Programovanie v Pythone

časť 1

Andrej Blaho

sep 07, 2016

Obsah

Fakulta matematiky, fyziky a informatiky

Univerzita Komenského v Bratislave

Autor Andrej Blaho

Názov Programovanie v Pythone 1 (prednášky k predmetu Programovanie (1) 1-AIN-130/13)

Vydavateľ Knižničné a edičné centrum FMFI UK

Rok vydania 2016

Miesto vydania Bratislava

Vydanie prvé

Počet strán 322

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

ISBN 978-80-8147-067-7

KAPITOLA 1

Štart Pythonu

Python je moderný programovací jazyk, ktorého popularita stále rastie.

- jeho autorom je Guido van Rossum (http://sk.wikipedia.org/wiki/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++) 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

Ako ho získať

- zo stránky https://www.python.org/ stiahnete najnovšiu verziu Pythonu - momentálne je to verzia **3.5.0**
- spustite inštalačný program (napr. `python-3.5.0.exe`)

Upozornenie

Nesťahujte verziu začínajúcu 2 (napr. 2.7.10) - tá nie je kompatibilná s verziou 3.x

Po spustení **IDLE (Python GUI)** - čo je vývojové prostredie (Integrated Development 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 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:27:37) [MSC v.1900 64 bit␣
↪(AMD64)] on win32
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 **príkazovom režime**: 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**

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 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
>>> 999999999*99999999999
9999999989000000001
>>> 999999999*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 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#1>", line 1, in <module>
    ahoj
NameError: name 'ahoj' is not defined
>>>
```

dostaneme chybovú správu **name 'ahoj' is not defined** - Python nepozná takéto slovo. 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

```
>>> '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 do apostrofov `'text'` alebo úvodzoviek `"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.

1.1 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. Napr. príkaz `print()` vypíše hodnoty uvedené v zátvorkách. Napr.

```
# môj prvý program
print('vitaj')
print()
print('výpočet je', 3 + 4 * 5)
```

Takto vytvorený program treba spustiť (**Run**, alebo klávesom <F5>) a vtedy sa v okne shell objaví

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

výpočet je 23
>>>
```

- takýmto programom hovoríme **skript** - väčšinou sú uložené v súbore s príponou `.py`
- po ich spustení sa v pôvodnom okne **shell** najprv celý Python reštartuje (zabudne, všetko, čo sme ho doteraz naučili) a takto vyčistený vykoná všetky príkazy programu
- do prvého riadka sme zapísali komentár - text za znakom `#` sa ignoruje, teda je to komentár pre nás

výrazy v programe

- po zadaní výrazov v príkazovom 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 nejakým príkazom alebo funkciou (napr. pomocou `print()`) alebo priradiť do nejakej premennej (uvidíme to neskôr)

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. ENTER, pridáme do programu nový riadok s funkciou `input()`

Do skriptu dopíšeme nový riadok:

```
# môj prvý program
print('vitaj')
print()
print('výpočet je', 3 + 4 * 5)

# bude sa čakať na stlačenie klávesu ENTER
input()
```

Po spustení takéhoto programu sa najprv vypíšu zadané texty a okno sa nezavrie, kým nestlačíme ENTER. Príkaz `input()` môže obsahovať aj nejaký text: ten sa potom vypíše, napr.

```
# 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
```

Po stlačení klávesu ENTER sa okno zatvorí.

Zhrňme oba tieto 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 zadaný reťazec

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

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

Príkaz `input()` tu čaká na stlačenie ENTER a teda, kým ho nestlačíme ale píšeme nejaký text, tak 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'
>>>
```

1.2 Typy údajov

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

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

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

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

Rôzne typy hodnôt majú zadefinované rôzne operácie.

celočíselné operácie

- oba operandy musia byť celočíselného typu
  - `+` súčet, napr. `1 + 2` má hodnotu `3`
  - `-` rozdiel, napr. `2 -5` má hodnotu `-3`
  - `*` násobenie, napr. `3 * 37` má hodnotu `111`
  - `//` celočíselné delenie, napr. `22 // 7` má hodnotu `3`
  - `%` zvyšok po delení, napr. `22 % 7` má hodnotu `1`
  - `**` umocňovanie, napr. `2 ** 8` má hodnotu `256`
- zrejme nemôžeme deliť 0

operácie s desatinnými číslami

- aspoň jeden operand musí byť desatinného typu (okrem delenia `/`)
  - `+` súčet, napr. `1 + 0.2` má hodnotu `1.2`
  - `-` rozdiel, napr. `6 -2.86` má hodnotu `3.14`
  - `*` násobenie, napr. `1.5 * 2.5` má hodnotu `3.75`
  - `/` delenie, napr. `23 / 3` má hodnotu `7.666666666666667`
  - `//` delenie zaokrúhlené nadol, napr. `23.0 // 3` má hodnotu `7.0`
  - `%` zvyšok po delení, napr. `23.0 % 3` má hodnotu `2.0`
  - `**` umocňovanie, napr. `3 ** 3.` má hodnotu `27.0`
- zrejme nemôžeme deliť 0

operácie so znakovými reťazcami

- `+` zreťazenie (spojenie dvoch reťazcov), napr. `'a' + 'b'` má hodnotu `'ab'`
- `*` viacnásobné zreťazenie toho istého reťazca, napr. `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'
>>> '=' * 20
'===================='
>>> 10 * ' :-) '
' :-)  :-)  :-)  :-)  :-)  :-)  :-)  :-)  :-)  :-) '
>>>
```

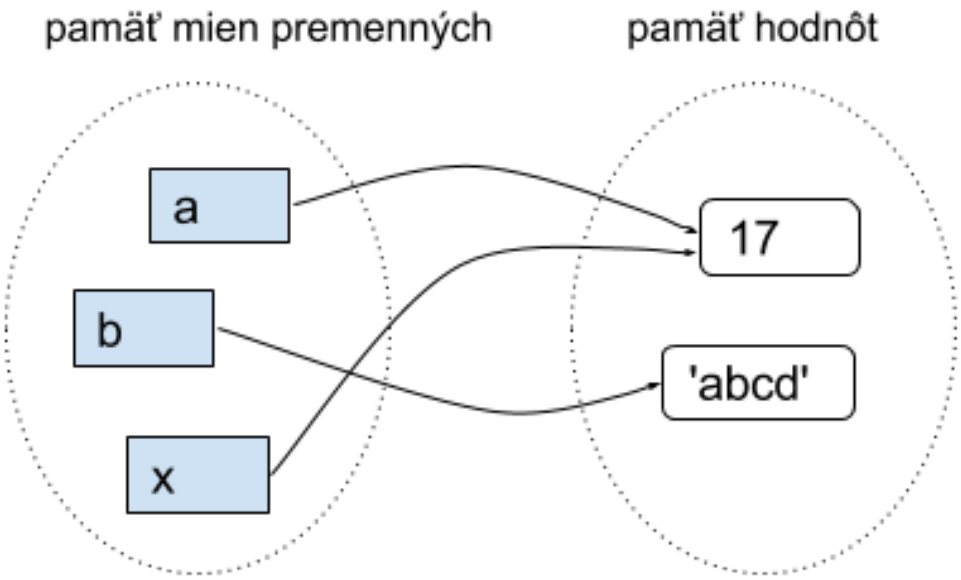
1.3 Premenné a priradenie

- 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 (napr. `for`, `if`, `return`, ...)
- premenná sa vytvorí priraďovacím príkazom (ak ešte doteraz neexistovala):
  - zapisujeme: `premenná = hodnota`
    - tento zápis znamená, že do *premennej* sa má priradiť zadaná *hodnota*
  - v skutočnosti sa v Pythone do premennej priradí **referencia** (odkaz) 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

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:



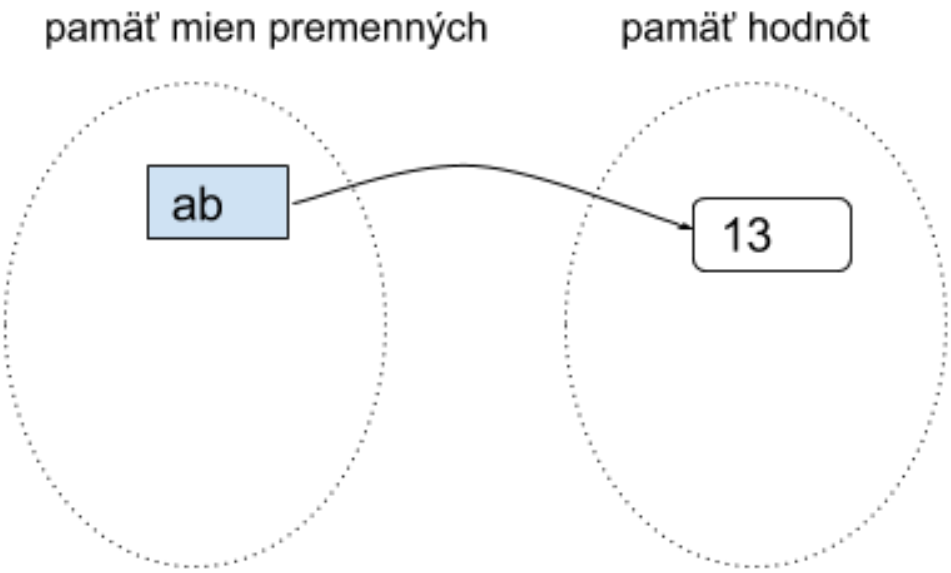
Vidíme, že

- priradenie do jednej premennej nejakej inej premennej (napr. `x = a`) neznamená 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

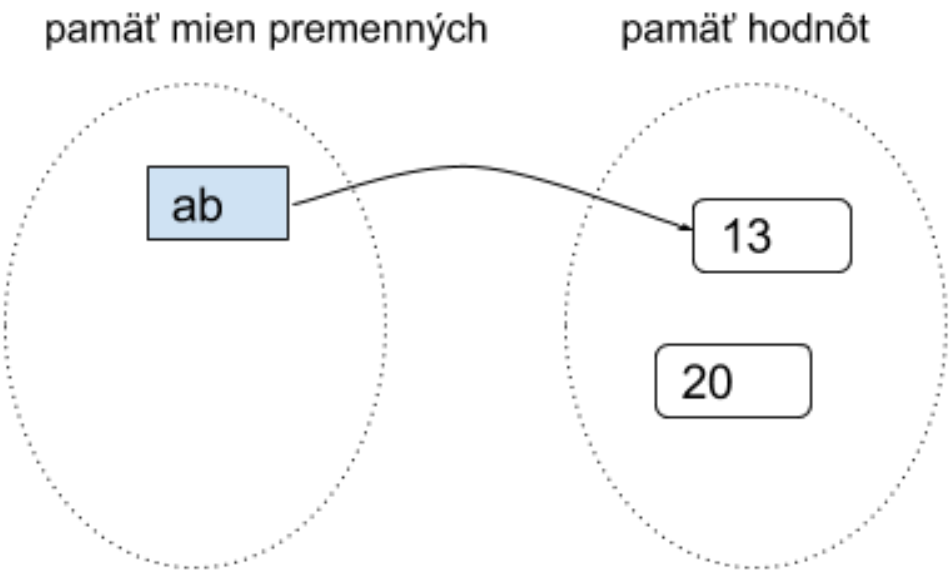
V ďalšom príklade vidíme, ako to funguje, keď vo výraze na pravej strane (kde je priraďovaná hodnota) sa nachádza tá istá premenná, ako na ľavej strane (teda kam priraďujeme):

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

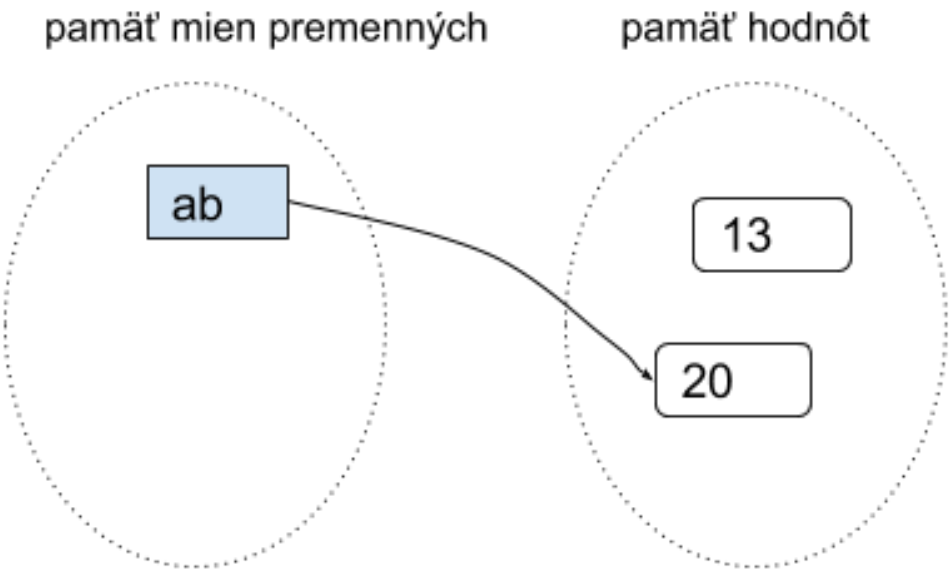
1. najprv má `ab` hodnotu `13`



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



3. do premennej `ab` sa priradí nová hodnota



možné problémy s menami premenných

Mená premenných môžu byť skoro ľubovoľné reťazce, zložené z písmen (malých a veľkých), číslic a znaku podčiarkovník. Keďže mená premenných si najčastejšie programátori vymýšľajú sami, neskúsený programátor môže pri tom narobiť problémy nielen druhým, ale aj sebe, keď sa nevyzná ani v svojom programe. Uvedieme niekoľko typických problémov.

Veľké písmená v menách premenných treba používať veľmi opatrne. Všimnite si, že všetky tieto premenné sú rôzne:

```
>>> Pocet = 10
>>> POCET = 20
>>> PoCet = 30
>>> PoceT = 40
```

V Pythone majú programátori dohodu, že na premenné sa používaú len malé písmená (prípadne aj číslice a podčiarkovník).

Tiež si treba dávať pozor na ľahko zameniteľné znaky, napr. písmeno `O` a číslica `0` a tiež písmeno `l` a číslica `1`:

```
>>> OO1 = 15
>>> O01 = 16
>>> print(OO1, O01)
15 16
>>> d1zka = 15
>>> dlzka = d1zka + 1
```

Snažíme sa nepoužívať ani slová, v ktorých často robíme preklepy, napr.

```
>>> kalerab = 'mnam'
>>> karelab = 'fuj'
```

správne mená

Pri písaní programov používame čo najvhodnejšie mená premenných.

Napr. tento program má veľmi nezrozumiteľné mená a preto sa horšie chápe, čo vlastne počíta:

```
vrk = 16
mrk = 3.14
drk = 2 * vrk * mrk
frk = vrk * mrk * vrk
print('drk =', drk)
print('frk =', frk)
```

Stačí dať menám premenných zodpovedajúce texty, napr.

```
polomer = 16
pi = 3.14
obvod = 2 * pi * polomer
obsah = pi * polomer ** 2
print('obvod =', obvod)
print('obsah =', obsah)
```

Tento zápis programu je zrazu oveľa zrozumiteľnejší (hoci robí presne to isté, ako predchádzajúci).

Uvedomte si

Zápisu programu má rozumieť hlavne človek. Hoci niekedy bude tento program vykonávať aj počítač, tomu je ale jedno, aké sú tam mená premenných.

1.4 Zhrnutie

príkazový režim za výzvu (prompt) >>> zadávame výrazy a príkazy

programovací režim **skripty** vytvárame v textovom súbore (prípona .py) a spúšťame pomocou <F5> alebo z operačného systému

funkcia print() vypisuje zadané hodnoty

funkcia input() čaká na stlačenie ENTER, funkcia vráti zadaný používateľom reťazec

typy hodnôt a premenné základné typy: `int`, `float`, `str`, funkcia `type()`

premenná obsahuje referenciu na hodnotu, vznikne pomocou priraďovacieho príkazu

KAPITOLA 2

Priradenia, for-cykly

V tejto prednáške dokončíme základné prvky jazyka: typy hodnôt, premenné a hlavne priradenia a tiež sa zoznámime s prvým zloženým príkazom na opakovanie nejakej činnosti for-cyklom.

Začneme s takýmto jednoduchým príkladom programu (skriptu), ktorý zapíšeme v editovacom okne:

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

V tomto príklade sa využíva funkcia `input()`, ktorá zastaví bežiaci výpočet, vypýta si od používateľa zadať nejaký text a tento uloží do premennej `meno`. Na koniec toto zadané meno vypíše. Program spustíme klávesom **F5**. Beh programu v konzolovom okne (shell pythonu) môže vyzerať napr. takto

```
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:

```
cislo = input('zadaj cenu jedneho vyrobku: ')
spolu = cislo * 4
print('4 vyrobky stoja', spolu, 'euro')
```

Týmto programom sme chceli prečítať cenu jedného výrobku a z toho vypočítať, aká je cena 4 takýchto výrobkov. Po spustení programu dostávame:

```
zadaj cenu jedneho vyrobku: 1.2
4 vyrobky stoja 1.21.21.21.2 euro
```

Takýto výsledok je zrejme nesprávny: očakávali sme celkovú cenu 4.8 euro. Problémom tu bolo to, že funkcia `input()` zadanú hodnotu vráti nie ako číslo, ale ako znakový reťazec, teda `'1.2'`. 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:

```
cislo_str = input('zadaj cenu jedneho vyrobku: ')
cislo = float(cislo_str)                          # pretypovanie
spolu = cislo * 4
print('4 vyrobky stoja', spolu, 'euro')
```

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 cenu jedneho vyrobku: 1.2
4 vyrobky stoja 4.8 euro
```

Kvôli čitateľnosti programu sme tu použili tri rôzne premenné: `cislo_str`, `cislo` a `spolu`. 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:

```
print('4 vyrobky stoja', 4*float(input('zadaj cenu jedneho vyrobku: ')),
 ↪'euro')
```

Tento program robí presne to isté, ako predchádzajúci zápis, len nevyužíva žiadne pomocné premenné.

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

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'>
>>> 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 hodnota, 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.
```

2.1 Priraďovacie príkazy

Vráťme sa k priraďovaciemu príkazu z 1. prednášky v tvare:

```
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
>>> 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 opät' 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
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.

2.2 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
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 funkcie (tzv. metódu znakového reťazca), pomocou ktorej môžeme vytvárať aj takto komplikované výrazy. Základom je 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.

2.3 Cyklus

Ak potrebujeme 5-krát vypísať ten istý text, môžeme to zapísať napr. takto:

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

Namiesto toho použijeme novú konštrukciu **for-cyklu**:

```
for i in 1, 2, 3, 4, 5:
    print('programujem v Pythone')
```

Ako to funguje:

- do premennej `i` sa bude postupne priraďovať nasledovná hodnota zo zoznamu hodnôt
- začíname s prvou hodnotou, teda v tomto prípade `1`
- pre každú hodnotu so zoznamu sa vykonajú príkazy, ktoré sú v **tele cyklu**, t.j. tie príkazy, ktoré sú odsadené
- v našom príklade sa päťkrát vypíše rovnaký text `'programujem v Pythone'`, hodnota premennej `i` nemá na tento výpis žiadny vplyv
- všimnite si znak ':' na konci riadka s `for` - ten je tu povinne, bez neho by to nefungovalo

Telo cyklu:

- tvoria príkazy, ktoré sa majú opakovať; definujú sa **odsadením** príslušných riadkov
- odsadenie je povinné a musí byť minimálne 1 medzera, odporúča sa odsadzovať vždy o **4 medzery**
- ak telo cyklu obsahuje viac príkazov, všetky **musia** byť odsadené o rovnaký počet medzier
- telo cyklu **nesmie** byť prázdne, musí obsahovať aspoň jeden príkaz
- niekedy sa môže hodiť **prázdny príkaz** `pass`, ktorý nerobí nič, len oznámi čitateľovi, že sme na telo cyklu nezabudli, ale zatiaľ tam nechceme mať nič
- prázdne riadky v tele cyklu nemajú žiaden význam, často slúžia na sprehľadnenie čitateľnosti kódu

Podobný výpis dostaneme aj takýmto zápisom for-cyklu:

```
for i in 1, 1, 1, 1, 1:
    print('programujem v Pythone')
    print('~~~~~~~~~~~~~~~~~~~~~')
```

Aj v tomto prípade sa telo cyklu bude opakovať 5-krát. Telo cyklu sa tu teraz skladá z dvoch príkazov `print()`, preto sa vypíše týchto 10 riadkov textu:

```
programujem v Pythone
~~~~~~~~~~~~~~~~~~~~~
programujem v Pythone
~~~~~~~~~~~~~~~~~~~~~
programujem v Pythone
~~~~~~~~~~~~~~~~~~~~~
programujem v Pythone
~~~~~~~~~~~~~~~~~~~~~
programujem v Pythone
~~~~~~~~~~~~~~~~~~~~~
```

Ak by sme ale druhý riadok tela cyklu neodsunuli:

```
for i in 1, 1, 1, 1, 1:
    print('programujem v Pythone')
print('~~~~~~~~~~~~~~~~~~~~~')
```

telo cyklu je teraz len jeden príkaz. Program najprv 5-krát vypíše text `'programujem v Pythone'` a až potom jeden riadok s vlnkami:

```
programujem v Pythone
programujem v Pythone
programujem v Pythone
programujem v Pythone
programujem v Pythone
~~~~~~~~~~~~~~~~~~~~~
```

V tele cyklu môžeme použiť aj **premennú cyklu**, ktorej hodnota sa pri každom prechode cyklom automaticky mení. V nasledovnom príklade je premennou cyklu `prem`:

```
for prem in 1, 2, 3, 4, 5:
    print(prem)
```

Po spustení programu sa postupne vypíšu všetky nadobudnuté hodnoty:

```
1
2
3
4
5
```

Ďalší program počíta druhé mocniny niektorých zadaných prvočísel:

```
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
```

Premenná cyklu nemusí nadobúdať len celočíselné hodnoty, ale hodnoty úplne ľubovoľného typu, napr.

```
>>> for x in 22/7, 3, 14, 8., 1000-1e-3:
        print('druha odmocnina z', x, 'je', x**.5)

druha odmocnina z 3.142857142857143 je 1.7728105208558367
druha odmocnina z 3 je 1.7320508075688772
druha odmocnina z 14 je 3.7416573867739413
druha odmocnina z 8.0 je 2.8284271247461903
druha odmocnina z 999.999 je 31.62276079029154
```

Niektoré hodnoty postupnosti hodnôt for-cyklu sú tu celočíselné, iné sú desatinné. V tomto príklade si všimnite, ako sme tu počítali druhú odmocninu čísla: číslo sme umocnili na 1/2, t.j. `0.5`.

V ďalšom cykle sme namiesto zoznamu hodnôt použili znakový reťazec: už vieme, že znakový reťazec je vlastne postupnosť znakov, preto for-cyklus tento znakový reťazec "rozoberie" na jednotlivé znaky a s každým znakom vykoná telo cyklu (znak vypíše 10-krát):

```
>>> for pismeno in 'Python':
        print(pismeno*10)

PPPPPPPPPP
yyyyyyyyyy
tttttttttt
hhhhhhhhhh
oooooooooo
nnnnnnnnnn
```

Rovnaký výsledok by sme dosiahli aj týmto zápisom for-cyklu:

```
for pismeno in 'P', 'y', 't', 'h', 'o', 'n':
    print(pismeno*10)
```

2.4 Generovanie postupnosti čísel

Využijeme ďalšiu štandardnú funkciu `range()`, pomocou ktorej budeme najmä vo for-cykloch generovať rôzne postupnosti hodnôt. Táto funkcia môže mať rôzny počet parametrov a podľa toho sa bude generovaná aj výsledná postupnosť. Všetky parametre musia byť celočíselné.

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 (resp. zníži pre záporný krok) každý nasledovný prvok postupnosti, ak tento parameter chýba, predpokladá sa 1

Väčšinou platí, že do parametra `stop` nastavíme o 1 väčšiu hodnotu, ako potrebujeme poslednú hodnotu vygenerovanej postupnosti.

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,-1)`	10,9,8,7,6,5,4,3,2
`range(10,1)`	*prázdna postupnosť*
`range(1,1)`	*prázdna postupnosť*

Niekoľko ukážok v priamom režime:

```
>>> for n in range(5):      # postupnost' čísel od 0 pre menšie ako 5
        print(n)

0
1
2
3
4
>>> for n in range(5, 9):  # postupnost' čísel od 5 pre menšie ako 9
        print(n)

5
6
7
8
>>> print(range(7, 100, 11))
range(7, 100, 11)
```

Všimnite si, že volaním `print(range(7,100,11))` sa nedozvieme nič užitočné o vygenerovanej postupnosti. Pre kontrolu hodnôt vygenerovaných pomocou `range()` musíme zatiaľ použiť for-cyklus.

Ak potrebujeme vypisovať pomocou for-cyklu väčší počet čísel, je veľmi nepraktické, ak každý jeden `print()` vypisuje do ďalšieho riadka. Hodilo by sa nám, keby `print()` v niektorých situáciách nekončil prechodom na nový riadok. Využijeme nový typ parametra:

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.

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

Tieto 3 volania `print()` vypíšu len jeden riadok:

```
programujem_10...rokov
```

Tento špeciálny parameter v `print()` sa využije najmä vo for-cykloch. V nasledujúcom príklade vypíšeme 100 hodnôt do jedného dlhého riadka, čísla pritom oddeľujeme čiarkami s medzerou:

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

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,
```

Zrejme v programovom režime ostane pozícia výpisu funkciou `print()` na konci posledného riadka a ďalšie výpisy by pokračovali tu. Zvykne sa po skončení takéhoto cyklu zavolať `print()` bez parametrov.

Pri tlačení napr. tabuliek nejakých hodnôt často potrebujeme slušne naformátovať takýto výstup. Napr.

```
>>> for i in range(1,11):
        print(i, i*i, i**3, i**4)

1 1 1 1
2 4 8 16
3 9 27 81
4 16 64 256
5 25 125 625
6 36 216 1296
7 49 343 2401
8 64 512 4096
9 81 729 6561
10 100 1000 10000
```

Najčastejšie použijeme formátovanie znakového reťazca (metódou `format()`):

```
>>> for i in range(1,11):
        print('{:3} {:4} {:5} {:6}'.format(i, i*i, i**3, i**4))

  1    1     1      1
  2    4     8     16
  3    9    27     81
  4   16    64    256
  5   25   125    625
  6   36   216   1296
  7   49   343   2401
  8   64   512   4096
  9   81   729   6561
 10  100  1000  10000
```

V tomto príklade vidíte, že formátovacia dvojica znakov `'{}'` môže obsahovať šírku vypisovanej hodnoty - uvádzame

ju za dvojbodkou, teda `'{:3}'` označuje, že hodnota sa vypisuje na šírku 3 a ak je kratšia (napr. jednociferné číslo), zľava sa doplní medzerami.

Ďalší príklad ilustruje for-cyklus, v ktorom počet prechodov určuje načítaná hodnota nejakej premennej:

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

Program vypíše `n` riadkov, pričom v prvom je jedna hviezdička, v druhom 2, atď. Napr. takéto spustenie programu:

```
zadaj pocet: 5
*
**
***
****
*****
```

Túto ideu môžeme rôzne rozvíjať tak, že v každom riadku bude nejaký počet medzier a nejaký iný počet hviezdičiek, napr.

```
n = int(input('zadaj pocet: '))
for i in range(n):
    print(' '*(n-1-i) + '*'*(2*i+1))
for i in range(n):
    print(' '*i + '*'*(2*n-2*i-1))
```

A spustenie, napr.

```
zadaj pocet: 5
    *
   ***
  *****
 *******
*********
*********
 *******
  *****
   ***
    *
```

For-cyklus sa výborne využije v prípadoch, keď sa v tele cyklu vyskytuje príkaz, ktorý inkrementuje nejakú premennú (alebo ju mení inou operáciou). Zrejme toto inkremenovanie prejde toľkokrát, koľko je prechodov cyklu a často sa pritom využije premenná cyklu.

Ukážme túto ideu na príklade, v ktorom spočítame všetky hodnoty premennej cyklu. Použijeme na to ďalšiu premennú, do ktorej sa bude postupne pripočítavať premenná cyklu:

```
n = int(input('zadaj pocet: '))
sucet = 0
for i in range(1, n+1):
    sucet = sucet + i                    # alebo  sucet += i
print('sucet cisel =', sucet)
```

V tomto programe sme použili premennú `sucet`, do ktorej postupne pripočítavame všetky hodnoty z intervalu (range) od 1 do `n`. Vďaka tomu sa po skončení cyklu v tejto premennej nachádza práve súčet všetkých celých čísel do `n`. Sputíme napr.

```
zadaj pocet: 10
sucet cisel = 55
```

Tento program nám môže slúžiť ako nejaká **šablóna**, pomocou ktorej riešime podobnú triedu úloh: v každom prechode cyklu niečo pripočítavame, násobime, delíme, ... Takáto **pripočítavacia šablóna** sa skladá z:

- inicializácia pomocnej premennej (alebo viacerých premenných), do ktorej sa bude *pripočítavať*, najčastejšie je to vynulovanie;
- v tele cyklu sa do tejto pomocnej premennej *pripočíta* nejaká hodnota, najčastejšie premenná cyklu;
- po skončení cyklu sa v tejto pomocnej premennej nachádza očakávaný výsledok *sčitovania*.

Ukážeme to na niekoľkých príkladoch.

Výpočet faktoriálu, čo je vlastne súčin čísel 1 * 2 * 3 * ... * `n`. Pomocná premenná musí byť inicializovaná **1**, namiesto pripočítavania bude násobenie:

```
n = int(input('zadaj cislo: '))
sucin = 1
for i in range(1, n+1):
    sucin = sucin * i                    # alebo  sucin *= i
print('{}! = {}'.format(n, sucin))
```

V záverečnom výpise sme použili formátovanie reťazca. Po spustení, napr.

```
zadaj cislo: 6
6! = 720
```

Úplne rovnaký princíp môžeme použiť aj keď premenná cyklu nebude obsahovať čísla nejakého intervalu, ale budú to znaky z rozoberaného znakového reťazca. Nasledovný príklad postupne prechádza znak za znakom daného reťazca a za každým k pomocnej premennej pripočíta **1**:

```
retazec = input('zadaj retazec: ')
pocet = 0
for znak in retazec:
    pocet = pocet + 1                    # alebo  pocet += 1
print('dlzka retazca =', pocet)
```

Tento program spočíta počet znakov zadaného znakového reťazca, napr.

```
zadaj retazec: python
dlzka retazca = 6
```

Ďalší príklad ilustruje, ako môžeme využiť premennú cyklu, ktorá obsahuje znak zadaného reťazca. Pomocná premenná bude znakového typu a v tomto príklade ju budeme inicializovať prázdnym reťazcom:

```
retazec = input('zadaj retazec: ')
novy = ''
for znak in retazec:
    novy = novy + znak + znak
print('novy retazec =', novy)
```

V tele cyklu sa k pomocnej premennej priraďujú dva rovnaké znaky. Vďaka tomuto sa vytvorí nový reťazec, v ktorom bude každý znak z pôvodného reťazca dvakrát, napr.

```
zadaj retazec: Python
novy retazec = PPyytthhoonn
```

2.5 Vnorené cykly

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)
```

Pos pustení 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.

2.6 Zhrnutie

konverzie medzi typmi pomocou funkcií: `int()`, `float()`, `str()`

paralelné (viacnásobné) priradenie viac rôznych hodnôt sa naraz priradí do viacerých premenných

funkcia 'reťazec'.format() slúži na formátovanie reťazca, ktorý môže obsahovať nejaké iné hodnoty (tieto sa automaticky prekonvertujú na reťazce)

for-cyklus pomocou **premennej cyklu** vykoná zadaný počet-krát **telo cyklu**

funkcia range() vytvorí postupnosť celočíselných hodnôt, najčastejšie pre for-cyklus

KAPITOLA 3

Grafika

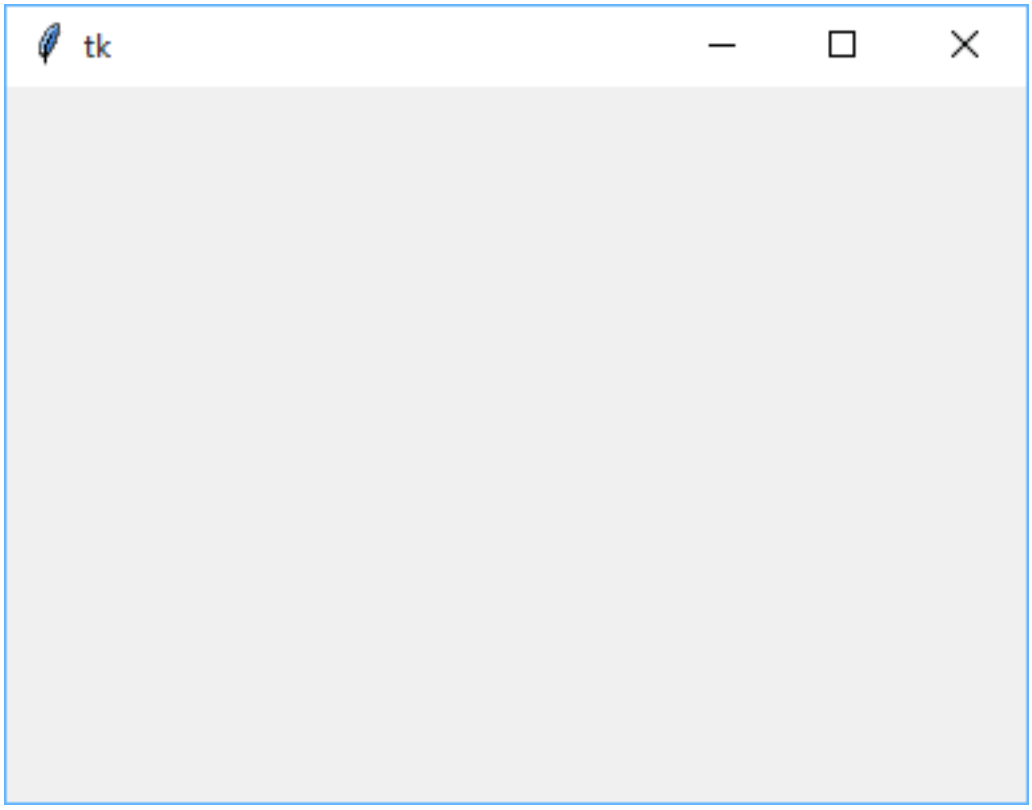
Doteraz sme pracovali buď v priamom (shell) alebo v programovom režime. Programy sme spúšťali priamo z editora (kláves F5) alebo dvojkliknutím na súbor z operačného systému. V dnešnej prednáške ukážeme, ako vytvoríme program, ktorý pracuje s grafickým oknom (v grafickom režime).

Už vieme, že príkaz (funkcia) `print()` vypisuje priamo do textového okna (shell). Existuje iná skupina príkazov (funkcií), ktoré nevypisujú to textového ale grafického okna. Takéto okno sa ale nevytvorí samo, musíme zadať špeciálne príkazy:

```
import tkinter

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

Keď takto vytvorený program spustíme (z editora klávesom F5), otvorí sa nové okno so šedou prázdnou plochou. Teraz môžeme v textovom okne (shell) zadávať grafické príkazy a tie budú priamo zasahovať do grafického okna. Ako to funguje:

- príkaz `import tkinter` oznámi, že budeme pracovať s **modulom** `tkinter`, ktorý obsahuje grafické príkazy
  - týmto príkazom vznikla nová premenná `tkinter`, ktorá obsahuje referenciu na tento modul, t.j. všetky funkcie a premenné, ktoré sú definované v tomto module, sú prístupné pomocou tejto premennej a preto k nim budeme pristupovať tzv. bodkovou notáciou, t.j. vždy uvedieme meno premennej `tkinter`, za tým bodku a meno funkcie alebo premennej, napr. `tkinter.Canvas`
- zápis `tkinter.Canvas()` vytvorí grafickú plochu a aby sme s touto plochou mohli ďalej pracovať, uložíme si jej referenciu do premennej `canvas`
  - nezabudnite uviesť okrúhle zátvorky: vďaka nim sa zavolá funkcia, ktorá vytvorí nové okno, bez týchto zátvoriek zápis `tkinter.Canvas` označuje len referenciu na neajkú funkciu bez jej zavolania
- kým nezadáme aj príkaz `canvas.pack()`, grafická plocha sa ešte nezobrazí - volanie `canvas.pack()` zabezpečí zobrazenie nového okna aj s novovytvorenou grafickou plochou



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

- Tkinter 8.5 reference: a GUI for Python (http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html)

Všetky grafické príkazy pracujú s grafickou plochou, ku ktorej máme prístup prostredníctvom premennej `canvas`. Tieto príkazy sú v skutočnosti funkciami, ktoré budeme volať s nejakými parametrami. Všeobecný tvar väčšiny grafických príkazov je:

```
canvas.create_utvar(x, y, x, y, ..., param=hodnota, param=hodnota, ...)
```

kde

- `create_utvar` je meno funkcie na vytvorenie grafického objektu, napr. `create_line`, `create_rectangle`, `create_oval`, ...
- `x,y,x,y,...` je postupnosť dvojíc súradníc bodov v grafickej ploche (rôzne príkazy majú rôzny počet bodov v rovine)
- `param=hodnota` je dvojica: meno doplnkového parametra (napr. `fill`, `width`, ...) a jej hodnota (napr. `'red'`, `5`, `'Arial 20'`, ...)

Pre všetky grafické príkazy platí, že keď pre doplnkové parametre nejaké neuvedieme hodnoty, tak tieto majú nastavené svoje inicializované (default) hodnoty (často takýmito náhradnými default hodnotami sú napr. `fill='black'` alebo `width=1`).

Súradnicová sústava

Súradnicová sústava v grafickej ploche je trochu inak orientovaná, ako sme zvyknutí z matematiky:

- **x-ová os** prechádza po hornej hrane grafickej plochy zľava doprava

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

Veľkosť grafickej plochy je zatiaľ 378x264 bodov (pixelov), ale neskôr uvidíme, ako môžeme túto veľkosť zmeniť.

Kreslenie čiar

Čiary v grafickom režime kreslíme pomocou funkcie `canvas.create_line()`. Táto funkcia dokáže kresliť lomenú čiaru, ktorá sa skladá z jednej alebo viacerých úsečiek.

- parametrom funkcie je postupnosť súradníc v tvare `canvas.create_line(x1,y1,x2,y2,...)`
- ak je zadaných bodov (dvojíc `x`, `y`) viac ako 2, kreslí sa lomená čiara zložená z navzájom nadväzujúcich úsečiek

Napr. takýto príkaz

```
>>> canvas.create_line(50, 120, 150, 70)
```

nakreslí úsečku z bodu `(50,120)` do bodu `(150,70)`. Zapíšme tento príkaz priamo do nášho programu:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()
canvas.create_line(50, 120, 150, 70)
```

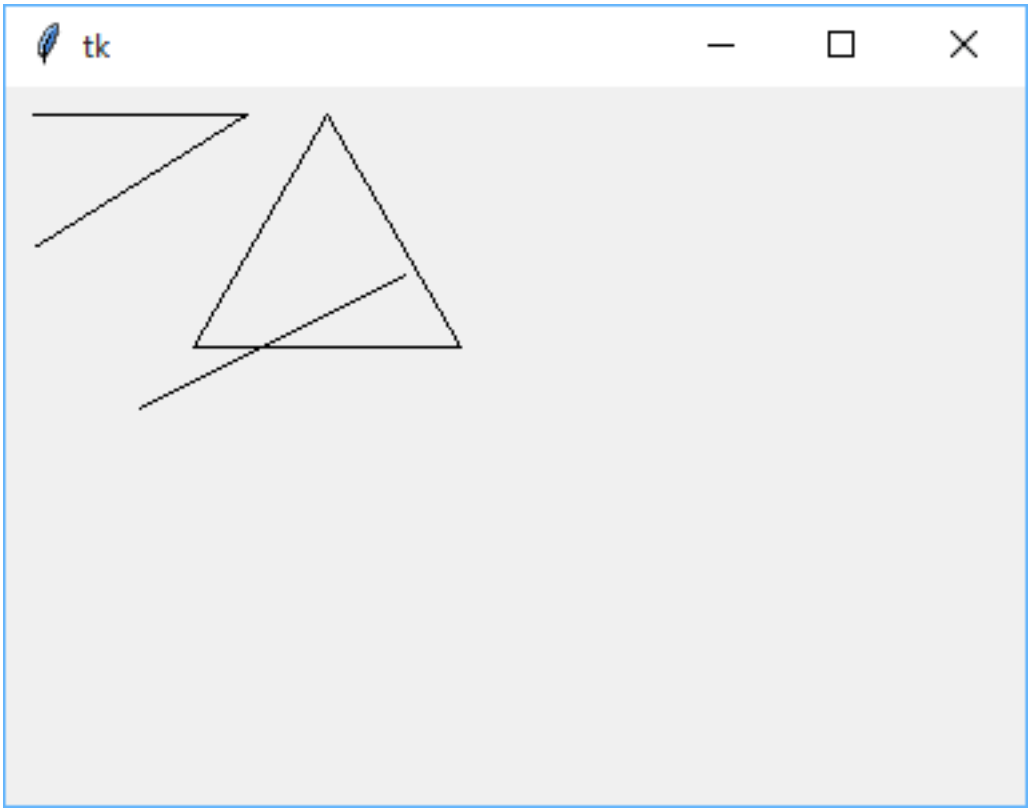
Po opätovnom spustení (F5) sa stará grafická plocha zatvorí a otvorí sa nová aj s nakreslenou úsečkou.

Pri práci s grafikou budeme často experimentovať v priamom režime a keď budeme s výsledkom spokojní, prekopírujeme príkazy zo shellu do programu v editovacom okne a prípadne znovu spustíme pomocou F5.

Volanie funkcie `canvas.create_line()` môže spájať čiarami aj viac ako 2 body, napr.

```
>>> canvas.create_line(10, 10, 90, 10, 10, 60)
>>> canvas.create_line(120, 10, 70, 97, 170, 97, 120, 10)
```

Nakreslí ďalšie dve lomené čiary, pričom druhá z nich je uzavretá a tvorí trojuholník:



Pri kreslení čiar môžeme ešte špecifikovať aj ďalšie parametre tejto kresby: za postupnosť bodov môžeme uviesť hrúbku nakreslenej čiary (parameter `width=`) a jej farbu (parameter `fill=`). Zrejme, keď tieto dva parametre nešpecifikujeme, kreslí sa čierna čiara hrúbky 1. Napr.

```
>>> canvas.create_line(10, 100, 110, 100, width=10)          # hrúbka čiary␣
↪je 10, farba je čierna
>>> canvas.create_line(10, 130, 110, 130, fill='red')        # hrúbka čiary␣
↪je 1, farba je červená
>>> canvas.create_line(10, 160, 110, 160, width=7, fill='blue')     # hrúbka␣
↪je 7, farba je modrá
```

Farby v `tkinter` zadávame ako znakové reťazce názvov farieb v angličtine. Kompletný zoznam akceptovaných mien nájdeme v súbore `rgb.txt`, ktorý sa nachádza v niektorom z priečinkov inštalovaného Pythonu. Najčastejšie budeme používať tieto farby:

- `'white'`
- `'black'`
- `'red'`
- `'blue'`
- `'green'`
- `'yellow'`

funkcia `canvas.create_line()`

`canvas.`**`create_line`**(*x*, *y*, *x*, *y*, ...)
`canvas.`**`create_line`**(*x*, *y*, *x*, *y*, ..., *width=číslo*, *fill=farba*)

Parametre

- **`x,y`** – dvojica súradníc jedného bodu lomenej čiary
- **`width=číslo`** – nastavenie hrúbky čiary, ak tento parameter chýba, predpokladá sa hrúbka 1
- **`fill=farba`** – nastavenie farby čiary, ak tento parameter chýba, predpokladá sa farba `'black'`

Funkcia `canvas.create_line()` kreslí lomenú čiaru, ktorá sa môže skladať aj z niekoľkých navzájom napojených úsečiek. Postupnosť dvojíc súradníc by mala obsahovať aspoň dva body (teda aspoň jednu úsečku). Číselné hodnoty pre súradnice môžu byť aj desatinné čísla.

Kreslenie obdĺžnika

Obdĺžniky kreslíme pomocou funkcie `canvas.create_rectangle()`. V tejto funkcii sú parametrami súradnice dvoch protiľahlých vrcholov kresleného obdĺžnika. Jej tvar je:

- `canvas.create_rectangle(x1,y1,x2,y2)`
- strany tohto obdĺžnika sú rovnobežné so súradnými osami

Napr.

```
>>> canvas.create_rectangle(50, 120, 150, 70)
```

Nakreslí obdĺžnik s vrcholmi so súradnicami `(50,120)` a `(150,70)`. Dĺžky strán tohto obdĺžnika sú `100` a `50`. Zrejme ďalšie dva vrcholy majú súradnice `(50,70)` a `(150,120)`.

Parametrami grafických funkcií nemusia byť len celočíselné konštanty, ale aj výrazy s premennými. napr.

```
>>> x, y, a = 80, 60, 35
>>> canvas.create_rectangle(x, y, x+a, y+a)
```

Nakreslí štvorec, ktorého súradnice ľavého horného vrcholu sú `(x,y)`, t.j. `(80,60)` a jeho strana je `35`. Podobne:

```
>>> x, y, a, b = 150, 90, 60, 100
>>> canvas.create_rectangle(x-a/2, y-b/2, x+a/2, y+b/2)
```

Nakreslí obdĺžnik, ktorého **súradnice stredu** sú `(x,y)`, t.j. `(150,90)` a jeho strany sú `60` a `100`. Všimnite si, že súradnice sme tu zadali ako desatinné čísla.

Takto kreslené obdĺžniky (resp. štvorce) majú nevyplnené vnútro. Tiež obrys je je tenká čierna čiara hrúbky 1. Pri kreslení obdĺžnikov môžeme ešte špecifikovať aj ďalšie parametre tejto kresby: za postupnosť bodov môžeme uviesť ďalšie doplnkové parametre: hrúbku obrysu (parameter `width=`), farbu obrysu (parameter `outline=`) a výplň obdĺžnika (parameter `fill=`). Napr.

```
>>> canvas.create_rectangle(10, 10, 100, 60, width=5)
>>>                     # obrys má hrúbku 10, farba je čierna, výplň nie je
>>> canvas.create_rectangle(120, 10, 200, 40, fill='red')
>>>                     # čierny obrys hrúbky 1, výplň je červená
>>> canvas.create_rectangle(10, 100, 110, 160, outline='', fill='blue')
>>>                     # bez obrysu, výplň je modrá
```

Všimnite si hodnotu parametra `outline=''`, ktorá označuje "priesvitný", t.j. žiaden obrys. Podobne by fungoval aj `fill=''`, ktorý označuje obdĺžnik bez výplne.

funkcia `canvas.create_rectangle()`

`canvas.`**`create_rectangle`**(*x*, *y*, *x*, *y*)
`canvas.`**`create_rectangle`**(*x*, *y*, *x*, *y*, *width=číslo*, *fill=farba*, *outline=farba*)

Parametre

- **`x,y`** – dvojica súradníc jedného vrcholu obdĺžnika
- **`width=číslo`** – nastavenie hrúbky čiary obrysu, ak parameter chýba, predpokladá sa hrúbka 1
- **`fill=farba`** – nastavenie farby výplne, ak parameter chýba, predpokladá sa priesvitná farba, t.j. `''`
- **`outline=farba`** – nastavenie farby čiary obrysu, ak tento parameter chýba, predpokladá sa farba `'black'`, prázdny reťazec `''` označuje obdĺžnik bez obrysu

Funkcia `canvas.create_rectangle()` na základe dvoch bodov v ploche nakreslí obdĺžnik, ktorého strany sú rovnobežné so súradnicovými osami. Číselné hodnoty pre súradnice môžu byť aj desatinné čísla.

Kreslenie elipsy

Parametre príkazu `canvas.create_oval()` vychádzajú z kreslenia obdĺžnika. Pre `tkinter` sú to presne rovnaké parametre, len elipsa ich trochu inak interpretuje: z dvoch protiľahlých vrcholov nenakreslí obdĺžnik ale **vpísanú** elipsu, t.j. elipsu, ktorá leží vo vnútri "mysleného" obdĺžnika a dotýka sa jeho strán. Parametrami príkazu sú súradnice dvoch protiľahlých vrcholov "mysleného" opísaného obdĺžnika v tvare: * `canvas.create_oval(x1,y1,x2,y2)` * strany tohto mysleného obdĺžnika sú rovnobežné so súradnými osami

Napr.

```
>>> canvas.create_oval(50, 120, 150, 70)
```

Nakreslí elipsu so stredom `(100,95)` a s poloosami `50` a `25`, keďže "myslený" opísaný obdĺžnik má vrcholy so súradnicami `(50,120)` a `(150,70)`. Ak by sme spolu s elipsou nakreslili aj tento opísaný obdĺžnik, lepšie by sme videli ich vzťah:

```
>>> canvas.create_rectangle(50, 120, 150, 70, outline='red')
>>> canvas.create_oval(50, 120, 150, 70)
```

Ďalšie parametre `width=`, `outline=` a `fill=` majú presne rovnaký význam ako pre obdĺžnik: hrúbka obrysu elipsy, farba obrysu (môže chýbať, ak má hodnotu `''`) a farba výplne (môže chýbať, ak má hodnotu `''`). Ak nešpecifikujeme žiaden parameter, tak sa kreslí elipsa s čiernym obrysom hrúbky 1, ktorá je bez výplne.

Ak zadáme elipsu, ktorá je vpísaná do štvorca, nakreslí sa kružnica. Všimnite si:

```
import tkinter

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

x, y = 150, 120
for r in range(100, 0, -10):
    canvas.create_oval(x-r, y-r, x+r, y+r, fill='red')
```

Tento program nakreslí 10 sústredných kruhov (so spoločným stredom (150, 120)) a s polomermi 100, 90, 80, ..., 20, 10. Kruhy sú vyplnené červenou farbou a preto ich kreslíme od najväčšieho po najmenší, inak by väčšie kruhy zakryli menšie. Ak by sme chceli tieto kruhy striedavo zafarbovať dvomi rôznymi farbami, môžeme využiť vymieňanie obsahov dvoch premenných:

```
x, y = 150, 120
farba, farba1 = 'red', 'yellow'
for r in range(100, 0, -10):
    canvas.create_oval(x-r, y-r, x+r, y+r, fill=farba)
    farba, farba1 = farba1, farba
```

funkcia `canvas.create_oval()`

`canvas.`**`create_oval`**(*x*, *y*, *x*, *y*)
`canvas.`**`create_oval`**(*x*, *y*, *x*, *y*, *width=číslo*, *fill=farba*, *outline=farba*)

Parametre

- **`x,y`** – dvojica súradníc jedného vrcholu "opísaného obdĺžnika" elipsy
- **`width=číslo`** – nastavenie hrúbky čiary obrysu, ak parameter chýba, predpokladá sa hrúbka 1
- **`fill=farba`** – nastavenie farby výplne, ak parameter chýba, predpokladá sa priesvitná farba, t.j. `''`
- **`outline=farba`** – nastavenie farby čiary obrysu, ak tento parameter chýba, predpokladá sa farba `'black'`, prázdny reťazec `''` označuje elipsu bez obrysu

Funkcia `canvas.create_oval()` na základe dvoch bodov "opísaného obdĺžnika" nakreslí elipsu. Strany takéhoto mysleného obdĺžnika sú rovnobežné sú súradnicovými osami.

Kreslenie polygonu

Polygónom rozumieme uzavretú krivku, ktorá môže byť vyplnená nejakou farbou. Súradnicami sa tento príkaz podobá na `canvas.create_line()` a ďalšie doplnkové parametre (`width=`, `outline=` a `fill=`) má rovnaké ako napr. `canvas.create_rectangle()`.

- parametrom je postupnosť súradníc v tvare `canvas.create_line(x1,y1,x2,y2,...)`
- aby mal polygon zmysel, postupnosť súradníc musí obsahovať aspoň 3 body

Napr.

```
>>> canvas.create_polygon(50, 120, 150, 70, 200, 200)
```

Spojí tromi úsečkami tri body `(50,120)`, `(150,70)` a `(200,200)` a potom tento trojuholník vyfarbí farbou výplne. Keďže sme nenastavili ani farbu obrysu ani farbu výplne, útvar sa vyplní čiernou farbou a obrys je vypnutý (ako keby bolo nastavené `outline=''` a `fill='black'`).

Napr.

