2):           # vykresli hranu k vrchol2
            ...                                    # netreba kreslit sipky

    #-------

    def __init__(self):
        self.graf = {}

    def pridaj_vrchol(self, meno, x, y):
        ...

    ...

    def kresli(self):                # vykresli cely graf
        ...
```

11. Do triedy z úlohy (10) doprogramujte inicializáciu `__init__()` tak, aby sa popis grafu mohol prečítať z textového súboru

    - inicializácia grafu

```
class Graf:
    ...

    #-------

    def __init__(self, meno_suboru=None):
        self.graf = {...}
```

    - textový súbor by mohol mať napr. takýto formát - každý riadok popisuje jeden vrchol:

```
meno x y zoznam_mien_susedov
```

    - zadefinujte 2 súbory pre grafy z úloh (1) a (5) tak, aby sa výsledné obrázky čo najviac podobali, napr. súbor pre graf z úlohy (5) môže začínať týmito dvoma riadkami:

```
0 130 130 2 3 4
1 50 50 3 4
```

32.3 8. Domáce zadanie

L.I.S.T.

- riešenie odovzdávajte na úlohový server https://list.fmph.uniba.sk/

Vytvoriť dátovú štruktúru `Graf`, ktorá bude implementovať graf reprezentovaný ako **množina hrán**:

```
class Graf:
    def __init__(self, meno_suboru):
        self.graf = set()
        ...

    def pridaj_hranu(self, v1, v2):
        ...

    def je_hrana(self, v1, v2):
        return False

    def daj_vrcholy(self):
        return set()                # vrati mnozinu mien vrcholov

    def daj_hrany(self):
        return set()                # vrati mnozinu dvojic mien vrcholov

    def uloz(self, meno_suboru, typ=1):
        ...
```

V triede môžete dodefinovať ďalšie pomocné metódy ale nie ďalšie atribúty s hodnotami. Trieda dokáže pracovať s týmito tromi typmi súborov:

- **typ=1** označuje takýto tvar súboru:
  - súbor začína riadkom: `#1`
  - každý ďalší riadok popisuje jeden vrchol: meno vrcholu a zoznam mien jeho susedov
  - riadky môžu nasledovať v ľubovoľnom poradí a aj zoznam mien susedov môže byť v ľubovoľnom poradí
  - meno vrcholu je ľubovoľný reťazec, ktorý neobsahuje medzeru
  - napr.

```
#1
a b c d e f
b c e
c d
d e
e f
f c g h
g f h
h f g
```

- **typ=2** označuje takýto tvar súboru:
  - súbor začína riadkom: `#2`
  - každý ďalší riadok popisuje jednu hranu ako dvojicu mien vrcholov oddelených medzerou
  - riadky môžu nasledovať v ľubovoľnom poradí
  - meno vrcholu je ľubovoľný reťazec, ktorý neobsahuje medzeru
  - napr.

```
#2
a c
a b
a e
a d
a f
b c
b e
c d
d e
e f
f g
f c
f h
g h
g f
h g
h f
```

- **typ=3** označuje takýto tvar súboru:
  - súbor začína riadkom: `#3`
  - ďalší riadok obsahuje zoznam mien všetkých vrcholov v nejakom poradí
  - všetky ďalšie riadky popisujú maticu susedností, pričom poradie vrcholov je dané prvým riadkom súboru
  - meno vrcholu je ľubovoľný reťazec, ktorý neobsahuje medzeru
  - napr.

```
#3
a b c d e f g h
0 1 1 1 1 1 0 0
0 0 1 0 1 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 1 1
0 0 0 0 0 1 0 1
0 0 0 0 0 1 1 0
```

Graf sa inicializuje ľubovoľným z týchto formátov (rozpozná sa z prvého riadku súboru). Graf sa dokáže zapísať do súboru (metódou `uloz()`) ľubovoľným z týchto formátov.

Všetky 3 ukážkové súbory vytvoria rovnaký graf.

32.3.1 Obmedzenia

- vaše riešenie odovzdajte v súbore `riesenie8.py`, pričom sa v ňom bude nachádzať **len jedna definícia triedy** `Graf`.
- prvé dva riadky tohto súboru budú obsahovať:

```
# autor: Janko Hrasko
# uloha: 8. domace zadanie graf
```

- zrejme ako autora uvediete svoje meno

- atribút `graf` v triede `Graf` musí obsahovať reprezentáciu grafu množinou hrán (typu `set`, pričom hrany sú dvojice mien vrcholov, t.j. `tuple`)
- váš program by nemal počas testovania testovačom nič vypisovať (žiadne vaše testovacie `print()`)
- testovač bude spúšťať vašu definíciu triedy s rôznymi textovými súbormi, ktoré si môžete stiahnuť z L.I.S.T.u

KAPITOLA 33

33. Algoritmy prehľadávania grafu

Budeme skúmať rôzne algoritmy, ktoré prechádzajú vrcholy grafu v nejakom poradí. Keďže sa zameriame na **neohodnotené neorientované grafy**, zvolíme reprezentáciu zoznam množín susedností. Inštancia triedy `Graf` bude obsahovať atribút (súkromnú premennú) `vrcholy`, ktorý bude zoznamom vrcholov grafu. Samotné vrcholy zadefinujeme vo vnorenej triede `Vrchol` a preto s nim musíme v triede `Graf` pracovať pomocou `self.Vrchol`. Graf obsahuje aj metódy, vďaka ktorým sa bude dať vykresliť do grafickej plochy:

```
import tkinter

class Graf:

    class Vrchol:
        canvas = None
        def __init__(self, meno, x, y):
            self.meno = meno
            self.xy = x, y
            self.sus = set()

        def kresli(self):
            x, y = self.xy
            self.canvas.create_oval(x-10, y-10, x+10, y+10, fill='lightgray')
            self.canvas.create_text(x, y, text=self.meno)

    #-----------------------------------------------

    def __init__(self):
        self.vrcholy = []
        self.canvas = tkinter.Canvas(bg='white', width=600, height=600)
        self.canvas.pack()
        self.Vrchol.canvas = self.canvas

    def pridaj_vrchol(self, x, y):
        cislo = len(self.vrcholy)
        self.vrcholy.append(self.Vrchol(cislo, x, y))
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def pridaj_hranu(self, v1, v2):
        self.vrcholy[v1].sus.add(v2)
        self.vrcholy[v2].sus.add(v1)

    def je_hrana(self, v1, v2):
        return v2 in self.vrcholy[v1].sus

    def kresli_hranu(self, v1, v2):
        self.canvas.create_line(self.vrcholy[v1].xy, self.vrcholy[v2].xy, width=3,␣
↪fill='gray')

    def kresli(self):
        for v1 in range(len(self.vrcholy)):
            for v2 in range(v1+1, len(self.vrcholy)):
                if self.je_hrana(v1, v2):
                    self.kresli_hranu(v1, v2)
        for vrch in self.vrcholy:
            vrch.kresli()
```

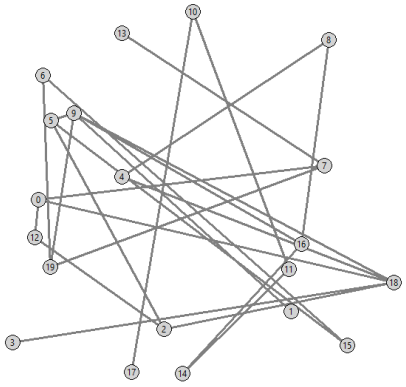
Vygenerujme graf s 20 náhodne rozloženými vrcholmi, pričom sa niektoré vrcholy grafu náhodne pospájajú hranami:

```
#testovanie

import random

graf = Graf()
for i in range(20):
    graf.pridaj_vrchol(random.randrange(20, 580), random.randrange(20, 580))
    for j in range(i):
        if random.randrange(5) == 0:
            graf.pridaj_hranu(i, j)
graf.kresli()
```

Takéto úplne náhodne vygenerované grafy sú veľmi neprehľadné a veľmi ťažko by sa na nich sledovali nejaké prehľadávacie algoritmy.



Vygenerujme preto vrcholy grafu do nejakej mriežky a hranami budeme spájať len niektoré susedné vrcholy v mriežke. Toto generovanie vrcholov zabudujeme priamo do inicializácie triedy `__init__()`, napr. takto:

```
import tkinter
import random

```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
class Graf:
    ...

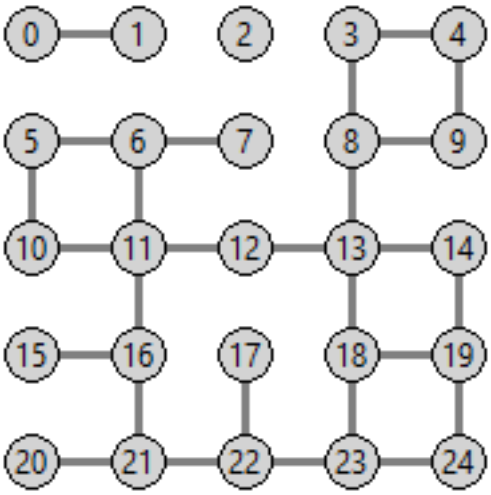
    def __init__(self, n):
        self.vrcholy = []
        for i in range(n):
            for j in range(n):
                self.pridaj_vrchol(j*40+20, i*40+20)
                cislo = len(self.vrcholy) - 1
                if j > 0 and random.randrange(3):
                    self.pridaj_hranu(cislo, cislo-1)
                if i > 0 and random.randrange(3):
                    self.pridaj_hranu(cislo, cislo-n)
        self.canvas = tkinter.Canvas(bg='white', width=600, height=600)
        self.canvas.pack()
        self.Vrchol.canvas = self.canvas
        self.kresli()

     ...

#testovanie

g = Graf(5)
```

Všimnite si, že parameter `n` teraz označuje veľkosť siete vrcholov, ktorá bude mať `n` x `n` vrcholov:



Počet náhodne generovaných hrán bude závisieť od konštánt pri volaní `random.randrange(3)`: vyskúšajte parameter `3` nahradiť `2` a uvidíte „redší“ graf, v ktorom je už menej hrán.

33.1 Algoritmus do hĺbky

Prehľadávanie grafu (*graph traversal*) označuje taký algoritmus, ktorý postupne v nejakom poradí prechádza vrcholy grafu. S každým vrcholom pritom vykoná nejakú akciu (napr. ho zafarbí) a ďalej pokračuje na niektorom z jeho susedných vrcholov. Prehľadávanie sa začne z nejakého (ľubovoľného) vrcholu. Každý vrchol sa pritom navštívi **maximálne raz**.

Ukážeme dva základné algoritmy:

- **prehľadávanie do hĺbky** (*depth-first search*) - funguje podobne ako **preorder** na stromoch: najprv spracuje vrchol a potom postupne pokračuje v prehľadávaní všetkých svojich susedov - opäť rekurzívnym algoritmom
- **prehľadávanie do šírky** (*breadth-first search*) - prehľadáva vrcholy podobne ako prehľadávanie v stromoch po úrovniach - najprv všetky, ktoré majú vzdialenosť len 1, potom všetky, ktoré majú vzdialenosť presne 2, atď.

Pri prehľadávaní grafu pomocou nejakého algoritmu si potrebujeme evidovať, ktoré vrcholy sa už spracovali. Keďže samotný algoritmus môže byť rekurzívny, túto evidenciu **nemôžeme** robiť v lokálnej premennej samotnej funkcie - väčšinou budeme používať nový atribút triedy `Graf` množinovú premennú `visited`:

```
self.visited = set()
```

Algoritmus **do hĺbky** začína prehľadávanie v nejakom vrchole `v1` a využíva metódu triedy `Vrchol`, ktorá ho zafarbí:

```
import tkinter, random

class Graf:

    class Vrchol:
        canvas = None
        def __init__(self, meno, x, y):
            self.meno = meno
            self.xy = x, y
            self.sus = set()

        def kresli(self):
            x, y = self.xy
            self.id = self.canvas.create_oval(x-10, y-10, x+10, y+10, fill='lightgray
↪')
            self.canvas.create_text(x, y, text=self.meno)

        def zafarbi(self, farba):
            self.canvas.itemconfig(self.id, outline=farba, width=2)

    #--------------------------------------------

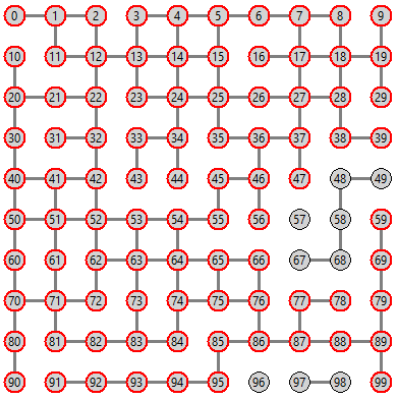
    ...

    def dohlbky(self, v1):
        self.visited.add(v1)
        self.vrcholy[v1].zafarbi('red')
        self.canvas.update()
        self.canvas.after(100)
        for v2 in self.vrcholy[v1].sus:
            if v2 not in self.visited:
                self.dohlbky(v2)          # rekurzivne volanie

#testovanie

g = Graf(10)
g.visited = set()
g.dohlbky(0)
```

Algoritmus **dohlbky** sa naštartoval vo vrchole 0 (v ľavom hornom rohu grafu), postupne navštevoval (a pritom zafarboval) jeho susedov a ich susedov atď. Nenavštívené (nezafarbené) ostali tie časti grafu, ktoré nie sú spojené so zvyškom nejakými hranami, napr.



Atribút `visited` sme tu zadefinovali mimo metód triedy `Graf` - čo samozrejme nie je pekné. Často to budeme riešiť ďalšou metódou, napr.

```
class Graf:

    ...

    def dohlbky(self, v1):
        self.visited.add(v1)
        self.vrcholy[v1].zafarbi('red')
##        self.canvas.update()
##        self.canvas.after(100)
        for v2 in self.vrcholy[v1].sus:
            if v2 not in self.visited:
                self.dohlbky(v2)

    def zafarbi_komponent(self, v1):
        self.visited = set()
        self.dohlbky(v1)

#testovanie

g = Graf(10)
g.zafarbi_komponent(55)
```

Ak metódu `dohlbky()` plánujeme volať len z metódy `zafarbi_komponent()`, samotná funkcia `dohlbky()` tu môže byť vnorená a vtedy aj premenná `visited` nemusí byť atribútom triedy, ale obyčajnou premennou. Ukážeme to na metóde `velkost_komponentu`. Táto metóda dostáva ako parameter jeden vrchol `v1` (jeho poradové číslo), od tohto vrcholu spustí algoritmus do hĺbky, nič pritom teraz už nezafarbuje, „len“ sa spolieha na to, že po jeho skončení bude v premennej `visited` množina všetkých navštívených vrcholov, t.j. tých, ktoré sú spolu s `v1` v jednom komponente:

```
class Graf:

    ...

    def velkost_komponentu(self, v1):
        def dohlbky(v1):
            visited.add(v1)
            for v2 in self.vrcholy[v1].sus:
                if v2 not in visited:
                    dohlbky(v2)
```

(pokračuje na ďalšej strane)

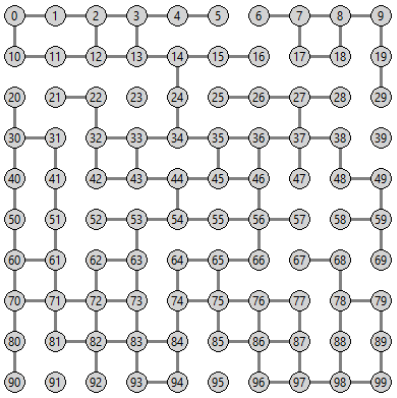
(pokračovanie z predošlej strany)

```
        visited = set()           # lokalna premenna funkcie velkost_komponentu
        dohlbky(v1)
        return len(visited)

#testovanie

g = Graf(10)
```

Dostávame napr.



teraz zavoláme:

```
>>> g.velkost_komponentu(55)
88
>>> g.velkost_komponentu(9)
8
```

Takto môžeme zadefinovať niekoľko verzií algoritmu `dohlbky` pre rôzne využitie: napr. zisťovanie množiny vrcholov v komponente, zisťovanie počtu vrcholov v komponente, zisťovanie počtu komponentov, zafarbovanie vrcholov v rôznych komponentoch rôznymi farbami, rôzne manipulácie s vrcholmi v jednom komponente, ...

Vidíme, že po skončení algoritmu `dohlbky()` bude premenná `visited` obsahovať množinu všetkých navštívených vrcholov. Na základe toho, môžeme ľahko zistiť, či je graf súvislý: zistíme veľkosť komponentu, ktorý obsahuje napr. vrchol `0`, ak obsahuje všetky vrcholy grafu, zrejme je graf súvislý. Zapíšeme to takto:

```
class Graf:

    ...

    def je_suvisly(self):
        def dohlbky(v1):
            visited.add(v1)
            for v2 in self.vrcholy[v1].sus:
                if v2 not in visited:
                    dohlbky(v2)

        visited = set()
        dohlbky(0)
        return len(visited) == len(self.vrcholy)

#testovanie
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
g = Graf(10)
print('graf je suvisly:', g.je_suvisly())
```

33.1.1 Všetky komponenty grafu

Algoritmus `dohlbky()` využijeme nielen pre jeden komponent grafu, ale postupne pre všetky. Ďalšia metóda zafarbuje všetky komponenty grafu - každý komponent inou (náhodnou) farbou:

```
class Graf:

    ...

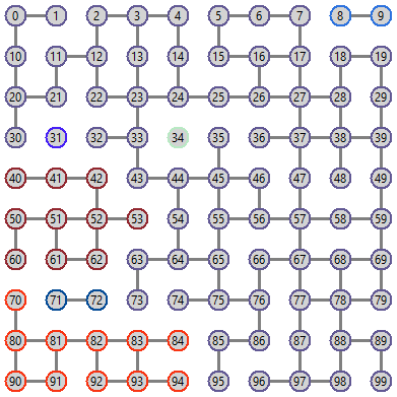
    def zafarbi_komponenty(self):
        def dohlbky(v1, farba):
            visited.add(v1)
            self.vrcholy[v1].zafarbi(farba)
            for v2 in self.vrcholy[v1].sus:
                #self.zafarbi_hranu(v1, v2, farba)
                if v2 not in visited:
                    dohlbky(v2, farba)

        visited = set()
        for v1 in range(len(self.vrcholy)):
            if v1 not in visited:
                dohlbky(v1, f'#{random.randrange(256**3):06x}')

#testovanie

g = Graf(10)
g.zafarbi_komponenty()
```

Dostávame napr.



Tu sa oplatí vygenerovať trochu redší graf: pri generovaní grafu v inicializácii môžeme nastaviť napr. `random.randrange(2)`. Ak ste si uvedomili, že parameter `farba` sa vo vnútri rekurzívnej funkcie `dohlbky` nemení, nepotrebovali sme ho posielať ako parameter, ale mohli sme opäť použiť lokálnu premennú rovnako, ako sme to urobili s premennou `visited`:

```
class Graf:

```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    ...

    def zafarbi_komponenty(self):
        def dohlbky(v1):
            visited.add(v1)
            self.vrcholy[v1].zafarbi(farba)
            for v2 in self.vrcholy[v1].sus:
                #self.zafarbi_hranu(v1, v2, farba)
                if v2 not in visited:
                    dohlbky(v2)

        visited = set()
        for v1 in range(len(self.vrcholy)):
            if v1 not in visited:
                farba = f'#{random.randrange(256**3):06x}'  # lokalna premenna
                dohlbky(v1)
```

Tiež si všimnite, že týmto algoritmom môžeme zabezpečiť aj farbenie všetkých hrán všetkých vrcholov komponentu. Ak chcete vidieť aj zafarbené hrany komponentu, okrem odkomentovania volania metódy `zafarbi_hranu()` ju treba aj zadefinovať. Budeme musieť zasahovať aj do iných metód (`kresli()`, `kresli_hranu()`):

```
class Graf:

    ...

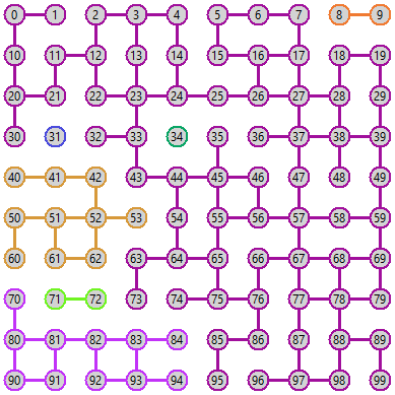
    def kresli_hranu(self, v1, v2):
        self.id[v1, v2] = self.id[v2, v1] = \
            self.canvas.create_line(self.vrcholy[v1].xy, self.vrcholy[v2].xy, width=3,
↪ fill='gray')

    def kresli(self):
        self.id = {}                          # na zapamatanie id kazdej nakreslenej␣
↪hrany
        for v1 in range(len(self.vrcholy)):
            for v2 in range(v1+1, len(self.vrcholy)):
                if self.je_hrana(v1, v2):
                    self.kresli_hranu(v1, v2)
        for vrch in self.vrcholy:
            vrch.kresli()

    def zafarbi_hranu(self, v1, v2, farba):
        self.canvas.itemconfig(self.id[v1, v2], fill=farba)

    ...
```

Teraz (odkomentujeme aj farbenie hrany v `zafarbi_komponenty()`) zafarbený graf vyzerá napr. takto:



Ešte ukážeme metódu, ktorá zisťuje počet komponentov grafu a je veľmi podobná metóde na farbenie komponentov:

```
class Graf:

    ...

    def pocet_komponentov(self):
        def dohlbky(v1):
            visited.add(v1)
            for v2 in self.vrcholy[v1].sus:
                if v2 not in visited:
                    dohlbky(v2)

        visited = set()
        pocet = 0
        for v1 in range(len(self.vrcholy)):
            if v1 not in visited:
                dohlbky(v1)
                pocet += 1
        return pocet

#testovanie

g = Graf(10)
print('pocet komponentov:', g.pocet_komponentov())
g.zafarbi_komponenty()
```

a dozvieme sa, že naposledy zobrazený graf má 7 komponentov.

Vidíme, že algoritmus do hĺbky môžeme použiť napríklad na:

- zistenie počtu komponentov grafu, resp. zisťovanie, či je graf súvislý
- zistenie, či sú dva vrcholy v tom istom komponente
- zisťovanie počtu vrcholov v jednotlivých komponentoch grafu (veľkosti komponentov)
- zisťovanie najväčšieho komponentu grafu
- zafarbovanie vrcholov a hrán jednotlivých komponentov grafu

33.1.2 Nerekurzívny algoritmus do hĺbky

Vieme, že rekurzia v Pythone má obmedzenú hĺbku volaní (okolo 1000), preto pre niektoré väčšie grafy rekurzívny algoritmus `dohlbky()` môže spadnúť na príliš veľa rekurzívnych volaní (napr. už aj pre `Graf(50)` má rekurzia

veľkú šancu spadnúť).

Prepíšme preto rekurzívny algoritmus `dohlbky()` na nerekurzívnu verziu. Ukážeme to na metóde `velkost_komponentu()`, ktorá používa rekurzívnu vnorenú funkciu `dohlbky()`. Na odstránenie rekurzie využijeme zásobník: mohli by sme použiť už predtým definovanú triedu `Stack`, ale vystačíme si s pomocným zoznamom s metódami `append()` (namiesto `push`) a `pop()`. Nerekurzívny algoritmus môže vyzerať napr. takto:

```
class Graf:

    ...

    def velkost_komponentu(self, v1):     # povodna rekurzivna verzia
        def dohlbky(v1):
            visited.add(v1)
            for v2 in self.vrcholy[v1].sus:
                if v2 not in visited:
                    dohlbky(v2)

        visited = set()
        dohlbky(v1)
        return len(visited)

    def velkost_komponentu1(self, v1):    # upravena nerekurzivna verzia
        visited = set()
        stack = [v1]          # vloz do zasobnika startovy vrchol
        while stack:          # kym je zasobnik neprazdny
            v1 = stack.pop() # vyber z vrchu zasobnika
            visited.add(v1)
            for v2 in self.vrcholy[v1].sus:
                if v2 not in visited:
                    stack.append(v2)    # vloz do zasobnika namiesto volania␣
↪dohlbky(v2)
        return len(visited)

#testovanie

g = Graf(10)
print(g.velkost_komponentu(0))
print(g.velkost_komponentu1(0))
```

Hoci tento algoritmus vyzerá v poriadku, ak ho otestujeme na nejakom jednoduchom grafe, dá sa zistiť, že niektorý vrchol sa bude spracovávať (navštevovať) viackrát a to môže byť v niektorých situáciách neprijateľné. Ak by sme si algoritmus odtrasovali ručne, zistili by sme, že niektoré čísla vrcholov sa dostávajú do zásobníka viackrát (napr. vrchol, ktorý ešte nebol navštívený, ale je susedom viacerých už navštívených vrcholov). Jednoducho to môžeme preveriť aj tak, že do tohto algoritmu vložíme počítadlo navštívených vrcholov a po skončení algoritmu porovnáme tento počet so skutočným počtom vrcholov komponentu. Napr. pridáme premennú `pocet`:

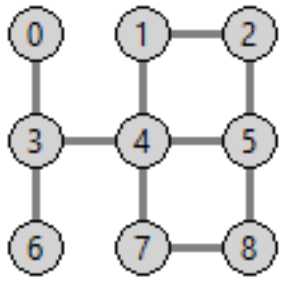
```
1  class Graf:
2
3      ...
4
5      def velkost_komponentu1(self, v1):
6          visited = set()
7          stack = [v1]              # vloz do zasobnika startovy vrchol
8          pocet = 0
9          while stack:              # kym je zasobnik neprazdny
10             v1 = stack.pop()      # vyber z vrchu zasobnika
11             visited.add(v1)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            pocet += 1
            for v2 in self.vrcholy[v1].sus:
                if v2 not in visited:
                    stack.append(v2)    # vloz do zasobnika namiesto volania␣
↪dohlbky(v2)
        return len(visited), pocet
```

Ak teraz spustíme `velkost_komponentu1()` s rôzne veľkými a s rôzne hustými grafmi, často dostaneme dve rôzne čísla: počet vrcholov v komponente a počet navštívených vrcholov je rôzny. Napr. už aj pre tento graf, ktorý má veľkosť komponentu 9 (je súvislý), navštívil niektoré vrcholy viackrát a vidíme:




```
>>> g = Graf(3)
>>> g.velkost_komponentu1(0)
(9, 11)
```

Môžeme to odkrokovať aj ručne:

```
riadok v1    stack           visited                     pocet
============================================================
6       0    []              {}
9       0    [0]             {}                          0
12      0    []              {0}                         1
15      0    [3]             {0}                         1
12      3    []              {0,3}                       2
15      3    [4,6]           {0,3}                       2
12      6    [4]             {0,3,6}                     3
15      6    [4]             {0,3,6}                     3
12      4    []              {0,3,4,6}                   4
15      4    [1,5,7]         {0,3,4,6}                   4
12      7    [1,5]           {0,3,4,6,7}                 5
15      7    [1,5,8]         {0,3,4,6,7}                 5
12      8    [1,5]           {0,3,4,6,7,8}               6
15      8    [1,5,5]         {0,3,4,6,7,8}               6
12      5    [1,5]           {0,3,4,5,6,7,8}             7
15      5    [1,5,2]         {0,3,4,5,6,7,8}             7
12      2    [1,5]           {0,2,3,4,5,6,7,8}           8
15      2    [1,5,1]         {0,2,3,4,5,6,7,8}           8
12      1    [1,5]           {0,1,2,3,4,5,6,7,8}         9
15      1    [1,5]           {0,1,2,3,4,5,6,7,8}         9
12      5    [1]             {0,1,2,3,4,5,6,7,8}         10
15      5    [1]             {0,1,2,3,4,5,6,7,8}         10
12      1    []              {0,1,2,3,4,5,6,7,8}         11
15      1    []              {0,1,2,3,4,5,6,7,8}         11
16                           len(visited) == 9
```

Správne by mal nerekurzívny algoritmus vyzerať takto:

```
class Graf:

    ...

    def velkost_komponentu1(self, v1):
        visited = set()
        stack = [v1]
        while stack:
            v1 = stack.pop()
            if v1 not in visited:    # kontrola, ci uz nebol spracovany
                visited.add(v1)
                for v2 in self.vrcholy[v1].sus:
                    if v2 not in visited:
                        stack.append(v2)
        return len(visited)
```

Teda, každý vrchol, ktorý vyberieme zo zásobníka (`stack.pop()`), ešte skontrolujeme, či už náhodou nebol spracovaný skôr.

33.2 Algoritmus do šírky

Ak by sme potrebovali prehľadávať napr. binárny strom po úrovniach (algoritmus do šírky), potrebovali by sme na to dátovú štruktúru **rad** (**queue**). Podobne budeme postupovať aj pri algoritme pre graf. Tento algoritmus sa bude od nerekurzívneho algoritmu do hĺbky líšiť len tým, že namiesto `stack` použijeme `queue` (a teda zmeníme aj metódu na výber z radu - namiesto `pop()` použijeme `pop(0)`):

```
class Graf:

    ...

    def velkost_komponentu2(self, v1):
        visited = set()
        queue = [v1]                # vloz do radu startovy vrchol
        while queue:                # kym je rad neprazdny
            v1 = queue.pop(0)       # vyber zo zaciatku radu
            if v1 not in visited:
                visited.add(v1)
                for v2 in self.vrcholy[v1].sus:
                    if v2 not in visited:
                        queue.append(v2)    # vloz na koniec radu
        return len(visited)
```

Takto zapísaný algoritmus bude robiť presne to isté ako nerekurzívny algoritmus `velkost_komponentu1()`. Rozdiel bude v tom, že sa všetky vrcholy grafu navštívi v inom poradí: najprv sa spracujú všetky najbližšie susediace vrcholy ku štartovému; potom sa spracujú všetky ich ešte nenavštívené susedné vrcholy atď. Aby sme lepšie videli toto poradie navštevovania vrcholov, môžeme pridať výpis poradových čísel:

```
class Graf:

    ...

    def velkost_komponentu2(self, v1):
        visited = set()
        queue = [v1]
```

(pokračuje na ďalšej strane)

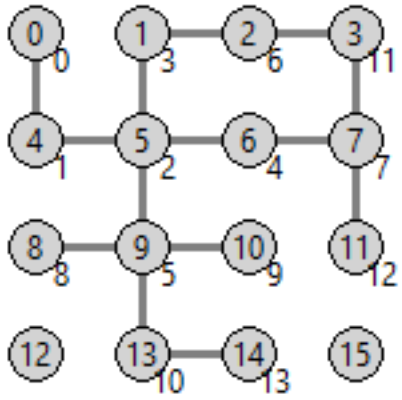
(pokračovanie z predošlej strany)

```
        pocet = 0
        while queue:
            v1 = queue.pop(0)
            if v1 not in visited:
                visited.add(v1)
                x, y = self.vrcholy[v1].xy
                self.canvas.create_text(x+10, y+10, text=pocet)
                pocet += 1
                for v2 in self.vrcholy[v1].sus:
                    if v2 not in visited:
                        queue.append(v2)
        return len(visited)

#testovanie

g = Graf(4)
print(g.velkost_komponentu2(0))
```

dostávame takýto obrázok:



Vďaka tejto veľmi užitočnej vlastnosti algoritmu ho môžeme jednoducho vylepšiť: pri každom spracovávanom vrchole si budeme pamätať aj jeho momentálnu vzdialenosť - teda niečo ako úroveň „vnorenia“ (vzdialenosť od štartu). Štartový vrchol prehľadávania bude mať úroveň 0 (vzdialenosť 0), všetci jeho bezprostrední susedia budú mať úroveň 1 (vzdialenosť 1 od štartu) a každý ďalší vrchol bude mať úroveň o 1 väčšiu ako vrchol, od ktorého sa sem prišlo.

Program prepíšeme tak, že pri každom vrchole v rade (v štruktúre `queue`) si budeme pamätať aj jeho úroveň a pri spracovaní vrcholu túto úroveň vypíšeme (algoritmus sme už premenovali na `dosirky()`):

```
class Graf:

    ...

    def dosirky(self, v1):
        visited = set()
        queue = [(v1, 0)]                                   # vlozili sme vrchol a␣
↪jeho uroven
        while queue:
            v1, uroven = queue.pop(0)
            if v1 not in visited:
                visited.add(v1)
                x, y = self.vrcholy[v1].xy
                self.canvas.create_text(x+12, y+12, text=uroven)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

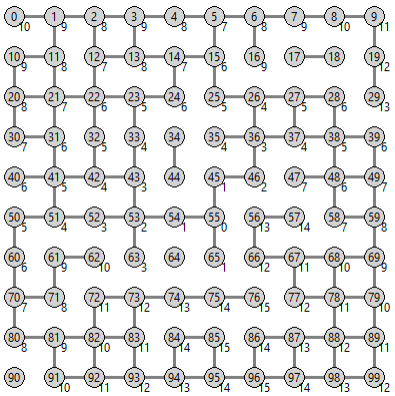
```
##                  self.canvas.update()          # spomalovanie vypisu
##                  self.canvas.after(100)
                for v2 in self.vrcholy[v1].sus:
                    if v2 not in visited:
                        queue.append((v2, uroven+1))        # vlozili sme vrchol a␣
↪jeho uroven

#testovanie

g = Graf(10)
g.dosirky(55)
```

Uvedomte si, že vrchol v ktorom začíname prehľadávanie má číslo úrovne 0, jeho susedia majú úroveň 1, ich susedia úroveň 2, ... Toto číslo vyjadruje vzdialenosť každého vrcholu od štartového.

Dostávame napr. takýto výpis:



V niektorých situáciách je vhodnejšie namiesto výpisu tohto čísla do grafickej plochy ho uložiť priamo do atribútu vo vrchole. (Namiesto `self.canvas.create_text...` zapíšeme napr. `self.vrcholy[v1].uroven = uroven`)

Tento algoritmus môžeme použiť aj na zafarbovanie vrcholov grafu podľa toho, ako ďaleko sú od nejakého konkrétneho vrcholu:

```
class Graf:

    ...

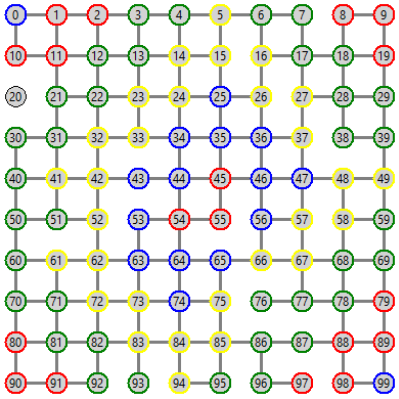
    def zafarbi(self, v1, farby):
        visited = set()
        queue = [(v1, 0)]
        while queue:
            v1, uroven = queue.pop(0)
            if v1 not in visited:
                visited.add(v1)
                self.vrcholy[v1].zafarbi(farby[uroven % len(farby)])
                for v2 in self.vrcholy[v1].sus:
                    if v2 not in visited:
                        queue.append((v2, uroven+1))
```

V parametri `farby` posielame zoznam nejakých farieb, pričom samotný štartovací vrchol sa zafarbí farbou `farby[0]`, jeho susedia farbou `farby[1]`, ich susedia ďalšou farbou, atď. Ak je farieb v zozname `farby` menej ako rôznych vzdialeností v grafe, táto funkcia začne používať farby opäť od začiatku zoznamu.

Algoritmus sme spustili pre 8 farieb, pričom úmyselne sme prvé dve farby dali rovnaké, aj ďalšie dve, atď. aby sa lepšie zobrazilo farbenie, teda štartový vrchol 55 aj jeho najbližší susedia sú červení, vrcholy vo vzdialenosti 2 a 3 od štartu 55 sú modré, atď.

```
g = Graf(10)
g.zafarbi(55, ['red', 'red', 'blue', 'blue', 'yellow', 'yellow', 'green', 'green'])
```

Dostávame takéto farbenie grafu:



33.2.1 Vzdialenosť a najkratšia cesta

Algoritmus do šírky môžeme využiť aj na zisťovanie vzájomnej vzdialenosti ľubovoľných dvoch vrcholov v grafe. Pod vzdialenosťou tu rozumieme dĺžku **najkratšej cesty** z jedného vrcholu do druhého. Ak takáto cesta neexistuje (vrcholy sú v rôznych komponentoch grafu), metóda môže vrátiť napr. hodnotu -1:

```
class Graf:

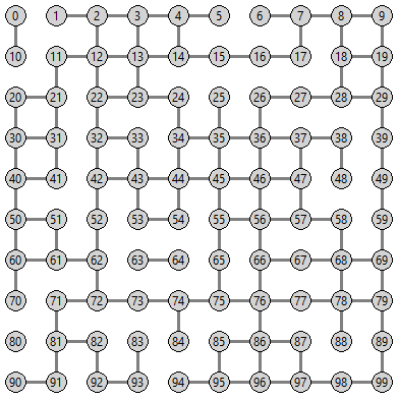
    ...

    def vzdialenost(self, v1, ciel):
        visited = set()
        queue = [(v1, 0)]
        while queue:
            v1, uroven = queue.pop(0)
            if v1 not in visited:
                visited.add(v1)
                if v1 == ciel:
                    return uroven
                for v2 in self.vrcholy[v1].sus:
                    if v2 not in visited:
                        queue.append((v2, uroven+1))
        return -1

#testovanie

g = Graf(10)
print(g.vzdialenost(11, 55))
```

Metóda `vzdialenost()` funguje na princípe algoritmu **dosirky** len pritom nič nezafarbuje. Napr. pre graf



môžeme otestovať:

```
>>> g.vzdialenost(55, 11)
8
>>> g.vzdialenost(11, 55)
8
>>> g.vzdialenost(1, 99)
17
>>> g.vzdialenost(0, 1)
-1
```

Rozšírime tento algoritmus nielen na zistenie vzdialenosti dvoch vrcholov, ale aj na zapamätanie najkratšej cesty. Použijeme takúto ideu: každý vrchol si namiesto úrovne bude pamätať svojho predchodcu, t.j. číslo vrcholu, z ktorého sme sem prišli. Na záver, keď dorazíme do cieľa, pomocou predchodcov vieme postupne skonštruovať výslednú postupnosť - cestu medzi dvoma vrcholmi. Ak pri zostavovaní cesty dostaneme ako predchodcu **None** znamená to, že tento vrchol už nemá svojho predchodcu lebo je to štartový vrchol (odtiaľto sme naštartovali hľadanie cesty).

Algoritmus hľadania najkratšej cesty v grafe:

```
class Graf:

    ...

    def cesta(self, v1, ciel):
        visited = set()
        queue = [(v1, None)]
        while queue:
            v1, predchodca = queue.pop(0)
            if v1 not in visited:
                visited.add(v1)
                self.vrcholy[v1].pred = predchodca
                if v1 == ciel:
                    vysl = []
                    while v1 is not None:
                        vysl.append(v1)
                        v1 = self.vrcholy[v1].pred
                    return vysl[::-1]
                for v2 in self.vrcholy[v1].sus:
                    if v2 not in visited:
                        queue.append((v2, v1))
        return []
```

Pomocou tejto metódy vieme teda zistiť nielen vzdialenosť dvoch vrcholov ale aj konkrétnu postupnosť vrcholov, cez ktoré treba prejsť. Napr. pre graf z predchádzajúceho obrázka dostávame:

```
>>> g.cesta(55, 11)
[55, 45, 35, 34, 24, 23, 13, 12, 11]
>>> g.cesta(11, 55)
[11, 12, 13, 23, 24, 34, 35, 45, 55]
>>> g.cesta(1, 99)
[1, 2, 3, 13, 23, 24, 34, 35, 36, 37, 47, 57, 58, 68, 69, 79, 89, 99]
>>> g.cesta(1, 0)
[]
```

Táto metóda vytvorí postupnosť vrcholov, ktoré tvoria najkratšiu cestu medzi dvoma vrcholmi. Túto cestu môžeme tiež aj vykresliť, napr. takto:

```
class Graf:

    ...

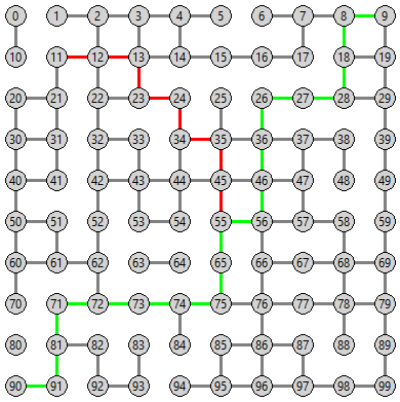
    def cesta(self, v1, v2l):
        ...

    def kresli_cestu(self, v1, ciel, farba):
        cesta = self.cesta(v1, ciel)
        if cesta:
            for v1 in range(len(cesta)-1):
                #self.vrcholy[cesta[v1]].zafarbi(farba)
                self.zafarbi_hranu(cesta[v1], cesta[v1+1], farba)
            #self.vrcholy[cesta[-1]].zafarbi(farba)
        else:
            print('cesta neexistuje')
```

a v predchádzajúcom grafe zobrazíme najkratšie cesty:

```
>>> g.cesta(55, 11)
[55, 45, 35, 34, 24, 23, 13, 12, 11]
>>> g.kresli_cestu(55, 11, 'red')
>>> g.cesta(90, 9)
[90, 91, 81, 71, 72, 73, 74, 75, 65, 55, 56, 46, 36, 26, 27, 28, 18, 8, 9]
>>> g.kresli_cestu(90, 9, 'lime')
>>> g.kresli_cestu(10, 19, 'green')
cesta neexistuje
```

V jednom obrázku vidíme obe cesty:



Zhrňme použitie algoritmu do šírky:

- podobne ako do hĺbky, vieme zistiť vrcholy, ktoré patria do jedného komponentu a teda aj zistiť počet komponentov
- vieme zistiť (zafarbiť) vrcholy (alebo hrany), ktoré sú rovnako vzdialené od daného vrcholu
- vieme zistiť vzdialenosť, resp. najkratšiu cestu medzi dvoma vrcholmi v grafe

33.3 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

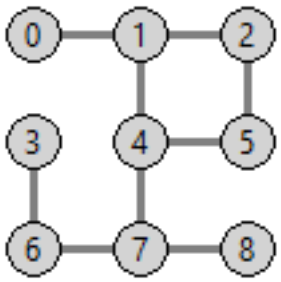
1. **Ručne** odtrasujte
   - tento algoritmus **do hĺbky**:

```
def poradie(self, v1):

    def dohlbky(v1):
        vysledok.append(v1)
        for v2 in sorted(self.vrcholy[v1].sus):
            if v2 not in vysledok:
                dohlbky(v2)

    vysledok = []       # zoznam namiesto množiny
    dohlbky(v1)
    return vysledok
```

   - pre tento graf:



   - zistite:

```
>>> g.poradie(0)
[....]
>>> g.poradie(4)
[....]
>>> g.poradie(7)
[....]
```

2. Skopírujte triedu `Graf` z prednášky a
   - upravte, resp. dodefinujte niektoré metódy (rekurzívne verzie):

```
def zafarbi_komponent(self, v1, farba='red')    # zafarbí všetky vrcholy␣
↪komponentu
def max_komponent(self)           # vrati velkost najvacsieho komponentu
def vsetky_komponenty(self)       # vráti zoznam vel'kostí všetkých komponentov
def v_komponente(self, v1, v2)    # zistí, či su dva vrcholy v tom istom␣
↪komponente
```

3. Metódy z úlohy (2) otestujte pre malé aj väčšie grafy
   - napr.

```
>>> g = Graf(5)
>>> g.zafarbi_komponent(12, 'blue')
>>> g.max_komponent()
...
>>> g.vsetky_komponenty()
[....]
>>> g.v_komponente(0, 24)
...
>>> g = Graf(15)
...
```

4. Pre všetky metódy z úlohy (2) vytvorte a otestujte ich nerekurzívne verzie
   - metódy:

```
def zafarbi_komponent1(self, v1, farba='red')
def max_komponent1(self)
def vsetky_komponenty1(self)
def v_komponente1(self, v1, v2)
```

5. **Ručne** odtrasujte pre graf z úlohy (1)
   - tento algoritmus **do šírky**:

```
def poradie2(self, v1):
    vysledok = []          # zoznam namiesto množiny
    queue = [v1]
    while queue:
        v1 = queue.pop(0)
        if v1 not in vysledok:
            vysledok.append(v1)
            for v2 in sorted(self.vrcholy[v1].sus):    # usporiadany zoznam␣
↪susedov
                if v2 not in vysledok:
                    queue.append(v2)
    return vysledok
```

   - zistite:

```
>>> g.poradie2(0)
[....]
>>> g.poradie2(4)
[....]
>>> g.poradie2(7)
[....]
```

6. Spojazdnite metódy z prednášky

- ktoré fungujú na princípe algoritmu do šírky

```
def vzdialenost(self, v1, ciel)
def cesta(self, v1, ciel)
def kresli_cestu(self, v1, ciel, farba)
```

- otestujte tieto metódy na väčšom grafe, napr.

```
>>> g = Graf(15)
>>> g.vzdialenost(0, 14)
...
>>> g.kresli_cestu(14, 210)
```

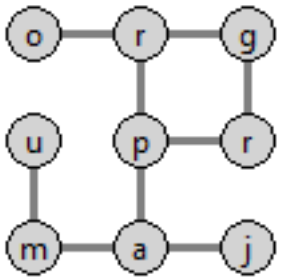
7. Zapíšte metódy, ktoré premenujú vrcholy (zmenia vo vrcholoch atribút `meno`) v zadanom poradí
   - metódy (kde `zoznam_mien` je dostatočne dlhý zoznam nových hodnôt pre mená vrcholov):

```
def premenuj_dohlbky(self, v1, zoznam_mien)
def premenuj_dosirky(self, v1, zoznam_mien)
```

   - napr. pre graf z úlohy (1):

```
>>> g.premenuj_dohlbky(4, list('programuj'))
```

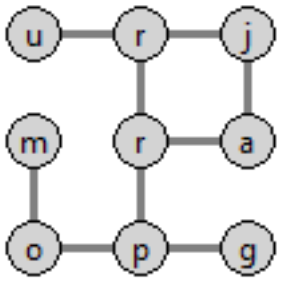
   sa graf premenuje takto:



   - a tiež pre ten istý graf:

```
>>> g.premenuj_dosirky(7, list('programuj'))
```

   sa graf premenuje takto:



8. Upravte metódy `poradie()` z úlohy (1) a `poradie2()` z úlohy (5) tak, aby namiesto postupnosti čísel vrcholov obe metódy vracali postupnosti mien vrcholov
   - napr. pre graf z úlohy (1):

```
>>> g.premenuj_dohlbky(4, list('programuj'))
>>> g.poradie(4)
['p', 'r', 'o', 'g', 'r', 'a', 'm', 'u', 'j']
>>> g.premenuj_dosirky(7, list('programuj'))
>>> g.poradie2(7)
['p', 'r', 'o', 'g', 'r', 'a', 'm', 'u', 'j']
```

33.4 9. Domáce zadanie

L.I.S.T.