```
canvas.create_polygon(10, 80, 110, 80, 30, 150, 60, 30, 90, 150, fill='red')
```

nakreslí červenú päťcípu hviezdu, ale bez obrysu. Ak chceme obrys, zadáme

```
canvas.create_polygon(10, 80, 110, 80, 30, 150, 60, 30, 90, 150, fill='red',␣
↪outline='blue')
```

Teraz ukážeme, že ak do nejakých premenných priradíme dvojice čísel, teda súradnice nejakých bodov v rovine, tieto potom môžeme používať aj v grafických príkazoch. Zadefinujme 4 vrcholy v ploche a potom ich spojíme a vyplníme šedou farbou:

```
b1 = (50, 20)
b2 = (150, 20)
b3 = (150, 80)
b4 = (50, 80)
canvas.create_polygon(b1, b2, b3, b4, fill='gray', outline='blue')
```

Vďaka takémuto zápisu s premennými by mohlo byť pre vás čitateľné aj:

```
canvas.create_polygon(b1, b2, b4, b3, fill='red')
canvas.create_polygon(b1, b3, b2, b4, fill='green')
```

Ak máme v premennej dvojicu čísel, môžeme ju "rozobrať" na dve samostatné čísla a priradiť ich do dvoch premenných, napr. takto:

```
a = (50, 20)
ax, ay = a
b = (ax+100, ay)
```

Dokonca budú fungovať aj takéto for-cykly:

```
for bod in (130,30), (150,60), (200,90):
    canvas.create_rectangle(bod, (175, 75))

for x,y  in (100,100), (150,150), (200,160):
    canvas.create_oval(x-20, y-20, x+20, y+20)
```

Prvý for-cyklus nakreslí tri obdĺžniky, ktoré majú jeden vrchol `(175,75)` spoločný a druhý je jeden z `(130,30)`, `(150,60)`, `(200,90)`. Druhý for-cyklus nakreslí 3 kružnice s polomerom `20`, ak poznáme stredy týchto kružníc: `(100,100)`, `(150,150)`, `(200,160)`. Všimnite si, že tento cyklus má dve premenné cyklu `x` a `y` a paralelne sa do nich priradzujú dvojice zadaných čísel.

funkcia `canvas.create_polygon()`

`canvas.`**`create_polygon`**(*x*, *y*, *x*, *y*, ...)
`canvas.`**`create_polygon`**(*x*, *y*, *x*, *y*, ..., *width=číslo*, *fill=farba*, *outline=farba*)

Parametre

- **`x,y`** – dvojica súradníc jedného vrcholu obrysu n-uholníka
- **`width=číslo`** – nastavenie hrúbky čiary obrysu, ak parameter chýba, predpokladá sa hrúbka 1
- **`fill=farba`** – nastavenie farby výplne, ak parameter chýba, predpokladá sa čierna, t.j. `'black'`, prázdny reťazec `''` označuje n-uholníka bez výplne
- **`outline=farba`** – nastavenie farby čiary obrysu, ak tento parameter chýba, predpokladá sa priesvitná farba `''`

Funkcia `canvas.create_polygon()` na základe postupnosti bodov nakreslí n-uholník.

Písanie textu

Pomocou funkcie `canvas.create_text()` môžeme do grafickej plochy písať aj texty. Prvé dva parametre funkcie sú súradnice stredu vypisovaného textu a ďalším doplnkovým parametromv je samotný text v tvare:

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

Napr.

```
>>> canvas.create_text(150, 120, text='Python')
```

Na súradnice `(150,120)` sa vypíše text `'Python'` (čiernou) - použije sa pritom "default" font aj veľkosť písma. Zadané súradnice určujú stred vypisovaného textu. Pri písaní textu môžeme ešte špecifikovať aj ďalšie parametre: môžeme uviesť font (parameter `font=`) a aj farbu textu (parameter `fill=`). Font definujeme ako znakový reťazec, ktorý obsahuje (jednoslovný) názov písma a veľkosť písma, napr. v tvare `'arial 20'`. Tento reťazec môže obsahovať aj informáciu o tvare písma `'bold'` a `'italic'`.

Napr.

```
>>> canvas.create_text(150, 50, text='Python', fill='blue', font='arial 30␣
↪bold')
```

funkcia `canvas.create_text()`

`canvas.`**`create_text`** (*x*, *y*, *text='text'*)
`canvas.`**`create_text`** (*x*, *y*, *text='text'*, *font='písmo'*, *fill=farba*, *angle=číslo*)

Parametre

- **`x,y`** – dvojica súradníc stredu vypisovaného textu
- **`text='text'`** – zadaný text
- **`fill=farba`** – nastavenie farby textu
- **`font='písmo'`** – nastavenie typu písma a aj jeho veľkosti
- **`angle=číslo`** – otočenie výpisu o zadaný uhol v stupňoch (tento parameter nemusí fungovať na počítačoch Mac)

Funkcia `canvas.create_text()` na zadanú súradnicu bodu vypíše text.

kreslenie obrázku

Aby sme do plochy nakreslili 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 vykreslenie do grafickej plochy ľubovoľný počet-krát.

Samotná funkcia `canvas.create_image()` má tri parametre: prvé dva sú súradnice stredu vykresľovaného obrázku a ďalší doplnkový 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)
```

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)
```

tieto parametre plochy môžeme dodatočne aj meniť, aj keď je už v ploche niečo nakreslené. Vtedy použijeme takýto formát:

```
>>> canvas['bg'] = 'yellow'
>>> sirka = int(canvas['width'])       # zapamätaj si momentálnu šírku plochy
>>> canvas['width'] = 600              # zmeň šírku
>>> canvas['height'] = 400
```

3.1 Zmeny nakreslených útvarov

Všetky útvary, ktoré kreslíme do grafickej plochy si systém pamätá tak, že ich dokáže dodatočne meniť (napr. 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.

Zmazanie nakresleného útvaru

funkcia `canvas.delete()`

- slúži na zmazanie ľubovoľného nakresleného útvaru
- jeho tvar je `canvas.delete(identifikátor)`
  - kde `identifikátor` je návratová hodnota príkazu kreslenia útvaru
- ak ako `identifikátor` použijeme reťazec `'all'`, príkaz zmaže všetky doteraz nakreslené útvary

Napr.

```
>>> id1 = canvas.create_line(10, 20, 30, 40)
>>> id2 = canvas.create_oval(10, 20, 30, 40)
>>> canvas.delete(id1)
```

zmaže prvý grafický objekt, t.j. úsečku, pričom druhý objekt kružnica ostáve bez zmeny.

Posúvanie útvarov

Pomocou identifikátora útvaru ho môžeme posúvať ľubovoľným smerom. Ostatné útvary sa pri tom nehýbu.

funkcia `canvas.move()`

- slúži na posúvanie ľubovoľného nakresleného útvaru
- jeho tvar je `canvas.move(identifikátor,dx,dy)`
  - kde `identifikátor` je návratová hodnota príkazu kreslenia útvaru
  - `dx` a `dy` označujú číselné hodnoty zmeny súradníc útvaru
- posúvaný útvar môže byť ľubovoľne komplikovaný (môže sa skladať aj z väčšieho počtu bodov), príkaz `canvas.move()` posunie všetky vrcholy útvaru
- ak ako `identifikátor` použijeme reťazec `'all'`, príkaz posunie všetky doteraz nakreslené útvary

Napr.

```
>>> id1 = canvas.create_line(10, 20, 30, 40)
>>> id2 = canvas.create_oval(10, 20, 30, 40)
>>> canvas.move(id1, -5, 10)
```

posunie prvý nakreslený útvar, teda úsečku, druhý útvar (kružnicu) pri tom nehýbe.

Zmena parametrov útvaru

Nakresleným útvarom môžeme zmeniť ľubovoľné doplnkové parametre pomocou príkazu `canvas.itemconfig()`.

funkcia `canvas.itemconfig()`

- slúži na zmenu ľubovoľných doplnkových parametrov nakresleného útvaru
- má tvar `canvas.itemconfig(identifikátor,parametre)`
  - kde `identifikátor` je návratová hodnota príkazu kreslenia útvaru
  - `parametre` sú ľubovoľné doplnkové parametre pre daný útvar

Zhrňme doplnkové parametre útvarov, s ktorými sme sa doteraz stretli:

`canvas.create_line()`	
	`width=` hrúbka obrysu
	`fill=` farba obrysu
`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_text()`	
	`text=` vypisovaný text
	`font=` písmo a veľkosť
	`fill=` farba textu
	`angle=` uhol otočenia
`canvas.create_polygon()`	
	`width=` hrúbka obrysu
	`outline=` farba obrysu
	`fill=` farba výplne
`canvas.create_image()`	
	`image=` obrázkový objekt

Napr.

```
>>> id1 = canvas.create_line(10, 20, 30, 40)
>>> id2 = canvas.create_oval(10, 20, 30, 40)
>>> canvas.itemconfig(id1, width=5, fill='blue')
>>> canvas.itemconfig(id2, outline='', fill='red')
```

Zmena súradníc

Okrem posúvania útvaru môžeme zmeniť aj jeho kompletnú postupnosť súradníc. Napr. pre `canvas.create_line()` alebo `canvas.create_polygon()` môžeme zmeniť aj počet bodov útvaru.

funkcia `canvas.coords()`

- slúži na zmenu súradníc nakresleného útvaru
- má tvar `canvas.coords(identifikátor,postupnost')`
  - kde `identifikátor` je návratová hodnota príkazu kreslenia útvaru
  - `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)
```

3.2 Generátor náhodných čísel

V mnohých našich programoch sa nám môže hodiť, keď niektoré premenné nebudú mať pri každom spustení rovnakú hodnotu, ale zakaždým dostanú podľa nejakých pravidiel nejakú náhodnú hodnotu. Využijeme modul `random`, ktorý obsahuje niekoľko užitočných funkcií. Aby sme mohli používať funkcie z tohto modulu, musíme na začiatku programu zapísať

```
import random
```

Vznikne premenná `random`, ktorá obsahuje referenciu na tento modul a pomocou nej budeme pristupovať k funkciám v tomto module. Každá takáto funkcia bude teda začínať `random` a za bodkou bude uvedené meno funkcie.

Modul obsahuje niekoľko užitočných funkcií, my budeme najčastejšie používať tieto tri.

funkcia `random.randint()`

- funkcia má dva parametre: hranice intervalu čísel
- vyberie náhodnú hodnotu z tohto intervalu, pričom sa do výberu počítajú aj hraničné body intervalu

funkcia `random.randrange()`

- funkcia má 1, 2 alebo 3 parametre s rovnakým významom ako `range()`
- vyberie náhodnú hodnotu z tohto rozsahu
- napr. `random.randrange(100)`, `random.randrange(10,100)`, `random.randrange(10,100,5)`

funkcia `random.choice()`

- funkcia má jeden parameter: ľubovoľnú postupnosť hodnôt
- vyberie náhodnú hodnotu z tejto postupnosti
- napr. `random.choice(('red','blue','green'))`

Príklad použitia:

```
import tkinter
import random

canvas = tkinter.Canvas(bg='white', width=400, height=300)
canvas.pack()

for i in range(random.randint(10 ,20)):
    x = random.randrange(400)
    y = random.randrange(300)
    r = 10
    farba = random.choice(('red', 'blue', 'green'))
    canvas.create_oval(x-r, y-r, x+r, y+r, fill=farba)
```

Pozdržanie výpočtu

Chceme napísať program, ktorý najprv nakreslí obrázok a potom ho 100-krát posunie v pravo o nejaký malý úsek. Radi by sme to videli ako animáciu

```
import tkinter

canvas = tkinter.Canvas(width=600, height=200)
canvas.pack()

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

for x in range(100):
    canvas.move(1, -5, 0)
```

Po spustení nevidíme žiadnu animáciu, ale len výsledný efekt, t.j. obrázok je v cieľovej pozícii.

Ak využijeme ďalšiu metódu grafickej plochy `after`, dokážeme pozdržať výpočet.

funkcia `canvas.after()`

- funkcia má jeden číselný parameter: počet tisícin sekúndy, na ktorý sa výpočet na tomto mieste pozdrží
- napr. `canvas.after(500)` pozdrží výpočet o **0,5** sekundy

A ešte jedna dôležitá vec: aby sme počas pozdržaného času videli zrealizovanú zmenu v grafickej ploche, musíme zavolať špeciálny príkaz `canvas.update()`, ktorý zabezpečí zobrazenie zmeny, t.j. animáciu

```
import tkinter

canvas = tkinter.Canvas(width=600, height=200)
canvas.pack()

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

for x in range(100):
    canvas.move(1, -5, 0)
    canvas.update()
    canvas.after(100)
```

Keďže program vykoná 100-krát pozdržanie 0,1 sekundy, celá animácia bude trvať 10 sekúnd.

3.3 spustenie programu z operačného systému

Ak by sme nejaký grafický program spustili priamo z operačného systému (napr. dvojkliknutím na súbor), napr.

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()
canvas.create_line(50, 120, 150, 70)
```

grafické okno by sa otvorilo a okamžite aj zatvorilo, lebo program po nakreslení úsečky už nemusí na nič ďalšie čakať a teda skončí. Z tohto dôvodu sa na koniec programu zapisuje nový riadok `canvas.mainloop()`:

```
import tkinter

canvas = tkinter.Canvas()
canvas.pack()
canvas.create_line(50, 120, 150, 70)
canvas.mainloop()
```

Takýto program

- ak spustíme z operačného systému, otvorí grafické okno, nakreslí úsečku a čaká na zatvorenie tohto okna
- ak spustíme z editora (F5), tiež sa otvorí grafické okno s nakreslenou úsečkou a čaká sa na zatvorenie tohto okna; toto ale znamená, že shell je počas tohto čakania zablokovaný (uvedomte si, že toto zablokovanie spôsobilo volanie `canvas.mainloop()`)
  - V niektorých inštaláciách Pythonu sa grafické okno zobrazí len vtedy, keď sa náš program ukončí príkazom `canvas.mainloop()` - t.j. neumožňuje sa ďalej komunikovať cez príkazový riadok (shell). Žiaľ v tomto prípade nemôžete experimentovať s grafickými príkazmi v shell, ale musíte ich zadávať len do programu.

KAPITOLA 4

Podmienené príkazy

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 platí, vykonaj 1. skupinu príkazov
    prikaz
    prikaz
    ...
else:                     # ak podmienka neplatí, vykonaj 2. skupinu príkazov
    prikaz
    prikaz
    ...
```

V našom príklade sú 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** za 90 a viac
- známka **B** za 80
- známka **C** za 70
- známka **D** za 60
- známka **E** za 50
- známka **Fx** za menej ako 50

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 platí, vykonaj 1. skupinu␣
↪príkazov
    prikaz
    ...
elif podmienka_2:           # ak podmienka_1 neplatí, ale platí podmienka_2, .
↪..
    prikaz
    ...
elif podmienka_3:           # ak podmienka_1 a podmienka_2 neplatia, ale␣
↪platí podmienka_3, ...
    prikaz
    ...
else:                       # ak žiadna z podmienok neplatí, ...
    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 platí, vykonaj skupinu príkazov
    prikaz
    prikaz
    ...                   # ak podmienka neplatí, nevykonaj nič
```

Zrejme každý `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. Program na náhodné pozície nakreslí 1000 malých krúžkov, pričom tie z nich, ktoré sú v ľavej polovici plochy budú červené a zvyšné v pravej polovici (teda `else` vetva) budú modré:

```
import tkinter, 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 a červené a modré bude len v hornej polovici. Jedno z možných riešení:

```
import tkinter, 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, 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 - majú obvyklý význam z matematiky:

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

Všimnite si, že podmienky ktoré platia majú hodnotu `True` a ktoré neplatia `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 logické operácie

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

Logický súčin **a súčasne**:

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

Logický súčet **alebo**:

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

Negácia:

A	not A
False	True
True	False

Ale fungujú aj pre skoro všetky ďalšie typy. V tomto prípade Python pre každý typ definuje prípady, ktoré sa chápu

ako `False` a tým je aj určené, že všetky ďalšie hodnoty tohto typu sa chápu ako `True`. Ukážme prípady pre doteraz známe typy, ktoré označujú logickú hodnotu `False`:

typ	`False`	`True`
int	== 0	! = 0
float	== 0.0	! = 0.0
str	== ‘’	! = ‘’

Toto znamená, že v prípadoch, keď Python očakáva logickú hodnotu (napr. v príkaze `if`, alebo v operáciách `and`, `or`, `not`) môžeme uvádzať aj hodnoty iných typov. Napr.

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

Logické operácie majú ale trochu rozšírenú interpretáciu:

operácia: prvý `and` druhý

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

operácia: prvý `or` druhý

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

operácia: `not` prvý

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

Napr.

```
>>> 1+2 and 3+4
7
>>> 'ahoj' or 'Python'
'ahoj'
>>> '' or 'Python'
'Python'
>>> 3<4 and '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 náhodný štvorec. Toto opakujeme 10-krát:

```
import tkinter, 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 (`random.randint(1,7)`) a pre čísla 1,2,3 by sme kreslili kružnicu inak štvorec.

Túto ideu môžeme využiť aj pre 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, 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:
        pocet += 1
        print(delitel, end=' ')
print()
```

```
print('počet deliteľov:', pocet)
```

Výstup môže byť napríklad takýto:

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

Malou modifikáciou tejto úlohy vieme urobiť ďalšie dva programy: jeden, ktorý zisťuje, či je číslo prvočíslo a druhý, zisťuje, či je dokonalé (súčet všetkých deliteľov menších ako číslo sa rovná samotnému číslu):

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

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 ===
```

4.2 while-cyklus

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

```
while podmienka:              # opakuj príkazy, kým platí podmienka
    prikaz
    prikaz
    ...
```

Vidíme podobnosť s podmieneným príkazom - vetvením.

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.

Program bude vedľa seba kresliť zväčšujúce sa štvorce so stranami 10, 20, 30, ... ale tak, aby posledne nakreslený štvorec "nevypadol" z plochy - skončíme skôr, ako by sme nakreslili štvorec, ktorý sa celý nezmestí. 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:

```
sirka = int(input('šírka plochy: '))

import tkinter
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
```

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
2. ak je podmienka splnená (pravdivá), postupne sa vykonajú všetky príkazy
3. po ich vykonaní 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, príkazy sa vykonávajú

Vyššie sme písali 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 číslo. Ak si uvedomíme, že na zisťovanie prvočísla nepotrebujeme skutočný počet deliteľov, ale stačí nám 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, 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`.

Zisťovanie druhej odmocniny

Ukážeme, ako zistíme druhú odmocninu čísla aj bez `math.sqrt` a `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` 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.

4.3 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 klábvesov **Ctrl/C**.

Aj nasledovný cyklus je úmyselne nekonečný:

```
while 1:
    pass
```

V Pythone existuje príkaz `break`, ktorý môžeme použiť v tele cyklu a vtedy sa zvyšok cyklu nevykoná, ale pokračuje sa až na príkazoch za cyklom (funguje to aj pre for-cykly nielen pre while). Najčastejšie sa `break` vyskytuje v príkaze `if`, napr.

```
sucet = 0
while True:
    retazec = input('zadaj cislo: ')
    if retazec == '':
        break
    sucet += int(retazec)
print('sucet precitanych cisel =', sucet)
```

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

KAPITOLA 5

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()`

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():
    prikaz
    prikaz
    ...
```

Keď zapíšeme definíciu funkcie, zatiaľ sa nič z bloku príkazov (hovoríme tomu **telo funkcie**) 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 v programovom režime:

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

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

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

- 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 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

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 programe (nie v interaktívnom režime), táto hodnota referencie by sa nevypísala, ale odignorovala. Toto býva dosť častá chyba, 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" (objekt nie je zavolateľný).

Hotové funkcie, s ktorými sme doteraz pracovali, napr. `print()` alebo `random.randin()`, 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 formálneho parametra (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 do nej hodnora **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

Už vieme, že priraďovací príkaz vytvára premennú a referenciou ju spojí s 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` = 26710 vypíše:

```
>>> vypis_sucet(10)
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55
```

Počas behu vzniknú 2 lokálne premenné 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 zrušia.

5.1 Menný priestor

Aby sme lepšie pochopili ako naozaj fungujú **lokálne premenné**, musíme rozumieť, čo to je a ako funguje **menný priestor** (namespace). Najprv trochu terminológie: všetky identifikátory v Pythone sú jedným z troch typov (Python má na ne 3 rôzne tabuľky mien):

- **štandardné**, napr. `int`, `print`, ...
  - hovorí sa tomu **builtins**
- **globálne** - definujeme ich na najvyššej úrovni mimo funkcií, napr. funkcia `vypis_sucet`
  - hovorí sa tomu **main**
- **lokálne** - vznikajú počas behu funkcie

Tabuľka štandardných mien je je len jedna, tiež tabuľka globálnych mien 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 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 nejakú hodnotu
>>> print
'ahoj'
>>> print('ahoj')
...
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 **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 v tele funkcie sa 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 funkcií: `vypis_hviezdiciek` a `trojuholnik`. Zavoláme funkciu `trojuholnik`:

```
>>> trojuholnik(5)
```

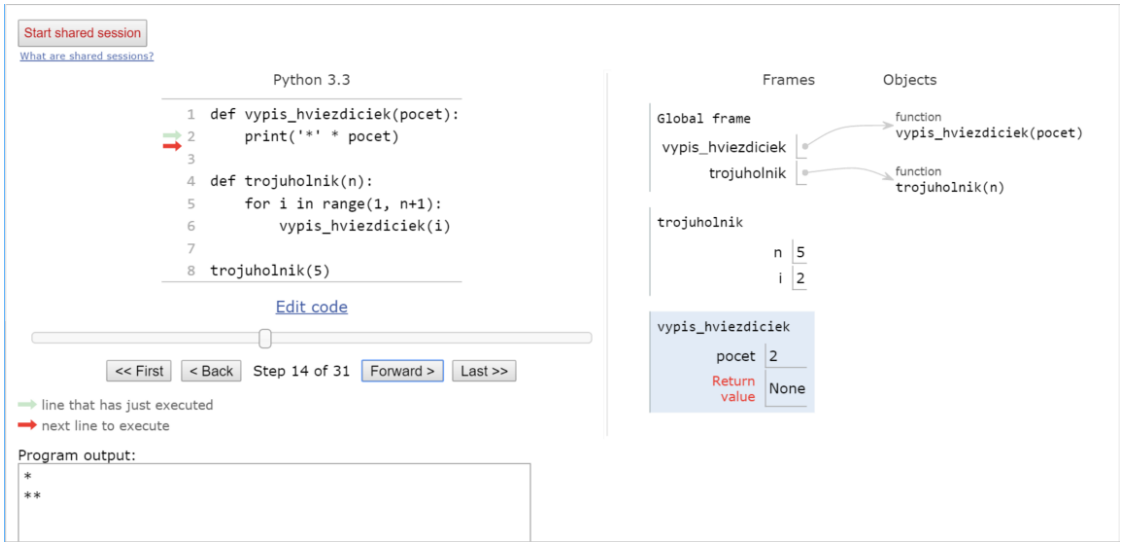
Najprv sa pre túto funkciu vytvorí jej menný priestor (lokálna tabuľka mien) s dvomi lokálnymi premennými: `n` a `i`. Teraz pri každom (vnorenom) volaní `vypis_hviezdiciek(i)` sa pre túto funkciu:

- vytvorí nový menný priestor s jedinou premennou `pocet`
- vykoná sa príkaz `print()`
- na koniec sa zruší jej menný priestor, t.j. premenná `pocet`

Môžeme odkrokovať pomocou http://www.pythontutor.com/visualize.html#mode=edit (zapneme voľbu `Python 3.3`):

- 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 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 nejakú hodnotu vrátiť ako výsledok funkcie, musí sa v jej tele objaviť príkaz `return`, napr.

```
def meno(parameter1, parameter2):    # zapamätaj si blok príkazov ako nový␣
↪príkaz
    prikaz
    prikaz
    ...
    return hodnota                   # takáto funkcia bude vracať výslednú␣
↪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):
    koruny = round(eura * 30.126, 2)
    return koruny
```

môžeme otestovať:

```
>>> print('dostal si', 123, 'euro, čo je', eura_na_koruny(123), 'korún')
dostal si 123 euro, čo je 3705.5 korún
```

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 lokálna premenná) môžeme v tele funkcie modifikovať.

Už sme videli, že funkcie rozlišujeme dvoch typov:

- 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 nejakého čísla, ktorú 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('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. Využije 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
```

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
```

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
```

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.

```
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 to bude stačiť riešiť takto jednoducho.

Napriek tomuto niektoré funkcie môžu fungovať rôzne pre rôzne typy, napr.

```
def urob(a, b):
    return 2 * a + 3 * b
```

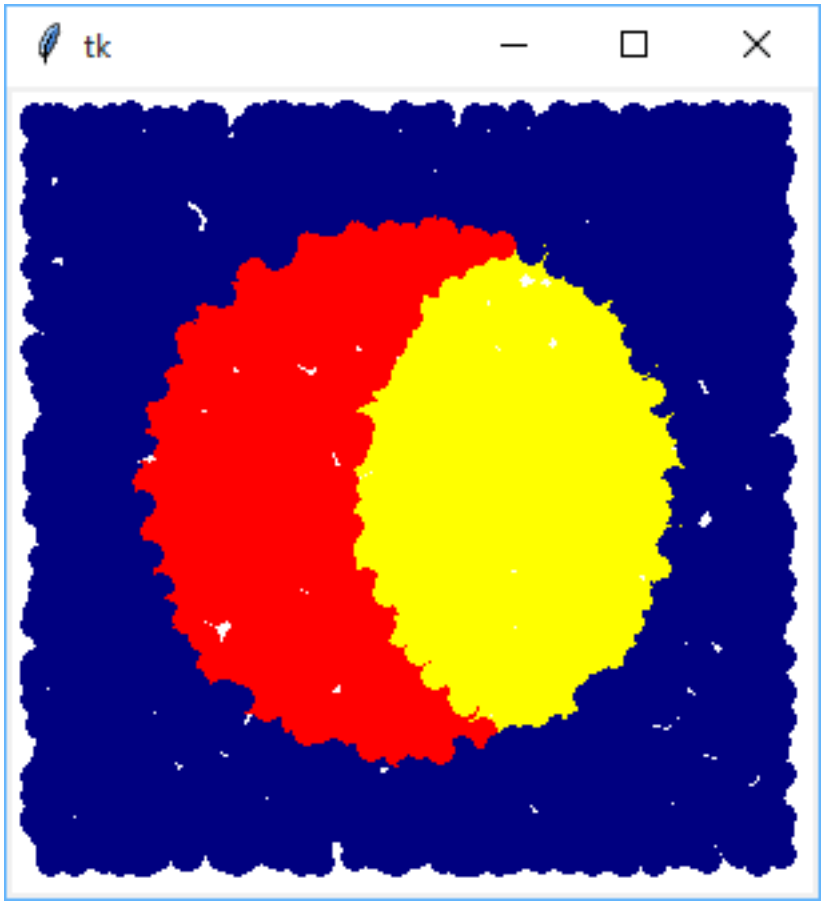
niekedy funguje pre čísla aj pre reťazce.

Grafické funkcie

Zadefinujeme funkcie, pomocou ktorých sa nakreslí 5000 náhodných farebných bodiek, ktoré budú zafarbené podľa nejakých pravidiel:

```
def vzd(x1, y1, x2, y2):
    return ((x1-x2)**2 + (y1-y2)**2)**.5

def kresli_bodku(x, y, farba):
    canvas.create_oval(x-5, y-5, x+5, y+5, fill=farba, outline='')

def farebne_bodky(pocet):
    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')

import tkinter, random

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. Ďalšia funkcia 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 ma 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:



5.3 Náhradná hodnota parametra

Naučíme sa zadefinovať parametre funkcie tak, aby sme pri volaní nemuseli uviesť všetky parametre, 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 to, 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.

Teraz môžeme zapísať aj takéto volania tejto funkcie:

```
kresli_bodku(100, 200, 'blue', 3)
kresli_bodku(150, 250, 'blue')       # r bude 5
kresli_bodku(200, 200)               # farba bude 'red' a r bude 5
```

5.4 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 viac 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.

Formátovanie textu

Pripomeňme si, ako vieme formátovať reťazce pomocou metódy `format`: niekoľko základných zápisov:

```
>>> 'Ahoj {}. Ako sa máš?'.format('Peter')
'Ahoj Peter. Ako sa máš?'
>>> '({}, {})'.format(150, 222)
'(150, 222)'
>>> 'prvý={}, druhý={:3} a tretí={:7}'.format('ahoj', 6*7, 1/4)
'prvý=ahoj, druhý= 42 a tretí=   0.25'
```

Formátovanie reťazcov sa dá využiť aj na výpis celých čísel v **šestnástkovej sústave**. Už poznáme formát `'{:5}'`, ktorý označuje, že príslušná hodnota sa vypíše na šírku 5. Za dvojbodku v zátvorkách `'{}'` okrem šírky môžeme písať aj typ výpisu. Tu sa nám bude hodiť typ výpisu `'x'`, ktorý označuje výpis v šestnástkovej sústave, napr.

```
>>> '{:x}'.format(122)
'7a'
>>> '{:x}'.format(12122)
'2f5a'
```

Prípadne aj so šírkou výpisu:

```
>>> '{:3x}'.format(122)
' 7a'
>>> '{:3x}'.format(12122)
'2f5a'
```

Načo teraz potrebujeme šestnástkovú sústavu? `tkinter` totiž umožňuje zadávať farby nielen ich menami ale aj v tzv. **RGB** formáte a vtedy ich treba zapísať v šestnástkovej sústave. Tu ale platí, že takto zadané šestnástkové číslo je 6-ciferné a zľava je doplnené nulami `'0'`. Formátovanie reťazca pomocou `'{:6x}'` by kratšie ako 6-ciferné čísla doplnilo medzerami a preto využijeme ešte jednu špecialitu tohto formátu, ktoré čísla dopĺňa nulami: zapíšeme to ako `'{:06x}'`, napr.

```
>>> '{:03x}'.format(122)
'07a'
>>> '{:06x}'.format(12122)
'002f5a'
```

Farebný model RGB

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. Napr.

- pre žltú ('yellow'): keďže platí `red=255`, `green=255`, `blue=0`, šestnástkovo je to trojica `ff`, `ff`, `00`, a ako farba je to `'#ffff00'`
- pre tmavozelenú ('darkgreen'): `red=0`, `green=100`, `blue=0`, šestnástkovo je to trojica `00`, `64`, `00`, a farba=‘‘’#006400’‘‘

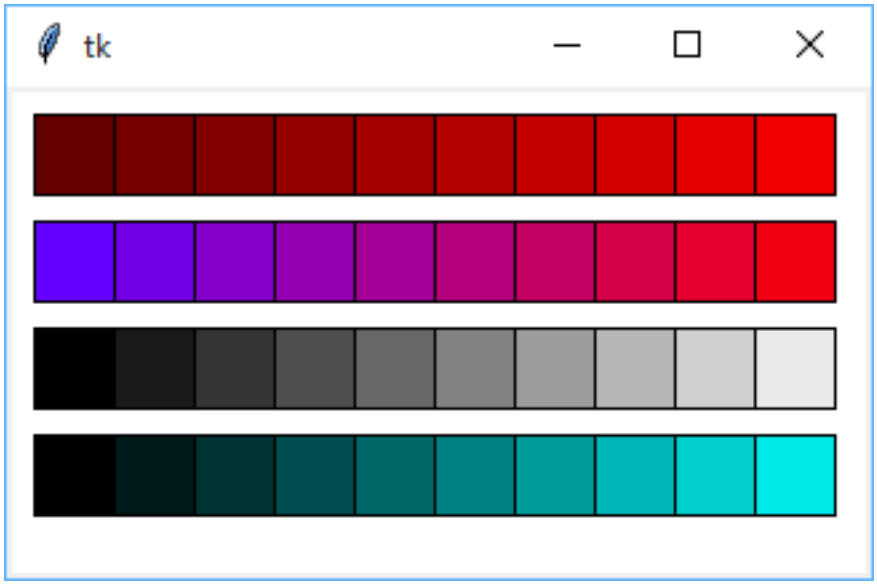
Keďže už vieme vytvárať reťazce so šestnástkovým zápisom čísel (napr. `'{:02x}'.format(číslo)`) zapíšme funkciu `rgb()`, ktorá bude vytvárať farby pomocou RGB-modelu:

```
def rgb(r, g, b):
    return '#{:02x}{:02x}{:02x}'.format(r, g, b)
```

a použitie tejto funkcie napr.:

```
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 '#{:02x}{:02x}{:02x}'.format(r, g, b)

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 '#{:06x}'.format(random.randrange(256*256*256))  # čo 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:


tk

KAPITOLA 6

Znakové reťazce

Čo už vieme o znakových reťazcoch:

- reťazec je postupnosť znakov uzavretá v `' '` alebo
- 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 `' '` alebo , platí:

- môže obsahovať ľubovoľné znaky (okrem znaku `'` v `' '` reťazci, a znaku `"` v)
- musí sa zmestiť do jedného riadka (nesmie prechádzať do druhého riadka)
- môže obsahovať špeciálne znaky (zapisujú sa dvomi znakmi, ale v reťazci reprezentujú len jeden):
  - `\n` - nový riadok
  - `\t` - tabulátor
  - `\'` - apostrof
  - `\"` - úvodzovka
  - `\\` - opačná lomka

Napríklad

```
>>> 'Monty\nPython'
'Monty\nPython'
>>> print('Monty\nPython')
Monty
Python
>>> print('Monty\\nPython')
Monty\nPython
```

Viacriadkové reťazce

Už vieme, že

- reťazec, ktorý začína trojicou buď `'''` alebo `"` môže obsahovať aj `'` a `"`, môže prechádzať cez viac riadkov (automaticky sa 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'
```

Dĺžka reťazca

Štandardná funkcia `len()` vráti dĺžku reťazca (špeciálne znaky ako `'\n'`, `'\''`, ... reprezentujú len 1 znak):

```
>>> a = 'Python'
>>> len(a)
6
>>> len('Peter\'s dog')
11
```

Túto funkciu už vieme naprogramovať aj sami, ale oproti štandardnej funkcii `len()` bude oveľa pomalšia:

```
def dlzka(retazec):
    pocet = 0
    for znak in retazec:
        pocet += 1
    return pocet
```

Operácia in

Binárna operácia `in` zisťuje, či sa zadaný podreťazec nachádza v nejakom 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
```

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.

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:

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`.

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:

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
```

Podreťazce

Indexovať môžeme nielen jeden znak, ale aj nejaký podreťazec celého reťazca, Opäť použijeme operátor indexovani, ale index obsahuje znak `':'`:

```
retʼazec[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:

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('{}:{} {}'.format(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('{}:{} {}'.format(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
```

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[ : ]
```

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 začiatku po 4. znak
'Monty'
>>> abc[-4:]                   # od 4. od konca až do konca
'thon'
>>> abc[16:]                   # indexujeme mimo ret'azca
''
```

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]
'kob ylam am alybok'
>>> abc[4:9]
'y Pyt'
>>> abc[4:9][2]           # ej podret'azce môžeme d'alej indexovat'
'P'
>>> abc[4:9][2:4]
'Py'
>>> abc[4:9][::-1]
'tyP y'
```

Reťazce sú v pamäti nemenné

Typ `str`, t.j. znakové reťazce, je nemenný 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'
```

Porovnávanie jednoznakových reťazcov

Jednoznakové reťazce môžeme porovnávať relačnými operátormi ==, !=, <, <=, >, >=, napr.

```
>>> 'x' == 'x'
True
>>> 'm' != 'M'
True
>>> 'a' > 'm'
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
```

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
```

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.

** 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
```

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 znak >= '0' and 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 znak>='a' and znak<='z' or znak>='A' and znak<='Z'
```

- parametrom funkcie je reťazec s menom a priezviskom (oddelené sú práve jednou medzerou) - funkcia vráti reťazec priezvisko a meno (oddelené medzerou)

```
def meno(r):
    ix = 0
    while ix < len(r) and r[ix] != ' ':      # nájde 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
```

6.1 Reťazcové metódy

Je to špeciálny spôsob zápisu volania funkcie:

```
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, resp. `retazec.upper()` prevedie na veľké písmená

- `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

Formátovanie reťazca

Možnosti formátovania pomocou metódy `format()` sme už videli predtým. Teraz ukážeme niekoľko užitočných formátovacích prvkov. Volanie má tvar:

```
'formátovací ret'azec'.format(parametre)
```

- kde `'formátovací ret'azec'` môže obsahovať ľubovoľný text, ale pre metódu `format()` sú zaujímavé len dvojice znakov `'{}'`
- `parametre` pri volaní metódy `format()` sú ľubovoľné hodnoty (teda môžu byť ľubovoľných typov), týchto parametrov by malo byť presne rovnaký počet ako dvojíc `'{}'` (špeciálnymi prípadmi sa tu zaoberať nebudeme)
- 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 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
i == i + 1
math.sin(uhol)
```

Python pri nich nehlási ani žiadnu chybu.

KAPITOLA 7

Súbory

So súbormi súvisia znakové reťazce ale aj príkazy `input()` a `print()` na vstup a výstup:

- znakové reťazce sú postupnosti znakov (v Unicode kódovaní), 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 byť aj prázdna), 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.

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'`.

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).

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'`.

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 tretí riadok prázdny alebo obsahuje aj nejaké medzery:

```
prvy riadok
druhy riadok
    treti riadok

posledny riadok
```

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'
```

Prepíšme while-cyklus tak, aby pri prečítaní prázdneho reťazca skončil:

```
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()
```

Takýto for-cyklus bude fungovať aj vtedy, keď sme už zo súboru niečo čítali a už len potrebujeme spracovať zvyšok, 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='')
```

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.

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()`:

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')
```

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('cislo {} je prvoccislo\n'.format(ix))
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
```

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')
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.

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 uvedenej premennej. 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))+' ')
```

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, nepriradíme to do premennej, ale 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...). Správnejší zápis by mal byť:

```
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:

```
with open('subor3.txt', 'w') as f:
    print('first line\nsecond line\nend of file', file=f)
```

Nezabudnite, že ak súbor 'subor3.txt' niečo 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:

```
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())
```

Hoci teraz už vieme zapísať príkaz, ktorý na konzolu vypíše obsah nejakého textového súboru takto:

```
>>> print(open('readme.txt').read())
```

budeme to zapisovať:

```
>>> 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.

7.3 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
    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.

KAPITOLA 8

n-tice (tuple)

Aké typy premenných už poznáme:

- jednoduché typy:
  - `int` - celé čísla
  - `float` - desatinné čísla
  - `bool` - logické hodnoty
- postupnosti:
  - `str` - znakové reťazce sú postupnosti znakov
  - pomocou `open()` - textové súbory sú postupnosti riadkov
  - pomocou `range()` - vytvorí postupnosť celých čísel
- okrem toho sme sa stretli s typmi ako **funkcia** a **modul**, ale to teraz neriešime

Naučíme sa pracovať s novým typom, tzv. **n-tice** (v Pythone **tuple**), ktorý je tiež postupnosťou a je veľmi podobný znakovým reťazcom. Na rozdiel od reťazcov, sú to postupnosti hodnôt ľubovoľných typov. Uvidíme, že sa s nimi pracuje veľmi podobne ako so znakovými reťazcami, a preto si najprv pripomeňme, čo vieme o znakových reťazcoch:

- postupnosť znakov
- vieme zistiť dĺžku: funkcia `len()`
- operácia + vytvorí sa nový reťazec, ktorý na začiatku obsahuje kópiu prvého reťazca, a za tým nasleduje kópia druhého
- operácia $*$ n-krát zreťazí 1 reťazec - výsledný reťazec obsahuje n-krát zreťazenú kópiu pôvodného reťazca
- prvky postupnosti môžeme indexovať pomocou `[]` (od 0 až po počet prvkov-1) - výsledkom je jednoznakový reťazec
- vieme indexovať časť reťazca (vytvoriť podreťazec) `[:]` - tzv. rez (slice)
- operácia `in` vie zistiť, či sa nejaký podreťazec nachádza v inom reťazci
- funkcia `str()` z iného typu (napr. `int`, `float`, ...) vytvorí reťazec
- for-cyklom vieme postupne prechádzať všetky znaky reťazca - hovoríme, že reťazce sú **iterovateľné** (iterable)
- reťazce môžeme porovnávať na rovnosť ==, != aj usporiadanie <, <=, >, >=
  - lexikografické usporiadanie
  - postupne porovnáva znaky oboch reťazcov a keď nájdeme prvý rôzny znak, tak výsledok porovnania týchto dvoch znakov (porovnajú sa ich unicode) je výsledkom celkového porovnania

S reťazcami fungujú aj nejaké metódy

Metódy sú špeciálne funkcie, ktoré sú definované len pre konkrétny typ, v našom prípade pre reťazce. Zapisujeme ich ako `ret'azec.metóda(parametre)`. Existuje ich dosť veľa a my sme sa zoznámili len s niekoľkými:

- `ret'azec.find(podret'azec)` - vráti index prvého výskytu podreťazca
- `ret'azec.count(podret'azec)` - vráti počet výskytov podreťazca
- `ret'azec.lower()` - vráti kópiu malými písmenami
- `ret'azec.upper()` - vráti kópiu veľkými písmenami
- `ret'azec.replace(dva podret'azce)` - vráti kópiu s nahradenými všetkými výskytmi prvého podreťazca druhým
- `ret'azec.strip()` - vráti kópiu bez medzier na začiatku a na konci
- `ret'azec.format(hodnoty)` - vráti kópiu, v ktorej nahradí `{}` nejakými hodnotami

Uvidíme, že pre n-tice sú definované len dve metódy.

8.1 n-tice (tuple)

Podobne ako reťazce je to **štruktúrovaný typ**, t.j. je to typ, ktorý obsahuje hodnoty nejakých iných typov:

- postupnosť ľubovoľných hodnôt (nielen znakov ako pri reťazcoch)
- konštanty n-tíc uzatvárame do okrúhlych zátvoriek a 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'>
```

Vidíme, 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 oddelené čiarkami.

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:

```
>>> 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'>
```

Operácie s n-ticami

Operácie fungujú presne rovnako ako fungovali s reťazcami:

- operácia + zreťazí 2 n-tice - zreťazuje sa ich kópia
- operácia * n-krát zreťazí 1 n-ticu - zreťazuje sa jej kópia
- 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)
>>> 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
```

Funkcia tuple()

Vyrobí n-ticu z ľubovoľnej postupnosti (iterovateľného objektu, objekt, ktorý sa dá prechádzať for-cyklom), napr. so znakového reťazca, vygenerovanej postupnosti celých čísel pomocou `range()`, ale iterovateľný je aj otvorený textový súbor. Znakový reťazec táto funkcia 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')
```

Postupne sa riadky súboru stanú prvkami n-tice.

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` aj celých riadkov textového súboru. 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. 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
```

Pomocou for-cyklu vieme n-tice skladať podobne, ako sme to robili so znakovými reťazcami. 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,)`.

Indexovanie

n-tice indexujeme rovnako ako sme indexovali reťazce:

- 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. Keď sme do znakového reťazca chceli pridať jeden znak (alebo aj viac), museli sme to robiť rozoberaním a potom skladaním:

```
>>> retazec = 'Pyton'
>>> retazec = retazec[:3] + 'h' + retazec[3:]
>>> retazec
'Python'
```

Úplne rovnako to spravíme aj s n-ticami:

```
>>> 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'
...
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`.

Porovnávanie n-tíc

Porovnávanie n-tíc je rovnaké ako s reťazcami. Pripomeňme si, ako je to pri reťazcoch:

- postupne porovnáva i-te prvky oboch reťazcov, kým sú rovnaké; pri prvej nerovnosti je výsledkom porovnanie týchto dvoch hodnôt
- ak je pri prvej nezhode v prvom reťazci menšia hodnota ako v druhom (porovnávajú sa ich unicode), tak prvý reťazec je menší ako druhý (napr. v reťazcoch `'abcde'` a `'abced'` je prvá nezhoda na prvku s indexom 3 a keďže `ord('d')<ord('e')`, platí `'abcde'<'abced'`)
- ak je prvý reťazec kratší ako druhý a zodpovedajúce prvky sa zhodujú, tak prvý reťazec je menší ako druhý (napr. `'abc'<'abcd'`)

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 keď sú zhodného 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
```

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 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 premennej priraďujeme viac hodnôt oddelených čiarkou, Python to chápe ako priradenie n-tice, nasledovné priradenia sú rovnaké

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)
```

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'
```

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 = y,x
```

Aj v tomto príklade je na pravej strane priradenia (za =) n-tica: `(y,x)`.

n-tica ako návratová hodnota funkcie

V Pythone sa dosť využíva to, že návratovou hodnotou funkcie môže byť n-tica, t.j. vlastne naraz niekoľko návratových hodnôt. Napr. nasledovný príklad počíta celočíselné delenie a súčasne zvyšok po delení:

```
def zisti(a, b):
    return a//b, a%b
```

a použiť ju môžeme napr. takto:

```
>>> podiel,zvysok = zisti(153,33)
>>> print('podiel =',podiel,'zvysok =',zvysok)
podiel = 4 zvysok = 21
```

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
>>> sucet = 0
>>> for i in deli:
        sucet += i

>>> print(sucet)
60
```

Príklad ukazuje, že keď je výsledkom n-tica, môžeme ju ďalej spracovať napr. for-cyklom, resp. ďalej testovať.

Ďalšie funkcie a metódy

S n-ticami vedia pracovať nasledovné štandardné funkcie:

- `len(ntica)` - vráti počet prvkov n-tice
- `sum(ntica)` - vypočíta súčet prvok (všetky musia byť čísla)
- `min(ntica)` - zistí najmenší prvok (prvky sa musia dať navzájom porovnať, nemôžeme tu miešať rôzne typy)
- `max(ntica)` - zistí najväčší prvok (ako pri `min` ani tu sa nemôžu typy prvkov miešať)

Na rozdiel od 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 výskytu danej hodnoty, ak sa hodnota v n-tici nenachádza, metóda spôsobí spadnutie na chybe

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
```

resp. koľko-krát sa nejaká teplota vyskytla v našich meraniach:

```
>>> teploty.count(20)
1
>>> teploty.count(20.1)
0
```

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 `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 môže záležať 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 z 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.

Pre for-cyklus existuje jedno 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 cykly (akokeby viacnásobné priradenie). 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),
(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.

KAPITOLA 9

Polia (list)

Poznáme jednoduché dátové typy:

- `int`, `float`, `bool` - nedajú sa iterovať

A zložené typy (majú nejaké prvky), ktoré sú iterovateľné (dajú sa prechádzať for-cyklom):

- znakové reťazce - postupnosti znakov (`str`)
- textové súbory - postupnosti riadkov
- generátory číselných postupností (`range()`)
- n-tice - postupnosti rôznych hodnôt (`tuple`)

Znakový reťazec aj n-tica sú postupnosti hodnôt, ktoré ale nemôžeme v pamäti meniť (sú tzv. **nemeniteľné**, teda **immutable** typy):

- keď zostavíme reťazec alebo n-ticu, tak táto sa už v pamäti nikdy nezmení, t.j. ak máme na ňu referenciu (premennú), tak máme istotu, že ak sa nezmení referencia, nezmení sa ani hodnota
- aj všetky jednoduché typy sú nemeniteľné
- existujú **meniteľné** teda **mutable** typy, pre ktoré platí, že ak máme na nejakú meniteľnú hodnotu referenciu, tak v priebehu výpočtu sa táto hodnota môže meniť
- najzaujímavejším dôsledkom bude pre takýto meniteľný typ to, keď na jednu meniteľnú hodnotu budeme mať viac referencií (viac premenných bude na ňu odkazovať), potom zmenou hodnoty jednej premennej sa tým budú meniť aj všetky zvyšné premenná s touto referenciou

Základným meniteľným typom v Pythone je **pole** (teda pythonovský `list`). V porovnaní s poľami v iných programovacích jazykoch, pythonovské pole je v skutočnosti **dynamické pole**, keďže mu môžeme meniť jeho veľkosť (môžeme ho zväčšovať alebo zmenšovať). Niekedy sa pythonovskému poľu hovorí aj **zoznam**, ale tento pojem sa môže mýliť s dátovou štruktúrou **spájaný zoznam** (s ktorým sa zoznámime v letnom semestri). Niekedy budeme pole zobrazovať ako tabuľku hodnôt, pre ktorú v každom políčku tabuľky je jeden prvok.

S typom **pole** pracujeme úplne rovnako ako s n-ticami (`tuple`) z predchádzajúcej prednášky, ale ďalšie vlastnosti, v ktorých sa polia líšia od n-tíc tomuto typu dávajú obrovskú programátorskú silu. Takže najprv. čo je úplne rovnaké (resp. analogické) ako pre n-tice:

- pole je postupnosť hodnôt rôznych typov oddelených čiarkami, ktorá je uzavretá v `[]` hranatých zátvorkách
  - prvkami polí môžu byť ľubovoľné typy: aj n-tice, znakové reťazce ale aj iné polia
  - na rozdiel od n-tíc hranaté zátvorky písať musíme (Python si veľakrát okrúhle zátvorky pre n-tice domyslí)
  - jednoprvkové pole nemusí mať pri zápise navyše čiarku, ako sme to robili pre n-tice (napr. n-tica `(1,)` a pole `[1]`)

  - pole môžeme skonštruovať aj pomocou špeciálnej funkcie `list()` - táto z ľubovoľného iterovateľného objektu vyrobí pole (napr. z reťazca, z `range()`, z n-tice, zo súboru)

- pre polia fungujú operácie + zreťazenia, * násobenia, `in` zisťovanie príslušnosti prvku
- polia môžeme prechádzať for-cyklom (je to tiež iterovateľný typ)
- štandardné funkcie: `len()` pre počet prvkov, `sum()` pre súčet prvkov (funguje len pre číselné polia), `min()` a `max()` pre najmenší a najväčší prvok (len pre polia s porovnávateľnými prvkami)
- k prvkom pristupujeme pomocou indexovania a z poľa môžeme pomocou rezov (slice) vybrať ľubovoľné "podpole" (vyrobia sa pritom kópie prvkov)
- porovnávanie pomocou relácií ==, !=, <, <=, >, >= funguje na rovnakom princípe ako pre n-tice, t.j. lexikografické porovnávanie

Všetky tieto vlastnosti, ak použijeme v programoch, zabezpečia, že zatiaľ sú hodnoty typu pole "nemenené", t.j. všetky premenné sa správajú podobne ako n-tice.

Niekoľko príkladov práce s poľami:

```
>>> pole = [1,'a']*10
>>> pole
[1, 'a', 1, 'a', 1, 'a', 1, 'a', 1, 'a', 1, 'a', 1, 'a', 1, 'a', 1, 'a', 1,
↪'a']
>>> sum(pole[::2])
10
>>> nove = pole[3:10]
>>> for i in nove:
        print(i, end=', ')

a, 1, a, 1, a, 1, a,
>>> len(nove)
7
>>> list(range(1,10)) + list(range(10,20,2))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 16, 18]
>>> [2,3,5,7,11,13][2:][::-1]
[13, 11, 7, 5]
```

Posledný príklad ukazuje, že s hranatými zátvorkami si ešte užijeme veľa programátorských potešení.