- riešenie odovzdávajte na úlohový server https://list.fmph.uniba.sk/

Vytvorte dátovú štruktúru `Graf`, ktorá bude implementovať neorientovaný neohodnotený graf reprezentovaný ako **asociatívne pole množín susedov**:

```
class Graf:
    def __init__(self, meno_suboru):
        self.vrcholy = {}
        ...

    def vo_vzdialenosti(self, v1, od, do=None):
        return set()

    def max(self, v1):
        return 0, set()

    def vsetky(self, v1):
        return []

    def v_strede(self, v1, v2):
        return set()
```

V triede môžete dodefinovať ďalšie pomocné metódy ale nie ďalšie atribúty s hodnotami.

Vstupný súbor má tento formát:

- prvý riadok obsahuje zoznam mien všetkých vrcholov v nejakom poradí
- každý ďalší riadok popisuje jednu alebo viac hrán ako dvojice mien vrcholov oddelených medzerou
- meno vrcholu je ľubovoľný reťazec, ktorý neobsahuje medzeru
- napr. `'subor1.txt'`:

```
5 0 3 1 7 8 4 2 6
5 4
8 7
1 0 3 6
6 7 5 2 2 1
1 4
4 7
```

  - kde `1 0 3 6` označuje tieto 2 hrany: `1-0`, `3-6`

Všetky metódy triedy `Graf` (zrejme okrem `__init__()`) budú pracovať s grafom algoritmom **do šírky**:

- **vo_vzdialenosti(self, v1, od, do=None)** vráti množinu mien vrcholov, ktorých vzdialenosť od daného vrcholu `v1` nie je menšia ako parameter `od` a nie je väčšia ako parameter `do`; ak má parameter `do` hodnotu `None`, označuje to, že `do=od`
- **max(self, v1)** vráti dvojicu (`tuple`): maximálnu vzdialenosť od vrcholu `v1` k nejakému vrcholu v grafe a množinu všetkých vrcholov s touto vzdialenosťou; napr. pre izolovaný vrchol (nemá susedov) metóda vráti `0` a jednoprvkovú množinu samotného vrcholu
- **vsetky(self, v1)** vráti zoznam (`list`), v ktorom `i`-ty prvok obsahuje množinu všetkých tých vrcholov v grafe, ktoré majú vzdialenosť od vrcholu `v1` práve `i`; napr. pre izolovaný vrchol metóda vráti `[{v1}]`
- **v_strede(self, v1, v2)** vráti množinu všetkých tých vrcholov v grafe, ktoré majú rovnakú vzdialenosť od `v1` ako od `v2`, t.j. vrcholy ležia presne medzi oboma danými vrcholmi; predpokladajte, že zadané vrcholy sú rôzne; môžete využiť volanie metód `vsetky(v1)` a `vsetky(v2)`

33.4.1 Obmedzenia

- vaše riešenie odovzdajte v súbore `riesenie9.py`, pričom sa v ňom bude nachádzať **len jedna definícia triedy** `Graf`.
- prvé dva riadky tohto súboru budú obsahovať:

```
# autor: Janko Hrasko
# uloha: 9. domace zadanie do_sirky
```

- zrejme ako autora uvediete svoje meno
- atribút `vrcholy` v triede `Graf` musí obsahovať reprezentáciu grafu asociatívnym poľom (typu `dict`, kde kľúčom je meno vrcholu a hodnotou je množina mien susedov)
- váš program by nemal počas testovania testovačom nič vypisovať (žiadne vaše testovacie `print()`)
- testovač bude spúšťať vašu definíciu triedy s rôznymi textovými súbormi, ktoré si môžete stiahnuť z L.I.S.T.u

33.4.2 Testovanie

Graf zo súboru `subor1.txt` si môžete nakresliť ako 9 vrcholov postupne s číslami od 0 do 8 v troch radoch po troch vrcholoch. Potom test:

```
if __name__ == '__main__':
    g = Graf('subor1.txt')
    for v1, i, j in ('4', 1, 3), ('2', 4, 4), ('1', 5, 7), ('7', 1, 3):
        print(f'vo_vzdialenosti({v1!r}, {i}, {j}) =', g.vo_vzdialenosti(v1, i, j))
    print("max('3') =", g.max('3'))
    print("vsetky('5') = ", g.vsetky('5'))
    for v1, v2 in ('6', '8'), ('3', '7'):
        print(f'v_strede({v1!r}, {v2!r}) =', g.v_strede(v1, v2))
```

by mal vrátiť tento výsledok:

```
vo_vzdialenosti('4', 1, 3) = {'6', '7', '5', '1', '3', '8', '0', '2'}
vo_vzdialenosti('2', 4, 4) = {'6', '8'}
vo_vzdialenosti('1', 5, 7) = set()
vo_vzdialenosti('7', 1, 3) = {'6', '5', '3', '1', '8', '4', '0', '2'}
max('3') = (5, {'2', '0'})
vsetky('5') =  [{'5'}, {'2', '4'}, {'7', '1'}, {'6', '8', '0'}, {'3'}]
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
v_strede('6', '8') = {'4', '2', '0', '7', '5', '1'}
v_strede('3', '7') = {'6'}
```

KAPITOLA 34

34. Backtracking

34.1 Generovanie štvoríc čísel

Napíšme program, ktorý vypíše všetky 4-ciferné čísla zložené len z cifier `0` až `n-1` a pritom žiadna cifra sa neopakuje viackrát. Takéto štvorice budeme generovať štyrmi vnorenými for-cyklami:

```
n = 4
pocet = 0
for i in range(n):
    for j in range(n):
        for k in range(n):
            for l in range(n):
                if i!=j and i!=k and j!=k and i!=l and j!=l and k!=l:
                    print(i, j, k, l)
                    pocet += 1
print('pocet =', pocet)
```

Program okrem všetkých vyhovujúcich štvoríc vypíše aj ich počet (zrejme pre `n=4` ich bude **24**, t.j. **4!**).

Program môžeme aj trochu vylepšiť: vnútorné cykly nebudú zbytočne skúšať rôzne možnosti napr. vtedy, keď `i == j`:

```
n = 4
pocet = 0
for i in range(n):
    for j in range(n):
        if i!=j:
            for k in range(n):
                if i!=k and j!=k:
                    for l in range(n):
                        if i!=l and j!=l and k!=l:
                            print(i, j, k, l)
                            pocet += 1
print('pocet =', pocet)
```

Jednoduchým vylepšením môžeme generovať nie čísla (z nejakého intervalu), ale slová, ktoré sú zložené len zo zadaných písmen vo vstupnom slove:

```
slovo = 'ahoj'
pocet = 0
for i in slovo:
    for j in slovo:
        if i!=j:
            for k in slovo:
                if i!=k and j!=k:
                    for l in slovo:
                        if i!=l and j!=l and k!=l:
                            print(i + j + k + l)
                            pocet += 1
print('pocet =', pocet)
```

Program takto vypíše 24 rôznych slov.

Takýto spôsob riešenia však nie je najvhodnejší. Ak by sme napr. potrebovali nie štvorice, ale všeobecne `n`-tice číslic, alebo, ak by sme chceli komplikovanejšiu podmienku (cifry sa môžu raz opakovať) a pod., budeme musieť použiť nejaký iný spôsob riešenia.

34.2 Rekurzia

Budeme riešiť generovanie n-tíc čísel pomocou rekurzívnej funkcie. Začnime s úlohou, v ktorej generujeme všetky štvorice čísel z intervalu `0` až `n-1`, pričom čísla sa môžu aj opakovať. Úlohu budeme riešiť pre ľubovoľné `n`. n-ticu postupne vytvárame v n-prvkovom zozname `pole` a keď je kompletná, tak ju vypíšeme:

```
n = 4
pole = [0] * n

def generuj(i):             # pridaj i-ty prvok do zoznamu pole
    for j in range(n):
        pole[i] = j
        if i == n - 1:
            print(*pole)
        else:
            generuj(i + 1)

generuj(0)
```

Toto riešenie generuje **všetky** `n`-tice čísel a nielen tie, v ktorých sú všetky cifry rôzne (teda 256 štvoríc). Pridáme teraz test na to, či je vygenerovaná `n`-tica naozaj správne riešenie: t.j. žiadne dva prvky nesmú byť rovnaké. Využijeme na to množinu, ktorú vytvoríme zo všetkých `n` prvkov zoznamu. Ak je aj táto množina n-prvková, žiaden prvok v zozname sa neopakoval a teda máme vyhovujúce riešenie:

```
n = 4
pole = [0] * n

def generuj(i):
    for j in range(n):
        pole[i] = j
        if i == n - 1:
            if len(set(pole)) == n:
                print(*pole)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        else:
            generuj(i + 1)

generuj(0)
```

Program teraz generuje 24 rôznych štvoríc.

Ak by sme ale potrebovali robiť kontrolu priebežne **pred každým pridaním** ďalšieho čísla, môžeme zapísať:

```
n = 4
pole = [0] * n

def generuj(i):
    for j in range(n):
        if j not in pole[:i]:    # kontrola, ci sme uz j nepouzili predtym
            pole[i] = j
            if i == n - 1:
                print(*pole)
            else:
                generuj(i + 1)

generuj(0)
```

34.2.1 Zapuzdrime

Zvykli sme si zapisovať komplexnejšie programy ako metódy triedy, pričom namiesto globálnych premenných pracujeme s atribútmi triedy. Zapíšme rekurzívne generovanie `n`-tíc ako metódu triedy `Uloha`. Začneme s jednoduchou verziou, v ktorej sa ešte nekontroluje, či sa nejaké cifry opakujú:

```
class Uloha:
    def __init__(self, n):
        self.n = n
        self.pole = [0] * n

    def generuj(self, i):
        for j in range(self.n):
            self.pole[i] = j
            if i == self.n - 1:
                print(*self.pole)
                self.pocet += 1
            else:
                self.generuj(i + 1)

    def ries(self):
        self.pocet = 0
        self.generuj(0)
        print('pocet =', self.pocet)

Uloha(4).ries()
```

Program vypíše 256 štvoríc.

Doplňme kontrolu, či vygenerovaná štvorica vyhovuje: do metódy **vyhovuje** zapíšme kontrolu kompletnej štvorice:

```
class Uloha:
    def __init__(self, n):
        self.n = n
        self.pole = [0] * n

    def vyhovuje(self):
        return len(set(self.pole)) == self.n

    def generuj(self, i):
        for j in range(self.n):
            self.pole[i] = j
            if i == self.n - 1:
                if self.vyhovuje():              # len tie n-tice, ktore vyhovuju
                    print(*self.pole)
                    self.pocet += 1
            else:
                self.generuj(i + 1)

    def ries(self):
        self.pocet = 0
        self.generuj(0)
        print('pocet =', self.pocet)

Uloha(4).ries()
```

Takéto riešenie už dáva správny výsledok. Treba si ale uvedomiť, že algoritmus zbytočne generuje dosť veľa štvoríc (256), z ktorých cez výstupnú kontrolu (tesne pred výpisom pripravenej n-tice) prejde len niekoľko z nich (24).

Ukážeme riešenie, v ktorom sa vnárame do rekurzie len vtedy, keď doterajšia vygenerovaná časť riešenia vyhovuje podmienkam. Pripravili sme pomocnú funkciu `moze(i, j)`, ktorá otestuje, či momentálna hodnota `j` môže byť priradená na `i`-tu pozíciu výsledného zoznamu `pole`:

```
class Uloha:
    def __init__(self, n):
        self.n = n
        self.pole = [0] * n

    def moze(self, i, j):      # ci na i-tu poziciu mozeme dat j
        return j not in self.pole[:i]

    def generuj(self, i):
        for j in range(self.n):
            if self.moze(i, j):
                self.pole[i] = j
                if i == self.n - 1:
                    print(*self.pole)
                    self.pocet += 1
                else:
                    self.generuj(i + 1)

    def ries(self):
        self.pocet = 0
        self.generuj(0)
        print('pocet =', self.pocet)

Uloha(4).ries()
```

Takže testujeme nie až keď je vytvorená kompletná n-tica, ale vždy pred pridaním ďalšej hodnoty.

Pri riešení niektorých úloh môžeme na kontrolu pridávaného prvku do riešenia využiť ďalšie pomocné štruktúry. Nám by sa tu hodila napr. množina už použitých čísel, prípadne asociatívne pole (`dict`), v ktorom budeme evidovať počet výskytov každého čísla, napr.

```
class Uloha:
    def __init__(self, n):
        self.n = n
        self.pole = [0] * n

    def moze(self, j):
        return self.vyskyty[j] == 0

    def generuj(self, i):
        for j in range(self.n):
            if self.moze(j):
                self.pole[i] = j
                self.vyskyty[j] += 1
                if i == self.n - 1:
                    print(*self.pole)
                    self.pocet += 1
                else:
                    self.generuj(i + 1)
                self.vyskyty[j] -= 1

    def ries(self):
        self.pocet = 0
        self.vyskyty = {i:0 for i in range(self.n)}      # zatial su vsetky vyskyty 0
        self.generuj(0)
        print('pocet =', self.pocet)

Uloha(4).ries()
```

Vidíme tu nový veľmi dôležitý detail:

- vždy, keď zaevidujeme nový prvok do pripravovaného zoznamu (`self.pole[i] = j`), zaznačíme ďalší výskyt tejto hodnoty aj do zoznamu `vyskyty`
- lenže, takto zaznačený výskyt treba niekedy aj zrušiť, lebo by sme príslušnú hodnotu už nikdy nedostali na žiadnej inej pozícii:
  - samotné zrušenie výskytu (`self.vyskyty[j] -= 1`) robíme na konci tela cyklu, lebo vtedy sa bude skúšať na túto pozíciu priradiť iná hodnota

Počítadlo výskytov by sme mohli využiť aj v úlohe, kde treba generovať n-tice, ale tak, že každá cifra sa môže raz opakovať. Zmeníme podmienku vo funkcii `moze`:

```
def moze(self, j):
    return self.vyskyty[j] <= 1
```

Pozrite si ešte nasledovný variant rekurzívnej funkcie `generuj`:

```
def generuj(self, i):
    if i == self.n:
        print(*self.pole)
        self.pocet += 1
    else:
        for j in range(self.n):
            if self.moze(j):
                self.pole[i] = j
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
                self.vyskyty[j] += 1
                self.generuj(i + 1)
                self.vyskyty[j] -= 1
```

Takto zapísaná metóda funguje presne rovnako ako predchádzajúca verzia, len sa testovanie, či je kompletné riešenie, presunulo na začiatok (aj s malou zmenou podmienky). V praxi si väčšinou vyberáme ten zápis, ktorý sa nám v konkrétnej situácii viac hodí (nižšie uvidíme aj ďalšie varianty).

34.3 Backtracking

Takéto generovanie všetkých možných n-tíc, ktoré spĺňajú nejakú podmienku, je základom algoritmu, ktorému hovoríme **prehľadávanie s návratom**, resp. **backtracking**. Tento algoritmus teda:

- v každom kroku vyskúša všetky možnosti (napr. pomocou for-cyklu)
- pre každú možnosť preverí, či spĺňa podmienky (napr. metódou `moze(...)`)
- ak áno, zaeviduje si túto hodnotu (hovoríme tomu, že sa **zaznačí ťah**) väčšinou v nejakých interných štruktúrach (zoznam, množina, asociatívne pole, ...)
- skontroluje, či riešenie nie je už kompletné a vtedy ho spracuje (napr. ho vypíše, alebo niekam zaznačí)
- ak riešenie ešte nie je kompletné, treba generovať ďalší prvok, čo zabezpečí rekurzívne volanie
- na konci cyklu, v ktorom sa postupne skúšajú všetky možnosti, treba ešte **vrátiť stav** pred zaevidovaním hodnoty (hovoríme tomu, že sa **odznačí ťah**)

Schematicky to môžeme zapísať takto:

```
def backtracking(param):
    for i in vsetky_moznosti:
        if moze(i):
            zaznac_tah(i)
            if hotovo:
                vypis_riesenie()
            else:
                backtracking(param1)
            odznac_tah(i)
```

alebo druhá schéma:

```
def backtracking(param):
    if hotovo:
        vypis_riesenie()
    else:
        for i in vsetky_moznosti:
            if moze(i):
                zaznac_tah(i)
                backtracking(param1)
                odznac_tah(i)
```

Zapamätajte si, že je veľmi dôležité vrátiť ešte pred koncom cyklu všetko, čo sme zaevidovali na začiatku cyklu. Uvedomte si, že bez tohto vrátenia pôvodného stavu sa tento algoritmus stáva obyčajným prehľadávaním do hĺbky, ktorý sa len rekurzívne vnára hlbšie a hlbšie ...

Pomocou **backtrackingu** môžeme riešiť úlohy typu:

- matematické hlavolamy (8 dám na šachovnici, domček jedným ťahom, kôň na šachovnici, sudoku, . . .)
- rôzne problémy na grafoch (nájsť cestu s najmenším ohodnotením z A do B, vyhodiť max. počet hrán, aby platila nejaká podmienka)

Vo všeobecnosti je backtracking **veľmi neefektívny algoritmus** (patrí medzi algoritmy tzv. *hrubá sila*, *brute force*), pomocou ktorého sa dá vyriešiť veľké množstvo úloh (postupne vyskúšam všetky možnosti). V praxi sa mnoho problémov dá vyriešiť oveľa efektívnejšie. Pre backtracking väčšinou platí, že jeho **zložitosť** je exponenciálna, t.j. čím je úloha väčšieho rozsahu, tým rýchlejšie rastie čas na jeho vyriešenie. Pri algoritmoch triedenia sme videli obrovský rozdiel vo výkone bublinkového a rýchleho (quick-sort) triedenia. Pritom bublinkové triedenie má zložitosť rádovo `n**2` a pre väčší zoznam je už neprijateľne pomalé. Čo potom algoritmy, ktorých zložitosť je rádovo `2**n`.

34.3.1 Dámy na šachovnici

Budeme riešiť takýto hlavolam: na šachovnicu s 8x8 treba umiestniť 8 dám tak, aby sa navzájom neohrozovali (vodorovne, zvislo ani uhlopriečne). Zrejme v každom riadku a tiež v každom stĺpci musí byť práve jedna dáma. Dámy očíslujeme číslami od 0 do 7 tak, že `i`-ta dáma sa bude nachádzať v `i`-tom riadku. Potom každé rozloženie dám na šachovnici môžeme reprezentovať osmicou čísel (v zozname `riesenie`): `i`-te číslo potom určuje číslo stĺpca `i`-tej dámy.

Na riešenie úlohy využime schému backtrackingu:

```
def hladaj(self, i):          # i-ta dáma v i.riadku -> hľadá pre ňu vhodný stĺpec
    for j in range(self.n):
        if self.moze(i, j):
            # zaznač položenie dámy
            self.riesenie[i] = j
            ...
            if i == self.n - 1:             # ak je už hotovo
                print(*self.riesenie)
                self.pocet += 1
            else:
                self.hladaj(i + 1)
            # odznač položenie dámy
            self.riesenie[i] = None
            ...
```

Na rýchle otestovanie, či je nejaká pozícia ohrozená ostatnými dámami použijeme 3 pomocné množiny:

- `stlpec` - tu si pamätáme, v ktorých stĺpcoch je už nejaká dáma
- `u1` - pamätáme si, v ktorých uhlopriečkach (sprava doľava) je už nejaká dáma
- `u2` - pamätáme si, v ktorých uhlopriečkach (zľava doprava) je už nejaká dáma

Využijeme vlastnosť políčok na šachovnici, vďaka ktorej vieme zistiť, či sa dve políčka nachádzajú na tej istej uhlopriečke:

- pre uhlopriečky sprava (hore) doľava (dole) platí, že súčet čísla riadku a čísla stĺpca je rovnaký (napr. (3,1),(2,2),(0,4),. . . ležia na jednej uhlopriečke)
- pre uhlopriečky zľava (hore) doprava (dole) platí, že rozdiel čísla riadku a čísla stĺpca je rovnaký (napr. (3,1),(4,2),(6,4),. . . ležia na jednej uhlopriečke)

Všetky tieto tri pomocné množiny musia byť na začiatku prázdne:

```
self.stlpec = set()
self.u1 = set()
self.u2 = set()
```

Zaznačiť ťah potom znamená:

- zapamätať si ho v zozname `riesenie`
- zaznačiť si obsadený príslušný stĺpec v množine `stlpec`
- zaznačiť si obsadenú príslušnú uhlopriečku v množine `u1`
- zaznačiť si obsadenú príslušnú 2. uhlopriečku v množine `u2`

```
# zaeviduj i-tu dámu v i riadku a j stĺpci
self.riesenie[i] = j
self.stlpec.add(j)
self.u1.add(i+j)
self.u2.add(i-j)
```

Kontrola jednej konkrétnej dámy (metóda `moze()`) potom len skontroluje tieto tri množiny:

```
def moze(self, i, j):          # ci moze polozit damu na poziciu (i, j)
    return j not in self.stlpec and i+j not in self.u1 and i-j not in self.u2
```

Celý program:

```
class Damy:
    def __init__(self, n):
        self.n = n
        self.riesenie = [None] * n

    def moze(self, i, j):          # ci moze polozit damu na poziciu (i, j)
        return j not in self.stlpec and i+j not in self.u1 and i-j not in self.u2

    def hladaj(self, i):           # i-ta dama v i.riadku -> hlada pre nu vhodny stlpec
        for j in range(self.n):
            if self.moze(i, j):
                # zaznac polozenie damy
                self.riesenie[i] = j
                self.stlpec.add(j)
                self.u1.add(i + j)
                self.u2.add(i - j)
                if i == self.n - 1:
                    print(*self.riesenie)
                    self.pocet += 1
                else:
                    self.hladaj(i + 1)
                # odznac polozenie damy
                self.riesenie[i] = None
                self.stlpec.remove(j)
                self.u1.remove(i + j)
                self.u2.remove(i - j)

    def ries(self):
        self.stlpec = set()
        self.u1 = set()
        self.u2 = set()
        self.pocet = 0
        self.hladaj(0)
        print('pocet rieseni:', self.pocet)

Damy(8).ries()
```

Rekurzívnu metódu `hladaj()` by sme mohli prepísať aj podľa druhej schémy backtrackingu:

```
def hladaj(self, i):           # i-ta dama v i.riadku -> hlada pre nu stlpec
    if i == self.n:
        print(*self.riesenie)
        self.pocet += 1
    else:
        for j in range(self.n):
            if self.moze(i, j):
                # zaznac polozenie damy
                self.riesenie[i] = j
                self.stlpec.add(j)
                self.u1.add(i + j)
                self.u2.add(i - j)
                self.hladaj(i + 1)
                # odznac polozenie damy
                self.riesenie[i] = None
                self.stlpec.remove(j)
                self.u1.remove(i + j)
                self.u2.remove(i - j)
```

Výsledný program aj teraz generuje rovnaký výsledok.

Niekedy potrebujeme organizovať backtrackingovú funkciu tak, že priamo v parametroch funkcie dostane konkrétnu hodnotu ťahu - vtedy si ho najprv zaznačí, potom vygeneruje ďalší ťah (rekurzívne sa zavolá) a na koniec odznačí ťah.

Úlohu s dámami môžeme zapísať aj takto:

```
class Damy:
    def __init__(self, n):
        self.n = n
        self.riesenie = [None] * n

    def moze(self, i, j):          # ci moze polozit damu na poziciu (i, j)
        return j not in self.stlpec and i+j not in self.u1 and i-j not in self.u2

    def dalsia(self, i, j):         # poloz dalsiu damu na (i, j)
        # zaznac polozenie damy
        self.riesenie[i] = j
        self.stlpec.add(j)
        self.u1.add(i+j)
        self.u2.add(i-j)

        if i == self.n-1:
            print(*self.riesenie)
            self.pocet += 1
        else:
            for k in range(self.n):
                if self.moze(i+1, k):
                    self.dalsia(i+1, k)

        # odznac polozenie damy
        self.riesenie[i] = None
        self.stlpec.remove(j)
        self.u1.remove(i+j)
        self.u2.remove(i-j)

    def ries(self):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        self.stlpec = set()
        self.u1 = set()
        self.u2 = set()
        self.pocet = 0
        for j in range(self.n):
            self.dalsia(0,j)
        print('pocet rieseni:', self.pocet)

Damy(8).ries()
```

Backtracking sme namiesto metódy `hladaj()` zapísali metódou `dalsia()`.

Ďalšia verzia tohto programu ukazuje, ako môžeme vykresliť nájdené riešenie (metóda `vypis()`). Tiež sme na začiatok backtrackingu pridali jeden test, vďaka ktorému program korektne skončí po nájdení a vypísaní jedného riešenia (ak nám jedno stačí):

```
class Damy:
    def __init__(self, n):
        self.n = n
        self.riesenie = [None] * n

    def moze(self, i, j):          # ci moze polozit damu na poziciu (i,j)
        return j not in self.stlpec and i+j not in self.u1 and i-j not in self.u2

    def vypis(self):
        for k in range(self.n):
            r = ['.'] * self.n
            r[self.riesenie[k]] = 'o'
            print(' '.join(r))
        print('=' * 2 * self.n)

    def hladaj(self, i):
        if self.pocet:                # uz mame 1 riesenie, nemusime dalej pokracovat
            return
        for j in range(self.n):
            if self.moze(i, j):
                # zaznac polozenie damy
                self.riesenie[i] = j
                self.stlpec.add(j)
                self.u1.add(i+j)
                self.u2.add(i-j)
                if i == self.n-1:
                    self.vypis()
                    self.pocet += 1
                else:
                    self.hladaj(i+1)
                # odznac polozenie damy
                self.riesenie[i] = None
                self.stlpec.remove(j)
                self.u1.remove(i+j)
                self.u2.remove(i-j)

    def ries(self):
        self.stlpec = set()
        self.u1 = set()
        self.u2 = set()
        self.pocet = 0
```

(pokračuje na ďalšej strane)

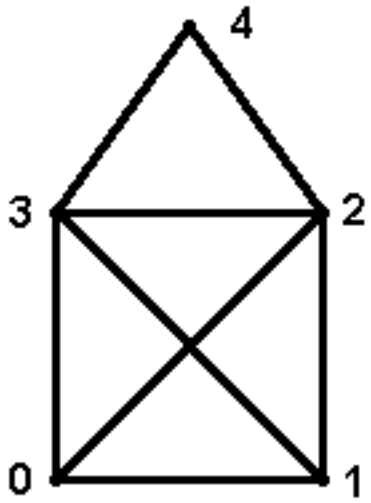
(pokračovanie z predošlej strany)

```
        self.hladaj(0)
        # print('pocet rieseni:', self.pocet)
        if self.pocet == 0:
            print('ziadne riesenie')

Damy(8).ries()
```

34.3.2 Domček jedným ťahom

Budeme riešiť takúto úlohu: potrebujeme zistiť, koľkými rôznymi spôsobmi sa dá nakresliť takýto domček jedným ťahom. Pri kreslení môžeme po každej čiare prejsť len raz.



Obrázok domčeka je vlastne neorientovaný graf s 5 vrcholmi. Každé nájdené riešenie vypíšeme v tvare postupnosti vrcholov, cez ktoré sa prechádza pri kreslení domčeka. Keďže hrán je v grafe 8, tak táto postupnosť bude obsahovať presne 9 vrcholov: jeden štartový a 8 nasledovných vrcholov.

Graf budeme reprezentovať čo najjednoduchšie, napr. pomocou zoznamu množín susedností. Pre každý z vrcholov si treba pamätať množinu jeho susedov:

```
graf = [{1,2,3},       # susedia vrcholu 0
        {0,2,3},       # susedia vrcholu 1
        {0,1,3,4},     # susedia vrcholu 2
        {0,1,2,4},     # susedia vrcholu 3
        {2,3},         # susedia vrcholu 4
        ]
```

Kompletný program postupne vytvára zoznam `riesenie` s postupnosťou prechádzaných vrcholov grafu:

```
class Domcek:
    def __init__(self):
        self.g = [{1,2,3}, {0,2,3}, {0,1,3,4}, {0,1,2,4}, {2,3}]

    def hladaj(self):
        v1 = self.riesenie[-1]
        for v2 in self.g[v1]:
            # zaznac tah
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            self.riesenie.append(v2)
            self.g[v1].remove(v2)
            self.g[v2].remove(v1)
            if len(self.riesenie) == 9:
                print(*self.riesenie)
                self.pocet += 1
            else:
                self.hladaj()
            # odznac tah
            self.riesenie.pop()
            self.g[v1].add(v2)
            self.g[v2].add(v1)

    def ries(self):
        self.pocet = 0
        for i in range(len(self.g)):    # postupne vyskusa pre vsetky vrcholy
            self.riesenie = [i]
            self.hladaj()
        print('pocet rieseni:', self.pocet)

Domcek().ries()
```

- prvý vrchol riešenia (štartový) sa do zoznamu priraďuje ešte pred zavolaním backtrackingu v štartovej metóde `ries`: keďže môžeme začínať z ľubovoľného vrcholu, v tejto metóde štartujeme backtracking v cykle postupne so všetkými možnými začiatočnými vrcholmi
- v backtrackingovej metóde `hladaj`:
  - `v1` obsahuje zatiaľ posledný vrchol riešenia, na ktorý bude teraz nadväzovať nasledovný vrchol `v2`
  - **zaznačenie ťahu** uskutočníme tak, že práve prejdenú hranu dočasne z grafu odstránime (graf je neorientovaný preto treba odstraňovať oba smery tejto hrany)
  - riešenie je kompletné vtedy, keď obsahuje presne `9` vrcholov
  - **odznačenie ťahu** (teda návrat do pôvodného stavu pre zaznačením) urobíme vrátením dočasne odstránenej hrany

34.3.3 Sudoku

Pomocou prehľadávania do hĺbky vyriešime veľmi populárny hlavolam Sudoku. Hracia plocha sa skladá z 9x9 políčok. Na začiatku sú do niektorých políčok zapísané čísla z intervalu <1,9>. Úlohou je zapísať aj do zvyšných políčok čísla 1 až 9 tak, aby v každom riadku a tiež v každom stĺpci bolo každé z čísel práve raz. Okrem toho je celá plocha rozdelená na 9 menších štvorcov veľkosti 3x3 - aj v každom z týchto štvorcov musí byť každé z čísel práve raz.

Napr. súbor s konkrétnym zadaním `'sudoku.txt'` môže vyzerať napr. takto:

```
0 6 0 9 0 0 0 7 0
0 4 0 8 0 0 0 0 0
0 0 0 0 5 0 3 0 0
0 0 0 0 0 0 0 0 0
0 9 0 0 4 0 0 6 0
8 0 5 0 6 0 0 0 0
7 0 8 3 0 0 0 0 4
3 0 0 0 2 1 0 0 8
0 0 0 0 0 0 0 0 2
```

- hodnota `0` označuje, že príslušné políčko je zatiaľ prázdne

Samotná „backtrackingová“ rekurzívna procedúra najprv nájde prvé zatiaľ voľné políčko (je tam hodnota 0). Potom sa sem pokúsi zapísať všetky možnosti čísel 1 až 9. Zakaždým otestuje, či zapisované číslo sa v príslušnom riadku, stĺpci a tiež v malom 3x3 štvorci ešte nenachádza:

```
def backtracking(self):
    i, j = najdi_volne()       # nájdi voľné políčko
    if nenasiel:               # už nie je žiadne ďalšie voľné políčko
        vypis_riesenie()
    else:
        for cislo in range(1, 10):         # pre všetky možnosti čísel od 1 do 9
            if moze(i, j, cislo):
                zaznac_tah()
                backtracking()
                odznac_tah()
```

Aby sa nám čo najjednoduchšie testovalo, či sa nejaké číslo ešte nenachádza v príslušnom riadku, stĺpci a 3x3 štvorci, pre každý riadok, stĺpec aj štvorec 3x3 zavedieme množinovú premennú, t.j. 9-prvkové množinové zoznamy pre riadky a stĺpce a dvojrozmerný 3x3 zoznam (matica) množín pre menšie štvorce:

```
self.vstlpci = [set(),set(),set(),set(),set(),set(),set(),set(),set()]
self.vriadku = [set(),set(),set(),set(),set(),set(),set(),set(),set()]
self.v3x3    = [[set(),set(),set()], [set(),set(),set()], [set(),set(),set()]]
```

Ak budeme teraz potrebovať zistiť, či sa v `j`-tom stĺpci nachádza dané `cislo` zapíšeme:

```
if cislo in self.vstlpci[j]: ...
```

podobne pre i-ty riadok:

```
if cislo in self.vriadku[j]: ...
```

alebo v príslušnom malom štvorci 3x3 (chceme zistiť štvorec, ktorý obsahuje políčko `(i,j)` teda `i` aj `j` delíme `3`):

```
if cislo in self.v3x3[i // 3][j // 3]: ...
```

Podobným spôsobom budeme do týchto množín vkladať nové čísla (pri zaznačovaní ťahu), resp. ich z množín odoberať (pri odznačovaní).

Všimnite si, že tieto množiny sa musia zaplniť počiatočnými hodnotami už pri čítaní vstupného súboru. Kompletný program:

```
class Sudoku:
    def __init__(self, subor):
        with open(subor) as subor:
            self.tab = [[int(i) for i in r.split()] for r in subor]
        self.vstlpci = [set() for i in range(9)]
        self.vriadku = [set() for i in range(9)]
        self.v3x3    = [[set(), set(), set()] for i in range(3)]
        for i in range(9):
            for j in range(9):
                cislo = self.tab[i][j]
                if cislo:
                    self.vstlpci[j].add(cislo)
                    self.vriadku[i].add(cislo)
                    self.v3x3[i // 3][j // 3].add(cislo)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def vypis(self):
        if self.pocet == 0:                  # vypise len prve riesenie
            for riadok in self.tab:
                print(*riadok)
            print('=' * 17)

    def moze(self, i, j, cislo):
        return (cislo not in self.vstlpci[j] and
                cislo not in self.vriadku[i] and
                cislo not in self.v3x3[i // 3][j // 3])

    def backtracking(self):
        i,j = 0,0
        while i < 9 and self.tab[i][j]:
            j += 1
            if j == 9:
                i += 1
                j = 0
        if i == 9:                       # uz nie je ziadne dalsie volne policko
            self.vypis()
            self.pocet += 1
        else:
            for cislo in range(1, 10):
                if self.moze(i, j, cislo):
                    self.tab[i][j] = cislo
                    self.vstlpci[j].add(cislo)
                    self.vriadku[i].add(cislo)
                    self.v3x3[i // 3][j // 3].add(cislo)
                    self.backtracking()
                    self.tab[i][j] = 0
                    self.vstlpci[j].remove(cislo)
                    self.vriadku[i].remove(cislo)
                    self.v3x3[i // 3][j // 3].remove(cislo)

    def ries(self):
        self.pocet = 0
        self.backtracking()
        print('pocet rieseni:', self.pocet)

Sudoku('sudoku.txt').ries()
```

34.4 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Napíšte funkciu `vypis_rastuce(n)`, ktorá vygeneruje a vypíše všetky `n`-tice čísel z `<0, n-1>`. Pre tieto vygenerované postupnosti musí platiť, že i-ty prvok nie je väčší ako `(i+1)`-ty.
   - funkcia by nemala používať žiadne globálne premenné (napr. `pole`)
   - napr.

```
>>> vypis_rastuce(2)
0 0
0 1
1 1
>>> vypis_rastuce(3)
0 0 0
0 0 1
0 0 2
0 1 1
0 1 2
0 2 2
1 1 1
1 1 2
1 2 2
2 2 2
```

- **pomôcka**: môžete využiť generovanie `n`-tíc z prednášky, pričom samotnú rekurzívnu funkciu vnoríte do funkcie `vypis_rastuce()`:

```
def vypis_rastuce(n):
    def generuj(i):
        ...
            generuj(i + 1)

    pole = [0] * n
    generuj(0)

vypis_rastuce(2)
vypis_rastuce(3)
```

2. Napíšte funkciu `vsetky(mnozina)`, ktorá pre danú množinu písmen (kde počet písmen v množine je `n`) vygeneruje a vypíše všetky `n`-znakové slová, ktoré obsahujú len písmená z danej množiny
   - napr. (výpis v ľubovoľnom poradí):

```
>>> vsetky({'x', 'y'})
xx
xy
yx
yy
```

   - **pomôcka**: opäť využijete vnorenú rekurzívnu funkciu, napr.:

```
def vsetky(mnozina):
    def generuj():
        ...
            generuj()

    slovo = []
    generuj()

vsetky({'x', 'y'})
vsetky(set('ahoj'))
```

3. Napíšte funkciu `vsetky_len_raz(mnozina)`, ktorá pre danú množinu písmen (kde počet písmen v množine je `n`) vygeneruje a vypíše všetky `n`-znakové slová, ktoré obsahujú všetky písmená z množiny (každé práve raz)

- napr.

```
>>> vsetky_len_raz({'a', 'k', 'm'})
akm
amk
kam
kma
mak
mka
```

- **pomôcka**: do riešenia z úlohy (2) pridajte test, ktorým zistíte, či sa pridávaný znak už nenachádza vo vytváranom slove:

```
def vsetky_len_raz(mnozina):
    ...
```

4. Úlohy (2) a (3) riešte tak, aby funkcie nič nevypisovali (pomocou `print()`), ale namiesto toho vrátili zoznam (alebo množinu) všetkých vygenerovaných slov

- napr.

```
>>> vsetky_zoznam({'x', 'y'})
['xx', 'xy', 'yx', 'yy']
>>> vsetky_len_raz_zoznam(set('akm'))
['akm', 'amk', 'kam', 'kma', 'mak', 'mka']
```

5. Spojazdnite riešenie úlohy **Dámy na šachovnici** z prednášky a upravte ho tak, aby sa vypísali len prvé tri nájdené riešenia aj s celkovým počtom všetkých riešení v tvare:

- napr.

```
>>> Damy(8).ries()
o . . . . . . .
. . . . o . . .
. . . . . . . o
. . . . . o . .
. . o . . . . .
. . . . . . o .
. o . . . . . .
. . . o . . . .
================
...
pocet rieseni: 92
```

6. Upravte program pre riešenie úlohy **Dámy na šachovnici**, ktorý `n`-dám rozmiestňoval na šachovnici `n` x `n` tak, aby našiel všetky riešenia pre `n`-dám na šachovnici `n` x `m`, kde `n` <= `m`

- napr.

```
>>> Damy(2, 3).ries()            # šachovnica vel'kosti 2x3
o . .
. . o
=====
. . o
o . .
=====
pocet rieseni: 2
```

- ```
  class Damy:
      def __init__(self, n, m):
          self.n, self.m = n, m
          ...

  Damy(2, 3).ries()
  Damy(3, 4).ries()
  ```

7. Riešte takýto problém:
   - `n` žiakov sa príhlásilo na `k` rôznych krúžkov (každý žiak si zvolil svoju množinu krúžkov)
   - keďže kapacita každého krúžku je obmedzená (hodnotou `limit`), vedenie školy rozhodlo, že každý žiak bude navštevovať len jeden z krúžkov, do ktorých sa prihlásil
   - priraďte každého žiaka do jedného z krúžkov tak, aby sa nepresiahol zadaný limit a pritom by každý zo žiakov chodil do jedného zo svojich zvolených krúžkov
   - napr. takto vyzerá textový súbor s menami žiakov a ich vybraných krúžkov:

```
ana kreslenie sach foto
boris futbal kreslenie
cyril futbal foto
dasa foto
eva sach kreslenie
fero futbal sach
gusto futbal kreslenie
hana kreslenie foto sach
ivan kreslenie futbal
jana foto sach
karol kreslenie foto
lucia sach futbal
```

   - vypíšte ku každému krúžku zoznam žiakov, napr. pre `limit = 3` môže byť takýto výsledok:

```
kreslenie: eva ivan karol
sach: jana lucia fero
futbal: cyril boris gusto
foto: hana dasa ana
```

   - **pomôcka**:
     - prečítajte súbor a informácie uchovajte v slovníkoch (asociatívnych poliach), v ktorých každý žiak má priradenú svoju množinu krúžkov, napr.

```
ziak['ana'] = {'kreslenie', 'sach', 'foto'}
ziak['boris'] = {'futbal', 'kreslenie'}
...
```

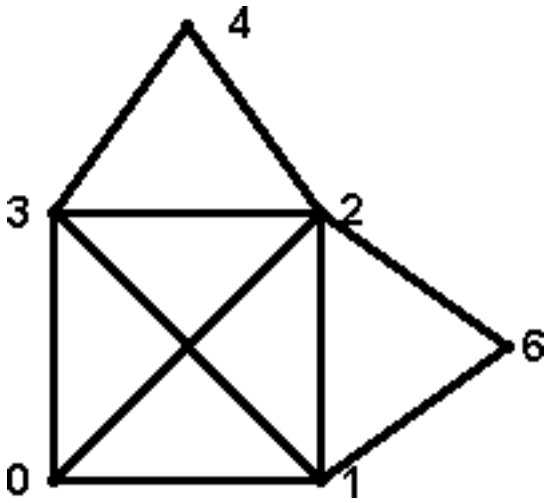
     - ďalej pripravte slovníky, kde pre každý krúžok priradíte prázdnu množinu (sem sa budú postupne pridávať žiaci, resp. sa budú odoberať), napr.

```
kruzok['kreslenie'] = set()
kruzok['sach'] = set()
...
```

     - rekurzívny backtracking spracuje jedno meno žiaka (postupne ho vkladá do príslušných množín krúžkov) a zakaždým sa rekurzívne zavolá s nasledovným menom žiaka
```

  - keď už sa spracovali všetci žiaci, vypíšu sa množiny krúžkov a prehľadávanie môže skončiť (stačí nám jedno riešenie, lebo ich môže byť veľmi veľa)

8. Spojazdnite riešenie úlohy **Domček jedným ťahom** z prednášky a upravte ho tak, aby fungovalo pre takto upravený domček:



9. **Ručne** zistite, čo vygeneruje tento backtracking

   - všimnite si, že `tab` je dvojrozmerným zoznamom (maticou) znakov:

```
def backtracking(r, s):
    if 0 <= r < len(tab) and 0 <= s < len(tab[r]):
        if tab[r][s] == 'x':
            print('\n'.join(' '.join(r) for r in tab), '\n')
        elif tab[r][s] == '.':
            tab[r][s] = '-'
            backtracking(r, s + 1)
            tab[r][s] = '|'
            backtracking(r + 1, s)
            tab[r][s] = '\\'                # '\\' oznacuje jeden znak \
            backtracking(r + 1, s + 1)
            tab[r][s] = '.'

tab = [list(r) for r in ('...', '..x')]
backtracking(0, 0)
```

10. Zmenou zoznamu v predchádzajúcom príklade sa mení aj výpis vygenerovaných riešení

    - zistite, **koľko rôznych riešení** sa vygeneruje pre takto definovaný dvojrozmerný zoznam `tab`:

```
tab = [list(r) for r in '... .m. ..x'.split()]
```

    alebo:

```
tab = list(map(list, '...m ..m. .m.. m..x'.split()))
```

11. Upravte funkciu `backtracking()` z úlohy (9) tak, aby nevypisovala všetky riešenia, len vrátila počet všetkých riešení

    - funkciu zapúzdrite do nejakej triedy tak, aby fungovalo, napr.

```
>>> Uloha('... ..x').pocet()
5
>>> Uloha('... .m. ..x').pocet()
4
>>> Uloha('...m ..m. .m.. m..x'.).pocet()
11
```

34.5 10. Domáce zadanie

L.I.S.T.

- riešenie odovzdávajte na úlohový server https://list.fmph.uniba.sk/

34.5.1 Zadanie skúšky 3.6.2015: Písmenkový graf

Máme daný neorientovaný graf, v ktorom sa na niektorých hranách môže nachádzať nejaké písmeno z množiny `{'a', 'b', 'c'}`. Ostatné hrany sú tzv. neoznačené. Vašou úlohou bude zostaviť čo najdlhšiu cestu, ktorá vychádza z nejakého zadaného štartového vrcholu a pre ktorú platí:

- prvá hrana v ceste má označenie `'a'` alebo je bez označenia
- druhá hrana v ceste má označenie `'b'` alebo je bez označenia
- atď. na hranách cesty sa stále striedajú písmená `'a'`, `'b'`, `'c'`, `'a'`, `'b'`, `'c'`, ..., resp. niektoré hrany v ceste môžu byť neoznačené (fungujú niečo ako žolík)
- cesta môže prechádzať aj viackrát cez tie isté vrcholy ale po hranách môže prejsť len raz

Vašou úlohou je prečítať textový súbor s popisom grafu a po zadaní štartového vrcholu vygenerovať všetky maximálne dlhé cesty, ktoré spĺňajú pravidlo striedania písmen.

Riešenie zapíšte do triedy `Graf` s týmito metódami:

```
class Graf:
    def __init__(self, meno_suboru):
        ...

    def hrana(self, v1, v2):
        return None

    def vrcholy(self):
        return set()

    def ries(self, v1):
        return []
```

Kde metódy:

- `__init__(meno_suboru)`: prečíta súbor a vytvorí z neho nejakú reprezentáciu neorientovaného ohodnoteného grafu
- `hrana(v1, v2)`: vráti hodnotu hrany: buď je to jedno z písmen `{'a', 'b', 'c'}`, alebo prázdny reťazec `''` pre neoznačenú hranu, alebo `None`, ak neexistuje hrana medzi týmito dvoma vrcholmi
- `vrcholy()`: vráti množinu mien všetkých vrcholov

- `ries(v1)`: pomocou backtrackingu nájde všetky najdlhšie cesty z vrcholu `v1` – metóda vráti buď prázdny zoznam (nenašiel žiadne riešenie, ktoré by obsahovalo aspoň jednu hranu), alebo zoznam (typ `list`), ktoré obsahuje **všetky najdlhšie cesty**; cesta je postupnosť (`list`) mien vrcholov

Môžete si dodefinovať aj ďalšie pomocné metódy.

Textový súbor obsahuje popis grafu v tvare:

- každý riadok obsahuje popis jednej hrany grafu buď ako dvojicu alebo ako trojicu
- dvojica v tvare `meno1:meno2` označuje hranu bez písmena (žolík)
- trojica v tvare `meno1:písmeno:meno2` označuje hranu so zadaným písmenom
- mená vrcholov sú nejaké znakové reťazce zložené len z písmen
- uvedomte si, že v neorientovanom grafe by mala byť každá hrana orientovaná oboma smermi
- môžete predpokladať, že súbor je zadaný korektne

Napr. súbor

```
mo:b:ix
ub:c:mo
ix:ub
er:c:ix
qa:a:ix
er:qa
ub:er
qa:c:ub
```

označuje graf s piatimi vrcholmi a 8 hranami, pričom niektoré majú označenie a niektoré nie.

Program môžete testovať napríklad takto:

```
if __name__ == '__main__':
    g = Graf('subor1.txt')
    print('vrcholy =', g.vrcholy())
    for v1, v2 in ('mo', 'ub'), ('er', 'ub'), ('er', 'mo'):
        print(f'hrana({v1!r},{v2!r}) = {g.hrana(v1,v2)!r}')
    riesenie = g.ries('qa')
    print('pocet rieseni =', len(riesenie))
    print(*riesenie, sep='\n')
```

a pre vyššie uvedený textový súbor by mal vypísať:

```
vrcholy = {'er', 'ix', 'ub', 'qa', 'mo'}
hrana('mo','ub') = 'c'
hrana('er','ub') = ''
hrana('er','mo') = None
pocet rieseni = 2
['qa', 'er', 'ub', 'qa', 'ix', 'mo', 'ub', 'ix']
['qa', 'ix', 'mo', 'ub', 'er', 'qa', 'ub', 'ix']
```

Za vyriešenie skúškovej úlohy môžete získať 10 bodov, pričom:

- 20% je za vytvorenie grafu zo súboru, t.j. správne fungovanie metód `vrcholy()` a `hrana()`
- 20% je za nájdenie nejakého aspoň jedného správneho riešenia, nemusí byť maximálnej dĺžky, ale malo by byť dostatočne dlhé
- 30% za nájdenie aspoň jedného maximálneho riešenia

- 30% za nájdenie všetkých maximálnych riešení

34.5.2 Obmedzenia

- vaše riešenie odovzdajte v súbore `riesenie10.py`, pričom sa v ňom bude nachádzať **len jedna definícia triedy** `Graf`.
- prvé dva riadky tohto súboru budú obsahovať:

```
# autor: Janko Hrasko
# uloha: 10. domace zadanie pismenkovy graf
```

- zrejme ako autora uvediete svoje meno
- váš program by nemal počas testovania testovačom nič vypisovať (žiadne vaše testovacie `print()`)
- testovač bude spúšťať vašu definíciu triedy s rôznymi textovými súbormi, ktoré si môžete stiahnuť z L.I.S.T.u

KAPITOLA 35

35. Backtracking na grafoch

Využijeme mierne upravenú triedu `Graf` z prednášky *33. Algoritmy prehľadávania grafu*:

```
import tkinter
import random

class Graf:

    class Vrchol:
        canvas = None

        def __init__(self, x, y):
            self.sus = {}
            self.xy = x, y

        def pridaj_hranu(self, v2, vaha):
            self.sus[v2] = vaha

        def kresli(self, farba='gray'):
            x, y = self.xy
            self.id = self.canvas.create_oval(x-15, y-15, x+15, y+15, fill=farba,␣
↪outline='')

        def zafarbi(self, farba):
            self.canvas.itemconfig(self.id, fill=farba)

        #-----------------------

    def __init__(self, n):
        self.vrcholy = []
        self.canvas = self.Vrchol.canvas = tkinter.Canvas(bg='white', width=600,␣
↪height=600)
        self.canvas.pack()
        d = (min(600, 600)-60) // (n-1)
        for i in range(n):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            for j in range(n):
                x1, y1 = d*j+random.randrange(20, 40), d*i+random.randrange(20, 40)
                v1 = self.pridaj_vrchol(x1, y1)
                if j > 0 and random.randrange(3):
                    x2, y2 = self.vrcholy[v1-1].xy
                    vaha = round(((x1-x2)**2+(y1-y2)**2)**.5)
                    self.pridaj_hranu(v1, v1-1, vaha)
                if i > 0 and random.randrange(3):
                    x2, y2 = self.vrcholy[v1-n].xy
                    vaha = round(((x1-x2)**2+(y1-y2)**2)**.5)
                    self.pridaj_hranu(v1, v1-n, vaha)
        self.id = {}
        for i in range(len(self.vrcholy)):
            for j in self.vrcholy[i].sus:
                if i < j:
                    self.kresli_hranu(i, j)
        for vrchol in self.vrcholy:
            vrchol.kresli()
        self.canvas.bind('<Button-1>', self.udalost_klik)
        self.klik = []

    def pridaj_vrchol(self, x, y):
        self.vrcholy.append(self.Vrchol(x, y))
        return len(self.vrcholy) - 1

    def pridaj_hranu(self, i, j, vaha):
        self.vrcholy[i].pridaj_hranu(j, vaha)
        self.vrcholy[j].pridaj_hranu(i, vaha)

    def je_hrana(self, i, j):
        return j in self.vrcholy[i].sus

    def kresli_hranu(self, i, j, farba='gray'):
        self.id[i, j] = self.id[j, i] = \
            self.canvas.create_line(self.vrcholy[i].xy, self.vrcholy[j].xy,
                                    fill=farba, width=3)

    def zafarbi_hranu(self, i, j, farba):
        self.canvas.itemconfig(self.id[i, j], fill=farba)

    def udalost_klik(self, event):
        for i in range(len(self.vrcholy)):
            x, y = self.vrcholy[i].xy
            if (x-event.x)**2 + (y-event.y)**2 < 15**2:
                self.vrcholy[i].zafarbi('blue')
                self.klik.append(i)
                return

graf = Graf(7)
```

Upravili sme:

- graf je ohodnotený, t.j. pre každú hranu si uchovávame jej váhu (zvolili sme vzdialenosť vrcholov v rovine)
  - namiesto zoznamu množín susedností graf reprezentujeme ako zoznam vrcholov, v ktorom si každý vrchol pamätá susedov v asociatívnom poli (`dict`) s hodnotami váha hrany
- generovanie pozícií vrcholov: náhodne sme ich trochu poposúvali

- pridali sme metódu `udalost_klik()`, ktorá nielen zafarbuje kliknuté vrcholy, ale si ich aj pamätá v zozname `self.klik`

Základná schéma backtrackingu

Pripomeňme si, na čo vieme použiť prehľadávanie do hĺbky a do šírky:

- algoritmus **do hlbky** využijeme hlavne na zistenie príslušnosti vrcholov do komponentov
- algoritmus **do sirky** slúži na nájdenie vzdialenosti (najkratšej cesty) dvoch vrcholov
  - takáto dĺžka cesty neberie do úvahy váhu na hranách, ale len počet vrcholov na ceste

Prehľadávanie s návratom

- pracuje na princípe prechádzania (generovania) všetkých možných ciest v grafe, ktoré vychádzajú z nejakého vrcholu
- pri tomto generovaní, môžeme zisťovať rôzne parametre týchto vygenerovaných ciest, napr. súčet váh na hranách cesty, alebo, či cesta prešla cez nejaký konkrétny vrchol a pod.
- uvedomte si, že takéto prehľadávanie (spomínali sme to ako *hrubá sila*, *brute force*) bude pre väčšie grafy časovo veľmi náročné, keďže vygenerovaných ciest môže byť obrovské množstvo
- často závisí od detailov v nejakom teste, ako rýchlo takéto prehľadávanie nájde riešenie (hovoríme tomu, že využijeme nejakú **heuristiku**)
- vo vyšších ročníkoch sa zoznámite s algoritmami, ktoré vedia riešiť niektoré z našich úloh aj bez *hrubej sily*

Základnú schému backtrackingu, môžeme pre graf upraviť napr. takto:

```
class Graf:
    ...

    def start(self, v1, v2):
        self.visited = set()
        self.backtracking(v1, v2)

    def backtracking(self, v1, v2):
        self.visited.add(v1)
        if v1 == v2:
            ... # spracuj riesenie
        else:
            for v in self.vrcholy[v1].sus:
                if v not in self.visited:
                    # vizualizuj prechadzanu hranu
                    self.backtracking(v, v2)
                    # zrus vizualizovanie prechadzanej hrany
        self.visited.remove(v1)
```

- podobne ako pri prechádzaní grafu do hĺbky, aj tu musíme mať nejakú štruktúru `visited`, aby sme neprechádzali cez nejaký vrchol viackrát
- parameter `v1` označuje práve prechádzaný vrchol, `v2` cieľový vrchol, do ktorého sa snažíme dostať
- tento algoritmus naozaj nájde všetky možné cesty z vrcholu `v1` do `v2`, pričom pri spracovaní riešenia môžeme tieto nájdené cesty nejako pretriediť a vybrať si len tú najvhodnejšiu

Vytvorme takýto program:

- po kliknutí na dva rôzne vrcholy sa naštartuje backtracking

- samotný algoritmus bude každú prechádzanú hranu prefarbovať a na malú chvíľu pritom spomalí výpočet
- konkrétne nasledovný program bude počítať počet rôznych ciest a tento počet na záver vypíše

Upravíme aj `udalost_klik()`:

```
class Graf:
    ...

    def udalost_klik(self, event):
        for i in range(len(self.vrcholy)):
            x, y = self.vrcholy[i].xy
            if (x-event.x)**2 + (y-event.y)**2 < 15**2:
                self.vrcholy[i].zafarbi('blue')
                self.klik.append(i)
                self.canvas.update()
                if len(self.klik) == 2:           # <=== po druhom kliknuti
                    self.start(*self.klik)
                return

    def start(self, v1, v2):
        self.visited = set()
        self.pocet = 0
        self.backtracking(v1, v2)
        # ked skonci backtracking
        print('pocet rieseni =', self.pocet)

    def backtracking(self, v1, v2):
        if v1 == v2:
            self.pocet += 1      # spracuj riesenie
            # print('nasiel som cestu')
        else:
            self.visited.add(v1)
            for v in self.vrcholy[v1].sus:
                if v not in self.visited:
                    # vizualizuj prechadzanu hranu
                    self.zafarbi_hranu(v1, v, 'red')
                    self.canvas.update()
                    self.canvas.after(100)
                    self.backtracking(v, v2)
                    # zrus vizualizovanie prechadzanej hrany
                    self.zafarbi_hranu(v1, v, 'gray')
            self.visited.remove(v1)

graf = Graf(7)
```

- volanie `self.canvas.after(100)` označuje, že tu program zastane na 100 ms, tento riadok môžeme vyhodiť, aby bolo hľadanie ciest rýchlejšie

V niektorých situáciách sa nám môže hodiť, keď po nájdení prvého riešenia, zastavíme beh backtrackingu ale tak, aby táto cesta ostala nakreslená:

```
class Graf:
    ...

    def backtracking(self, v1, v2):
        if self.pocet > 0:           # aby sa dalej nevnaral
            return
        if v1 == v2:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            self.pocet += 1          # spracuj riesenie
            print('nasiel som cestu')
        else:
            self.visited.add(v1)
            for v in self.vrcholy[v1].sus:
                if v not in self.visited:
                    # vizualizuj prechadzanu hranu
                    self.zafarbi_hranu(v1, v, 'red')
                    self.canvas.update()
                    self.canvas.after(100)
                    self.backtracking(v, v2)
                    if self.pocet > 0:               # aby sa nezrusilo zafarbenie␣
↪hran
                        return
                    # zrus vizualizovanie prechadzanej hrany
                    self.zafarbi_hranu(v1, v, 'gray')
            self.visited.remove(v1)
```

Častejšie ale budeme prechádzať všetky vygenerované cesty a hľadať medzi nimi jednu s nejakou vlastnosťou, napr. cestu s najmenším ohodnotením: ak je na hranách napr. cena, tak hľadáme najlacnejšiu cestu. Zrejme je rozdiel medzi najkratšou cestou (s najmenej prechádzanými vrcholmi) a najlacnejšou cestou.

35.1 Hodnota cesty

Pri generovaní cesty budeme evidovať aj jej hodnotu (ako súčet ohodnotení hrán). Najprv to ukážeme pomocou atribútu `hodnota` v triede `Graf`, ktorý sa pri prechode hranou zvyšuje a pri návrate znižuje:

```
class Graf:
    ...

    def start(self, v1, v2):
        self.visited = set()
        self.pocet = 0
        self.hodnota = 0               # hodnota celej cesty
        self.backtracking(v1, v2)
        print('pocet =', self.pocet)

    def backtracking(self, v1, v2):
        if v1 == v2:
            self.pocet += 1
            print('cesta s hodnotou', self.hodnota)
        else:
            self.visited.add(v1)
            for v in self.vrcholy[v1].sus:
                if v not in self.visited:
                    self.hodnota += self.vrcholy[v1].sus[v]
                    self.zafarbi_hranu(v1, v, 'red')
                    self.canvas.update()
                    #self.canvas.after(100)

                    self.backtracking(v, v2)

                    self.hodnota -= self.vrcholy[v1].sus[v]
                    self.zafarbi_hranu(v1, v, 'gray')
            self.visited.remove(v1)
```

Hodnotu cesty môžeme vytvárať aj v parametri funkcie `backtracking()`. Na začiatku je hodnota cesty 0, vždy keď algoritmus prejde po nejakej hrane, tak sa táto hodnota zvýši. Keď cesta dorazí do cieľa (do vrcholu `v2`), zistí sa, či táto nová vygenerovaná cesta je lepšia (napr. menšia ako doterajšie minimum, alebo väčšia ako doterajšie maximum) ako doteraz uchovávané hodnoty v premenných `self.min` a `self.max`. Všimnite si, že takto uchovávanú hodnotu v parametri nemusíme pri návrate z rekurzie znižovať, ako sme to robili v predchádzajúcej verzii:

```
class Graf:
    ...

    def start(self, v1, v2):
        self.visited = set()
        self.pocet = 0
        self.min = self.max = None
        self.backtracking(v1, v2, 0)
        print('pocet =', self.pocet)
        print('najkratsia =', self.min)
        print('najdlhsia  =', self.max)

    def backtracking(self, v1, v2, hodnota):
        if v1 == v2:
            self.pocet += 1
            # print('cesta s hodnotou', hodnota)
            if self.min is None or self.min > hodnota:
                self.min = hodnota
            if self.max is None or self.max < hodnota:
                self.max = hodnota
        else:
            self.visited.add(v1)
            for v in self.vrcholy[v1].sus:
                if v not in self.visited:
                    self.zafarbi_hranu(v1, v, 'red')
                    self.canvas.update()
                    #self.canvas.after(100)

                    self.backtracking(v, v2, hodnota + self.vrcholy[v1].sus[v])

                    self.zafarbi_hranu(v1, v, 'gray')
            self.visited.remove(v1)
```

35.2 Zapamätanie celej cesty

Doterajšie verzie algoritmu generovali všetky cesty, ale si ich nikde neuchovávali. Využijeme pomocný zoznam (premenná `self.cesta`), do ktorého budeme ukladať momentálnu cestu. Tiež si ju uložíme do premennej `self.najcesta`, keď vyhovuje nejakej podmienke. Po skončení algoritmu `backtracking` túto cestu vypíšeme:

```
class Graf:
    ...

    def start(self, v1, v2):
        self.visited = set()
        self.pocet = 0
        self.min = self.max = None
        self.cesta, self.najcesta = [], []
        self.backtracking(v1, v2, 0)
        print('pocet =', self.pocet)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        print('min =', self.min)
        print('max =', self.max)
        print('najkratsia cesta =', self.najcesta)

    def backtracking(self, v1, v2, hodnota):
        self.cesta.append(v1)
        if v1 == v2:
            self.pocet += 1
            #print('nasiel som cestu s hodnotou', hodnota)
            if self.min is None or self.min > hodnota:
                self.min = hodnota
                self.najcesta = self.cesta[:]              # zapamatame kopiu cesty
            if self.max is None or self.max < hodnota:
                self.max = hodnota
        else:
            self.visited.add(v1)
            for v in self.vrcholy[v1].sus:
                if v not in self.visited:
                    #self.zafarbi_hranu(v1, v, 'red')
                    #self.canvas.update()
                    #self.canvas.after(100)

                    self.backtracking(v, v2, hodnota + self.vrcholy[v1].sus[v])

                    #self.zafarbi_hranu(v1, v, 'gray')
            self.visited.remove(v1)
        self.cesta.pop()
```

- pri uchovávaní si najlepšej cesty musíme urobiť kópiu z premennej `self.cesta`, lebo nestačí urobiť len obyčajné priradenie, ktoré by priradilo len referenciu na premennú
- aby sme algoritmus urýchlili, odstránili sme aj vykresľovanie hrán

Tento algoritmus nájde najlepšiu cestu, ale ju nevykreslí, len vypíše čísla vrcholov. Ďalšia verzia programu vykresľuje najlepšiu cestu - teraz hľadáme cestu nie s najmenšou hodnotou, ale s najväčšou:

```
class Graf:
    ...

    def start(self, v1, v2):
        self.visited = set()
        self.min = self.max = None
        self.cesta, self.najcesta = [], []
        self.backtracking(v1, v2, 0)
        print('min =', self.min)
        print('max =', self.max)
        for i in range(len(self.najcesta)-1):
            self.zafarbi_hranu(self.najcesta[i], self.najcesta[i+1], 'blue')

    def backtracking(self, v1, v2, hodnota):
        self.cesta.append(v1)
        if v1 == v2:
            if self.min is None or self.min > hodnota:
                self.min = hodnota
            if self.max is None or self.max < hodnota:
                self.max = hodnota
                self.najcesta = self.cesta[:]
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        else:
            self.visited.add(v1)
            for v in self.vrcholy[v1].sus:
                if v not in self.visited:
                    self.backtracking(v, v2, hodnota + self.vrcholy[v1].sus[v])
            self.visited.remove(v1)
        self.cesta.pop()
```

- možno vám napadlo, že keď si pamätáme cestu, nepotrebujeme udržiavať množinu `self.visited`, veď `self.cesta` obsahuje tie isté vrcholy - je to pravda, len hľadanie vrcholu v množine (typ `set`) je pre veľa prvkov rýchlejšie ako v zozname (typ `list`) a takýto program sa môže pre niekoho zdať menej čitateľný
- podobne ako sme namiesto atribútu `self.hodnota` pridali nový parameter `hodnota` do metódy `backtracking()`, môžeme pridať aj parameter `cesta`, v ktorom sa vytvára momentálna postupnosť prechádzaných vrcholov

Hľadanie najkratšej cesty môžeme teraz zapísať:

```
class Graf:
    ...

    def start(self, v1, v2):
        self.naj = None
        self.najcesta = []
        self.backtracking(v1, v2, 0, [v1])        # vo vytvaranej ceste je uz prvy␣
↪vrchol
        print('naj =', self.naj)
        for i in range(len(self.najcesta)-1):
            self.zafarbi_hranu(self.najcesta[i], self.najcesta[i+1], 'blue')

    def backtracking(self, v1, v2, hodnota, cesta):
        if v1 == v2:
            if self.naj is None or self.naj > hodnota:
                self.naj = hodnota
                self.najcesta = cesta
        else:
            for v in self.vrcholy[v1].sus:
                if v not in cesta:                # kontrolujeme cestu namiesto␣
↪visited
                    self.backtracking(v, v2, hodnota + self.vrcholy[v1].sus[v], cesta␣
↪+ [v])
```

35.3 Hľadanie cyklov v grafe

Backtracking môžeme využiť aj na generovanie všetkých cyklov, t.j. takých ciest, v ktorých je posledný vrchol ten istý ako prvý. Využijeme poslednú verziu backtrackingu, v ktorej sme pridali parameter `cesta`. Pritom zmien v programe pre hľadanie cyklu (hľadáme cyklus s maximálnou hodnotou) je veľmi málo:

```
class Graf:
    ...

    def udalost_klik(self, event):
        for i in range(len(self.vrcholy)):
            x, y = self.vrcholy[i].xy
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            if (x-event.x)**2 + (y-event.y)**2 < 15**2:
                self.vrcholy[i].zafarbi('blue')
                self.start(i)
                return

    def start(self, v1):                        # metoda ma iba jeden parameter
        self.naj = None
        self.najcesta = []
        self.backtracking(v1, v1, 0, [])       # backtracking startujeme s prazdnou␣
↪cestou
        print('naj =', self.naj)
        for i in range(len(self.najcesta)-1):
            self.zafarbi_hranu(self.najcesta[i], self.najcesta[i+1], 'blue')

    def backtracking(self, v1, v2, hodnota, cesta):
        if cesta != [] and v1 == v2:
            if len(cesta) > 2 and (self.naj is None or self.naj < hodnota):    #␣
↪hlahame maximum
                self.naj = hodnota
                self.najcesta = [v2] + cesta     # k najdenemu rieseniu pridame na␣
↪zaciatok startovy vrchol
        else:
            for v in self.vrcholy[v1].sus:
                if v not in cesta:
                    self.backtracking(v, v2, hodnota + self.vrcholy[v1].sus[v], cesta␣
↪+ [v])
```

Zmenili sme:

- zjednodušili sme metódu `udalost_klik()`, lebo nepotrebujeme ukladať viac kliknutých vrcholov a metódu `start()` voláme len s jedným parametrom: začiatkom cyklu, čo je zároveň aj koncom cyklu
- v metóde `start()` sme zmenili naštartovanie backtrackingu: posielame ako začiatok aj koniec cesty ten istý vrchol a tiež prvý vrchol nevkladáme do vytváranej cesty (parameter `cesta`), aby sme ho tam mohli vložiť aj druhýkrát
- úvodný test v metóde `backtracking()` kontroluje nielen to, či sme už v cieli (teda `v1 == v2`), ale aj dĺžku vytvoreného cyklu, ktorá by mala mať aspoň 2 rôzne hrany, t.j. aspoň 3 vrcholy
- v metóde `backtracking()`, keď nájdeme nejaké vyhovujúce riešenie, ktoré je lepšie ako doteraz nájdené najlepšie (v atribúte `self.najcesta`), pridáme k nemu na začiatok štartový vrchol, lebo ten sme úmyselne do cesty nezaradili

Malým zjednodušením tohto riešenie vieme hľadať najdlhší (alebo aj najkratší) cyklus na počet prejdených vrcholov a to bez ohľadu na hodnoty na hranách:

```
class Graf:
    ...

    def start(self, v1):
        self.najcesta = []
        self.backtracking(v1, v1, [])
        if self.najcesta == []:
            print('nenasiel ziadnu cestu')
        for i in range(len(self.najcesta)-1):
            self.zafarbi_hranu(self.najcesta[i], self.najcesta[i+1], 'blue')

    def backtracking(self, v1, v2, cesta):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        if cesta != [] and v1 == v2:
           if len(cesta) > 2 and len(self.najcesta) < len(cesta):
                self.najcesta = [v2] + cesta
        else:
            for v in self.vrcholy[v1].sus:
                if v not in cesta:
                    self.backtracking(v, v2, cesta + [v])
```

V prípade, že takýto cyklus prejde cez všetky vrcholy, hovoríme mu **Hamiltonovská kružnica** (v skutočnosti je Hamiltonovská kružnica podgrafom, ktorý obsahuje všetky vrcholy grafu a len hrany, ktoré sú z tohto cyklu).

35.4 Cvičenia

L.I.S.T.

- riešenia odovzdávajte na úlohový server https://list.fmph.uniba.sk/

1. Spojazdnite triedu `Graf` z prednášky - verziu, ktorá nájde a vyznačí najkratšiu cestu medzi dvoma kliknutými vrcholmi.
   - metóda `backtracking()` má parametre `hodnota` a `cesta`:

```
class Graf:
    ...

    def backtracking(self, v1, v2, hodnota, cesta):
        ...
```

2. Do triedy `Graf` dopíšte generovanie náhodných hodnôt na hranách grafu z intervalu `<1, 4>`, túto hodnotu vypíšte do stredu príslušnej hrany
   - upravte tieto metódy:

```
class Graf:
    ...

    def __init__(self, n):
        ...

    def kresli_hranu(self, i, j, farba='gray'):
        ...
```

   - otestujte, ako teraz funguje hľadanie najkratšej cesty
3. Opravte metódu `udalost_klik()` tak, aby po nájdení a vykreslení cesty ďalšie kliknutie odfarbilo dovtedy vybraté dva vrcholy aj vyznačenú cestu a znovu začalo výber štartového a cieľového vrcholu.
   - dopíšte do metódy odfarbovanie:

```
class Graf:
    ...

    def udalost_klik(self, event):
        ...
```

- otestujte opakované hľadanie minimálnej cesty

4. Do náhodného generovania hrán grafu v inicializácii `__init__()` pridajte generovanie aj šikmých hrán
   - s nejakou pravdepodobnosťou spojí momentálny vrchol `v1` s vrcholom, ktorý je o riadok aj o stĺpec o 1 menší (zrejme to nerobí v prvom riadku ani stĺpci):

```
class Graf:
    ...

    def __init__(self, n):
        ...
```

   - otestujte, ako teraz funguje hľadanie najkratšej cesty

5. Upravte backtracking tak, aby hľadal a vyznačil nie minimálnu, ale maximálnu cestu
   - upravte:

```
class Graf:
    ...

    def backtracking(self, v1, v2, hodnota, cesta):
        ...
```

6. Upravte backtracking tak, aby hľadal a vyznačil maximálne dlhú cestu (na počet prejdených vrcholov), ale ak je takých viac vyberie tú z nich, ktorá má minimálny súčet váh
   - upravte:

```
class Graf:
    ...

    def backtracking(self, v1, v2, hodnota, cesta):
        ...
```

7. Zmeňte metódu `backtracking()` tak, aby po druhom kliknutí na nejaký vrchol našla najkratší (resp. najdlhší) **cyklus**, ktorý začína (a teda aj končí) vo vrchole `v1` a prechádza cez druhý kliknutý vrchol
   - upravte:

```
class Graf:
    ...

    def start(self, v1, v2):
        ...
        self.backtracking(v1, v1, v2, 0, [])
        ...

    def backtracking(self, v1, v2, v3, hodnota, cesta):   # v3 je vrchol, cez␣
↪ktory prechadza cyklus
        ...
```

   - backtracking generuje všetky cykly z `v1` (do `v2`) a pre každý ešte skontroluje, či prechádza cez `v3`

8. Napíšte metódu `zisti5()`, ktorá pomocou backtrackingu zistí, koľko existuje v danom grafe rôznych cyklov dĺžky presne 5 (zaujíma nás, že počet vrcholov v ceste je 5 a nie aké sú váhy na hranách)
   - zrejme v pôvodnom grafe, v ktorom boli hrany náhodne generované len vodorovne a zvislo, nemôže existovať žiaden cyklus dĺžky 5

- metóda na záver vypíše tento zistený počet:

```
class Graf:
    ...
    def zisti5(self):
        ...

graf = Graf(7)
graf.zisti5()
```

9. Pre danú konštantu `D` a kliknutím na jeden vrchol zafarbite (napr. žltou farbou) **všetky** také vrcholy, pre ktoré existuje cesta štartujúca zo zadaného (kliknutého) vrcholu a jej hodnota (súčet váh na hranách) je presne `D`

- upravte metódy:

```
D = 10

class Graf:
    ...

    def udalost_klik(self, event):
        ...

    def start(self, v1):
        ...

    def backtracking(self, ...):
        ...
        self.vrcholy[v1].zafarbi('yellow')
        self.canvas.update()
        ...
```

10. Zmeňte metódu `backtracking()` tak, aby hľadaná najdlhšia cesta mohla prechádzať cez tie isté vrcholy aj viackrát, len treba strážiť, aby nešla viackrát po tej istej hrane

- upravte metódy:

```
class Graf:
    ...

    def start(self, v1, v2):
        ...

    def backtracking(self, v1, v2, ...):
        ...
```

11. Ohodnotený neorientovaný graf je definovaný takto:

- v každom riadku je popísaná jedna hrana pomocou dvoch čísel vrcholov a váhy (reťazec), napr.:

```
1 2 a
1 4 ma
1 5 a
3 4 m
4 2 am
6 4 a
5 6 u
5 3 am
4 5 em
```

- zistite, koľko existuje rôznych ciest (cez vrcholy aj hrany môžeme prechádzať ľubovoľný počet krát), ktorých hodnota je reťazec `'mamamaemuaemamamamu'`
- úlohu riešte prečítaním grafu zo súboru a potom pomocou backtrackingu
- program vypíše všetky cesty ako postupnosti navštívených vrcholov:

```
class Graf:
    def __init__(self, meno_suboru):
        ...

    def start(self, retazec):
        ...

    def backtracking(self, ...):
        ...

graf = Graf('subor.txt')
graf.start('mamamaemuaemamamamu')
```

35.5 11. Domáce zadanie

L.I.S.T.

- riešenie odovzdávajte na úlohový server https://list.fmph.uniba.sk/

35.5.1 Zadanie skúšky 3.6.2016: Labyrint

Labyrint je štvorcová sieť veľkosti `m` x `n` a dozvieme sa, medzi ktorými políčkami siete sú steny, teda sa tadiaľ nebude dať prechádzať. Na niektorých políčkach sa nachádzajú odmeny (napr. mince). Úlohou je nájsť ľubovoľnú (stačí jednu) trasu zo štartového políčka, ktorá pozbiera všetky odmeny. Treba dať pozor, aby sa na žiadne políčko nestúpilo 2-krát.

Labyrint reprezentujte ako graf, v ktorom sú políčka labyrintu vrcholy grafu a hrany sú priechody medzi políčkami. Uvedomte si, že každý vrchol môže mať maximálne štyroch susedov (nemusí mať žiadneho).

Súbor má tvar:

- prvý riadok obsahuje dve čísla: počet riadkov a počet stĺpcov labyrintu
- ďalej nasledujú riadky so stenami a odmenami (v ľubovoľnom poradí)
- ak riadok obsahuje iba 2 čísla, označuje to políčko s odmenou (riadok, stĺpec), riadky aj stĺpce číslujeme o 0
- inak riadok obsahuje postupnosť aspoň dvoch políčok (riadok1, stĺpec1, riadok2, stĺpec2, ...) - touto postupnosťou definuje nejaké steny v labyrinte: stena je medzi prvým a druhým políčkom, aj druhým a tretím, ... atď.

Zadefinujte triedu Labyrint:

```
class Labyrint:
    class Vrchol:
        def __init__(self, riadok, stlpec):
            self.sus = []       # zoznam susedov, susedia su typu Vrchol
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            self.poloha = riadok, stlpec
            self.odmena = False

        def __repr__(self):
            return '<{},{}>'.format(*self.poloha)

    def __init__(self, meno_suboru):
        self.g = ...            # zoznam vrcholov grafu - obsahuje objekty typu Vrchol
        ...

    def daj_vrchol(self, riadok, stlpec):
        return ...

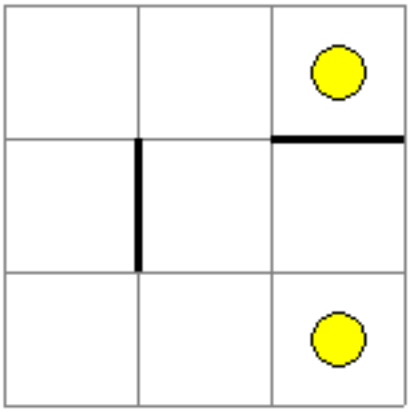
    def zmen_odmeny(self, *post):
        ...

    def start(self, riadok, stlpec):
        return ...
```

kde

- vnorená definícia triedy `Vrchol` má už zadefinované nejaké metódy aj atribúty – tieto atribúty využite pre reprezentáciu grafu; atribút `sus` bude obyčajný zoznam (typu `list`), ktorý bude obsahovať zoznam susediacich vrcholov, teda prvkami tohto zoznamu budú opäť objekty typu `Vrchol`;
- metóda `__init__(meno_suboru)`: prečíta súbor a vytvorí z neho graf: atribút `g` bude obsahovať všetky vrcholy grafu, teda objekty typu `Vrchol`; pre atribút `g` si môžete zvoliť buď obyčajný zoznam vrcholov (`list`), alebo dvojrozmerný zoznam vrcholov (`list of list`, teda dvojrozmernú tabuľku) alebo asociatívne pole (`dict`), v ktorom kľúčom je poloha políčka (riadok, stĺpec) a príslušnými hodnotami sú samotné vrcholy;
- metóda `daj_vrchol(riadok, stlpec)`: vráti referenciu (typ `Vrchol`) na príslušný vrchol grafu, ktorý reprezentuje zadané políčko;
- metóda `zmen_odmeny(*post)`: parameter `post` je postupnosť (`list` alebo `tuple`) dvojíc (riadok, stĺpec); metóda zmení na všetkých zadaných políčkach to, či sa na nich nachádza odmena: ak na políčku bola odmena, odstráni ju, inak na políčko umiestni odmenu;
- metóda `start(riadok, stlpec)`: pomocou backtrackingu hľadá správnu trasu; metóda vráti postupnosť navštívených políčok (zrejme prvé políčko v tejto postupnosti bude zadaný (riadok, stĺpec)); ak trasa, ktorá prejde cez všetky políčka s odmenou sa skonštruovať nedá, metóda vráti prázdny zoznam `[]`; testovač môže zavolať túto metódu viac krát s rôznymi štartovými políčkami.

Napr. pre takéto zadanie labyrintu:



```
3 3
1 0 1 1
2 2
0 2
1 2 0 2
```

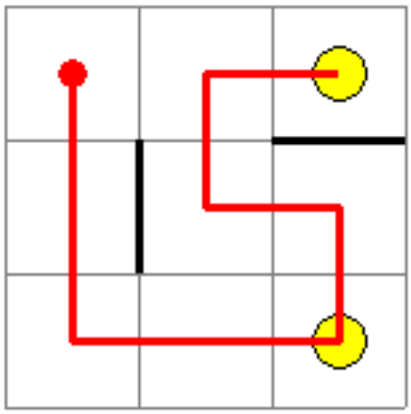
tento test:

```
if __name__ == '__main__':
    lab = Labyrint('subor1.txt')
    v = lab.daj_vrchol(1, 0)
    print('vrchol:', v, 'susedia:', v.sus, 'odmena:', v.odmena)
    v = lab.daj_vrchol(0, 2)
    print('vrchol:', v, 'susedia:', v.sus, 'odmena:', v.odmena)
    print(lab.start(0, 0))
    print(lab.start(0, 2))
    lab.zmen_odmeny((2, 2))
    print(lab.start(0, 0))
```

vypíše:

```
vrchol: <1,0> susedia: [<0,0>, <2,0>] odmena: False
vrchol: <0,2> susedia: [<0,1>] odmena: True
[(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (1, 2), (1, 1), (0, 1), (0, 2)]
[(0, 2), (0, 1), (1, 1), (1, 2), (2, 2)]
[(0, 0), (0, 1), (0, 2)]
```

Prvé z riešení nájde túto trasu:



Obmedzenia

Vaše riešenie odovzdajte v súbore `riesenie11.py`, pričom prvé dva riadky súboru budú obsahovať:

```
# autor: Janko Hrasko
# uloha: 11. domace zadanie labyrint
```

zrejme ako autora uvediete svoje meno.

Testovač:

- odkontroluje, či sa vo vašom module nachádzať **len jedna definícia triedy** `Labyrint` s vnorenou triedou `Vrchol` (do existujúcich tried môžete pridávať vlastné atribúty a metódy)

- bude kontrolovať štruktúru vami vytvoreného grafu (atribút `g`) s rôznymi textovými súbormi, ktoré si môžete stiahnuť z L.I.S.T.u
- postupne preveruje vlastnosti vašich algoritmov, pri prvej chybe sa testovanie preruší a ďalšie časti sa netestujú:
  - 50% bodov za vytvorenie grafu
  - 50% bodov za algoritmus hľadania trasy v grafe

KAPITOLA 36

Riešenia niektorých cvičení

36.1 Riešenie 8. cvičenia

1. Napíšte funkciu `sucin(zoznam)`, ktorá vráti súčin prvkov zoznamu (obsahuje len čísla). Ak je zoznam prázdny, funkcia vráti `1`.

   - riešenie

```
def sucin(zoznam):
    vysl = 1
    for cislo in zoznam:
        vysl *= cislo
    return vysl
```

2. Napíšte funkciu `mocniny(n)`, ktorá vráti `n` prvkový zoznam druhých mocnín celých čísel `[1, 4, 9, 16, 25, ..., n**2]`

   - riešenie

```
def mocniny(n):
    vysl = []
    for i in range(1, n+1):
        vysl.append(i**2)
    return vysl
```

3. Napíšte funkciu `len_parne(zoznam)`, ktorá z daného zoznamu celých čísel, vyrobí nový, ale ponechá v ňom len párne hodnoty

   - riešenie

```
def len_parne(zoznam):
    vysl = []
    for cislo in zoznam:
        if cislo % 2 == 0:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            vysl.append(cislo)
    return vysl
```

4. Napíšte funkciu `spoj(zoznam)`, ktorá pre daný zoznam slov (znakových reťazcov) vytvorí jeden znakový reťazec zlepením všetkých slov v zozname. Slová v tomto výslednom reťazci budú oddelené medzerami. Nepoužite metódu `join()`.

   - riešenie

```
def spoj(zoznam):
    vysl = ''
    for slovo in zoznam:
        vysl += slovo + ' '
    return vysl[:-1]
```

5. Funkcia `zisti(zoznam)` zistí počet celých čísel v zadanom zozname, ktoré sú deliteľné 7

   - riešenie

```
def zisti(zoznam):
    vysl = 0
    for prvok in zoznam:
        if type(prvok) == int and prvok % 7 == 0:
            vysl += 1
    return vysl
```

6. Funkcia `zoznam2(n, hodn1, hodn2)` vytvorí `n` prvkový zoznam (predpokladajte, že `n` je párne), ktorý bude obsahovať striedajúce sa hodnoty `hodn1` a `hodn2`

   - riešenie

```
def zoznam2(n, hodn1, hodn2):
    return n // 2 * [hodn1, hodn2]
```

7. Funkcia `na_parnych(zoznam)` z daného zoznamu vytvorí nový, ktorý obsahuje len prvky na párnych indexoch (funkcia nemodifikuje vstupný zoznam)

   - riešenie

```
def na_parnych(zoznam):
    return zoznam[::2]
```

8. Funkcia `zostupne(zoznam)` zistí, či je daný `zoznam` utriedený **zostupne** (každý ďalší prvok v zozname nie je väčší ako predchádzajúci). Funkcia vráti `True` alebo `False` (funkcia nemodifikuje vstupný zoznam). Nepoužite štandardné funkcie ani metódy.

   - riešenie