Zadefinujme ešte niekoľko funkcií:

```
def sucet(pole):
    '''vráti súčet prvkov postupnosti

       predpokladajú sa číselné prvky
    '''
    vysl = 0
    for prvok in pole:
        vysl += prvok
    return vysl

def najmensi(pole):
    '''vráti najmenší prvok postupnosti

       predpokladá sa neprázdna postupnost',
       prvky sa musia dat' porovnávat' na reláciu <
    '''
    vysl = pole[0]
    for prvok in pole:
```

```
        if prvok < vysl:
            vysl = prvok
    return vysl

def usporiadane(pole):
    '''vráti True, ak je postupnosť usporiadaná vzostupne

       prvky sa musia dať porovnávať na reláciu <
    '''
    for ix in range(len(pole)-1):
        if pole[ix+1] < pole[ix]:    # porovnáva dva susedné prvky
            return False
    return True

import random

def nahodny(pole):
    '''vráti náhodne vybraný prvok postupnosti'''
    return pole[random.randrange(len(pole))]
```

- funkcia `sucet()` vráti súčet prvkov postupnosti (táto funkcia funguje pre n-ticu čísel aj pre pole čísel) - už vieme, že v Pythone rovnako funguje štandardná funkcia `sum()`
- funkcia `najmensi()` vráti najmenší prvok postupnosti (pre parameter typu reťazec, n-tica, pole) - v Pythone fungujú štandardné funkcie `min()` a `max()`
- funkcia `usporiadane()` vráti `True`, ak sú prvky postupnosti (reťazec, n-tica, pole) usporiadané vzostupne, skúste napr. `usporiadane('amor')`
- funkcia `nahodny()` vygeneruje náhodné číslo z intervalu `<0,len(pole)-1>` ako index do zadanej postupnosti (parameter `pole` môže byť reťazec, n-tica, pole)
- v module `random` je definovaná funkcia, ktorá robí to isté: `random.choice()`

V každom programovacom jazyku existuje konštrukcia, pomocou ktorej vyhradíme (možno aj inicializujeme) pole nejakej počiatočnej veľkosti. V Pythone najčastejšie využívame násobenie jednoprvkového poľa nejakým celým číslom, napr.

```
>>> pole1 = 1000 * [0]
>>> pole2 = ['a'] * 2000
>>> pole3 = [None] * 30000
```

Vyhradili sme 3 rôzne polia:

- `pole1` 1000 nulových hodnôt
- `pole2` 2000 jednoznakových reťazcov `'a'`
- `pole3` 30000 špeciálnych hodnôt `None`, ktoré označujú prázdnu hodnotu (napr. výsledok funkcií, ktoré nevracajú žiadnu hodnotu)

Môžeme si na to vytvoriť funkciu:

```
def vyrob_pole(dlzka, hodnota=0):
    return [hodnota] * dlzka
```

```
>>> pole1 = vyrob_pole(1000)
>>> pole2 = vyrob_pole(2000, 'a')
>>> pole3 = vyrob_pole(30000, None)
```

9.1 Pole je meniteľný typ

Už vieme, že napr. znakový reťazec (nemeniteľný typ) nám Python nedovolí zmeniť takýmto priradením:

```
>>> ret = 'Python'
>>> ret[3] = 'X'
...
TypeError: 'str' object does not support item assignment
```

Pozrime, ako je to s poľom. Vyrobme 6-prvkové pole, napr.

```
>>> pole = list('Python')
>>> pole
['P', 'y', 't', 'h', 'o', 'n']
```

Takéto pole môžeme meniť zmenou ľubovoľného prvku, napr.

```
>>> pole[3] = 'X'
>>> pole
['P', 'y', 't', 'X', 'o', 'n']
```

Samozrejme, že to funguje nielen pre pole znakov, ale pre pole hodnôt ľubovoľných typov.

Všetko, čo sme doteraz vedeli robiť s obyčajnými premennými, teraz vieme robiť aj s prvkami poľa, napr. môžeme navzájom vymeniť hodnoty dvoch rôznych prvkov poľa:

```
>>> cisla = list(range(0, 100, 10))
>>> cisla
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> cisla[3], cisla[6] = cisla[6], cisla[3]
>>> cisla
[0, 10, 20, 60, 40, 50, 30, 70, 80, 90]
```

Rezy (slice)

Pole môžeme meniť aj zmenou rezu (teda nejakého intervalu indexov):

```
>>> prvo = [2, 3, 5, 7, 11, 13, 17, 19, 23]
>>> prvo[3:5]
[7, 11]
>>> prvo[3:5] = []
>>> prvo
[2, 3, 5, 13, 17, 19, 23]
>>> prvo[4]
17
>>> prvo[4:4] = [100, 101, 102]
>>> prvo
[2, 3, 5, 13, 100, 101, 102, 17, 19, 23]
```

Na tomto príklade si všimnite:

- keď priraďujeme nejakú hodnotu do rezu, táto hodnota **musí** byť opäť **pole** (alebo nejaká iterovateľná hodnota) a toto pole nahradí všetky prvky v reze
- ak má rez nulovú dĺžku (napr. `pole[0:0]`, `pole[4:4]`, `pole[len(pole):len(pole)]`), vkladané pole sa vloží pred štartový index rezu

Videli sme, že priradením prázdneho poľa do rezu sa všetky prvky v reze vyhodia, to isté sa dá dosiahnuť aj príkazom `del`, ktorý vyhodí nielen niektorý konkrétny prvok, ale aj ľubovoľnú časť poľa, napr.

```
>>> pole = [2, 3, 5, 7, 11, 13, 17, 19, 23]
>>> del pole[3:5]
>>> pole
[2, 3, 5, 13, 17, 19, 23]
>>> del pole[4]
>>> pole
[2, 3, 5, 13, 19, 23]
>>>
```

Dve mená na to isté pole

Ak sme doteraz v Pythone do jednej premennej priradili obsah inej (napr. číslo alebo reťazec), nemuseli sme sa obávať, žiadnych "vedľajších účinkov": obsah takejto premennej sa zmenil, len ak sme do nej priradili inú hodnotu.

S poľami je to iné: ak je obsahom premennej pole (t.j. referencia na pole v pamäti) a túto hodnotu priradíme aj do inej premennej, obe premenné teraz referencujú na tú istú hodnotu. Lenže túto hodnotu môžeme meniť priamo v pamäti a teda sa zmenia hodnoty oboch premenných. Napr.

```
>>> a = [2, 3, 5, 7, 11, 13, 17]
>>> b = a
>>> b[3] = 'kuk'
>>> a
[2, 3, 5, 'kuk', 11, 13, 17]
>>> del a[4:]                  # vyhodí všetky prvky od 4-tého do konca
>>> len(b)                     # tým sa zmení dĺžka pol'a
4
```

teda, kým do jednej z týchto premenných nepriradíme inú hodnotu, budú stále odkazovať (referencovať) na to isté pole, napr.

```
>>> b = b + [99]               # do b sme priradili už inú hodnotu
>>> a
[2, 3, 5, 'kuk']
>>> b
[2, 3, 5, 'kuk', 99]
```

Pri práci s poľami si budeme musieť dávať veľký pozor na to, ktoré operácie a funkcie kedy použijeme. Priradenie do premennej v každom prípade mení referenciu (niekedy aj na tu istú, aká tam už bola). Ak máme dve premenné, ktoré referencujú na to isté pole, tak kým s nimi budeme robiť tieto operácie, tak sa referencie stále udržujú:

- priradenie do prvku poľa (indexovanie)
- priradenie do rezu
- vyhadzovanie prvku alebo rezu z poľa (pomocou príkazu `del`)
- volanie metód a štandardných funkcií

Rovnaké referencie môžeme mať nielen na celé polia, ale aj na prvky poľa, ak sú prvkami opäť polia. Zadefinujme pole `c` tak, že jeho prvkami budú iné dve polia `a` a `b`:

```
>>> a = [1]
>>> b = [2]
>>> c = [a, b, a, b, a, b]
```

Pole `c` obsahuje 6 referencií na iné polia: 3 z nich sú na jednoprvkové pole, na ktoré odkazuje `a` a tri sú na pole `b`. Pole `c` vypíšeme:

```
>>> c
[[1], [2], [1], [2], [1], [2]]
```

Ak si uvedomíme, že hodnota `[1]` je tu obsahom premennej `a`, tak zmenou hodnoty `a` (ale bez zmeny referencie!) zmeníme aj obsah poľa `c`:

```
>>> a[0] = 'kuk'
>>> c
[['kuk'], [2], ['kuk'], [2], ['kuk'], [2]]
```

Pomocou rezu môžeme meniť počet prvkov poľa. Napr. pridáme ďalší prvok do poľa `b`:

```
>>> b[1:1] = [2014]
>>> b
[2, 2014]
>>> c
[['kuk'], [2, 2014], ['kuk'], [2, 2014], ['kuk'], [2, 2014]]
```

Odteraz si bude treba dávať pozor aj na funkcie, ktoré pracujú s parametrom typu pole. Ak takáto funkcia zmení obsah parametra (typu pole), mali by sme o tom dopredu vedieť. Väčšinou by sme sa mali v dokumentačnom reťazci dozvedieť, čo funkcia robí s **mutable** (meniteľnými) parametrami funkcie, t.j. či funkcia robí nejaký **vedľajší účinok**.

Zapíšme funkciu, ktorá priamo modifikujú parameter pole, napr.

```
def zmen(pole):
    '''pridá nový prvok (ret'azec 'begin') na úplný začiatok pola, t.j. ešte␣
↪pred nultý

       funkcia nič nevracia len modifikuje parameter pole
    '''
    pole[0:0] = ['begin']
```

Otestujeme:

```
>>> pole = ['jeden', [2, 'dva'], 3]
>>> druhy = pole[1]
>>> zmen(pole)
>>> zmen(druhy)                          # mohli sme zapísat' aj zmen(pole[1])
>>> pole
['begin', 'jeden', ['begin', 2, 'dva'], 3]
```

V tomto príklade vidíme, že premenná `druhy` má rovnakú hodnotu ako tretí prvok poľa `pole[2]`, t.j. zrejme platí:

```
>>> druhy == pole[2]
True
```

Ale to, že nejaké premenné majú tú istú hodnotu ešte nemusí znamenať, že obsahujú tú istú referenciu. V niektorých situáciách môže byť užitočné vedieť, či sú dve rovnaké hodnoty tou istou referenciou. Vtedy využijeme novú operáciu `is`, ktorá vráti `True` len vtedy, keď je to identická referencia a nestačí, že je to tá istá hodnota. Napr.

```
>>> druhy == pole[2]
True
>>> druhy is pole[2]
True
>>> iny = ['begin', 2, 'dva']
>>> druhy == iny
True
>>> druhy is iny
False
```

```
>>> iny is pole[2]
False
```

Kým si zvyknete na takúto prácu s pythonovskými typmi, treba si referencie na hodnoty kresliť, alebo využiť http://pythontutor.com/, v ktorom sa treba prepnúť na 3.3 verziu Pythonu.

9.2 Metódy pre polia

Podobne ako pri reťazcoch a n-ticiach aj pri poliach sú preddefinované niektoré metódy.

- syntax volania metódy je `pole.metóda(parametre)`
- tieto metódy sú funkcie, niektoré z nich vracajú hodnotu, iné nie
- niektoré metódy modifikujú hodnotu samotného poľa - teda majú **vedľajšie účinky** (často vtedy nevracajú žiadnu hodnotu)

Niektoré metódy pre polia (list)

- `pole.count(hodnota)` vráti počet výskytov hodnoty v poli (alebo v n-tici)
- `pole.index(hodnota)` vráti index prvého výskytu hodnoty v poli (alebo v n-tici)
- `pole.append(hodnota)` pridá novú hodnotu na koniec pôvodného poľa
- `pole.insert(index,hodnota)` vloží hodnotu do pôvodného poľa pred zadaný index
- `pole.pop()` odstráni posledný prvok pôvodného poľa a vráti tento prvok ako hodnotu
- `pole.pop(0)` odstráni prvý prvok pôvodného poľa a vráti tento prvok ako hodnotu
- `pole.remove(hodnota)` vyhodí z pôvodného poľa prvý výskyt hodnoty
- `pole.sort()` vzostupne utriedi pôvodné pole (priamo v pamäti), prvky poľa sa musia dať porovnávať

metóda `append()`

Je najčastejšie používanou metódou a využíva sa najmä pri konštruovaní poľa. Metóda pridá na koniec pôvodného poľa nový prvok. Jej tvar je:

```
pole.append(prvok)
```

Funguje ako **vedľajší účinok**, t.j. modifikuje hodnotu v pamäti a nič nevracia.

Napr.

```
pole = []                  # zatiaľ prázdne pole
for i in range(100):
    if i % 7 < 5:
        pole.append(i)
```

V tomto príklade postupne konštruujeme prvky poľa tak, že pridávame tie hodnoty

Ďalšie štyri funkcie robia to isté, ale zakaždým trochu inak: vytvárajú pole, ktoré je kópiou už existujúceho, pritom každý prvok násobia 2. Vidíme, že nemodifikujú pôvodné pole, ale konštruujú nové, teda ***nemajú vedľajší účinok**:

```
def kopia2(pole):
    vysl = []                  # zatiaľ prázdne pole
    for prvok in pole:
```

```
        vysl.append(prvok * 2)
    return vysl

def kopia2(pole):
    vysl = []                   # zatiaľ prázdne pole
    for prvok in pole:
        vysl += prvok * 2
    return vysl

def kopia2(pole):
    vysl = [0]*len(pole)        # rovnako veľké pole núl
    for ix in range(len(pole)):
        vysl[ix] = pole[ix] * 2
    return vysl

def kopia2(pole):
    vysl = pole[:]              # presná kópia poľa
    for ix in range(len(vysl)):
        vysl[ix] *= 2
    return vysl
```

Prvá verzia tejto funkcie ilustruje použitie metódy `append` a pre takýto typ úloh sa používa najčastejšie.

Častou začiatočníckou chybou pri práci s metódami býva to, že výsledok volania tejto funkcie priradíme do samotného poľa, napr.

```
>>> a = [1, 2, 3, 4]
>>> a = a.append(5)
>>> print(a)
None
```

Metóda `append()` modifikuje samotné pole a nič nevracia, teda vlastne vracia hodnotu `None`. V príklade hoci do poľa `a` na koniec pridáme nový prvok s hodnotou 5, hneď aj zmeníme hodnotu tejto premennej, a priradíme do nej výsledok `append()`, t.j. `None`.

metóda `pop()`

Vyhadzuje z poľa buď posledný prvok (ak je volaná bez parametrov), alebo prvok na zadanom indexe:

```
pole.pop()

pole.pop(index)
```

Metóda vždy vráti vyhadzovaný prvok ako výsledok volania funkcie. Takže táto funkcia nielen modifikuje pole (má **vedľajší účinok**), ale aj vracia hodnotu vyhadzovaného prvku poľa. Indexom je celé číslo od `0` do `len(pole)-1` alebo od `-1` do `-len(pole)`.

Napr. môžeme ju použiť takto:

```
pole = [5, 10, 15, 20, 25]
while pole:                 # kým nie je pole prázdne
    print(pole.pop(), end=' ')
print()
```

vypíše:

```
25 20 15 10 5
```

Alebo skoro to isté, ale budeme vyhadzovať zo začiatku poľa:

```
pole = [5, 10, 15, 20, 25]
while pole:                 # kým pole nie je prázdne
    print(pole.pop(0), end=' ')
print()
```

vypíše:

```
5 10 15 20 25
```

metóda `insert()`

Vkladá do poľa jednu hodnotu na pozíciu pred zadaný index:

```
pole.insert(index, prvok)
```

Ak je index == 0, vloží na úplný začiatok, ak je index == dĺžka poľa (ale môže byť aj väčší), zadanú hodnotu pridá na koniec poľa (teda urobí vlastne `append()`). Rovnako ako `append()` aj táto metóda "iba" modifikuje pole a nič nevracia (teda vracia hodnotu `None`).

Napr.

```
>>> pole = [1, 2, 3]
>>> pole.insert(0, -99)
>>> pole
[-99, 1, 2, 3]
>>> pole.insert(2, 98)
>>> pole
[-99, 1, 98, 2, 3]
>>> pole.insert(5, 97)
>>> pole
[-99, 1, 98, 2, 3, 97]
>>> pole.insert(-1, 'koniec')
>>> pole
[-99, 1, 98, 2, 3, 'koniec', 97]
```

Všimnite si, že záporný index -1 označuje posledný prvok poľa, ale keďže `insert()` vkladá pred zadaný prvok, tak s -1 vkladá pred posledný.

Ak by sme chceli túto metódu využiť na to, aby sme pred každý jeho prvok vložili reťazec 'a', tak to nemôžeme zapísať takto:

```
>>> pole = [1, 2, 3]
>>> for i in range(len(pole)):
        pole.insert(i, 'a')

>>> pole
['a', 'a', 'a', 1, 2, 3]
```

buď poopravíme index miesta, kam treba vložiť 'a':

```
>>> pole = [1, 2, 3]
>>> for i in range(len(pole)):
```

```
        pole.insert(2*i, 'a')

>>> pole
['a', 1, 'a', 2, 'a', 3]
```

alebo vkladať môžeme nie od prvého po posledný prvok, ale naopak od konca:

```
>>> pole = [1, 2, 3]
>>> for i in range(len(pole)-1,-1,-1):
        pole.insert(i, 'a')

>>> pole
['a', 1, 'a', 2, 'a', 3]
```

metóda `remove()`

Táto metóda najprv nájde prvý (najľavejší) výskyt danej hodnoty v poli a potom tento prvok z poľa vyhodí. Jej tvar:

```
pole.remove(hodnota)
```

Ak sa ale v poli zadaná hodnota nenachádza, program spadne s chybovou správou.

Napr.

```
>>> pole = ['a', 'b', 'c']
>>> pole.remove('b')
>>> pole
['a', 'c']
>>> pole.remove('b')
...
ValueError: list.remove(x): x not in list
```

Ak by sme chceli vyhodiť všetky výskyty nejakej hodnoty v poli, môžeme to urobiť napr. takýmto cyklom:

```
>>> pole = list('programovanie pre radost')
>>> pole
['p', 'r', 'o', 'g', 'r', 'a', 'm', 'o', 'v', 'a', 'n', 'i', 'e', ' ',
'p', 'r', 'e', ' ', 'r', 'a', 'd', 'o', 's', 't']
>>> while 'r' in pole:
        pole.remove('r')

>>> pole
['p', 'o', 'g', 'a', 'm', 'o', 'v', 'a', 'n', 'i', 'e', ' ', 'p', 'e',
' ', 'a', 'd', 'o', 's', 't']
```

metóda `sort()`

Metódu `sort()` sa zatiaľ naučíme používať bez parametrov, neskôr budeme využívať aj parametre. Táto metóda preusporiada prvky poľa tak, že budú nasledovať usporiadané vzostupne (od najmenšieho po najväčší, resp. od prvého podľa abecedy po posledného). Tvar volania:

```
pole.sort()
```

Metóda nič nevracia (iba hodnotu `None`).

Napr.

```
>>> a = [15, 22, 7, 17, 4, 29, 1, 7, 11, 10]
>>> a.sort()
>>> a
[1, 4, 7, 7, 10, 11, 15, 17, 22, 29]
>>> b = ['emil', 'fero', 'hana', 'cyril', 'dasa', 'adam', 'gita', 'beta']
>>> b.sort()
>>> b
['adam', 'beta', 'cyril', 'dasa', 'emil', 'fero', 'gita', 'hana']
```

Pre prvky poľa musí platiť rovnaké pravidlo ako pre štandardné funkcie `min()` a `max()`: prvky sa musia dať navzájom porovnávať a teda nie je dovolené miešanie typov, napr.

```
>>> c = [15, 'adam', 17]
>>> c.sort()
...
TypeError: unorderable types: str() < int()
```

Metóda `sort()` funguje iba pre polia. Existuje ešte variant tejto funkcie:

funkcia `sorted()`

Je to štandardná funkcia (nie metóda), ktorá dostáva ako parameter ľubovoľný iterovateľný objekt (napr. pole, n-ticu, reťazec, súbor, ale aj `range()`) a vytvorí z neho **pole** utriedených hodnôt, ktoré sú z daného iterovateľného objektu. Jej tvar:

```
premenná = sorted(postupnost')
```

Funkcia vracia hodnotu a nemodifikuje (nemá vedľajší účinok). Teda funkcia vždy skonštruuje nové pole a to je výsledkom funkcie. Aj táto funkcia môže mať ďalšie parametre rovnako ako metóda `sort()`, ale to ukážeme neskôr.

Napr.

```
>>> sorted('python')
['h', 'n', 'o', 'p', 't', 'y']
>>> sorted((5, 1, 4, 2, 3))
[1, 2, 3, 4, 5]
>>> sorted([(10,30), (20,10), (10,20)])
[(10, 20), (10, 30), (20, 10)]
```

Zhrňme: vkladanie do poľa

Videli sme viac rôznych spôsobov, ako môžeme pridať jednu hodnotu do poľa. Vkladanie nejakej `hodnoty` pred prvok s indexom `i`:

- pomocou rezu:

```
pole[i:i] = [hodnota]
```

- pomocou metódy `insert()`:

```
pole.insert(i, hodnota)
```

- ak `i` = `len(pole)`, pridávame na koniec, môžeme použiť metódu `append()`:

```
pole.append(hodnota)
```

Vo vašich programoch použijete ten zápis, ktorý sa vám bude najlepšie hodiť, ale zápis s rezom `pole[i:i]` je najmenej čitateľný a používa sa veľmi zriedkavo.

Zrejme funguje aj:

```
pole = pole[:i] + [hodnota] + [i:]
```

resp.

```
pole += [hodnota]
```

Tieto dve priradenia nemodifikujú pôvodné pole, ale vytvárajú nové s pridanou hodnotou.

Zhrňme: vyhadzovanie z poľa

Aj vyhadzovanie prvku z poľa môžeme robiť viacerými spôsobmi. Ak vyhadzujeme prvok na indexe `i`, môžeme zapísať:

- pomocou rezu:

```
pole[i:i+1] = []
```

- pomocou príkazu `del`:

```
del pole[i]
```

- pomocou metódy `pop()`, ktorá nám aj vráti vyhadzovanú hodnotu:

```
hodnota = pole.pop(i)
```

- veľmi neefektívne pomocou metódy `remove()`, ktorá ako parameter očakáva nie index ale vyhadzovanú hodnotu:

```
pole.remove(pole[i])
```

  tento spôsob je veľmi neefektívny (zbytočne sa hľadá prvok v poli) a okrem toho niekedy môže vyhodiť nie `i`-ty prvok, ale prvok s rovnakou hodnotou, ktorý sa v poli nachádza skôr ako index `i`.

Zhrňme: vyhodenie všetkých prvkov z poľa

Najjednoduchší spôsob:

```
pole = []
```

môžeme použiť len vtedy, keď nepotrebujeme uchovať referenciu na pole - toto priradenie nahradí momentálnu referenciu na pole referenciou na úplne nové pole; ak to použijeme vo vnútri funkcie, stratí sa tým referencia na pôvodné pole.

Ďalšie spôsoby uchovávajú referenciu na pole:

- pomocou cyklu postupne vyhodíme všetky prvky:

```
while pole:
    pole.pop()
```

  toto je zbytočne veľmi neefektívne riešenie

- priradením do rezu:

```
pole[:] = []
```

ťažšie čitateľné a menej pochopiteľné riešenie

- metódou `clear()`:

```
pole.clear()
```

9.3 Polia a reťazce

Už sme videli niekoľko prípadov, keď polia spolupracovali s reťazcami, napr.

```
>>> pole = [2, 4, 6, 8, 10]
>>> pole[2:4] = 'ahoj'
>>> pole
[2, 4, 'a', 'h', 'o', 'j', 10]
>>> pole2 = list('python')
>>> pole2
['p', 'y', 't', 'h', 'o', 'n']
```

Všade tam, kde sa očakáva zadanie postupnosti (iterovateľný objekt) a objaví sa reťazec, tak tento sa automaticky prerobí na pole znakov (prvky reťazca sa iterujú - prechádzajú cyklom, a postupne sa z nich stávajú prvky poľa).

Existujú dve metódy, ktoré uľahčujú prevod medzi poľami reťazcov a jedným reťazcom:

metóda `split()`

Keďže je to reťazcová metóda, má tvar:

```
ret'azec.split()
```

Metóda rozbije jeden reťazec na samostatné reťazce a uloží ich do poľa (teda vracia pole reťazcov).

metóda `join()`

Opäť je to reťazcová metóda. Má tvar:

```
oddel'ovač.join(postupnost'_ret'azcov)
```

Metóda zlepí všetky reťazce z danej postupnosti (môže to byť n-tica alebo pole) reťazcov do jedného, pričom ich navzájom oddelí uvedeným `oddel'ovačom`, t.j. nejakým zadaným reťazcom.

Najlepšie to pochopíme na niekoľkých príkladoch. Metóda `split()` sa často využíva pri rozdelení prečítaného reťazca zo vstupu (`input`) na viac častí, napr.

```
>>> ret = input('zadaj 2 čísla: ')
zadaj 2 čísla: 15 999
>>> 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 `retazec.join(pole)` zlepí všetky reťazce v danom poli, pričom medzi ne vkladá zadaný oddeľovač (reťazec, ktorý je zadaný pri volaní metódy pred bodkou), napr.

```
>>> 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'
```

Preštudujte:

```
>>> veta = 'kto druhemu jamu kope'
>>> ' '.join(veta[::-1].split()[::-1])
'otk umehurd umaj epok'
>>> ' '.join(sorted(input('?').split()))
?anicka dusicka kde si bola ked si si cizmicky zarosila
'anicka bola cizmicky dusicka kde ked si si si zarosila'
```

Polia a grafika

Väčšina grafických príkazov, napr. `create_line`, `create_polygon`, ... akceptujú ako parametre nielen čísla, ale aj polia (ale aj n-tice) čísel, resp. polia dvojíc 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([random.randrange(350), random.randrange(250)])
canvas.create_line(krivka)
```

Ak by sme chceli využiť grafickú funkciu `coords()`, ktorá modifikuje súradnice nakreslenej krivky, nemôžeme jej poslať pole súradníc, ale vyžaduje pole 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):
    krivka.append(random.randrange(350))
    krivka.append(random.randrange(250))
    canvas.coords(poly, krivka)
    canvas.update()
    canvas.after(300)
```

Udalosti v grafickej ploche

Naučíme sa v našich programoch využívať tzv. **udalosti**, ktoré vznikajú v bežiacej grafickej aplikácii buď aktivitami používateľa (klikaie 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 umôžňujú postupne zobrazovať vyt áranú 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 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 format je:

```
canvas.bind(meno_udalosti, funkcia)
```

kde `meno_udalosti` je znakový reťazec s popisom udalosti (napr. pre kliknutie tlačidlom myši) a `funkcia` sa spustí pri vzniku tejto udalosti. Táto funkcia, musí byť definovaná s práve jedným parametrom, v ktorom nám systém prezradí detaily vzniknutej udalosti.

Ukážme 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 tlačidlo) - reťazec `'<B1-Motion>'`
- **pustenie myši** - reťazec `'<ButtonRelease-1>'`

10.1 Klikanie myšou

Kliknutie myšou do grafickej plochy vyvolá udalosť s menom `'<Button-1>'`. Ukážme ako vyzerá samotné zviazanie funkcie:

```
import tkinter, random

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def nahodny_kruh(parameter):
    x = random.randrange(600)
    y = random.randrange(450)
    canvas.create_oval(x-10, y-10, x+10, y+10, fill='red')

canvas.bind('<Button-1>', nahodny_kruh)
```

Na konci programu je príkaz `bind()`, ktorý má druhý parameter **referenciu na funkciu**, t.j. funkcia bez zátvoriek. Táto funkcia zrejme už musí byť definovaná skôr, ako sa použije ako parameter `bind()`. Funkcia, ktorá obsluhuje udalosť, musí mať jeden parameter, ktorý sa v tejto ukážke ešte nevyužíva. V tomto parametri (najčastejšie ho budeme označovať `event` alebo skrátene `e`) nám Python oznámi detaily udalosti, napr. súradnice, kde sme klikli (sú to atribúty parametra `event.x` a `event.y`).

V ďalšom príklade ukážeme, ako využijeme súradnice kliknutého bodu v ploche:

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def klik(event):
    x = event.x
    y = event.y
    canvas.create_oval(x-10, y-10, x+10, y+10, fill='red')

canvas.bind('<Button-1>', klik)
```

Opäť sa pri kliknutí nakreslí červený kruh, len sa pritom využijú suradnice kliknutého miesta: stred kruhu je kliknuté miesto.

Akcia, ktorá sa vykoná pri kliknutí môže byť veľmi jednoduchá, napr. spájanie kliknutého bodu so stredom grafickej plochy:

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def klik(event):
    canvas.create_line(300, 225, event.x, event.y)

canvas.bind('<Button-1>', klik)
```

Ale môžu sa nakresliť aj komplexnejšie kresby, napr. 10 sústredných farebných kruhov:

```
import tkinter, random

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def klik(event):
    x = event.x
    y = event.y
    for r in range(50, 0, -5):
        farba = '#{:06x}'.format(random.randrange(256**3))
        canvas.create_oval(x-r, y-r, x+r, y+r, fill=farba)

canvas.bind('<Button-1>', klik)
```

Vráťme sa k príkladu, v ktorom sme kreslili malé krúžky:

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
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)
```

Chceme do tohto programu pridať takéto správanie: až od druhého kliknutia sa tieto kliknuté body (červené krúžky) budú postupne spájať úsečkami. Pridáme dve globálne premenné `xx` a `yy`, v ktorých si budeme pamätať predchádzajúci kliknutý bod. Pred prvým kliknutím sme do `xx` priradili `-1`, čo bude označovať, že predchádzúci vrchol ešte nie je:

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

xx, yy = 0, 0

def klik(event):
```

```
    x, y = event.x, event.y
    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)
```

Žiaľ to nefunguje: po spustení a kliknutí sa dozvieme:

```
File ..., line 13, in klik
    if xx >= 0:
UnboundLocalError: local variable 'xx' referenced before assignment
```

Problémom su tu **globálne premenné**. Používať globálne premenné vo vnútri funkcii môžeme, len keď ich chceme vo vnútri funkcie meniť (použíli sme priradenie `xx, yy = x, y`), Python pochopí, že obe tieto premenné sú lokálne (priradenie vo funkcii predsa vytvára novú premennú v **lokálnom mennom priestore**) a v príkaze `if` sme použili lokálnu premennú `x` ešte skôr ako sme do nej niečo priradili. Takže s globálnymi premennými vo funkcii sa bude musieť pracovať nejako inak. Zrejme, keď nepotrebujeme do takejto premennej priradzovať, iba ju používať, problémy nie sú.

Tento problém nám pomôže vyriešiť nový príkaz `global`:

príkaz `global`

príkaz má tver:

```
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) a Python ju nevytvorí v lokálnom mennom priestore, ale v globálnom.

Po doplnení tohto príkazu do predchádzajúceho príkladu všetko funguje tak, ako má:

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

xx, yy = 0, 0

def klik(event):
    global xx, yy
    x, y = event.x, event.y
    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)
```

Varovanie: Príkaz `global` umožňuje modifikovať globálne premenné vo funkciách, teda vlastne má **vedľajší účinok**. Toto je ale veľmi nesprávny spôsob programovania (*bad programming practice*), väčšinou svedčí o prog-

ramátorovi začiatočníkovi, amatérovi. My to nebudeme používať skoro vôbec, možno výnimočne pri ladení.

Správne sa takéto problémy riešia definovaním vlastných tried a použitím atribútov tried. Toto sa naučíme až neskôr. V tejto prednáške sa príkaz `global` ešte objaví, ale myslite na to, že je to len dočasné, kým sa nenaučíme slušne objektovo programovať.

10.2 Ťahanie myšou

Obsluha udalosti ťahanie myšou (pohyb myši si zatlačeným tlačidlom) je veľmi podobné klikaniu. Udalosť ma meno `'<B1-Motion>'`. Pozrime, čo sa zmení, keď kliknutie `'<Button-1>'` nahradíme ťahaním `'<B1-Motion>'`:

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def klik(event):
    canvas.create_line(300, 225, event.x, event.y)

canvas.bind('<B1-Motion>', klik)
```

Funguje to veľmi dobre: priťahaní sa kreslia lúče zo stredu plochy k pozícii myši; pri pomalom ťahaní sú čiary kreslené veľmi nahusto. Často v našich programoch budeme spracovávať obe udalosti: kliknutie aj ťahanie. Niekedy je to tá istá funkcia, inokedy sú rôzne, napr.

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def klik(event):
    canvas.create_line(300, 225, event.x, event.y, fill='red')

def tahanie(event):
    canvas.create_line(300, 225, event.x, event.y)

canvas.bind('<Button-1>', klik)
canvas.bind('<B1-Motion>', tahanie)
```

Pri kliknutí (ešte bez ťahania) sa nakreslí červená úsečka, pri ťahaní sa kreslia čierne.

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 pole (`list`).

Budeme ťahať (ťahaním myši) jednu dlhú lomenú čiaru, pričom si súradnice prijaté z udalosti budeme ukladať do poľa čísel.

```
import tkinter

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

pole = []
ciara = canvas.create_line(0, 0, 0, 0)
```

```
def klik(event):
    pole[:] = [event.x, event.y]

def tahanie(event):
    pole.append(event.x)
    pole.append(event.y)
    canvas.coords(ciara, pole)

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()`
- `pole` - pole súradníc je meniteľný objekt, teda môžeme meniť obsah poľa bez toho, aby sme do premennej `pole` priraďovali (priraďujeme buď do rezu alebo voláme metódu `append()`)
  - modifikovanie poľa vo funkcii, pričom pole 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(width=600, height=450, bg='white')
canvas.pack()

pole = []

def klik(event):
    global ciara
    pole[:] = [event.x, event.y]
    ciara = canvas.create_line(0, 0, 0, 0)

def tahanie(event):
    global ciara
    pole.append(event.x)
    pole.append(event.y)
    canvas.coords(ciara, pole)

canvas.bind('<Button-1>', klik)
canvas.bind('<B1-Motion>', tahanie)
```

10.3 Udalosti od klávesnice

Každé zatlačenie nejakého klávesu na klávesnici môže tiež 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(e):
    print(e.keysym)

canvas.bind_all('<Key>', test)
```

Všimnite si, že sme museli zapísať `bind_all()` namiesto `bind()`.

Každý jeden kláves môže vyvolať samostatnú udalosť. Ako meno udalosti treba uviesť meno klávesu 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)
```

Najčastejšie to využijeme tak, ako v tomto príklade:

```
import tkinter

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

def kresli():
    global pole
    pole += [x, y]
    canvas.coords(ciara, pole)

def udalost_vlavo(event):
    global x
    x -= 10
    kresli()

def udalost_vpravo(event):
    global x
    x += 10
    kresli()

def udalost_hore(event):
    global y
    y -= 10
    kresli()

def udalost_dolu(event):
    global y
```

```
    y += 10
    kresli()

x, y = 200, 200
pole = [x, y]
ciara = canvas.create_line(0,0,0,0)         # zatiaľ 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)
```

10.4 Časovač

Pripomeňme si, ako sme doteraz v grafickej ploche kreslili krúžky na náhodné pozície s nejakým časovým pozdržaním (napr. 100 ms):

```
import tkinter, random

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def kresli():
    while True:
        x = random.randrange(600)
        y = random.randrange(450)
        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 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 grafickej plochy, teda `canvas`, 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 (skutočným parametrom 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)
```

Funkcia teda 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.

Predchádzajúci program teraz prepíšeme s použitím časovača:

```
import tkinter, random

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def kresli():
    x = random.randrange(600)
    y = random.randrange(450)
    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 pokračuje ďalším príkazom, t.j. sa vypíše `print('hotovo')`, program končí a v "shelli" môžeme zadávať ďalšie príkazy. Priotom ale stále beží náš rozbehnutý **časovač**. Počas behu časovača môže program vykonávať ďalšie akcie, napr. spustiť aj ďalší časovač. Zapíšme:

```
import tkinter, random

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def kresli():
    x = random.randrange(600)
    y = random.randrange(450)
    canvas.create_oval(x-10, y-10, x+10, y+10, fill='red')
    canvas.after(100, kresli)

def kresli1():
    x = random.randrange(600)
    y = random.randrange(450)
    canvas.create_rectangle(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

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, random

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def kresli():
    x = random.randrange(600)
    y = random.randrange(450)
    canvas.create_oval(x-10, y-10, x+10, y+10, fill='red')
    if not skonci:
        canvas.after(100, kresli)

def kresli1():
    x = random.randrange(600)
    y = random.randrange(450)
    canvas.create_rectangle(x-10, y-10, x+10, y+10, fill='blue')
    if not skonci1:
        canvas.after(300, kresli1)

skonci = False
skonci1 = False
kresli()
kresli1()
print('hotovo')
```

Teraz bežia oba časovače, ale stačí zavolať, napr.

```
>>> skonci = True
```

V tomto momente sa prvý časovač zastaví a beží iba druhý, teda sa kreslia len modré štvorčeky. Ak by sme ho chceli znovu naštartovať, nesmieme zabudnúť zmeniť premennú `skonci` a zavolať `kresli()`:

```
>>> skonci = False
>>> 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, random

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

def kresli():
    x = random.randrange(600)
    y = random.randrange(450)
    canvas.create_oval(x-10, y-10, x+10, y+10, fill=farba)
    if not skonci:
        canvas.after(cas, kresli)

skonci = False
farba = 'red'
cas = 100
kresli()
```

Pohyb 2 obrázkov rôznou rýchlosťou:

```
import tkinter, random

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

obrazok1 = tkinter.PhotoImage(file='auto1.png')
obrazok2 = tkinter.PhotoImage(file='auto2.png')
x1, x2 = 50, 550
prvy = canvas.create_image(x1, 200, image=obrazok1)
druhy = canvas.create_image(x2, 200, image=obrazok2)
dx1, dx2 = 8, 5

def start(e):
    canvas.coords(prvy, x1, 200)
    canvas.coords(druhy, x2, 200)
    pohyb()

def pohyb():
    #global x1, x2
    canvas.move(prvy, dx1, 0)
    #x1 += dx1
    canvas.move(druhy, -dx2, 0)
    #x2 -= dx2
    canvas.update()
    if canvas.coords(prvy)[0]+50 < canvas.coords(druhy)[0]-50:
        canvas.after(50, pohyb)
    else:
        canvas.create_text(300, 250, text='BUM', fill='red', font='arial 40')

canvas.bind('<Button-1>', start)
```

Časovač tu zastane sám pri splnení nejakej podmienky. Lenže klikanie aj počas animácie autíčiek funguje a opätovné spustenie časovača tu môže narobiť nepríjemnosti. Bolo by dobre, keby sme vedeli počas behu časovača **zablokovať** klikanie a po skončení opäť povoliť. Využijeme metódu na zrušenie zviazania udalosti:

metóda `unbind()`

Metóda zruší zviazanie príslušnej udalosti:

```
canvas.unbind(meno_udalosti)
```

Upravíme program:

```
import tkinter, random

canvas = tkinter.Canvas(width=600, height=450, bg='white')
canvas.pack()

obrazok1 = tkinter.PhotoImage(file='auto1.png')
obrazok2 = tkinter.PhotoImage(file='auto2.png')
x1, x2 = 50, 550
prvy = canvas.create_image(x1, 200, image=obrazok1)
druhy = canvas.create_image(x2, 200, image=obrazok2)
dx1, dx2 = 8, 5
```

```
def start(e):
    canvas.unbind('<Button-1>')          # zruší klikaciu udalost'
    canvas.coords(prvy, x1, 200)
    canvas.coords(druhy, x2, 200)
    pohyb()

def pohyb():
    canvas.move(prvy, dx1, 0)
    canvas.move(druhy, -dx2, 0)
    canvas.update()
    if canvas.coords(prvy)[0]+50 < canvas.coords(druhy)[0]-50:
        canvas.after(50, pohyb)
    else:
        canvas.create_text(300, 250, text='BUM', fill='red', font='arial 40')
        canvas.bind('<Button-1>', start)    # obnoví klikaciu udalost'

canvas.bind('<Button-1>', start)
```

KAPITOLA 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()
>>> print(t.pos())
(0.0,0.0)
>>> print(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

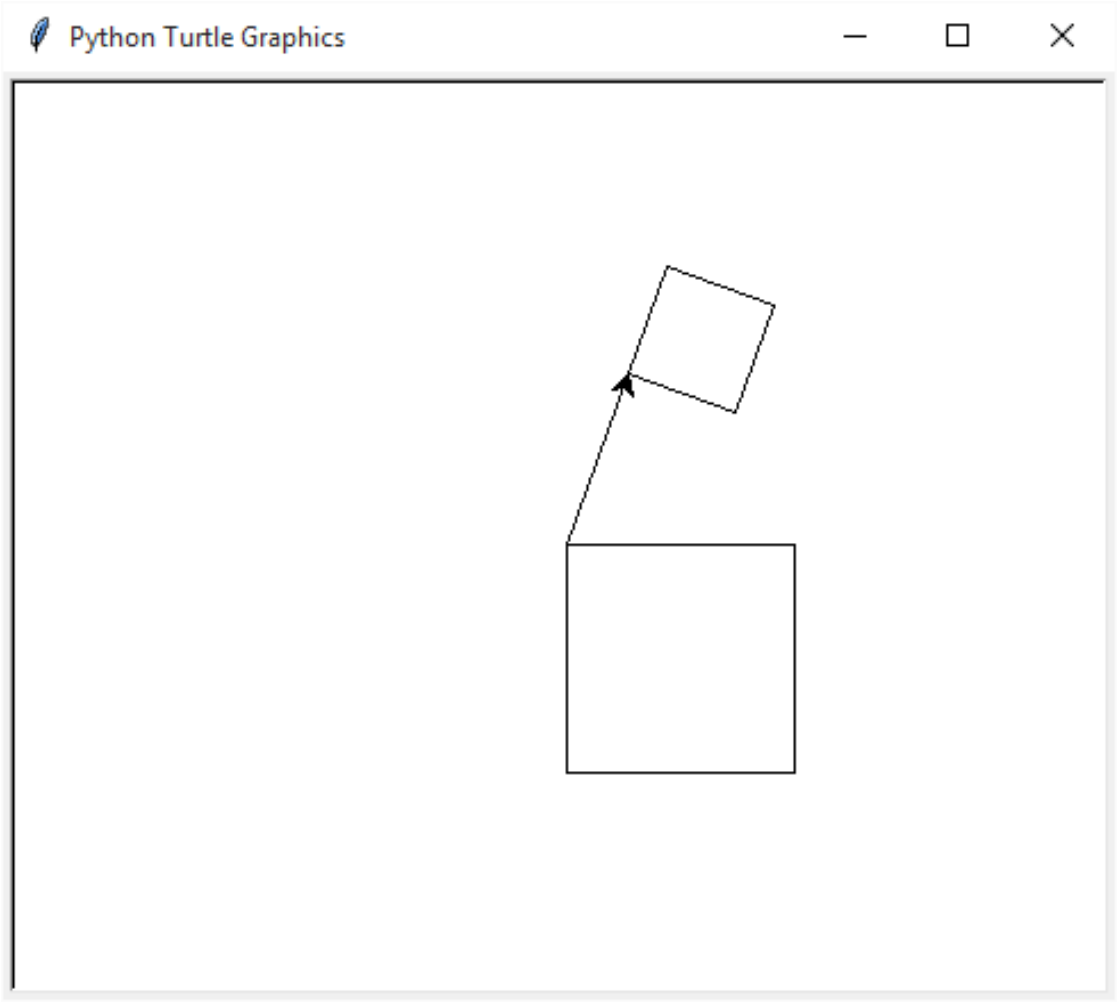
- y-ová súradnica ide nahor zvislo od počiatku: smerom nahor sú kladné y-ové hodnoty, smerom nadol idú záporné hodnoty súradníc
- smer natočenia určujeme v stupňoch (nie v radiánoch) a v protismere otáčania hodinových ručičiek:
  - na východ je to 0
  - na sever je to 90
  - na západ je to 180
  - na 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).

Zapíšme funkciu, pomocou ktorej korytnačka nakreslí štvorec:




```
def stvorec(dlzka):
    for i in range(4):
        t.fd(dlzka)
        t.rt(90)

import turtle
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:

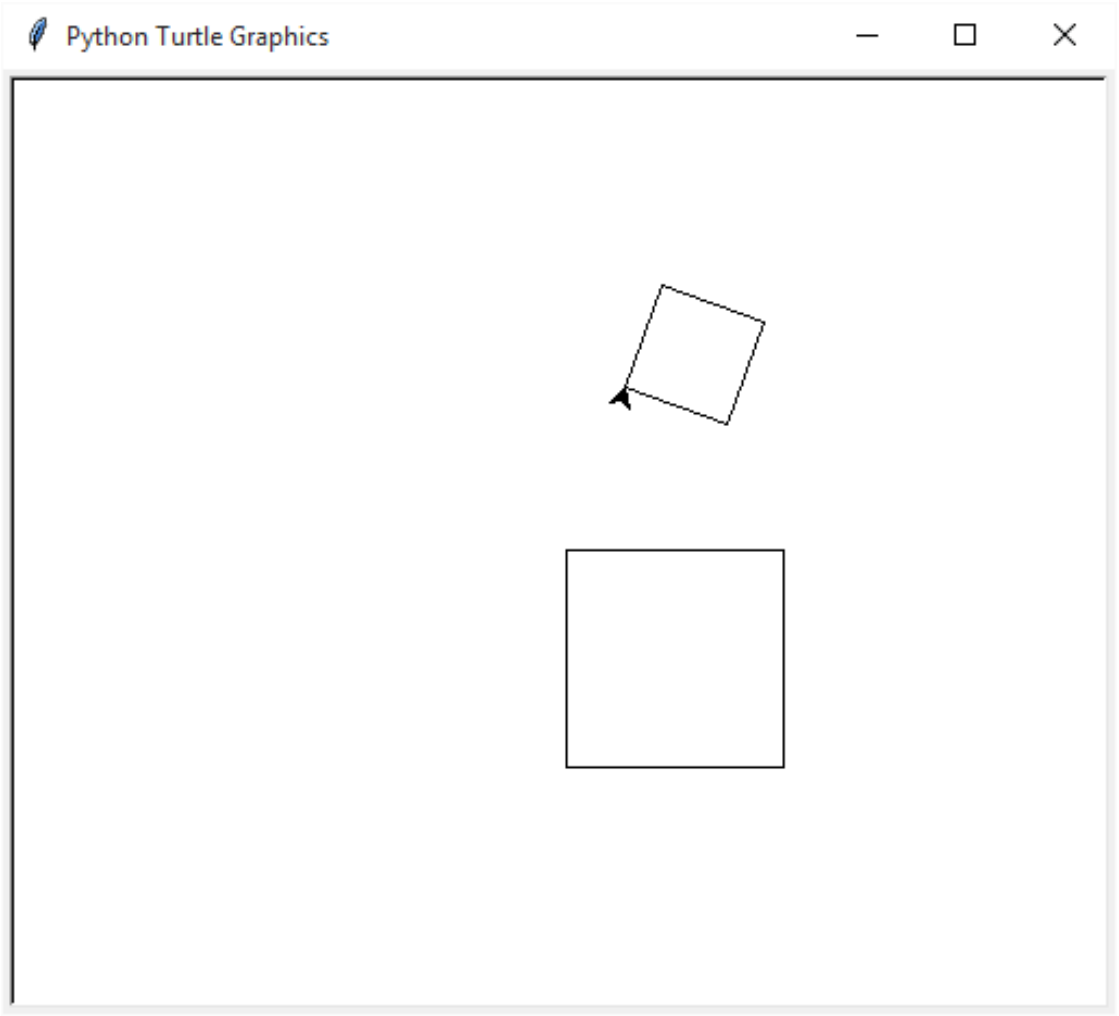
```
>>> t.clear()      # zmaže grafickú plochu a korytnačku nechá tam, kde sa␣
↪momentálne nachádza
```

alebo pri zmazaní plochy aj inicializuje korytnačku v strede plochy otočenú na východ:

```
>>> t.reset()      # zmaže grafickú plochu a inicializuje korytnačku
```

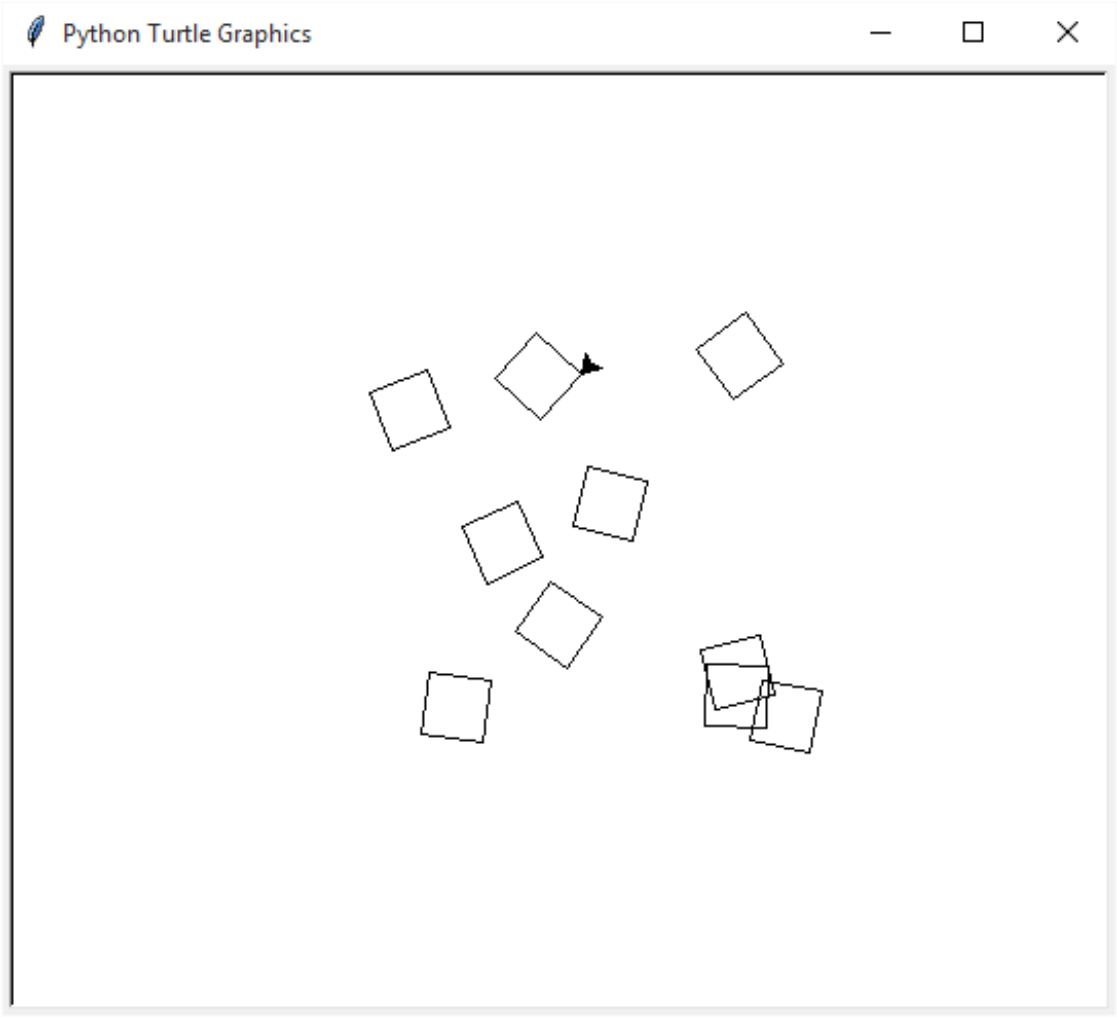
a teraz znovu kreslíme do prázdnej plochy.

Korytnačka má pero, ktorým pri svojom pohybe po grafickej ploche kreslí. 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:




```
def stvorec(dlzka):
    for i in range(4):
        t.fd(dlzka)
        t.rt(90)

import turtle
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:



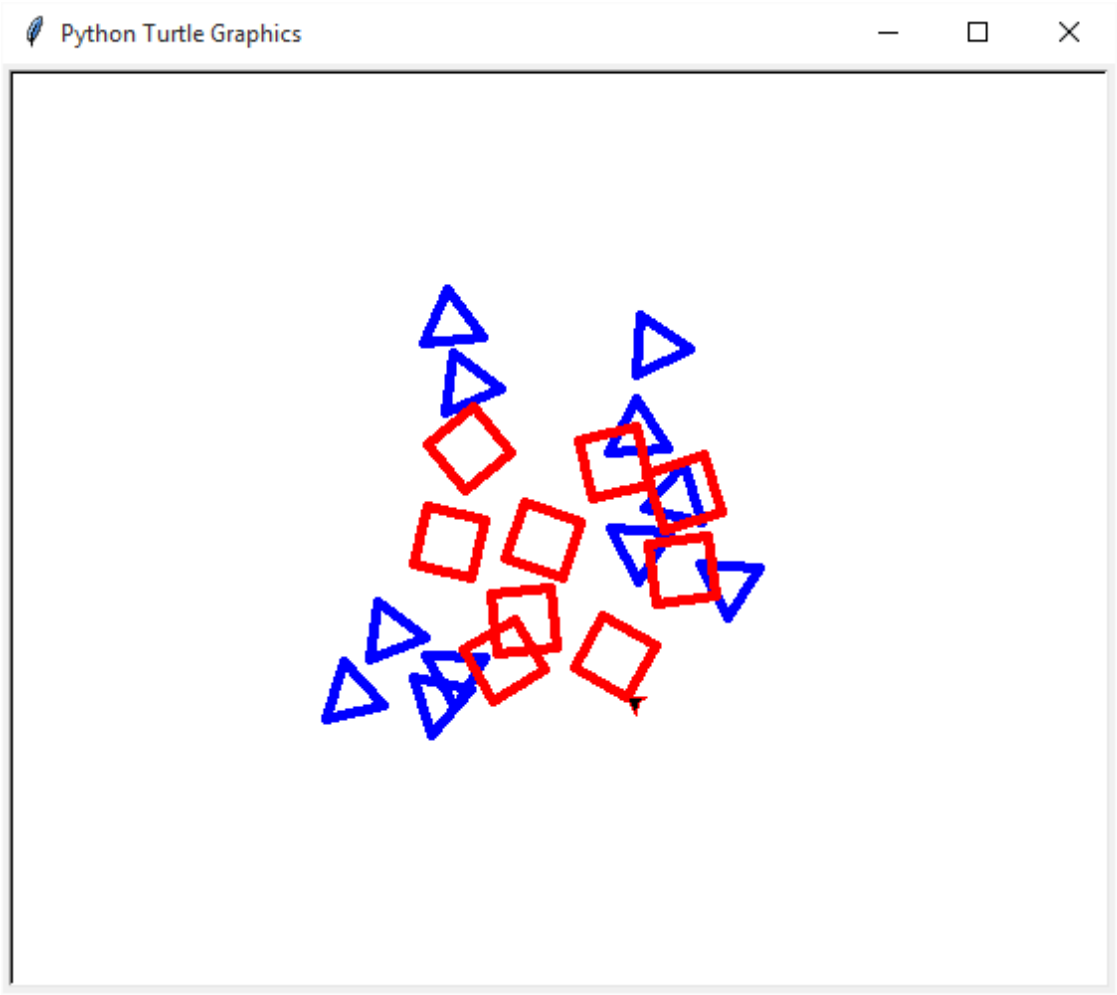

```
def posun():
    t.pu()
    t.setpos(random.randint(-300, 300),
             random.randint(-300, 300))
    t.seth(random.randrange(360))
    t.pd()

def stvorec(dlzka):
    for i in range(4):
        t.fd(dlzka)
        t.rt(90)

import turtle, random
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):

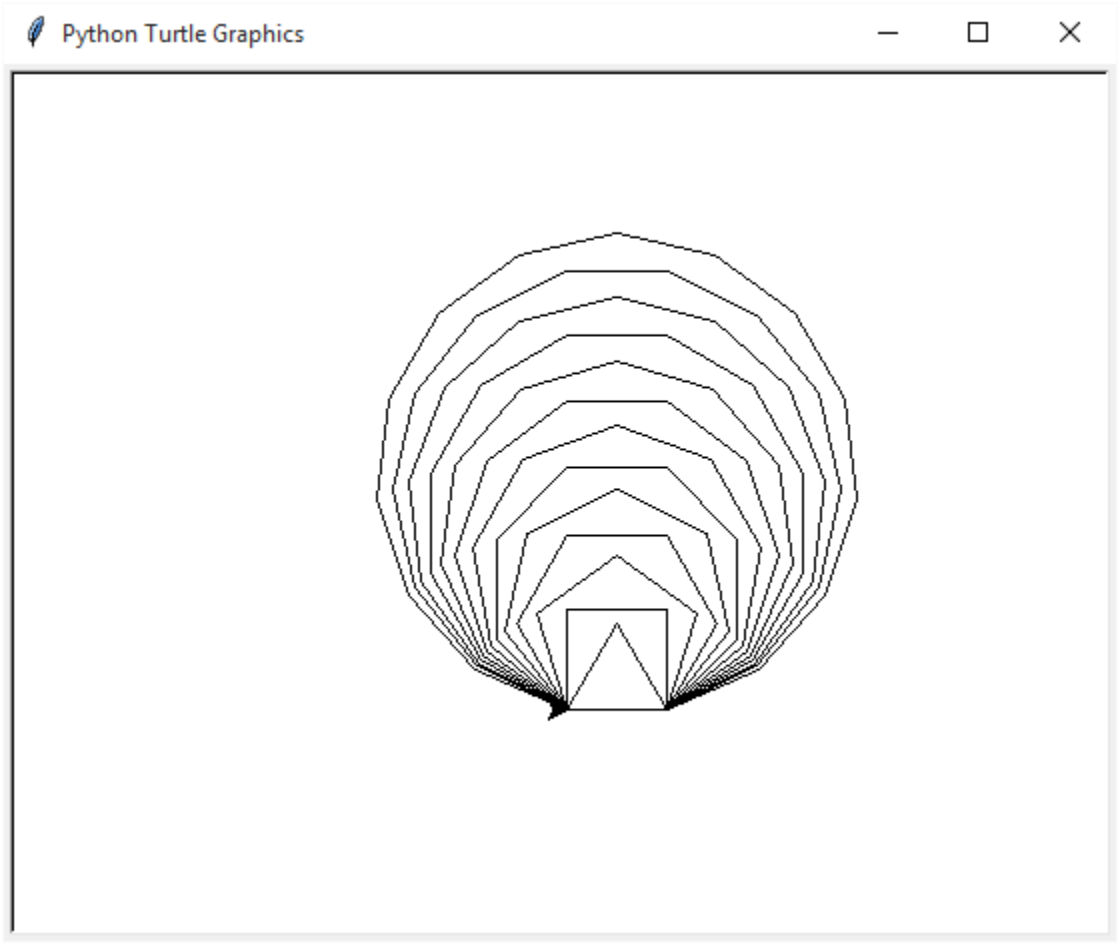



```
def stvorec(dlzka):
    for i in range(4):
        t.fd(dlzka)
        t.rt(90)

def trojuholnik(dlzka):
    for i in range(3):
        t.fd(dlzka)
        t.rt(120)

def posun():
    t.pu()
    t.setpos(random.randint(-300, 300),
             random.randint(-300, 300))
```

```
    t.seth(random.randrange(360))
    t.pd()

import turtle, random
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 n-uholník a tiež príkaz `clear()`, ktorý zmaže nakreslený obrázok, aby sa mohol kresliť ďalší v prázdnej grafickej ploche:



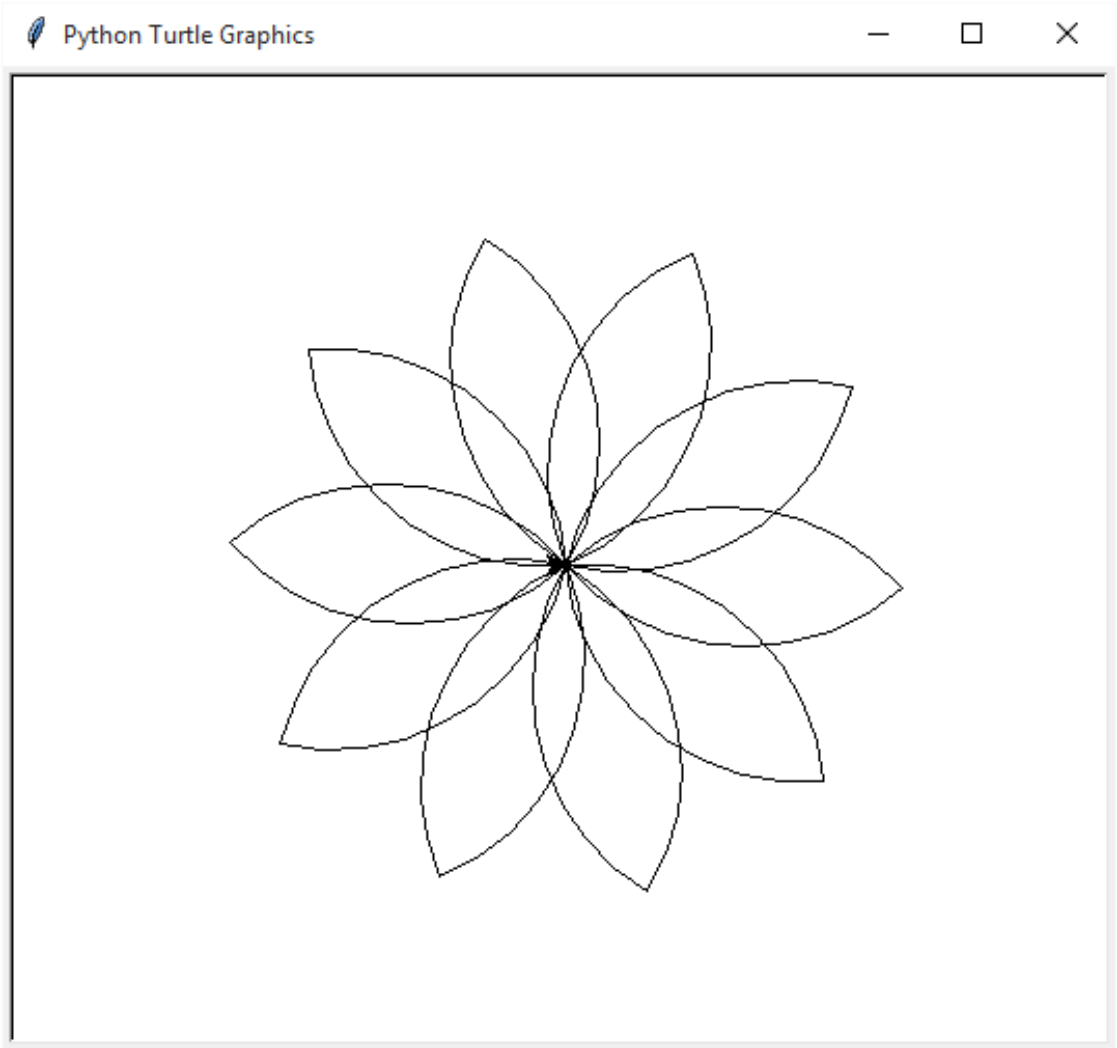

```
def n_uholnik(n, d):
    for i in range(n):
        t.fd(d)
```

```
        t.lt(360/n)

import turtle
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 kresliť 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.

- nasledovní príklad najprv definuje `oblúk` (štvrťkruh), potom `lupen` (dva priložené štvrťkruhy) a nakoniec kvet ako `n` lupeňov:

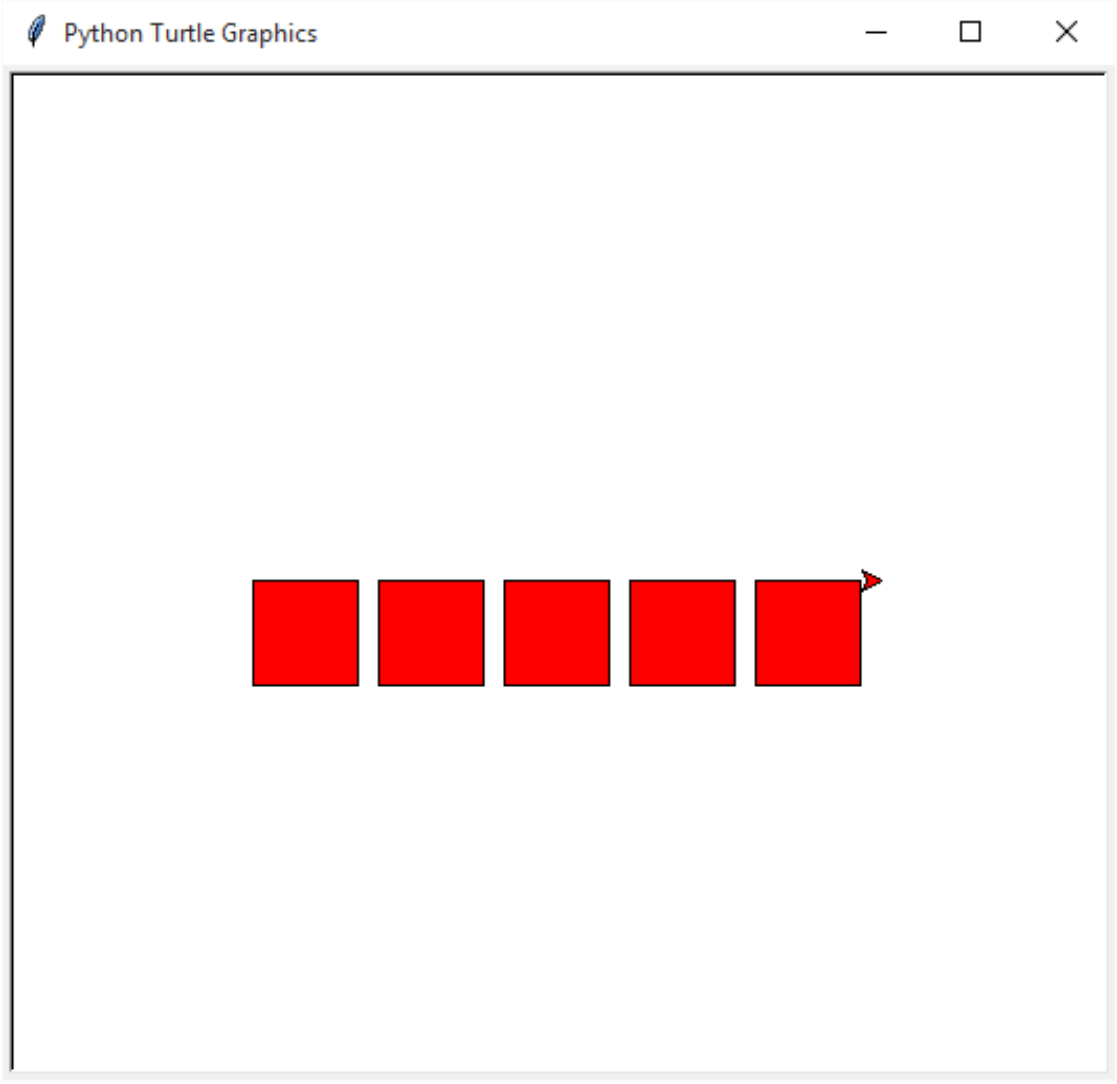



```
def obluk(d):
    for i in range(9):
        t.fd(d)
        t.rt(10)

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)

import turtle
turtle.delay(0)
t = turtle.Turtle()
kvet(10, 20)
```

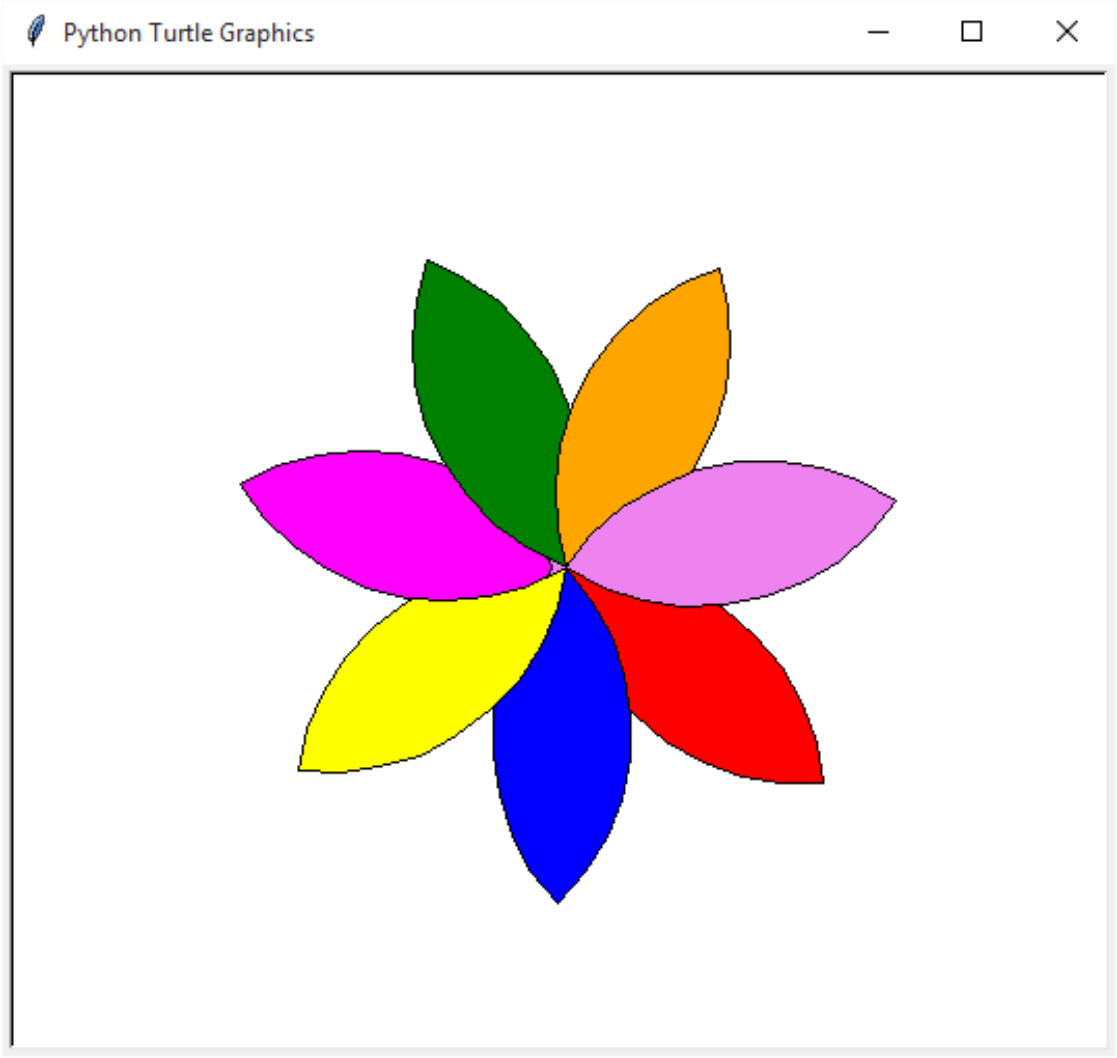
11.2 Vyfarbenie útvaru

Útvary, ktoré nakreslí korytnačka sa dajú aj vyfarbiť. Princíp je taký, že na začiatok postupnosti korytnačích príkazov umiestnime príkaz `begin_fill()`. Na koniec tejto postupnosti dáme príkaz `end_fill()`, ktorý vyfarbí kreslený útvar farbou výplne (na začiatku je nastavená čierna farba). Farbu výplne meníme príkazom `fillcolor(farba)`. Napr. farebné štvorce v jednom rade:




```
def stvorec(d):
    for i in range(4):
        t.fd(d)
        t.rt(90)

import turtle
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()
```

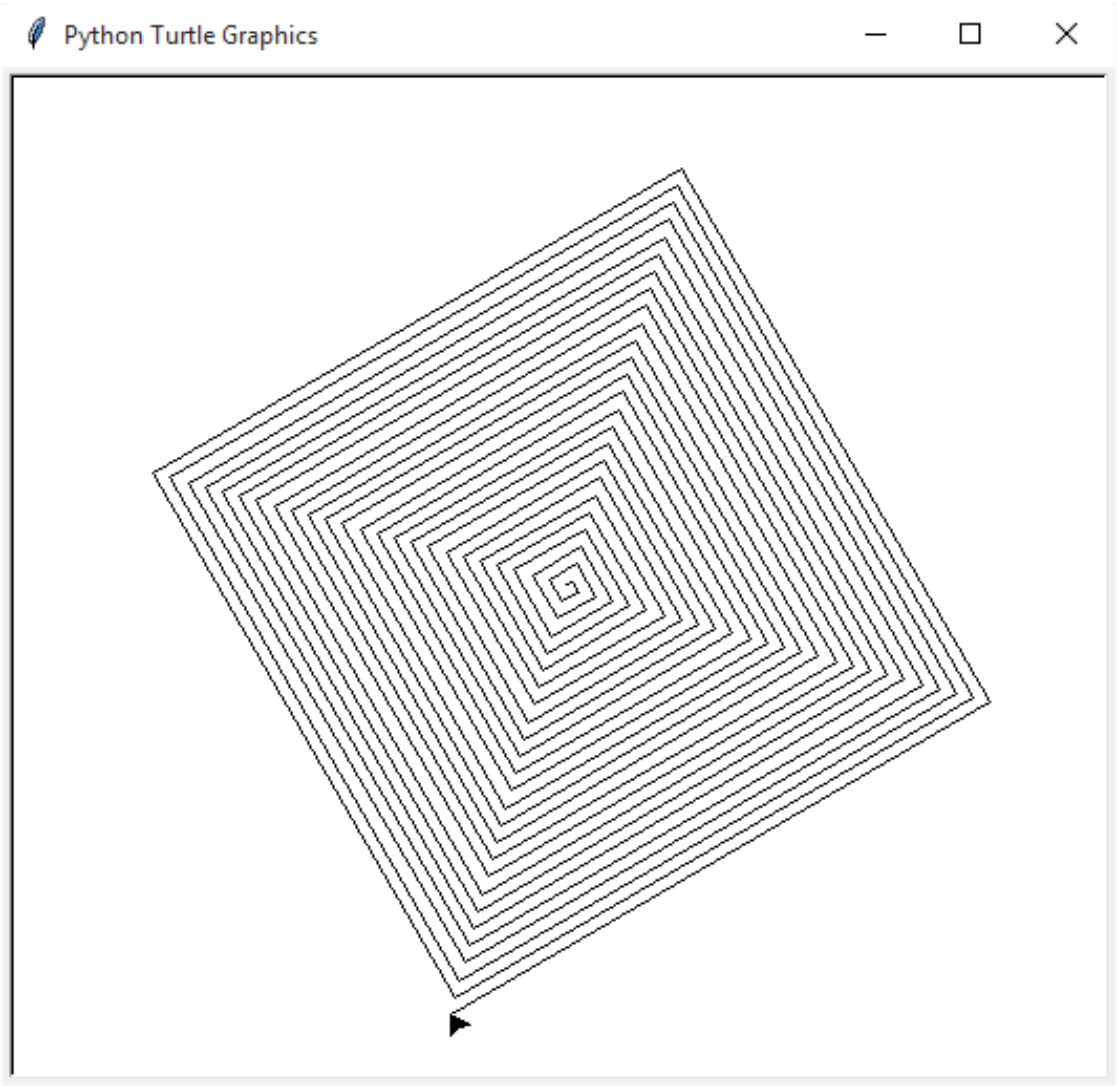
alebo kvet zložený z farebných lupeňov:




```
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))

import turtle
turtle.delay(0)
t = turtle.Turtle()
```

```
kvet(20, ['red', 'blue', 'yellow', 'magenta',
          'green', 'orange', 'violet'])
```

Špirály

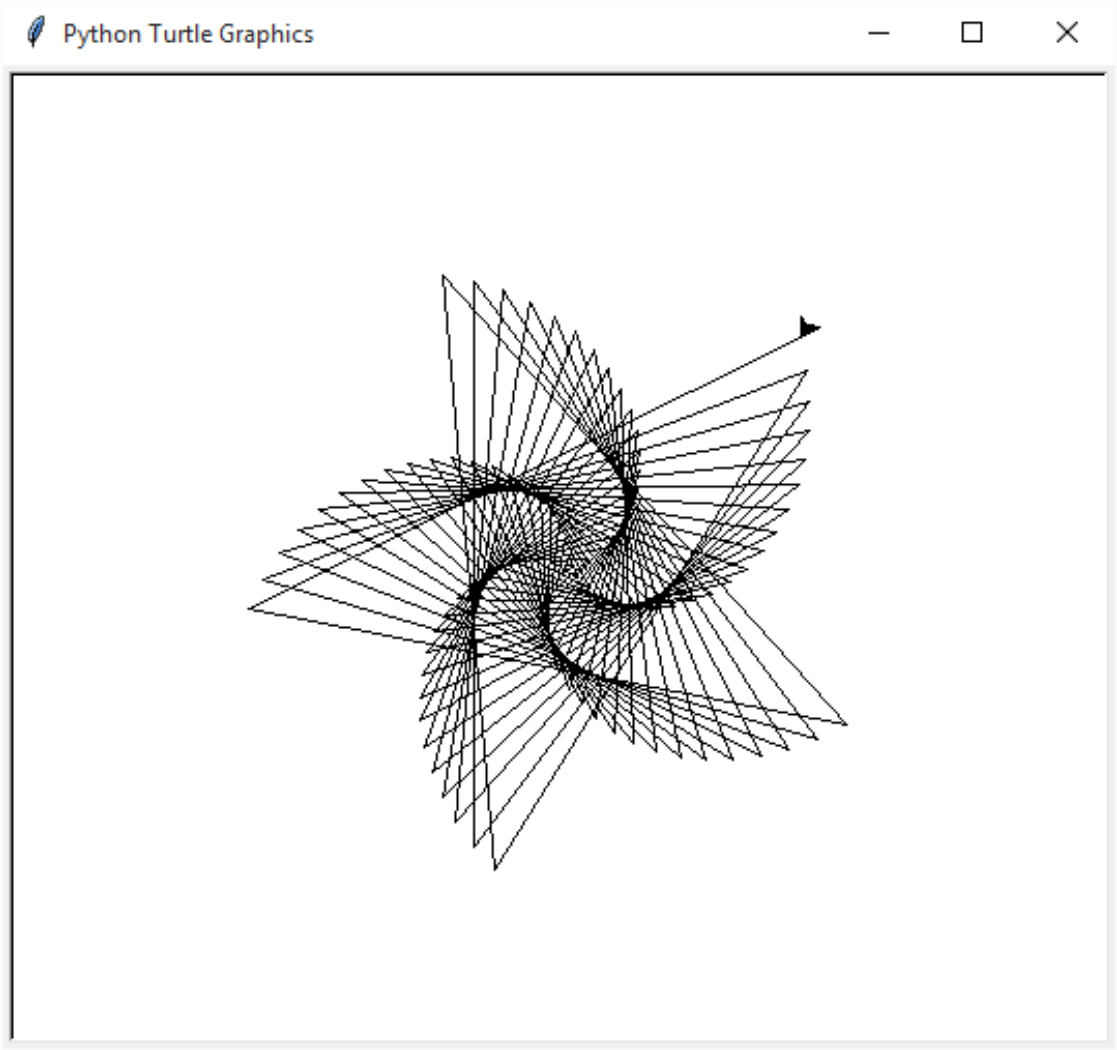


Rôzne špirály môžeme kresliť tak, že opakujeme kreslenie stále sa zväčšujúcich čiar a za každým sa otočíme o pevný uhol, napr.

```
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.

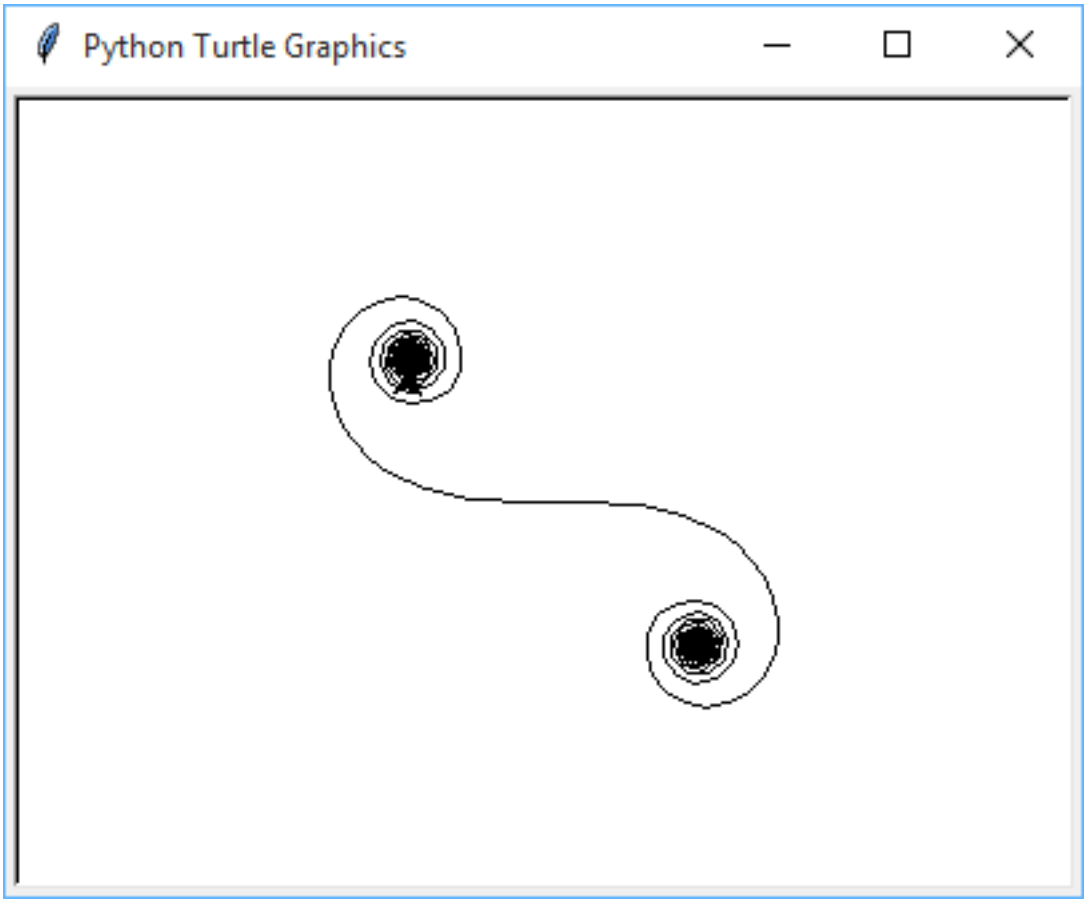
S uhlami môžete experimentovať, napr.




```
import turtle, random

turtle.delay(0)
t = turtle.Turtle()
while True:
    uhol = random.randint(30, 170)
    print('spirala s uhlom', uhol)
    for i in range(3, 300, 3):
        t.fd(i)
        t.rt(uhol)
    t.reset()
```

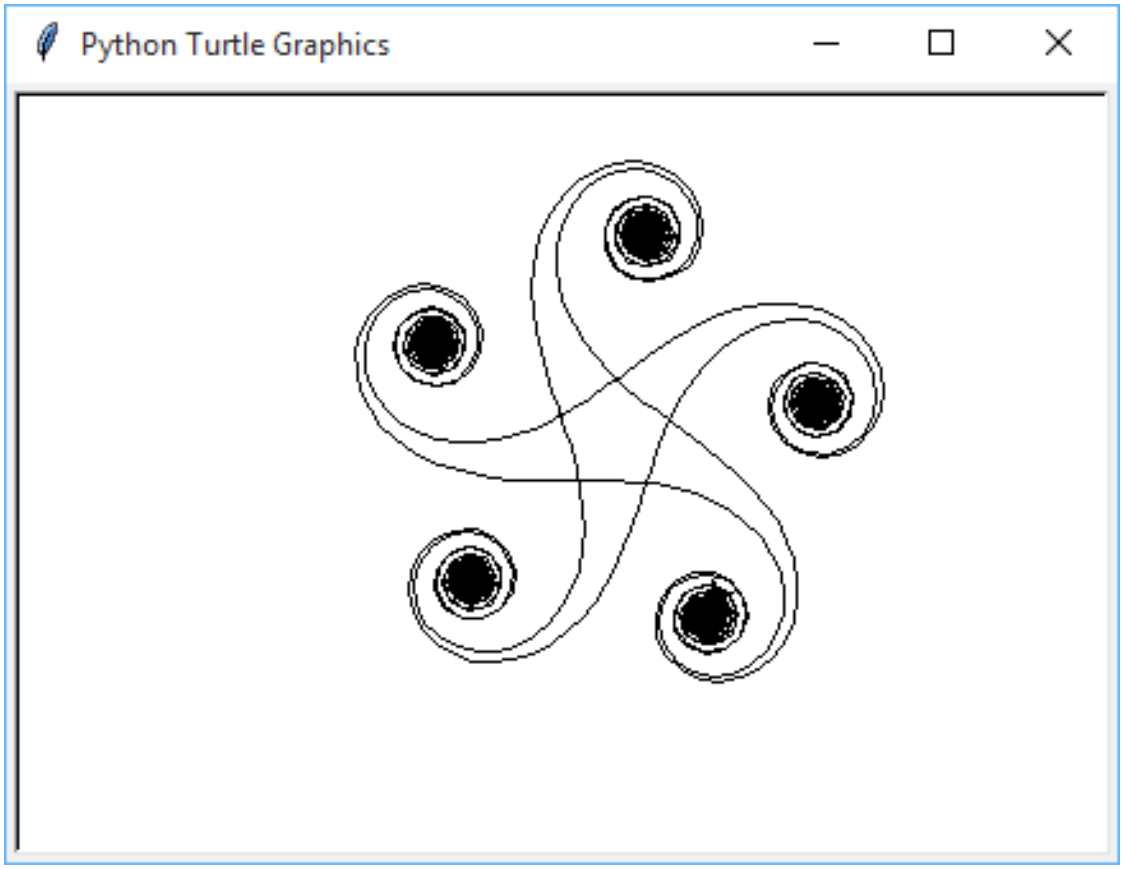
Tento program kreslí špirály s rôznymi náhodne generovanými uhlami. Zároveň do textovej plochy vypisuje informáciu o uhle momentálne kreslenej špirály.

Zaujímavé špirály vznikajú, keď nemeníme dĺžku čiar ale uhol, napr.




```
import turtle

turtle.delay(0)
t = turtle.Turtle()
for uhol in range(1, 2000):
    t.fd(8)
    t.rt(uhol)
```

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()`.

11.3 Zhrnutie užitočných metód

Tabuľka 11.1: korytnačie príkazy

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(u)`	`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()`

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

Tvar korytnačky

Korytnačkám môžeme meniť ich tvar - momentálne je to malý trojuholník.

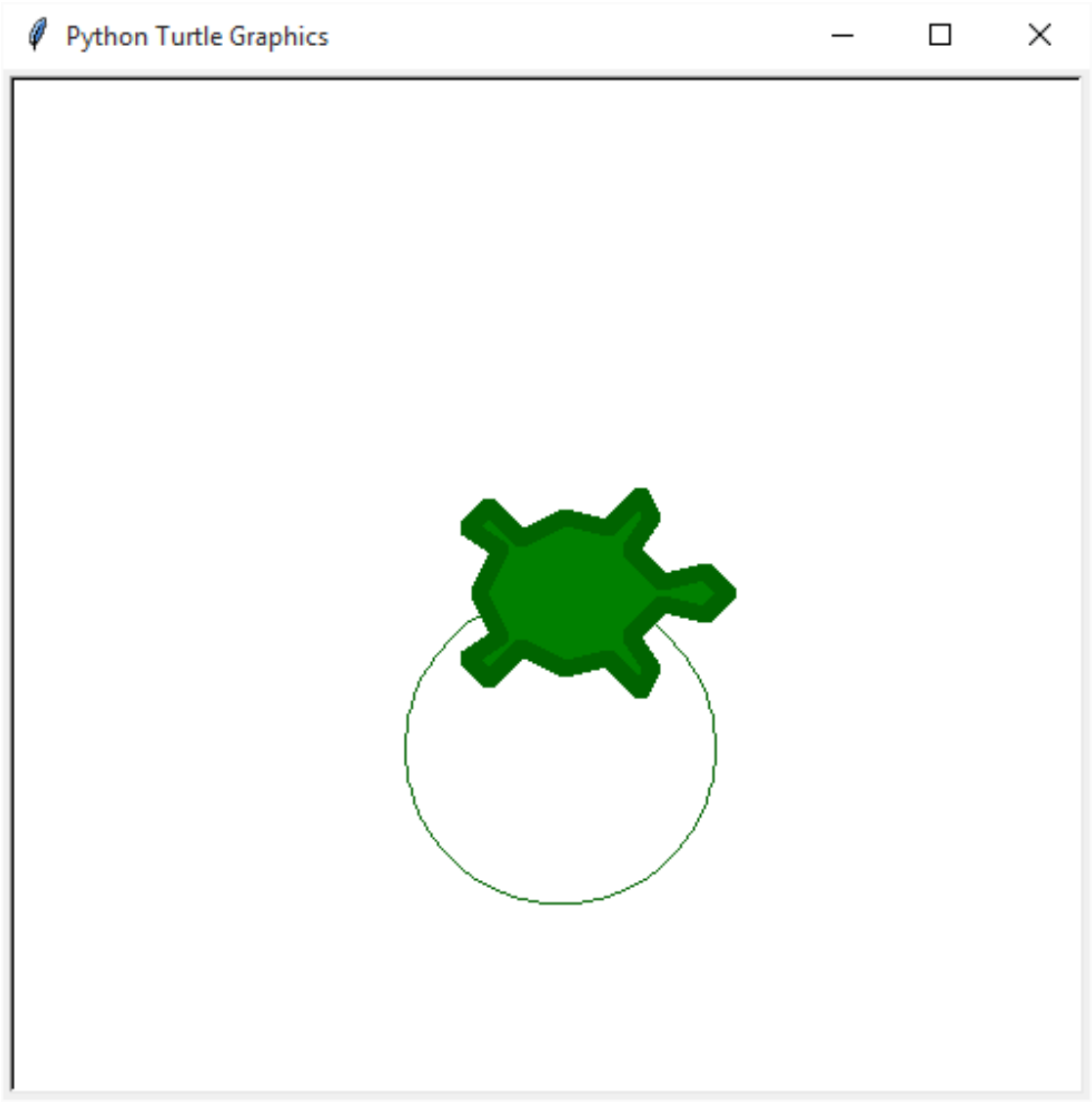
Príkaz `shape()` zmení tvar na jeden s preddefinovaných:

- `'arrow'`, `'turtle'`, `'circle'`, `'square'`, `'triangle'`, `'classic'` (default je `'classic'`)

Príkaz `shapesize()` nastavuje zväčšenie tvaru a hrúbku obrysu tvaru:

- `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, 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)
```

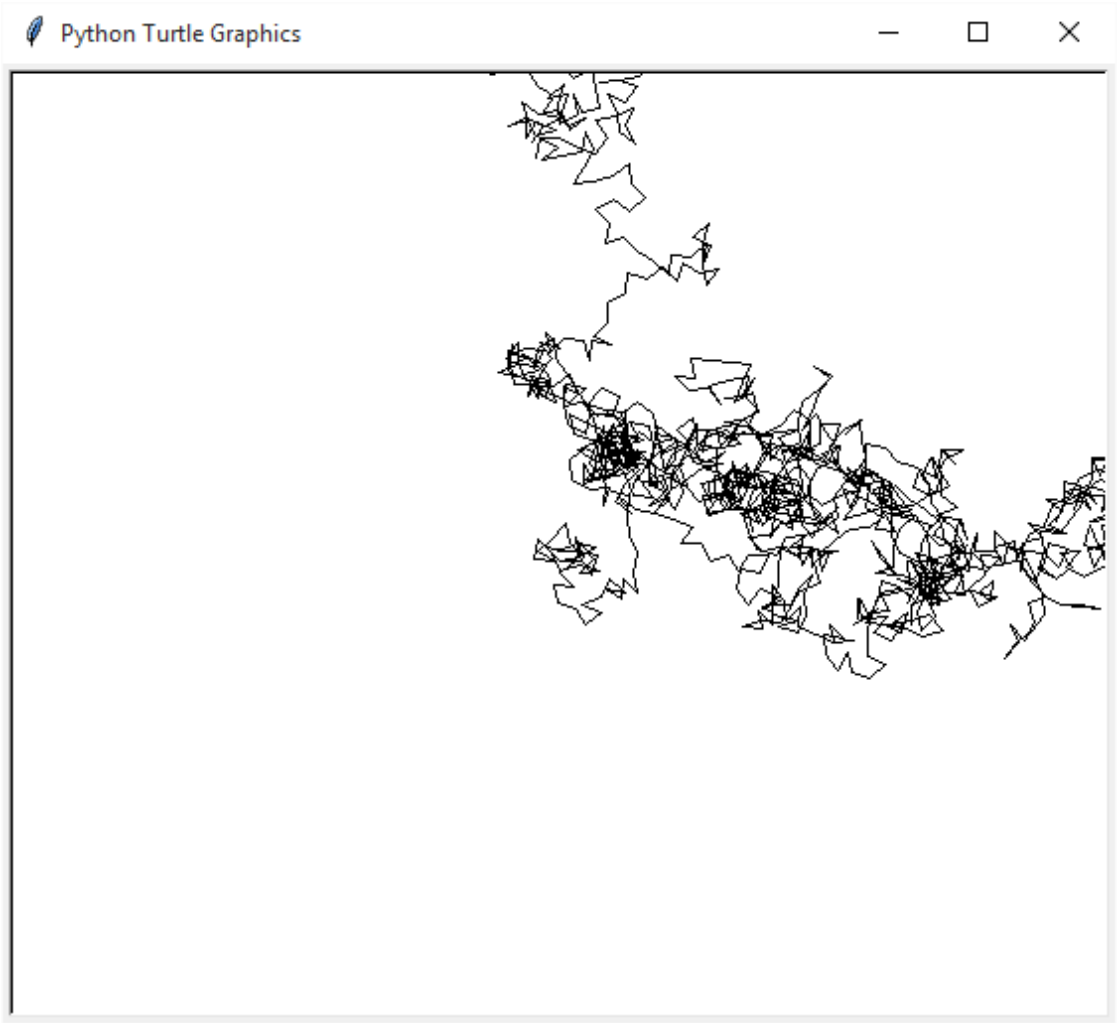
```
    t.rt(4)
```

- 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()`).

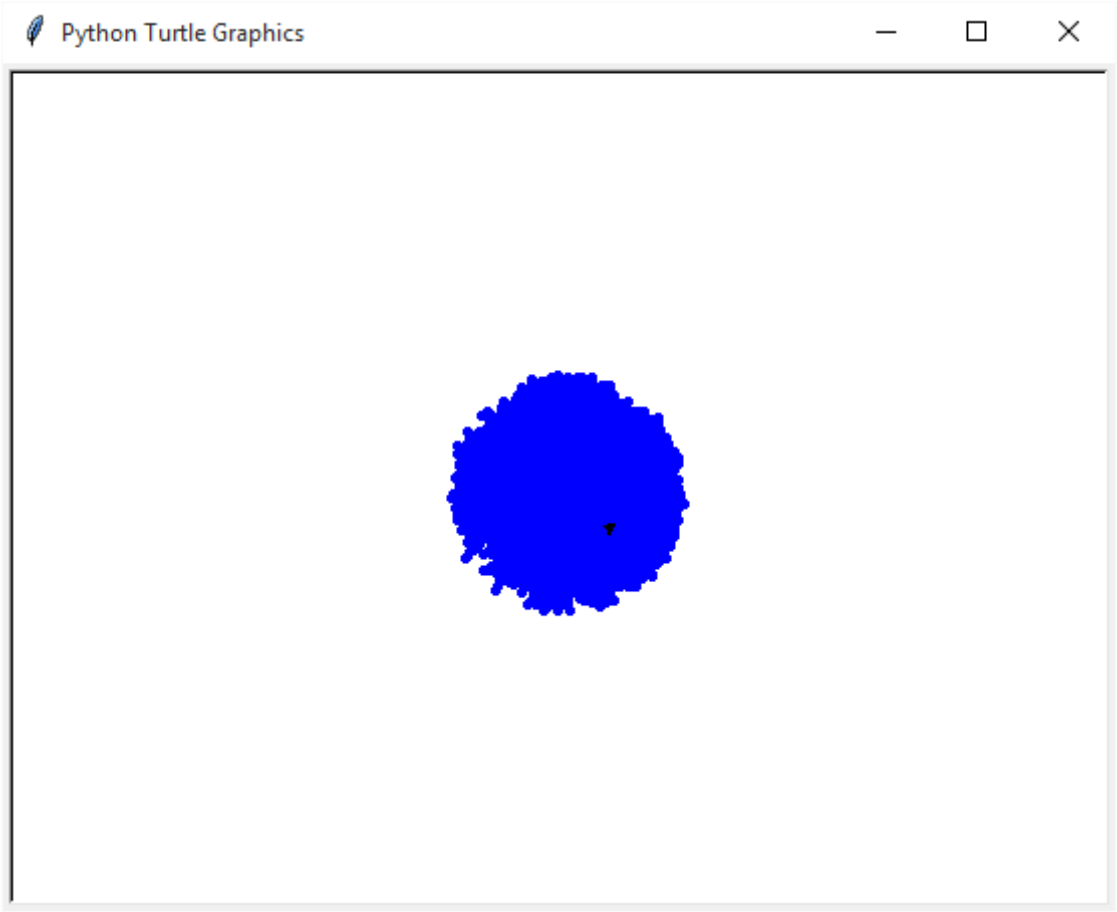
11.4 Náhodné prechádzky

Pohyb korytnačky, pri ktorom sa veľa krát náhodne otočí a prejde malú vzdialenosť, napr.



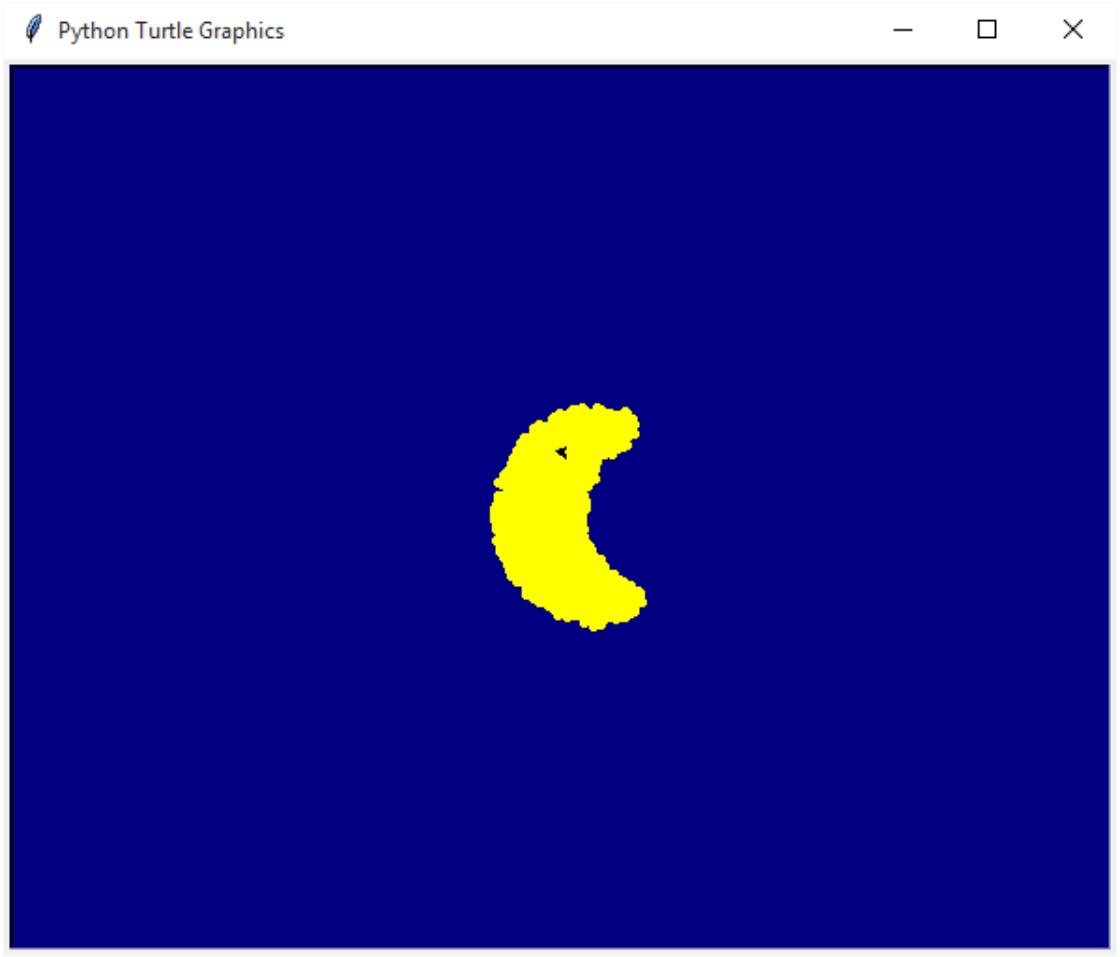

```
import turtle, random
turtle.delay(0)
t = turtle.Turtle()
for i in range(10000):
    t.seth(random.randrange(360))
    t.fd(10)
```

po čase odíde z grafickej plochy - upravíme tak, aby nevyšla z nejakej oblasti, napr.