```
def zostupne(zoznam):
    for i in range(1, len(zoznam)):
        if zoznam[i-1] < zoznam[i]:
            return False
    return True
```

9. Funkcia `vyrob1(zoznam)` vyrobí (vráti) kópiu celočíselného zoznamu, ale každé párne číslo pritom zväčší o 1 (funkcia nemodifikuje vstupný zoznam)

   - riešenie

```
def vyrob1(zoznam):
    vysl = []
    for cislo in zoznam:
        if cislo % 2:
            vysl.append(cislo)
        else:
            vysl.append(cislo + 1)
    return vysl
```

- alebo

```
def vyrob1(zoznam):
    vysl = list(zoznam)
    for i in range(len(vysl)):
        if vysl[i] % 2 == 0:
            vysl[i] += 1
    return vysl
```

10. Funkcia `vyrob2(zoznam)` vyrobí kópiu vstupného zoznamu ale v kópii ponechá len tie prvky, ktoré sú znakové reťazce (funkcia nemodifikuje vstupný zoznam)

    - riešenie

```
def vyrob2(zoznam):
    vysl = []
    for prvok in zoznam:
        if type(prvok) == str:
            vysl.append(prvok)
    return vysl
```

11. Funkcia `zoznam_cifier(cislo)` z daného nezáporného celého čísla vytvorí (vráti) zoznam cifier

    - riešenie

```
def zoznam_cifier(cislo):
    vysl = []
    for cifra in str(cislo):
        vysl.append(int(cifra))
    return vysl
```

    - riešenie

```
def zoznam_cifier(cislo):
    if cislo == 0:
        return [0]
    vysl = []
    while cislo:
        vysl.insert(0, cislo % 10)
        cislo //= 10
    return vysl
```

12. Funkcia `gener(a, b, c=1)` vytvorí (vráti) zoznam, ktorého prvky sú celočíselné hodnoty od `a` do `b-1` krokom `c` (rovnako ako `range(a, b, c)`). Nepoužite pritom štandardnú funkciu `range()`.

    - riešenie

```
def gener(a, b, c=1):
    vysl = []
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    if c > 0:
        while a < b:
            vysl.append(a)
            a += c
    else:
        while a > b:
            vysl.append(a)
            a += c
    return vysl
```

13. Funkcia `cele(zoznam)` zo zoznamu desatinných čísel vyrobí (vráti) zoznam celých čísel - ich celých častí (volaním funkcie `int()`). Funkcia nemodifikuje vstupný zoznam.

    - riešenie

```
def cele(zoznam):
    vysl = []
    for prvok in zoznam:
        vysl.append(int(prvok))
    return vysl
```

14. Napíšte funkciu `vymen(zoznam)`, ktorá dostane **dvojprvkový zoznam** a vráti nový zoznam, ktorý má tieto prvky vymenené

    - riešenie

```
def vymen(zoznam):
    return zoznam[::-1]
```

15. Napíšte funkciu `vymen2(zoznam)`, ktorá dostane **dvojprvkový zoznam** a navzájom vymení jeho prvky (bude **mutable**). Funkcia nič nevracia.

    - riešenie

```
def vymen2(zoznam):
    zoznam.append(zoznam.pop(0))
```

16. Napíšte funkciu `vyhod(zoznam, hodnota)`, ktorá z pôvodného zoznamu vytvorí nový ale už bez prvkov s danou hodnotou.

    - riešenie

```
def vyhod(zoznam, hodnota):
    vysl = []
    for prvok in zoznam:
        if prvok != hodnota:
            vysl.append(prvok)
    return vysl
```

17. Napíšte funkciu `vyhod2(zoznam, hodnota)`, ktorá v danom zozname vyhodí všetky výskyty danej hodnoty. Funkcia nič nevracia, ale zmení obsah zoznamu (**mutable**).

    - riešenie

```
def vyhod2(zoznam, hodnota):
    for i in reversed(range(len(zoznam))):
        if zoznam[i] == hodnota:
            del zoznam[i]       # alebo zoznam.pop(i)
```

18. Predpokladáme, že nejaký textový súbor má v každom riadku 1 alebo 2 celé čísla. Napíšte funkciu `zo_suboru(meno_suboru)`, ktorá prečíta tento súbor a vráti takýto zoznam: z každého riadku súboru vytvorí jeden prvok výsledného zoznamu, pričom, ak bolo v riadku len jedno číslo, toto bude priamo prvkom zoznamu, ak tam boli dve čísla, vytvorí dvojprvkový zoznam čísel

    - riešenie

```
def zo_suboru(meno_suboru):
    vysl = []
    with open(meno_suboru) as subor:
        for riadok in subor:
            r = riadok.split()
            if len(r) == 1:
                vysl.append(int(r[0]))
            else:
                vysl.append([int(r[0]), int(r[1])])
    return vysl
```

    - alebo

```
def zo_suboru(meno_suboru):
    vysl = []
    with open(meno_suboru) as subor:
        for riadok in subor:
            ix = riadok.strip().find(' ')
            if ix == -1:
                vysl.append(int(riadok))
            else:
                cislo1, cislo2 = int(riadok[:ix]), int(riadok[:ix+1])
                vysl.append([cislo1, cislo2])
    return vysl
```

19. Napíšte funkciu `do_suboru(meno_suboru, zoznam)`, ktorá zapíše do súboru daný zoznam. Každý prvok vstupného zoznamu bude zapísaný do samostatného riadku. Predpokladáme, že prvkami zoznamu môžu byť buď čísla, alebo dvojprvkové zoznamy čísel. Ak je prvkom číslo, v riadku súboru bude priamo toto číslo inak bude riadok obsahovať dve čísla oddelené medzerou (rovnaký formát súboru ako bol v predchádzajúcom príklade)

    - riešenie

```
def do_suboru(meno_suboru, zoznam):
    with open(meno_suboru, 'w') as subor:
        for prvok in zoznam:
            if type(prvok) == int:
                print(prvok, file=subor)
            else:
                print(prvok[0], prvok[1], file=subor)
```

20. Funkcia `krat2(zoznam)` vynásobí každý prvok zoznamu číslom 2; funkcia nič nevracia len mení obsah zoznamu (**mutable**)

    - riešenie

```
def krat2(zoznam):
    for i in range(len(zoznam)):
        zoznam[i] *= 2
```

21. Funkcia `zdvoj(zoznam)` do daného zoznamu pridá nové hodnoty tak, že každý prvok z pôvodného obsahu sa tu objaví dvakrát za sebou, funkcia nič nevracia, len modifikuje daný zoznam (**mutable**)

- riešenie

```
def zdvoj(zoznam):
    for i in reversed(range(len(zoznam))):
        zoznam.insert(i, zoznam[i])
```

- alebo

```
def zdvoj(zoznam):
    for i in range(0, 2 * len(zoznam), 2):
        zoznam.insert(i, zoznam[i])
```

22. Prvkami zoznamu sú čísla 0, 1 alebo 2 (v ľubovoľnom poradí). Napíšte funkciu `uprac(zoznam)`, ktorá preusporiada prvky zoznamu tak, že na začiatku zoznamu budú všetky 2, za tým 0 a na koniec 1, funkcia nič nevracia, len modifikuje daný zoznam (**mutable**)

- riešenie

```
def uprac(zoznam):
    index = 0
    for i in range(len(zoznam)):
        if zoznam[index] == 2:
            zoznam.pop(index)
            zoznam.insert(0, 2)
            index +=1
        elif zoznam[index] == 1:
            zoznam.pop(index)
            zoznam.append(1)
        else:
            index += 1
```

- alebo

```
def uprac(zoznam):
    zoznam[:] = [2] * zoznam.count(2) + [0] * zoznam.count(0) + [1] * zoznam.
↪count(1)
```

23. Funkcia `nahodny_zoznam(n, vyber)` vyrobí (vráti pomocou `return`) `n` prvkový zoznam, ktorého prvky sú náhodne vybrané hodnoty zo zoznamu `vyber`

- riešenie

```
def nahodny_zoznam(n, vyber):
    vysl = []
    for i in range(n):
        vysl.append(random.choice(vyber))
    return vysl
```

36.2 Riešenie 9. cvičenia

1. Napíšte funkciu `zisti_sucet(zoznam, sucet)`, ktorá nájde taký maximálny index do daného zoznamu, pre ktorý súčet všetkých prvkov pred daným indexom nebol väčší ako zadaná hodnota `sucet`. Riešte pomocou while-cyklus, ale nepoužite štandardnú funkciu `sum()`.

- riešenie

```
def zisti_sucet(zoznam, sucet):
    suma = index = 0
    while index < len(zoznam) and suma + zoznam[index] < sucet:
        suma += zoznam[index]
        index += 1
    return index
```

2. Napíšte funkciu `zluc(zoz1, zoz2)`, ktorá dostáva dva utriedené zoznamy a vytvorí z nich nový, ktorý obsahuje tie isté prvky ako boli v oboch zoznamoch a tiež je utriedený. Nepoužite `sort()` ani `sorted()`.

   - riešenie

```
def zluc(zoz1, zoz2):
    vysl = []
    i = j = 0
    while i+j < len(zoz1) + len(zoz2):
        if j >= len(zoz2) or i < len(zoz1) and zoz1[i] < zoz2[j]:
            vysl.append(zoz1[i])
            i += 1
        else:
            vysl.append(zoz2[j])
            j += 1
    return vysl
```

3. Napíšte funkciu `vnoreny_sucet(zoznam)`, pre ktorú vstupný zoznam (alebo n-tica) obsahuje len vnorené zoznamy alebo n-tice celých čísel a funkcia vráti súčet všetkých týchto čísel.

   - riešenie

```
def vnoreny_sucet(zoznam):
    vysl = 0
    for prvok in zoznam:
        vysl += sum(prvok)
    return vysl
```

4. Napíšte funkciu `daj_max(zoznam)`, ktorá vyhodí z daného zoznamu maximálny prvok (jeho prvý výskyt) a túto hodnotu vráti ako výsledok funkcie (`return`). Nepoužite metódu `remove()` ani štandardnú funkciu `max()`.

   - riešenie

```
def daj_max(zoznam):
    index = 0
    for i in range(len(zoznam)):
        if zoznam[i] > zoznam[index]:
            index = i
    return zoznam.pop(index)
```

5. Napíšte funkciu `zisti(meno_suboru)`, ktorá prečíta daný súbor a vypíše počet slov, najdlhšie slovo a slovo, ktoré je posledné v abecede (najväčšie zo všetkých). Nepoužívajte štandardnú funkciu `max()`.

   - riešenie

```
def zisti(meno_suboru):
    najdlhsie = posledne = ''
    with open(meno_suboru) as subor:
        vsetky = subor.read().split()
    for slovo in vsetky:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        if len(slovo) > len(najdlhsie):
            najdlhsie = slovo
        if slovo > posledne:
            posledne = slovo
    print('pocet slov: ', len(vsetky))
    print('najdlhsie: ', najdlhsie)
    print('posledne: ', posledne)
```

6. Napíšte funkciu `vyrovnaj(ntica)`, ktorá dostáva ako parameter n-ticu len s prvkami n-tíc. Funkcia vytvorí n-ticu, ktorá obsahuje všetky prvky vnorených n-tíc.

   - riešenie

```
def vyrovnaj(ntica):
    vysl = ()
    for prvok in ntica:
        vysl += prvok
    return vysl
```

7. Napíšte funkciu `histogram(pole)`, ktorá vypíše stĺpcový histogram z daného zoznamu, resp. n-tice kladných čísel.

   - riešenie

```
def histogram(pole):
    mx = max(pole)
    for i in range(mx):
        for j in range(len(pole)):
            if pole[j] >= mx - i:
                print('*', end=' ')
            else:
                print(' ', end=' ')
        print()
```

8. Hovoríme, že dve slová tvoria **anagram**, ak jedno slovo vznikne zmenou poradia písmen v druhom slove. Napr. tieto tri slová sú navzájom anagramami: ‚reklama‘, ‚makrela‘, ‚karamel‘. Napíšte funkciu `anagram(slovo1, slovo2)`, ktorá pre dve slová zistí, či sú anagramom. Funkcia vráti `True` alebo `False`.

   - riešenie

```
def anagram(slovo1, slovo2):
    zoz1 = list(slovo1)
    zoz1.sort()
    zoz2 = list(slovo2)
    zoz2.sort()
    return zoz1 == zoz2
```

9. Napíšte funkciu `ma_duplikaty(zoznam)`, ktorá pre daný zoznam, resp. n-ticu zistí, či sa tam nejaký prvok nachádza viackrát. Funkcia nemodifikuje vstupný zoznam a vráti `True` alebo `False`.

   - riešenie

```
def ma_duplikaty(zoznam):
    for i in range(len(zoznam) - 1):
        if zoznam[i] in zoznam[i+1:]:
            return True
    return False
```

10. Napíšte funkciu `pocet_dni_v_mesiaci(mesiac)`, ktorá vráti číslo od 28 do 31 pre mesiac od 1 do 12. Nepoužite príkaz `if`.

    - riešenie

```
def pocet_dni_v_mesiaci(mesiac):
    return (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)[mesiac - 1]
```

11. Napíšte funkciu `nahodny_datum()`, ktorá vygeneruje dvojicu (typu `tuple`), ktorá reprezentuje jeden deň v roku, t.j. `(den, mesiac)`. Počítajte len s nepriestupným rokom, teda každý dátum má pravdepodobnosť 1/365. Funkcia nič nevypisuje, ale vráti dvojicu.

    - riešenie

```
def nahodny_datum():
    mesiac = random.randrange(1, 13)
    den = random.randrange(1, pocet_dni_v_mesiaci(mesiac)+1)
    return den, mesiac
```

12. Napíšte funkciu `narodeninovy_problem(pocet)`, ktorá najprv vygeneruje príslušný `pocet` náhodných dátumov (využije funkciu z predchádzajúcej úlohy) a vráti `True`, ak sa medzi týmito dátumami nejaký opakuje. Dá sa matematicky ukázať, že už pre počet **23** je takáto pravdepodobnosť väčšia ako 50% (pozri narodeninový problém na wikipedii).

    - riešenie

```
def narodeninovy_problem(pocet):
    datumy = []
    for i in range(pocet):
        jeden = nahodny_datum()
        if jeden in datumy:
            return True
        datumy.append(jeden)
    return False
```

13. Napíšte funkciu `rozdel(retazec)`, ktorá vráti zoznam podreťazcov daného reťazca, ktorý sa rozdelil (pomocou `split()`) podľa riadkov a nie slov. Prázdne riadky na konci súboru odfiltrujte.

    - riešenie

```
def rozdel(retazec):
    vysl = retazec.split('\n')
    while vysl and vysl[-1] == '':
        vysl.pop()
    return vysl
```

14. Napíšte funkciu `na_cisla(pole)`, ktorá zo zadanej postupnosti (zoznam alebo n-tica) reťazcov vráti (`return`) zoznam celých čísel. Predpokladáme, že reťazce v postupnosti obsahujú len celé čísla.

    - riešenie

```
def na_cisla(pole):
    vysl = []
    for prvok in pole:
        vysl.append(int(prvok))
    return vysl
```

15. Napíšte funkciu `ciferny_sucet(cislo)`, ktorá pomocou funkcie `na_cisl()` z predchádzajúcej úlohy vypočíta ciferný súčet daného čísla.

- riešenie

```
def ciferny_sucet(cislo):
    return sum(na_cisla(str(cislo)))
```

16. Napíšte funkciu `len_int(ntica)`, ktorá v danej n-tici ponechá len celé čísla. Funkcia nič nevypisuje len vráti `tuple`.

    - riešenie

```
def len_int(ntica):
    vysl = ()
    for prvok in ntica:
        if type(prvok) == int:
            vysl += (prvok, )
    return vysl
```

17. Napíšte funkciu `zip(zoz1, zoz2)`, ktorá dostáva dva rovnako dlhé zoznamy (alebo n-tice) a vráti (`return`) jeden zoznam dvojíc (teda `list`, ktorého prvkami sú `tuple`): v každej dvojici je prvý prvok z prvého `zoz1` a druhý z druhého `zoz2`. Nepoužívajte štandardnú funkciu `zip()`.

    - riešenie

```
def zip(zoz1, zoz2):
    vysl = []
    for i in range(len(zoz1)):
        vysl.append((zoz1[i], zoz2[i]))
    return vysl
```

18. Napíšte funkciu `ocisluj(zoznam)`, ktorá vráti zoznam dvojíc (`list` s prvkami `tuple`): prvý prvok v dvojici je číslo od 0 do počet prvkov zoznamu - 1 a druhým prvkom je zodpovedajúci prvok daného zoznamu.

    - riešenie

```
def ocisluj(zoznam):
    vysl = []
    for i in range(len(zoznam)):
        vysl.append((i, zoznam[i]))
    return vysl
```

    - alebo

```
def ocisluj(zoznam):
    return list(enumerate(zoznam))
```

19. Napíšte funkciu `do_dvojic(zoznam)`, ktorá z daného zoznamu (alebo n-tice) vyrobí zoznam dvojíc (`list` s prvkami `tuple`): prvým prvkom dvojice budú postupne prvky daného zoznamu (okrem posledného) a druhým jeho nasledovným prvkom.

    - riešenie

```
def do_dvojic(zoznam):
    vysl = []
    for i in range(len(zoznam) - 1):
        vysl.append((zoznam[i], zoznam[i+1]))
    return vysl
```

    - alebo

```
def do_dvojic(zoznam):
    return zip(zoznam[:-1], zoznam[1:])
```

20. Napíšte funkciu `body(n, r, x, y)`, ktorá vráti zoznam dvojíc čísel. Tieto dvojice desatinných čísel reprezentujú daný počet bodov na kružnici s polomerom `r` a so stredom (`x`, `y`). Body sú rovnomerne rozložené na kružnici. Funkcia nič nekreslí ani nevypisuje len vráti `n`-prvkový zoznam súradníc.

    - riešenie

```
def body(n, r, x, y):
    vysl = []
    for i in range(n):
        uhol = math.radians(i * 360 / n)
        bod = x + r * math.cos(uhol), y + r * math.sin(uhol)
        vysl.append(bod)
    return vysl
```

21. Napíšte funkciu `uhlopriecky(n, r, x, y)`, ktorá vykreslí (pomocou `tkinter`) všetky strany a uhlopriečky `n`-uholníka, ktorého vrcholy ležia na kružnici s polomerom `r` a so stredom (`x`, `y`). Využite funkciu `body()` z predchádzajúcej úlohy.

    - riešenie

```
def uhlopriecky(n, r, x, y):
    vrcholy = body(n, r, x, y)
    for i in range(len(vrcholy)):
        for j in range(i, len(vrcholy)):
            canvas.create_line(vrcholy[i], vrcholy[j])
```

22. Napíšte funkciu `vektorovy_sucet(nt1, nt2)`, ktorá dostáva dve rovnako veľké n-tice čísel a vráti (`return`) n-ticu, v ktorej každý prvok súčtom zodpovedajúcich prvkov vo vstupných n-ticiach.

    - riešenie

```
def vektorovy_sucet(nt1, nt2):
    vysl = ()
    for i in range(len(nt1)):
        vysl += (nt1[i] + nt2[i],)
    return vysl
```

23. Napíšte funkciu `indexy(zoznam, hodnota)`, ktorá pre zadaný zoznam (alebo n-ticu) vráti postupnosť indexov (ako `tuple`), na ktorých sa zadaná hodnota v danom zozname vyskytuje.

    - riešenie

```
def indexy(zoznam, hodnota):
    vysl = ()
    for i in range(len(zoznam)):
        if zoznam[i] == hodnota:
            vysl += (i,)
    return vysl
```

36.3 Riešenie 12. cvičenia

4. Funkcia `urob(a, b)` z prvej úlohy počíta bez cyklov súčin dvoch nezáporných celých čísel a to len pomocou sčitovania. Napíšte funkciu `umocni(a, b)`, ktorá pre dve celé čísla vypočíta mocninu `a**b` ale bez cyklu a

bez násobenia - na násobenie použite funkciu `urob()`.

- riešenie:

```
def urob(a, b):
    if a == 0:
        return 0
    return b + urob(a - 1, b)

def umocni(a, b):
    if b==0:
        return 1
    return urob(a, umocni(a, b-1))
```

5. Napíšte dve rekurzívne funkcie `tostr(cislo)` a `toint(retazec)`, ktoré bez cyklov a len pomocou štandardných funkcií `ord()` a `chr()` prevedú celé nezáporné číslo na znakový reťazec a naopak. Obe funkcie nič nevypisujú len vracajú nejakú hodnotu.

   - riešenie:

```
def tostr(cislo):
     if cislo < 10:
         return chr(ord('0')+cislo)
     return tostr(cislo // 10) + tostr(cislo % 10)

def toint(retazec):
    if retazec == '':
        return 0
    return ord(retazec[-1]) - ord('0') + 10 * toint(retazec[:-1])
```

6. Napíšte rekurzívnu funkciu `pocet(znak, retazec)`, ktorá bez cyklu a reťazcových metód zistí počet výskytov zadaného znaku vo vstupnom reťazci.

   - riešenie:

```
def pocet(znak, retazec):
    if retazec == '':
        return 0
    if retazec[0] == znak:
        return 1 + pocet(znak, retazec[1:])
    else:
        return pocet(znak, retazec[1:])
```

   - alebo

```
def pocet(znak, retazec):
    if retazec == '':
        return 0
    return int(retazec[0] == znak) + pocet(znak, retazec[1:])
```

7. Prechádzajúci príklad vyriešte tak, aby fungoval aj pre dlhšie reťazce. Inšpirujte sa funkciou `otoc()` z prednášky

   - riešenie:

```
def pocet(znak, retazec):
    if len(retazec) <= 1:
        return int(retazec == znak)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    stred = len(retazec) // 2
    return pocet(znak, retazec[:stred]) + pocet(znak, retazec[stred:])
```

8. Napíšte funkciu `vela(n)`, ktorá vráti znakový reťazec `'='*(2**n)`, t.j. obsahuje len znak `'='`, ktorý sa opakuje `2**n` krát. Riešte rekurzívne bez cyklov a viacnásobného zreťazovania reťazcov (bez operácie `*` s reťazcami).

   - riešenie:

```
def vela(n):
    if n == 0:
        return '='
    return vela(n-1) + vela(n-1)
```

9. Zapíšte funkciu `nsd(a, b)` (najväčší spoločný deliteľ) rekurzívne: triviálny prípad je vtedy, keď `a==b`, inak ak `a>b`, tak rekurzívne vypočíta `nsd(b, a)`, inak rekurzívne zavolá `nsd(a, b-a)`.

   - riešenie:

```
def nsd(a, b):
    if a == b:
        return a
    if a > b:
        return nsd(b, a)
    return nsd(a, b-a)
```

10. Napíšte rekurzívnu funkciu `sucet(zoznam)`, ktorá bez cyklov zistí súčet prvkov zoznamu. Prvkami sú len celé čísla.

    - riešenie:

```
def sucet(zoznam):
    if zoznam == []:
        return 0
    if len(zoznam) == 1:
        return zoznam[0]
    stred = len(zoznam) // 2
    return sucet(zoznam[:stred]) + sucet(zoznam[stred:])
```

11. Upravte funkciu `sucet(zoznam)` z predchádzajúceho príkladu tak, aby prvkami zoznamu mohli byť nielen celé čísla, ale aj zoznamy, ktoré obsahujú celé čísla.

    - riešenie:

```
def sucet(zoznam):
    if zoznam == []:
        return 0
    if len(zoznam) == 1:
        if type(zoznam[0]) == list:
            return sucet(zoznam[0])
        else:
            return zoznam[0]
    stred = len(zoznam) // 2
    return sucet(zoznam[:stred]) + sucet(zoznam[stred:])
```

Rekurzívne krivky

13. Na prednáške sme kreslili binárny strom.
    - dopíšte do tejto funkcie kreslenie farebných bodiek (pomocou `t.dot(10, farba)`) na koncových vetvičkách tak, aby sa pravidelne striedali dve farby červená a žltá; okrem korytnačky `t` nepoužívajte iné globálne premenné
    - riešenie:

```
def strom(n, d, farba):
    t.fd(d)
    if n == 0:
        t.dot(10, farba)
    if n > 0:
        t.lt(40)
        strom(n - 1, d * 0.7, 'red')
        t.rt(75)
        strom(n - 1, d * .6, 'yellow')
        t.lt(35)
    t.bk(d)
```

14. Nakreslite rekurzívnu krivku `stvorce(n, a)`, ktorá pre `n > 0` nakreslí štvorec so stranou `a`, v ktorom sú vpísané štvorce (s vrcholmi v stredoch strán vonkajšieho štvorca). Tieto vnorené štvorce vzniknú volaním `stvorce(n-1, ...)`. Pre `n=2` sme to riešili bez rekurzie na predchádzajúcom cvičení.
    - riešenie:

```
def stvorce(n, a):
    t.fd(a/2)
    if n > 0:
        t.lt(45)
        stvorce(n-1, a*2**0.5/2)
        t.rt(45)
    t.fd(a/2)
    t.lt(90)
    for i in range(3):
        t.fd(a)
        t.lt(90)
```

    - malou zmenou rekurzívnej funkcie vieme nakresliť napr. vpísané trojuholníky
    - riešenie:

```
def trojuholniky(n, a):
    t.fd(a/2)
    if n > 0:
        t.lt(60)
        trojuholniky(n-1, a/2)
        t.rt(60)
    t.fd(a/2)
    t.lt(120)
    for i in range(2):
        t.fd(a)
        t.lt(120)
```

15. Zadefinujte funkciu `kriziky(n, d)`, ktorá nakreslí takúto rekurzívnu krivku:
    - pre `n=0` funkcia nerobí nič

- pre `n=1` funkcia nakreslí kríž (4 na seba kolmé čiary zadanej dĺžky `d`) - kreslenie sa skončí tam, kde sa začalo
- pre `n=2` funkcia opäť nakreslí kríž zadanej veľkosti, ale na konci všetkých 4 ramien kríža nakreslí tiež kríž ale s ramenami tretinovej veľkosti, pričom tieto menšie krížiky budú oproti ramenu otočené o 45 stupňov (nakreslí sa 1 kríž veľkosti `d` a 4 veľkosti `d/3`)
- každá ďalšia úroveň krivky nakreslí menšie a menšie krížiky na konci ramien vnorených krížov (pre `n=3` sa nakreslí 1 kríž veľkosti `d`, 4 veľkosti `d/3` a 16 veľkosti `d/9`)
- riešenie:

```
def kriziky(n, d):
    if n > 0:
        for i in range(4):
            t.fd(d)
            if n > 1:
                t.rt(45)
                kriziky(n-1, d/3)
                t.lt(45)
            t.bk(d)
            t.rt(90)
```

- malou zmenou rekurzívnej funkcie vieme zmeniť 4 ramená krížikov na ľubovoľný iný počet
- riešenie:

```
def kriziky(n, d, p=4):
    if n > 0:
        for i in range(p):
            t.fd(d)
            if n > 1:
                t.rt(180/p)
                kriziky(n-1, d/3, p)
                t.lt(180/p)
            t.bk(d)
            t.rt(360/p)
```

36.4 Riešenie 20. cvičenia

36.4.1 Zbalené a rozbalené parametre

1. Napíšte funkciu `ntica`, ktorá bude mať ľubovoľný počet parametrov a vráti n-ticu z týchto parametrov
   - riešenie:

```
def ntica(*param):
    return tuple(param)
```

2. Napíšte funkciu `min` s ľubovoľným počtom parametrov, ktorá vráti najmenšiu hodnotu medzi parametrami.
   - riešenie:

```
def min(*param):
    vysl = None
    for prvok in param:
        if vysl is None or prvok < vysl:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            vysl = prvok
    return vysl
```

3. Napíšte funkciu `zisti` s ľubovoľným počtom parametrov, ktorá zistí (vráti `True`), či aspoň jeden z parametrov je n-tica (typ `tuple`).

   - riešenie:

```
def zisti(*param):
    for prvok in param:
        if isinstance(prvok, tuple):
            return True
    return False
```

4. Napíšte funkciu `zlep` s ľubovoľným počtom parametrov, pričom všetky sú typu `list`. Výsledkom funkcie je zreťazenie všetkých týchto parametrov.

   - riešenie:

```
def zlep(*param):
    vysl = []
    for prvok in param:
        vysl += prvok       # alebo vysl.extend(prvok)
    return vysl
```

5. Napíšte funkciu `vypis(zoznam)`, ktorá pomocou `print` vypíše všetky prvky zoznamu do jedného riadka. Nepoužite for-cyklus.

   - riešenie:

```
def vypis(zoznam):
    print(*zoznam)
```

36.4.2 Funkcie ako parametre

6. Napíšte funkciu `retazec(zoznam)`. Funkcia vráti znakový reťazec, ktorý reprezentuje prvky zoznamu. Prvky zoznamu budú v reťazci oddelené znakom bodkočiarka. Nepoužite žiadne cykly, ale namiesto toho štandardnú funkciu `map` a metódu `join`.

   - riešenie:

```
def retazec(zoznam):
    return '; '.join(map(repr, zoznam))
```

7. Napíšte funkciu `aplikuj`, ktorej parametrami sú nejaké funkcie, okrem posledného parametra, ktorým je nejaká hodnota. Funkcia postupne zavolá všetky tieto funkcie s danou hodnotou, pričom každú ďalšiu funkciu aplikuje na predchádzajúci výsledok. Napr. `aplikuj(f1, f2, f3, x)` vypočíta `f3(f2(f1(x)))`. Funkcia by mala pracovať pre ľubovoľný nenulový počet parametrov.

   - riešenie:

```
def aplikuj(*param):
    hodnota = param[-1]
    for funkcia in param[:-1]:
        hodnota = funkcia(hodnota)
    return hodnota
```

8. Napíšte funkciu `urob(k)`. Funkcia ako svoj výsledok **vráti funkciu** s jedným parametrom, ktorá bude počítať `k`-tu mocninu parametra.

   - riešenie:

```
def urob(k):
    return lambda x: x**k
```

   alebo:

```
def urob(k):
    def fun(x):
        return x**k
    return fun
```

36.4.3 Generátorová notácia

9. Napíšte funkciu `mocniny(n)`, ktorá vráti zoznam druhých mocnín čísel od 1 do `n`.

   - riešenie:

```
def mocniny(n):
    return [i**2 for i in range(1, n+1)]
```

10. Napíšte funkciu `zistí(veta)`, ktorá zistí dĺžku najdlhšieho slova vo vete.

    - riešenie:

```
def zisti(veta):
    return max(len(slovo) for slovo in veta.split())
```

11. Napíšte funkciu `prevrat_slova(veta)`, ktorá vráti zadanú vetu tak, že každé slovo v nej bude otočené.

    - riešenie:

```
def prevrat_slova(veta):
    return ' '.join(slovo[::-1] for slovo in veta.split())
```

12. Napíšte funkciu `najdlhsie_slovo(veta)`, ktorá vráti najdlhšie slovo vo vete. Riešte to takto:

    - funkcia najprv z danej vety vytvorí postupnosť dvojíc (dĺžka slova, slovo)
    - pomocou `sorted()` túto postupnosť dvojíc utriedi
    - funkcia vráti slovo z poslednej dvojice (je to najdlhšie slovo) utriedenej postupnosti
    - riešenie:

```
def najdlhsie_slovo(veta):
    return sorted((len(slovo), slovo) for slovo in veta.split())[-1][1]
```

   alebo by fungovalo aj:

```
def najdlhsie_slovo(veta):
    return sorted((len(slovo), slovo) for slovo in veta.split())[-1][1]
```

13. Napíšte funkciu `zoznam2(m, n, hodnota=None)`, ktorá vygeneruje dvojrozmerný zoznam veľkosti `m` x `n` pričom všetky prvky majú zadanú hodnotu

    - riešenie:

```
def zoznam2(m, n, hodnota=None):
    return [n*[hodnota] for i in range(m)]
```

14. Predpokladáme, že textový súbor v každom riadku obsahuje niekoľko celých čísel. Napíšte funkciu `citaj_zoznam(meno_suboru)`, ktorá z neho vytvorí dvojrozmerný zoznam čísel.

    - riešenie:

```
def citaj_zoznam(meno_suboru):
    with open(meno_suboru) as subor:
        return [list(map(int, riadok.split())) for riadok in subor.read().
↪split('\n')]
```

15. Napíšte funkciu `rozdel(zoznam, x)`, ktorá ako výsledok vráti dva zoznamy: prvý obsahuje všetky menšie prvky ako `x` a druhý všetky zvyšné.

    - riešenie:

```
def rozdel(zoznam, x):
    mensi = [prvok for prvok in zoznam if prvok < x]
    vacsi = [prvok for prvok in zoznam if prvok >= x]
    return mensi, vacsi
```

16. Už poznáme **štandardnú funkciu** `enumerate()`, ktorá z danej postupnosti vracia postupnosť dvojíc. Napíšte vlastnú funkciu `enumerate(postupnost)`, ktorá vytvorí takýto zoznam dvojíc (`list` s prvkami `tuple`): prvým prvkom bude poradové číslo dvojice a druhým prvkom prvok zo vstupnej postupnosti.

    - riešenie:

```
def enumerate(postupnost):
    return [(i, postupnost[i]) for i in range(len(postupnost))]
```

17. Napíšte funkciu `zip(p1, p2)`, ktorá z dvoch postupností rovnakých dĺžok vytvorí zoznam zodpovedajúcich dvojíc, t.j. zoznam v ktorom prvým prvkom bude dvojica prvých prvkov postupností, druhým prvkom dvojica druhých prvkov, ...

    - riešenie:

```
def zip(p1, p2):
    return [(p1[i], p2[i]) for i in range(len(p1))]
```

    - pokúste sa to zapísať tak, aby to fungovala aj pre postupnosti rôznych dĺžok: vtedy vytvorí len toľko dvojíc, koľko je prvkov v kratšej z týchto postupností, napr.

```
def zip(p1, p2):
    return [(p1[i], p2[i]) for i in range(min(len(p1), len(p2)))]
```

    - pokúste sa to zapísať tak, aby funkcia fungovala pre ľubovoľný počet ľubovoľne dlhých postupností, napr.

```
def zip(*postupnosti):
    dlzka = min(map(len, postupnosti))
    return [tuple(p[i] for p in postupnosti) for i in range(dlzka)]
```

36.5 Riešenie 25. cvičenia

1. Zistite, čo by sa vypísalo.
   - riešenie:

```
>>> s = Stack()
>>> s.push(7); s.push(10); s.push(13); s.pop(); s.push(9); s.pop(); s.pop()
13
9
10
>>> s.push(2); s.push(5); s.pop(); s.pop(); s.pop(); s.pop()
5
2
7
...
EmptyError: prazdny zasobnik
```

2. Zistite, čo by sa vypísalo.
   - riešenie:

```
1 4 7
9 6 3 0
```

3. Zistite, čo by sa vypísalo.
   - riešenie:

```
p n o h t y
```

4. Vytvorte súbor `struktury.py` a prekopírujte do neho definíciu triedy `Stack` z prednášky. V ďalších úlohách budete používať `import` z tohto súboru. Do inicializácie triedy `Stack` ešte pridajte nepovinný parameter `seq`, ktorý označuje postupnosť hodnôt, ktorou sa inicializuje zásobník (použite metódu `push()`). Pridajte ešte novú metódu `get_list()`, ktorá pomocou volania `pop()` vráti zoznam všetkých prvkov zasobníka (zásobník sa pritom vyprázni).
   - riešenie:

```
class Stack:

    def __init__(self, seq=None):
        '''inicializuje zoznam'''
        self._prvky = []
        for p in seq or []:
            self.push(p)

    ...

    def get_list(self):
        res = []
        while not self.is_empty():
            res.append(self.pop())
        return res
```

5. Napíšte funkciu `dno(stack)`, ktorá z daného zásobníka vyberie a vráti prvok z dna tohto zásobníka. Zvyšné prvky v ňom zostanú nezmenené. Použite pomocný zásobník.
   - riešenie:

```
def dno(s):
    s1 = Stack()
    while not s.is_empty():
        s1.push(s.pop())
    res = s1.pop()
    while not s1.is_empty():
        s.push(s1.pop())
    return res
```

6. V prednáške bola funkcia `palindrom()`, ktorá pomocou zásobníka kontrolovala, či je daná postupnosť palindrom. Lenže pritom každú dvojicu prvkov zoznamu kontrolovala dvakrát (prvý prvok s posledným, druhý s predposledným, ..., predposledný s druhým, posledný s prvým). Zrejme by na zisťovanie palindromu stačila polovica testov. Opravte túto funkciu tak, aby sa využil zásobník a pritom bolo vo funkcii maximálne polovica porovnaní. Vtedy aj do zásobníka stačí vložiť len polovicu prvkov zoznamu.

   - riešenie:

```
def palindrom(post):
    stack = Stack()
    n = len(post)
    for prvok in post[:n // 2]:
        stack.push(prvok)
    for prvok in post[n - n // 2:]:
        if prvok != stack.pop():
            return False
    return True
```

7. Ručne prepíšte infixové zápisy do prefixu aj postfixu:

   - riešenie:

```
7 * 6 * 5 * 4 * 3 * 2
prefix: * * * * * 7 6 5 4 3 2
postfix: 7 6 * 5 * 4 * 3 * 2 *

(1 + 2) * 3 / (4 + 5) * 6
prefix: / * + 1 2 3 * + 4 5 6
postfix: 1 2 + 3 * 4 5 + 6 * /
```

8. Ručne vyhodnoťte tieto zápisy

   - riešenie:

```
>>> pocitaj_prefix('+ 8 * / - 14 6 3 - 8 * 2 3')
12
>>> pocitaj('1 2 3 * + 4 5 * + 6 7 * +')
69
```

9. Opravte funkciu `pocitaj()` z prednášky, ktorá vyhodnocuje postfix tak, aby správne reagovala na chyby. Na každú chybovú situáciu sa vyvolá výnimka `ExpressionError` s príslušným komentárom, napr.

   - riešenie:

```
from struktury import Stack, EmptyError

class ExpressionError(Exception): pass

def pocitaj(vyraz):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    s = Stack()
    for prvok in vyraz.split():
        try:
            if prvok == '+':
                s.push(s.pop() + s.pop())
            elif prvok == '-':
                s.push(-s.pop() + s.pop())
            elif prvok == '*':
                s.push(s.pop() * s.pop())
            elif prvok == '/':
                op2, op1 = s.pop(), s.pop()
                s.push(op1 // op2)
            else:
                s.push(int(prvok))
        except ZeroDivisionError:
            raise ExpressionError('delenie nulou')
        except ValueError:
            raise ExpressionError('očakávalo sa celé číslo')
        except EmptyError:
            raise ExpressionError('málo operandov pre operáciu')
    if s.is_empty():
        raise ExpressionError('prázdny výraz')
    vysl = s.pop()
    if s.is_empty():
        return vysl
    raise ExpressionError('málo operátorov pre tol'ko operandov')
```

10. Odrekurzívnite rekurzívnu krivku Sierpiňského trojuholník z 12. prednášky v zimnom semestri.

    - riešenie:

```
from struktury import Stack

def trojuholniky(n, a):
    s = Stack()
    s.push((1, n, a))
    while not s.is_empty():
        x, n, a = s.pop()
        if n > 0:
            if x == 1:
                t.fd(a)
                t.rt(120)
                #trojuholniky(n - 1, a / 2)
                s.push((2, n, a))
                s.push((1, n - 1, a / 2))
            elif x == 2:
                t.fd(a)
                t.rt(120)
                #trojuholniky(n - 1, a / 2)
                s.push((3, n, a))
                s.push((1, n - 1, a / 2))
            elif x == 3:
                t.fd(a)
                t.rt(120)
                #trojuholniky(n - 1, a / 2)
                s.push((4, n, a))
                s.push((1, n - 1, a / 2))
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
t = turtle.Turtle()
trojuholniky(4, 200)
```

- alebo to isté zapísané kompaktnejšie:

```
from struktury import Stack

def trojuholniky(n, a):
    s = Stack()
    s.push((1, n, a))
    while not s.is_empty():
        x, n, a = s.pop()
        if n > 0 and x < 4:
            t.fd(a)
            t.rt(120)
            s.push((x + 1, n, a))
            s.push((1, n - 1, a / 2))

t = turtle.Turtle()
trojuholniky(4, 200)
```

11. Do súboru `struktury.py` pridajte aj definíciu triedy `Queue` z prednášky. Naprogramujte aj tieto dve funkcie s parametrom typu `Queue` tak, aby sa nepoškodil obsah radu:

    - `pocet(rad)` zistí počet prvkov v rade
    - `posledny(rad)` vráti hodnotu posledného prvku radu (posledne pridaného prvku)

    Pre obe funkcie to riešte najprv s pomocným radom (podobne, ako sme to riešili so zásobníkom) a potom aj bez pomocného radu (resp. inej štruktúry). Vyberané prvky totiž nemusíte ukladať niekam inam a potom ich vracať späť, ale ukladáte ich na koniec samotného radu.

    - riešenie:

```
from struktury import Queue, EmptyError

def pocet(rad):
    pom = Queue()
    vysl = 0
    while not rad.is_empty():
        pom.enqueue(rad.dequeue())
        vysl += 1
    while not pom.is_empty():
        rad.enqueue(pom.dequeue())
    return vysl

def posledny(rad):
    if rad.is_empty():
        raise EmptyError('prazdny rad')
    pom = Queue()
    while not rad.is_empty():
        vysl = rad.dequeue()
        pom.enqueue(vysl)
    while not pom.is_empty():
        rad.enqueue(pom.dequeue())
    return vysl
```

- riešenie bez pomocného radu:

```
from struktury import Queue, EmptyError

def pocet(rad):
    zarazka = ...   # hodnota, ktora nebude prvkom radu
    vysl = 0
    rad.enqueue(zarazka)
    while True:
        prvok = rad.dequeue()
        if prvok == zarazka:
            break
        rad.enqueue(prvok)
        vysl += 1
    return vysl

def posledny(rad):
    if rad.is_empty():
        raise EmptyError('prazdny rad')
    zarazka = ...   # hodnota, ktora nebude prvkom radu
    rad.enqueue(zarazka)
    while True:
        prvok = rad.dequeue()
        if prvok == zarazka:
            break
        rad.enqueue(prvok)
        vysl = prvok
    return vysl
```

12. Naprogramujte dve funkcie `otoc_zasobnik(stack)` a `otoc_rad(queue)`, ktoré obe otočia poradie svojej vstupnej dátovej štruktúry

    - na otáčanie zásobníka použite pomocný rad a naopak na otáčanie radu použite pomocný zásobník
    - riešenie:

```
from struktury import Stack, Queue

def otoc_zasobnik(stack):
    pom = Queue()
    while not stack.is_empty():
        pom.enqueue(stack.pop())
    while not pom.is_empty():
        stack.push(pom.dequeue())

def otoc_rad(queue):
    pom = Stack()
    while not queue.is_empty():
        pom.push(queue.dequeue())
    while not pom.is_empty():
        queue.enqueue(pom.pop())
```

13. Do inicializácie triedy `Stack` okrem nepovinnej vstupnej postupnosti pridajte aj parameter `maxlen`, ktorý obmedzí dĺžku, pri prekročení ktorej (pri volaní `push()`) sa najstarší prvok vyhodí - takýto zásobník bude mať maximálnu dĺžku `maxlen`. Ak má `maxlen` hodnotu `None`, zásobník pracuje normálne bez obmedzenia.

    - riešenie:

```
class EmptyError(Exception): pass

class Stack:

    def __init__(self, seq=None, maxlen=None):
        '''inicializuje zoznam'''
        self._prvky = []
        self._maxlen = maxlen
        for p in seq or []:
            self.push(p)

    def push(self, data):
        '''na vrch zasobnika vlozi novu hodnotu'''
        if self._maxlen is not None and len(self._prvky) == self._maxlen:
            del self._prvky[0]
        self._prvky.append(data)

    ...
```

36.6 Riešenie 26. cvičenia

1. Bez spúšťania na počítači zistite, čo urobia nasledovné programy.

   - prvý spájaný zoznam:

```
v1 = Vrchol('A')
v2 = Vrchol('B')
v3 = Vrchol('C')
v4 = Vrchol('D')
v3.next = v1
v1.next = v2
v2.next = v3
v1.next = v4
zoz = v2
vypis(zoz)
```

   - riešenie:

```
B -> C -> A -> D -> None
```

   - druhý spájaný zoznam:

```
v1 = None
v2 = Vrchol('X', v1)
v2 = Vrchol('Y', v2)
v3 = Vrchol('Z', v1)
v3 = Vrchol('T', v3)
v2 = Vrchol('U', v2)
v3.next.next = v2
vypis(v3)
```

   - riešenie:

```
T -> Z -> U -> Y -> X -> None
```

2. Napíšte funkciu `pripocitaj1(zoznam)`, ktorá ku každému prvku zoznamu pripočíta 1, ale len vtedy, ak sa dá (nevznikne pritom chyba), inak tento prvok nezmení a pokračuje na ďalších. Funkcia nič nevracia.

   - riešenie:

```
def pripocitaj1(zoznam):
    while zoznam is not None:
        try:
            zoznam.data += 1
        except TypeError:
            pass
        zoznam = zoznam.next
```

3. Prerobte funkciu `vypis()` tak, aby najprv vytvorila zoznam reťazcov z jednotlivých prvkov spájaného zoznamu a až na záver pomocou `' -> '.join(zoznam)` z toho vyrobí reťazec, ktorý vypíše

   - riešenie:

```
def vypis(zoznam):
    lst = []
    while zoznam is not None:
        lst.append(str(zoznam.data))
        zoznam = zoznam.next
    lst.append('None')
    print(' -> '.join(lst))
```

4. Bez spúšťania na počítači zistite, čo urobí:

   - tretí spájaný zoznam:

```
zoz = vyrob((1, 3, 5, 7, 9, 11, 13))
v = zoz.next.next
v1 = v.next.next
v.next.next = v1.next
v1.next = v.next
v.next = v1
vypis(zoz)
```

   - riešenie:

```
1 -> 3 -> 5 -> 9 -> 7 -> 11 -> 13 -> None
```

5. Napíšte rekurzívnu verziu funkcie `pocet(zoznam)`, t.j. funkcia prejde všetky prvky zoznamu bez použitia cyklu len pomocou rekurzie. Nepridávajte ďalšie parametre do definície funkcie.

   - riešenie:

```
def pocet(zoznam):
    if zoznam is None:
        return 0
    return 1 + pocet(zoznam.next)
```

6. Napíšte funkciu `spoj(zoz1, zoz2)`, ktorá na koniec zoznamu `zoz1` pripojí zoznam `zoz2`. Funkcia ako výsledok vráti začiatok takéhoto nového zoznamu. Nepoužívajte žiadne pomocné zoznamy (napr. typu `list`).

   - riešenie:

```
def spoj(zoz1, zoz2):
    if zoz1 is None:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        return zoz2
    pom = zoz1
    while pom.next is not None:
        pom = pom.next
    pom.next = zoz2
    return zoz1
```

7. Napíšte funkciu `prevratena_kopia(zoznam)`, ktorá vytvorí a vráti z daného zoznamu nový zoznam. Tento bude mať všetky prvky z pôvodného v opačnom poradí. Pôvodný zoznam musí ostať bez zmeny. Nepoužívajte žiadne pomocné zoznamy (typ `list`).

   - riešenie:

```
def prevratena_kopia(zoznam):
    novy = None
    while zoznam is not None:
        novy = Vrchol(zoznam.data, novy)
        zoznam = zoznam.next
    return novy
```

8. Napíšte funkciu `oprav(zoznam, funkcia)`, ktorá pre každý vrchol v danom zozname spustí zadanú funkciu s parametrom hodnota vo vrchole a ak to nespadne na chybe, zmení hodnotu vrcholu. Funkcia nič nevracia.

   - riešenie:

```
def oprav(zoznam, funkcia):
    while zoznam is not None:
        try:
            zoznam.data = funkcia(zoznam.data)
        except:
            pass
        zoznam = zoznam.next
```

9. Napíšte funkciu `vyhod_prvy(zoznam)`. Funkcia vráti pôvodný zoznam bez prvého prvku

   - riešenie:

```
def vyhod_prvy(zoznam):
    if zoznam is None:
        return zoznam
    return zoznam.next
```

10. Napíšte funkciu `vyhod_posledny(zoznam)`. Funkcia vráti pôvodný zoznam bez posledného prvku

    - riešenie:

```
def vyhod_posledny(zoznam):
    if zoznam is None or zoznam.next is None:
        return None
    zoz = zoznam
    while zoz.next.next is not None:
        zoz = zoz.next
    zoz.next = None
    return zoznam
```

11. Napíšte funkciu `vyhod_kazdy_druhy(zoznam)`, ktorá zo zoznamu vyhodí každý druhý prvok. Funkcia nič nevracia.

    - riešenie:

```
def vyhod_kazdy_druhy(zoznam):
    while zoznam is not None:
        if zoznam.next is not None:
            zoznam.next = zoznam.next.next
        zoznam = zoznam.next
```

12. Napíšte funkciu `vyhod(zoznam, podmienka)`, ktorá vyhodí všetky prvky zo zoznamu, pre ktoré zavolanie parametra `podmienka` s hodnotou vo `data` vo vrchole vráti `True`.

    - riešenie:

```
def vyhod(zoznam, podmienka):
    while zoznam is not None and podmienka(zoznam.data):
        zoznam = zoznam.next
    if zoznam is None:
        return None
    zoz = zoznam
    while zoz.next is not None:
        if podmienka(zoz.next.data):
            zoz.next = zoz.next.next
        else:
            zoz = zoz.next
    return zoznam
```

KAPITOLA 37

Prílohy

37.1 Výsledky priebežného testu

študent	1	2	3	4	5	6	7	8	9	10	súčet	
Babál Šimon	1	1.8	0	1	2	0	1.5	1.5	2.5	0	11.3	
Balintová Iveta	2	1	1.2	2	1.6	1.7	1.5	0.5	0.7	0	12.2	
Baluch Miroslav	1.5	1.5	0	0.3	0	1.7	1	0.5	1.5	0	8	
Bartoš Jakub	2	1.5	1.5	-	1	0	1.5	1.5	1.2	0.5	10.7	
Benková Veronika	0.5	0	0.5	0.3	-	0	1	0	0.2	0	2.5	
Bilichenko Oleksiy	0	1.5	-	-	1.5	-	1.5	0.5	1.2	0	6.2	
Bíly Erik	2	0.8	1.5	2	1.6	2	1.5	1.5	-	-	12.9	
Budinský Martin	1.5	0	0	-	0.5	0	1	1	0	0	4	
Csitáriová Radoslava	1.5	-	-	-	-	0	1.5	1	0	0	4	
Čapkovič Denis	1.5	1	0.8	2	2.2	2	1.5	1.5	2.5	1	15	
Čech Daniel	1.5	0.5	1	0	0	0	1.5	0.5	1	0	6	
Dodoková Katarína	2	2	1.5	2	0	1.9	1.5	2	1.2	1	15	
Dojčan Nicolas	1	-	1	0	-	-	1.5	-	-	-	3.5	
Dominik Richard	2	0.5	0.5	0.5	0	0	1.5	1	0.5	0	6.5	
Dóša Barnabás	0	0	0	1	-	-	0	0.5	0.2	0	1.7	
Dráb Matej	1.5	1	0.8	2	-	0	1.5	0.5	1.2	1	9.5	
Drastich Šimon	1	1	0.5	0	-	1	1.5	1	0.2	0	6.2	
Drgoňová Veronika	2	1.5	0.5	1	-	0	1	2	2	-	10	
Ďurana Marek	2	1	1.2	2	0.5	0	1	0.3	0.2	0	8.2	
Ďurica Andrej	1.5	1	1	2	-	1.2	1.5	1.5	1	0.5	11.2	
Eliaš Filip	2	0.8	0.5	0.5	1.3	0	1.5	1	0.3	0	7.9	
Gajdoš Radoslav	1	-	0	0	-	0	-	-	0.5	-	1.5	
Gál Matúš	0	0.8	0.5	2	2	1	1.5	2	2	0	11.8	
Gyurcsovicsová Tatiana	2	1	0.5	2	-	2	1.5	2.5	1	1	13.5	
Hajná Andrea	1	1	0	0	0	0.3	1.5	1	0.8	0	5.6	
Halagačka Jozef	2	0	1.5	1.8	-	0	-	-	-	-	5.3	
Harnádek Juraj	1	0	-	0.3	0	0	1	1	0	0	3.3	
Homola Andrej	1.5	1	0	0	1.2	0	0	1	0.2	0	4.9	
Horecká Rebeka	1.5	-	0.2	0.3	0	0	1.5	0.5	0.2	0	4.2	
Horníková Nikola	1	1.5	1.2	2	0.5	0	1.5	1	2.3	0	11	
Hrtánek Viktor	1.5	1	0.3	2	0	0	1	0.5	1	0	7.3	
Chorvatovič Michal	1	0.5	0.2	0	0	1.7	1.5	0.5	0.5	0	5.9	
Jaroš Marek	2	-	1.5	2	0	0	1.5	2	2.2	1	12.2	
Joštiak Matej	1.5	0	1.2	0.3	2.5	0	0	1.5	2.5	0	9.5	
Jurík Norbert	2	-	0.2	0.5	1	2	1.5	1	-	-	8.2	
Jurkasová Linda	2	1	0.5	0.3	0.3	1.7	1.5	2	2.5	2	13.8	
Kabai Tamás	0.5	1	1.5	2	2	1.5	1.5	1.5	1.2	0	12.7	
Kerák Filip	2	1.8	1.5	0	1.6	2	1.5	2	1.5	1	14.9	
Keszeghová Dáša	2	1	1.5	0.5	2	2	1.5	3	2	1	15	
Kniha Nikolaj	1.5	0	0.5	2	0.5	-	-	2	-	0	6.5	
Kočalka Andrej	1.5	0	0.5	0	0	0	1.5	2	1.2	0	6.7	
Kormuth Matej	2	1.8	1.5	1.7	2.5	1	1.5	1.5	1.8	0	15	
Kosec Tomáš	2	0.5	0.3	0.3	0	1.5	1.5	2	0.2	0	8.3	
Kozubaľová Magdaléna	1	0.5	0.5	0.3	0	0	2	0.5	0.5	0.5	5.8	
Krempaská Iveta	1.5	0.5	0	0.3	0	0	1.5	1	0.4	0	5.2	

Pokračovanie na ďalšej strane

Tabuľka 1 – pokračovanie z predošlej strany

študent	1	2	3	4	5	6	7	8	9	10	súčet	
Krivánek Eduard	1	1.5	1.2	2	0	0	2	1.5	0.5	0	9.7	
Kuruc Kristián	1.5	0.5	0	0.3	-	-	1.5	1.5	0.2	0	5.5	
Lomen Martin	2	1	0.2	0.3	-	-	1.5	-	-	-	5	
Mackovič Matúš	1.5	0	0	2	1.2	0	1.5	1	0.2	1	8.4	
Magát Matej	0.8	1	1.5	2	0.8	0	1.5	2	0.5	1	11.1	
Malý Maroš	1	1	1.2	0	0	0	1.5	0.5	1.2	1	7.4	
Maťko Tomáš	1	0.8	1.5	0	0	1	1.5	1	1	-	7.8	
Melišík Juraj	1.5	0.5	0.5	2	0	0	1.5	0.5	0.2	0	6.7	
Mészáros Richard	1	0	1.5	0	0	0	1.5	0.5	0.7	0	5.2	
Mizerík Jozef	2	1	1.5	2	-	2	1.5	2	2.2	0	14.2	
Miženková Veronika	2	1	1.2	1	0	0	1.5	1.5	0.2	0	8.4	
Nemsilajová Ivana	1.5	1	0.2	2	0	2	1.5	1	1.5	0.5	11.2	
Oravcová Jana	1.4	1	0.5	0	1	0	1.5	1.5	1.2	0	8.1	
Orság Nicolas	1	-	1.5	2	-	2	1.5	1	-	-	9	
Pukkai Krisztián	1	0.8	0.5	1	-	1.5	1.5	0.5	0.7	0	7.5	
Slimák Martin	1.5	-	1	2	0	0	1.5	2	1.2	0	9.2	
Smiešna Andrea	1.5	1	0.3	0.3	1.5	0	1.5	2	1.2	0	9.3	
Sternmüller Robert	1.5	0.5	0.3	2	0	0	0	1	0.7	0	6	
Švehlík Ondrej	1.5	0.8	1	2	0.5	1.7	1.5	1	1.5	0	11.5	
Švorc Jakub	1.5	1	1	0.3	-	0	1.5	1	1.5	0	7.8	
Tamáš Peter	2	-	1.5	2	0	2	1.5	0.5	-	-	9.5	
Valentovič Adam	1.5	-	1.5	1.2	0	0.7	0.3	1	1	-	7.2	
Veselá Martina	0	0.5	-	1.8	1.3	0	1.5	1	0.2	0	6.3	
Vetrák Juraj	1.5	1	1.2	0.3	1.7	1.5	0	1.5	2.2	0	10.9	
Vrábelová Ivana	0	0.5	0	0.3	0	0	1.5	0.5	0.3	0	3.1	
Zajíc Áron	1.5	1.8	0.8	1.5	2.3	2	1.5	0	1.2	2	14.6	
Ziman Daniel	-	0	0	-	-	0	1.5	0	0	0	1.5	
Zrubec Michal	1.5	1	1	0	0	0	1.5	0.5	1.2	0	6.7	

37.2 Výsledky záverečného testu

študent	1	2	3	4	5	6	7	8	9	10	súčet	
Babál Šimon	0	0.5	-	2	3	1.5	2	1.5	0.5	1	12	
Balintová Iveta	2	-	3	1.5	3	1	2.5	2	4	1	20	
Baluch Miroslav	0.5	1	0.5	0.5	3	1.3	1.5	1.5	2	1	12.8	
Baráth Kristián												
Bartoš Jakub	2.5	1.3	3	2	2	1.5	4	2.5	4	0	22.8	
Benková Veronika	0.5	1	0	0	2	1.3	1	1	0.5	1	8.3	
Bilichenko Oleksiy	0	2.5	0	-	1.8	0.8	0	1	4	-	10.1	
Bíly Erik	2.5	2.5	0.5	2	2.6	1.3	2	2.5	3.9	3	22.8	
Budinský Martin	0.5	1	-	0.5	3	-	0	0	0.2	0	5.2	
Csitáriová Radoslava	0	0	0	0.5	2.7	-	1	0	1	-	5.2	
Čapkovič Denis	2.5	2.5	3	2.5	2.7	1.5	2.5	4	3	4	25	
Čech Daniel	0	1	0	0.5	2	0.3	1.5	0.5	0	-	5.8	
Dojčan Nicolas	0.5	0	0	-	0.5	-	0	-	0	-	1	
Dominik Richard	0	1	2.5	2	3	1.2	2	0	0.5	1	13.2	
Dóša Barnabás	0	1	0	0	1	1.3	-	-	-	0	3.3	
Dráb Matej	2.5	1	1	1	2	1	2	1.5	1	1.5	14.5	
Drastich Šimon	0	2.5	-	1	2.2	0.8	0	1.5	0	1	9	
Drgoňová Veronika	0.5	1	3	0.5	3	1.5	-	1	1	-	11.5	
Ďurana Marek	2.5	1	3	0	2.7	-	0	1	4	0	14.2	
Ďurica Andrej	0	2.5	3	1.5	1.5	0.3	0	1	0	-	9.8	
Eliaš Filip	0	1.5	0	1	2.7	1.3	0	1	1	1	9.5	
Fekiač David	2	2.5	1	2	-	1.3	1.5	2	2	0	14.3	
Furinda Ľubomír	0	3	0	2	2	1.3	0	1.5	0	0	9.8	
Gál Matúš	0.5	0	2.5	1.5	2.7	1.5	2	1	3	4	18.7	
Gyurcsovicsová Tatiana	1	2.5	0	1	2	0.5	0	2	4	0	13	
Hajná Andrea	2.5	2.5	3	1	3	1.2	3	1.5	4	2	23.7	
Halagačka Jozef	0.5	0	-	1	0	-	-	-	-	-	1.5	
Harnádek Juraj	0	0	0.5	0.5	1.5	1.3	0	0.5	0	0	4.3	
Horecká Rebeka	0.5	1	3	2	2.7	1.5	1	1.5	-	0.5	13.7	
Horníková Nikola	2.5	2	3	1.5	2.5	1.5	2	3	1	1	20	
Hrtánek Viktor	2	2.5	-	1.5	-	1.3	1	2	1	0	11.3	
Chorvatovič Michal	0	2	0	1.5	1.7	1.3	0	1	2.9	1.5	11.9	
Jaroš Marek	2.5	2.5	3	2.5	2.5	1.5	2	2.5	-	4	23	
Joštiak Matej	0	0	0	3	1.2	-	0	2	3.5	0	9.7	
Jurík Norbert	-	2.5	2	1.5	3	1.5	0	2	4	1	17.5	
Jurkasová Linda	0	2.5	2	2.5	1.5	1.5	2	3	4	1.5	20.5	
Kabai Tamás	2.5	1	3	1.5	1.5	1.3	2	1.3	3	1	18.1	
Kerák Filip	2.5	2.5	3	2.5	2.7	1.5	1.5	3	4	4	25	
Keszeghová Dáša	2.5	2.5	3	2.5	3	1.5	2.5	3.5	4	4	25	
Kniha Nikolaj	0	-	0.5	2.5	3	1.3	3	3	4	0	17.3	
Kočalka Andrej	0.5	2	1	0.5	3	1.2	1	2	0	1	12.2	
Kormuth Matej	2.5	2.5	1.5	2.5	3	1.3	2	3	4	1	23.3	
Kosec Tomáš	-	1	2.5	0.5	2.6	1.5	0.5	1.5	2	1	13.1	
Kozubaľová Magdaléna	0	0	0	0.5	1.2	0.5	0	0.7	0.2	1	4.1	
Krempaská Iveta	0	2.5	0	1	2.5	-	0	1	0	0	7	

Pokračovanie na ďalšej strane

Tabuľka 2 – pokračovanie z predošlej strany

študent	1	2	3	4	5	6	7	8	9	10	súčet	
Krivánek Eduard	2.5	2.5	3	1.5	2.7	1.2	0	1	0	1	15.4	
Kuruc Kristián	0	1	2	1	1.7	1.3	1	1	0	0	9	
Lomen Martin	0	1	0	0.5	0	0	0	0	-	0	1.5	
Mackovič Matúš	0	1	2.5	1	2	0.5	1	1.5	2	0	11.5	
Magát Matej	2.5	1.2	2	3	2.9	1.2	1	1	2	4	20.8	
Malý Maroš	-	-	3	1	3	1.3	1	2.5	1	1	13.8	
Maťko Tomáš	0	2.5	2	0	2.7	1.5	1	1.5	0	0	11.2	
Mališík Juraj	0	1	0	0	1.5	1	1	0.5	3	0.5	8.5	
Mészáros Richard	0	2.5	0	1.5	1.7	1.3	1	1.3	0	1	10.3	
Mizerík Jozef	1	2.5	1	2.5	3	1.5	1	1.8	4	2	20.3	
Miženková Veronika	0	2.5	0.5	0.5	0	0	0	1	-	0	4.5	
Nemsilajová Ivana	1	2.5	3	0.5	2.7	1.5	1	2.5	2	1	17.7	
Oravcová Jana	1	2	3	1.5	2.5	1.3	2.5	2	3	1	19.8	
Orság Nicolas	2	0	3	1	2	0.8	0	-	-	0	8.8	
Pukkai Krisztián	0.5	1	3	1	1.5	0.5	0	0.5	1	1	10	
Slimák Martin	2.5	1	0	1.5	2	1.1	0	0.5	2.5	1	12.1	
Smiešna Andrea	2.5	1.5	1.5	2.5	3	1.5	1.5	3	0	2	19	
Sternmüller Robert	0	1	2.5	1.5	2.7	0.7	1	3	1	2.5	15.9	
Švehlík Ondrej	2	2.5	3	1.5	1.5	1.5	0	2.2	4	1	19.2	
Švorc Jakub	-	1	0	1.5	3	1.5	0	1.5	0	0	8.5	
Tamáš Peter	0	2.5	1	0.5	2.5	1	0	1.5	0	-	9	
Valentovič Adam	2.5	2	-	0.5	3	1.5	-	-	3.5	0	13	
Veselá Martina	2.5	2.5	3	1.5	3	1.5	0	2	3	0	19	
Vetrák Juraj	0.5	1	3	0.5	3	1.5	1	3	4	2.5	20	
Vrábelová Ivana	0	0.5	0	1	1.5	1	1	0	0.2	1	6.2	
Zajíc Áron	2.5	2.5	3	1.5	2.5	1.5	0	2.5	0	3.9	19.9	
Ziman Daniel	2	0	0	-	0.5	-	-	0	-	0	2.5	
Zrubec Michal	1	0.5	0.5	2.5	0.7	0.5	0	0	-	1	6.7	

37.3 Výsledky záverečného testu

študent	1	2	3	4	5	6	7	8	9	10	súčet	
Babál Šimon	2	6	5	5.5	2.5	3	2.5	2.5	3.5	1	33.5	
Balintová Iveta	2	6	5	0	5	3	3	3.5	4	0	31.5	
Baluch Miroslav	2	4	4.5	0	3	3	3	4	3	0	26.5	
Bartoš Jakub	0.2	6	4	4	5	2	2.7	4.5	4	4.5	36.9	
Benková Veronika	0	6	1.5	0	0	3	0.5	5	3	0	19	
Bilichenko Oleksiy	2	4	3	4.5	0	1.2	3	5	3	0	25.7	
Bíly Erik	0	5	5	6	5	3	3	5	2.5	-	34.5	
Csitáriová Radoslava	2	2	0	-	0	3	2	0	2.8	-	11.8	
Čapkovič Denis	2	6	5	6	5	3	3	4.5	4	0	38.5	
Dominik Richard	1.5	3	1.5	0	0	3	1.5	2	0.5	1	14	
Dráb Matej	1	5	0	-	5	2.8	3	4.5	4	3	28.3	
Drastich Šimon	2	5	5	0	0	3	3	5	3	1.5	27.5	
Drgoňová Veronika	1.5	2	5	-	5	3	-	5	3	0	24.5	
Ďurana Marek	1.8	6	5	0	0	1.3	0.5	0	4	0.5	19.1	
Eliaš Filip	2	5	5	0	5	3	3	2.5	4	0.5	30	
Gál Matúš	1.8	6	5	6	5	3	3	2.5	4	-	36.3	
Gyurcsovicsová Tatiana	2	4	3.5	0	5	3	2	4.5	4	1	29	
Hajná Andrea	2	6	5	6	5	3	2.7	5	3	5	40	
Harnádek Juraj	1.5	0	0	-	-	3	0.5	0	3	-	8	
Horecká Anna Rebeka	1.8	4	3.5	-	0	3	1	-	4	-	17.3	
Horníková Nikola	1.8	4	1.5	0	5	3	2.7	4.5	3	0	25.5	
Chorvatovič Michal	1.8	0	5	0	3	3	3	4	3.5	0	23.3	
Jaroš Marek	2	6	5	6	5	3	3	4.5	1	-	35.5	
Jurík Norbert	2	5	5	-	5	3	3	2.5	4	4	33.5	
Jurkasová Linda	2	6	5	3	5	3	3	4	4	6	40	
Kabai Tamás	1.5	6	0	0	5	3	3	2.5	4	0.5	25.5	
Kerák Filip	2	5	5	4.5	5	3	3	5	4	4	40	
Keszeghová Dáša	2	6	5	3	5	3	3	4.5	4	6	40	
Kniha Nikolaj	1.5	5	5	-	5	3	2.4	3.5	3.5	0	28.9	
Kočalka Andrej	0.8	5	5	0	5	3	1.5	2	3.5	0	25.8	
Kormuth Matej	2	6	5	6	5	3	2.7	2.5	4	6	40	
Kosec Tomáš	1	4.5	0	0	0	3	2.5	2.5	3	0	16.5	
Kozubaľová Magdaléna	0	2	1.5	2	0	3	1.5	2.5	0.5	0	13	
Krempaská Iveta												
Krivánek Eduard	2	5	5	0	0	3	3	3	4	6	31	
Laho Matúš	2	4.5	1.5	-	5	3	2.7	3.5	4	0	26.2	
Mackovič Matúš	1.8	2.5	4	0	0	3	3	2.5	4	0	20.8	
Magát Matej	2	5.8	5	4	5	3	3	3.7	4	0.5	36	
Malý Maroš	2	5	3.5	0	5	3	3	5	4	0	30.5	
Maťko Tomáš	2	6	5	0	0	3	3	4.5	4	0	27.5	
Mészáros Richard	1.5	5	3.5	3	0	3	3	4.5	4	2	29.5	
Mizerík Jozef	1.8	6	5	3	5	3	3	5	4	4	39.8	
Miženková Veronika	0.5	-	0	-	-	3	3	5	4	-	15.5	
Nemsilajová Ivana	2	5	5	0	5	3	3	5	3	3	34	

Pokračovanie na ďalšej strane

Tabuľka 3 – pokračovanie z predošlej strany

študent	1	2	3	4	5	6	7	8	9	10	súčet	
Oravcová Jana	2	6	3.5	1.5	5	3	2.4	4.5	4	3	34.9	
Orság Nicolas	0.2	6	5	-	5	3	3	5	4	-	31.2	
Slimák Martin	2	5.8	5	-	5	3	3	4.5	3	1	32.3	
Smiešna Andrea	0	4	4	2	5	3	2	3	3.5	0	26.5	
Sternmüller Robert	2	6	5	0	0	3	2.7	5	4	0	27.7	
Švehlík Ondrej	2	5	5	5	5	3	3	4.5	4	6	40	
Tamáš Peter	1	4	1.5	-	5	3	3	2	-	-	19.5	
Tomana František	1.8	5	3.5	0	4	3	2.7	1.5	4	-	25.5	
Valentovič Adam	2	6	3.5	6	5	3	2.7	4.5	4	0	36.7	
Veselá Martina	2	5	1.5	0	1	3	3	2.5	0.5	0	18.5	
Vetrák Juraj	2	6	4.5	3	0	3	3	3	4	6	34.5	
Zajíc Bálint Áron	1.5	5.5	5	6	5	3	3	5	3	0	37	
Zrubec Michal	0	5	0	6	5	3	3	5	0	-	27	

37.4 Semestrálny projekt v zimnom semestri

37.4.1 Zadanie

Napíšte program, ktorý umožní jednému alebo viacerým hráčom zahrať sa nejakú (najlepšie doskovú) hru. Váš program vykreslí hraciu plochu, umožní hráčom ťahať, bude pritom kontrolovať prípustnosť ťahov a zároveň bude kontrolovať, či niektorý z hráčov nevyhral, resp. neprehral.

37.4.2 Témy

Vyberte si jednu z týchto hier, alebo sa nimi inšpirujte:

- hracie dosky s figúrkami:
  - dáma, reversi, šach, mlyn, človeče nehnevaj sa, ludo, hex, bridge it, monopoly, go, lines of action, backgammon, dots and boxes
- kartičkové hry:
  - pexeso, kartové hry, domino, scrabble, mahjong, black, trax, blokus, posúvací puzzle
- hry pre jedného hráča:
  - anglický solitaire, pasians, tangram, míny, sokoban, sudoku, puzzle, light-bot, kravička
- edukačné hry:
  - ovládanie robota Karla, programovanie Baltazára, programovanie Logo (kreslenie korytnačkou)

Môžete sa inšpirovať online hrami na internete, napr.

- logické hry
- logicke hry
- online hry
- detske hry
- logické hry pre deti

Pri výbere hry myslite na to, že by sa mala ovládať najmä myšou (klikanie, ťahanie), mala by obsahovať aspoň jeden animovaný prvok (striedanie niekoľkých fáz animácie, napr. postavičky hráča).

Je vhodné si tému projektu nechať schváliť cvičiacimi, aby ste mali istotu, že to čo idete programovať je dobrá téma. Samozrejme, že tému si môžete hocikedy zmeniť.

Pravidlá vybranej hry si môžete prispôsobiť alebo aj dosť radikálne zmeniť.

37.4.3 Požiadavky

Váš program musí spĺňať nasledovné požiadavky:

- program musí byť realizovaný ako inštancia vašej triedy `Program`, napr.

```
import tkinter

class Program:
    def __init__(self):
        ...
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        tkinter.mainloop()

    ...

Program()
```

- okrem tejto triedy definujte aspoň jednu ďalšiu triedu pre popis nejakej časti hry, napr. figúrky, kartičky, políčka, hráč, nepriateľ, ... túto triedu využijete v hlavnom programe (tieto pomocné triedy môžete definovať buď v tom istom module alebo v ďalšom)
- okrem definícií týchto tried a inštancie triedy `Program`, nepoužívajte **žiadne globálne premenné ani funkcie**

Ďalej musí platiť:

- grafika je realizovaná modulom `tkinter`:
  - na ovládanie hry použite udalosti myši (klikanie, ťahanie)
  - využite časovač (pomocou `after()`)
  - môžete využiť aj klávesnicu (ale hlavné ovládanie hry by malo byť ťahaním a klikaním myšou)
  - aspoň nejaká časť grafiky musí byť realizovaná obrázkami (.png, .gif), pričom musíte simulovať aspoň jednu animáciu (striedanie niekoľkých obrázkov v časovači) počas hry (nestačí na napr. iba „úvodnej obrazovke“)
  - môžete využiť aj také funkcie a vlastnosti `tkinter`, ktoré sme sa neučili
  - pre projekt môžete využiť ľubovoľnú grafiku z internetu alebo z nejakej hry (ak neplanujete nejako šíriť túto hru, inak by ste mali dbať na legálne použitie grafiky)
- program by mal čítať aj zapisovať aspoň jeden textový súbor, napr. rozohratú partiu, úvodné nastavenie, rôzne nastavenia hry, tabuľka najlepších výsledkov a pod.
- môžete ešte použiť aj moduly `random`, `math` a `json`

Dopredu si dohodnote s cvičiacimi, ak by mala vaša téma problém s niektorou z týchto požiadaviek.

37.4.4 Hodnotenie

Projekt budú hodnotiť vaši cvičiaci, ale musíte ho odovzdať na úlohový server L.I.S.T., pričom

1. za splnenie **všetkých požiadaviek** bude základných 5 bodov (nedáva sa menej ako 5 bodov)
2. za umelecký dojem, pohodlné ovládanie, nové zaujímavé prvky, použitie nejakého náročnejšie algoritmu a pod. do 5 bodov - tieto body sa budú pridávať k základným 5 bodom
3. projekt by ste mali realizovať samostatne, bez cudzej pomoci, samozrejme, že môžete rôzne problémy konzultovať s vyučujúcimi ale aj so svojimi kolegami

Takto získané body sa pripočítavajú k bodom ku skúške len vtedy, ak ich získate ešte **pred samotným termínom skúšky**.

37.5 Semestrálny projekt

37.5.1 Zadanie

Napíšte program, ktorý realizuje jednu z ponúkaných oblastí:

- ľubovoľná hra, v ktorej je protihráčom počítač - počítač by mal hrať podľa pravidiel a aspoň trochu inteligentne
- vizualizácia dátovej štruktúry - vytváranie a modifikovanie niektorej dátovej štruktúry z letného semestra
- editor grafických objektov - umožní skladať z nejakej množiny objektov rôzne obrazy, ukladať ich do súboru, resp. prečítať
- edukačný softvér - program, ktorý trénuje nejaké zručnosti z niektorého predmetu, napr. matematika, informatika, fyzika, angličtina, ... - cieľom nie je testovať žiakov
- ďalšie témy podobnej obťažnosti, ktoré schváli cvičiaci

Môžete využiť aj vlastný projekt zo zimného semestra.

37.5.2 Požiadavky

Váš program musí spĺňať nasledovné požiadavky:

- program by mal byť grafický, najlepšie s využitím `tkinter`
- okrem hlavnej triedy definujte aspoň jednu ďalšiu triedu pre popis nejakej časti
- okrem definícií týchto tried a inštancie hlavnej triedy nepoužívajte žiadne globálne premenné ani funkcie
- program by sa mal dať ovládať najmä klikaním a ťahaním myšou (môžete využiť aj klávesnicu)
- program by mal čítať aj zapisovať aspoň jeden textový súbor

37.5.3 Hodnotenie

Projekt budú hodnotiť vaši cvičiaci, ale musíte ho odovzdať na úlohový server L.I.S.T., pričom

1. za splnenie **všetkých požiadaviek** bude základných 5 bodov
2. za umelecký dojem, pohodlné ovládanie, nové zaujímavé prvky, použitie nejakého náročnejšie algoritmu a pod. do 5 bodov - tieto body sa budú pridávať k základným 5 bodom

Takto získané body sa pripočítavajú k bodom ku skúške len vtedy, ak ich získate ešte **pred samotným termínom skúšky**.

37.6 Priebežný test z Programovania (1) 2014/2015

1. Zistite, čo vypíše tento program:

```
def urob(hodnota):
    prem1, prem2 = 0, 0
    while hodnota > 0:
        if hodnota % 2 == 0:
            prem1 += 1
        else:
            prem2 -= 1
        hodnota //= 2
    print(prem1, prem2)

>>> urob(13)
>>> urob(99)
```

2. Nasledovné priradenia vytvoria 7 prvkový zoznam `zoz`, ktorý obsahuje len čísla a n-tice. Zapíšte do tabuľky hodnoty prvkov tohto zoznamu:

```
a = tuple(range(10,30))
zoz = [a[3], a[8:11], a[:1], a[-3:-1], a[4:14:4], a[::5], a[7:3:-1]]
```

zoz[0]	zoz[1]	zoz[2]	zoz[3]	zoz[4]	zoz[5]	zoz[6]

3. Funkcia `dokonale(n)` zisťuje, či zadané celé číslo je dokonalé (rovná sa súčtu svojich deliteľov). Doplňte chýbajúce časti tak, aby výsledkom funkcie bola buď hodnota `True` alebo `False`. Nedopisujte žiadne ďalšie príkazy.

```
def dokonale(n):
    sucet = 0
    for i in _____________:
        if ____________________:
            sucet += i
    return ________________________
```

4. Funkcia `urob(meno, zoznam)` vyrobí textový súbor, v ktorom budú všetky prvky zadaného zoznamu celých čísel, ale v zmenenom poradí: najprv idú všetky s nadpriemernou hodnotou a až za nimi všetky zvyšné. Funkcia teda otvorí súbor na zápis, do každého riadku zapíše jedno z čísel z daného zoznamu:

```
def urob(meno, zoznam):
    t = open(_____________________________)
    priemer = ____________________________________
    for i in True, False:
        for p in zoznam:
            if ____________________ or ____________________:
                print(_________________)
    t.close()

>>> urob('cisla.txt', [2, 9, 3, 8, 4, 1, 7, 5])
... vytvorí súbor s číslami: 9 8 7 5 2 3 4 1
```

   Doplňte chýbajúce časti príkazov. Nevkladajte žiadne ďalšie príkazy.

5. Funkcia `urob(n)` vypisuje hviezdičky do viacerých riadkov:

```
def urob(n):
    for i in range(n):
        print(' '*i, '*'*(2*(n-i)-1))
```

Zistite

(a) koľko hviezdičiek sa vypíše, ak zavoláme `urob(4)`

(b) pre aké najmenšie `n` sa vypíše aspoň `555` hviezdičiek

6. Funkcia `zoznam_cisel(retazec)` dostáva ako parameter znakový reťazec, ktorý obsahuje celé čísla oddelené medzerou. Funkcia z neho vytvorí zoznam čísel (využite metódu `find()` a funkciu `int()`). Funkcia nič nevypisuje, ale vráti tento zoznam ako výsledok funkcie. Vo funkcii je niekoľko chýb, kvôli ktorým nebeží správne. Opravte ich! Nepridávajte žiadne ďalšie príkazy.

```
def zoznam_cisel(retazec):
    zoznam = []
    retazec += '\n'
    while retazec == '':
        i = zoznam.find(' ')
        retazec.append(int(retazec[i+1:]))
        retazec = retazec[:i]
    print(zoznam)

>>> a = zoznam_cisel('19 173 7 97 1001')
>>> print(a)
[19, 173, 7, 97, 1001]
```

7. Funkcia `urob(utvary)` nakreslí podľa zadaného zoznamu niekoľko farebných štvorcov a kruhov:

```
def urob(utvary):
    canvas = tkinter.Canvas()
    canvas.pack()
    for i in utvary:
        u = 0
        for j in range(i[0]):
            x, y = random.randrange(50, 200), random.randrange(50, 200)
            sur = (x-5, y-5, x+5, y+5)
            if u != 0:
                canvas.create_rectangle(sur, fill=i[1])
            else:
                canvas.create_oval(sur, fill=i[1])
            u = (u + 1) % 3

>>> urob([(10, 'red'), (2, 'blue'), (5, 'red'), (8, 'blue'), (3, 'red')])
```

Zistite, aký počet červených štvorcov a modrých kruhov sa nakreslí pre konkrétne volanie.

- počet červených štvorcov =
- počet modrých kruhov =

8. Funkcia `kopiruj(vstup, vystup)` prekopíruje obsah súboru s menom `vstup` do súboru s menom `vystup`, pričom nekopíruje riadky, ktoré obsahujú znak `'#'`:

```
def kopiruj(vstup, vystup):
    with open(vstup, 'w') = t2:
        with open(vystup) = t1:
            riadok = t1.input()
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
            while riadok:
                if '#' in not riadok:
                    t2.print(riadok)
                riadok = t1.input()
```

V programe je niekoľko chýb. Opravte ich, ale nepridávajte žiadne ďalšie príkazy.

9. Funkcia `uprac(zoznam)` presťahuje všetky prvky s hodnotou číslo 0 na začiatok zoznamu, ostatné prvky posunie vpravo. Funkcia nevracia žiadnu hodnotu, len modifikuje vstupný zoznam. Doplňte chýbajúce príkazy:

```
def uprac(zoznam):
    pocet0 = 0
    for i in reversed(range(len(zoznam))):
        if zoznam[i] == 0:
            pocet0 += 1
        else:
            ____________________________
    while pocet0:
        ____________________________
        ____________________________

>>> a = [0, 5, 0, 0, 2, '0', 7, 0]
>>> uprac(a)
>>> a
[0, 0, 0, 0, 5, 2, '0', 7]
```

10. Funkcia `rozdel(zoznam)` rozdelí zoznam celých čísel na dva nové zoznamy: všetky prvky s párnymi hodnotami budú v prvom výslednom zozname a všetky zvyšné v druhom. Funkcia vráti výsledok ako dvojprvkovú n-ticu (typ `tuple`) práve vytvorených zoznamov:

```
def rozdel(zoznam):
    prvy, druhy = [], []
    for prvok in zoznam:
        if prvok % 2:
            druhy.append(prvok)
        else:
            prvy.append(prvok)
    return prvy, druhy

>>> a, b = rozdel([1, -2, 3, 4, 0, -5])
>>> print(a)
[-2, 4, 0]
>>> print(b)
[1, 3, -5]
```

Opravte túto funkciu tak, aby namiesto dvoch zoznamov vracala dve n-tice (`tuple`). Môžete prepísať ľubovoľné riadky funkcie okrem záverečného `return`.

37.7 Priebežný test z Programovania (1) 2015/2016

1. Druhý parameter funkcie `xy()` je nejaká n-tica celých čísel:

```
def xy(ret, ix, znaky):
    p = list(ret)
    for i in range(len(ix)):
        p[ix[i]] = znaky[0]
        znaky = znaky[1:]
    return ''.join(p)

>>> xy('najpython', (_____), 'anopye')
'neopytany'
```

   Zistite, aký bol druhý parameter pri volaní tejto funkcie.

2. Príkaz `input()` sme testovali v príkazovom režime:

```
>>> x = input(input(input('zadaj:')))
zadaj:preco:
preco:lebo:
lebo:nic:
>>>
```

   Zistite, čo sa takto priradilo do premennej `x`.

3. Funkcia `generuj()` vytvára nejaký znakový reťazec:

```
def generuj(n, retazec):
    for i in range(n):
        retazec = 'a\nb' + retazec[1:-1] + 'x\ny' + retazec[2:-2] + 'b\na'
    return retazec

print(generuj(2, 'xyz'))
```

   Po zavolaní tejto funkcie sa vypíše niekoľko riadkov textu. Zistite, koľko riadkov sa vypíše a koľkokrát sa v nich vyskytnú písmená `'a'` a `'b'`.

4. Funkciu `kresli()` sme zavolali s parametrom typu zoznam dvojíc:

```
def kresli(bodky):
    for x, y in bodky:
        if x < y:
            farba = 'red'
        elif x+y < 100:
            farba = 'blue'
        else:
            farba = 'yellow'
        canvas.create_oval(x-2, y-2, x+2, y+2, fill=farba, outline='')

kresli([(80, 70), (30, 90), (80, 50), (70, 40), (70, 50), (50, 10), (10, 10),
        (40, 90), (40, 10), (80, 20), (60, 20), (50, 60), (40, 50), (20, 40),
        (20, 70), (90, 80), (70, 90), (50, 20), (30, 70)])
```

   Funkcia kreslí farebné bodky. Zistite, koľko bodiek bolo červených, koľko modrých a koľko žltých?

5. Funkcia `zapis()` vytvára textový súbor:

```
def zapis(subor, n):
    with open(subor, 'w') as vystup:
        j = 3
        for i in range(n):
            if i < j:
                ret = ' '
            else:
                ret = '\n'
                j += i
            print(i, i*i, end=ret, file=vystup)
        print(file=vystup)
```

Opravte túto funkciu tak, aby pracovala presne rovnako, ale nevolala štandardnú funkciu `print()`.

6. Funkcia `ntica()` vytvára nejakú n-ticu dvojíc celých čísel:

```
def ntica(a, b):
    vysl = ((a ,b),)
    while b > 0:
        a, b = b, a%b
        vysl += ((a, b),)
    return vysl
```

Takto vytváraná postupnosť dvojíc čísel má nejaký súvis s výpočtom najväčšieho spoločného deliteľa dvoch celých čísel. Dopíšte:

```
def nsd(a, b):
    x = ntica(a, b)
    return ________________________
```

7. Funkcia `urob()` s tromi parametrami spracuje nejaké tri postupnosti (napr. sú to tri polia, n-tice alebo reťazce):

```
def urob(a, b, c):
    return c[-1:len(c)] + b[1:] + a[:] + c[:-1] + b[0:1]
```

Do premennej `p` sme priradili nejakú hodnotu a zavolali s ňou funkciu `urob()` takto:

```
>>> p = ____________________________
>>> urob(p[2:7], p[4:], p[:5])
(4, 9, 6, 11, 10, 5, 8, 3, 4, 9, 6, 7, 12, 8, 3, 4)
```

Zistite, s akou hodnotou premennej `p` sme zavolali túto funkciu.

8. Dopíšte funkciu `porovnaj()`, ktorá porovná dva dátumy (dátum je trojica celých čísel (deň, mesiac, rok)) a vráti jeden z týchto troch reťazcov: `'mensi'`, `'rovny'`, `'vacsi'`:

```
def porovnaj(datum1, datum2):
    
    
    
    
    
    
>>> porovnaj((26, 10, 2015), (15, 11, 2015))
'mensi'
>>> porovnaj((26, 10, 2015), (15, 11, 2014))
'vacsi'
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
>>> porovnaj((26, 10, 2015), (26, 10, 2015))
'rovny'
```

9. Funkcia `rozsekaj()` vytvorí zo znakového reťazca nticu podreťazcov, pričom oddeľovačom týchto podreťazcov v pôvodnom reťazci je znak bodkočiarka `';'`. Funkcia je podobná štandardnej metóde `split()`, ktorú ale použiť nesmiete:

```
def rozsekaj(retazec):
    vysl = ()
    while retazec:
        i = ';'.index(retazec+';')
        vysl.append(retazec[:i])
        retazec[i+1:] = ''
    return vysl

>>> rozsekaj('a;12;b')
('a', '12', 'b')
```

Vo funkcii sú chyby, opravte ich. Nedopisujte nové príkazy.

10. Funkcia `uprac()` nejako mení obsah zoznamu `pole`:

```
def uprac(pole):
    i = 0
    for j in range(len(pole)):
        if pole[j] < 0:
            pole.insert(i, pole.pop(j))
            i += 1
```

Prerobte túto funkciu tak, aby pracovala nielen pre zoznamy, ale aj pre n-tice. Preto nebude táto funkcia modifikovať svoj parameter, ale bude vytvárať novú nticu a tú vráti ako výsledok funkcie:

```
def uprac(pole):
    i = 0
    pole = tuple(pole)
    for j in range(len(pole)):
        if pole[j] < 0:



            i += 1
    return pole
```

Doplňte chýbajúcu časť funkcie (zvyšok funkcie už nemeňte!).

37.8 Priebežný test z Programovania (1) 2016/2017

1. Čo presne vypíše tento program?