```
import turtle, 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)
```

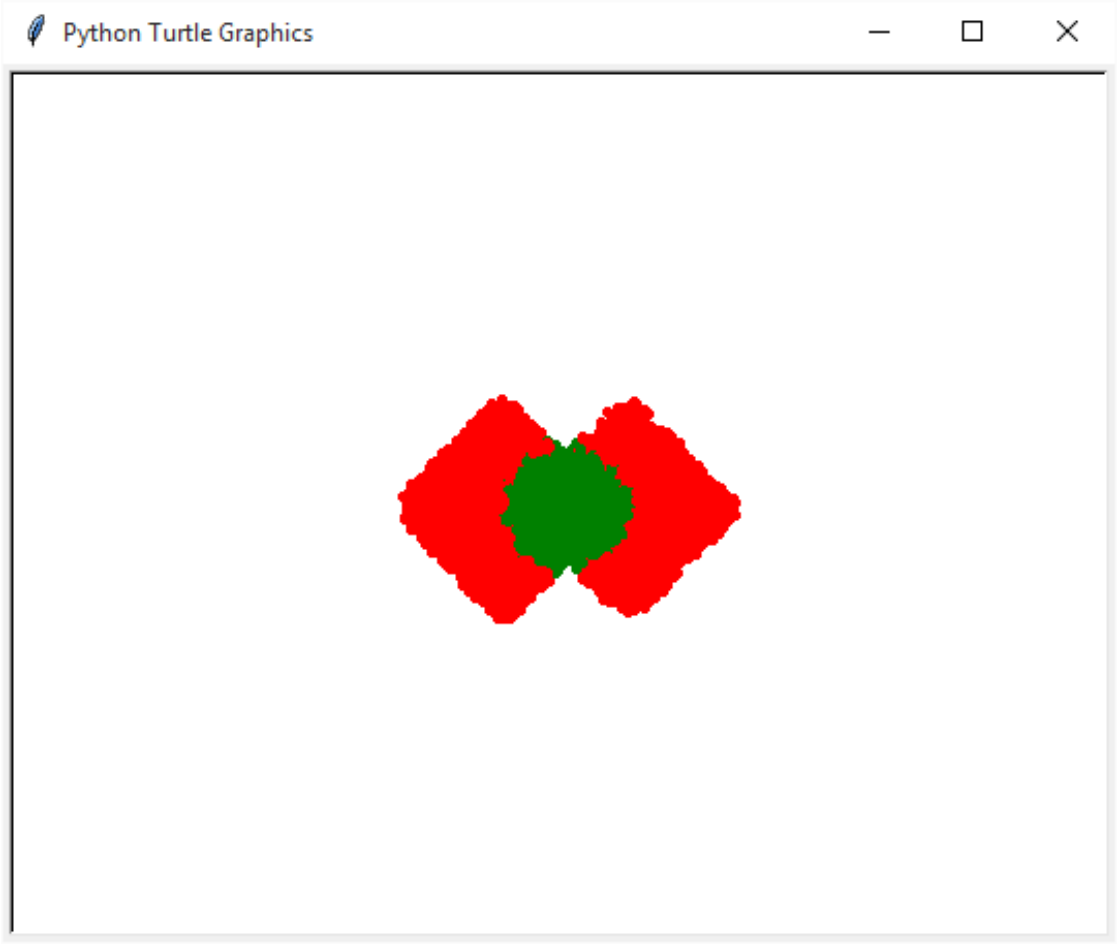
Príkaz `if` nedovolí korytnačke vzdialiť sa od (0,0) viac ako 50 - počítali sme tu vzdialenosť korytnačky od počiatku môžeme využiť metódu `distance` ktorá vráti vzdialenosť korytnačky od nejakého bodu alebo inej korytnačky:




```
import turtle, 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, ktorý ma tvar mesiaca: nesmie vyjsť z prvého kruhu a zároveň vojsť do druhého

tvar stráženej oblasti môže byť definovaný aj zložitejšou funkciou, napr.




```
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

import turtle, random
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 oblasť
        t.fd(-5)
```

Okrem stráženia oblasti tu meníme farbu pera podľa nejakej podmienky

11.5 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:

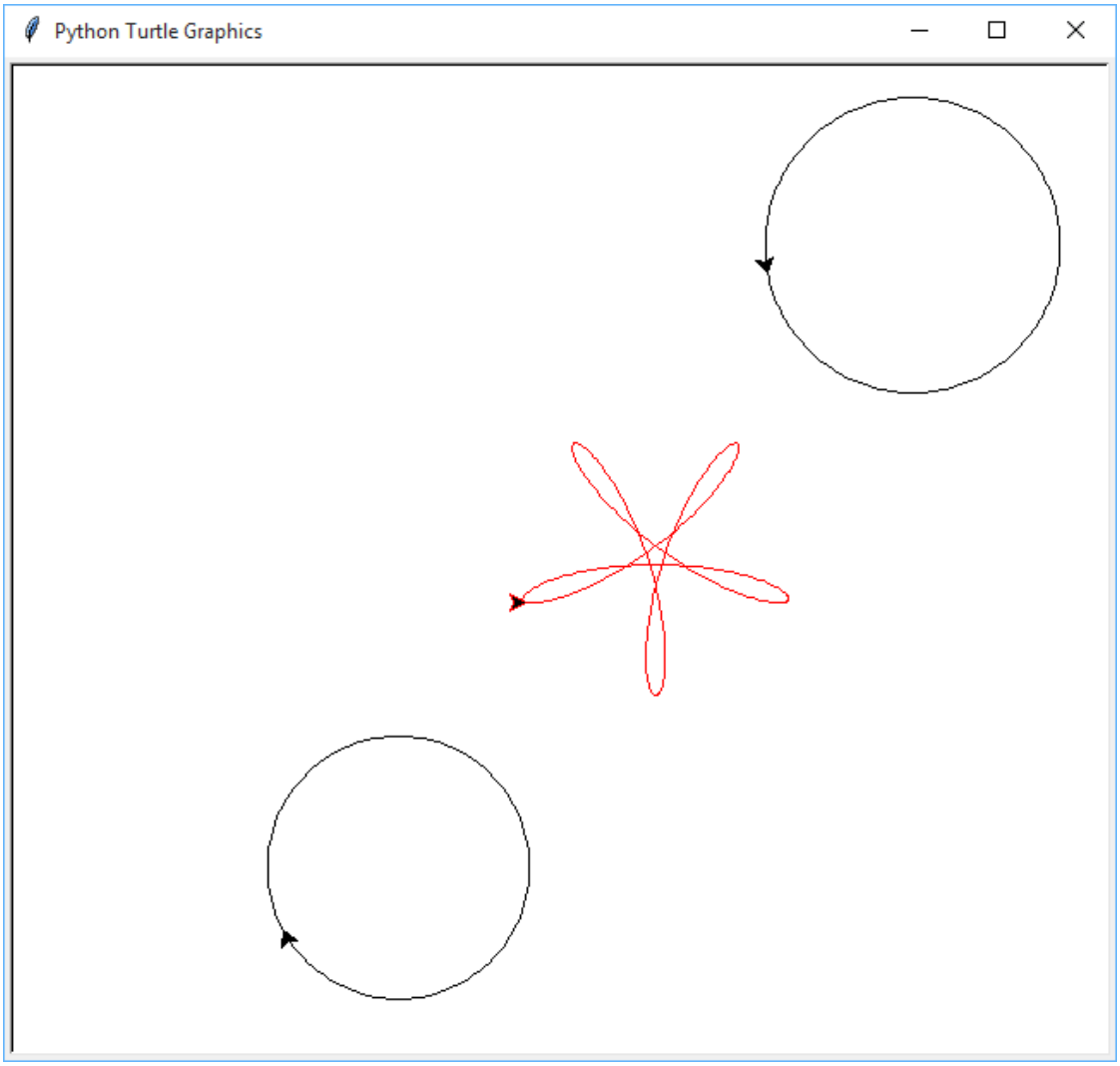
```
def stvorec(tu, velkost):
    for i in range(4):
        tu.fd(velkost)
        tu.rt(90)

import turtle
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):




```
def posun(t, pos, pero=True):  # pos je pozícia v tvare (x,y)
    if pero:
        t.pu()
    t.setpos(pos)
    if pero:
        t.pd()

def stred(t1, t2):
    x = (t1.xcor()+t2.xcor())/2
    y = (t1.ycor()+t2.ycor())/2
    return (x, y)

import turtle
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')
```

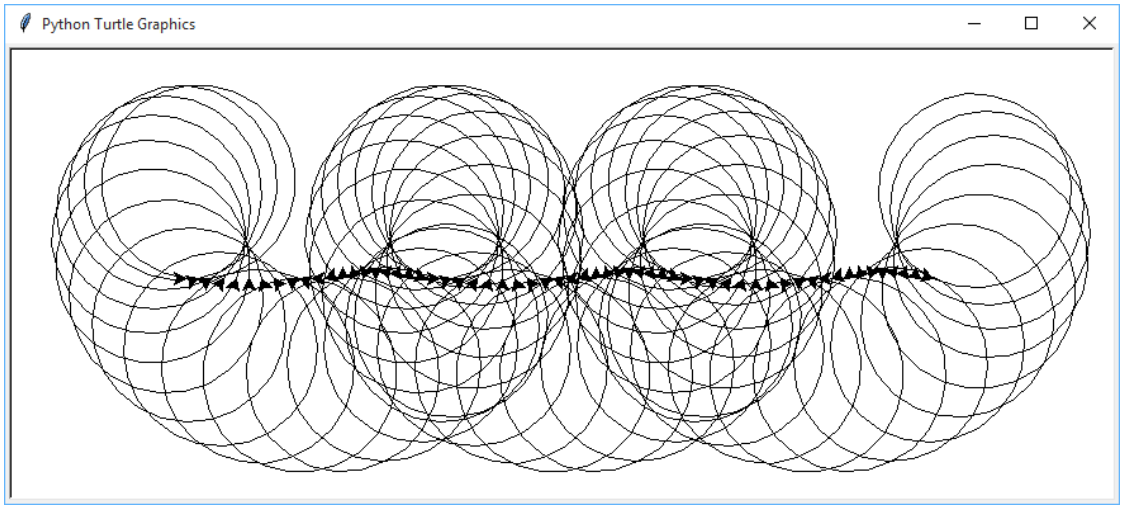
```
while True:
    t1.fd(4)
    t1.rt(3)
    t2.fd(3)
    t2.lt(2)
    posun(t3, stred(t1,t2), False)
```

Pole korytnačiek

Do poľa 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)
pole = []
for i in range(60):
    t = turtle.Turtle()
    t.pu()
    t.setpos(-300+10*i, 0)
    t.pd()
    t.seth(i*18)
    pole.append(t)
```

Necháme ich kresliť rovnaké kružnice:




```
import turtle
turtle.delay(0)
pole = []
for i in range(60):
    t = turtle.Turtle()
    t.pu()
    t.setpos(-300+10*i, 0)
    t.pd()
    t.seth(i*18)
    pole.append(t)

for t in pole:
    for i in range(24):
        t.fd(20)
        t.lt(15)
```

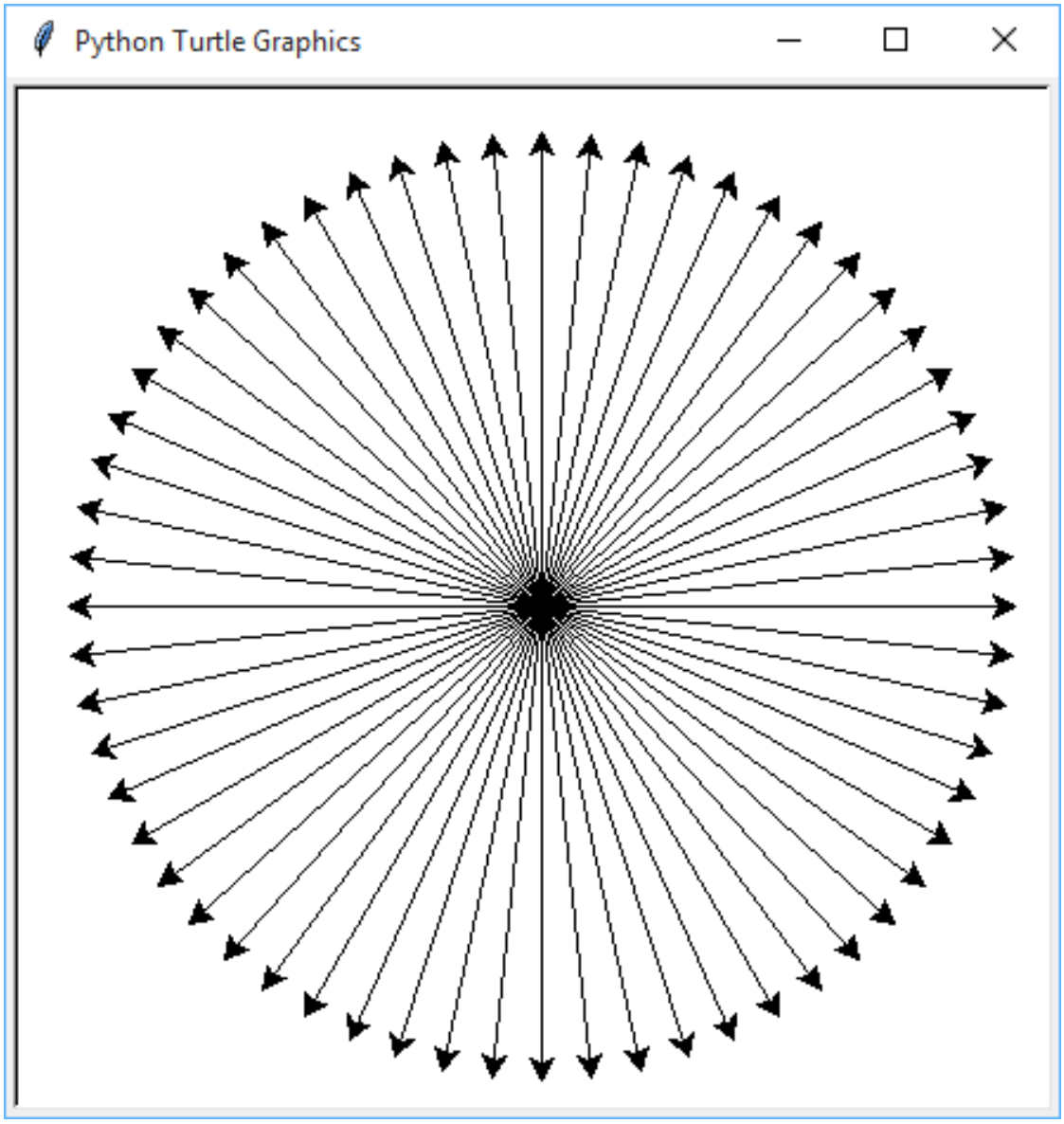
Takto kreslila jedna za druhou: ďalšia začala kresliť až vtedy, keď predchádzajúca skončila

Pozmeňme to tak, aby všetky kreslili naraz:

```
import turtle
turtle.delay(0)
pole = []
for i in range(60):
    t = turtle.Turtle()
    t.pu()
    t.setpos(-300+10*i, 0)
    t.pd()
    t.seth(i*18)
    pole.append(t)

for i in range(24):
    for t in pole:
        t.fd(20)
        t.lt(15)
```

zmenili sme len poradie for-cyklov

V ďalšom príklade vygenerujeme všetky korytnačky v počiatku s rôznymi smermi a necháme ich prejsť dopredu:




```
import turtle
turtle.delay(0)
pole = []
for i in range(60):
    pole.append(turtle.Turtle())
    pole[-1].seth(i*6)

for t in pole:
    t.fd(200)
```

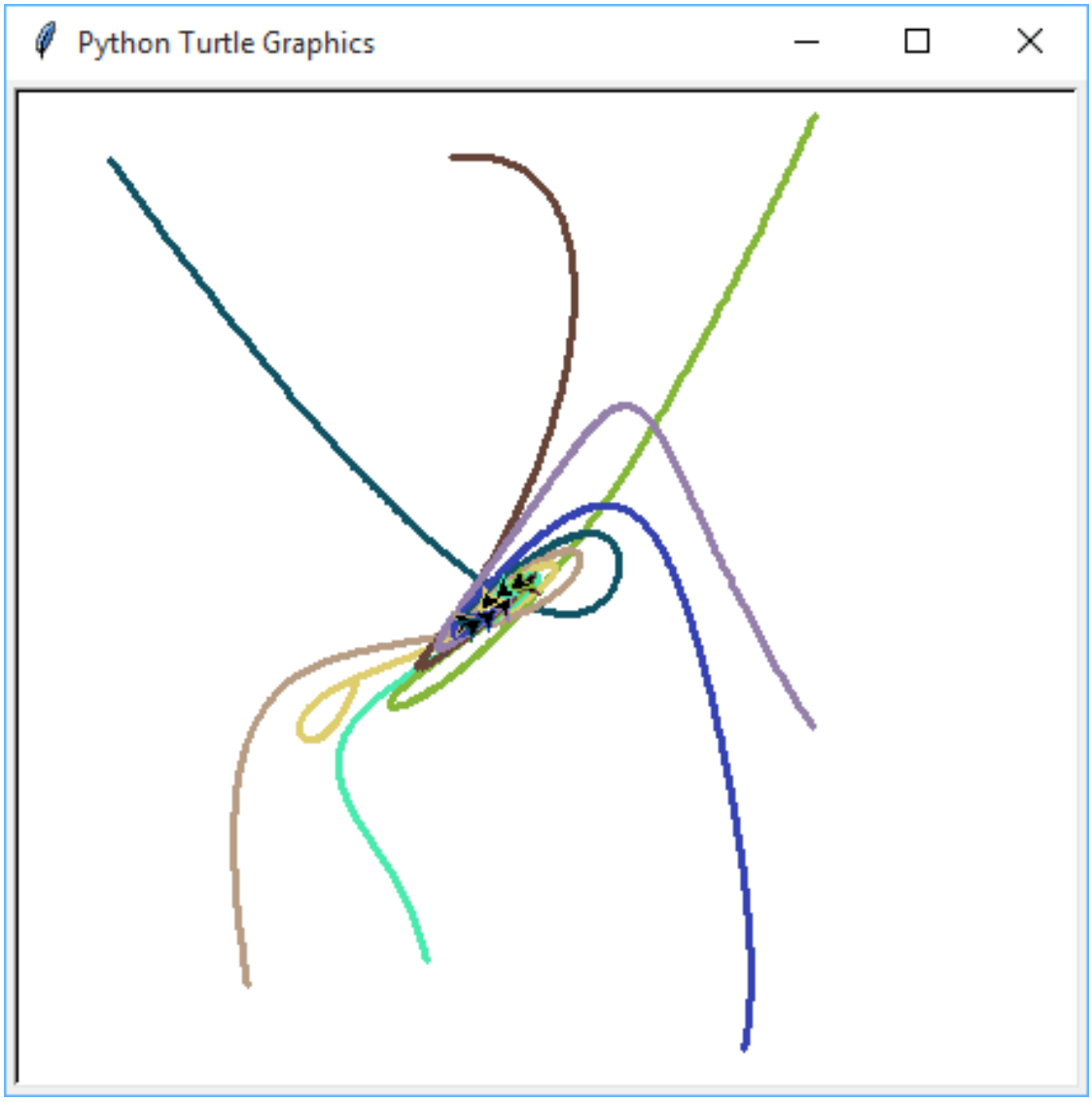
všimnite si ako pracujeme s poľom korytnačiek (`pole[-1]` označuje posledný prvok poľa, t.j. posledne vygenerovanú korytnačku)

Korytnačky sa naháňajú

Na náhodných pozíciách vygenerujeme `n` korytnačiek a potom ich necháme sa naháňať 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

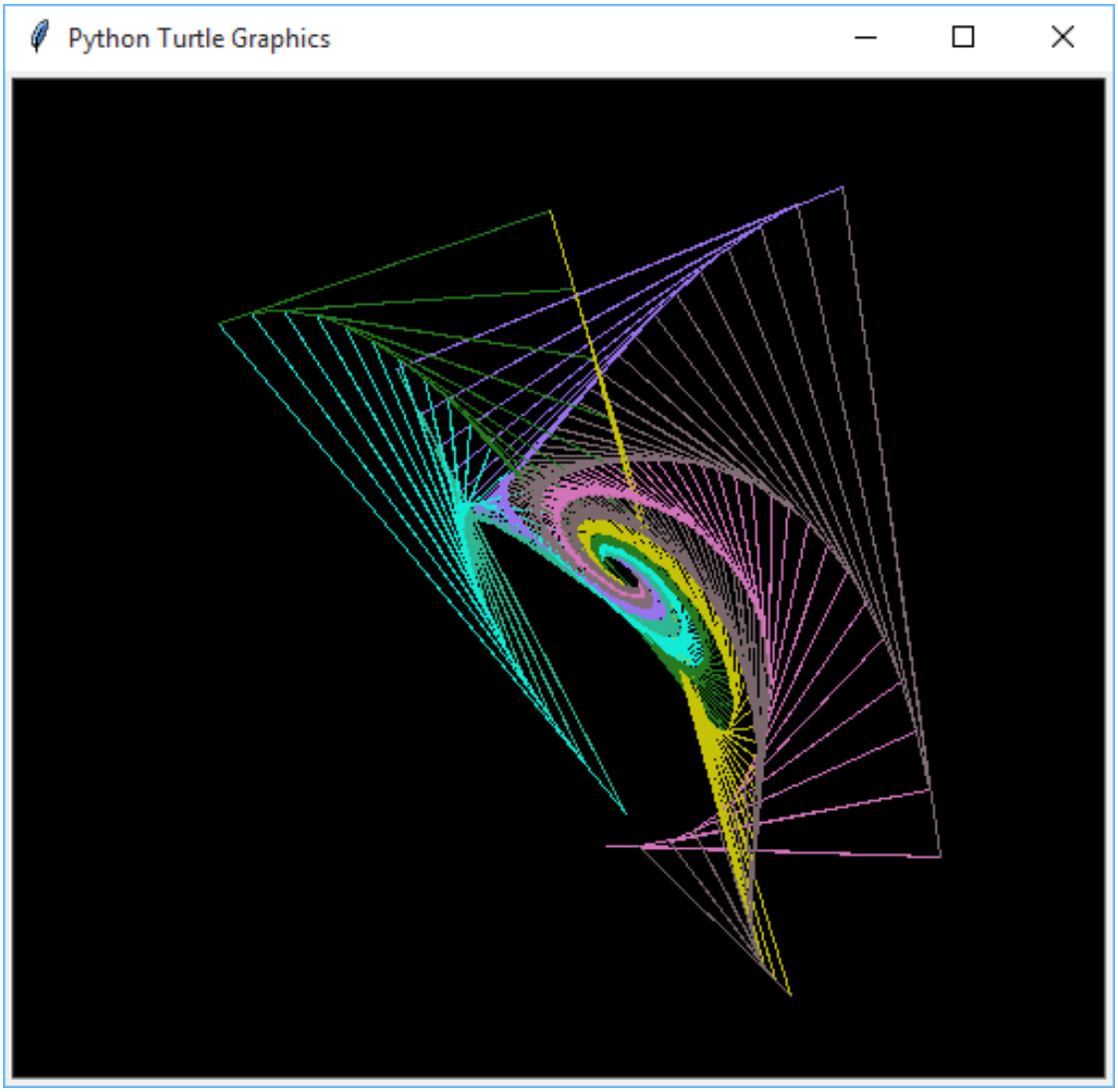



```
import turtle, random

n = 8
t = []
turtle.delay(0)
for i in range(n):
    nova = turtle.Turtle()
    nova.pu()
    nova.setpos(random.randint(-300, 300),
                random.randint(-300, 300))
    nova.pencolor('#{:06x}'.format(random.randrange(256**3)))
    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, random

turtle.bgcolor('black')
turtle.delay(0)
while True:
    n = random.randint(3,8)
    t = []
    for i in range(n):
        nova = turtle.Turtle()
        nova.speed(0)
```

```
        nova.pu()
        nova.setpos(random.randint(-300, 300),
                    random.randint(-300, 300))
        farba = '#{:06x}'.format(random.randrange(256**3))
        nova.pencolor(farba)
        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()
            del tt                        # zruší korytnačku
```

Po dokreslení, obrázok zmaže a začne kresliť nový

KAPITOLA 12

Rekurzia

Naučíme sa nový mechanizmus, ktorý ... - rekurziu. Na úvod je dobré si pripomenúť, 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
- vykonávanie programu sa vráti na návratovú adresu

12.1 Nekonečná rekurzia

Rekurzia v programovaní znamená, že funkcia najčastejšie zavolá samú seba, t.j. že funkcia je definovaný pomocou samej seba. Na prvej ukážke vidíme rekurzívnu funkciu, ktorá nerobí nič iné, len volá samú seba:

```
def xy():
    xy()

xy()
```

```
...
RuntimeError: maximum recursion depth exceeded
```

Takýto program veľmi rýchlo skončí chybovou správou.

To, že funkcia naozaj volala samú seba, môžeme vidieť, keď popri rekurzívnom volaní urobíme nejakú akciu, napr. vypisovanie nejakého počítadla:

```
def xy(p):
    print('volanie xy({})'.format(p))
    xy(p+1)

xy(0)
```

```
volanie xy(0)
volanie xy(1)
volanie xy(2)
volanie xy(3)
volanie xy(4)
...
```

```
volanie xy(977)
volanie xy(978)
volanie xy(979)
Traceback (most recent call last):
...
RuntimeError: maximum recursion depth exceeded while calling a Python object
```

V tomto prípade program opäť spadne, hoci môžeme vidieť, ako sa 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 **'RuntimeError: maximum recursion depth exceeded'**.

My sa budeme v našich programoch snažiť vyvarovať takejto nekonečnej rekurzii. Hoci ešte si ju ukážeme aj pri práci s korytnačkou:

```
def xy(d):
    t.fd(d)
    t.lt(60)
    xy(d+0.3)

import turtle

turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
xy(1)
```

```
...
RuntimeError: maximum recursion depth exceeded while calling a Python object
```

Aj tento program, veľmi rýchlo spadne.

Odkrokujme si to:

- v príkazovom riadku je volanie funkcie `xy` => vytvorí sa lokálna premenná `d`, 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:

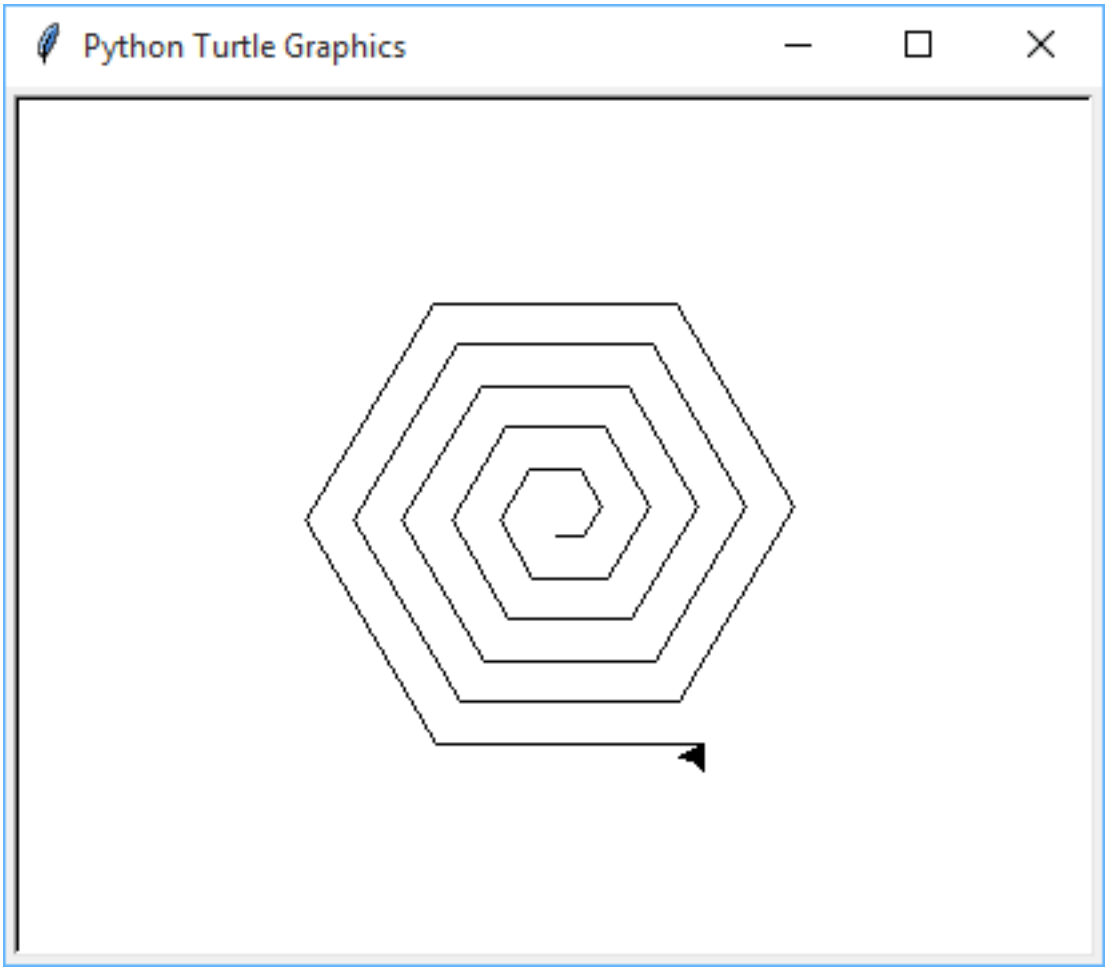
- nové prvky pridávame na vrch (napr. kopa tanierov, resp. na koniec radu čakajúcich)
- keď potrebujeme zo zásobníka nejaký prvok, vždy ho odoberáme z vrchu (posledne položený tanier, resp. posledný v rade čakajúcich)

12.2 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. 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 rekuzívneho 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. Triviálnym prípadom je tu *nič*, t.j. žiadna akcia pre príliš veľké špirály:




```
def spir(d):
    if d > 100:
        pass      # nerob nič
```

```
    else:
        t.fd(d)
        t.lt(60)
        spir(d+3)

import turtle

turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
spir(10)
```

Trasujme volanie so simuláciou na zásobníku s počiatočnou hodnotou parametra `d`, napr. 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);
        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)
```

```
>>> 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,
```

12.3 Pravá rekurzia

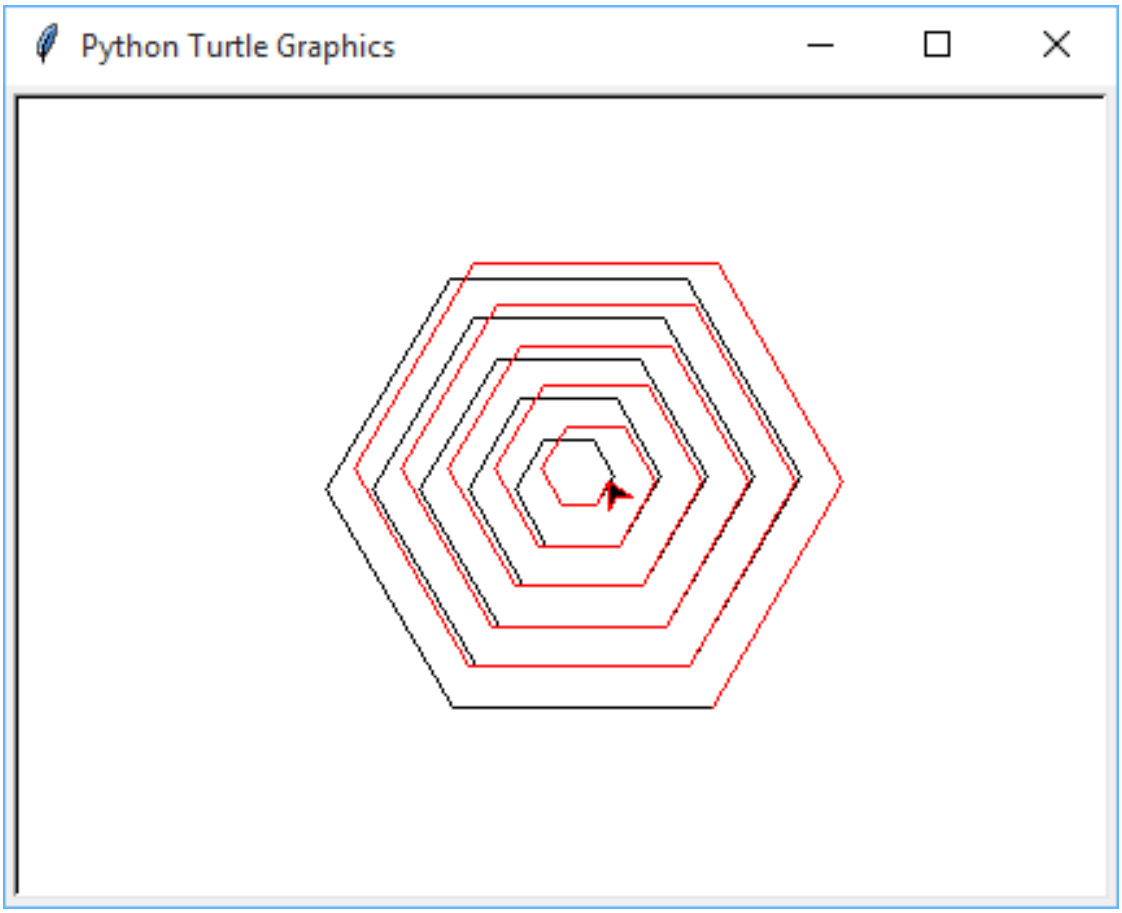
Rekuzie, ktoré už nie sú obyčajné chvostové, sú na pochopenie trochu zložitejšie. Pozrime takéto kreslenie špirály:

```
def spir(d):
    if d > 100:
        t.pencolor('red')      # a skonči
    else:
        t.fd(d)
        t.lt(60);
```

```
        spir(d + 3)
        t.fd(S)
        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)
```

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:
        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 jednoduchšie ako `retazec[::-1]`):

```
def otoc(retazec):
    if len(retazec) <= 1:
        return retazec
    return otoc(retazec[1:]) + retazec[0]

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 raťazce otáčame tak, že z neho najprv odtrhneme prvý znak, otočíme zvyšok reťazca (to je už kratší reťazec) 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 krašia 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 prí 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.

Vieme ale tie koeficienty zobraziť pomocou **Pascalovho trojuholníka**, napr.:

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   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 koeficentov 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 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
```

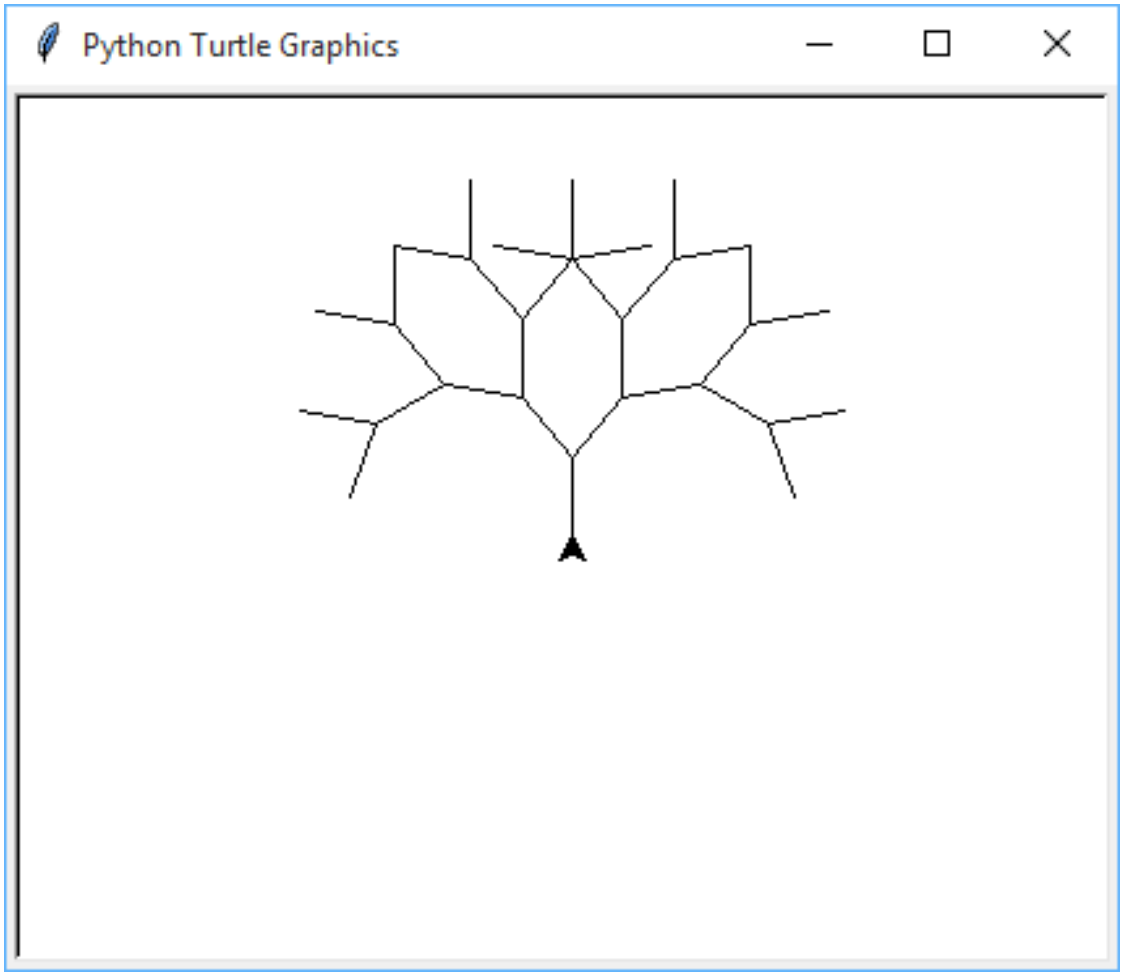
Vidíme, že tento počet volaní veľmi rýchlo rastie a je určite väčší ako samotné fibonnaciho číslo. Preto aj `fib(100)` by trvalo veľmi dlho (vyše **354224848179261915075** volaní funkcie).

12.4 Binárne stromy

Medzi informatikmi sú veľmi populárne binárne stromy. Najlepšie sa kreslia pomocou korytnačky. Hovoríme, že binárne stromy majú nejakú svoju úroveň `n` a sú definované takto:

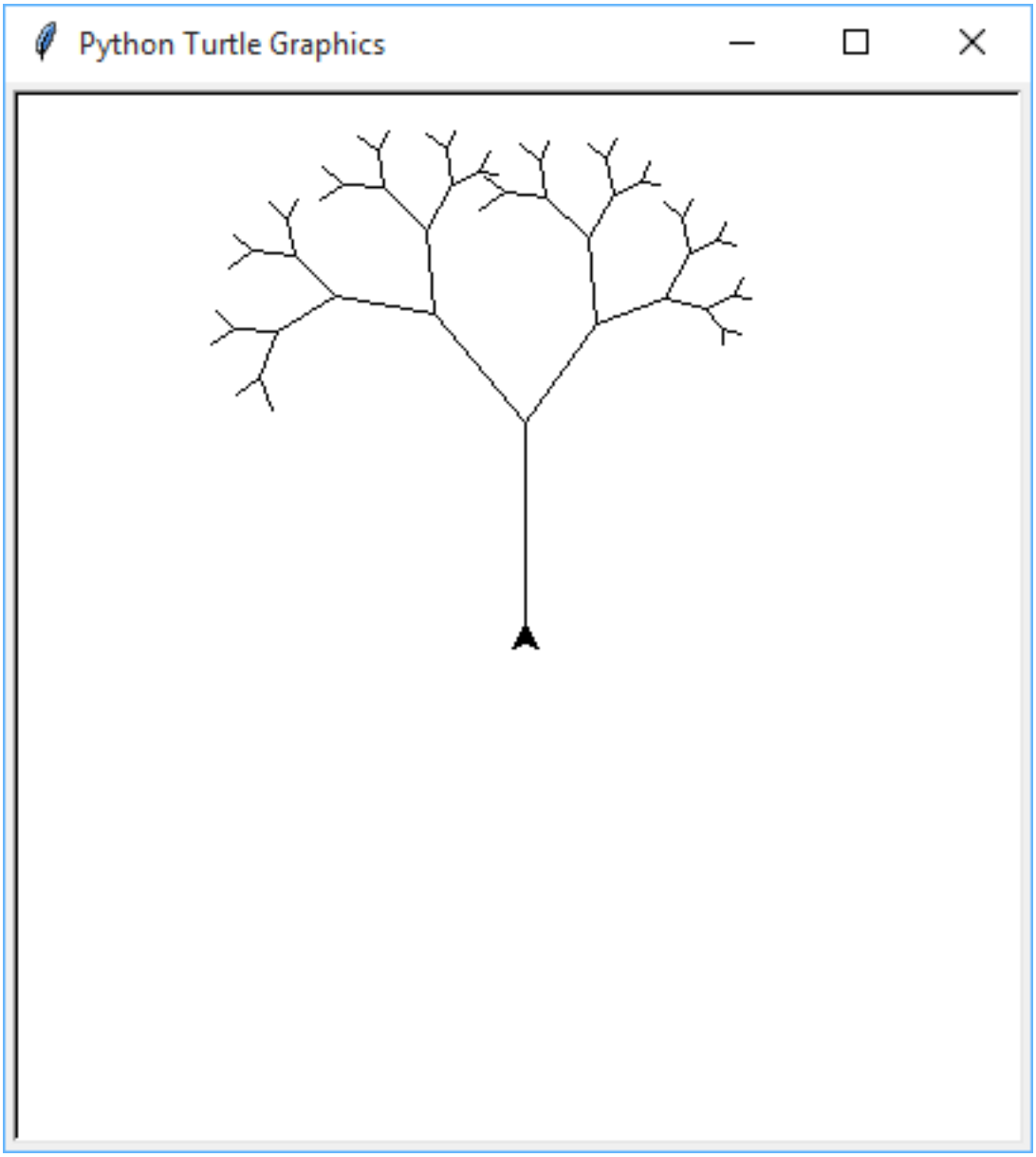
- ak je úroveň stromu `n` = 0, nakreslí sa len čiara nejakej dĺžky
- pre `n` >= 1, sa najprv nakreslí čiara, potom sa na jej konci nakreslí najprv **vľavo** binárny strom `(n-1)`. úrovne a potom **vpravo** opäť binárny strom `(N-1)`. úrovne (hovoríme im podstromy) - po nakreslení týchto podstromov sa 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 zmenšovať

Najprv ukážeme binárny strom, ktorý má vo všetkých úrovniach rovnako veľké podstromy:




```
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 l'avého podstromu
        strom(n-1)    # nakresli l'avý 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)      # vrát' sa na pôvodné miesto

import turtle
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 triviálny prípad:



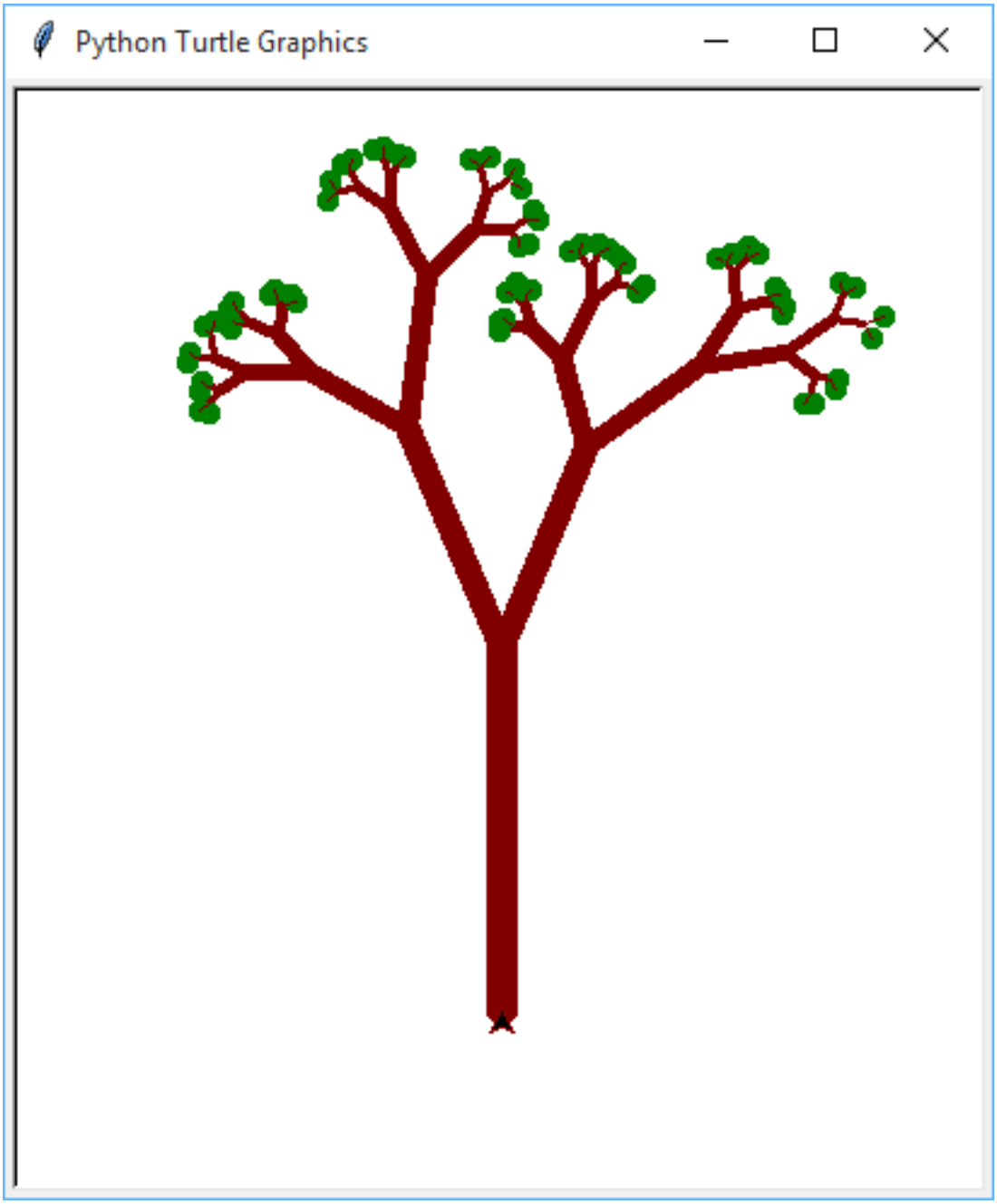

```
def strom(n, d):
    t.fd(d)
    if n > 0:
        t.lt(40)
        strom(n-1, d*.7)
        t.rt(75)
        strom(n-1, d*.6)
        t.lt(35)
    t.bk(d)

import turtle
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:

```
def strom(d):
    t.fd(d)
    if d > 5:
        t.lt(40)
        strom(d*.7)
        t.rt(75)
        strom(d*.6)
        t.lt(35)
    t.bk(d)

import turtle
turtle.delay(0)
t = turtle.Turtle()
t.lt(90)
strom(80)
```

Ak využijeme náhodný generátor, môžeme vytvárať úplne rôzne stromy:

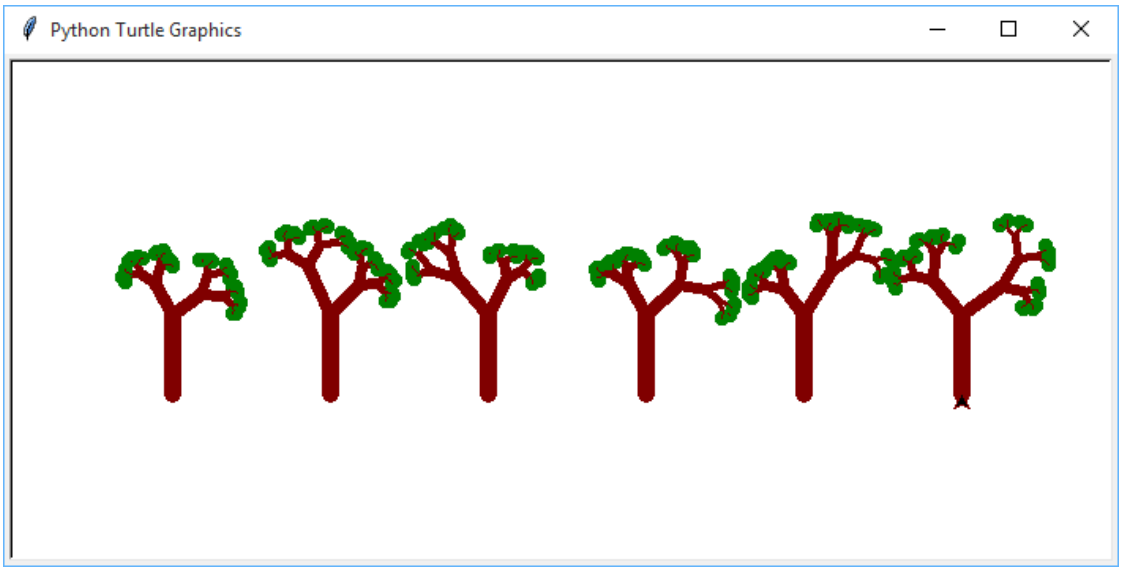



```
def strom(n, d):
    t.pensize(2*n+1)
    t.fd(d)
    if n == 0:
        t.dot(10, 'green')
    else:
        uhol1 = random.randint(20,40)
        uhol2 = random.randint(20,60)
        t.lt(uhol1)
        strom(n-1, d * random.randint(40,70) / 100)
        t.rt(uhol1 + uhol2)
        strom(n-1, d * random.randint(40,70) / 100)
```

```
        t.lt(uhol2)
    t.bk(d)

import turtle, random
turtle.delay(0)
t = turtle.Turtle()
t.lt(90)
t.pencolor('maroon')
strom(6, 150)
```

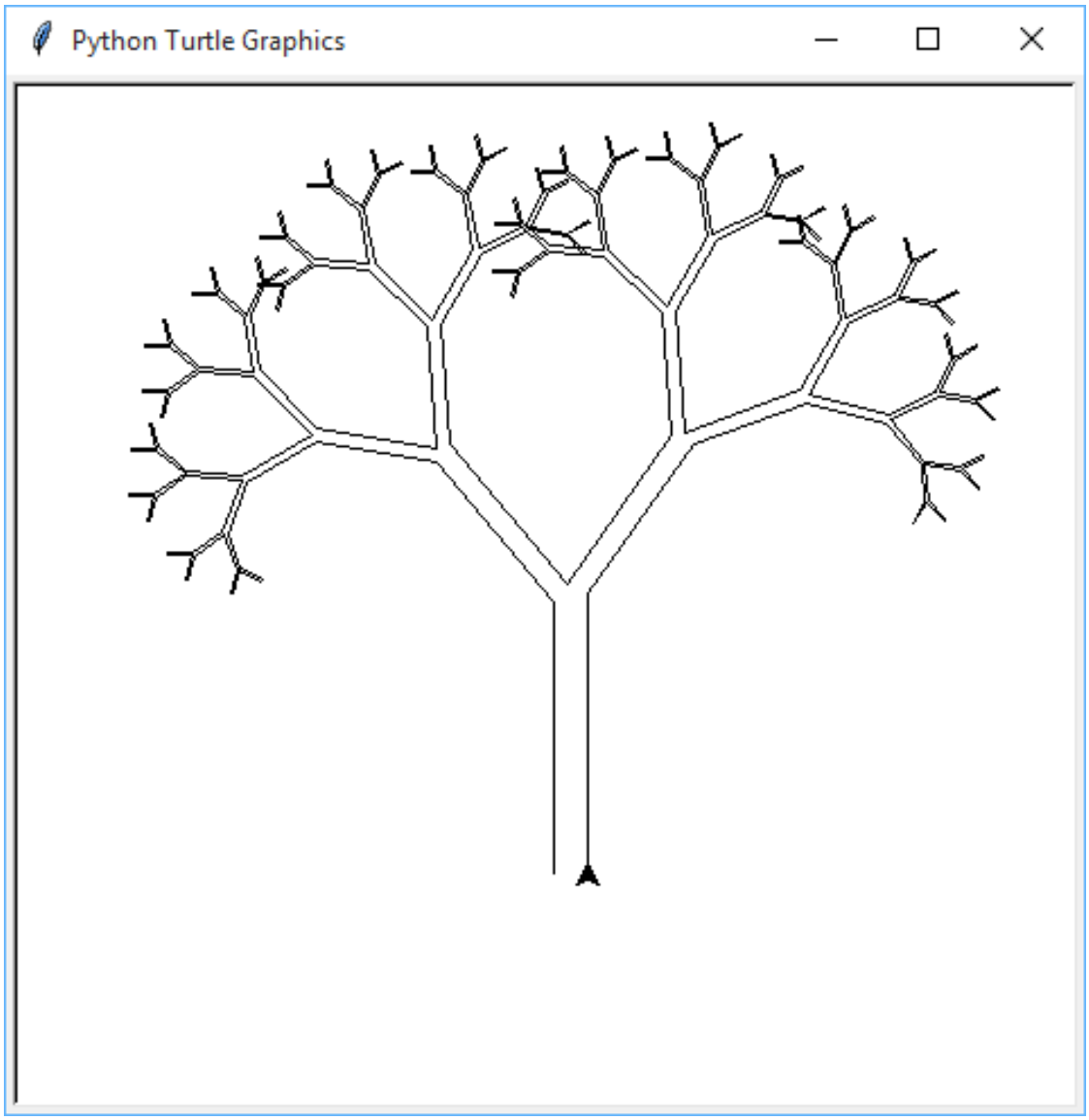
V tomto riešení si všimnite, kde sme zmenili hrúbku pera, aby sa strom kreslil rôzne hrubý v rôznych úrovniach. Tiež sa tu na posledných "konároch" nakreslili zelené listy - pridali sme ich v triviálnom prípade. Využili sme tu korytnačiu metódu `t.dot(vel'kost',farba)`, ktorá na pozícii korytnačky nakreslí bodku danej veľkosti a farby.

Každé spustenie tohto programu nakreslí trochu iný strom. Môžeme vytvoriť celú aleju stromov, v ktorej bude každý strom trochu iný:




```
import turtle
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 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:

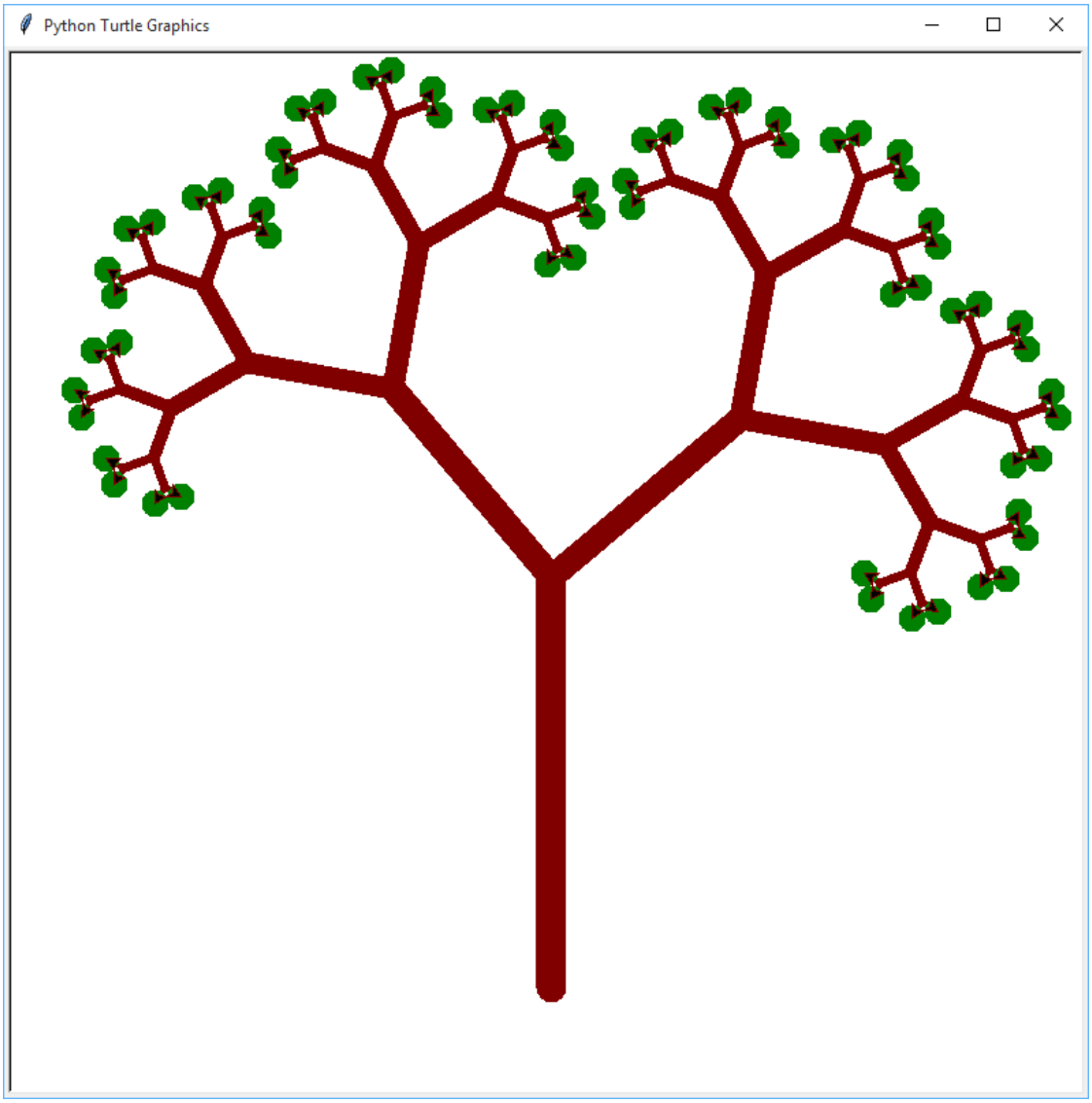



```
def strom(n, d):
    t.fd(d)
    if n == 0:
        t.rt(90)
        t.fd(1)
        t.lt(90)
    else:
        t.lt(40)
        strom(n-1,d*.67)
        t.rt(75)
        strom(n-1,d*.67)
        t.lt(35)
    t.bk(d)

import turtle
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 pole korytnačiek, pričom každá z nich po nakreslení jednu úsečky "narodí" na svojej pozícii ďalšiu korytnačku, pričom ju trochu otočí. Idea algoritmu je takáto:

- prvá korytnačka nakreslí koreň stromu - prvú úsečku dĺžky `d`
- na jeho konci (na momentálnej pozícii tejto korytnačky) sa vyrobí jedna nová korytnačka, snovým relatívnym natočením o 40 stupňov vľavo a sama sa otočí o 50 stupňov vpravo
- dĺžka `d` sa zníži napr. na `d` * 0.6
- všetky korytnačky teraz prejdú v svojom smere dĺžku `d` a opäť sa na ich 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




```
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))

import turtle
#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:

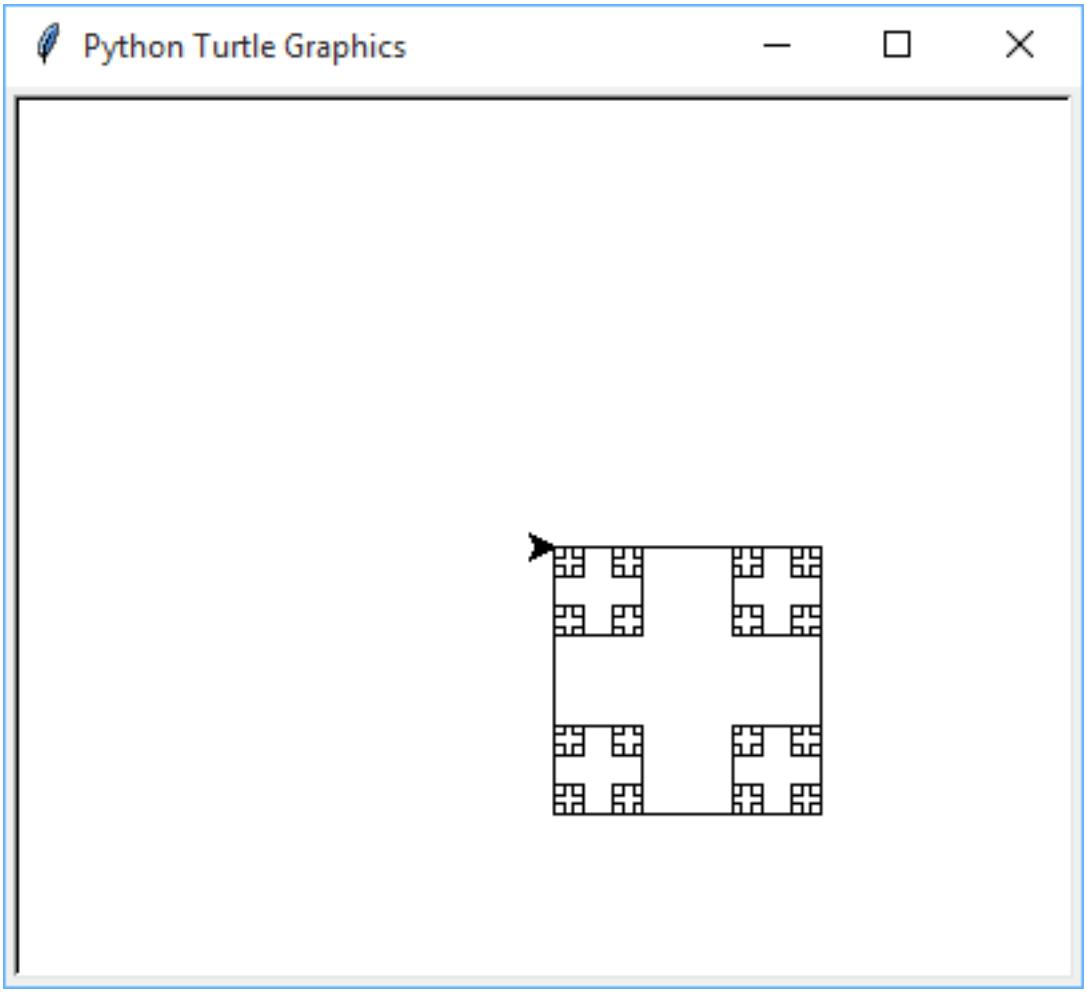
```
pocet korytnaciek = 64
```

12.5 Ďalšie rekurzívne obrázky

Napíšeme funkciu, ktorá nakreslí obrázok `stvorce` úrovne `n`, veľkosti `a` s týmito vlastnosťami:

- pre `n` = 0 nerobí 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

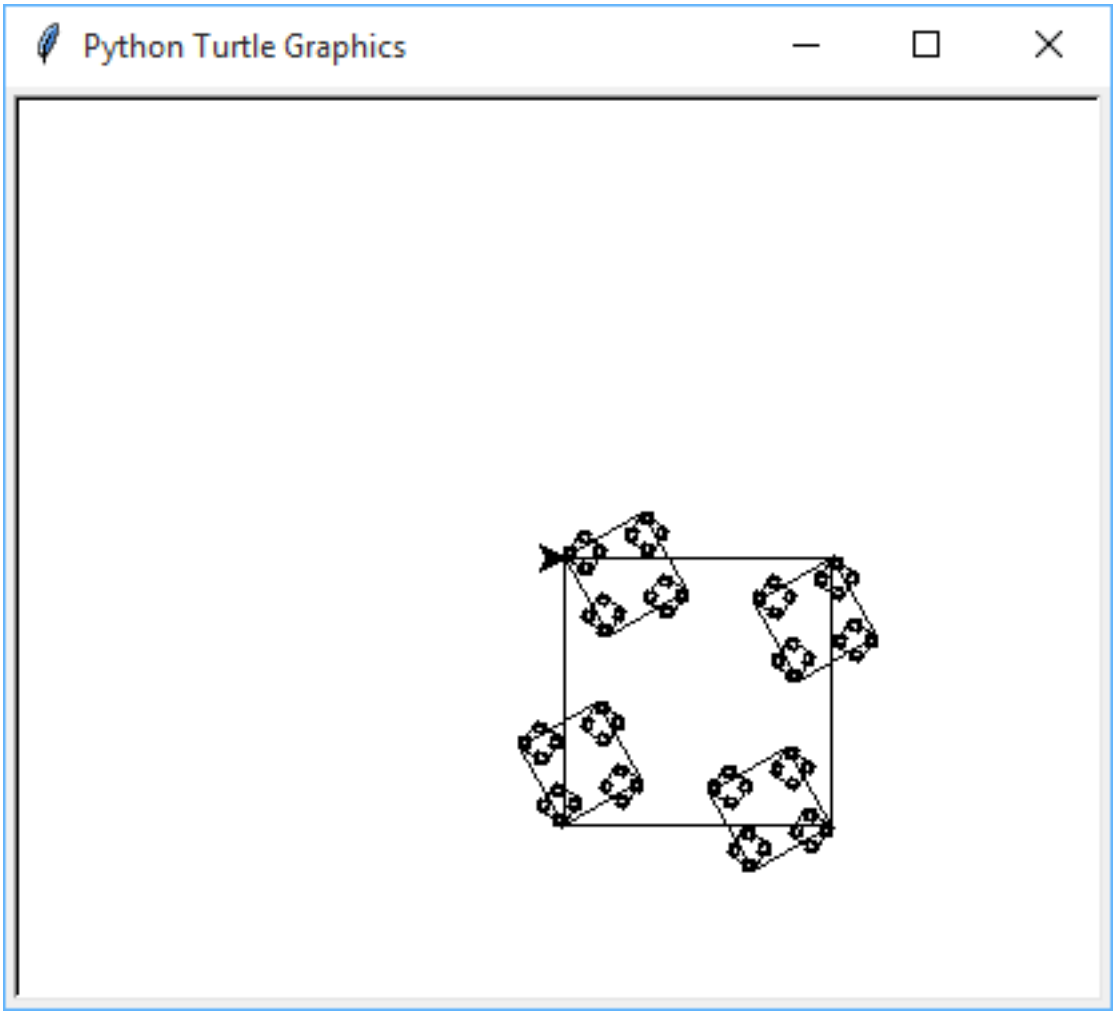
Štvorce v každom rohu štvorca:




```
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)

import turtle
turtle.delay(0)
t = turtle.Turtle()
stvorce(4, 300)
```

Uvedomte si, že to nie je chvostová rekurzia.

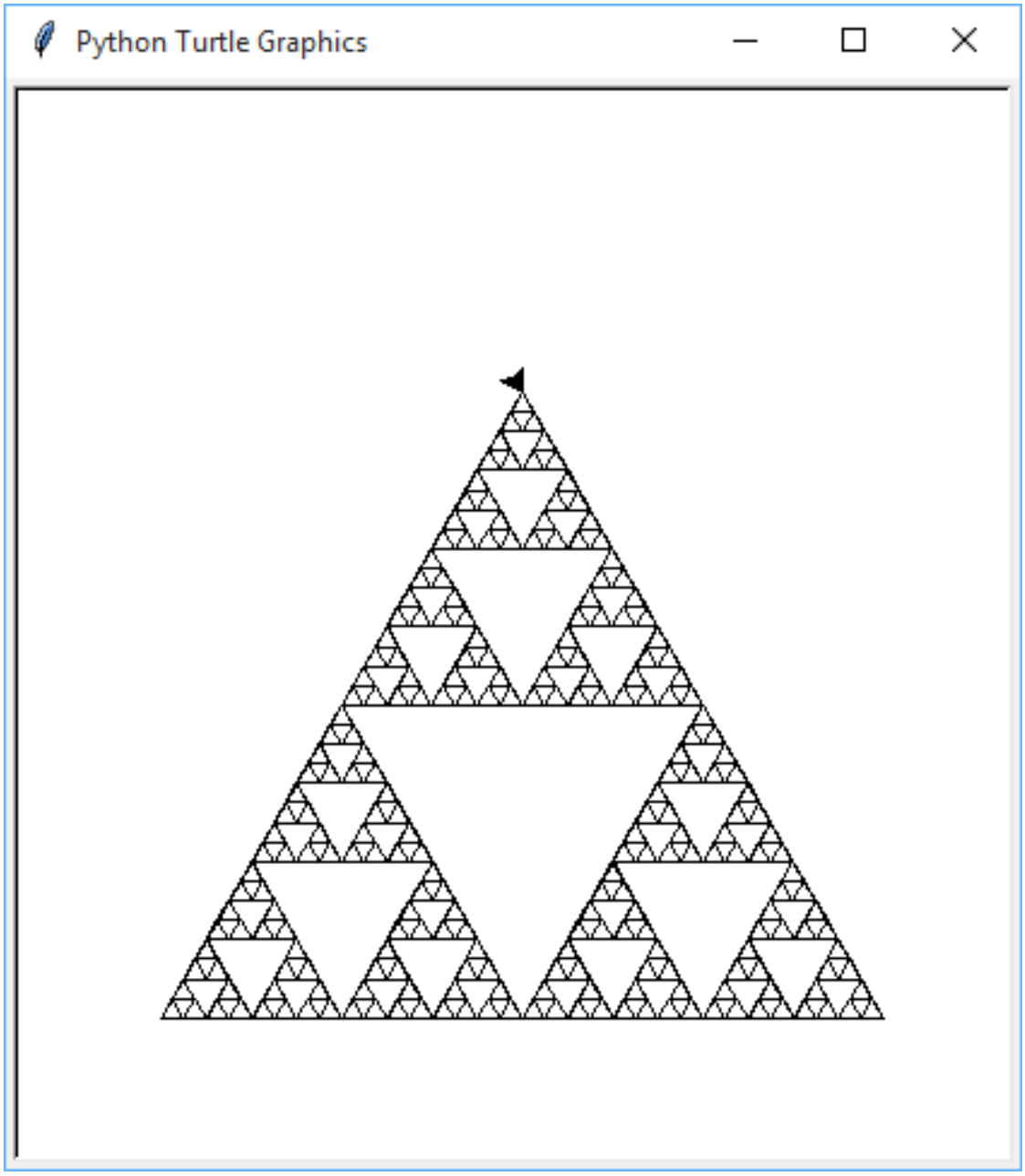
Tesne pred rekurzívnym volaním otočíme korytnačku o 30 stupňov a po návrate z rekurzie týchto 30 stupňov vrátime:




```
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)

import turtle
turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
stvorce(5, 300)
```

Sierpiňského trojuholník

Rekurzívny obrázok na rovnakom princípe ale trojuholníkového tvaru navrhol poľský matematik Sierpiňský (http://en.wikipedia.org/wiki/Sierpinski_triangle) ešte v roku 1915:

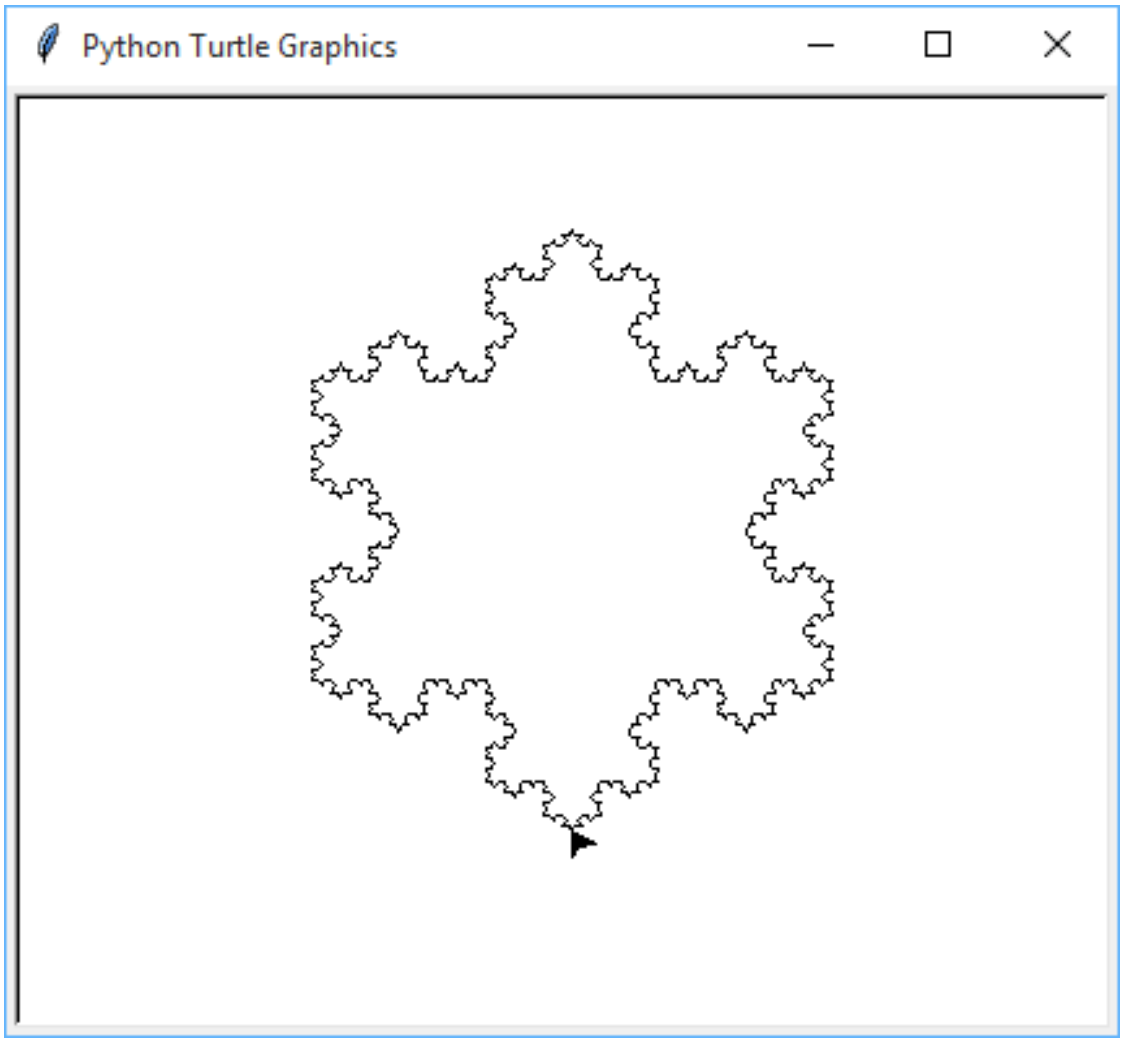



```
def trojuholniky(n, a):
    if n > 0:
        for i in range(3):
            t.fd(a)
            t.rt(120)
            trojuholniky(n-1, a/2)

import turtle
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

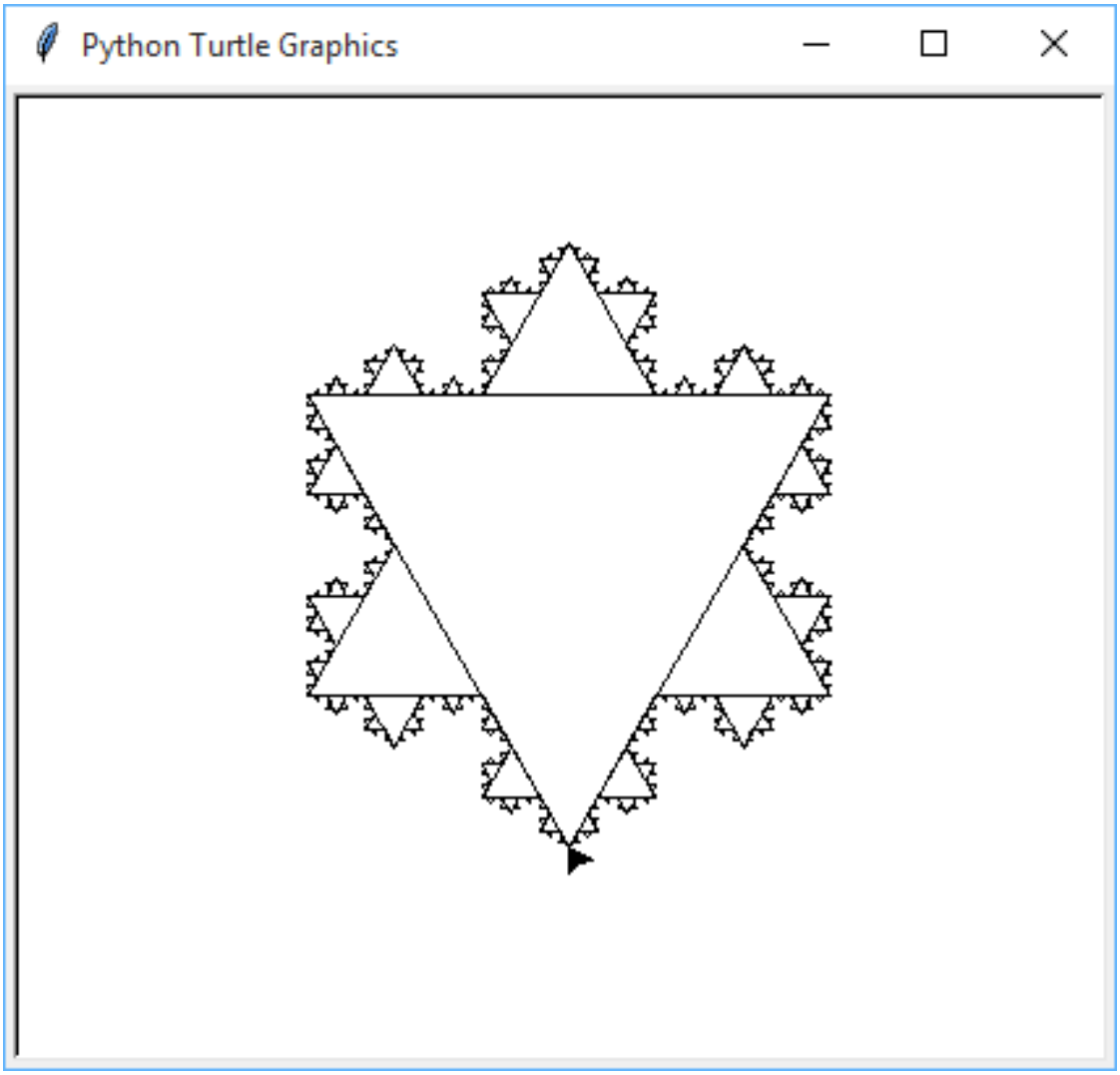
Ďalej ukážeme veľmi známu rekurzívnu krivku - snehovú vločku (známu tiež ako Kochova krivka (http://en.wikipedia.org/wiki/Koch_snowflake)):




```
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)

import turtle
turtle.delay(0)
t = turtle.Turtle()
t.speed(0)
t.lt(120)
sneh_vlocka(4, 300)
```

Ak namiesto jedného volania `vlocka` zapíšeme nakreslenie aj všetkých predchádzajúcich úrovní krivky, dostávame tiež zaujímavú kresbu:



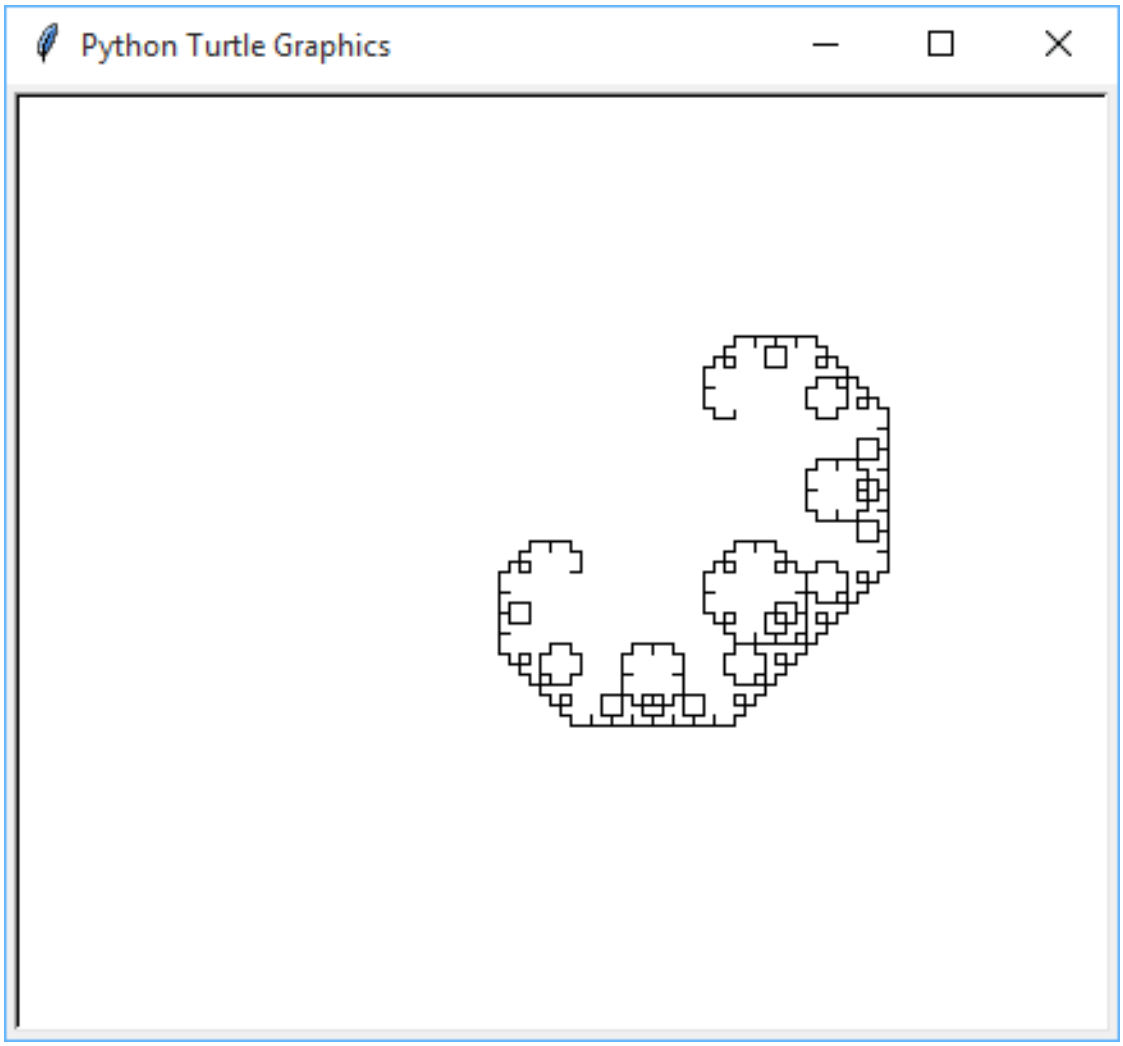

```
import turtle
turtle.delay(0)
t = turtle.Turtle()
#t.speed(0)
t.lt(120)
for i in range(5):
    sneh_vlocka(i, 300)
```

Ďalšie fraktálové krivky

Špeciálnou skupinou rekurzívnych kriviek sú fraktály (pozri aj na wikipédii (http://en.wikipedia.org/wiki/Fractal)). 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 (http://sk.wikipedia.org/wiki/Helge_von_Koch)). Zjednodušene by sme mohli povedať, že fraktál je taká krivka, ktorá sa skladá zo 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 (http://en.wikipedia.org/wiki/L%C3%A9vy_C_curve):

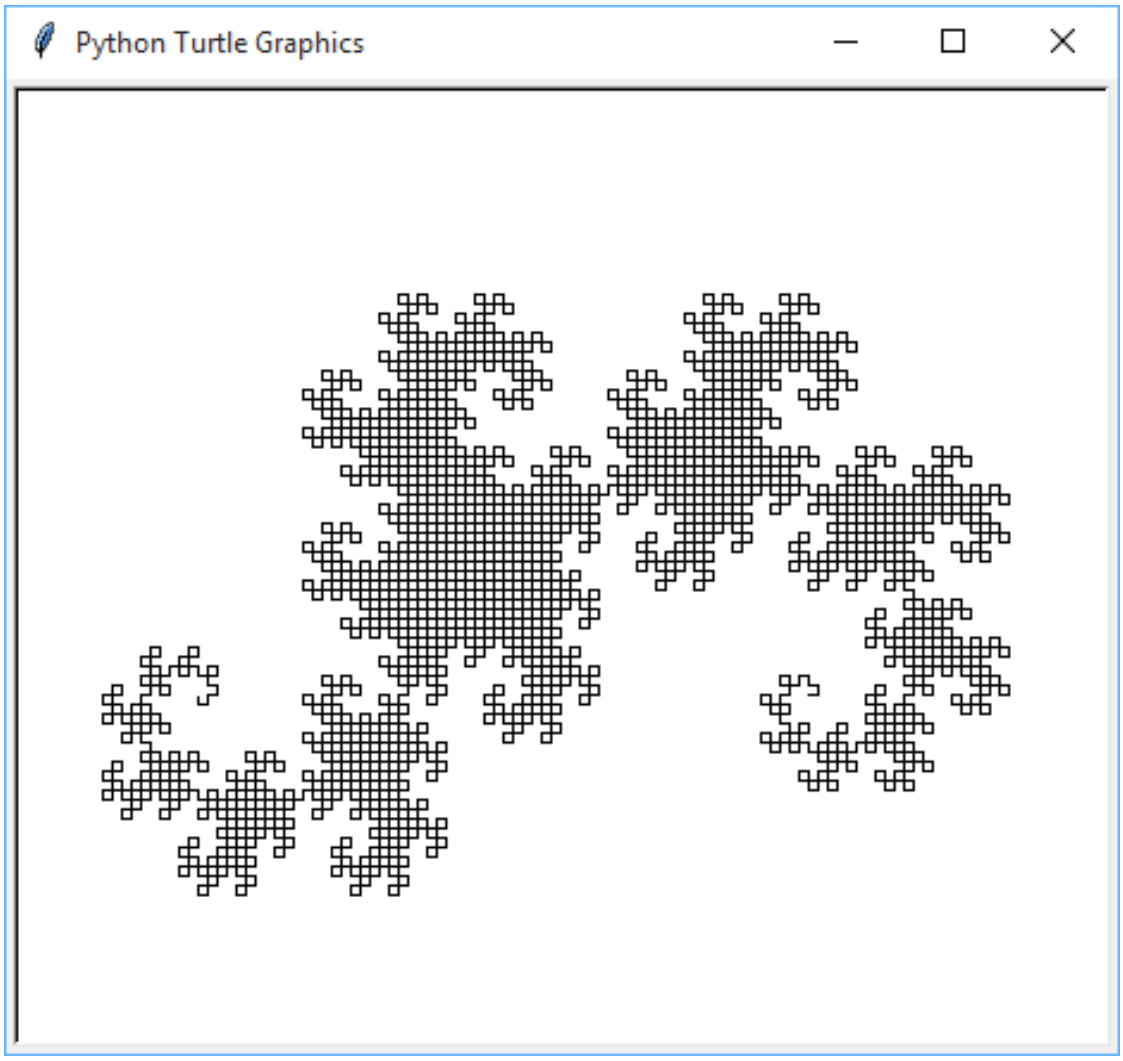



```
def ckrivka(n, s):
    if n == 0:
        t.fd(s)
    else:
        ckrivka(n-1, s)
        t.lt(90)
        ckrivka(n-1, s)
        t.rt(90)

import turtle
turtle.delay(0)
t = turtle.Turtle()
```

```
t.speed(0)
t.ht()
ckrivka(9, 4)            # skúste aj: ckrivka(13, 2)
```

Dračia krivka

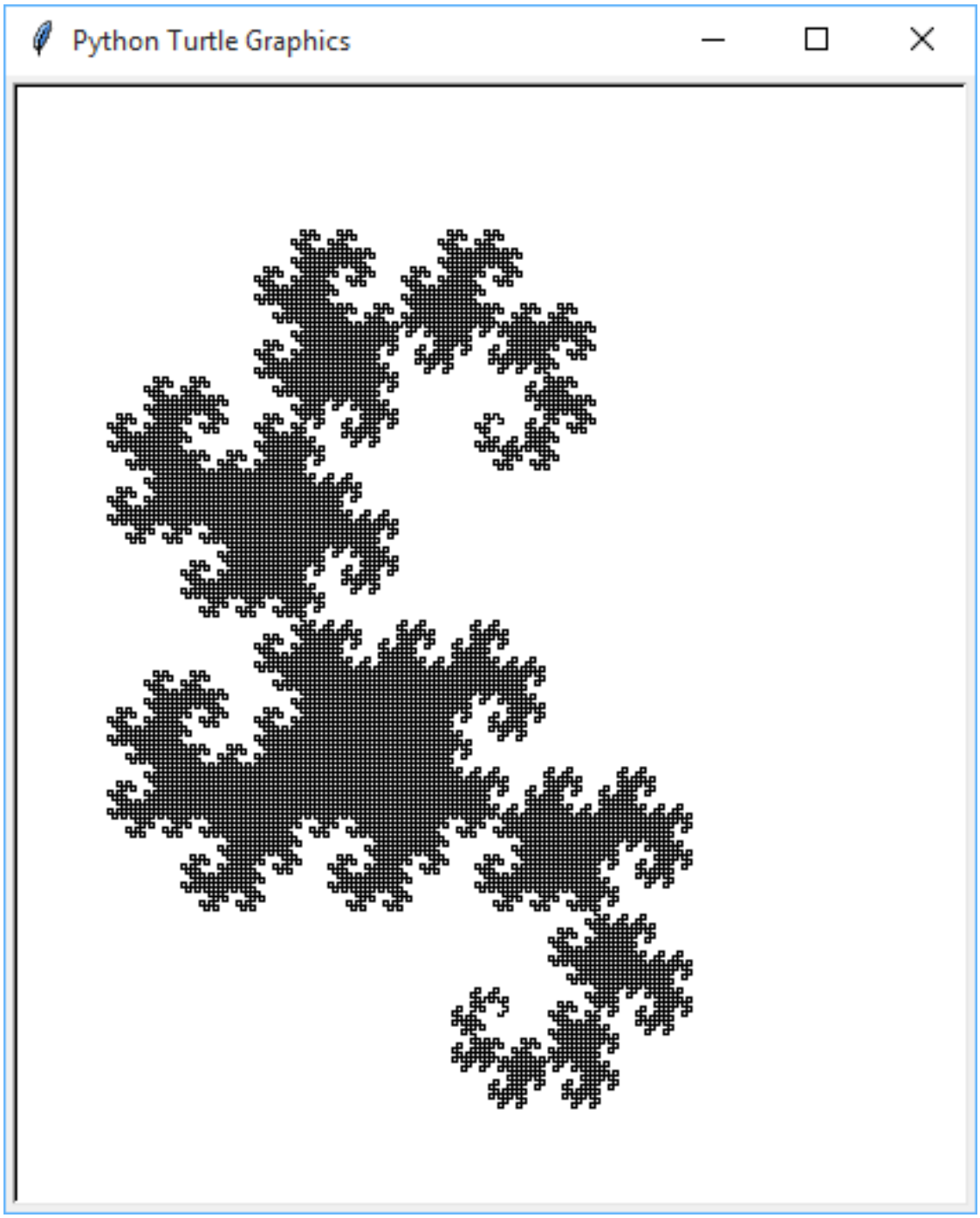
C-krivke sa veľmi podobá Dračia krivka (http://en.wikipedia.org/wiki/Dragon_curve), ktorá sa skladá z dvoch "zrkadlových" funkci: `ldrak` a `pdrak`. Všimnite si zaujímavú vlastnosť týchto dvoch rekurzívnych funkcií: prvá rekurzívne volá samu seba ale aj druhú a druhá volá seba aj prvú. Našťastie Python toto zvláda veľmi dobre:




```
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)

import turtle
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:

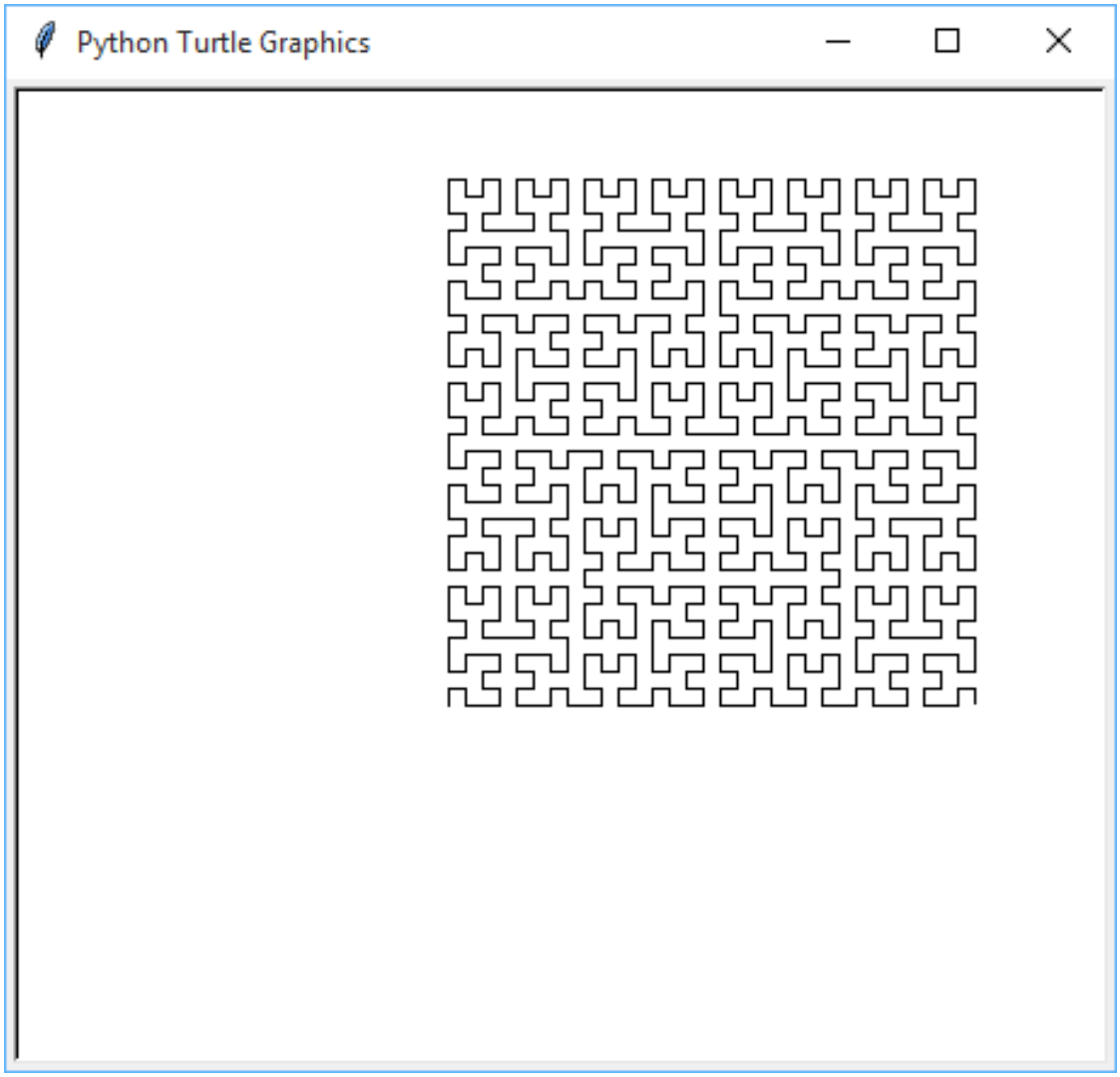



```
def drak(n, s, u=90):
    if n == 0:
        t.fd(s)
    else:
        drak(n-1, s, 90)
        t.lt(u)
        drak(n-1, s, -90)

import turtle
turtle.delay(0)
```

```
t = turtle.Turtle()
t.speed(0)
t.ht()
drak(14, 2)
```

Hilbertova krivka

V literatúre je veľmi známou Hilbertova krivka (http://en.wikipedia.org/wiki/Hilbert_curve), 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):




```
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)

import turtle
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 (http://en.wikipedia.org/wiki/List_of_fractals_by_Hausdorff_dimensior

Dvojrozmerné polia

Jednorozmerné polia (pythonovský typ `list`) slúžia hlavne na uchovávanie nejakej postupnosti alebo skupiny údajov. V takejto štruktúre sa dajú uchovávať aj dvojrozmerné tabuľky, ale bolo by to zbytočne prekomplikované. Dvojrozmerné údaje sa pre nás vyskytujú, napr. v rôznych hracích plochách (štvorčekový papier pre piškvorky, šachovnica pre dosková hry, rôzne typy labyrintov), ale napr. aj rastrové obrázky sú často uchovávané v dvojrozmernej tabuľke. Aj v matematike sa niekedy pracuje s dvojrozmernými tabuľkami čísel (matice).

Už vieme, že prvkami poľa (`list`) môžu byť opäť polia. 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]]
```

Pole `m` má tri prvky: sú to tri riadky (jednorozmerné polia). Našou prvou úlohou bude vypísať takéto pole. 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 dvojrozmerným poľom:

```
def vypis(pole):
    for riadok in pole:
        for prvok in riadok:
            print(prvok, end=' ')
        print()
```

To isté vieme zapísať aj pomocou indexovania:

```
def vypis(pole):
    for i in range(len(pole)):
        for j in range(len(pole[i])):
            print(pole[i][j], end=' ')
        print()
```

Obe tieto funkcie sú veľmi častými šablónami pri práci s dvojrozmernými poľami. Teraz výpis poľa vyzerá takto:

```
>>> vypis(m)
1 2 3
4 5 6
7 8 9
```

Pozrime teraz, ako budeme najčastejšie vytvárať nové dvojrozmerné pole. Najčastejšie to bude pomocou nejakých cyklov. Bude to závisieť od toho, či sa vo výslednom poli niečo opakuje. Vytvorme dvojrozmerné pole veľkosti 3x3, ktoré 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 poľa 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 pre definovaní matice sa zdá, že sa 3-krát opakuje to isté. Zapíšme to pomocou násobenia polí:

```
>>> matica1 = [[0, 0, 0]]*3
```

Žiaľ, toto je veľmi nesprávny spôsob vytvárania dvojrozmerného poľa: zápis `[0,0,0]` označuje **referenciu** na jednorozmerné trojprvkové pole, potom `[[0,0,0]]*3` rozkopíruje túto jednu referenciu trikrát, Teda vytvorili sme pole, ktoré trikrát obsahuje referenciu na ten istý riadok. Presvedčíme sa o tom priradením do niektorých prvkov takéhoto poľa:

```
>>> matica1[0][1] = 9
>>> matica1[1][2] += 1
>>> vypis(matica1)
0 9 1
0 9 1
0 9 1
```

Zapamätajte si! Dvojrozmerné pole nikdy nevytvárame tak, že násobíme jeden riadok viackrát. Pritom

```
>>> matica2 = [[0]*3, [0]*3, [0]*3]
```

je už v poriadku, lebo sme v tomto poli vytvorili tri rôzne riadky.

Niekedy sa na vytvorenie "prázdneho" dvojrozmerného poľa 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 pripraveného poľa priradiť nejaké hodnoty, napr. postupným zvyšovaním nejakého počítadla:

```
def ocisluj(pole):
    poc = 0
    for i in range(len(pole)):
        for j in range(len(pole[i])):
            pole[i][j] = poc
            poc += 1
```

Všimnite si, že táto funkcia vychádza z druhej funkcie (šablóny) pre vypisovanie dvojrozmerného poľa: namiesto výpisu prvku (`print()`) sme do neho niečo priradili. Táto funkcia `ocisluj()` nič nevypisuje ani nevracia žiadnu hodnotu "len" modifikuje obsah poľa, ktorý 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
```

Ukážme niekoľko príkladov práce s dvojrozmerným poľom:

1. zvýšime obsah všetkých prvkov o 1

```
def zvys_o_1(pole):
    for riadok in pole:
        for i in range(len(riadok)):
            riadok[i] += 1
```

   Zrejme musia byť všetky prvky tohto poľa nejaké čísla, inak funkcia spadne 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 samotné pole, ale vytvárajú nové pole, ktorého prvky sú o jedna väčšie:

```
def o_1_viac(pole):
    nove_pole = [None] * len(pole)
    for riadok in pole:
        novy_riadok = [0] * len(riadok)
        for i in range(len(riadok)):
            novy_riadok[i] = riadok[i]+1
        nove_pole.append(novy_riadok)
    return nove_pole
```

3. číslovanie prvkov poľa ale nie po riadkoch (ako to robila funkcia `cisluj()`) ale po stĺpcoch. Predpokladáme, že všetky riadky sú rovnako dlhé:

```
def ocisluj_po_stlpcoch(pole):
    poc = 0
    for j in range(len(pole[0])):
        for i in range(len(pole)):
            pole[i][j] = poc
            poc += 1
```

Všimnite si, že táto funkcia má oproti `ocisluj()` len vymenené 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
```

4. spočítame počet výskytov nejakej hodnoty:

```
def pocet(pole, hodnota):
    vysl = 0
    for riadok in pole:
        for prvok in riadok:
            if prvok == hodnota:
                vysl += 1
    return vysl
```

Využili sme tu prvú verziu funkcie (šablóny) pre výpis dvojrozmerného poľa. Ak si ale pripomenieme, že niečo podobné robí štandardná metóda `count()`, ale táto funguje len pre jednorozmerné polia, môžeme našu funkciu vylepšiť:

```
def pocet(pole, hodnota):
    vysl = 0
    for riadok in pole:
        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 (dvojrozmerné pole) symetrická, t.j. či sú prvky pod a nad hlavnou uhlopriečkou rovnaké, čo znamená, že má platiť `pole[i][j] == pole[j][i]` pre každé `i` a `j`:

```
def index(pole, hodnota):
    vysl = True
    for i in range(len(pole)):
        for j in range(len(pole[i])):
            if pole[i][j] != pole[j][i]:
                vysl = False
    return vysl
```

   Hoci je toto riešenie správne, má niekoľko nedostatkov:

   - funkcia zbytočne testuje každú dvojicu prvkov `pole[i][j]` a `pole[j][i]` dvakrát, napr. `pole[0][2] == pole[2][0]` aj `pole[2][0] == pole[0][2]`, tiež zrejme netreba kontrolovať prvky na hlavnej uhlopriečke, či `pole[i][i] == pole[i][i]`
   - keď sa vo vnútornom cykle zistí, že sme našli dvojicu `pole[i][j]` a `pole[j][i]`, ktoré sú navzájom rôzne, zapamätá sa, že výsledok funkcie bude `False` a ďalej sa pokračuje prehľadávať dvojrozmerné pole - 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 index(pole, hodnota):
    vysl = True
    for i in range(len(pole)):
        for j in range(len(pole[i])):
            if pole[i][j] != pole[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 návratovou hodnotou `False`.

   Prepíšme funkciu tak, aby zbytočne dvakrát nekontrolovala každú dvojicu prvkov a aby sa korektne ukončila, keď nájde nerovnakú dvojicu:

```
def index(pole, hodnota):
    for i in range(1, len(pole)):
        for j in range(i):
            if pole[i][j] != pole[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 poľa, použijeme šablónu s indexmi:

```
def index(pole, hodnota):
    for i in range(len(pole)):
        for j in range(len(pole[i])):
            if pole[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)
(1, 2)
```

```
>>> index(a, 5)
>>>
```

Na tomto poslednom príklade vidíme, že naša funkcia `index()` v nejakom prípade nevrátila "nič". V skutočnosti vrátila špeciálnu hodnotu `None`, ktorá sa v príkazovom režime nevypisuje. 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 hľadaná hodnota v poli nenašla vrátila `None`. Je zvykom takýto výsledok testovať takto:

```
vysledok = index(pole, hodnota)
if vysledok is None:
    print('nenasiel')
else:
    riadok, stlpec = vysledok
```

Teda namiesto `premenna == None` alebo `premenna != None` radšej používame `premenna is None` alebo `premenna is not None`.

13.1 Polia s rôzne dlhými riadkami

Doteraz sme predpokladali, že všetky riadky dvojrozmerného poľa majú rovnakú dĺžku. Sú ale aj situácie, keď dvojrozmerné pole môže mať riadky rôznych dĺžok. 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
```

Toto pole obsahuje prvých niekoľko riadkov Pascalovho trojuholníka. Našťastie funkciu `vypis()` (obe verzie) sme napísali tak, že správne vypíšu aj polia s rôzne dlhými riadkami.

Niektoré polia nemusia mať takto pravidelný tvar, napr.

```
>>> delitele = [[6, 2, 3], [13, 13], [280, 2, 2, 2, 5, 7], [1]]
>>> vypis(delitele)
```

```
6 2 3
13 13
280 2 2 2 5 7
1
```

Pole `delitele` má v každom riadku rozklad nejakého čísla (prvý prvok) na prvočinitele (súčin zvyšných prvkov)

Preto už pri písaní funkcií myslíme na to, že parametrom môže byť aj pole s rôznou dĺžkou riadkov. Zapíšme funkciu, ktorá nám vráti zoznam všetkých dĺžok riadkov daného dvojrozmerného poľa:

```
def dlzky(pole):
    vysl = []
    for riadok in pole:
        vysl.append(len(riadok))
    return vysl
```

Pre naše dva príklady polí dostávame:

```
>>> dlzky(pt)
[1, 2, 3, 4, 5, 6]
>>> dlzky(delitele)
[3, 2, 6, 1]
```

Podobným spôsobom môžeme generovať nové dvojrozmerné pole s rôznou dĺžkou riadkov, o ktorom vieme akurát 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 polia:

```
>>> n = 7
>>> m3 = vyrob([n]*n)
>>> m4 = vyrob(range(n))
>>> m5 = vyrob(range(n,0,-2))
```

13.2 Pole farieb

Ukážme malú aplikáciu, v ktorej vytvoríme dvojrozmerné pole náhodných farieb, potom toto pole vykreslíme do grafickej plochy ako pole malých farebných štvorčekov - vznikne farebná mozaika a na záver to otestujeme klikaním myšou.

Prvá verzia programu vygeneruje dvojrozmerné pole náhodných farieb, vykreslí ho a uloží do textového súboru:

```
import tkinter, random

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

pole = []
for i in range(20):
    p = []
    for j in range(30):
        p.append('#{:06x}'.format(random.randrange(256**3)))
    pole.append(p)

d, x0, y0 = 10, 30, 10
for i in range(len(pole)):
    for j in range(len(pole[i])):
        x, y = d*j+x0, d*i+y0
        canvas.create_rectangle(x, y, x+d, y+d, fill=pole[i][j], outline='')

with open('obr.txt', 'w') as subor:
    for riadok in pole:
        print(' '.join(riadok), file=subor)
```

V druhej verzii programu už nebudeme generovať dvojrozmerné pole, 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 pomocné dvojrozmerné pole `re` (rovnakých rozmerov ako pole farieb). Na záver doplníme funkciu na zabezpečenie klikania: kliknutý štvorček sa zafarbí, napr. na bielo:

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

pole = []
with open('obr.txt') as subor:
    for riadok in subor:
        pole.append(riadok.split())

# inicializuj pomocné pole re[][]
re = []
for i in range(len(pole)):
    re.append([0]*len(pole[i]))
# vykresli a id. cisla uloz do pola re[][]
d, x0, y0 = 10, 30, 10
for i in range(len(pole)):
    for j in range(len(pole[i])):
        x, y = d*j+x0, d*i+y0
        re[i][j] = canvas.create_rectangle(x, y, x+d, y+d, fill=pole[i][j],␣
↪outline='')

def klik(e):
    stlpec, riadok = (e.x-x0)//d, (e.y-y0)//d
    if 0<=riadok<len(pole) and 0<=stlpec<len(pole[riadok]):
        canvas.itemconfig(re[riadok][stlpec], fill='white')
        #pole[riadok][stlpec] = 'white'

canvas.bind('<Button-1>', klik)
```

Všimnite si, ako sme počítali pozíciu kliknutého štvorčeka.

13.3 Hra LIFE

Informácie k tejto informatickej simulačnej hre nájdete na wikipedii (http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)

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 napr. **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 dvojrozmernom poli veľkosti `n` x `n`. V tomto poli je momentálna generácia bunkových živočíchov. Na to, aby sme vyrobili novú generáciu, si pripravíme pomocné pole rovnakej veľkosti a do tohto budeme postupne zapisovať bunky novej generácie. Keď už bude celé toto pomocné pole hotové, prekopírujeme ho do pôvodného poľa. Dvojrozmerné pole budeme vykresľovať do grafickej plochy.

```
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():
    pole = []
    for i in range(n):
        p = []
        for j in range(n):
            p.append(random.randrange(2))
        pole.append(p)
    #pole[5][5] = pole[5][6] = pole[5][7] = pole[4][7] = pole[3][6] = 1
    return pole

def kresli():
    for i in range(n):
        for j in range(n):
            canvas.itemconfig(re[i][j], fill=['white','black'][pole[i][j]])

def pocet_susedov(rr, ss):
    pocet = -pole[rr][ss]
    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 += pole[r][s]
    return pocet

def nova():
    pole1 = []
    for i in range(n):
        pole1.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 pole[i][j]:
                pole1[i][j] = 1

    for i in range(n):
        pole[i] = pole1[i]

    kresli()

def rob(kolko=100, ms=100):
    for i in range(kolko):
        nova()
        canvas.update()
        canvas.after(ms)

import tkinter, random

canvas = tkinter.Canvas(width=600, height=500)
canvas.pack()

n = 50
re = inicializuj_siet()
pole = nahodne()
kresli()
rob()
```

KAPITOLA 14

Asociatívne polia (dict)

Asociatívne pole je taká dátová štruktúra, v ktorej k prvkom neprichádzame cez poradové číslo (index) tak ako pri obyčajných poliach 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
```

Samotné asociatívne pole zapisujeme ako takéto dvojice (`kľúč : hodnota`) a celé je to uzavreté v `'{'` a `'}'` zátvorkách, napr.

```
>>> vek = {'Jan':17, 'Maria':15, 'Ema':18}
```

Takto sme vytvorili asociatívne pole (pythonovský typ `dict`), ktoré má tri prvky: `'Jan'` má 17 rokov, `'Maria'` má 15 a `'Ema'` má 18. V priamom režime vidíme, ako toto asociatívne pole 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 pythonovského poľa:

```
>>> 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 toto asociatívne pole nemá definovaný kľúč `'Juraj'`.

Rovnako, ako priraďujeme hodnotu do obyčajného poľa, 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` je v samotnom asociatívnom poli v nejakom neznámom poradí, dokonca, ak spustíme program viackrát, môžeme dostať rôzne poradia prvkov.

Pripomeňme si, ako funguje operácia `in` s pythonoským poľom (typ `list`):

```
>>> pole = [17, 15, 16, 18]
>>> 15 in pole
True
```

Operácia `in` s takýmto poľom prehľadáva hodnoty a ak takú nájde, vráti `True`.