```
for i in 4, 7, 1, 2, 0, 5:
    while i:
        print(end=str(i))
        i -= 1
    print()
```

2. Funkcia `zisti(cislo)` vráti `True` alebo `False` podľa toho, či je dané číslo dokonalé alebo nie , t. j. či platí, že súčet všetkých deliteľov menších ako samotné číslo sa rovná samotnému číslu. V definícii funkcie je jedna chyba. Opravte ju.

```
def zisti(cislo):
    x, y = 0, 1
    while y < cislo and x < cislo:
        if cislo % y == 0:
            x += y
        y += 1
    return x == cislo
```

3. Funkciu `nahrad_samo(text, znak)` v zadanom texte nahradí všetky samohlásky (`'aeiouy'`) zadaným znakom. Doplňte chýbajúce časti programu.

```
def nahrad_samo(text, znak):
    for i in range(len(text)):
        if ____________________________:
            ____________________________________________________
    return text
```

4. Funkcia `sifra(text)` nejako zakóduje zadaný text:

```
def sifra(text, n):
    text = (text + 'x' * n * n)[:n * n]
    vysl = ''
    for k in range(n):
        vysl += text[k::n]
    return vysl
```

   Zistite, ako sa zakódujú nasledovné dva reťazce:

```
sifra('bratislava', 3)
sifra('programujeme v pythone', 5)
```

5. Nasledovný program:

```
with open('subor1.txt') as t1:
    n = int(t1.readline())
    with open('subor2.txt', 'w') as t2:
        for h in range(n):
            for i in range(n):
                p = int(t1.readline())
                if i == 0:
                    s = p
                else:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
                    s += p
                print(p, end=' ', file=t2)
            print(s, file=t2)
```

vytvoril takýto súbor:

```
3 1 7 11
7 3 1 11
1 7 3 11
```

Aký bol obsah vstupného súboru `'subor1.txt'`?

6. Funkcia `sucet(subor)` otvorí zadaný textový súbor (tento súbor obsahuje len celé čísla, aj záporné) a vypočíta súčet všetkých týchto čísel. Opravte všetky chyby, pričom nemeňte ideu algoritmu, t. j. čítanie súboru po znakoch, postupné skladanie pomocného reťazca a pričitovanie reťazca prevedeného na celé čísla.

```
def sucet(meno_suboru):
    vysl = ''
    with open(meno_suboru, 'w') as t:
        ret, znak = '', t.read(1)
        while znak != '':
            if znak in ' \n':
                if ret != '':
                    vysl = int(vysl + ret)
                ret = ''
            elif '0' <= znak <= '9' and znak == '-':
                ret = znak + ret
            znak = t.read(1)
    if ret != '':
        vysl += int(ret)
    return vysl
```

7. Funkcia `rozdel(n_tica)` rozloží prvky vstupnej n-tice na dve n-tice, pričom prvá obsahuje prvky na párnych pozíciách (0., 2., 4., . . .) a druhá na nepárnych (1., 3., 5., . . .). Dopíšte do funkcie dva chýbajúce riadky.

```
def rozdel(n_tica):
    prva = druha = ()
    for i in range(len(n_tica)):
        if i % 2:
            _____________________________
        else:
            _____________________________
    return prva, druha
```

8. Funkcia `rozloz(n_tica)` nejako spracuje prvky vstupnej n-tice a vytvorí z nich inú n-ticu a tú vráti ako výsledok:

```
def rozloz(n_tica):
    vysl = p0 = ()
    for p1 in n_tica:
        for p2 in p1:
            p0 += p2,
            if len(p0) == 2:
                vysl += p0,
                p0 = ()
    if len(p0):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        vysl += p0,
    return vysl
```

Zistite, čo vráti volanie:

```
rozloz(((0,), (), (9, 0, 7), (1, 2, 3)))
```

9. Zoznam súradníc obsahuje dvojice celých čísel. Funkcia `prerob1(zoznam)` prerobí vstupný zoznam tak, že každú dvojicu rozloží na 2 prvky. Funkcia nič nevracia, len mení obsah zoznamu. Dopíšte do funkcie vyznačenú časť.

```
def prerob1(zoznam):
    for i in range(_____________________________________):
        zoznam[i:i+1] = zoznam[i]
```

napr.

```
>>> xy = [(100, 50), (0, -20), (350, 200)]
>>> prerob1(xy)
>>> xy
[100, 50, 0, -20, 350, 200]
```

10. Dopíšte funkciu `prevrat(zoznam)`, tak aby z pôvodného zoznamu vyrobila zoznam s rovnakými hodnotami, ale v opačnom poradí. V dopisovanej časti funkcie použite len metódy `append()` a `pop()` ale nepoužite žiadne `[ ]` zátvorky.

```
def prevrat(zoznam):
    vysl = list()




    return vysl
```

Funkcia nesmie zmeniť obsah pôvodného vstupného zoznamu.

37.9 Záverečný test z Programovania (1) 2014/2015

1. Zadefinujte rekurzívnu funkciu `mocnina(n,k)`, ktorá vypočíta `n**k` pre celé nezáporné `k` len pomocou násobenia a umocňovania na 2 (čo je opäť len násobením so samým sebou):
   - `mocnina(n, 0) = 1`
   - `mocnina(n, k) = mocnina(n, k//2) ** 2` ... pre párne `k`
   - `mocnina(n, k) = mocnina(n, k-1) * n` ... pre nepárne `k`

```
def mocnina(n, k):



    ...
```

2. Zistite, koľko hviezdičiek sa vypíše pre volania `rekurzia(10)` a `rekurzia(13)`?

```
def rekurzia(n):
    if n < 2:
        print('*')
    else:
        rekurzia(n-1)
        rekurzia(n-2)
```

3. V triede `Kniha` si ukladáme informácie o knihách:

```
class Kniha:
    def __init__(self, autor, titul, cena):
        self.zoznam = [autor, titul, cena]

    def autor(self, zmen=None):


        ...
```

   Dopíšte metódu `autor()` tak, aby volanie bez parametrov vrátilo autora knihy a volanie s jedným parametrom zmenilo autora, napr.

```
>>> k = Kniha('Doyle', 'Sherlock Holmes', 11.5)
>>> k.autor()
'Doyle'
>>> k.autor('sir Arthur Conan Doyle')
>>> k.zoznam
['sir Arthur Conan Doyle', 'Sherlock Holmes', 11.5]
```

4. V triede `Kniha` si ukladáme informácie o knihách:

```
class Kniha:
    def __init__(self, autor, titul, cena):
        self.zoznam = [autor, titul, cena]

    def __getitem__(self, co):
        ________________
        return ________________

    def __setitem__(self, co, zmen):
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        ________________
        ____________________ = zmen
```

Dopíšte metódy `__getitem__()` a `__setitem__()` tak, aby pre parameter `co`, ktorý je jeden z reťazcov `'autor'`, `'titul'` alebo `'cena'`, fungovalo:

```
>>> k = Kniha('Doyle', 'Sherlock Holmes', 11.5)
>>> k['titul']
'Sherlock Holmes'
>>> k['cena'] = 8.1
>>> k.zoznam
['Doyle', 'Sherlock Holmes', 8.1]
```

5. Dopíšte funkciu `vyrob(dlzky)`, ktorá vytvorí dvojrozmernú tabuľku celých čísel tak, že ak parameter `dlzky` je zoznam celých čísel, tak tieto označujú dĺžky riadkov vytváranej dvojrozmernej tabuľky (zoznam zoznamov). Počet prvkov zoznamu `dlzky` označuje počet riadkov vytváranej tabuľky. Prvky vytváranej tabuľky pritom postupne zaplňte hodnotami od `1`.

```
def vyrob(dlzky):
    tab = []
    ________________
    for d in dlzky:
        ________________
        ________________
    return tab

>>> zoz = vyrob([3, 2, 4])
>>> vypis(zoz)
1 2 3
4 5
6 7 8 9
```

Funkciu `vypis()` neprogramujte. Pri dopisovaní kódu do funkcie nemusíte dodržať naznačený počet riadkov programu.

6. Textový súbor `'subor.txt'` v každom riadku obsahuje niekoľko slov oddelených medzerami. Nasledovná funkcia by mala vytvoriť dvojrozmernú tabuľku znakových reťazcov, ktorá bude v každom riadku obsahovať ako prvky slová z príslušného riadku súboru:

```
def urob(meno):
    with open(meno) as subor:
        vysledok = []
        while subor:
            riadok = subor.read()
            if riadok:
                return vysledok
            riadok = riadok.strip()
            riadok.append(vysledok)
```

Opravte všetky chyby.

7. Dané sú dva zoznamy `zoz1` a `zoz2`, ktoré majú rovnaký počet prvkov. Vytvorte funkciu, ktorá z takýchto dvoch zoznamov vytvorí a vráti nový slovník (asociatívne pole). V tomto slovníku sú kľúčmi prvky z prvého zoznamu a hodnotami sú príslušné prvky druhého zoznamu:

```
def urob(zoz1, zoz2):



>>> a = urob(['druh', 'vaha', 'vek'], ['slon', 1000, 10])
>>> a
{'druh': 'slon', 'vek': 10, 'vaha': 1000}
```

8. Vytvorili sme zoznam informácií (slovníkov, teda asociatívnych polí) o zvieratách v zoo, napr.:

```
zoo = [{'druh':'slon', 'meno':'Bimbo'}, {'druh':'opica', 'meno':'Milica'}, ...]
```

   Napíšte funkciu `vsetky_mena(zoo)`, ktorá vráti množinu všetkých mien zvierat v `zoo`:

```
def vsetky_mena(zoo):


    ...
```

9. Zistite, čo sa vypíše:

```
>>> zoz1 = [3, 'sedem', 3.14]
>>> zoz2 = ['dog', 'cat', 'mouse', 'duck']
>>> zoz3 = {'sedem': [3]*3, 3.14: zoz2, 'cat': zoz1}
>>> zoz3[zoz3[zoz2[1]][2]][2]
```

10. Dopíšte funkciu `urob(m1, m2, m3)`, ktorá dostáva ako parametre 3 množiny. Výsledkom funkcie bude nová množina, ktorá obsahuje všetky také prvky, ktoré nie sú v `m1`, ale sú buď v `m2` alebo v `m3` (ale nie naraz v oboch).

```
def urob(m1, m2, m3):
    mnozina = _________________
    ___________________
    ___________________
    return mnozina
```

   Pri dopisovaní kódu do funkcie nemusíte dodržať naznačený počet riadkov programu. Použitie, napr.

```
>>> urob({1,3,5,7}, {1,2,3,4,5}, {4,5,6,7})
{2, 6}
>>> urob(set(),{1,2,3,4,5},{4,5,6,7})
{1, 2, 3, 6, 7}
```

37.10 Záverečný test z Programovania (1) 2015/2016

1. Zistite, čo vypočíta daná rekurzívna funkcia:

```
def pocitaj(n):
    if n <= 1:
        return list(range(n + 1))
    pred = pocitaj(n-1)
    return pred + [sum(pred[-2:])]

print(pocitaj(10))
```

2. Máme zadefinovať funkciu `ocisluj(tab)`, ktorá zmení všetky prvky dvojrozmernej tabuľky tak, že ich postupne prechádza po stĺpcoch a čísluje ich celými číslami od 0, pritom riadky tabuľky nemusia mať rovnakú dĺžku, napr.

```
>>> ab = [[1, 1], [1, 1, 1], [1], [1, 1, 1, 1]]
>>> ocisluj(ab)
>>> ab
[[0, 4], [1, 5, 7], [2], [3, 6, 8, 9]]
```

Dopíšte chýbajúce časti funkcie:

```
def ocisluj(tab):
    m = 0
    for r in tab:
        m = ____________________________
    poc = 0
    for j in range(m):
        for i in range(len(tab)):
            if ____________________________:
                tab[i][j] = poc
                poc += 1
```

3. Funkcia `zisti()` pracuje so slovníkom (asociatívnym poľom):

```
def zisti(zoznam):
    slovnik, neviem = {}, None
    for prvok in zoznam:
        slovnik[prvok] = slovnik.get(prvok, 0) + 1
        if neviem is None or slovnik[prvok] > slovnik[neviem]:
            neviem = prvok
    return neviem
```

Zistite, čo sa vypíše pre volania:

```
print(zisti([1,2,3,4,5,4,3,2,3,2,4,6,4]))
print(zisti(list('programujem v pythone')))
```

4. V triede `Mnozina` je 5 chýb. Opravte ich! Neopravujte kód, ktorý nie je chybný.

```
class Mnozina:
    def __init__(self):
        self.zoznam = []

    def __repr__(self):
        print(tuple(sorted(self.zoznam)))
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def add(self, cislo):
        if cislo in self:
            self.zoznam.append(cislo)

    def discard(self, cislo):
        if cislo in self:
            self.zoznam.pop(cislo)

    def __contains__(self, cislo):
        return self.zoznam.index(cislo) >= 0

    def __len__(self):
        return len(self.zoznam)

    def zjednotenie(self, ina):
        for i in ina.zoznam:
            self.append(i)
```

5. Dopíšte funkciu `nechaj_float(zoz)`, ktorá v zozname znakových reťazcov ponechá len tie, ktoré reprezentujú desatinné čísla (dajú sa previesť konverznou funkciou `float()`). Dopisuje len medzi riadky `while` a `del`:

```
def nechaj_float(zoz):
    i = 0
    while i < len(zoz):




        del zoz[i]

>>> zoznam = ['3..', '1e1', '7', '' , '1e1e', '.7.']
>>> nechaj_float(zoznam)
>>> zoznam
['1e1', '7']
```

6. **a)** Na začiatku bol zásobník prázdny. Potom sme vykonali niekoľko operácií `push` a `pop`. Zistite, čo sa vypíše po spustení:

```
print('pre zasobnik:')
for x in 7, 3, 5, 11, 3, 8, 4:
    push(x)
for x in range(6):
    print(pop(), pop(), end=' ')
    push(x)
print(pop())
```

b) Na začiatku bol rad prázdny. Potom sme vykonali niekoľko operácií `enqueue` a `dequeue`. Zistite, čo sa vypíše po spustení:

```
print('pre rad:')
for x in 7, 3, 5, 11, 3, 8, 4:
    enqueue(x)
for x in range(6):
    print(dequeue(), dequeue(), end=' ')
    enqueue(x)
print(dequeue())
```

7. Nasledovný príklad definície triedy demonštruje nie najlepšie využitie operátora indexovania. Zistite ale, čo sa vypíše:

```
class Trieda:
    x = 7
    def __getitem__(self, i):
        return sum(range(self.x - i))

    def __setitem__(self, i, y):
        self.x = 2 * i + y

t = Trieda()
print(t[4], t[5])
t[4] = 1
print(t[4], t[5])
```

8. Preveďte infixový výraz do prefixu a aj do postfixu:

```
3*(4+5)-6*7/(8+6)+(2/3)/(4/5)
```

prefix:

postfix:

9. Funkcia `vsetky()` nejako spracováva zoznam množín:

```
def vsetky(zoznam_mnozin):
    vysl, b = set(), True
    for m in zoznam_mnozin:
        if b:
            vysl |= m
        else:
            vysl -= m
        b = not b
    return vysl
```

Zistite, čo treba dosadiť do premennej `x`, aby sme dostali tento výsledok:

```
>>> x = _________________________
>>> vsetky([x, set(range(3, 10, 2)), set(range(5, 15, 3))])
{2, 5, 6, 8, 11, 14}
```

10. Napíšte program pre **Turingov stroj**, ktorý bude akceptovať vstup na páske len vtedy, keď je na nej číslo v dvojkovej sústave, pričom toto číslo je nepárne. Štartový stav má meno `'0'` a koncový `'1'`:

```
t = t.Turing('''









''')
for slovo in '', '11a', '0101010101', '11110', '1', '0':
    t.znovu(slovo)
    print(repr(slovo), t.rob(False))
```

Vypíše:

```
'' False
'11a' False
'0101010101' True
'11110' False
'1' True
'0' False
```

37.11 Záverečný test z Programovania (1) 2016/2017

1. Čo presne vypíše tento program?

```
def rek(n):
    vysl = 1
    for i in range(1, n):
        vysl += rek(i)
    return vysl

for i in range(6):
    print(i, rek(i))
```

2. Zmenili sme metódu `fd()` pre korytnačku:

```
class MojaTurtle(turtle.Turtle):
    def fd(self, d):
        if d <= 10:
            super().fd(d)
        else:
            self.fd(d // 2 - 5)
            self.pu()
            self.fd(10)
            self.pd()
            self.fd(d - d // 2 - 5)
```

   Zistite, aká je prejdená dĺžka čiar so spusteným perom pre tieto volania:

   (a) `MojaTurtle().fd(30)`

   (b) `MojaTurtle().fd(40)`

   (c) `MojaTurtle().fd(50)`

   Uvedomte si, že táto funkcia je rekurzívna.

3. Zistite, čo vráti táto funkcia:

```
def test(zoznam):
    vysl, n = 0, len(zoznam)
    for i in range(n):
        for j in range(n):
            vysl += abs(zoznam[i][j] - zoznam[j][i])
    return vysl
```

   pre volanie:

```
>>> test([[1, 2, 3], [2, 1, 0], [3, 2, 1]])
```

4. Zadefinovali sme triedu `Zoznam`, pomocou ktorej si vieme udržiavať zoznam svojich záväzkov. Trieda obsahuje tieto metódy:

   - `pridaj(prvok)`, ak sa tam takýto záväzok ešte nenachádza, pridá ho na koniec
   - `vyhod(prvok)`, ak sa tam takýto záväzok nachádzal, vyhodí ho
   - `__contains__(prvok)` vráti `True` alebo `False` podľa toho, či sa tam tento záväzok nachádzal
   - `vypis()` vypíše všetky záväzky v tvare *záväzok, záväzok, záväzok*

   Opravte všetky chyby (to, čo nie je chyba, neopravujte):

```
class Zoznam:
    zoz = []
    def pridaj(self, prvok):
        if prvok in self:
            zoz.insert(prvok)

    def vypis(self):
        print(*self.zoz, sep=', ')

    def __contains__(self, prvok):
        return prvok in zoz

    def vyhod(self, prvok):
        if prvok not in self:
            zoz.delete(prvok)
```

5. V utriedenom zozname máme tieto hodnoty:

```
zoz = [10, 25, 30, 32, 43, 45, 51, 53, 58, 59, 63, 65, 70, 81, 82]
```

Pomocou algoritmu binárneho vyhľadávania zisťujeme, či sa v ňom nachádza nejaká hodnota. Do základného cyklu algoritmu sme vložili príkaz na vypísanie indexu aj hodnoty stredného prvku v hľadanom úseku:

```
def hladaj(hodnota):
    zac, kon = 0, len(zoz)-1
    while zac <= kon:
        stred = (zac + kon) // 2
        print(stred, zoz[stred])            # pridali sme vypis
        if zoz[stred] < hodnota:
            zac = stred + 1
        elif zoz[stred] > hodnota:
            kon = stred - 1
        else:
            return True
    return False
```

   (a) Vypíšte všetky tieto dvojice stredný index a hodnota, keď budeme hľadať číslo 63, t.j. volanie `hladaj(63)`.

   (b) Vypíšte tieto dvojice pre hľadanú hodnotu 15, t.j. volanie `hladaj(15)`.

6. Napíšte funkciu `pocet_riadkov(meno_suboru)`, ktorá vráti počet riadkov zadaného súboru. Ak daný súbor neexistuje (nepodaril sa `open()`), funkcia vráti `-1`.

```
def pocet_riadkov(meno_suboru):



    ...
```

7. Napíšte funkciu `farba(retazec)`, ktorá z daného reťazca - mena farby v slovenčine vráti správny názov farby pre `tkinter`. Ak danú farbu nerozpozná, vráti farbu `'pink'`. Funkcia by mala akceptovať tieto slovenské mená farieb: `'biela'`, `'cervena'`, `'modra'`, `'zlta'`, `'zelena'` . Vo funkcii nepoužite príkaz ``if.

```
def farba(retazec):

```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    return ...
```

8. Mali sme napísať funkciu `viac(zoz)`, ktorá vráti množinu tých prvkov zoznamu `zoz`, ktoré sa v ňom vyskytujú viac ako raz. Napr.

```
>>> viac(['prvy', 2, (3, 4), 'dva', 3, 4, 'prvy', 3])
{'prvy', 3}
```

Lenže v našom programe, je niekoľko chýb. Opravte ich. Nepridávajte nové príkazy.

```
def viac(zoz):
    mnozina1, mnozina2 = {}, {}
    for prvok1 in zoz:
        for prvok2 in mnozina1:
            if prvok1 == prvok2:
                mnozina2.add(prvok2)
            else:
                mnozina1.add(prvok1)
     return mnozina1
```

9. Funkcia:

```
def pocitaj(a=0, b=0, c=0, d=0, e=0, f=0):
    return a - b + c - d + e - f
```

funguje pre maximálne 6 parametrov (postupne k výsledku pripočítava a odpočítava číselné parametre). Prerobte túto funkciu tak, aby mohla mať ľubovoľný počet parametrov. Napr.

```
>>> pocitaj(17, 15, 13, 11, 9, 7, 5, 3, 1)
9
```

Zapíšte:

```
def pocitaj(.....................):

    return ...
```

10. Zapíšte funkciu `vyber()`, ktorá dostáva dva parametre: nejakú postupnosť hodnôt a postupnosť indexov. Funkcia vráti zoznam hodnôt (z prvého parametra), ktoré sú na príslušných indexoch (druhého parametra). Funkciu zapíšte generátorovou notáciou:

```
def vyber(postupnost, indexy):

    return ...
```

Napr.

```
>>> vyber('python', (1, 3, 5, 4, 2, 0))
['y', 'h', 'n', 'o', 't', 'p']
>>> vyber([11, 12, 13, 14], (2, 1, 2, 0, 1, 0, 0, 1))
[13, 12, 13, 11, 12, 11, 11, 12]
```

37.12 Záverečný test z Programovania (1) 2017/2018

1. Pre danú rekurzívnu funkciu `pocet(pole1, pole2)`:

```
def pocet(pole1, pole2):
    if len(pole2) == 0:
        return 0
    if len(pole2) == 1:
        return int(pole2[0] in pole1)
    stred = len(pole2) // 2
    return pocet(pole1, pole2[:stred]) + pocet(pole1, pole2[stred:])
```

   zistite výsledok volania:

```
>>> pocet('a m', 'mama ma emu')
```

2. Korytnačka `t` nakreslí pomocou tejto rekurzívnej funkcie binárny strom:

```
def strom(d):
    t.fd(d)
    if d > 10:
        t.lt(40)
        strom(d // 2)
        t.rt(75)
        strom(d // 3)
        t.lt(35)
    t.bk(d)
```

   Pri tomto kreslení sa otáča vľavo aj vpravo. Zistite o aký celkový uhol sa otočí vľavo (súčet všetkých otočení vľavo) a o aký vpravo po zavolaní:

```
>>> strom(50)
```

3. Programovali sme funkciu `zisti_dlzky(tab)`, ktorá zistí, či sú všetky riadky neprázdnej vstupnej tabuľky rovnako dlhé, ak áno, funkcia vráti túto dĺžku, inak vráti `None`:

```
def zisti_dlzky(tab):
    i = 0
    while i < len(tab):
        if i == 0:
            ____________________________
        elif ____________________________:
            vysl, i = None, len(tab)
        i += 1
    return vysl
```

   Dopíšte vyznačené časti.

4. Máme definované tieto dve triedy, pričom trieda `FarebnyBod` je odvodená od `Bod`:

```
class Bod(object):
    def __init__(self, x, y):
        self.x, self.y = x, y

    def posun(self, dx=0, dy=0):
        self.x += dx
        self.y += dy
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
class FarebnyBod(Bod):
    def __init__(self, x, y, farba='black'):
        self = Bod(x, y)
        farba = self.farba

    def zmen(self, farba=None, dx=0, dy=0):
        if self.farba is not None:
            farba = self.farba
        Bod.posun(dx, dy)
```

Opravte všetky chyby. Neopravujte to, čo nie je chyba.

5. Dopíšte funkciu `ako(hodnota1, hodnota2)`, ktorá najprv zistí typ prvého parametra `hodnota1` a potom sa pokúsi pretypovať na tento typ druhý parameter `hodnota2`. Ak sa toto pretypovanie úspešne podarí, funkcia vráti túto pretypovanú hodnotu, inak vráti `None`. Ošetrite výnimky `ValueError` a `TypeError`:

```
def ako(hodnota1, hodnota2):
    typ = type(hodnota1)
    ____________________________________:
        return ____________________________
    ____________________________________:
        return ____________________________
```

6. Prepíšte nasledovný výraz tak, aby v ňom neboli volania magických metód:

```
a.__sub__(b.__mul__(c)).__add__(d.__pow__(e))
```

7. Opravte chyby vo funkcii `vsetky(postupnost)`, ktorá by mala vrátiť množinu všetkých dvojíc z prvkov zadanej postupnosti. Ak je v tejto množine dvojica `(a, b)` nemala by sa v nej objaviť dvojica `(b, a)`. Funkcia tiež odfiltruje dvojice rovnakých hodnôt. Nedopisujte ďalší kód, len opravte chyby.

```
def vsetky(postupnost):
    vysl = {}
    for prvy in postupnost:
        for druhy in set(postupnost) - prvy:
            if druhy, prvy not in vysl:
                vysl.add(prvy, druhy)
    return vysl
```

Napr. by malo fungovať:

```
>>> vsetky('java')
{('j', 'v'), ('j', 'a'), ('a', 'v')}
```

8. Dopíšte funkciu `dve_kocky(n)`, ktorá bude simulovať hody dvoch hracích kociek (s číslami od 1 do 6) a evidovať si, koľkokrát padol ktorý súčet. Zrejme súčty budú čísla od 2 do 12. Funkcia bude simulovať `n` takýchto hodov dvomi kockami a vráti dvojicu (sučet, počet) najčastejšieho padnutého súčtu a počet koľkokrát padol.

```
from random import randint as ri

def dve_kocky(n):
    pocet = {}
    for i in range(n):
        sucet = ri(1, 6) + ri(1, 6)
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
        pocet[sucet] = ____________________________________
    naj = None
    for suc, poc in ____________________________:
        if ____________________________________:
            naj = suc
    return ____________________________________
```

9. Zistite, čo sa vypíše:

```
def zisti_mn(veta):
    v1, v2 = veta[:-1], veta[1:]
    z = [(v1[i], v2[i]) for i in range(len(v1))]
    return {a+b for a, b in z}

>>> zisti_mn('mama_ma_emu_a_ema_ma_mamu')
```

10. Dopíšte funkciu `vyrob(n)`, ktorá vyrobí dvojrozmernú tabuľku veľkosti `n` x `n`, pričom všetky prvky sú očíslované po riadkoch od 1 do `n*n`. Napr.

```
>>> vyrob(3)
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Zapíšte to pomocou generátorovej notácie:

```
def vyrob(n):
    return ____________________________________
```

37.13 Záverečný test z Programovania (2) 2015/2016

1. Máme daný slovník:

```
d = {'bum' :7, 'bom':3, 'bam' :5, 'bim' :6, 'bem' :2}
```

   Zistite, čo vráti tento kód:

```
'.'.join(b for a, b in sorted((d[x],x) for x in d))
```

2. Vrchol spájaného zoznamu je definovaný takto:

```
class Vrchol:
    def __init__(self, data, next=None):
        self.data, self.next = data, next
```

   Napíšte funkciu `kopia`, ktorá dostáva ako parameter začiatok nejakého zoznamu a vráti nový zoznam, ktorý je kópiou pôvodného:

```
def kopia(zoznam):
    ...
```

3. Do triedy `Zoznam` dopíšte metódu `vyhod2()`, ktorá z momentálneho zoznamu vyhodí každý druhý vrchol:

```
class Vrchol:
    def __init__(self, data, next):
        self.data = data
        self.next = next

class Zoznam:
    def __init__(self):
        self.zac = None

    def vyhod2(self):
        pom = self.zac
        while ________________________________________:
            ____________________
            pom = pom.next
```

4. Na začiatku bol **zásobník** prázdny. Potom sme vykonali niekoľko operácií `push` a `pop`.

   Zistite, čo sa vypíše po spustení:

```
print('pre zasobnik:')
for x in 7, 3, 5, 11, 3, 8, 4:
    push(x)
for x in range(6):
    print(pop(), pop(), end=' ')
    push(x)
print(pop())
```

   Na začiatku bol **rad** prázdny. Potom sme vykonali niekoľko operácií `enqueue` a `dequeue`.

   Zistite, čo sa vypíše po spustení:

```
print('pre rad:')
for x in 7, 3, 5, 11, 3, 8, 4:
```

(pokračuje na ďalšej strane)

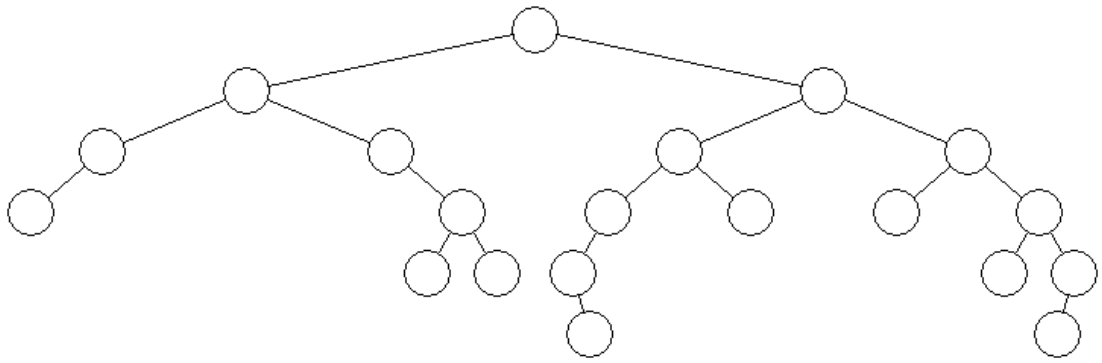
(pokračovanie z predošlej strany)

```
    enqueue(x)
for x in range(6):
    print(dequeue(), dequeue(), end=' ')
    enqueue(x)
print(dequeue())
```

5. Máme dané dva výpisy toho istého binárneho stromu. Nakreslite tvar tohto stromu a vypíšte aj `strom.preorder()`:

```
>>> print(strom.inorder())
3 5 10 2 1 6 7 9 4 8
>>> print(strom.postorder())
3 10 2 5 7 6 1 8 4 9
```

6. Dopíšte do stromu na obrázku čísla z postupnosti `range(20)` tak, aby vznikol **binárny vyhľadávací strom**:



7. Nakreslite aritmetický strom, pre ktorý je táto postupnosť **postorderom**:

```
4 5 + 22 7 / 9 - *
```

8. Máme daný algoritmus **quick sort**:

```
def quick_sort(pole):
    def rozdel(z, k):
        pivot, index = pole[z], z
        for i in range(z + 1, k + 1):
            if pole[i] < pivot:
                index += 1
                pole[i], pole[index] = pole[index], pole[i]
        pole[z], pole[index] = pole[index], pole[z]
        return index

    def quick_sort1(z, k):
        if z < k:
            index = rozdel(z, k)
            quick_sort1(z, index - 1)
            quick_sort1(index + 1, k)

    quick_sort1(0, len(pole) - 1)
```

Zapíšte obsah poľa po každom návrate z funkcie rozdel, pričom na začiatku boli v poli:

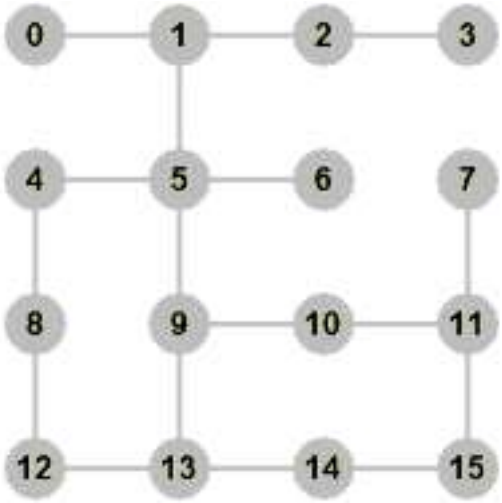
68	58	25	32	88	82	37	86	95	15	

Prvok v riadku tabuľky, ktorý bol pritom pivotom, zakrúžkujte.

9. Nakreslite orientovaný ohodnotený graf, ktorý sme zapísali pomocou asociatívneho poľa vrcholov s asociatívnymi poľami susedov:

```
graf = {15: {14: 'ke', 10: 'bl'}, 4: {17: 'sc', 14: 'pk'}, 10: {14: 'nz'},
        17: {10: 'ma'}, 14: {10: 'ds'}}
```

10. Na daný graf sme spustili algoritmus do šírky, pričom sa nevypisovalo poradové číslo vrcholu ale hodnota uložená vo vrchole (jednoznakový reťazec v premennej meno). Ako boli uložené hodnoty vo vrcholoch grafu, keď sa presne v tomto poradí vypísali znaky `'p r o g r a m u j e p y t h o n'` a pritom prehľadávanie začalo vo vrchole **10**.




```
def vypis_dosirky(v):
    queue = [v]
    visited = set()
    while len(queue) != 0:
        v = queue.pop(0)
        if v not in visited:
            visited.add(v)
            print(pole[v].meno, end=' ')
            for i in sorted(pole[v].sus):
                if i not in visited:
                    queue.append(i)
```

37.14 Záverečný test z Programovania (2) 2016/2017

1. Turingov stroj je zadaný tabuľkou:

	start	jeden	dva	
x	> jeden	> dva	> start	
y	> start	> jeden	> dva	
_		= stop		

   Stavy sú: **start**, **jeden**, **dva**, **stop** (koncový) a symboly na páske: **x**, **y**, _ (prázdny).

   Zistite, ktoré z uvedených reťazcov budú týmto turingovým strojom akceptované?

   (a) **xyxyxyx**

   (b) **yyxyyxy**

   (c) **xyyyyyx**

   (d) **xxxxxxx**

2. Zistite, čo vypíše táto funkcia:

```
def test(slovo):
    queue = Queue()
    for znak in slovo:
        try:
            p = queue.dequeue()
            queue.enqueue(znak)
            queue.enqueue(p)
        except EmptyError:
            queue.enqueue(znak)
    while not queue.empty():
        print(queue.dequeue(), end='')
    print()

test('abcdef')
```

   Výpis:

```

```

3. Zadaný aritmetický výraz:

```
1 + (2 + 3) * 4 / (5 - 6) + 7 * 8
```

   prepíšte do

   (a) prefixového zápisu

   (b) postfixového zápisu

   Operácie s rovnakou prioritou sa vyhodnocujú zľava doprava.

4. Napíšte funkciu `vyhod_posledny(zoznam)`, v ktorej parametrom je referencia na začiatok zoznamu typu `Vrchol` (alebo `None`). Funkcia vráti referenciu na pôvodný zoznam, ktorý už bude bez posledného prvku:

```
class Vrchol:
    def __init__(self, data, next=None):
        self.data, self.next = data, next

def vyhod_posledny(zoznam):




    return zoznam
```

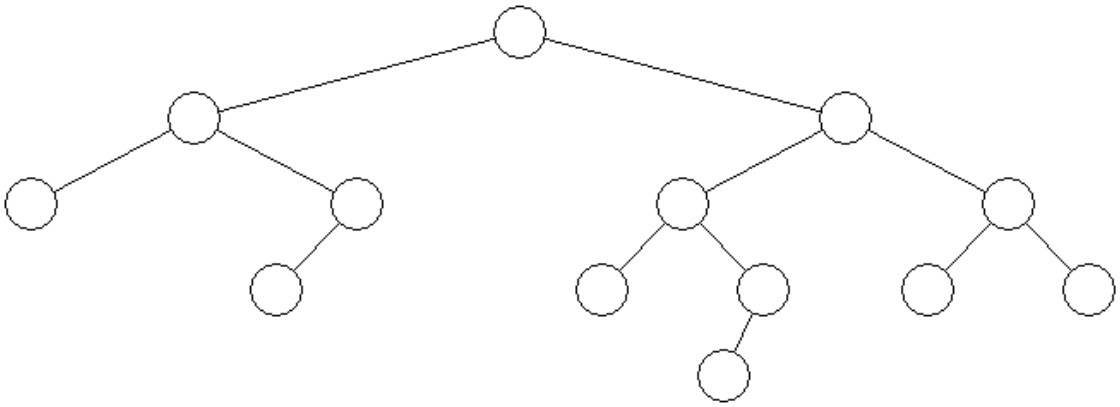
5. Rekurzívna funkcia `vytvor_uplny()` by mala vrátiť úplný binárny strom, v ktorom v listoch budú `0`, v ich predkoch budú `1`, v každej úrovni bližšie ku koreňu bude o 1 väčšie číslo, až v koreni by teda mala byť výška celého stromu - teda `n`:

```
class Vrchol:
    def __init__(self, data, left=None, right=None):
        self.data, self.left, self.right = data, left, right

def vytvor_uplny(n, cislo=0):
    if cislo > n:
        return 0
    lavy = pravy = vytvor_uplny(n - 1, cislo + 1)
    koren = Vrchol(n - cislo, lavy, pravy)
    return koren
```

   Opravte všetky (štyri) chyby v tele funkcie.

6. Pre daný strom vpíšte do vrcholov hodnoty tak, aby výpisom pre postupnosť **postorder** bolo `'programovanie'`.

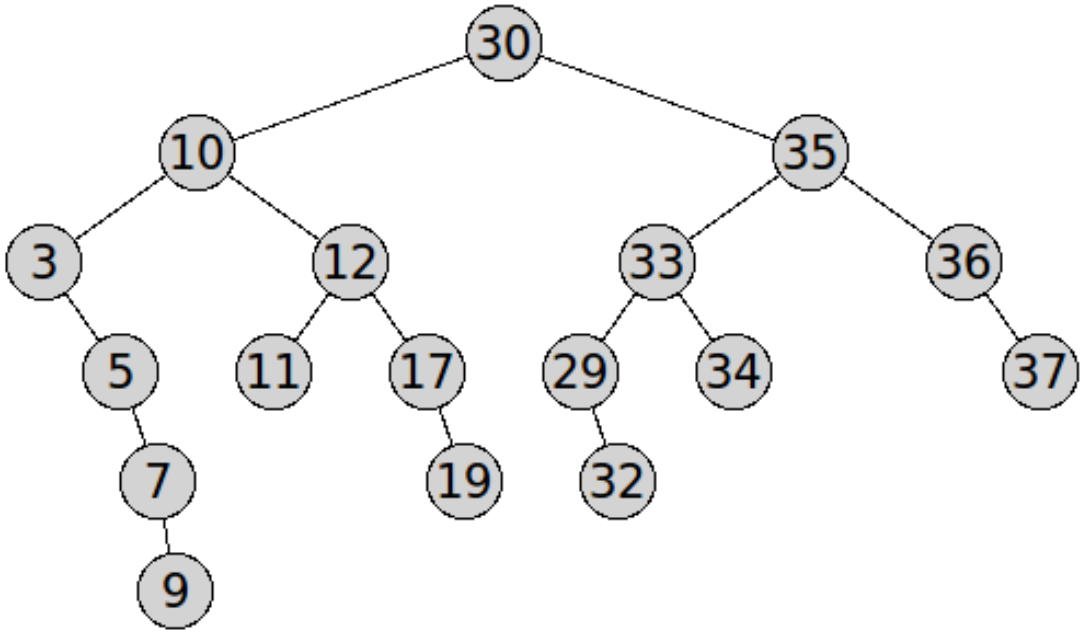


7. Pre daný prefixový zápis

```
* + / a b - c d + e f
```

   nakreslite príslušný **aritmeticky strom**.

8. Tento strom možno nespĺňa podmienky **BVS** (binárny vyhľadávací strom). Označte minimálny počet vrcholov stromu, ktorým keby sme navzájom vymenili ich hodnoty, dostaneme **BVS**.



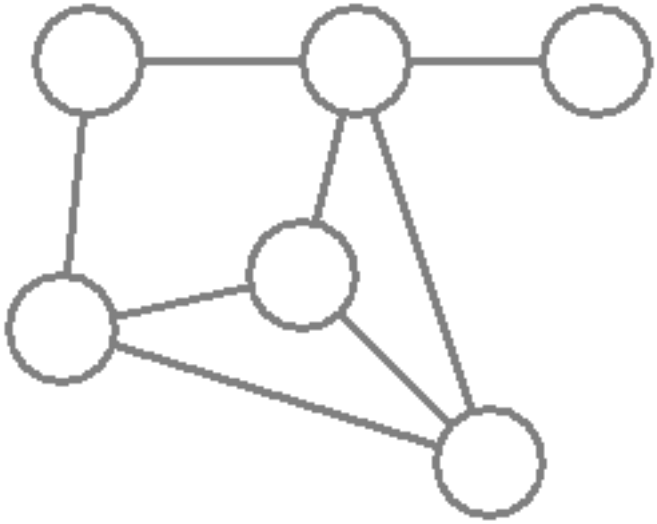
9. Táto verzia triedenia vkladaním v niektorých situáciách vypisuje momentálny obsah celého zoznamu. Vypíšte tieto kontrolné výpisy:

```
def insert_sort(zoz):
    for i in range(len(zoz) - 1):
        prvok = zoz.pop(i + 1)
        while i >= 0 and zoz[i] > prvok:
            i -= 1
        zoz.insert(i + 1, prvok)
        print(zoz)

p = [5, 9, 4, 3, 6, 10, 1, 8, 2, 7]
insert_sort(p)
```

riadky výpisu:

10. Označte písmenami vrcholy grafu na obrázku



ak poznáte túto jeho reprezentáciu:

	A	B	C	D	E	F	
A	0	1	0	1	0	1	
B	1	0	0	1	1	0	
C	0	0	0	0	0	1	
D	1	1	0	0	0	1	
E	0	1	0	0	0	1	
F	1	0	1	1	1	0	

37.15 Záverečný test z Programovania (2) 2017/2018

1. Prepíšte do prefixového zápisu tento výraz:

```
1 2 3 * + 4 5 * + 6 7 * +
```

2. Naprogramujte funkciu `otoc_rad(queue)`, ktorá otočí poradie prvkov v danej dátovej štruktúre rad. Na otáčanie radu použite pomocný zásobník:

```
def otoc_rad(queue):
    stack = Stack()

    while ____________________________:

        stack.____________________________

    while ____________________________:

        queue.____________________________
```

3. Máme napísať funkciu `spoj(zoz1, zoz2)`, ktorá na koniec spájaného zoznamu `zoz1` pripojí spájaný zoznam `zoz2` a ako výsledok vráti začiatok takéhoto nového zoznamu. Opravte chyby:

```
def spoj(zoz1, zoz2):
    if zoz1 is None:
        return None

    while zoz1.next is not None:
        zoz1 = zoz1.next
    zoz1.next = zoz2
    return zoz1
```

4. Metóda `mapuj()` postupne prejde všetky vrcholy spájaného zoznamu a každému (ktorý spĺňa podmienku) zmení hodnotu podľa zadanej funkcie:

```
class SpajanyZoznam:
    ...

    def mapuj(self, podmienka, funkcia):
        pom = self.zac
        while pom is not None:
            if podmienka(pom.data):
                pom.data = funkcia(pom.data)
            pom = pom.next
```

Doplňte chýbajúce definície funkcií:

```
>>> zoz = SpajanyZoznam(range(7))

>>> pdm = lambda ____________________________

>>> fun = lambda ____________________________
>>> zoz.mapuj(pdm, fun)
```

(pokračuje na ďalšej strane)

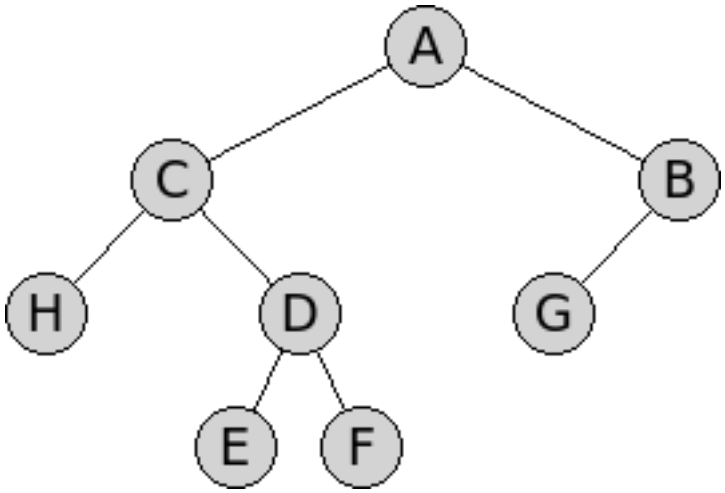
(pokračovanie z predošlej strany)

```
>>> zoz
6 -> 5 -> 2 -> 3 -> 4 -> 1 -> 0 -> None
```

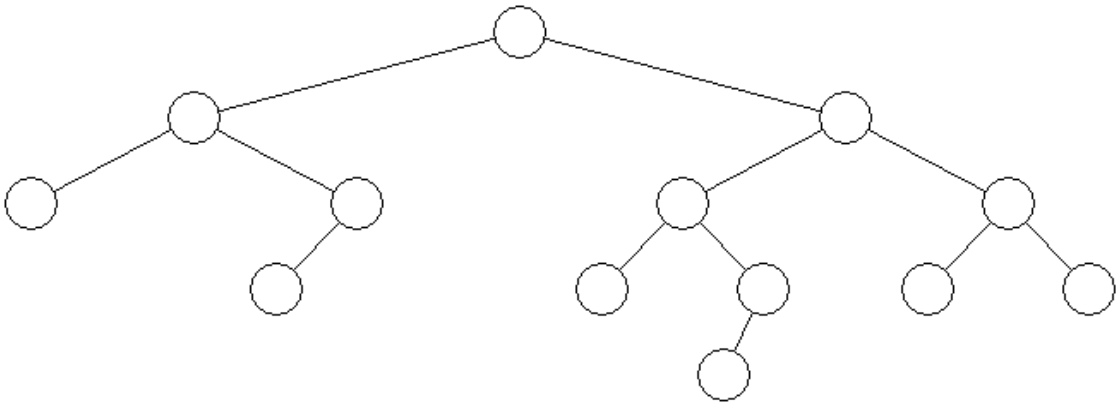
5. Funkcia `zisti(vrch)` niečo zisťuje o neprázdnom binárnom strome:

```
def zisti(vrch, h=0):
    if vrch.left is None:
        if vrch.right is None:
            return h
        return zisti(vrch.right, h+1)
    pom = zisti(vrch.left, h+1)
    if vrch.right is None:
        return pom
    return min(pom, zisti(vrch.right, h+1))
```

   Zistite, čo táto funkcia vráti pre tento binárny strom (funkciu voláme s koreňom stromu):



6. Do daného stromu:



   vpíšte do vrcholov hodnoty tak, aby **postorder**-vypisom tohto stromu bola postupnosť písmen `'programovanie'`.

7. Inicializácia `__init__()` triedy `BVS` môže mať parameter, ktorý je postupnosťou hodnôt, z ktorej sa vytvorí **binárny vyhľadávací strom** postupným volaním metódy `vloz()`. Nakreslite strom, ktorý vznikne volaním:

```
s = BVS('KE BB PO PE PK BA BS NI SC BL RK BR'.split())
s.kresli()
```

8. Aby sa nasledovný orientovaný graf stal neorientovaným, musíme niektorým hranám dodefinovať aj opačný smer:

```
graf = [{2, 3},          # 0
        {3, 5},          # 1
        {0},             # 2
        {0, 1, 5},       # 3
        {6, 7, 8},       # 4
        {1, 2},          # 5
        {},              # 6
        {4},             # 7
        {3, 4}]          # 8
```

   Zadefinujte minimálny počet pridaných orientovaných hrán (vypíšte ich dvojice čísel vrcholov) tak, aby bola každá hrana orientovaná oboma smermi. V tomto grafe potom nájdite aj **maximálny cyklus**.

9. Táto verzia triedenia vkladaním v niektorých situáciách vypíše momentálny obsah celého zoznamu. Odtrasujte tento algoritmus a vypíšte tieto kontrolné výpisy:

```
def insert_sort1(zoz):
    for i in range(1, len(zoz)):
        j, t = i, zoz[i]
        while j > 0 and zoz[j-1] > t:
            zoz[j] = zoz[j-1]
            j -= 1
        if j < i:
            zoz[j] = t
            print(*zoz)

z = [5, 9, 4, 3, 6, 10, 1, 8, 2, 7]
insert_sort1(z)
```

   riadky výpisu:

10. Do triedy `Graf` dopíšte metódu `max_komponent()`, ktorá zistí veľkosť najväčšieho komponentu:

```
def max_komponent(self):

    def dohlbky(v1):
        visited.add(v1)
        for v2 in self.vrcholy[v1].sus:
            if v2 not in visited:
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
                dohlbky(v2)

    visited = set()




    return _______________
```

37.16 Zadanie skúšky z 19.1.2018 - Indiana Jones

V niekoľkých počítačových hrách je hlavnou postavou dobrodruh **Indiana Jones**. Aj v tejto hre sa Indiana pohybuje v jaskynnom bludisku (štvorcovej sieti) pomocou šípok, zbiera zlaté mince, pričom v jaskyni môžu byť chodby alebo celé úseky, ktoré ho automaticky presúvajú ďalej (niečo ako bežiace pásy, výťahy, vodopády). Cieľom je odsledovať, kam sa Indiana po zadanej postupnosti šípok dostane, koľko pritom prejde krokov a koľko pritom zozbiera zlatých mincí.

Textový súbor popisuje hraciu plochu (jaskyňu) ako štvorcovú sieť (všetky riadky sú rovnako dlhé). Prvý riadok súboru obsahuje dve celé čísla: rozmery hracej plochy (počet riadkov a počet stĺpcov). Nasledovný počet riadkov obsahuje samotný popis jaskyne , pričom znaky v súbore majú tento význam:

- `'.'` voľné
- `'x'` stena
- `'>'`, `'v'`, `'<'`, `'^'` políčka s automatickým posunom: vpravo, dole, vľavo, hore

Nasledovný riadok obsahuje pozíciu Indiana Jonesa ako poradové číslo riadka a stĺpca (číslujeme od 0). Tento sa bude v priebehu hry zobrazovať znakom `'i'`. Na začiatku hry stojí indiana na voľnom políčku (znak `'.'`). Za týmto riadkom až do konca súboru nasledujú dvojice celých čísel, ktoré označujú pozície zlatých mincí. Tieto sa budú počas hry zobrazovať znakom `'o'`. Mince sa na začiatku hry môžu nachádzať na ľubovoľných políčkach (okrem štartovej pozície Indiana), t.j. aj na posúvacích políčkach aj na políčkach so stenou (tie sa zrejme nebudú dať zozbierať).

Vašou úlohou je prečítať textový súbor s popisom jaskyne a odsimulovať zadanú postupnosť stláčaných šípok (`'l'` vľavo, `'r'` vpravo, `'u'` hore, `'d'` dole). Samotný pohyb hráča sa bude riadiť podľa týchto pravidiel:

- šípka na voľné políčko (prípadne s mincou) presunie hráča (a zdvihne mincu)
- šípka smerom k stene, resp. za okraj plochy, sa ignoruje (Indiana stojí na mieste)
- šípka na políčko s posunom (jedno z `'>'`, `'v'`, `'<'`, `'^'`) spôsobí automatický presun na ďalšie políčko daným smerom, ak je týmto smerom opäť políčko s posunom, znovu sa to opakuje; ak je týmto smerom stena alebo okraj plochy, hráč zastane na tomto políčku; ak sa na niektorých z týchto políčok nachádzali mince, tie sa počas pohybu zdvihnú.

Riešenie zapíšte do triedy `IndianaJones` s týmito metódami:

```
class IndianaJones:
    def __init__(self, meno_suboru):
        ...

    def __repr__(self):
        return ''

    def rob(self, postupnost):
        return 0, 0

    def pocet_minci(self):
        return 0
```

kde

- metóda `__repr__()` vráti znakový reťazec, ktorý reprezentuje momentálny stav plochy aj s pozíciou hráča (znak `'i'`); s pozíciami všetkých mincí (znak `'o'`), pričom medzi riadkami je znak `'\n'`
- metóda `rob(postupnost)` kde parameter postupnost je znakový reťazec s postupnosťou stláčaných šípok (znaky `'l'`, `'r'`, `'u'`, `'d'`) – hráč sa v ploche pohybuje podľa týchto zadaných príkazov; ak sa daným smerom nemôže pohnúť (okraj plochy, stena), tento konkrétny príkaz sa ignoruje, ak prechádza cez políčko s posunom, tak zrealizuje automatický pohyb; metóda vracia dvojicu čísel: (počet prejdených krokov, počet

zdvihnutých mincí); každé volanie tejto metódy vráti len tie počty krokov a mincí, ktoré sa prešli a pozbierali počas tohto volania

- metóda `pocet_minci()` vráti momentálny počet mincí, ktoré ešte ostali v hracej ploche.

Napr. pre súbor

```
5 7
xxxxxxx
xxx.xxx
x.....x
xxx.xxx
xxxxxxx
2 3
2 5
1 3
3 3
2 1
```

takýto test:

```
if __name__ == '__main__':
    p = IndianaJones('subor1.txt')
    print(p)
    print(p.rob('udrr'))
    print(p.rob('lldull'))
    print(p.rob('lluudd'))
    print(p)
```

vypíše:

```
xxxxxxx
xxxoxxx
xo.i.ox
xxxoxxx
xxxxxxx
(4, 2)
(6, 2)
(0, 0)
xxxxxxx
xxx.xxx
xi....x
xxx.xxx
xxxxxxx
```

Alebo pre súbor

```
3 9
....>>>>v
........v
.....<<<v
0 0
0 5
```

takýto test:

```
if __name__ == '__main__':
    p = IndianaJones('subor2.txt')
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    print(p)
    print('pocet minci =', p.pocet_minci())
    print(p.rob('rrrr'))
    print(p)
    print(p.rob('lr'))
    print(p)
    print('pocet minci =', p.pocet_minci())
```

vypíše:

```
i...>o>>v
........v
.....<<<v
pocet minci = 1
(10, 1)
....>>>>v
........v
.....<<<i
(6, 0)
....>>>>v
........v
....i<<<v
pocet minci = 0
```

Z úlohového servera L.I.S.T. si stiahnite kostru programu `skuska.py`. Pozrite si testovacie dáta v súboroch `'subor1.txt'`, `'subor2.txt'`, `'subor3.txt'`, ..., ktoré bude používať testovač.

Aby ste mohli spúšťať skúškové testy, program uložte do súboru `skuska.py`. Riešenie (bez dátových súborov) odovzdajte na úlohový server https://list.fmph.uniba.sk/.

Praktická časť končí o 11:00 a skúška pokračuje od 12:00 vyhodnotením v kancelárii **m162**.

37.17 Zadanie skúšky z 24.1.2018 - Čistiaci robot

Keďže v priemyselnom parku treba niektoré plochy čistiť každý deň, vlastníci sa rozhodli nainštalovať čistiaceho robota. Robot má nejakú svoju počiatočnú polohu a tiež smer, v ktorom je natočený (predpokladáme plánik, ktorý je zakreslený do štvorcovej siete). V danom smere sa pohybuje samostatne, kým nepríde ku nejakej prekážke, alebo na okraj priemyselného parku. Pri svojom pohybe, vyčistí každé políčko štvorcovej siete. Lenže niektoré políčka sú tak intenzívne zašpinené, že robot cez ne musí prejsť aspoň dvakrát, aby sa naozaj očistili. Keď robot pri svojom upratovaní narazí na prekážku (aj okraj plochy je prekážkou), tak zastane a očakáva zadanie jedného z povelov:

- `'l'` otočí sa vľavo vbok (o 90 stupňov) a pokračuje v upratovaní v tomto novom smere;
- `'p'` otočí sa vpravo vbok (o 90 stupňov) a pokračuje v upratovaní v tomto novom smere;
- `'z'` otočí sa čelom vzad (o 180 stupňov) a pokračuje v upratovaní v tomto novom smere;

Robot skončí (zastane), keď už nedostáva ďalšie príkazy.

Zadanie plániku pre robota je v textovom súbore, v ktorom v každom riadku je jeden riadok štvorcovej siete (všetky riadky sú rovnako dlhé), pričom jednotlivé znaky majú tento význam:

- `'.'` voľné políčko
- `'X'` políčko s prekážkou
- `'1'` znečistené políčko
- `'2'` dvojnásobne znečistené políčko

Naprogramujte triedu `Robot`:

```
class Robot:
    def __init__(self, meno_suboru):
        ...

    def start(self, riadok, stlpec, smer):
        return 0

    def __str__(self):
        return ''

    def rob(self, povely):
        return 0

    def pocet_znecistenych(self):
        return 0
```

kde

- `init` prečíta súbor - robot tam zatiaľ nie je
- `start` položí robota na zadaný riadok a stĺpec, natočí ho v jednom zo 4 smerov: **0** na sever, **1** na východ, **2** na juh, **3** na západ; môžete počítať s tým, že toto štartové políčko je voľné (obsahuje `'.'`); zároveň sa robot daným smerom rozbehne a zastane až na prekážke; metóda vráti počet znečistených políčok, ktoré musel pritom očistiť
- `__str__` vráti znakovú reprezentáciu plochy: políčka, ktoré úplne očistil označí `'o'`, políčka s `'2'`, cez ktoré prešiel len raz, označí `'1'`, pozíciu robota zapíšte podľa jeho momentálneho smeru natočenia jedným zo znakov `'^'` pre sever, `'>'` pre východ, `'v'` pre juh a `'<'` pre západ
- `rob` dostáva jeden povel, alebo postupnosť za sebou nasledujúcich povelov, pričom povel je jedno z písmen `'l'`, `'p'` alebo `'z'`; metóda vráti počet znečistených políčok, ktoré musel pritom očistiť (možno niektoré aj dvakrát)

- `pocet_znecistenych` metóda vráti počet znečistených políčok, ktoré sa nachádzajú v celej ploche (dvojnásobne znečistené počítame dvakrát)

Napr. pre súbor:

```
XXXXXXXXXX
.....1...X
.........X
...2.....X
```

takýto test:

```
if __name__ == '__main__':
    r = Robot('subor1.txt')
    print(r)
    print('pocet znecistenych =', r.pocet_znecistenych())
    print('start(2, 3, 1) vratil', r.start(2, 3, 1))
    print(r)
    print('pocet znecistenych =', r.pocet_znecistenych())
    print("rob('ll') vratil", r.rob('ll'))
    print(r)
    print('pocet znecistenych =', r.pocet_znecistenych())
    print("rob('pzl') vratil", r.rob('pzl'))
    print(r)
    print('pocet znecistenych =', r.pocet_znecistenych())
    print("rob('pp') vratil", r.rob('pp'))
    print(r)
    print('pocet znecistenych =', r.pocet_znecistenych())
```

vypíše:

```
XXXXXXXXXX
.....1...X
.........X
...2.....X
pocet znecistenych = 3
start(2, 3, 1) vratil 0
XXXXXXXXXX
.....1...X
...ooooo>X
...2.....X
pocet znecistenych = 3
rob('ll') vratil 1
XXXXXXXXXX
<oooooooX
...oooooX
...2.....X
pocet znecistenych = 2
rob('pzl') vratil 1
XXXXXXXXXX
ooooooooX
o..oooooX
ooo1oooo>X
pocet znecistenych = 1
rob('pp') vratil 1
XXXXXXXXXX
ooooooooX
o..oooooX
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
<oooooooX
pocet znecistenych = 0
```

Z úlohového servera L.I.S.T. si stiahnite kostru programu `skuska.py`. Pozrite si testovacie dáta v súboroch `'subor1.txt'`, `'subor2.txt'`, `'subor3.txt'`, ..., ktoré bude používať testovač.

Aby ste mohli spúšťať skúškové testy, program uložte do súboru `skuska.py`. Riešenie (bez dátových súborov) odovzdajte na úlohový server https://list.fmph.uniba.sk/.

Praktická časť končí o 11:00 a skúška pokračuje od 12:00 vyhodnotením v kancelárii **m162**.

37.18 Zadanie skúšky z 5.2.2018 - Robot Mravec

Máme cvičeného malého robota mravca, ktorý sa pohybuje po štvorcovej sieti a pritom vie pred sebou tlačiť malé kocky. Mravec poslúcha na povely `'l'` (vľavo), `'p'` (vpravo), `'h'` (hore), `'d'` (dole), pričom sa v danom smere posunie na susedné políčko štvorcovej siete. Ak sa v danom smere nachádza kocka, tak ju pred sebou v tomto smere potlačí. Ak je v danom smere tesne za sebou viac kociek, tak ich tlačí všetky. Mravec z plochy nikdy nevyjde, hoci kocky, ktoré pred sebou tlačí, z plochy vypadnúť môžu.

Na každej kocke je zapísané jedno písmeno. Samotná štvorcová sieť vie indikovať, či sa na niektorých špeciálnych políčkach nachádzajú kocky s písmenami a vie zistiť množinu písmen na týchto kockách.

Zadanie štvorcovej siete s počiatočným rozložením kociek je v textovom súbore. V prvých riadkoch tohto súboru sú nachádzajú riadky štvorcovej siete (všetky sú rovnako dlhé), pričom voľné políčka sú označené znakom `'.'` a iné znaky označujú kocky s týmito znakmi. Za štvorcovou sieťou je v súbore jeden riadok prázdny a za tým nasleduje postupnosť súradníc špeciálnych políčok - v každom ďalšom riadku je dvojica celých čísel, ktorá označuje riadok a stĺpec jedného takéhoto políčka (číslujeme od 0).

Naprogramujte triedu `Mravec`:

```
class Mravec:
    def __init__(self, meno_suboru):
        ...

    def __str__(self):
        return ''

    def start(self, riadok, stlpec):
        ...

    def rob(self, prikazy):
        ...

    def zisti(self):
        return set()
```

kde

- `init` prečíta súbor - mravec tam zatiaľ nie je
- `start` položí mravca na zadaný riadok a stĺpec
- `__str__` vráti znakovú reprezentáciu plochy: pozíciu mravca zapíšte znakom `'@'` a špeciálne políčka, na ktorých sa nenachádza ani mravec ani kocka, zapíšte znakom `'+'`
- `rob` dostáva jeden povel, alebo postupnosť za sebou nasledujúcich povelov, pričom povel je jedno z písmen `'l'`, `'p'`, `'h'` alebo `'d'`; mravec sa postupne pohybuje v daných smeroch, pričom pred sebou môže tlačiť kocky; povely, ktoré sa nedajú vykonať, ignoruje
- metóda `zisti` vráti množinu písmen na špeciálnych políčkach hracej plochy

Napr. pre súbor:

```
.....
.A.B.
.CD..
.....

1 2
1 4
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
2 1
2 2
```

takýto test:

```
if __name__ == '__main__':
    m = Mravec('subor1.txt')
    print(m)
    print('zisti =', m.zisti())
    m.start(1, 0)
    m.rob('pp')
    print(m)
    print('zisti =', m.zisti())
    m.rob('dl')
    print(m)
    print('zisti =', m.zisti())
```

vypíše:

```
.....
.A+B+
.CD..
.....
zisti = {'D', 'C'}
.....
..@AB
.CD..
.....
zisti = {'B', 'D', 'C'}
.....
..+AB
C@+..
..D..
zisti = {'B'}
```

Z úlohového servera L.I.S.T. si stiahnite kostru programu `skuska.py`. Pozrite si testovacie dáta v súboroch `'subor1.txt'`, `'subor2.txt'`, `'subor3.txt'`, ..., ktoré bude používať testovač.

Aby ste mohli spúšťať skúškové testy, program uložte do súboru `skuska.py`. Riešenie (bez dátových súborov) odovzdajte na úlohový server https://list.fmph.uniba.sk/.

Praktická časť končí o 11:00 a skúška pokračuje od 12:00 vyhodnotením v kancelárii **m162**.

37.19 Zadanie skúšky z 12.2.2018 - Robot Mravec

Máme cvičeného malého robota mravca, ktorý sa pohybuje po štvorcovej sieti a pritom vie pred sebou tlačiť malé kocky. Mravec poslúcha na povely `'l'` (vľavo), `'p'` (vpravo), `'h'` (hore), `'d'` (dole), pričom sa v danom smere posunie na susedné políčko štvorcovej siete. Ak sa v danom smere nachádza kocka, tak ju pred sebou v tomto smere potlačí. Ak je v danom smere tesne za sebou viac kociek, tak ich tlačí všetky. Mravec z plochy nikdy nevyjde, hoci kocky, ktoré pred sebou tlačí, z plochy vypadnúť môžu.

Na každej kocke je zapísané jedno písmeno. Samotná štvorcová sieť vie indikovať, či sa na niektorých špeciálnych políčkach nachádzajú kocky s písmenami a vie zistiť množinu písmen na týchto kockách.

Zadanie štvorcovej siete s počiatočným rozložením kociek je v textovom súbore. V prvých riadkoch tohto súboru sa nachádzajú riadky štvorcovej siete (všetky sú rovnako dlhé), pričom špeciálne políčka sú označené znakom `'+'` a zvyšné sú označené znakom `'.'`. Za štvorcovou sieťou je v súbore jeden riadok prázdny a za tým nasleduje postupnosť súradníc kociek s písmenami - v každom ďalšom riadku je trojica: písmeno a dve celé čísla. Táto trojica označuje písmeno na kocke a jej pozíciu v ploche: riadok a stĺpec (číslujeme od 0).

Naprogramujte triedu `Mravec`:

```
class Mravec:
    def __init__(self, meno_suboru):
        ...

    def __str__(self):
        return ''

    def start(self, riadok, stlpec):
        ...

    def rob(self, prikazy):
        ...

    def zisti(self):
        return set()
```

kde

- `init` prečíta súbor - mravec tam zatiaľ nie je
- `start` položí mravca na zadaný riadok a stĺpec
- `__str__` vráti znakovú reprezentáciu plochy: pozíciu mravca zapíšte znakom `'@'` a špeciálne políčka, na ktorých sa nenachádza ani mravec ani kocka, zapíšte znakom `'+'`
- `rob` dostáva jeden povel, alebo postupnosť za sebou nasledujúcich povelov, pričom povel je jedno z písmen `'l'`, `'p'`, `'h'` alebo `'d'`; mravec sa postupne pohybuje v daných smeroch, pričom pred sebou môže tlačiť kocky; povely, ktoré sa nedajú vykonať, ignoruje
- metóda `zisti` vráti množinu písmen na špeciálnych políčkach hracej plochy

Napr. pre súbor:

```
.....
..+.+
.++..
.....

D 2 2
C 2 1
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
A 1 1
B 1 3
```

takýto test:

```
if __name__ == '__main__':
    m = Mravec('subor1.txt')
    print(m)
    print('zisti =', m.zisti())
    m.start(1, 0)
    m.rob('pp')
    print(m)
    print('zisti =', m.zisti())
    m.rob('dl')
    print(m)
    print('zisti =', m.zisti())
```

vypíše:

```
.....
.A+B+
.CD..
.....
zisti = {'D', 'C'}
.....
..@AB
.CD..
.....
zisti = {'B', 'D', 'C'}
.....
..+AB
C@+..
..D..
zisti = {'B'}
```

Z úlohového servera L.I.S.T. si stiahnite kostru programu `skuska.py`. Pozrite si testovacie dáta v súboroch `'subor1.txt'`, `'subor2.txt'`, `'subor3.txt'`, ..., ktoré bude používať testovač.

Aby ste mohli spúšťať skúškové testy, program uložte do súboru `skuska.py`. Riešenie (bez dátových súborov) odovzdajte na úlohový server https://list.fmph.uniba.sk/.

Praktická časť končí o 11:00 a skúška pokračuje od 12:00 vyhodnotením v kancelárii **m162**.

37.20 Zadanie skúšky z 4.6.2018 - Usilovný ježko

Usilovný ježko behá po chodníkoch záhradky a na svoj pichľavý chrbát zbiera všetky kusy ovocia, ktoré pri tom nájde. Lenže na chrbát sa toho veľa nezmestí, preto sa rozhodol, že z každého druhu ovocia bude brať len po jednom kuse. Keďže je ale dosť chamtivý, nezvládne prejsť okolo ovocia a nezobrať ho. Aby sme ježka netrápili, zostavíme mu takú trasu, aby prešiel okolo všetkých typov ovocia ale popri každom druhu len raz. Ešte bude treba dodržať to, aby neprešiel po žiadnom chodníku viac ako raz.

Mapa záhradky tvorí **neorientovaný ohodnotený graf**, v ktorom hrany reprezentujú chodníky a vrcholy sú križovatky, kde sa chodníky rozvetvujú. Na niektorých chodníkoch sa môže nachádzať jeden kus nejakého ovocia (chodník môže byť aj bez ovocia). Program potom pre zadaný štartový vrchol nájde ľubovoľnú trasu, ktorou sa vyzbierajú všetky druhy ovocia a pritom neprejde po žiadnom chodníku viackrát. Niektorý druh ovocia sa môže nachádzať na viacerých chodníkoch, vtedy nájdená trasa prejde len cez jeden z týchto chodníkov.

Záhradka je zadaná v textovom súbore, ktorý má takúto štruktúru:

- každý riadok popisuje jeden chodník ako dvojicu vrcholov, na ktorých sa môže nachádzať jeden kus ovocia nejakého druhu;
- vrcholy (križovatky chodníkov) sú znakové reťazcom (napr. `'3'`, `'v1'`, `'abc'`), ovocie je označené tiež nejakým reťazcom (napr. `'a'`, `'jablko'`, `'7'`);
- ak sa na chodníku nenachádza žiadne ovocie, riadok súboru obsahuje len 2 mená vrcholov;
- ak sa na chodníku nachádza nejaké ovocie, v riadku súboru sú okrem dvoch mien vrcholov aj meno ovocia a to v poradí: prvý vrchol, ovocie, druhý vrchol;
- graf je neorientovaný, preto, ak je vrchol `A` spojený chodníkom s `B`, tak potom aj vrchol `B` susedí s vrcholom `A`, zrejme to platí aj pre ovocie – možno to budete musieť do grafu dodefinovať.

Zadefinujte triedu `Zahradka`:

```
class Zahradka:
    def __init__(self, meno_suboru):
        ...

    def vrcholy(self):
        return ()              # vrati mnozinu alebo postupnost vrcholov

    def hrana(self, v1, v2):
        return None            # vrati retazec alebo None

    def typy_ovocia(self):
        return ()              # vrati mnozinu alebo postupnost mien ovocia

    def start(self, v1):
        return []              # vrati postupnost vrcholov cesty alebo []
```

kde

- metóda `__init__(self, meno_suboru)`: prečíta súbor a vytvorí z neho neorientovaný ohodnotený graf; zvoľte si ľubovoľnú reprezentáciu grafu, pričom si môžete zadefinovať aj vlastné podtriedy, napr. podtriedu `Vrchol`;
- metóda `vrcholy(self)`: vráti zoznam mien všetkých vrcholov grafu ako množinu alebo postupnosť (`list` alebo `tuple`);
- metóda `typy_ovocia(self)`: vráti zoznam mien všetkých druhov ovocia, ktoré sa nachádzajú v grafe; tento zoznam metóda vráti ako množinu alebo postupnosť (`list` alebo `tuple`);

- metóda `hrana(self, v1, v2)`: vráti hodnotu na hrane z vrcholu `v1` do vrcholu `v2`; ak takáto hrana neexistuje, metóda vráti `None`, inak vráti meno ovocia, resp. prázdny reťazec (ak na hrane nie je žiadne ovocie);
- metóda `start(self, v1)`: pomocou backtrackingu nájde ľubovoľnú cestu, ktorá prejde cez každý typ ovocia práve raz a po žiadnej hrane grafu neprejde viackrát (výsledok bude typu `list`); ak takáto cesta neexistuje, metóda vráti `[ ]`.

Napr. pre takéto zadanie grafu:

```
5 d 6
1 4
4 a 2
3 a 2
1 c 2
3 4
4 5
1 3
5 b 3
```

tento test:

```
if __name__ == '__main__':
    z = Zahradka('subor1.txt')
    for v1, v2 in ('3','2'), ('2','3'), ('5','4'), ('2','6'), ('5','1'):
        print(f'hrana({v1!r}, {v2!r}) = {z.hrana(v1, v2)!r}')
    print('typy ovocia =', z.typy_ovocia())
    for v1 in sorted(z.vrcholy()):
        print(f'trasa z {v1!r}:', z.start(v1))
```

vypíše napr.:

```
hrana('3', '2') = 'a'
hrana('2', '3') = 'a'
hrana('5', '4') = ''
hrana('2', '6') = None
hrana('5', '1') = None
typy ovocia = {'c', 'b', 'a', 'd'}
trasa z '1': ['1', '4', '2', '1', '3', '5', '6']
trasa z '2': []
trasa z '3': ['3', '2', '1', '4', '3', '5', '6']
trasa z '4': ['4', '1', '2', '4', '3', '5', '6']
trasa z '5': ['5', '4', '1', '2', '4', '3', '5', '6']
trasa z '6': ['6', '5', '4', '1', '2', '4', '3', '5']
```

Uvedomte si, že ak existuje viac ciest štartujúcich z nejakého zadaného vrcholu, backtracking môže vrátiť ľubovoľnú z nich.

Z úlohového servera L.I.S.T. si stiahnite kostru programu `skuska.py`. Pozrite si testovacie dáta v súboroch `'subor1.txt'`, `'subor2.txt'`, `'subor3.txt'`, ..., ktoré bude používať testovač. Aby ste mohli spúšťať skúškové testy, program uložte do súboru `skuska.py`. Riešenie (bez dátových súborov) odovzdajte na úlohový server https://list.fmph.uniba.sk/.

Testy postupne na všetkých testovacích súboroch preverujú vlastnosti vašich algoritmov, pri prvej chybe sa testovanie s daným súborom preruší a ďalšie časti sa netestujú:

- 42% bodov za vytvorenie grafu
- 58% bodov za algoritmus hľadania cesty v grafe

Praktická časť končí o 11:00 a skúška pokračuje od 12:00 vyhodnotením v kancelárii **m162**.

37.21 Zadanie skúšky z 6.6.2018 - Turistická kancelária

Turistická kancelária organizuje prehliadky po starom meste a maximálne sa snaží vychádzať v ústrety požiadavkám turistov. Turisti si môžu vybrať ľubovoľné pamiatky a sprievodca s nimi prejde zvolenú prehliadku, pričom kancelária si účtuje jednotný poplatok za každú navštívenú pamiatku. Majiteľ kancelárie je zvedavý, aké rôzne prehliadky sa dajú zostaviť, ak prehliadka začne aj skončí na tom istom mieste a prejde cez presne zadaný počet pamiatok.

Mapu starého mesta máme uloženú v tvare **neorientovaného grafu**, v ktorom vrcholy reprezentujú pamiatky a hrany sú ulice. Program pre zadaný štartový vrchol a dĺžku trasy nájde ľubovoľný cyklus. Cyklom je taká cesta v grafe, ktorá začína aj končí v tom istom vrchole a pritom prechádza len cez navzájom rôzne vrcholy.

Graf je zadaný v textovom súbore, ktorý má takúto štruktúru:

- každý riadok popisuje jeden vrchol a skladá sa z jednej alebo viacerých dvojíc celých čísel;
- prvá dvojica čísel označujú súradnice pamiatky na mape (bude to pre nás meno vrcholu);
- za týmito súradnicami nasledujú súradnice ďalších pamiatok, ktoré sú spojené s danou pamiatkou ulicou;
- graf je neorientovaný, preto, ak má pamiatka `A` susednú pamiatku `B`, tak potom aj `B` susedí s `A` – možno to bude treba do grafu doplniť.

Zadefinujte triedu `Graf`:

```
class Graf(object):
    class Vrchol:
        def __init__(self):
            self.sus = set()

    def __init__(self, meno_suboru):
        self.g = {}
        ...

    def najdi(self, start, dlzka):
        ...
        return set()
```

kde

- vnorená definícia triedy `Vrchol` má atribút `sus`, ktorý bude obsahovať množinu (typ `set`) všetkých susediacich vrcholov (teda ich mien), to znamená, že `sus` je množina dvojíc (`tuple`) celých čísel;
- metóda __init__(self, meno_suboru): prečíta súbor a vytvorí z neho graf: atribút `g` (typu `dict`) bude ako hodnoty obsahovať všetky vrcholy grafu, teda objekty typu `Vrchol`; kľúčmi budú polohy vrcholov na mape, teda dvojice (`tuple`) celých čísel;
- metóda najdi(self, start, dlzka): pomocou backtrackingu nájde ľubovoľný cyklus zadanej dĺžky, ktorý začína (a končí) v zadanom vrchole start; metóda vráti cyklus ako postupnosť (`tuple` alebo `list`) mien vrcholov (dvojíc čísel); ak neexistuje žiaden cyklus, ktorý by spĺňal tieto podmienky, metóda vráti `None`; testovač zavolá túto metódu viac krát s rôznymi štartovými vrcholmi a aj dĺžkami.

Napr. pre takéto zadanie grafu:

```
(70, 10) (40, 10) (70, 40)
(10, 10) (10, 40) (40, 10)
(10, 40) (10, 10) (40, 40)
(40, 10)
(40, 40) (10, 40) (40, 10) (70, 40)
(70, 40) (70, 10) (40, 40)
```

tento test:

```
if __name__ == '__main__':
    g = Graf('subor1.txt')
    for key, value in g.g.items():
        print('vrchol =', key, 'sus =', value.sus)
    print(g.najdi((70, 10), 3))
    print(g.najdi((10, 40), 4))
    print(g.najdi((40, 40), 4))
```

vypíše napríklad:

```
vrchol = (10, 40) sus = {(40, 40), (10, 10)}
vrchol = (40, 40) sus = {(10, 40), (70, 40), (40, 10)}
vrchol = (70, 40) sus = {(70, 10), (40, 40)}
vrchol = (10, 10) sus = {(10, 40), (40, 10)}
vrchol = (70, 10) sus = {(70, 40), (40, 10)}
vrchol = (40, 10) sus = {(70, 10), (40, 40), (10, 10)}
None
[(10, 40), (40, 40), (40, 10), (10, 10), (10, 40)]
[(40, 40), (10, 40), (10, 10), (40, 10), (40, 40)]
```

Z úlohového servera L.I.S.T. si stiahnite kostru programu `skuska.py`. Pozrite si testovacie dáta v súboroch `'subor1.txt'`, `'subor2.txt'`, `'subor3.txt'`, ..., ktoré bude používať testovač. Aby ste mohli spúšťať skúškové testy, program uložte do súboru `skuska.py`. Riešenie (bez dátových súborov) odovzdajte na úlohový server https://list.fmph.uniba.sk/.

Testy postupne na všetkých testovacích súboroch preverujú vlastnosti vašich algoritmov, pri prvej chybe sa testovanie s daným súborom preruší a ďalšie časti sa netestujú:

- 40% bodov za vytvorenie grafu
- 60% bodov za algoritmus hľadania cyklu v grafe

Praktická časť končí o 11:00 a skúška pokračuje od 12:00 vyhodnotením v kancelárii **m162**.

37.22 Zadanie skúšky z 18.6.2018 - Skauti a bobríci

Členovia skautského oddielu majú veľký deň - počas preteku musia nazbierať čo najviac rôznych odznakov zdatnosti (tzv. bobríkov). Každý mladý skaut dostal kompletnú mapu (**neorientovaný ohodnotený graf**), v ktorej boli vyznačené kontrolné body (vrcholy grafu) a chodníky (hrany). Na každom chodníku skauti môžu získať celú množinu bobríkov (sú to hodnoty na hranách). Úlohou skauta je prebehnúť po chodníkoch zo štartového miesta do cieľového, pričom cez kontrolné miesta (vrcholy grafu) môže prejsť maximálne raz. Samozrejme, že by bolo dobre, keby na tejto trase získal čo najviac **rôznych** bobríkov.

Napíšte program, ktorý pre danú mapu a štartové a cieľové miesto nájde takú cestu, na ktorej nazbiera čo najväčší počet rôznych bobríkov a zo všetkých takýchto ciest je táto najkratšia.

Mapa je zadaná v textovom súbore, ktorý má takúto štruktúru:

- prvý riadok obsahuje počet vrcholov (menami vrcholov sú celé čísla od 0 po tento počet - 1);
- každý ďalší riadok popisuje jeden chodník (hranu grafu) ako dvojicu vrcholov, na ktorom sa môže nachádzať niekoľko mien bobríkov;
- formátom takéhoto riadku je postupnosť reťazcov oddelených medzerou, pričom prvý a posledný reťazec v riadku sú celé čísla - mená dvoch vrcholov, ktoré tvoria chodník a všetky reťazce medzi týmito dvoma vrcholmi sú mená bobríkov;
- na chodníku sa môže nachádzať žiaden, jeden ale aj viac bobríkov;
- graf je neorientovaný, preto, ak je vrchol `A` spojený chodníkom s `B`, tak potom aj vrchol `B` susedí s vrcholom `A`, zrejme to platí aj pre bobríky – možno to budete musieť do grafu dodefinovať.

Zadefinujte triedu `Mapa`:

```
class Mapa:
    def __init__(self, meno_suboru):
        ...

    def vrcholy(self):
        return set()           # vrati mnozinu vrcholov

    def hrana(self, v1, v2):
        return None            # vrati mnozinu alebo None

    def hladaj(self, start, ciel):
        return None            # vrati postupnost vrcholov cesty alebo None

    def bobriky(self):
        return None            # vrati mnozinu alebo None
```

kde

- metóda `__init__(self, meno_suboru)`: prečíta súbor a vytvorí z neho neorientovaný ohodnotený graf; zvoľte si ľubovoľnú reprezentáciu grafu, pričom si môžete zadefinovať aj vlastné podtriedy, napr. podtriedu `Vrchol`;
- metóda `vrcholy(self)`: vráti množinu mien (celé čísla) všetkých vrcholov grafu;
- metóda `hrana(self, v1, v2)`: vráti hodnotu na hrane z vrcholu `v1` do vrcholu `v2`; ak takáto hrana neexistuje, metóda vráti `None`, inak vráti množinu bobríkov (môže byť aj prázdna);
- metóda `hladaj(self, start, ciel)`: pomocou backtrackingu nájde najkratšiu cestu, ktorá zozbiera čo najviac rôznych bobríkov; prejde cez každý vrchol maximálne raz (výsledok bude typu `list` alebo `tuple`); ak takáto cesta neexistuje, metóda vráti `None`;

- metóda `bobriky(self)`: vráti množinu všetkých zozbieraných bobríkov pri poslednej hľadanej ceste; ak sa cesta nenašla, metóda vráti `None`.

Napr. pre takéto zadanie grafu:

```
9
0 b a 1
1 2
3 b 4
4 a 5
6 ab 7
7 8
0 a 3
3 a c 6
1 4
4 c a 7
2 a b 5
5 8
```

tento test:

```
if __name__ == '__main__':
    m = Mapa('subor1.txt')
    vv = m.vrcholy()
    print(f'vrcholy = {vv}')
    for v1, v2 in (1, 0), (0, 3), (2, 6):
        print(f'm[{v1}][{v2}] = {m.hrana(v1, v2)}')
    for v1, v2 in (0, 8), (1, 4), (2, 6):
        print(f'cesta z {v1} do {v2} = {m.hladaj(v1, v2)}, bobriky = {m.bobriky()}')
```

vypíše napr.:

```
vrcholy = {0, 1, 2, 3, 4, 5, 6, 7, 8}
m[1][0] = {'b', 'a'}
m[0][3] = {'a'}
m[2][6] = None
cesta z 0 do 8 = [0, 1, 4, 3, 6, 7, 8], bobriky = {'ab', 'b', 'a', 'c'}
cesta z 1 do 4 = [1, 0, 3, 6, 7, 4], bobriky = {'ab', 'b', 'a', 'c'}
cesta z 2 do 6 = [2, 5, 4, 7, 6], bobriky = {'ab', 'b', 'a', 'c'}
```

Uvedomte si, že ak existuje viac najkratších ciest s maximálnym počtom bobríkov, backtracking môže vrátiť ľubovoľnú z nich.

Z úlohového servera L.I.S.T. si stiahnite kostru programu `skuska.py`. Pozrite si testovacie dáta v súboroch `'subor1.txt'`, `'subor2.txt'`, `'subor3.txt'`, ..., ktoré bude používať testovač. Aby ste mohli spúšťať skúškové testy, program uložte do súboru `skuska.py`. Riešenie (bez dátových súborov) odovzdajte na úlohový server https://list.fmph.uniba.sk/.

Testy postupne na všetkých testovacích súboroch preverujú vlastnosti vašich algoritmov, pri prvej chybe sa testovanie s daným súborom preruší a ďalšie časti sa netestujú:

- 35% bodov za vytvorenie grafu
- 65% bodov za algoritmus hľadania cesty v grafe

Praktická časť končí o 11:00 a skúška pokračuje od 12:00 vyhodnotením v kancelárii **m162**.

37.23 Zadanie skúšky z 27.6.2018 - Výstavisko

Prišli sme na veľtrh, pričom na výstavisku každý vystavovateľ rozdáva nejaký reklamný darček, napr. pero, prívesok, odznak, atď. Radi by sme nazbierali čo najviac **rôznych** reklamných darčekov. Dostali sme kompletnú mapu (**neorientovaný ohodnotený graf**), v ktorej boli vyznačené križovatky (vrcholy grafu) a uličky s vystavovateľmi s darčekmi (hrany grafu). V každej uličke preto môžeme získať celú množinu darčekov (sú to hodnoty na hranách grafu). Našou úlohou je prejsť po uličkách výstaviska zo štartového miesta do cieľového, pričom cez križovatky (vrcholy grafu) môžeme prejsť maximálne raz. Samozrejme, že by bolo dobre, keby sme na tejto trase získali čo najviac **rôznych** darčekov.

Napíšte program, ktorý pre danú mapu a štartové a cieľové miesto nájde takú cestu (postupnosť vrcholov), na ktorej nazbiera čo najväčší počet rôznych darčekov a zo všetkých takýchto ciest je táto najkratšia.

Mapa je zadaná v textovom súbore, ktorý má takúto štruktúru:

- prvý riadok obsahuje počet vrcholov (menami vrcholov sú celé čísla od 0 po tento počet - 1);
- každý ďalší riadok popisuje jednu uličku (hranu grafu) ako dvojicu vrcholov, na ktorej sa môže nachádzať niekoľko mien darčekov;
- formátom takéhoto riadku je postupnosť reťazcov oddelených medzerou, pričom prvý a druhý reťazec v riadku sú celé čísla - mená dvoch vrcholov, ktoré tvoria uličku a všetky zvyšné reťazce v riadku sú mená darčekov;
- na uličke sa môže nachádzať žiaden, jeden ale aj viac darčekov;
- graf je neorientovaný, preto, ak je vrchol `A` spojený uličkou s `B`, tak potom aj vrchol `B` susedí s vrcholom `A`, zrejme to platí aj pre darčeky – možno to budete musieť do grafu dodefinovať.

Zadefinujte triedu `Mapka`:

```
class Mapka:
    def __init__(self, meno_suboru):
        ...

    def vrcholy(self):
        return set()           # vrati mnozinu vrcholov

    def hrana(self, v1, v2):
        return None            # vrati mnozinu alebo None

    def hladaj(self, start, ciel):
        return []              # vrati postupnost vrcholov cesty

    def darceky(self):
        return set()           # vrati mnozinu
```

kde

- metóda `__init__(self, meno_suboru)`: prečíta súbor a vytvorí z neho neorientovaný ohodnotený graf; zvoľte si ľubovoľnú reprezentáciu grafu, pričom si môžete zadefinovať aj vlastné podtriedy, napr. podtriedu `Vrchol`;
- metóda `vrcholy(self)`: vráti množinu mien (celé čísla) všetkých vrcholov grafu;
- metóda `hrana(self, v1, v2)`: vráti hodnotu na hrane z vrcholu `v1` do vrcholu `v2`; ak takáto hrana neexistuje, metóda vráti `None`, inak vráti množinu darčekov (môže byť aj prázdna);
- metóda `hladaj(self, start, ciel)`: pomocou backtrackingu nájde najkratšiu cestu, ktorá zozbiera čo najviac rôznych darčekov; prejde cez každý vrchol maximálne raz (výsledná postupnosť bude typu `list` alebo `tuple`); ak takáto cesta neexistuje, metóda vráti prázdnu postupnosť;

- metóda `darceky(self)`: vráti množinu všetkých zozbieraných darčekov pri poslednej hľadanej ceste; ak sa cesta nenašla, metóda vráti prázdnu množinu.

Napr. pre takéto zadanie grafu:

```
9
0 1 b a
1 2
3 4 b
4 5 a
6 7 ab
7 8
0 3 a
3 6 a c
1 4
4 7 c a
2 5 a b
5 8
```

tento test:

```
if __name__ == '__main__':
    m = Mapka('subor1.txt')
    vv = m.vrcholy()
    print(f'vrcholy = {vv}')
    for v1, v2 in (1, 0), (0, 3), (2, 6):
        print(f'm[{v1}][{v2}] = {m.hrana(v1, v2)}')
    for v1, v2 in (0, 8), (1, 4), (2, 6):
        print(f'cesta z {v1} do {v2} = {m.hladaj(v1, v2)}, darceky = {m.darceky()}')
```

vypíše napr.:

```
vrcholy = {0, 1, 2, 3, 4, 5, 6, 7, 8}
m[1][0] = {'b', 'a'}
m[0][3] = {'a'}
m[2][6] = None
cesta z 0 do 8 = [0, 1, 4, 3, 6, 7, 8], darceky = {'ab', 'b', 'a', 'c'}
cesta z 1 do 4 = [1, 0, 3, 6, 7, 4], darceky = {'ab', 'b', 'a', 'c'}
cesta z 2 do 6 = [2, 5, 4, 7, 6], darceky = {'ab', 'b', 'a', 'c'}
```

Uvedomte si, že ak existuje viac najkratších ciest s maximálnym počtom darčekov, backtracking môže vrátiť ľubovoľnú z nich.

Z úlohového servera L.I.S.T. si stiahnite kostru programu `skuska.py`. Pozrite si testovacie dáta v súboroch `'subor1.txt'`, `'subor2.txt'`, `'subor3.txt'`, ..., ktoré bude používať testovač. Aby ste mohli spúšťať skúškové testy, program uložte do súboru `skuska.py`. Riešenie (bez dátových súborov) odovzdajte na úlohový server https://list.fmph.uniba.sk/.

Testy postupne na všetkých testovacích súboroch preverujú vlastnosti vašich algoritmov, pri prvej chybe sa testovanie s daným súborom preruší a ďalšie časti sa netestujú:

- 35% bodov za vytvorenie grafu
- 65% bodov za algoritmus hľadania cesty v grafe

Praktická časť končí o 11:00 a skúška pokračuje od 12:00 vyhodnotením v kancelárii **m162**.

37.24 1. tréningové zadanie - skúška z 16.1.2017 - Harry Potter a preskakovadlá

Na čarodejníckom hrade Harryho Pottera (hracia plocha v tvare v štvorcovej siete) sa na niektorých políčkach nachádzajú magické **preskakovadlá**. Keď sa Harry rozbehne a z niektorého smeru na takéto políčko stúpi, tak ho to môže o niekoľko políčok prehodiť - ale len v danom smere. Každé preskakovadlo má na sebe číslo, ktoré označuje, o koľko políčok Harry preletí. Ak by ale v danom smere Harry vyletel z plochy, alebo sa objavil v stene, preskakovadlo nemá v tomto smere žiaden účinok a teda je to ako obyčajné políčko bez čísla. Ak pri preskoku skočíme na ďalšie preskakovadlo, tak aj ono nás môže presunúť na ďalšie políčko (samozrejme v tom istom smere ale so svojou dĺžkou skoku), atď.

Harry pri prechádzaní hradu (pomocou šípok `'l'` vľavo, `'p'` vpravo, `'h'` hore, `'d'` dole), okrem preskakovania zbiera aj všetky magické predmety, ktoré nájde na svojej ceste (stúpi na toto políčko). Aj pri presune pomocou preskakovadla môže skočiť na predmet a ten samozrejme musí tiež zdvihnúť.

Cieľom v programu je odsledovať, kam sa Harry po zadanej postupnosti šípok dostane a koľko predmetov pritom zozbiera.

Textový súbor popisuje hraciu plochu (hrad) ako štvorcovú sieť (všetky riadky sú rovnako dlhé). Znaky v súbore majú tento význam:

- čísla od `'1'` do `'9'` sú preskakovadlá
- `'o'` je predmet
- `'x'` je stena
- `'.'` je voľné políčko

Vašou úlohou je prečítať textový súbor s popisom hradu a odsimulovať zadanú postupnosť stláčaných šípok. Samotný pohyb hráča sa bude riadiť podľa týchto pravidiel:

- šípka na voľné políčko (prípadne s predmetom) presunie hráča (a zdvihne predmet)
- šípka na políčko so stenou, resp. za okraj plochy, sa ignoruje (Harry stojí na mieste)
- šípka na políčko s preskakovadlom (jedno z `'1'` až `'9'`) spôsobí automatický presun na ďalšie políčko daným smerom, ak je týmto smerom opäť políčko s preskakovadlom, znovu sa to opakuje; ak je týmto smerom stena alebo okraj plochy, hráč zastane na tomto políčku s číslom

Riešenie zapíšte do triedy `Program` s týmito metódami:

```
class Program:
    def __init__(self, meno_suboru):
        ...

    def __str__(self):
        ...

    def start(self, riadok, stlpec):
        ...

    def rob(self, postupnost):
        ...
```

kde

- metóda `__str__()`: vráti znakový reťazec, ktorý reprezentuje momentálny stav plochy aj s pozíciou hráča (znak `'h'`); medzi riadkami je znak `'\n'`
- metóda `start(riadok,stlpec)`: nastaví štartovú pozíciu Harryho - toto políčko je určite voľné

- metóda `rob(postupnost)`: kde parameter `postupnost` je znakový reťazec s postupnosťou príkazov, teda stláčaných šípok (znaky `'l'`, `'p'`, `'h'`, `'d'`) – hráč sa v ploche pohybuje podľa týchto zadaných príkazov; ak sa daným smerom nemôže pohnúť (okraj plochy, stena), tento konkrétny príkaz sa ignoruje, ak príde na políčko s preskakovadlom, tak, ak sa dá, zrealizuje tento presun, inak pokračuje ďalším príkazom; metóda vracia počet zdvihnutých predmetov; každé volanie tejto metódy vráti len ten počet predmetov, ktoré sa pozbierali počas tohto volania

Napr. pre súbor

```
xxxxxxxxxx
x.......ox
x.xxxxxxxx
xox.2....x
xxx..xxx.x
xo..xxo..x
xxxxxxxxxx
```

takýto test:

```
if __name__ == '__main__':
    p = Program('subor2.txt')
    p.start(3,5)
    print(p)
    print(p.rob('ldhpppp'))
    print(p.rob('llllllldd'))
    print(p)
    p.start(3,5)
    print(p.rob('llddllpphhpppddll'))
    print(p)
```

vypíše:

```
xxxxxxxxxx
x.......ox
x.xxxxxxxx
xox.2h...x
xxx..xxx.x
xo..xxo..x
xxxxxxxxxx
1
1
xxxxxxxxxx
x........x
x.xxxxxxxx
xhx.2....x
xxx..xxx.x
xo..xxo..x
xxxxxxxxxx
2
xxxxxxxxxx
x........x
x.xxxxxxxx
x.x.2....x
xxx..xxx.x
x...xxh..x
xxxxxxxxxx
```

Z úlohového servera L.I.S.T. si stiahnite kostru programu `skuska.py`. Pozrite si testovacie dáta v súboroch `'subor1.txt'`, `'subor2.txt'`, `'subor3.txt'`,..., ktoré bude používať testovač.

Aby ste mohli spúšťať skúškové testy, program uložte do súboru `skuska.py`. Riešenie (bez dátových súborov) odovzdajte na úlohový server https://list.fmph.uniba.sk/.

Praktická časť končí o 11:00 a skúška ďalej pokračuje od 12:00 vyhodnotením v kancelárii **m162**.

37.25 2. tréningové zadanie - skúška z 18.1.2017 - Sokoban

Sokoban je hra pre jedného hráča, ktorého cieľom je presunúť v štvorcovej sieti všetky škatule na vyznačené cieľové políčka. Hráč pri prechádzaní hracej plochy (pomocou šípok `'l'` vľavo, `'p'` vpravo, `'h'` hore, `'d'` dole) môže škatuľu, ktorá leží pred ním, potlačiť v danom smere na políčko za škatuľou, ale len vtedy, keď je toto políčko prázdne (nie je tam ani stena ani iná škatuľa).

Cieľom v programu je odkontrolovať, kam sa hráč po zadanej postupnosti šípok dostane, ktoré škatule kam presunie a na koľkých cieľových políčkach sa pritom nachádzajú škatule a na koľkých nie.

Textový súbor popisuje hraciu plochu ako štvorcovú sieť (riadky môžu byť rôzne dlhé). Riadky v prvej časti súboru obsahujú priamo riadky hracej plochy. Ďalej za jedným prázdnym riadkom nasledujú pozície cieľových políčok ako dvojice celých čísel. Znaky popisujúce hraciu plochu majú tento význam:

- `'*'` je pozícia hráča
- `'O'` je škatuľa
- `'M'` je stena
- `' '` je voľné políčko

Zrejme na niektorých zadaných cieľových políčkach sa na začiatku hry môže nachádzať nielen hráč ale aj škatuľa.

Vašou úlohou je prečítať textový súbor s popisom plochy a odsimulovať zadanú postupnosť stláčaných šípok. Samotný pohyb hráča sa bude riadiť podľa týchto pravidiel:

- šípka na voľné políčko (môže byť cieľové) presunie hráča
- šípka na políčko so stenou sa ignoruje (hráč zostane stáť na mieste); môžete počítať s tým, že celá hracia plocha je obkolesená stenami a hráč sa nemôže dostať za okraj plochy
- šípka na políčko so škatuľou bude mať účinok (posunie túto škatuľu) jedine vtedy, keď je za škatuľou v danom smere voľné políčko, inak sa šípka ignoruje; zrejme posunutie škatule znamená, že škatuľa sa posunie na voľné miesto a hráč sa presunie na pôvodnú pozíciu škatule

Riešenie zapíšte do triedy `Sokoban` s týmito metódami:

```
class Sokoban:
    def __init__(self, meno_suboru):
        ...

    def __str__(self):
        return ''

    def rob(self, postupnost):
        return 0

    def kontrola(self):      # na kolkych cielovych policKach nie je skatula
        return None
```

kde

- metóda `__str__()`: vráti znakový reťazec, ktorý reprezentuje momentálny stav hracej plochy aj s pozíciou hráča (znak `'*'`), všetkých škatúľ (znak `'O'`) a všetkých cieľových políčok (znak `'+'`); medzi riadkami je znak `'\n'`;
- metóda `rob(postupnost)`: kde parameter `postupnost` je znakový reťazec s postupnosťou príkazov, teda stláčaných šípok (znaky `'l'`, `'p'`, `'h'`, `'d'`) – hráč sa v ploche pohybuje podľa týchto zadaných príkazov; ak sa daným smerom nemôže pohnúť (stena alebo prekážka za tlačenou škatuľou), tento konkrétny príkaz sa ignoruje; metóda vráti počet naozaj vykonaných krokov

- metóda `kontrola()`: vráti dvojicu (typ `tuple`) celých čísel: (na koľkých cieľových políčkach je škatuľa, na koľkých cieľových políčkach nie je škatuľa) – zrejme v súčte tieto dve čísla dávajú celkový počet cieľových políčok

Napr. pre súbor

```
MMMMM
M M
M MMMMM
M O M
M *O M
MMMMMMMMM

2 2
4 4
```

takýto test:

```
if __name__ == '__main__':
    s = Sokoban('subor1.txt')
    print(s)
    print('kontrola', s.kontrola())
    print('rob', s.rob('hddllpp'))
    print(s)
    print('kontrola', s.kontrola())
```

vypíše:

```
MMMMM
M M
M + MMMMM
M O M
M *O+ M
MMMMMMMMM
kontrola (0, 2)
rob 5
MMMMM
M M
M O MMMMM
M M
M *O M
MMMMMMMMM
kontrola (2, 0)
```

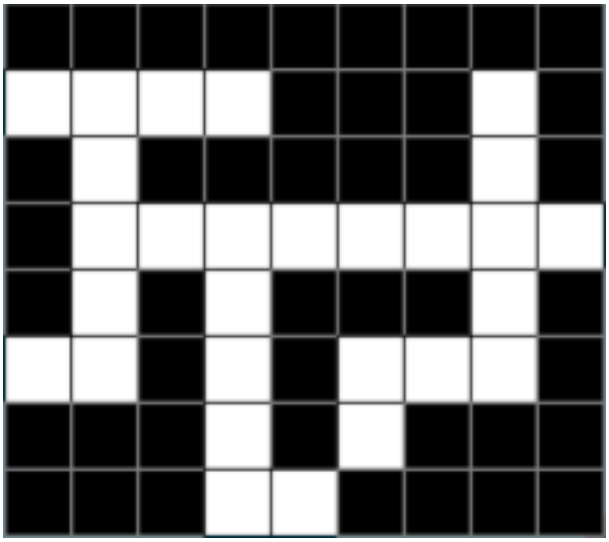
Z úlohového servera L.I.S.T. si stiahnite kostru programu `skuska.py`. Pozrite si testovacie dáta v súboroch `'subor1.txt'`, `'subor2.txt'`, `'subor3.txt'`, ..., ktoré bude používať testovač.

Aby ste mohli spúšťať skúškové testy, program uložte do súboru `skuska.py`. Riešenie (len súbor `skuska.py` bez dátových súborov) odovzdajte na úlohový server https://list.fmph.uniba.sk/.

Praktická časť končí o 11:00 a skúška ďalej pokračuje od 12:00 vyhodnotením v kancelárii **m162**.

37.26 3. tréningové zadanie - skúška z 6.2.2017 - Kamery v Kocúrkove

Na mape Kocúrkova môžete vidieť bielou farbou označené ulice:



Mestská rada chce do mesta umiestniť kamery, ktoré budú sledovať ulice všetkými smermi ľubovoľne ďaleko, ale len vo vodorovných a zvislých smeroch (kamera cez domy nevidí). Zrejme majú v pláne rozmiestniť kamery tak, aby mohli monitorovať všetky ulice mesta. Preto pre nich pripravíme aplikáciu, pomocou ktorej budú môcť experimentovať s rozmiestňovaním rôzneho počtu kamier.

Táto aplikácia by mala fungovať takto:

- na začiatku je mapa prázdna - nie je tu žiadna kamera, len ulice a domy;
- používateľ našej aplikácie môže na ľubovoľné miesta mapy umiestňovať nové kamery (každá kamera má svoj číselný identifikátor), prípadne už predtým umiestnenú kameru presunúť na ľubovoľné iné miesto, alebo ju z mapy odstrániť;
- používateľ si hocikedy môže nechať zobraziť mapu aj s vyznačenými monitorovanými políčkami mapy;
- tiež dostáva informáciu o tom, koľko políčok mapy je už monitorovaných a koľko ešte nie.

Textový súbor popisuje hraciu plochu ako štvorcovú sieť, v ktorej sú všetky riadky rovnako dlhé. Každý riadok súboru popisuje jeden riadok štvorcovej siete: obsahuje postupnosť celých čísel (oddelených medzerami), pričom prvé číslo v postupnosti označuje počet plných štvorčekov (domov) na začiatku riadku, za tým nasleduje počet prázdnych políčok (ktoré označujú ulice), každé ďalšie číslo v riadku označuje buď počet plných alebo prázdnych políčok. Zrejme súčet všetkých čísel v každom riadku musí byť rovnaký a rovná sa šírke hracej plochy. Počítajte s tým, že medzi týmito číslami môžu byť aj nuly.

Riešenie zapíšte do triedy `Kocurkovo` s týmito metódami:

```
class Kocurkovo:

    def __init__(self, meno_suboru):
        ...

    def __str__(self):
        return ''

    def kamera(self, cislo, pozicia):
        return 0
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def kontrola(self):
        return 0, 0

    def zoznam(self):
        return ()
```

kde

- metóda `__str__()` vráti znakový reťazec, ktorý reprezentuje kompletnú mapu: v tejto mape je políčko s domom vyjadrené znakom `'M'`, monitorované políčko (aspoň jednou kamerou) znakom `'+'`, a nemonitorovanú ulicu znakom ‚.‘; medzi riadkami mapy je znak `'\n'`;
- metóda `kamera(cislo, pozicia)` umiestni kameru s daným číslom (číslo kamery je ľubovoľné celé číslo) na zadanú pozíciu (dvojicu čísel pre riadok a stĺpec, pričom číslujeme od 0); ak už bola predtým táto kamera v ploche, tak sa takto premiestni na túto novú pozíciu; metóda vráti počet políčok, ktoré táto kamera monitoruje (bez ohľadu na iné kamery); ak je umiestnená na políčko mimo ulice (teda dom, ktorý je na mape označený ako `'M'`), vráti hodnotu `0`; ak je umiestnená mimo súradníc mapy, je takto z mapy odstránená a metóda vráti hodnotu `None`
- metóda `kontrola()` vráti dvojicu (typ `tuple`): celkový počet monitorovaných políčok a počet políčok, ktoré žiadna kamera nemonitoruje;
- metóda `zoznam()` vráti n-ticu (typ `tuple`) informácií o kamerách; každá kamera je v n-tici reprezentovaná dvojicou (číslo kamery, pozícia), kde pozícia je dvojica (riadok, stĺpec).

Napr. pre súbor (popisuje mapu z úvodu):

```
9
0 4 3 1 1
1 1 5 1 1
1 8
1 1 1 1 3 1 1
0 2 1 1 1 3 1
3 1 1 1 3
3 2 4
```

takýto test:

```
if __name__ == '__main__':
    k = Kocurkovo('subor1.txt')
    print(k)
    print('kontrola =', *k.kontrola())
    print('kamera', k.kamera(1, (3, 3)))
    print(k)
    print('kontrola =', *k.kontrola())
    print('kamera', k.kamera(3, (5, 7)))
    print(k)
    print('kontrola =', *k.kontrola())
    print('zoznam:', *k.zoznam())
```

vypíše:

```
MMMMMMMMM
....MMM.M
M.MMMMM.M
M........
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
M.M.MMM.M
..M.M...M
MMM.M.MMM
MMM..MMMM
kontrola = 0 28
kamera 12
MMMMMMMMM
....MMM.M
M.MMMMM.M
M++++++++
M.M+MMM.M
..M+M...M
MMM+M.MMM
MMM+.MMMM
kontrola = 12 16
kamera 7
MMMMMMMMM
....MMM+M
M.MMMMM+M
M++++++++
M.M+MMM+M
..M+M+++M
MMM+M.MMM
MMM+.MMMM
kontrola = 18 10
zoznam: (1, (3, 3)) (3, (5, 7))
```

Z úlohového servera L.I.S.T. si stiahnite kostru programu `skuska.py`. Pozrite si testovacie dáta v súboroch `'subor1.txt'`, `'subor2.txt'`, `'subor3.txt'`, ..., ktoré bude používať testovač.

Aby ste mohli spúšťať skúškové testy, program uložte do súboru `skuska.py`. Riešenie (len súbor `skuska.py` bez dátových súborov) odovzdajte na úlohový server https://list.fmph.uniba.sk/.

Praktická časť končí o 11:00 a skúška ďalej pokračuje od 12:00 vyhodnotením v kancelárii **m162**.

37.27 1. tréningové zadanie - skúška z 5.6.2017

V Kocúrkove majú veľmi zaujímavo organizované všetky pozemky a chodníky:

- každý pozemok je dookola obkolesený chodníkom
- ak nejaké dva pozemky susedia, majú spoločnú nejakú časť chodníka
- chodníky okolo pozemkov (teda hranice pozemkov) sú definované pomocou postupnosti názvov hraničných kameňov – niekto si dal robou a všetkým hraničným kameňom v Kocúrkove pridelil nejaké názvy (väčšinou ako nejaké písmeno a číslo), napr.
  - ak má nejaký pozemok hranicu popísanú názvami kameňov `['a3', 'b1', 'a2']` a nejaký iný názvami `['a2', 'a1', 'a3']`, oba pozemky sú trojuholníkového tvaru a majú spoločnú časť chodníka `'a2'-'a3'`
  - zrejme takto popísané hranice pozemkov pomocou názvov hraničných kameňov sú uzavreté oblasti a teda spojený chodníkom je aj prvý a posledný kameň
- počítajte s tým, že žiadna časť chodníka neprechádza cez vnútro nejakého pozemku – všetky chodníky sú len okolo pozemkov

Ak sa niekto prechádza po chodníkoch Kocúrkova, vždy prechádza okolo aspoň jedného pozemku a niekedy súčasne okolo dvoch rôznych. Pri dlhších prechádzkach niekedy takto môže prejsť aj okolo všetkých pozemkov.

Vašou úlohou bude navrhnúť takú prechádzku z nejakého štartového miesta, pomocou ktorej prejdeme okolo všetkých dopredu určených pozemkov. Samotné rozloženie pozemkov, teda popis chodníkov, prečítate z textového súboru. Ak existuje viac rôznych prechádzok, pomocou ktorých sa dá prejsť okolo určených pozemkov, vyberiete najkratšiu z nich. Dĺžka prechádzky je v tomto prípade definovaná ako počet hraničných kameňov, popri ktorých sa prechádza.

Na navrhnutú prechádzku máme ešte tieto dôležité požiadavky:

- popri žiadnom hraničnom kameni nesmieme prejsť viac ako raz
- počas prechádzky akceptujeme návštevu nejakého pozemku len vtedy, keď prejdeme po nejakej časti chodníka okolo neho, teda nestačí prejsť len popri jednom hraničnom kameni daného pozemku

Z pohľadu dátovej štruktúry graf:

- hraničné kamene sú vrcholy grafu
- časti chodníkov, teda spojnice medzi kameňmi (vrcholmi) sú hrany grafu
- každá hrana prechádza okolo nejakého jedného alebo možno dvoch pozemkov – táto informácia (množina pozemkov) je hodnotou (váhou) hrany

Riešenie zapíšte do triedy `Graf` s týmito metódami:

```
class Graf:
    def __init__(self, meno_suboru):
        ...

    def pozemky(self):
        return set()

    def vrcholy(self):
        return set()

    def hrana(self, v1, v2):
        return None
```

(pokračuje na ďalšej strane)

(pokračovanie z predošlej strany)

```
    def ries(self, v1, ciel):
        return []
```

Kde metódy:

- `__init__(meno_suboru)` prečíta súbor a vytvorí z neho reprezentáciu **neorientovaného ohodnoteného** grafu
- `hrana(v1, v2)` vráti **hodnotu hrany**: buď je to jedno alebo dvoj prvková množina mien pozemkov, alebo `None`, ak neexistuje hrana medzi týmito dvoma vrcholmi
- `vrcholy()` vráti množinu názvov všetkých vrcholov (hraničných kameňov)
- `pozemky()` vráti množinu názvov všetkých pozemkov
- `ries(v1, ciel)` pomocou backtrackingu nájde najkratšiu cestu z vrcholu `v1`, ktorá prejde okolo všetkých pozemkov špecifikovaných v parametri `ciel` (typu `set`) – metóda vráti buď prázdny zoznam (nenašiel žiadnu cestu), alebo zoznam (typ `list`) s nájdenou cestou, teda postupnosť názvov vrcholov (ak je takých viac, vráti ľubovoľnú z nich)

Môžete si dodefinovať aj ďalšie pomocné metódy. Ak chcete definovať nejaké ďalšie pomocné triedy, zapíšte ich ako vnorené do triedy `Graf`.

Textový súbor obsahuje popis grafu v tvare:

- každý riadok obsahuje popis jedného pozemku: je to meno pozemku, za ktorým nasleduje postupnosť názvov vrcholov na obvode pozemku (zrejme aj prvý a posledný vrchol sú spojené hranou)
- uvedomte si, že v neorientovanom grafe by mala byť každá hrana orientovaná oboma smermi
- môžete predpokladať, že súbor je zadaný korektne

Napr. súbor

```
p1 a3 b1 a2
p2 a1 b2 a3
p3 a1 b3 a2
p4 a2 a1 a3
```

označuje graf so šiestimi vrcholmi a 9 hranami, ktoré definujú 4 pozemky. Všimnite si, že pozemok `p4` susedí s troma zvyšnými pozemkami, pričom tieto tri pozemky majú len jedného suseda – pozemok `p4`.

Program môžete testovať napr. takto:

```
if __name__ == '__main__':
    g = Graf('subor1.txt')
    print('pozemky =', g.pozemky())
    print('vrcholy =', g.vrcholy())
    for v1,v2 in ('a3','b2'),('a2','a1'),('b1','a1'):
        print('hrana({!r},{!r}) = {!r}'.format(v1,v2,g.hrana(v1,v2)))
    print('riesenie =', g.ries('b1',{'p2','p4'}))
```

a pre vyššie uvedený textový súbor by mal vypísať (existuje viac správnych riešení, toto je jedno z nich):

```
pozemky = {'p2', 'p3', 'p1', 'p4'}
vrcholy = {'b3', 'a2', 'a3', 'a1', 'b1', 'b2'}
hrana('a3','b2') = {'p2'}
hrana('a2','a1') = {'p3', 'p4'}
hrana('b1','a1') = None
riesenie = ['b1', 'a3', 'a1']
```

Za vyriešenie skúškovej úlohy môžete získať 60 bodov, pričom:

- **30%** je za vytvorenie grafu zo súboru, t.j. správne fungovanie metód `pozemky()`, `vrcholy()` a `hrana()`
- **30%** je za nájdenie nejakého riešenia, ktoré nemusí mať minimálnu dĺžku
- **30%** za riešenie minimálnej dĺžky
- **10%** za správny výsledok takých zadaní, pre ktoré neexistuje riešenie

Aby ste mohli spúšťať skúškové testy, program uložte do súboru `skuska.py`. Riešenie (bez dátových súborov) odovzdajte na úlohový server https://list.fmph.uniba.sk/.

Skúška pokračuje od 12:00 vyhodnotením v kancelárii **m162**.

37.28 2. tréningové zadanie - skúška z 7.6.2017

Vesmírna stanica má pre posádku pripravených **N** obytných modulov, ktoré sú na začiatku zapečatené. Ak sa posádka do niektorého z týchto modulov nasťahuje, predpokladá sa, že tam bude bývať celý rok a pritom využije všetky pripravené zdroje. Po roku musí posádka tento modul opustiť, lebo tento sa z vesmírnej stanice odpojí ako už vyčerpaný (zahodí sa do vesmíru). Posádka sa teda presťahuje do niektorého ďalšieho modulu a v tomto býva ďalší rok.

Zrejme, takto sa dá na vesmírnej stanici bývať maximálne **N** rokov. Je tu jeden problém: moduly sú navzájom pospájané prepojovacími chodbami a nie z každého modulu sa dá dostať do ľubovoľného ďalšieho. Asi by vznikol veľký problém, keby posádka bývala v module, z ktorého sa po roku nedá prejsť do žiadneho ďalšieho. Uvedomte si, že každý predtým navštívený modul je už od vesmírnej stanice odpojený.

Máme daný súbor s informáciami o prepojení modulov: moduly majú svoje celočíselné kódy a všetky riadky súboru majú tvar:

```
123-1749
```

t.j. modul s prvým kódom **123**, je prepojený s nejakým iným modulom, ktorého kód je **1749**.

Schéma vesmírnej stanice tvorí **neorientovaný neohodnotený** graf, v ktorom hrany reprezentujú chodby a vrcholy sú obytné moduly. Program potom pre zadaný štartový vrchol nájde ľubovoľnú maximálne dlhú trasu, ktorou sa prejde cez najväčší možný počet vrcholov a pritom neprejde cez žiaden vrchol viackrát. Niekedy budeme potrebovať nájsť maximálnu trasu, ktorá obíde zadanú množinu vrcholov (napr. zistili sme, že niektoré moduly sú neobývateľné, lebo boli poškodené letiacimi asteroidmi).

Zadefinujte triedu `VesmirnaStanica`:

```
class VesmirnaStanica:
    def __init__(self, meno_suboru):
        self.m = []   # zoznam mien modulov
        self.g = []   # tabulka susednosti
        ...

    def moduly(self):
        return set()

    def chodba(self, m1, m2):
        return False

    def zisti(self, m1, bez=None):
        return []
```

kde

- metóda `__init__(self, meno_suboru)`: prečíta súbor a vytvorí z neho neorientovaný neohodnotený graf; realizujte ho ako tabuľku susedností (dvojrozmerný zoznam hodnôt `True` alebo `False`), pričom v atribúte `self.m` bude zvolené poradie mien modulov;
- metóda `moduly(self)`: vráti množinu číselných kódov všetkých vrcholov grafu;
- metóda `chodba(self, m1, m2)`: zistí, či medzi modulmi `m1` a `m2` je chodba, t.j. či vrcholy `m1` a `m2` sú spojené hranou; metóda vráti `True` alebo `False`;
- metóda `zisti(self, m1)`: pomocou backtrackingu nájde ľubovoľnú maximálne dlhú trasu, ktorá prejde cez maximálny počet vrcholov práve raz (výsledok bude typu `list`); ak takáto cesta neexistuje, metóda vráti `[]`;
- metóda `zisti(self, m1, bez)`: ďalší parameter `bez` obsahuje množinu vrcholov, ktoré sa pri hľadaní maximálnej trasy vynechajú; metóda potom rovnako ako verzia bez tohto parametra nájde ľubovoľnú maxi-

málne dlhú trasu, ktorá prejde cez maximálny počet vrcholov ale bez uvedených v množine `bez`; táto metóda môže byť zavolaná s rôznymi množinami takto zablokovaných vrcholov.

Napr. pre takéto zadanie grafu:

```
111-112
111-121
122-112
122-121
211-212
211-221
222-212
222-221
111-211
112-212
121-221
122-222
```

takýto test:

```
if __name__ == '__main__':
    vs = VesmirnaStanica('subor1.txt')
    print('moduly =', vs.moduly())
    for m1 in vs.moduly():
        pocet = len({m2 for m2 in vs.moduly() if vs.chodba(m1, m2)})
        print(m1, pocet, vs.zisti(m1), vs.zisti(m1, {111, 212}))
```

vypíše napr.:

```
moduly = {111, 112, 211, 212, 121, 122, 221, 222}
111 3 [111, 112, 122, 121, 221, 211, 212, 222] []
112 3 [112, 122, 121, 221, 222, 212, 211, 111] [112, 122, 121, 221, 211]
211 3 [211, 212, 112, 122, 222, 221, 121, 111] [211, 221, 121, 122, 112]
212 3 [212, 112, 122, 121, 111, 211, 221, 222] []
121 3 [121, 122, 112, 212, 222, 221, 211, 111] [121, 122, 222, 221, 211]
122 3 [122, 112, 212, 211, 111, 121, 221, 222] [122, 121, 221, 211]
221 3 [221, 121, 122, 112, 111, 211, 212, 222] [221, 121, 122, 112]
222 3 [222, 122, 112, 212, 211, 221, 121, 111] [222, 122, 121, 221, 211]
```

Uvedomte si, že ak existuje viac maximálnych trás, backtracking môže vrátiť ľubovoľnú z nich.

Z úlohového servera L.I.S.T. si stiahnite kostru programu `skuska.py`. Mali by ste dodržať tieto vlastnosti programu:

- Nemeňte mená už zadaných atribútov (metód).
- Do existujúcej triedy môžete pridávať vlastné atribúty a metódy, nepoužívajte ale atribúty typu `dict`.
- Pri testovaní vášho riešenia sa bude kontrolovať aj štruktúra vami vytvoreného grafu.

Aby ste mohli spúšťať skúškové testy, riešenie (bez ďalších súborov) odovzdajte na úlohový server https://list.fmph.uniba.sk/. Testy postupne na všetkých testovacích súboroch preverujú vlastnosti vašich algoritmov, pri prvej chybe sa testovanie s daným súborom preruší a ďalšie časti sa netestujú:

- 40% bodov za vytvorenie grafu
- 40% bodov za algoritmus hľadania maximálnej trasy
- 20% bodov za hľadanie trasy, ktorá bude bez označených vrcholov
- pozrite si testovacie dáta v súboroch `'subor1.txt'`, `'subor2.txt'`, `'subor3.txt'`, ..., ktoré bude používať testovač

Praktická časť končí o 11:00 a skúška ďalej pokračuje od 12:00 vyhodnotením v kancelárii m162.

37.29 3. tréningové zadanie - skúška z 19.6.2017

Do veľkého skladu kúpili upratovací robot, ktorý je schopný prejsť nejakú zadanú trasu a pri každom kroku vyčistiť plochu veľkosti 1x1. Počas jeho čistiacej trasy nechceme, aby stúpil na už vyčistenú plochu a požadujeme, aby sa na koniec vrátil na svoje štartové pole, kde sa nachádza jeho nabíjacia stanica (zrejme túto štartovaciu plochu nečistí). Ak si predstavíte veľký sklad ako štvorcovú sieť, ktorého políčka sú veľkosti 1x1, robot sa môže presúvať len na susedné políčka, ktoré majú spoločnú jednu stranu. Na niektorých políčkach môžu byť pre robota prekážky a zrejme na tieto vojsť nemôže.

Napíšte program, ktorý pre robota navrhne takú čo najdlhšiu trasu, ktorá začína na vopred zadanom políčku (a teda na ňom aj končí). Celý sklad reprezentujte ako **neorientovaný neohodnotený** graf, v ktorom vrcholy sú prázdne políčka štvorcovej siete a hranami sú spojené tie vrcholy, ktorých políčka majú spoločné hrany. Políčka s prekážkami nie sú vrcholmi grafu.

Zadefinujte triedu `Graf`:

```
class Graf:
    def __init__(self, meno_suboru):
        self.g = {}                    # reprezentácia grafu
        ...

    def vrcholy(self):
        return []

    def hrana(self, v1, v2):
        return False

    def start(self, riadok, stlpec):
        return 0, []
```

kde

- metóda `__init__(self, meno_suboru)`: prečíta súbor a vytvorí z neho neorientovaný neohodnotený graf; realizujte ho ako asociatívne pole množín súsedov, kde kľúčmi sú všetky vrcholy (dvojice čísel pre riadok a stĺpec) a hodnotami pre tieto vrcholy sú množiny susediacich vrcholov, t.j. množiny dvojíc čísel pre riadok a stĺpec;
- metóda `vrcholy(self)`: vráti zoznam všetkých vrcholov grafu, t.j. zoznam dvojíc (`tuple`) čísel pre riadok a stĺpec;
- metóda `hrana(self, v1, v2)`: zistí, či je hrana medzi vrcholmi grafu `v1` a `v2`, metóda vráti `True` alebo `False`;
- metóda `start(self, riadok stlpec)`: pomocou backtrackingu nájde ľubovoľnú maximálne dlhú trasu, ktorá začína v zadanom vrchole, prejde cez maximálny počet vrcholov práve raz a skončí v štartovom vrchole (tento už do cesty nevkladajte); metóda vráti dvojicu: počet rôznych maximálne dlhých trás a ľubovoľnú z nich ako zoznam (typu `list`) dvojíc čísel pre riadok a stĺpec; ak takáto cesta neexistuje, metóda vráti dvojicu `0, []`.

Sklad je zadaný v súbore takto:

- v každom riadku je popísaný jeden riadok štvorcovej siete - všetky radky sú rovnako dlhé;
- každý znak reprezentuje jedno políčko: ak je to znak bodka `'.'`, označuje to prázdne políčko, inak je to políčko s prekážkou

Napr. pre takéto zadanie grafu:

```
x...
...x
....
```

takýto test:

```
if __name__ == '__main__':
    g = Graf('subor1.txt')
    print('vrcholy =', g.vrcholy())
    for v1, v2 in ((0,0),(0,1)), ((1,0),(0,1)), ((1,0),(1,1)):
        print('hrana', v1, v2, g.hrana(v1, v2))
    print('cesta', (0, 0), g.start(0, 0))
    print('cesta', (1, 1), g.start(1, 1))
```

vypíše napr.:

```
vrcholy = [(0, 1), (1, 2), (2, 3), (2, 0), (1, 0), (0, 3), (2, 2), (1, 1), (2, 1), (0,
↪ 2)]
hrana (0, 0) (0, 1) False
hrana (1, 0) (0, 1) False
hrana (1, 0) (1, 1) True
cesta (0, 0) (0, [])
cesta (1, 1) (2, [(1, 1), (1, 0), (2, 0), (2, 1), (2, 2), (1, 2), (0, 2), (0, 1)])
```

Z úlohového servera L.I.S.T. si stiahnite kostru programu `skuska.py`. Mali by ste dodržať tieto vlastnosti programu:

- Nemeňte mená už zadaných atribútov (metód).
- Do existujúcej triedy môžete pridávať vlastné atribúty a metódy, nepoužívajte ale atribúty, v ktorých si budete uchovávať pôvodný dvojrozmerný zoznam.
- Pri testovaní vášho riešenia sa bude kontrolovať aj štruktúra vami vytvoreného grafu.

Aby ste mohli spúšťať skúškové testy, riešenie (bez ďalších súborov) odovzdajte na úlohový server https://list.fmph.uniba.sk/. Testy postupne na všetkých testovacích súboroch preverujú vlastnosti vašich algoritmov, pri prvej chybe sa testovanie s daným súborom preruší a ďalšie časti sa netestujú:

- 40% bodov za vytvorenie grafu
- 40% bodov za algoritmus hľadania maximálnej trasy
- 20% bodov za správny počet maximálnych trás
- pozrite si testovacie dáta v súboroch `'subor1.txt'`, `'subor2.txt'`, `'subor3.txt'`, ..., ktoré bude používať testovač

Praktická časť končí o 11:00 a skúška ďalej pokračuje od 12:00 vyhodnotením v kancelárii m162.

37.30 Copyright

ISBN PDF verzie 978-80-8147-083-7

ISBN webovej verzie 978-80-8147-084-4

37.31 Zimný semester

Semestrálny projekt

Môžete začať uvažovať nad semestrálnym projektom.

- tému si zvolíte z ponúknutej množiny tém
- za načas odovzdaný projekt môžete získať maximálne 10 bodov - tieto sa pripočítavajú k bodom ku skúške

Na stránke *Semestrálny projekt v zimnom semestri* sú detailné informácie k projektu.

Záverečný test

V pondelok 18. decembra 2017 bol *Záverečný test z Programovania (1) 2017/2018*. Pozrite si *Výsledky záverečného testu*.

Môžete sa inšpirovať zadaniami záverečných testov z minulých školských rokov:

- *Záverečný test z Programovania (1) 2014/2015*
- *Záverečný test z Programovania (1) 2015/2016*
- *Záverečný test z Programovania (1) 2016/2017*

Priebežný test

V pondelok 13. novembra 2017 bol priebežný test z programovania. Pozrite si *Výsledky priebežného testu*.

Môžete sa inšpirovať zadaniami priebežných testov z minulých školských rokov:

- *Priebežný test z Programovania (1) 2014/2015*
- *Priebežný test z Programovania (1) 2015/2016*
- *Priebežný test z Programovania (1) 2016/2017*

Skúšky v zimnom semestri

Riadne termíny skúšok boli: *19.1.2018* a *24.1.2018*

- praktická časť skúšky prebiehala v halách H3 a H6 od **8:30** do **11:00**
- po obede bola vyhodnotenie (m-162) a zapísanie známky do indexu
- na skúšku sa prihlasujete cez AIS2
- prvý opravný termín bol *5.2.2018* a druhý opravný *12.2.2018*

Testovacie zadania skúšky z minulého školského roku

Zadania:

- *1. tréningové zadanie - skúška z 16.1.2017 - Harry Potter a preskakovadlá*
- *2. tréningové zadanie - skúška z 18.1.2017 - Sokoban*

- *3. tréningové zadanie - skúška z 6.2.2017 - Kamery v Kocúrkove*

37.32 Letný semester

Záverečný test

V pondelok 14.5.2018 bol *Záverečný test z Programovania (2) 2017/2018*. Maximálne sa dá získať okolo 45 bodov, ale ku skúške si nesiete len 40. Pozrite si *Výsledky záverečného testu*.

Môžete sa inšpirovať zadaniami záverečných testov z minulých školských rokov:

- *Záverečný test z Programovania (2) 2015/2016*
- *Záverečný test z Programovania (2) 2016/2017*

Skúšky v letnom semestri

Riadne termíny skúšok boli: *4.6.2018* a *6.6.2018*

- praktická časť skúšky bude prebiehať v halách H3 a H6 od **8:30** do **11:00**
- po obede bude vyhodnotenie (m-162) a zapísanie známky do indexu
- na skúšku sa prihlasujete cez AIS2
- prvý opravný termín bol *18.6.2018* a druhý opravný *27.6.2018*

Testovacie zadania skúšky z minulého školského roku:

- *1. tréningové zadanie - skúška z 5.6.2017*
- *2. tréningové zadanie - skúška z 7.6.2017*
- *3. tréningové zadanie - skúška z 19.6.2017*

Riešenia týchto zadaní môžete otestovať v https://list.fmph.uniba.sk/.

Semestrálny projekt v letnom semestri

Môžete začať uvažovať nad semestrálnym projektom.

- tému si zvolíte z ponúknutej množiny tém
- za načas odovzdaný projekt môžete získať maximálne 10 bodov - tieto sa pripočítavajú k bodom ku skúške
- projekt, ktorý získa aspoň 5 bodov môžete odovzdať aj ako **ročníkový projekt**:
  - ak si chcete zaevidovať tento projekt ako ročníkový **RP(1)**, stači na adresu `lucan @ fmph.uniba.sk` poslať:
    - vaše meno aj email
    - projekt: RP(1)
    - téma projektu: napr. počítačová hra s inteligenciou protihráča, vizualizácia dátovej štruktúry, editor grafických objektov, edukačný softvér, ...
    - prostredie: Python

        * vedúci RP(1): moje meno alebo meno vašej cvičiacej
    - táto téma projektu nie je definitívna, môžete ju neskôr zmeniť

Na stránke *Semestrálny projekt* sú detailnejšie informácie k projektu.