S asociatívnym poľom 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` s asociatívnom poli prehľadáva kľúče, tak aj for-cyklus bude fungovať na rovnakom princípe:

```
>>> for i in pole:
        print(i)
17
15
16
18
>>> for i in vek:
        print(i)
Jan
Hana
Maria
Ema
```

Z tohto istého dôvodu funkcia `list()` s parametrom asociatívne pole vytvorí pole kľúčov a nie pole hodnôt:

```
>>> list(vek)
['Jan', 'Hana', 'Maria', 'Ema']
```

Keď chceme vypísať z asociatívneho poľa všetky kľúče aj s ich hodnotami, zapíšeme:

```
>>> for kluc in vek:
        print(kluc, vek[kluc])

Jan 17
Hana 15
```

```
Maria 16
Ema 18
```

Zrejme asociatívne pole rovnako ako obyčajné 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 o1starsi(vek):
    for kluc in vek:
        vek[kluc] = vek[kluc] + 1
```

Funkcia zvýši hodnotu v každej dvojici asociatívneho poľa o 1:

```
>>> o1starsi(vek)
>>> vek
{'Jan': 19, 'Hana': 16, 'Maria': 17, 'Ema': 19}
```

Pythonovské funkcie teda môžu meniť (ako vedľajší účinok) nielen obyčajné pole ale aj asociatívne.

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
```

Asociatívne polia majú definovaných viac zaujímavých metód, my si ukážeme len 3 z nich. Táto skupina troch metód vrátia nejaké špeciálne "postupnosti":

- `keys()` - postupnosť kľúčov
- `values()` - postupnosť hodnôt
- `items()` - postupnosť prvkov ako dvojice 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
```

Najpoužívanejšou metódou 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:

```
apole.get(kluc)
apole.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 s obyčajným poľom, ale aj s asociatívnym:

```
>>> pole = [17, 15, 16, 18]
>>> del pole[1]
>>> pole
[17, 16, 18]
```

príkaz `del` s asociatívnym poľom

Príkaz `del` vyhodí z asociatívneho poľa príslušný kľúč aj s jeho hodnotou:

```
del apole[kluc]
```

Ak daný kľúčv poli 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 asociatívne pole `apole`:

`apole = {}` prázdne asociatívne pole

`apole = {kľúč:hodnota,...}` priame priradenie celého asociatívneho poľa

`kľúč in apole` zistí, či v asociatívnom poli existuje daný `kľúč` (vráti `True` alebo `False`)

`len(apole)` zistí počet prvkov (dvojíc `kľúč:hodnota`) v asociatívnom poli

`apole[kľúč]` vráti príslušnú hodnotu alebo príkaz spadne na chybe `KeyError`, ak neexistuje

`apole[kľúč] = hodnota` vytvorí novú asociáciu `kľúč:hodnota`

`del apole[kľúč]` zruší dvojicu `kľúč:hodnota` alebo príkaz spadne na chybe `KeyError`, ak neexistuje

`for kľúč in apole: ...` prechádzanie všetkých kľúčov

`for kľúč in apole.values(): ...` prechádzanie všetkých hodnôt

`for kľúč,hodnota in apole.items(): ...` prechádzanie všetkých dvojíc `kľúč:hodnota`

`apole.get(kľúč)` vráti príslušnú hodnotu alebo `None`, ak kľúč neexistuje

`apole.get(kľúč,náhradná)` vráti príslušnú hodnotu alebo vráti hodnotu parametra `náhradná`, ak kľúč neexistuje

---

Predstavte si teraz, že máme dané nejaké obyčajné pole dvojíc a chceme z neho urobiť asociatívne pole. Môžeme to urobiť, napr. for-cyklom:

```
>>> pole_dvojic = [('one', 1), ('two', 2), ('three', 3)]
>>> apole= {}
>>> for k, h in pole_dvojic:
        apole[k] = h
>>> apole
{'one': 1, 'two': 2, 'three': 3}
```

alebo priamo použitím štandardnej funkcie `dict()`, ktorá takto funguje ako konverzná funkcia:

```
>>> apole = dict(pole_dvojic)
>>> apole
{'one': 1, 'two': 2, 'three': 3}
```

Takéto pole dvojíc vieme vytvoriť aj z nejakého asociatívneho poľa pomocou metódy `items()`:

```
>>> list(apole.items())
[('one', 1), ('two', 2), ('three', 3)]
```

14.1 Asociatívne pole ako slovník (dictionary)

Anglický názov tohto typu `dict` je zo slova **dictionary**, teda slovník. Naozaj je "obyčajný" slovník príkladom pekného využitia asociatívneho poľa. 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, vedeli by sme takto veľmi jednoducho realizovať tento "kuchársky" preklad. V skutočnosti by sme asi mali mať slovník, v ktorom jednému anglickému slovu zodpovedá niekoľko (napr. n-tica) slovenských slov.

14.2 Asociatívne pole ako frekvenčná tabuľka

Frekvenčnou tabuľkou nazývame takú tabuľku, ktorá pre rôzne hodnoty obsahuje informácie o počte výskytov v nejakom zozname hodnôt (napr. súbor, reťazec, pole, ...).

Ukážeme to na zisťovaní počtu výskytov písmen v nejakom texte. Budeme konštruovať asociatívne pole, v ktorom každému písmenu (kľúču) bude zodpovedať jedno celé číslo - počítadlo výskytov tohto písmena:

```
def pocty_znakov(text):
    vysl = {}
    for znak in text.lower():
        if 'a' <= znak <= 'z':
            vysl[znak] = vysl.get(znak, 0) + 1
    return vysl

pocet = pocty_znakov('Anicka dusicka nekasli,/naby 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 príslušné písmeno v slovníku už nachádza. Zápis `vysl.get(znak,0)` pre spracovávaný `znak` vráti momentálny počet jeho výskytov, alebo 0, ak sa toto písmeno vyskytlo prvýkrát.

Podobne môžeme spracovať aj nejaký celý textový súbor:

```
with open('dobs.txt') as subor:
    pocet = pocty_znakov(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 písmen bolo:

```
>>> sum(pocet.values())
```

V ďalšom príklade budeme zisťovať počty výskytov nejakého poľa náhodných čísel:

```
def pocty_cisel(pole):
    vysl = {}
    for cislo in pole:
        vysl[cislo] = vysl.get(cislo, 0) + 1
    return vysl

import random

pole = []
for i in range(100000):
    pole.append(random.randrange(1000))

pocet = pocty_cisel(pole)
```

Všimnite si, že v tomto príklade sú kľúčmi nie znakové reťazce, ale celé čísla. Vo všeobecnosti platí, že kľúčmi môže byť ľubovoľný nemeniteľný (immutable) typ, teda `str`, `int`, `float`, `tuple`.

Ak chceme zistiť, ktoré čísla boli najčastejšie, alebo najmenej často, tak môžeme zapísať:

```
minh = min(pocet.values())
print('min', minh, 'pre cisla:', end=' ')
for k,h in pocet.items():
    if h == minh:
        print(k, end=' ')
print()
maxh = max(pocet.values())
print('max', maxh, 'pre cisla:', end=' ')
for k,h in pocet.items():
    if h == maxh:
        print(k, end=' ')
print()
```

V ďalšom príklade upravíme túto ideu pre zisťovanie počtu výskytov celých slov, napr.:

```
pocet = {}
with open('dobs.txt') as subor:
    for slovo in subor.read().split():
        pocet[slovo] = pocet.get(slovo, 0) + 1
```

Potom 10 najčastejších slov vo veľkom texte zistíme tak, že najprv vytvoríme pole dvojíc `(hodnota,kľúč)` a potom toto pole 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 metóde `sort` (pre obyčajné pole) sme pridali parameter `reverse=True`, aby sa pole utriedilo zostupne, teda od najväčších po najmenšie:

```
pole = []
for k, h in pocet.items():
    pole.append((h, k))

pole.sort(reverse=True)

print(pole[:10])
```

14.3 Pole asociatívnych polí

Asociatívne pole môže byť aj hodnotou v inom asociatívnom poli. 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-prvkové obyčajné pole, v ktorom každý prvok je asociatívne pole. V týchto asociatívnych poliach sú po 3 kľúče `'meno'`, `'adresa'`, `'narodeny'`, pričom dva z nich majú hodnoty opäť asociatívne polia. Možeme 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 poli ale aj ich miesto narodenia.

14.4 Textové súbory JSON

Ak pythonovská štruktúra obsahuje iba:

- obyčajné polia `list`
- asociatívne polia `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. pole `skola`) zapísať do súboru:

```
import json

with open('subor.txt', 'w') as f:
    json.dump(skola, f)
```

Takto vytvorený súbor je dosť nečitateľný, čo môžeme zmeniť parametrom `indent`:

```
with open('subor.txt', 'w') as f:
    json.dump(skola, f, indent=2)
```

Prečítať takýto súbor a zrekonštruovať z neho celú štruktúru je potom veľmi jednoduché:

```
j = json.load(open('subor.txt'))
print(skola==j)
```

Vytvorila sa nová štruktúra `j`, ktorá má presne rovnaký obsah, ako zapísané pole `skola`.

KAPITOLA 15

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(pole)`, `min(ntica)`
  - metódy: `'11 7 234'.split()`, `pole.append('novy')`, `g.create_line(1,2,3,4)`, `t.fd(100)`
- funkcia `type(hodnota)` vráti typ hodnoty

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 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'>
>>> pole = list()
>>> type(pole)
<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`.

15.1 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:

```
>>> 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 vypíše ich:

```
>>> 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
```

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. polia sú **mutable**, lebo niektoré operácie zmenia obsah poľa ale nie referenciu na objekt (`pole.append('abc')` pridá do poľa nový prvok)
- ak dve premenné referencujú ten istý objekt (napr. priradili sme `pole2 = pole`), 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

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.

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, ktoré 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 tú 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.

15.2 Metódy

Všetky doteraz vytvárané funkcie dostávali ako jeden z parametrov objekt typu `Student` 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)

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 `for` 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:

```
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. 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
>>> 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 metód sme sa už stretli skôr, ale asi to bola pre nás doteraz veľká záhada, napr.

```
>>> pole = [2, 5, 7]
>>> pole.append(11)
>>> print(pole.pop(0))
2
```

znamená:

```
>>> pole = [2, 5, 7]
>>> list.append(pole, 11)
>>> print(list.pop(pole, 0))
2
```

Teda `append()` je metóda triedy `list` (pythonovské pole), ktorá má dva parametre: `self` (samotné pole, ktoré sa bude modifikovať) a `hodnota`, ktorá sa bude do poľa 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):
        '''L.append(object) -> None -- append object to end'''
        ...
```

Magické metódy

Do novo vytvárenej triedy môžeme pridávať ľubovoľné množstvo metód (súkromných funkcií), pričom majú jediné obmedzenie: prvý parameter by mal mať meno `self`. 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 `__repr__()`.

metóda `__repr__()`

Je magická metóda, ktorá slúži na vyjadrenie znakovej **reprezentácie** daného objektu. Má tvar:

```
def __repr__(self):
    ...
```

Funkcia by mala vrátiť nejaký znakový reťazec.

Python má štandardnú funkciu `repr()`, ktorá z ľubovoľnej hodnoty vyrobí reťazec. Túto funkciu zavolá Python napr. vždy vtedy, keď potrebuje v príkazovom režime vypísať nejakú hodnotu. Napr.

```
>>> 7 * 11 * 13
1001
>>> ['a'] * 3
['a', 'a', 'a']
```

```
>>> 'a\n' * 3 + 'b'
'a\na\na\nb'
>>> fero
<__main__.Student object at 0x00000000039B3F28>
```

Takže `repr()` z tých typov, ktoré Python už pozná, vyrobí reťazec, ktorý sa dá vypísať. Tie typy, ktoré Python nevie, ako ich treba vypísať, vypíše informáciu, čo je to za objekt a kde sa v pamäti nachádza (napr. aj pre inštancie triedy `Student`). Magická metóda `__repr__()` Pythonu vysvetlí, ako má vypísať samotnú inštanciu: keď zavoláme `repr(fero)` (alebo to zavolá Python v príkazovom režime), Python sa pozrie do príslušného typu, či sa tam nachádza metóda `__repr__()`. ak áno tak ju použije, inak použije náhradný reťazec:

```
>>> repr(fero)
'<__main__.Student object at 0x00000000039B3F28>'
>>> fero.__repr__()
'<__main__.Student object at 0x00000000039B3F28>'
```

Takže zadefinujme vlastnú metódu:

```
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)

    def __repr__(self):
        return 'Student: {} {}'.format(self.meno, self.priezvisko)
```

```
>>> fero = urob('Ferdinand','Fyzik')
>>> fero
Student: Ferdinand Fyzik
>>> print(fero)
Student: Ferdinand Fyzik
```

Uvedomte si, že Python tu v oboch prípadoch zavolal našu magickú metódu `__repr__()`:

```
>>> Student.__repr__(fero)
'Student: Ferdinand Fyzik'
>>> fero.__repr__()
'Student: Ferdinand Fyzik'
```

Ďalšou metódou je `__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 mať (ale nemusí) 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.

Napr. keď zapíšeme `instancia = trieda(parametre)`, tak sa postupne:

1. vytvorí sa nový objekt typu `trieda` - zatiaľ je to prázdny kontajner, teda vytvorí sa referencia `objekt`
2. ak existuje metóda `__init__()`, zavolá ju s príslušnými parametrami: `trieda.__init__(objekt,parametre)`
3. do premennej `instancia` priradí práve vytvorený `objekt`

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 __repr__(self):
        return 'Student: {} {}'.format(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:

```
>>> fero = Student('Ferdinand','Fyzik')
>>> fero.nastav_hoby('gitara')
Ferdinand Fyzik ma hoby gitara
>>> evka = Student('Eva', 'Matikova, 'cyklistika')
```

15.3 Štandardná funkcia dir()

Funkcia `dir()` vráti postupnosť (pole) 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. Niektoré z nich už poznáme: napr. `__init__` aj `__repr__` sú magické metódy. 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`.

15.4 Triedne a inštančné atribúty

Už vieme, že

- triedy sú kontajnery na súkromné funkcie (metódy)
- inštancie sú kontajnery na súkromné premenné (atribúty)

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
```

tak tento atribút automaticky získavajú (vidia) aj inštancie tejto triedy:

```
>>> print(t.x, t.y, t.z)
100 200 300
```

Aj novovytvorená inštancia získava tento atribút:

```
>>> t2 = Test()
>>> t2.z
300
```

Lenže tento atribút sa stále nachádza v kontajneri triedy `Test` a v kontajneroch inštancií nie sú. Ak ho chceme mať ako súkromnú premennú v inštancii, musíme mu 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.

KAPITOLA 16

Triedy a metódy

Zhrňme, čo už vieme o triedach a objektoch:

Novú triedu najčastejšie definujeme takto:

```
class Meno_triedy:
    triedny_atribut1 = hodnota1
    triedny_atribut2 = hodnota2
    ...

    def metoda1(self, parametre):
        ...

    def metoda2(self, parametre):
        ...

    def metoda3(self, parametre):
        ...
```

Triedne atribúty sú viditeľné vo všetkých inštanciách, kým sa neprekryjú vlastným súkromným atribútom s rovnakým menom:

```
>>> ins = Trieda()
>>> Trieda.atrib = 123      # vytvorenie triedneho atribútu
>>> print(ins.atrib)        # inštancia vidí tento triedny atribút
123
>>> ins.atrib = 'moj'       # inštancia prekryla triedny atribút
>>> print(Trieda.atrib, ins.atrib)
123 moj
```

Najčastejšie ich budeme používať vtedy, keď budeme chcieť, aby boli nejaké atribúty spoločné pre všetky inštancie, napr. spoločný `canvas` grafickej plochy, nejaké spoločné počítadlo, nejaké spoločné súradnice, ...

Niektoré metódy sú špeciálne (tzv. **magické**) - nimi definujeme špeciálne správanie (nová trieda sa lepšie prispôsobí filozofii Pythonu). Zatiaľ sme sa zoznámili s týmito dvoma magickými metódami:

- inicializácia `__init__(self,...)` - automaticky sa vyvolá hneď po vytvorení (skonštruovaní) objektu
- znaková reprezentácia `__repr__(self)` - automaticky sa vyvolá v priamom režime a tiež vo funkciách `print()` a `str()`

Objekty 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, môžu byť dvoch typov:

- **modifikátor** - mení niektorú hodnotu atribútu (alebo aj viacerých), napr.

```
class Bod:
    ...
    def vynuluj(self):
        self.x = self.y = 0
```

- **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 Bod:
    ...
    def vzdialenost(self, iny_bod):
        return math.sqrt((self.x-iny_bod.x)**2 + (self.y-iny_bod.y)**2)
```

- uvedomte si, že nemeniteľné typy (**immutable**) obsahujú iba pravé funkcie (okrem inicializácie `__init__()`)

16.1 1. príklad - trieda obdĺžnik

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. Zapíšme triedu obdĺžnikov (typ, pomocou ktorého budeme vytvárať objekty obdĺžniky):

```
class Obdlznik:

    def __init__(self, a, b):
        '''parametrami sú vel'kosti strán obdĺžnika'''
        self.a = a
        self.b = b

    def __repr__(self):
        return 'Obdlznik({},{})'.format(self.a,self.b)

    def obsah(self):
        return self.a * self.b

    def obvod(self):
```

```
        return 2*(self.a + self.b)

    def zmen_velkost(self, pomer):
        self.a *= pomer
        self.b *= pomer

    def kopia(self):
        return Obdlznik(self.a, self.b)
```

Všimnite si:

- metóda `__init__()` znamená, že pri vytváraní novej inštancie pomocou zápisu `Obdlznik()` **musíme** uviesť aj dva parametre: tieto sa uložia do súkromných premenných (atribútov) `a` a `b`
- metóda `__repr__()` označuje, že takýto objekt sa bude vedieť slušne vypísať, t.j. namiesto `<__main__.Obdlznik object at 0x0223F990>` dostaneme, napr. reťazec `Obdlznik(10,20)`
- metódy `obsah()` a `obvod()` sú pravé funkcie a počítajú obsah a obvod obdĺžnika
- metóda `zmen_velkost()` zmení veľkosť oboch strán obdĺžnika (vynásobí ich číslom `pomer`) - táto funkcia je modifikátor
- metóda `kopia()` vytvorí novú inštanciu, ktorá má rovnaké stavové premenné - toto je tiež pravá funkcia

Môžeme otestovať:

```
obd1 = Obdlznik(10,20)
print(obd1)
print('obvod =', obd1.obvod())
print('obsah =', obd1.obsah())
obd1.zmen_velkost(2.5)
print('nova_velkost:', obd1)
print('obvod =', obd1.obvod())
print('obsah =', obd1.obsah())
obd2 = obd1.kopia()
obd2.a += 5
print('novy:', obd2)
print('obvod =', obd2.obvod())
print('obsah =', obd2.obsah())
print('povodny:', obd1)
```

a výpis je potom:

```
Obdlznik(10,20)
obvod = 60
obsah = 200
nova_velkost: Obdlznik(25.0,50.0)
obvod = 150.0
obsah = 1250.0
novy: Obdlznik(30.0,50.0)
obvod = 160.0
obsah = 1500.0
povodny: Obdlznik(25.0,50.0)
```

16.2 2. príklad - trieda čas

Trieda dátum 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** (nemenný typ):

```
class Cas:

    def __init__(self, hodiny=0, minuty=0, sekundy=0):
        self.hod = hodiny
        self.min = minuty
        self.sek = sekundy

    def __repr__(self):
        return 'Cas {}:{:02}:{:02}'.format(self.hod, self.min, self.sek)

    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)
```

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 = Cas 10:22:30
cas2 = Cas 10:08:00
sucet = Cas 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:

- môžeme vytvoriť čas (napr. pomocou metódy `sucet()`), v ktorej minúty alebo sekundy majú viac ako 59
- dosť komplikovane sa porovnávajú dva časy

Vytvorme pomocnú funkciu, 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

    def __repr__(self):
        return 'Cas {}:{:02}:{:02}'.format(self.hod, self.min, self.sek)

    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 __repr__(self):
        return 'Cas {}:{: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 vacsi(self, iny):
        return self.sek > iny.sek
```

16.3 3. príklad - grafické objekty

Postupne zadefinujeme niekoľko tried pracujúcich s objektmi v grafickej ploche (pomocou `tkinter`)

Objekt Kruh

Zadefinujeme:

```
class Kruh:
    def __init__(self, x, y, r, farba='red'):
        self.x = x
        self.y = y
        self.r = r
        self.farba = farba

    def kresli(self, c):
        c.create_oval(self.x-self.r,self.y-self.r,
                      self.x+self.r,self.y+self.r,
                      fill=self.farba)

import tkinter

canvas = tkinter.Canvas(bg='white')
canvas.pack()
k1 = Kruh(50, 50, 30, 'blue')
k2 = Kruh(150, 100, 80)
k1.kresli(canvas)
k2.kresli(canvas)
```

Aby sme mohli nakreslený objekt posunúť alebo zmeniť jeho veľkosť alebo farbu, musíme si zapamätať jeho identifikačné číslo - vráti ho 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`:

```
class Kruh:
    def __init__(self, x, y, r, farba='red'):
        self.x = x
        self.y = y
        self.r = r
        self.farba = farba

    def kresli(self, canvas):
        self.canvas = canvas
        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):
        self.farba = farba
        self.canvas.itemconfig(self.id, fill=farba)

import tkinter

canvas = tkinter.Canvas(bg='white')
canvas.pack()
k1 = Kruh(50, 50, 30, 'blue')
k2 = Kruh(150, 100, 80)
k1.kresli(canvas)
k2.kresli(canvas)

k1.posun(30,10)
k2.zmen(50)
k1.prefarbi('green')
```

Všimnime si metódu `kresli()`: slúži na to, aby sme objektu odovzdali `canvas`, do ktorého sa má kresliť a vďaka tomu sa tento objekt aj nakreslí. Keď máme viac objektov kruhov, každý z nich si pamätá svoju súkromnú premennú `canvas`. Pričom asi všetky si budú pamätať jedinú grafickú plochu. Tu je pekná príležitosť využiť triedny atribút. Ukážme, ako sa zmení tento predchádzajúci program (zároveň vyhodíme teraz už zbytočnú metódu `kresli()`):

```
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):
        self.farba = farba
        self.canvas.itemconfig(self.id, fill=farba)

import tkinter

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')
```

Trieda Obdlznik

Skopírujeme triedu `Kruh` a zmeníme na `Obdlznik`:

```
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)

import tkinter

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:

```
class Skupina:
    def __init__(self):
        self.pole = []

    def pridaj(self, utvar):
        self.pole.append(utvar)

import tkinter

c = tkinter.Canvas(bg='white')
```

```
c.pack()
Kruh.canvas = Obdlznik.canvas = c

sk = Skupina()
sk.pridaj(Kruh(50, 50, 30, 'blue'))
sk.pridaj(Obdlznik(100, 20, 100, 50))
sk.pole[0].prefarbi('green')
sk.pole[1].posun(50)
```

Vidíme, ako sa menia už nakreslené útvary.

Ak budeme potrebovať meniť viac útvarov, použijeme cyklus:

```
for i in range(len(sk.pole)):
    sk.pole[i].prefarbi('orange')
```

alebo

```
for utvar in sk.pole:
    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.pole:
            utvar.prefarbi(farba)

    def posun(self, dx=0, dy=0):
        for utvar in self.pole:
            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:
    def __init__(self, x, y, r, farba='red'):
        ...
        self.typ = 'kruh'

class Obdlznik:
    def __init__(self, x, y, sirka, vyska, farba='red'):
        ...
        self.typ = 'obdlznik'

class Skupina:
    ...
    def posun_typ(self, typ, dx=0, dy=0):
        for utvar in self.pole:
            if utvar.typ == typ:
                utvar.posun(dx, dy)

    def prefarbi_typ(self, typ, farba):
        for utvar in self.pole:
            if utvar.typ == typ:
                utvar.prefarbi(farba)
```

Môžeme vygenerovať skupinu 20 náhodných útvarov - kruhov a obdĺžnikov:

```
import tkinter, random

c = Kruh.canvas = Obdlznik.canvas = tkinter.Canvas(bg='white')
c.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 __repr__()

Do oboch tried `Kruh` aj `Obdlznik` pridáme magickú metódu `__repr__()` a vďaka tomu môžeme vypísať všetky útvary v skupine:

```
class Kruh:
    ...
    def __repr__(self):
        return 'Kruh({},{},{},{})'.format(
            self.x, self.y, self.r, repr(self.farba))

class Obdlznik:
    ...
    def __repr__(self):
        return 'Obdlznik({},{},{},{},{})'.format(
            self.x, self.y, self.sirka, self.vyska, repr(self.farba))


...
for utvar in sk.pole:
    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')
...
```

KAPITOLA 17

Triedy a dedičnosť

Čo už vieme o triedach a ich inštanciách:

- triedy sú kontajnery atribútov:
- väčšinou sú to funkcie, t.j. metódy
- iné sú triedne premenné
- niektoré metódy sú "magické": začínajú aj končia dvomi znakmi '__' a každá z nich má pre Python svoje špeciálne využitie
- triedy sú vlastne vzormi na vytvárame inštancií
- aj inštancie sú kontajnery atribútov:
- väčšinou sú to súkromné premenné inštancií
- ak nejaký atribút nie je definovaný v inštancii, tak Python zabezpečí, že sa použije atribút z triedy (inštancia automaticky "vidí" triedne atribúty)

17.1 Objektové programovanie

je v programovacom jazyku charakterizované týmito tromi vlastnosťami:

- **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 pracuje s údajmi len pomocou týchto metód
- pripomeňme si triedu `Zlomok` z cvičení: v atribútoch `citatel` a `menovatel` sa vždy nachádzajú tieto hodnoty v základnom tvare, pričom menovateľ by mal byť vždy kladné (nenulové) číslo, predpokladáme, že s týmito atribútmi nepracujeme priamo, ale len pomocou metód `__init__()`, `sucet()`, `sucin()`, ...
- z tohto dôvodu, by sme niekedy potrebovali dáta skryť a doplniť funkcie, tzv. **getter** a **setter** pre tie atribúty, ktoré chceme nejako ochrániť, neskôr uvidíme ďalší pojem, ktorý s týmto súvisí, tzv. **vlastnosť** (property)
- **dedičnosť** (inheritance)
- novú triedu nevytvárame z nuly, ale využijeme už existujúcu triedu
- **polymorfizmus**
- budeme sa týmto zaoberať v ďalších prednáškach

17.2 Dedičnosť

Začneme definíciou jednoduchej triedy:

```
class Bod:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Bod({},{})'.format(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__()`, `__repr__()` a `posun()`, ale aj veľké množstvo neznámych identifikátorov, o ktorých 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**
- triede, ktorá vznikne dedením z inej triedy, hovoríme **odvodená trieda**, alebo **podtrieda**

Niekedy sa vzťahu základná trieda a odvodená trieda hovorí aj terminológiou **rodič** a **potomok** (potomok zdedil nejaké vlastnosti od svojho rodiča).

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__()`, `__repr__()`, `posun()` a metódu `zmen_farbu()` sme jej dodefinovali. 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)
fbod.zmen_farbu('red')
fbod.posun(dy=50)
print('fbod =', fbod)
```

Zdedené metódy môžeme v novej triede nielen využívať, ale aj predefinovať - napr. 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 = Farebny_bod(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):
        self.__init__(x, y)
        self.farba = farba
    ...
```

Toto je totiž rekurzívne volanie, ktoré spôsobí spadnutie programu. Musíme to zapísať takto:

```
class FarebnyBod(Bod):
    def __init__(self, x, y, farba):
        Bod.__init__(self, x, y)
        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):
        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.

Grafické objekty

Trochu sme upravili grafické objekty `Kruh`, `Obdlznik` a `Skupina` z prednášky: *Triedy a metódy*:

```
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)
        self.typ = 'kruh'

    def __repr__(self):
        return 'Kruh({},{},{},{})'.format(
            self.x, self.y, self.r, repr(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):
        self.farba = farba
        self.canvas.itemconfig(self.id, fill=farba)

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)
        self.typ = 'obdlznik'

    def __repr__(self):
        return 'Obdlznik({},{},{},{},{})'.format(
            self.x, self.y, self.sirka, self.vyska, repr(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)

class Skupina:
    def __init__(self):
        self.pole = []

    def pridaj(self, utvar):
        self.pole.append(utvar)

    def prefarbi(self, farba):
        for utvar in self.pole:
            utvar.prefarbi(farba)

    def posun(self, dx=0, dy=0):
        for utvar in self.pole:
            utvar.posun(dx, dy)

    def posun_typ(self, typ, dx=0, dy=0):
        for utvar in self.pole:
            if utvar.typ == typ:
                utvar.posun(dx, dy)

    def prefarbi_typ(self, typ, farba):
        for utvar in self.pole:
            if utvar.typ == typ:
                utvar.prefarbi(farba)

#----------------------------------------

import tkinter

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:

```
class Utvar:
    canvas = None

    def __init__(self, x, y, farba='red'):
        self.x = x
        self.y = y
        self.farba = farba
        self.id = None

    def posun(self, dx=0, dy=0):
        self.x += dx
        self.y += dy
        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 nebude fungovať ani jedna ďalšia metóda. 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 = sirka
        self.vyska = 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 = sirka
        self.vyska = 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.

17.3 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
>>> type(t2) == Obdlznik
True
>>> type(t1) == Utvar
False
```

Okrem tohto testu môžeme použiť štandardnú funkciu `isinstance(i,t)` 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`:

```
class Skupina:
    def __init__(self):
        self.pole = []

    def pridaj(self, utvar):
        self.pole.append(utvar)

    def prefarbi(self, farba):
        for utvar in self.pole:
            utvar.prefarbi(farba)

    def posun(self, dx=0, dy=0):
        for utvar in self.pole:
            utvar.posun(dx, dy)

    def posun_typ(self, typ, dx=0, dy=0):
        for utvar in self.pole:
            if isinstance(utvar, typ):
                utvar.posun(dx, dy)

    def prefarbi_typ(self, typ, farba):
        for utvar in self.pole:
            if isinstance(utvar, typ):
                utvar.prefarbi(farba)
```

a použiť

```
import random

def ri(a, b):
    return random.randint(a, b)

sk = Skupina()
for i in range(20):
    if ri(0, 1):
        sk.pridaj(Kruh(ri(50,350), ri(50,350), ri(10,50)))
    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()`. 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
```

17.4 Odvodená trieda od Turtle

aj od triedy `Turtle` z prednášky: *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:

```
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)
```

Zároveň sme tu zadefinovali metódu `domcek()`, ktorá nakreslí domček zadanej veľkosti. Otestujeme:

```
t = MojaTurtle1(-200, 100)
t.domcek(100)
```

Vytvorme odvodené triedy od triedy `MojaTurtle`, v ktorých pozmeníme kreslenie rovnej 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)

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)
```

Otestujme:

```
from random import randint as ri
turtle.delay(0)
MojaTurtle1(-100, 100).domcek(100)
MojaTurtle2(100, 100).domcek(100)
```

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
>>>
```

18.1 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`

Príkazy medzi `try` a `except` bude 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 sa dá 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
>>> {'a':1, 'b':2}['c']
...
KeyError: 'c'
>>> 5()
...
TypeError: 'int' object is not callable
>>> ''.x
...
AttributeError: 'str' object has no attribute 'x'
```

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.

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 poľa. 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 poľa

Zapíšme funkciu:

```
pole = ['prvy', 'druhy', 'treti', 'stvrty']

def zisti():
    while True:
        try:
            vstup = input('zadaj index: ')
            index = int(vstup)
            print('prvok pola =', pole[index])
            break
        except ValueError:
            print('*** chybne zadane cele cislo ***')
        except IndexError:
            print('*** index mimo rozsah pola ***')
```

otestujeme:

```
>>> zisti()
zadaj index: 22
*** index mimo rozsah pola ***
zadaj index: 2.
*** chybne zadane cele cislo ***
zadaj index: 2
prvok pola = treti
>>>
```

To isté by sme dosiahli aj vtedy, keby sme to zapísali pomocou dvoch vnorených príkazov `try`:

```
def zisti():
    while True:
        try:
            try:
                vstup = input('zadaj index: ')
                index = int(vstup)
                print('prvok pola =', pole[index])
                break
            except ValueError:
                print('*** chybne zadane cele cislo ***')
        except IndexError:
            print('*** index mimo rozsah pola ***')
```

Ukážme využitie konštrukcie `try` - `except` pri riešení známej úlohy s frekvenčnou tabuľkou, t.j. vytvoríme asociatívne pole počítadiel, v ktorom si budeme zisťovať počet výskytov hodnôt v poli. Doteraz sme to riešili buď takto:

```
def tabulka(pole):
    vysl = {}
    for prvok in pole:
        if prvok in vysl:
            vysl[prvok] += 1
        else:
            vysl[prvok] = 1
    return vysl
```

alebo

```
def tabulka(pole):
    vysl = {}
    for prvok in pole:
        vysl[prvok] = vysl.get(prvok, 0) + 1
    return vysl
```

Prvé z týchto riešení najprv pozrie, či sa hľadaný `prvok` už nachádza v tabuľke, a ak áno, k počítadlu pripočíta 1. Inak vyrobí nové počítadlo s počiatočnou hodnotou 1. Práve toto prvé riešenie môžeme elegantne zapísať aj s využitím výnimky:

```
def tabulka(pole):
    vysl = {}
    for prvok in pole:
        try:
            vysl[prvok] += 1
        except KeyError:
            vysl[prvok] = 1
    return vysl
```

Takýto spôsob riešenia úloh pomocou výnimiek je v Pythone dosť bežný. Všimnite si, že v tejto funkcii sa zachytáva iba výnimka `KeyError`, t.j. v asociatívnom poli sa nenachádza tento kľúč. Lenže príkaz `vysl[prvok] += 1` môže vyvolať ešte jednu chybu a to v prípade, že daná hodnota nemôže byť kľúčom v asociatívnom poli. Vyskúšajme:

```
>>> tabulka([1, 2, [3], 4])
...
: unhashable type: 'list'
```

Ak by sa nám v takomto prípade hodilo ignorovať takéto prvky poľa, vyriešime to takto jednoducho:

```
def tabulka(pole):
    vysl = {}
    for prvok in pole:
        try:
            vysl[prvok] += 1
        except KeyError:
            vysl[prvok] = 1
        except TypeError:
            pass
    return vysl
```

a znovu otestujeme:

```
>>> tabulka([1, 2, [3], 4])
{1: 1, 2: 1, 4: 1}
```

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():
    while True:
        try:
            print('prvok pola =', pole[int(input('zadaj index: '))])
            break
```

```
        except (ValueError, IndexError):
            print('*** chybne zadany index pola ***')
```

```
>>> zisti()
zadaj index: 22
*** chybne zadany index pola ***
zadaj index: 2.
*** chybne zadany index pola ***
zadaj index: 2
prvok pola = 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():
    while True:
        try:
            print('prvok pola =', pole[int(input('zadaj index: '))])
            break
        except:
            print('*** chybne zadany index pola ***')
```

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 takéto bloky `try - except` bez uvedenia výnimky sú zdrojom veľmi nečakaných chýb.

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):
  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.

18.2 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
*** chybne zadane cele cislo ***
zadaj cislo: styri
...
ValueError: invalid literal for int() with base 10: 'styri'
```

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')
```

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
>>>
```

18.3 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). 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
```

Triedy a operácie 1.

Pripomeňme si:

- pri definovaní triedy môžeme využiť tieto magické metódy:
- `__init__()` - inicializačná metóda
- `__repr__()` - metóda na reťazcovú reprezentáciu údajov v inštancii
- magické (špeciálne) metódy vždy začínajú aj končia dvomi znakmi `'__'`
- pre Python majú vždy špeciálny význam

19.1 Trieda Tabuľka

Postupne budeme vytvárať triedu, ktorá umožní spravovať napr. telefónny zoznam. Hoci sa bude veľmi podobať na asociatívne pole (typ `dict`), budeme ju riešiť len pomocou obyčajného poľa (typ `list`):

- bude to tabuľka s dvomi stĺpcami: v prvom je tzv. kľúč, ktorému zodpovedá nejaká hodnota, ktorá bude v druhom stĺpci
- napr. pre telefónny zoznam každému menu (v prvom stĺpci) zodpovedá telefónne číslo (hodnota v druhom stĺpci)
- s položkami tabuľky (dvojice: kľúč a hodnota) budeme pracovať pomocou dvoch metód:
  - `daj_hodnotu()` - pre zadaný kľúč vráti príslušnú hodnotu
  - `nastav_hodnotu()` - ak v tabuľke už existuje riadok s daným kľúčom, tak zmení príslušnú hodnotu, inak pridá nový riadok s dvojicou kľúč a hodnota

Zapíšme prvú verziu triedy `Tabulka`:

```
class Tabulka:
    def __init__(self):
        self.pole = []

    def daj_hodnotu(self, kluc):
        for prvok in self.pole:
            if prvok[0] == kluc:
                return prvok[1]
        raise KeyError(kluc)

    def nastav_hodnotu(self, kluc, hodnota):
        for prvok in self.pole:
            if prvok[0] == kluc:
```

```
                prvok[1] = hodnota
                return
        self.pole.append([kluc, hodnota])
```

Zadefinujme tabuľku:

```
tel = Tabulka()
tel.nastav_hodnotu('Evka', '0903123456')
tel.nastav_hodnotu('Danka','0911223344')
tel.nastav_hodnotu('Zuzka','0912131415')
tel.nastav_hodnotu('Anka', '0901010101')
tel.nastav_hodnotu('Petka','0909090909')
```

Otestujme:

```
>>> tel
<__main__.Tabulka object at 0x00000000035820B8>
>>> tel.daj_hodnotu('Zuzka')
'0912131415'
>>> tel.daj_hodnotu('anka')
...
KeyError: 'anka'
>>>
>>> tel.pole
[['Evka', '0903123456'], ['Danka', '0911223344'], ['Zuzka', '0912131415'],
['Anka', '0901010101'], ['Petka', '0909090909']]
```

Vidíme, že tabuľka je uložená v poli (atribút `pole`), ktoré v každom prvku obsahuje dvojprvkové pole [kľúč, hodnota].

Aby sa s takouto tabuľkou pracovalo pohodlnejše, dodefinujeme ďalšie metódy:

- `__repr__()` - vráti prvky tabuľky v čitateľnom tvare
- `pocet()` - vráti počet prvkov tabuľky
- `zrus_hodnotu()` - vyhodí z tabuľky dvojicu kľúč a jeho hodnota
- `je_kluc()` - zistí, či je v tabuľke zadaný kľúč (vráti `True` alebo `False`)
- `kluce()` - vráti zoznam (pole) všetkých kľúčov

```
class Tabulka:
    def __init__(self):
        self.pole = []

    def daj_hodnotu(self, kluc):
        for prvok in self.pole:
            if prvok[0] == kluc:
                return prvok[1]
        raise KeyError(kluc)

    def nastav_hodnotu(self, kluc, hodnota):
        for prvok in self.pole:
            if prvok[0] == kluc:
                prvok[1] = hodnota
                return
        self.pole.append([kluc, hodnota])

    def zrus_hodnotu(self, kluc):
        for i in range(len(self.pole)):
```

```
            prvok = self.pole[i]
            if prvok[0] == kluc:
                del self.pole[i]
                return
        raise KeyError(kluc)

    def je_kluc(self, kluc):
        for prvok in self.pole:
            if prvok[0] == kluc:
                return True
        return False

    def __repr__(self):
        vysl = ''
        for prvok in self.pole:
            vysl += '{} = {}\n'.format(repr(prvok[0]),repr(prvok[1]))
        return vysl

    def pocet(self):
        return len(self.pole)

    def kluce(self):
        vysl = []
        for prvok in self.pole:
            vysl.append(prvok[0])
        return vysl
```

Otestujeme s rovnako vytvorenou tabuľkou:

```
>>> tel
'Evka' = '0903123456'
'Danka' = '0911223344'
'Zuzka' = '0912131415'
'Anka' = '0901010101'
'Petka' = '0909090909'

>>> tel.daj_hodnotu('Zuzka')
'0912131415'
>>> tel.je_kluc('anka')
False
>>> tel.kluce()
['Evka', 'Danka', 'Zuzka', 'Anka', 'Petka']
>>> tel.zrus_hodnotu('Anka')
>>> tel.pocet()
4
>>> tel.pole
[['Evka', '0903123456'], ['Danka', '0911223344'], ['Zuzka', '0912131415'], [
↪'Petka', '0909090909']]
```

19.2 Vyhľadávanie

Pozrime na metódy `daj_hodnotu()`, `nastav_hodnotu()`, `zrus_hodnotu()` a `je_kluc()`: všetky 4 postupne prechádzajú všetky prvky poľa `self.pole` a porovnávajú zadaný kľúč s prvým prvkom poľa. Toto samozrejme funguje správne, len pre trochu väčšie polia to môže trvať naozaj dlho. Skúsme to nejako odmerať. Python umožňuje odmerať približný čas trvania nejakého kódu pomocou funkcie `time.time()` z modulu `time`. Napr.

tento kód vypíše, koľko sekúnd trvalo vypočítať súčet čísel od 1 do `n`:

```
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)
```

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). Takúto schému merania času bubeme odteraz používať častejšie:

```
import time

start = time.time()                    # začiatok merania času

# nejaký výpočet

print(round(time.time()-start, 3))    # koniec merania času
```

Samozrejme, že si treba dať pozor, aby v meranom výpočte neboli žiadne kontrolné tlače (`print()`), resp. čítanie hodnôt zo vstupu (`input()`), lebo tie veľmi skreslia odmeraný čas.

Po spustení nášho merania času dostávame:

```
zadaj n: 10000
cas = 0.002
sucet = 50005000
```

```
zadaj n: 1000000
cas = 0.246
sucet = 500000500000
```

Môžete si tu všimnúť, že keď sme prvýkrát spustili program s hodnotou `n=1000`, dostali sme čas približne 2 milisekundy. Keď sme hop spustili s hodnotou `n=100000`, t.j. 100-krát väčšou, dostali sme približne 100-krát väčší čas. A pre ešte 10-krát väčší vstup sa aj približne 10-krát predĺži výpočtový čas:

```
zadaj n: 10000000
cas = 2.226
sucet = 50000005000000
```

Poďme týmto spôsobom merať čas hľadania kľúča v telefónnom zozname. Použijeme tri rôzne veľké textové súbory `telefon.txt`, `telefon1.txt` a `telefon2.txt`, ktoré majú 1000, 10000, resp. 100000 dvojíc meno a telefónne číslo:

```
import time

start = time.time()
tel = Tabulka()
with open('telefon.txt') as text:
    for riadok in text:
        kluc, hodnota = riadok.split()
```

```
        tel.nastav_hodnotu(kluc, hodnota)
print('cas1 =', round(time.time()-start, 3))

print('pocet =', tel.pocet())
kluce = tel.kluce()
start = time.time()
for k in kluce:
    tel.daj_hodnotu(k)
print('cas2 =', round(time.time()-start, 3))
```

Tento program, najprv vytvorí novu tabuľku `Tabulka()` a potom do nej pridá všetky dvojice mien s telefónnymi číslami. Po spustení dostávame:

```
cas1 = 0.087
pocet = 1000
cas2 = 0.058
```

Prvý odmeraný čas označuje, koľko trvalo vytvorenie celej tabuľky, druhý odmeraný čas je doba trvania vyhľadania všetkých mien v telefónnom zozname. Zaujímavé bude, keď toto meranie spustíme pre 10-krát väčší súbor `telefon1.txt`:

```
cas1 = 5.566
pocet = 9991
cas2 = 5.128
```

Vidíme, že odmerané časy je skoro 100-krát väčšie ako pri 10-krát menšom súbore. Ak budete mať trpezlivosť odmerať aj súbor `telefon2.txt`, zistíte že oba odmerané časy sa ešte približne 100-krát zväčšili.

Samozrejme, že túto istú úlohu s telefónnym zoznamom vieme vyriešiť pomocou asociatívneho poľa (typ `dict`). Len pre porovnanie odmerajme, koľko to bude trvať teraz (pre súbor `telefon1.txt`):

```
import time

start = time.time()
tel = dict()
with open('telefon1.txt') as text:
    for riadok in text:
        kluc, hodnota = riadok.split()
        tel[kluc] = hodnota
print('cas1 =', round(time.time()-start, 3))

print('pocet =', len(tel))
kluce = tel.keys()
start = time.time()
for k in kluce:
    tel[k]
print('cas2 =', round(time.time()-start, 3))
```

Po spustení:

```
cas1 = 0.019
pocet = 9991
cas2 = 0.002
```

Väčším prekvapením bude odmeraný čas pre 100000-riadkový súbor `telefon2.txt`:

```
cas1 = 0.258
pocet = 99486
cas2 = 0.03
```

Pre 10-krát väčší súbor sa čas nepredĺžil 100-násobne ale asi len 15-násobne. Zrejme má Python pri hľadaní v asociatívnom poli nejakú *veľkú fintu*, vďaka ktorej funguje veľmi rýchlo. My si tu ukážeme jedno malé vylepšenie realizácie metód našej triedy `Tabulka`, ktoré nám trochu urýchlia vyhľadávanie v poli aj vkladanie do poľa.

Pripomeňme si algoritmus na približný výpočet druhej odmocniny nejakého čísla pomocou delenia intervalu na polovice zo štvrtej prednášky (*Podmienené príkazy*):

```
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)
```

V tomto cykle sa interval desatinných čísel stále delil na polovice a pokračovalo sa v tej polovici, v ktorej sa nachádza hľadaná hodnota.

Presne tento istý nápad využijeme aj pre hľadanie nejakej hodnoty v poli - budeme mu hovoriť **binárne vyhľadávanie** (binary search (https://en.wikipedia.org/wiki/Binary_search_algorithm)). Jedine čo bude treba zabezpečiť, aby bolo toto pole utriedené. Potom bude algoritmus vyzerať takto:

- pozrieme na stredný prvok poľa a ak je jeho hodnota (hodnota kľúča) menšia ako stredný prvok, tak ďalej bude stačiť hľadať len v prvej polovici, inak budeme pokračovať v druhej polovici
- opäť pozrieme do stredu časti poľa, v ktorej máme pokračovať a opäť sa rozhodneme, v ktorej z polovíc budeme pokračovať ň
- takto budeme pokračovať (stále v menšom a menšom úseku poľa), kým nenájdeme hľadanú hodnotu, resp. zistíme, že sa tam nenachádza
- takýchto delení poľa na menšie úseky nemôže byť viac ako **dvojkový logaritmus** počtu prvkov poľa, teda napr. pre 100000 tento algoritmus prejde maximálne 17 opakovaní cyklu

Pozrime, ako by vyzerala metóda `daj_hodnotu()`:

```
class Tabulka:
    ...

    def daj_hodnotu(self, kluc):
        zac = 0                          # zaciatok intervalu
        kon = len(self.pole)-1           # koniec intervalu
        while zac <= kon:
            stred = (zac + kon) // 2     # stred intervalu
            if self.pole[stred][0] < kluc:
```

```
                zac = stred + 1
            elif self.pole[stred][0] > kluc:
                kon = stred - 1
            else:
                return self.pole[stred][1]
        raise KeyError(kluc)
```

Samozrejme, že toto bude fungovať len pre utriedené `self.pole`: každým prechodom while-cyklu sa skúmaný interval poľa zmenšuje na polovicu - vždy sa vyberie ten, v ktorom je šanca, že sa bude nachádzať hľadaný kľúč. Ak cyklus skončí, znamená to, že sa takýto kľúč nenašiel a vtedy funkcia vyvolá

Takže musíme opraviť aj ďalšie metódy. Metóda `nastav_hodnotu()` bud zmení hodnotu pre už existujúci kľúč, alebo pridá na správne miesto novú dvojicu `[kluc,hodnota]` tak. aby bolo `self.pole` usporiadané:

```
class Tabulka:
    ...

    def nastav_hodnotu(self, kluc, hodnota):
        zac = 0                             # zaciatok intervalu
        kon = len(self.pole)-1              # koniec intervalu
        while zac <= kon:
            stred = (zac + kon) // 2        # stred intervalu
            if self.pole[stred][0] < kluc:
                zac = stred + 1
            elif self.pole[stred][0] > kluc:
                kon = stred - 1
            else:
                self.pole[stred][1] = hodnota
                return
        self.pole.insert(zac, [kluc, hodnota])
```

Aj ďalšie dve metódy vylepšíme binárnym vyhľadávaním:

```
class Tabulka:
    ...

    def zrus_hodnotu(self, kluc):
        zac = 0
        kon = len(self.pole)-1
        while zac <= kon:
            stred = (zac + kon) // 2
            if self.pole[stred][0] > kluc:
                kon = stred - 1
            elif self.pole[stred][0] < kluc:
                zac = stred + 1
            else:
                del self.pole[stred]
                return
        raise KeyError(kluc)

    def je_kluc(self, kluc):
        zac = 0
        kon = len(self.pole)-1
        while zac <= kon:
            stred = (zac + kon) // 2
            if self.pole[stred][0] > kluc:
                kon = stred - 1
            elif self.pole[stred][0] < kluc:
```

```
                zac = stred + 1
            else:
                return True
        return False
```

Všetky tieto štyri metódy využívajú ten istý cyklus - toto sa zvykne riešiť tak, že sa zapíše jediná pomocná metóda napr. `index()`, ktorá vráti index prvku poľa `self.pole` s daným kľúčom alebo miesto, kde bude treba tento prvok vložiť. Potom samotné tieto metódy len zavolajú túto pomocnú metódu a urobia zvyšok práce.

Teraz, keď opäť spustíte meranie času, tak sa tieto odmerané časy výrazne vylepšia:

pre súbor `telefon.txt` s 1000 dvojicami kľúč, hodnota:

```
cas1 = 0.038
pocet = 1000
cas2 = 0.009
```

pre súbor `telefon1.txt` s 10000 dvojicami kľúč, hodnota:

```
cas1 = 0.169
pocet = 9991
cas2 = 0.097
```

pre súbor `telefon2.txt` s 10000 dvojicami kľúč, hodnota:

```
cas1 = 4.16
pocet = 99486
cas2 = 1.355
```

Môžete vidieť, že tento algoritmus výrazne pomohol rýchlosti hľadania v poli.

19.3 Operátory indexovania

Keď vytvárame v Pythone vlastnú triedu, môžeme sa rozhodnúť, či chceme aby sa s ňou pracovalo podobne ako, napr. s poľom (typ `list` alebo typ `dict`), t.j. keď pracujeme s poľom, môžeme zapisovať:

- `pole[index]` - vráť prvok so zadaným indexom, resp. kľúčom
- `pole[index] = hodnota` - zmeň príslušný prvok poľa
- `del pole[index]` - vyhodí prvok na zadanom indexe, resp. so zadaným kľúčom
- `len(pole)` - vráti počet prvkov poľa
- `hodnota in pole` - zistí, či sa zadaná hodnota nachádza v poli, resp. či je `hodnota` nejakým kľúčom

Využijeme na to ďalšie magické metódy, tzv. operátory indexovania.

Štandardná funkcia len()

V triede `Tabulka` sme definovali metódu `pocet()`, ktorá vráti počet prvkov poľa. Keď ale zavoláme:

```
>>> tel.pocet()
5
>>> len(tel)
...
TypeError: object of type 'Tabulka' has no len()
```

Python nám oznámil, že nevie, ako sa dá zistiť veľkosť nášej tabuľky. Totiž, keď zapíšeme `len(tel)`, Python sa pozrie, či existuje jedna špeciálna (magická) metóda `__len__()`, pomocou ktorej by vedel oznámiť výsledok `len()`. Teda by stačilo našu metódu `pocet()` premenovať na `__len__()` a už by to malo fungovať:

```
class Tabulka:
    ...
    def __len__(self):
        return len(self.pole)
```

Uvedomte si, že teraz sú oba zápisy identické (volá sa pri nich tá istá metóda):

```
>>> tel.__len__()
5
>>> len(tel)
5
```

Môžeme ešte preveriť, či to takto funguje aj pre štandardné polia (typ `list` alebo `tuple`) a prípadne aj pre znakové reťazce:

```
>>> len([2,3,5])
3
>>> [2,3,5].__len__()
3
>>> len('Python')
6
>>> 'Python'.__len__()
6
>>> str.__len__('Python')
6
```

Naozaj štandardná funkcia `len()` pracuje na takomto jednoduchom princípe: pre daný typ zavolá metódu `__len__()`.

Operátor in

Podobne, ako to bolo s funkciou `len()`, to bude aj s operátorom `in`. Zatiaľ máme definovanú metódu `je_kluc()`, ktorá zisťuje, či sa daný kľúč nachádza v tabuľke, ale `in`, ktorý by mal robiť to isté, nefunguje:

```
>>> tel.je_kluc('Danka')
True
>>> 'Danka' in tel
...
TypeError: argument of type 'Tabulka' is not iterable
```

Keď zapíšeme `'Danka' in tel`, Python zisťuje, či máme definovanú magickú metódu `__contains__()`. Ak áno, zavolá ju, inak hlási chybu `TypeError`. Zrejme aj v tomto prípade stačí nahradiť meno našej metódy `je_kluc()` magickým menom `__contains__()`:

```
class Tabulka:
    ...
    def __contains__(self, kluc):
        zac = 0
        kon = len(self.pole)-1
        while zac <= kon:
            stred = (zac + kon) // 2
            if self.pole[stred][0] > kluc:
                kon = stred - 1
            elif self.pole[stred][0] < kluc:
                zac = stred + 1
```

```
            else:
                return True
        return False
```

Teraz bude fungovať napr. aj:

```
>>> 'Danka' in tel
True
>>> 'Monika' in tel
False
```

čo je to isté ako:

```
>>> tel.__contains__('Danka')
True
>>> tel.__contains__('Monika')
False
>>> Tabulka.__contains__(tel,'Danka')
True
```

ale zrejme zápis s operátorom `in` je čitateľnejší a elegantnejší.

S operátorom `in` sme sa zatiaľ stretli pri poliach (typ `list` a `tuple`) a pri znakových reťazcoch (typ `str`). Teraz už vieme, že elegantný zápis je v skutočnosti volaním magickej metódy, napr.

```
>>> 7 in [2,3,5,7,11]
True
>>> [2,3,5,7,11].__contains__(7)
True
>>> 'x' in ('a','e','i','o','u')
False
>>> ('a','e','i','o','u').__contains__('x')
False
>>> 'th' in 'Python'
True
>>> 'Python'.__contains__('th')
True
```

Operátor indexovania [ ]

Pri práci s poliami a znakovými reťazcami používame zápisy `pole[index]` a `pole[index] = hodnota`. V skutočnosti aj takéto indexovanie Python prerába na volanie magických metód `__getitem__()` a `__setitem__()`. Môžeme otestovať:

```
>>> pole = [2,3,5,7,11]
>>> pole[3]
7
>>> pole.__getitem__(3)
7
>>> pole[2] = 99
>>> pole.__setitem__(2,99)
>>> pole
[2, 3, 99, 7, 11]
>>> 'Python'[3]
'h'
>>> 'Python'.__getitem__(3)
'h'
```

Zápisy pomocou `__getitem__()` a `__setitem__()` zrejme normálne používať nebudeme, tu nám len ilustrujú, ako to funguje v Pythone.

Takže, ak by sme zapísali `tel['Evka']`, Python bude hľadať magickú metódu `__getitem__()` a keďže ju nenájde, hlási chybu:

```
>>> tel['Evka']
...
TypeError: 'Tabulka' object is not subscriptable
```

Premenujme teda `daj_hodnotu()` na magické meno `__getitem__()` a tiež `nastav_hodnotu()` na `__setitem__()`:

```
class Tabulka:
    ...
    def __getitem__(self, kluc):
        zac = 0                             # zaciatok intervalu
        kon = len(self.pole)-1              # koniec intervalu
        while zac <= kon:
            stred = (zac + kon) // 2        # stred intervalu
            if self.pole[stred][0] < kluc:
                zac = stred + 1
            elif self.pole[stred][0] > kluc:
                kon = stred - 1
            else:
                return self.pole[stred][1]
        raise KeyError(kluc)

    def __setitem__(self, kluc, hodnota):
        zac = 0                             # zaciatok intervalu
        kon = len(self.pole)-1              # koniec intervalu
        while zac <= kon:
            stred = (zac + kon) // 2        # stred intervalu
            if self.pole[stred][0] < kluc:
                zac = stred + 1
            elif self.pole[stred][0] > kluc:
                kon = stred - 1
            else:
                self.pole[stred][1] = hodnota
                return
        self.pole.insert(zac, [kluc, hodnota])
```

a zmeníme aj vytvorenie tabuľky:

```
tel = Tabulka()
tel['Evka'] = '0903123456'
tel['Danka'] = '0911223344'
tel['Zuzka'] = '0912131415'
tel['Anka'] = '0901010101'
tel['Petka'] = '0909090909'
```

Zrejme Python očakáva, že `__getitem__()` má okrem `self` ešte jeden paramater a to je vlastne index (v našom prípade kľúč). Tiež magická metóda `__setitem__()` musí mať ďalšie dva parametre: index a priraďovanú hodnotu, inak by sa hlásila chyba.

Príkaz del

Pomocou príkazu `del` môžeme vyhodiť ľubovoľný prvok poľa:

```
>>> pole = [2, 3, 5, 7, 11]
>>> del pole[3]
>>> pole
[2, 3, 5, 11]
```

Asi nie je prekvapením, že Python v tomto prípade volá metódu `__delitem__()`, ktorej pošle index vyhadzovaného prvku poľa, t.j. :

```
>>> pole = [2, 3, 5, 7, 11]
>>> pole.__delitem__(3)
>>> pole
[2, 3, 5, 11]
```

Aj v našej triede `Tabulka` premenujeme metódu `zrus_hodnotu()` na magické meno `__delitem__()`. Kompletne zmenená trieda `Tabulka` teraz vyzerá tako (algoritmus binárneho hľadania sme tu presunuli do pomocnej metódy `index()` a metódu `kluce()` sme premenovali na `keys()`):

```
class Tabulka:
    def __init__(self):
        self.pole = []

    def index(self, kluc):
        zac = 0
        kon = len(self.pole)-1
        while zac <= kon:
            stred = (zac + kon) // 2
            if self.pole[stred][0] > kluc:
                kon = stred - 1
            elif self.pole[stred][0] < kluc:
                zac = stred + 1
            else:
                return stred, True
        return zac, False

    def __setitem__(self, kluc, hodnota):
        index, nasiel = self.index(kluc)
        if nasiel:
            self.pole[index][1] = hodnota
        else:
            self.pole.insert(index, [kluc, hodnota])

    def __getitem__(self, kluc):
        index, nasiel = self.index(kluc)
        if nasiel:
            return self.pole[index][1]
        raise KeyError(kluc)

    def __delitem__(self, kluc):
        index, nasiel = self.index(kluc)
        if nasiel:
            del self.pole[index]
        else:
            raise KeyError(kluc)

    def __contains__(self, kluc):
        return self.index(kluc)[1]

    def __repr__(self):
```

```
        vysl = ''
        for kluc, hodnota in self.pole:
            vysl += '{} = {}\n'.format(kluc, hodnota)
        return vysl

    def __len__(self):
        return len(self.pole)

    def keys(self):
        vysl = []
        for prvok in self.pole:
            vysl.append(prvok[0])
        return vysl
```

Takže, ak chceme vyhodiť z tabuľky nejaký riadok, môžeme zapísať:

```
>>> del tel['Evka']
>>> tel.__delitem__('Danka')
>>> tel
'Zuzka' = '0912131415'
'Anka' = '0901010101'
'Petka' = '0909090909'
```

Všimnite si, že naša trieda `Tabulka` má skoro všetky metódy magické (okrem `keys()` a pomocnej `index()`). Toto je dosť častý prípad pri definovaní tried v Pythone. Keď budete čítať nejaké hotové pythonovské triedy, pozrite si všetky magické metódy (začínajú aj končia s `'__'`) a zistite si, ktorá má akú špeciálnu interpretáciu pre Python.

Náš testovací program teraz vyzerá takto:

```
import time

start = time.time()
t = Tabulka()
with open('telefon1.txt') as text:
    for riadok in text:
        kluc, hodnota = riadok.split()
        t[kluc] = hodnota
print('cas1 =', round(time.time()-start,3))
print('pocet =', len(t))

kluce = t.keys()
start = time.time()
for k in kluce:
    t[k]
print('cas2 =', round(time.time()-start,3))
```

Tento zápis je teraz identický so zápisom pre testovanie asociatívneho poľa. Môžeme to združiť do jednej testovacej funkcie:

```
import time

def test(typ, subor):
    print('*** testovanie suboru', subor, 'pre triedu',typ.__name__, '***')
    start = time.time()
    t = typ()
    with open(subor) as text:
        for riadok in text:
            kluc, hodnota = riadok.split()
```

```
            t[kluc] = hodnota
    print('cas1 =', round(time.time()-start,3))
    print('pocet =', len(t))

    kluce = t.keys()
    start = time.time()
    for k in kluce:
        t[k]
    print('cas2 =', round(time.time()-start,3))

for subor in 'telefon.txt', 'telefon1.txt', 'telefon2.txt':
    test(Tabulka, subor)
    test(dict, subor)
```

Zhrnutie magických metód

Doteraz sme poznali len dve magické metódy (`__init__()` a `__repr__()`) a dnes sme sa zoznámili s ďalšími 5. Zhrňme do tabuľky, čo ktorá metóda znamená a kedy ju Python zavolá. `inst` tu znamená nejakú inštanciu (objekt), v prvom stĺpci tabuľky je bežné použitie v Pythone, v druhom stĺpci je to, ako to vidí Python (teda, čo urobí namiesto toho), a v treťom je stručné vysvetlenie:

Tabuľka 19.1: Magické metódy

`inst = Trieda(param)`	`Trieda.__init__(inst,param)`	inicializácia inštancie (funkcia nič nevracia)
`repr(inst)`	`inst.__repr__()`	znaková reprezentácia objektu (funkcia vráti reťazec)
`len(inst)`	`inst.__len__()`	počet prvkov (funkcia vráti celé číslo)
`hodnota in inst`	`inst.__contains__(hodnota)`	zistí, či je `hodnota` prvkom (funkcia vráti `True` alebo `False`)
`inst[index]`	`inst.__getitem__(index)`	vráti hodnotu na danom indexe (funkcia niečo vráti)
`inst[index] = hodnota`	`inst.__setitem__(index,hodnota)`	zmení hodnotu na danom indexe (funkcia nič nevracia)
`del inst[index]`	`inst.__delitem__(index)`	vyhodí hodnotu na danom indexe (funkcia nič nevracia)

KAPITOLA 20

Zásobníky a rady

20.1 Stack - zásobník

Zásobník je jedna so základných dátových štruktúr v programovaní a využíva sa v mnohých zložitejších algoritmoch. Zásobník si pamätá množinu (tzv. kolekciu) údajov, ktoré do nej vložíme, pričom vždy môžeme vyberať iba naposledy vložený údaj, resp. viac údajov, ale vyberáme ich v opačnom poradí, ako sme ich vložili. Zásobník si môžeme predstaviť napr. ako *rad* prichádzajúcich ľudí: každý nový sa postaví na koniec tohto radu. Lenže odoberať ľudí zo zásobníka (obsluhovať zákazníkov v rade) budeme len od konca, t.j. kto prišiel posledný, bude prvý obslúžený. Hovoríme tomu **LIFO** (last in first out), t.j. posledný vošiel a ako prvý odišiel. Môžeme tomu hovoriť aj **nespravodlivý rad**, lebo asi by sme nechceli čakať v takomto rade, kým nás obslúžia (len ak by sme prišli ako posledný).

Nová dátová štruktúra sa často definuje pomocou operácií, ktoré umožňujú s touto dátovou štruktúrou pracovať. Pre zásobník sú to operácie na pridávanie a odoberanie. Je zvykom ich nazývať anglickými názvami, t.j. **stack** má tieto operácie:

- **push(údaj)** - vloží údaj na koniec (tzv. vrch) zásobníka
- zásobník sa pritom predĺži o 1
- ak bol zásobník pred operáciou ešte prázdny, nový údaj vkladáme na tzv. dno zásobníka
- **pop()** - vyberie z vrchu zásobníka jeden údaj a tento vráti ako výsledok operácie
- zásobník sa pritom skráti o jeden prvok
- ak bol ale zásobník už prázdny, tak sa operácie nevykoná, ale spadne na chybovom stave (výnimka **EmptyError**)
- **top()** - podobne ako **pop()** vráti najvrchnejší prvok zásobníka
- ak bol zásobník prázdny, operácia nemá čo vrátiť, teda spadne na chybovom stave (výnimka **EmptyError**)
- hoci operácia vráti najvrchnejší prvok, zo zásobníka ho neodoberie, takže touto operáciou sa nemení počet prvkov v zásobníku
- **is_empty()** - zistí či je zásobník prázdny, t.j. vráti `True`, ak je prázdny a `False`, ak sa v ňom niečo nachádza

Zásobník sa často kreslí ako úzka jama (na šírku prvkov, ktoré do nej vkladáme): vloženie prvku (`push()`) vloží do jamy novú hodnotu ale na úplný vrch, t.j. nad všetky doterajšie prvky. Odoberáme (`pop()`) prvky vždy iba z vrchu jamy, t.j. najprv sa odoberie naposledy vložený prvok. Táto jama sa automaticky nafukuje, keď do nej vkladáme nové prvky, takže má skoro nekonečnú kapacitu.

V počítači budeme zásobník realizovať ako triedu `Stack`, ktorej metódy sú operácie zásobníka. Na ukladanie údajov môžeme využiť rôzne pythonovské štruktúry, napr. pole (`list`) alebo spájaný zoznam (uvidíme neskôr). Ukážeme dve rôzne realizácie zásobníka, obe pomocou poľa.

Zásobník pomocou poľa

V tejto realizácii využívame pythonovské pole (`list`): zásobník vytvárame tak, že dno je prvý prvok poľa (`self.pole[0]`) a vrch zásobníka je posledný prvok (`self.pole[-1]`). Pri práci s poľom Python ponúka dve základné metódy `pole.append(data)` a `pole.pop()`, ktoré robia presne to, čo potrebujeme, teda pridajú na koniec (vložia na vrch) a odoberú posledný prvok (odoberú z vrchu):

```
class EmptyError(Exception): pass

class Stack:
    def __init__(self):
        '''inicializácia vnútornej reprezentácie'''
        self.pole = []

    def push(self, data):
        '''vloží na vrch zásobníka novú hodnotu

        funkcia nevracia žiadnu hodnotu
        '''
        self.pole.append(data)

    def pop(self):
        '''vyberie z vrchu zásobníka jednu hodnotu

        prvok z vrchu zásobníka sa vráti ako výsledná hodnota funkcie
        zásobník sa pritom skráti o tento jeden prvok

        metóda vyvolá výnimku EmptyError pri prázdnom zásobníku
        '''
        if self.is_empty():
            raise EmptyError('prazdny zasobnik pri operacii pop()')
        return self.pole.pop()

    def is_empty(self):
        '''zistí, či je zásobník prázdny

        metóda vráti True alebo False
        '''
        return self.pole == []

    def top(self):
        '''vráti z vrchu zásobníka jednu hodnotu

        prvok z vrchu zásobníka sa vráti ako výsledná hodnota funkcie
        zásobník sa pritom neskracuje, ale ostáva v pôvodnom stave

        metóda vyvolá výnimku EmptyError pri prázdnom zásobníku
        '''
        if self.is_empty():
            raise EmptyError('prazdny zasobnik pri operacii top()')
        return self.pole[-1]
```

Otestujeme v interaktívnom režime:

```
>>> zasobnik = Stack()
>>> zasobnik.is_empty()
True
>>> zasobnik.push(37)
>>> zasobnik.push('ahoj')
```

```
>>> zasobnik.is_empty()
False
>>> zasobnik.top()
'ahoj'
>>> zasobnik.push(3.14)
>>> zasobnik.pop()
3.14
>>> zasobnik.pop()
'ahoj'
>>> zasobnik.pop()
37
>>> zasobnik.pop()
...
EmptyError: prazdny zasobnik
>>> zasobnik.is_empty()
True
```

Všimnite si, že vďaka dokumentačným riadkom na začiatku každej metódy vieme získať k tejto štruktúre takýto **help**:

```
>>> help(Stack)
Help on class Stack in module __main__:

class Stack(builtins.object)
 |  Methods defined here:
 |
 |  __init__(self)
 |      inicializácia vnútornej reprezentácie
 |
 |  is_empty(self)
 |      zistí, či je zásobník prázdny
 |
 |      metóda vráti True alebo False
 |
 |  pop(self)
 |      vyberie z vrchu zásobníka jednu hodnotu
 |
 |      prvok z vrchu zásobníka sa vráti ako výsledná hodnota funkcie
 |      zásobník sa pritom skráti o tento jeden prvok
 |
 |      metóda vyvolá výnimku EmptyError pri prázdnom zásobníku
 |
 |  push(self, data)
 |      vloží na vrch zásobníka novú hodnotu
 |
 |      funkcia nevracia žiadnu hodnotu
 |
 |  top(self)
 |      vráti z vrchu zásobníka jednu hodnotu
 |
 |      prvok z vrchu zásobníka sa vráti ako výsledná hodnota funkcie
 |      zásobník sa pritom neskracuje, ale ostáva v pôvodnom stave
 |
 |      metóda vyvolá výnimku EmptyError pri prázdnom zásobníku
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
```

```
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

>>> help(Stack.top)
Help on function top in module __main__:

top(self)
    vráti z vrchu zásobníka jednu hodnotu

    prvok z vrchu zásobníka sa vráti ako výsledná hodnota funkcie
    zásobník sa pritom neskracuje, ale ostáva v pôvodnom stave

    metóda vyvolá výnimku EmptyError pri prázdnom zásobníku
```

Presne takto isto si Python vytvára helpy ku všetkým funkciám, triedam, modulom. Zrejme, my budeme vytvárať takéto dokumentačné reťazce len v prípadoch, ak predpokladáme dlhodobejšie využívanie takejto triedy, resp. programujeme nové moduly/triedy pre ďalších programátorov.

Zásobník otestujeme jednoduchým programom, v ktorom prečítame nejaký súbor a vytvoríme z neho nový, ktorý má otočené poradie riadkov:

```
st = Stack()
with open('p.py') as subor:
    for riadok in subor:
        st.push(riadok)

with open('test.py','w') as subor:
    while not st.is_empty():
        subor.write(st.pop())
```

Ďalší príklad ilustruje použitie zásobníka pri testovaní, či je nejaká postupnosť palindrom, t.j. či má rovnaké poradie prvkov, keď sa číta odpredu alebo odzadu:

```
def zisti(pole):
    s = Stack()
    for prvok in pole:
        s.push(prvok)
    for prvok in pole:
        if prvok != s.pop():
            return False
    return True
```

otestujeme:

```
>>> zisti([1,2,5,2,1])
True
>>> zisti([1,2,3,5,2,1])
False
>>> zisti('jelenovipivonelej')
True
```

Zásobník pomocou poľa - druhá verzia

Pythonovské pole môžeme použiť pre zásobník aj opačným spôsobom:

- vrch zásobníka bude na začiatku poľa, teda `self.pole[0]` (dno je posledným prvkom `self.pole[-1]`)

- na vrch zásobníka budeme pridávať operáciou `pole.insert(0,data)` a odoberať operáciou `pole.pop(0)`
- toto bude fungovať úplne rovnako ako zásobník s vrchom na konci, len to bude v niektorých prípadoch trvať výrazne dlhšie:
- operácie `insert(0,data)` a `pop(0)` sú pri práci s veľmi dlhým poľom veľmi pomalé a budeme sa snažiť ich používať menej ako veľmi rýchle operácie `append(data)` a `pop()`
- tieto vlastnosti operácií s poľom uvidíme až neskôr, ale je dobre si nechať poradiť už teraz

Zapíšme to a otestujme, či je to naozaj výrazne pomalšie ako predchádzajúca verzia zásobníka:

```
class Stack0:
    def __init__(self):
        '''inicializácia vnútornej reprezentácie'''
        self.pole = []

    def push(self, data):
        '''vloží na vrch zásobníka novú hodnotu

        funkcia nevracia žiadnu hodnotu
        '''
        self.pole.insert(0, data)

    def pop(self):
        '''vyberie z vrchu zásobníka jednu hodnotu

        prvok z vrchu zásobníka sa vráti ako výsledná hodnota funkcie
        zásobník sa pritom skráti o tento jeden prvok

        metóda vyvolá výnimku EmptyError pri prázdnom zásobníku
        '''
        if self.is_empty():
            raise EmptyError('prazdny zasobnik pri operacii pop()')
        return self.pole.pop(0)

    def is_empty(self):
        '''zistí, či je zásobník prázdny

        metóda vráti True alebo False
        '''
        return self.pole == []

    def top(self):
        '''vráti z vrchu zásobníka jednu hodnotu

        prvok z vrchu zásobníka sa vráti ako výsledná hodnota funkcie
        zásobník sa pritom neskracuje, ale ostáva v pôvodnom stave

        metóda vyvolá výnimku EmptyError pri prázdnom zásobníku
        '''
        if self.is_empty():
            raise EmptyError('prazdny zasobnik pri operacii top()')
        return self.pole[0]
```

Aby sme mohli obe verzie porovnať, napíšme testovaciu funkciu, ktorej prvý parameter je testovaný typ (trieda `Stack` alebo `Stack0`):

```
import time

def test_rychlosti(typ_zasobnika, n):
    stack = typ_zasobnika()
    start = time.time()
    for i in range(n):
        stack.push(i)
    while not stack.is_empty():
        stack.pop()
    return round(time.time()-start, 3)

for n in 10000, 20000, 40000, 80000, 160000, 1000000:
    print(n)
    print('   Stack ', test_rychlosti(Stack, n))
    print('   Stack0', test_rychlosti(Stack0, n))
```

po stustení dostávame niečo takéto (trochu to závisí od počítača, na ktorom to spúšťate):

```
10000
   Stack  0.032
   Stack0 0.147
20000
   Stack  0.069
   Stack0 0.564
40000
   Stack  0.118
   Stack0 2.384
80000
   Stack  0.216
   Stack0 8.804
160000
   Stack  0.439
   Stack0 35.94
1000000
   Stack  2.861
   Stack0 1771.761
```

Všimnite si, že každý ďalší test (okrem posledného) má vždy zdvojnásobené `n` a preto aj oba cykly vo funkcii `test()` bežia dvojnásobne dlhšie. Pritom pre `Stack` sa aj čas trvania tohto testu približne zdvojnásobuje. Lenže pre `Stack0` sa každý ďalší test zoštvornásobuje: keď porovnáte časy trvania pre `n=10000` a pre `n=1000000` (teda pre 100-násobne väčšie `n`), čas pre `Stack` sa zvýšil približne 100-krát, ale pre `STack0` 10000-krát. Môžeme odhadovať, že pre `Stack` čas trvania lineárne závisí od `n` a pre `Stack0` kvadraticky.

Modul stack

Keďže ďalej budeme pracovať s našou novou triedou zásobník, uložme túto definíciu `Stack`, prípadne aj verziu `Stack0`, do samostatného modulu s menom `stack` (súbor `stack.py`). V tomto module môžeme nechať aj testovanie rýchlosti oboch verzií, ale tieto testovacie riadky zabalíme do špeciálneho príkazu `if`.

súbor `stack.py`:

```
class Stack:    # prvá verzia zásobníka
    ...

class Stack0:   # druhá pomalá verzia
    ...

if __name__ == '__main__':
```

```
import time

def test_rychlosti(typ_zasobnika, n):
    stack = typ_zasobnika()
    start = time.time()
    for i in range(n):
        stack.push(i)
    while not stack.is_empty():
        stack.pop()
    return round(time.time()-start, 3)

for n in 10000, 20000, 40000, 80000, 160000, 1000000:
    print(n)
    print('   Stack ', test_rychlosti(Stack, n))
    print('   Stack0', test_rychlosti(Stack0, n))
```

Takto zapísaný modul môžeme spúšťať v IDLE pomocou **F5** (Run) a vtedy sa spustí aj testovanie. Podmienka `if __name__ == '__main__':` označuje, že ak sa tento modul spustí ako hlavný program spustia sa aj príkazy v tele `if`. Ak ale niekto použije tento modul príkazom `import stack`, resp. `from stack import Stack`, príkazy v tele `if` sa nevykonajú (premenná `__name__` má vtedy hodnotu `'stack'` a nie `'__main__'`). Tento modul budeme používať v ďalších príkladoch.

Ďalej ukážeme dve užitočné aplikácie využitia dátovej štruktúry zásobník.

20.1.1 Aritmetické výrazy

Z matematiky už zo základnej školy vieme, že aritmetické výrazy sa zapisujú tak, že znamienko operácií sa zapisuje medzi dva operandy, napr.

```
2 + 3 * 4 = 14
```

Tento výsledok je 14, lebo najprv sa vypočíta 3 * 4 (násobenie má vyššiu prioritu ako súčet) a potom 2 + 12.

Ak chceme meniť poradie vyhodnocovania operácií, musíme niektoré časti výrazu uzavrieť do zátvoriek, napr.

```
(2 + 3) * 4 = 20
```

Takémuto zápisu hovoríme **infixový**, lebo každá operácia je medzi (in) svojimi dvoma operandmi.

V informatike sa zvykne pracovať aj s **postfixým** zápisom, v ktorom je operácia až **za** oboma svojimi operandmi, napr.

```
2 3 + = 5
3 4 * = 12
2 3 4 * + = 14
2 3 + 4 * = 20
4 2 3 + * = 20
```

V postfixovom zápise (niekedy sa mu hovorí aj poľská forma) neexistujú zátvorky ani priority operácií - vyhodnocovanie výrazu je vždy jednoznačné:

- vo výraze nájdeme prvú dvojicu čísel (dva operandy), za ktorou bezprostredne nasleduje znamienko operácie
- túto operáciu s týmito dvoma operandmi vyhodnotíme
- výsledkom nahradíme oba operandy aj s operáciou
- toto celé opakujeme, kým neostane jediná hodnota - táto je výsledkom celého výrazu

```
2 3 + 4 * -> 5 4 * -> 20
2 3 4 * + -> 2 12 + -> 14
4 2 3 + * -> 4 5 * -> 20
1 2 + 3 4 + 5 6 + * * -> 3 3 4 + 5 6 + * * -> 3 7 5 6 + * * -> 3 7 11 * * ->␣
↪3 77 * -> 231
```

Existuje ešte jeden typ zápisu: **prefixový** zápis, v ktorom je operácia **pred** dvoma svojimi operandmi. Napr.

```
+ 2 3 = 5
* 3 4 = 12
* + 2 3 4 = 20
+ 2 * 3 4 = 14
+ * 3 4 2 = 14
```

Tieto prefixové výrazy (niekedy sa im hovorí prevrátená poľská forma) sa vyhodnocujú veľmi podobným postupom ako postfixové.

Teda každý výraz vieme zapísať tromi rôznymi spôsobmi a po ich vyhodnotení samozrejme musíme dostať rovnaký výsledok. Zaujímavými sú algoritmy, ktorými vieme ľubovoľný typ výrazu previesť na nejaký iný. Bolo by ale dobré, keby ste si natrénovali ručný prevod medzi rôznymi typmi (môže sa to objaviť na záverečnom teste).

Napr. ten istý výraz v rôznych zápisoch:

- infix: `3 + 4 * 5`
- postfix: `3 4 5 * +`
- prefix: `+ 3 * 4 5`

alebo

- infix: `(3 + 4) * 5`
- postfix: `3 4 + 5 *`
- prefix: `* + 3 4 5`

Na vyhodnotenie výrazu, ktorý je zadaný v **postfixovom tvare** využijeme zásobník:

```
import stack

def pocitaj(retazec):
    s = stack.Stack()
    for p in retazec.split():
        if p == '+':
            s.push(s.pop()+s.pop())
        elif p == '*':
            s.push(s.pop()*s.pop())
        else:
            s.push(int(p))
    return s.pop()
```

V tejto verzii sme zabezpečili len operácie sčítania a násobenia. Podobne sa realizujú aj ďalšie operácie. Zrejme vo vstupnom reťazci sú operandy a operátory navzájom oddelené aspoň jednou medzerou, napr.

```
>>> pocitaj('3 4 5 * +')
23
>>> pocitaj('3 4 + 5 *')
35
```

20.1.2 Rekurzia

V Pythone je rekurzia obmedzená približne na 1000 rekurzívnych volaní. V niektorých situáciách je vhodné, keď rekurziu nahradíme nejakými cyklami. Napr. chvostová rekurzia sa prerába veľmi jednoducho. V zložitejších prípadoch využijeme zásobník. Totiž aj rekurzívne volanie Python realizuje pomocou zásobníka, ale tento je pred nami skrytý, tzv. systémový zásobník. Python sem pri každom volaní ukladá návratovú adresu ale zabezpečuje ním aj nové verzie lokálnych premenných (teda aj parametrov). Ak budeme vedieť toto nasimulovať, môžeme sa zbaviť rekurzie. Pozrime túto jednoduchú rekurzívnu funkciu:

```
def test(n):
    if n > 0:
        print('*' * n)
        test(n // 2)
        print('+' * n)
```

otestujeme:

```
>>> test(7)
*******
***
*
+
+++
+++++++
```

Prepisovať ju budeme na nerekurzívny tvar takto:

- využijeme zásobník, do ktorého budeme vkladať (teda aj vyberať) dvojice: pozícia v programe a hodnota lokálnej premennej (teda parametra) `n`
- ak by sme mali viac lokálnych premenných, resp. parametrov, do zásobníka by sme ich vkladali (a vyberali) naraz všetky tieto premenné
- rekurzívnu funkciu rozdelíme na časti, pričom deliacou čiarou bude rekurzívne volanie
    - v našom programe budú dve časti: jedna pred jediným rekurzívnym volaním a druhá za volaním:

```
def test(n):
    # prvá časť
    if n > 0:
        print('*'*n)
        test(n//2)
        # druhá časť
        print('+'*n)
```

- samotné rekurzívne volanie sa nahradí dvoma vloženými prvkami do zásobníka:
- keďže treba zabezpečiť rekurzívne volanie s novou hodnotou `n`, do zásobníka sa najprv vloží dvojica: chceme pracovať od pozície **prvej** časti (teda od začiatku) s novou hodnotou parametra `n//2`
- tiež po skončení rekurzívneho volania chceme, aby sa pokračovalo od **druhej** pozície programu s pôvodnou hodnotou `n` - teda aj toto vložíme do zásobníka
- keďže do zásobníka vkladáme dve dvojice a chceme, aby sa vybrala najprv ta prvá, vložíme ich v opačnom poradí
- celé vnútro rekurzívnej funkcie zabalíme do while-cyklu, ktorý bude bežať, kým je zásobník neprázdny (teda je ešte čo robiť)
- pred týmto while-cyklom ešte vytvoríme zásobník a vložíme do neho prvú dvojicu, aby sa to mohlo celé naštartovať: chceme pracovať od **prvej** pozície s pôvodnou hodnotou `n`

Zmenená **nerekurzívna** funkcia teraz vyzerá takto:

```
import stack

def test(n):
    s = stack.Stack()
    s.push((1, n))
    while not s.is_empty():
        a, n = s.pop()
        if a == 1:
            # prvá časť
            if n > 0:
                print('*' * n)
                s.push((2, n))
                s.push((1, n // 2))
        elif a == 2:
            # druhá časť
            print('+' * n)
```

Môžete vyskúšať, že dostávate rovnaké výsledky ako z rekurzívnej funkcie.

Tú istú ideu vyskúšajme na trochu zložitejšej verzii rekurzívnej funkcie:

```
def test(n):
    if n > 0:
        print('*' * n)
        test(n // 2)
        print('+' * n)
        test(n // 2)
        print('*' * n)
```

otestujeme:

```
>>> test(7)
*******
***
*
+
*
+++
*
+
*
***
+++++++
***
*
+
*
+++
*
+
*
***
*******
```

V tejto funkcii sú teraz už dve rekurzívne volania, teda funkciu rozdelíme na 3 časti:

1. od začiatku až po prvé rekurzívne volanie

2. od prvého volania po druhé
3. od druhého až do konca

```
def test(n):
    # prvá časť
    if n > 0:
        print('*' * n)
        test(n // 2)
        # druhá časť
        print('+' * n)
        test(n // 2)
        # tretia časť
        print('*' * n)
```

Prepísaná funkcia vyzerá takto:

```
import stack

def test(n):
    s = stack.Stack()
    s.push((1, n))
    while not s.is_empty():
        a,n = s.pop()
        if a == 1:
            # prvá časť
            if n > 0:
                print('*' * n)
                s.push((2, n))
                s.push((1, n // 2))
        elif a == 2:
            # druhá časť
            print('+' * n)
            s.push((3, n))
            s.push((1, n // 2))
        elif a == 3:
            # tretia časť
            print('*' * n)
```

Porovnajte túto funkciu s predchádzajúcou jednoduchšou verziou funkcie.

Pripomeňme si rekurzívne kreslenie binárneho stromu pomocou korytnačky:

```
import turtle

def strom(n, d):
    # prva cast
    t.fd(d)
    if n > 0:
        t.lt(40)
        strom(n-1, d*.7)
        # druha cast
        t.rt(75)
        strom(n-1, d*.6)
        # tretia cast
        t.lt(35)
    t.bk(d)

t = turtle.Turtle()
```

```
t.speed(0)
t.lt(90)
strom(5, 80)
```

V tejto rekurzívnej funkcii sú opäť dve rekurzívne volania, preto sme naznačili, kde vzniknú deliace čiary. V tejto funkcii si ešte treba uvedomiť, čo sa stane v prípade, že neplatí `n > 0`: vtedy sa vykoná príkaz za `if`, teda `t.bk(d)`. Samotné vloženie práce so zásobníkom je veľmi podobné predchádzajúcemu príkladu:

```
import turtle
import stack

def strom(n, d):
    s = stack.Stack()
    s.push((1, n, d))
    while not s.is_empty():
        a, n, d = s.pop()
        if a == 1:
            # prva cast
            t.fd(d)
            if n > 0:
                t.lt(40)
                s.push((2, n, d))
                s.push((1, n-1, d*.7))
            else:
                t.bk(d)
        elif a == 2:
            # druha cast
            t.rt(75)
            s.push((3, n, d))
            s.push((1, n-1, d*.6))
        elif a == 3:
            # tretia cast
            t.lt(35)
            t.bk(d)

t = turtle.Turtle()
t.speed(0)
t.lt(90)
strom(5, 80)
```

Keď si na túto ideu odrekurzívnenia zvyknete, prídete na to, kde sa dá tento zápis ešte zjednodušiť, napr. nasledujúca verzia obsahuje menej vkladaní prvkov do zásobníka (namiesto niektorých `s.push()` sme priamo priradili do premenných, do ktorých sa predtým priraďovalo na začiatku `while`-cyklu):

```
def strom(n, d):
    s = stack.Stack()
    a = 1
    while True:
        if a == 1:
            # prva cast
            t.fd(d)
            if n > 0:
                t.lt(40)
                s.push((2, n, d))
                a, n, d = 1, n-1, d*.7
            else:
                t.bk(d)
                if s.is_empty():
```

```
                    break
                a, n, d = s.pop()
        elif a == 2:
            # druha cast
            t.rt(75)
            s.push((3, n, d))
            a, n, d = 1, n-1, d*.6
        elif a == 3:
            # tretia cast
            t.lt(35)
            t.bk(d)
            if s.is_empty():
                break
            a, n, d = s.pop()
```

20.2 Queue - rad, front

Dátová štruktúra **rad** (niekedy **front**, po anglicky **queue**) sa veľmi podobá zásobníku: tiež je to nejaká kolekcia údajov, s ktorou sa pracuje len pomocou týchto štyroch operácií:

- `enqueue(údaj)` - vloží požadovaný údaj na koniec radu
- `dequeue()` - vyberie prvý údaj z radu, ak bol rad prázdny, metóda spôsobí vyvolanie výnimky `EmptyError`
- `front()` - podobne ako metóda `dequeue()` vráti prvý prvok z radu, len tento prvok v rade ponechá
- `is_empty()` - zistí, či je rad prázdny

Takejto dátovej štruktúre hovoríme **spravodlivý rad**, lebo operáciou `dequeue()` sú odoberané tie prvky zo štruktúry, ktoré čakajú najdlhšie, prišli prvé. Označujeme to ako **FIFO** (first in, first out), teda "kto skôr príde, ten skôr melie".

Rad pomocou poľa

Dátovú štruktúru rad budeme realizovať v poli podobne ako sme realizovali zásobník: začiatok radu bude začiatok poľa (`self.pole[0]`), odoberať prvý prvok budeme pomocou metódy `pop(0)`, pridávať budeme na koniec poľa metódou `append(udaj)`. Už by sem mohli tušiť, že táto realizácia nebude najvhodnejšia: metóda `pop(0)` pre pythonovské pole (typ `list`) je dosť pomalá, ale zatiaľ s tým nebudeme nič robiť.

```
class EmptyError(Exception): pass

class Queue:
    def __init__(self):
        '''inicializácia vnútornej reprezentácie'''
        self.pole = []

    def is_empty(self):
        '''zistí, či je zásobník prázdny

        metóda vráti True alebo False
        '''
        return self.pole == []

    def enqueue(self, data):
        '''na koniec radu vloží novú hodnotu

        funkcia nevracia žiadnu hodnotu
        '''
```

```
        self.pole.append(data)

    def dequeue(self):
        '''vyberie zo začiatku radu prvú hodnotu

        prvok zo začiatku radu sa vráti ako výsledná hodnota funkcie
        rad sa pritom skráti o tento jeden prvok

        metóda vyvolá výnimku EmptyError pri prázdnom rade
        '''
        if self.is_empty():
            raise EmptyError('rad je prazdny')
        return self.pole.pop(0)                    # pomala operacia

    def front(self):
        '''vráti prvú hodnotu zo začiatku radu

        prvok zo začiatku radu sa vráti ako výsledná hodnota funkcie
        rad sa pritom neskracuje, ale ostáva v pôvodnom stave

        metóda vyvolá výnimku EmptyError pri prázdnom rade
        '''
        if self.is_empty():
            raise EmptyError('rad je prazdny')
        return self.pole[0]
```

Otestujme podobne, ako sme testovali zásobník: prečítame nejaký textový súbor, pritom riadky zapisujeme na koniec radu. Potom postupne vyberáme z radu tieto riadky a zapisujeme ich do iného textového súboru:

```
q = Queue()
with open('p.py') as subor:
    for riadok in subor:
        q.enqueue(riadok)

with open('test.py', 'w') as subor:
    while not q.is_empty():
        subor.write(q.dequeue())
```

Aj dátová štruktúra rad má v informatike veľmi dôležité využitie, ale to uvidíme až v ďalšom semestri.

KAPITOLA 21

Triedy a operácie 2.

Už vieme, že pri definovaní triedy môžeme využiť tieto magické metódy:

- `__init__()` - inicializácia inštancie (funkcia nič nevracia)
- `__repr__()` - znaková reprezentácia objektu (funkcia vráti reťazec)
- `__len__()` - počet prvkov (funkcia vráti celé číslo)
- `__contains__()` - zistí, či je `hodnota` prvkom (funkcia vráti `True` alebo `False`)
- `__getitem__()` - vráti hodnotu na danom indexe (funkcia niečo vráti)
- `__setitem__()` - zmení hodnotu na danom indexe (funkcia nič nevracia)
- `__delitem__()` - vyhodí hodnotu na danom indexe (funkcia nič nevracia)

Budeme pokračovať vo využívaní špeciálnych (magických) metód pri definovaní vlastných tried.

21.1 Trieda Zlomok

Pozrime triedu `Zlomok`, ktorú sme definovali na poslednom cvičení:

```
class Zlomok:
    def __init__(self, c=0, m=1):

        def nsd(a, b):
            while b > 0:
                a, b = b, a%b
            return a

        if m == 0:
            m = 1
        if m < 0:
            c, m = -c, -m
        delitel = nsd(c, m)
        self.citatel = c // delitel
        self.menovatel = m // delitel

    def __repr__(self):
        return '{}÷{}'.format(self.citatel, self.menovatel)

    def sucet(self, iny):
        a, b = self.citatel, self.menovatel
```

```
        if isinstance(iny, int):
            c, d = iny, 1
        else:
            c, d = iny.citatel, iny.menovatel
        return Zlomok(a*d + b*c, b*d)

    def rozdiel(self, iny):
        a, b = self.citatel, self.menovatel
        if isinstance(iny, int):
            c, d = iny, 1
        else:
            c, d = iny.citatel, iny.menovatel
        return Zlomok(a*d - b*c, b*d)

    def sucin(self, iny):
        a, b = self.citatel, self.menovatel
        if isinstance(iny, int):
            c, d = iny, 1
        else:
            c, d = iny.citatel, iny.menovatel
        return Zlomok(a*c, b*d)

    def podiel(self, iny):
        a, b = self.citatel, self.menovatel
        if isinstance(iny, int):
            c, d = iny, 1
        else:
            c, d = iny.citatel, iny.menovatel
        return Zlomok(a*d, b*c)

    def int(self):
        return self.citatel // self.menovatel

    def float(self):
        return self.citatel / self.menovatel

    def mensi(self, iny):
        a, b = self.citatel, self.menovatel
        if isinstance(iny, int):
            c, d = iny, 1
        else:
            c, d = iny.citatel, iny.menovatel
        return a*d < b*c

    def rovny(self, iny):
        a, b = self.citatel, self.menovatel
        if isinstance(iny, int):
            c, d = iny, 1
        else:
            c, d = iny.citatel, iny.menovatel
        return a*d == b*c
```

Pridali sme sem aj dve nové metódy pre matematické relácie: rovný a menší.

Ukážeme, ako prerobímu túto triedu tak, aby práca so zlomkami získala pythonovský štýl. Radi by sme sčitovali zlomky nie pomocou `x.sucet(y)` ale pekne ako `x+y`. Musíme pochopiť, ako Python spracováva všetky matematické (infixové) operácie:

- každý výskyt operácie +, napr. `x+y` sa automaticky nahradí pomocou `x.__add__(y)`, t.j. podľa typu prvého

operandu zavolá jeho metódu `__add__()` a ak táto neexistuje, hlási chybu

```
>>> Zlomok(1,2) + Zlomok(1,3)
...
TypeError: unsupported operand type(s) for +: 'Zlomok' and 'Zlomok'
```

- podobne je to so všetkými aritmetickými a logickými operáciami, môžete vyskúšať:

```
>>> 'Pyt' + 'hon'
'Python'
>>> 'Pyt'.__add__('hon')
'Python'
>>> 'Pyt' * 3
'PytPytPyt'
>>> 'Pyt'.__mul__(3)
'PytPytPyt'
>>> [3,4,5] + [6,7]
[3, 4, 5, 6, 7]
>>> [3,4,5].__add__([6,7])
[3, 4, 5, 6, 7]
>>> x = 11
>>> x + 310
321
>>> x.__add__(310)
321
```

V dokumentácii Pythonu by sme vedeli nájsť takúto tabuľku:

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.

Opravme metódy pre operácie so zlomkami:

```
class Zlomok:
    def __init__(self, c=0, m=1):

        def nsd(a, b):
            while b > 0:
                a, b = b, a%b
            return a

        if m == 0:
            m = 1
        if m < 0:
            c, m = -c, -m
        delitel = nsd(c, m)
        self.citatel = c // delitel
```

```
        self.menovatel = m // delitel

    def __repr__(self):
        return '{}÷{}'.format(self.citatel, self.menovatel)

    def __add__(self, iny):
        a, b = self.citatel, self.menovatel
        if isinstance(iny, int):
            c, d = iny, 1
        else:
            c, d = iny.citatel, iny.menovatel
        return Zlomok(a*d + b*c, b*d)

    __radd__ = __add__

    def __sub__(self, iny):
        a, b = self.citatel, self.menovatel
        if isinstance(iny, int):
            c, d = iny, 1
        else:
            c, d = iny.citatel, iny.menovatel
        return Zlomok(a*d - b*c, b*d)

    def __mul__(self, iny):
        a, b = self.citatel, self.menovatel
        if isinstance(iny, int):
            c, d = iny, 1
        else:
            c, d = iny.citatel, iny.menovatel
        return Zlomok(a*c, b*d)

    __rmul__ = __mul__

    def __truediv__(self, iny):
        a, b = self.citatel, self.menovatel
        if isinstance(iny, int):
            c, d = iny, 1
        else:
            c, d = iny.citatel, iny.menovatel
        return Zlomok(a*d, b*c)

    __floordiv__ = __truediv__

    def __int__(self):
        return self.citatel // self.menovatel

    def __float__(self):
        return self.citatel / self.menovatel

    def __lt__(self, iny):
        a, b = self.citatel, self.menovatel
        if isinstance(iny, int):
            c, d = iny, 1
        else:
            c, d = iny.citatel, iny.menovatel
        return a*d < b*c

    def __eq__(self, iny):
```

```
        a, b = self.citatel, self.menovatel
        if isinstance(iny, int):
            c, d = iny, 1
        else:
            c, d = iny.citatel, iny.menovatel
        return a*d == b*c
```

Niekedy by sa hodilo, aby Python volal príslušnú metódu nie podľa typu prvého operandu, ale podľa druhého, napr. tak, ako to robí pri násobení čísla a znakového reťazca:

```
>>> 'Pyt' * 3
'PytPytPyt'
>>> 'Pyt'.__mul__(3)
'PytPytPyt'
>>> 3 * 'Pyt'
'PytPytPyt'
>>> (3).__mul__('Pyt')
NotImplemented
```

V tomto prípade Python pri neúspešnom `(3).__mul__('Pyt')` sa pokúsi pohľadať verziu násobenia s vymeneným poradím operandov (tzv. reflected), t.j. skúsi

```
>>> 'Pyt'.__rmul__(3)
'PytPytPyt'
```

Pre nás to znamená, že niekedy treba definovať ešte aj metódy, keď zlomkom je druhý operand operácie a prvý je napr. celé číslo. Tieto "prevrátené" funkcie začínajú písmenom `r`, napr. `__radd__()`, `__rsub__()`, ...

Všimnite si, že v programe sme zapísali `__radd__ = __add__`, čo na tomto mieste označuje, že definujeme metódu (triedny atribút) `__radd__` s rovnakou hodnotou ako iná metóda (triedny atribút) `__add__`.

Podobne, ako sme definovali aritmetické operácie, definujeme aj relačné operácie porovnania:

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`

Ale aj niektoré štandardné funkcie:

metóda	relácia
`x.__abs__()`	`abs(x)`
`x.__int__()`	`int(x)`
`x.__float__()`	`float(x)`

polymorfizmus

Polymorfizmus je jednou zo základných vlastností jazyka, ktorý umožňuje objektové programovanie (popri zapuzdrení a dedičnosti). Zjednodušene označuje:

- objektom, ktoré sú odvodené z rôznych tried, môžeme volať tú istú metódu s rovnakým významom v kontexte ich triedy

  - napr. metódu `__len__()` môžeme zavolať pre objekty typu `str`, `list`, `dict`, prípadne pre našu triedu, ak sme ju definovali, pre Python je to potom volanie `len(instancia)` - vždy to označuje počet prvkov, ale závisí to od konkrétneho typu

- preťažovanie operátorov, teda vykonanie rôznych operácií v závislosti od typov operandov (overloading)
  - napr. operácia `+` sa vykoná v závislosti od typov operandov, zakaždým sa zavolá iná metóda pre `3.14+7`, `'a'+'b'`, `[1]+[2]`, `Zlomok(1,2)+Zlomok(1,3)`
- funkcia môže pracovať s parametrami rôznych typov (parametrický polymorfismus)
  - v Pythone je to veľmi prirodzený spôsob, napr.

```
def sucet(pole):
    vysl = pole[0]
    for prvok in pole[1:]:
        vysl += prvok
    return vysl
```

táto funkcia funguje nielen pre pole čísel, ale aj pole reťazcov, pole polí ale aj volanie `range()`

Keď už máme dobre definované metódy triedy `Zlomok`, môžeme zapísať:

```
>>> pole = []
>>> for i in range(1,6):
        for j in range(1,6):
            pole.append(Zlomok(i,j))

>>> pole
[1÷1, 1÷2, 1÷3, 1÷4, 1÷5, 2÷1, 1÷1, 2÷3, 1÷2, 2÷5, 3÷1, 3÷2, 1÷1,
 3÷4, 3÷5, 4÷1, 2÷1, 4÷3, 1÷1, 4÷5, 5÷1, 5÷2, 5÷3, 5÷4, 1÷1]
>>> min(pole)
1÷5
>>> sum(pole)
137÷4
>>> sorted(pole)
[1÷5, 1÷4, 1÷3, 2÷5, 1÷2, 1÷2, 3÷5, 2÷3, 3÷4, 4÷5, 1÷1, 1÷1, 1÷1,
 1÷1, 1÷1, 5÷4, 4÷3, 3÷2, 5÷3, 2÷1, 2÷1, 5÷2, 3÷1, 4÷1, 5÷1]
```

Všimnite si, že tu fungujú štandardné funkcie `min()`, `sum()`, `sorted()`, ktoré sme ale my neprogramovali. Našťastie funguje polymorfizmus: keďže vo vnútri štandardných funkcií sa pre daný parameter využívajú len operácie `__add__()` a `__lt__()`, fungujú aj pre typ `Zlomok`.

21.2 Trieda množina

Na poslednom cvičení sme definovali aj triedu `Mnozina`, ktorá je "množinou čísel". Využili sme pritom asociatívne pole (`dict`):

```
class Mnozina:

    def __init__(self, inic=None):
        self.pole = {}
        if inic:
            for i in inic:
                self.pridaj(i)

    def __repr__(self):
```

```
        return repr(tuple(sorted(self.pole)))

    def pridaj(self, cislo):
        self.pole[cislo] = None

    def vyhod(self, cislo):
        del self.pole[cislo]

    def zisti(self, cislo):
        return cislo in self.pole

    def pocet(self):
        return len(self.pole)

    def min(self):
        return min(self.pole)

    def max(self):
        return max(self.pole)

    def zjednotenie(self, ina):
        for i in ina.pole:
            self.pridaj(i)

    def prienik(self, ina):
        pom, self.pole = self.pole, {}
        for i in ina.pole:
            if i in pom:
                self.pridaj(i)

    def rozdiel(self, ina):
        for i in ina.pole:
            if i in self.pole:
                self.vyhod(i)
```

Pomocou takto definovanej dátovej štruktúry môžeme napr. zostaviť funkciu na vytvorenie Eratostenovho sita (http://sk.wikipedia.org/wiki/Eratostenovo_sito):

- zoberieme zoznam všetkých celých čísel od 2 do nejakého zadaného `n`
- ďalej budeme opakovať toto: prvé číslo v zozname je prvočíslo, vyberieme ho a zo zoznamu odstránime všetky jeho násobky, teda:
- najprv vyberieme 2 (prvé prvočíslo) a vyhodíme zo zoznamu všetky násobky 2, t.j. všetky párne čísla
- potom vyberieme 3 (druhé prvočíslo) a vyhodíme všetky jeho násobky
- potom vyberieme 5 (tretie prvočíslo) a vyhodíme všetky jeho násobky
- ...
- takto sa pokračuje, kým v zozname ešte ostali nejaké čísla (maximálne do `n`, skutočnosti by stačili do odmocniny `n`)

Zapíšme to ako funkciu:

```
def eratostenovo_sito(n):
    mnoz = Mnozina(range(2, n+1))
    for i in range(2, n+1):
        if i in mnoz:
            nasobky = Mnozina(range(2*i, n+1, i))
```

```
            mnoz.rozdiel(nasobky)
    return mnoz
```

```
>>> 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)
```

Ak potrebujeme v Pythone pracovať s množinami, nemusíme si na to definovať vlastnú triedu, môžeme využiť štandardný typ `set`, ktorý robí presne to, čo by sme očakávali aj od bežnej matematickej množiny.

21.3 Štandardný typ set

Štandardný Pythonovský typ **set** má kompletnú sadu množinových operácií a veľa užitočných metód. Podobne ako pre typ asociatívne pole (`dict`) aj pre množiny platí, že prvkami nemôžu byť ľubovoľné hodnoty, ale musia to byť nemenné typy (immutable), napr. čísla, reťazce, n-tice.

Operácie a metódy s množinami

pre množiny `M`, `M1` a `M2`:

	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

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
`sorted(M)`	vráti usporiadaný zoznam prvkov (ale všetky prvky sa musia dať navzájom porovnávať)

Niektoré metódy

(je ich oveľa viac):

	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)

Vytvorenie množiny

	popis
`M = set()`	vytvorí prázdnu množinu
`M = {hodnota,hodnota,...}`	vytvorí neprázdnu množinu so zadanými prvkami
`M = set(pole)`	so zadaného poľa vytvorí množinu
`M = set(M1)`	vytvorí kópiu množiny `M1`

Príklady

Funkcia vráti počet rôznych samohlások v danom slove:

```
def pocet_samohlasok(slovo):
    return len(set(slovo) & set('aeiouy'))
```

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 v§ačšie ako zadané `n`:

```
def urob(n):
    m = {1}
    for i in range(n+1):
        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:

```
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 Eratostenovho sita presne tak, ako sme to robili pomocou našej triedy `Mnozina`:

```
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
```

Python nezaručuje, že prvky v množine sú v rastúcej postupnosti. Preto niekedy množinu prevedieme na usporiadané pole

```
>>> 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]
```

KAPITOLA 22

Animovaný obrázok

Postupne vytvoríme takúto aplikáciu:

- v grafickej ploche `tkinter` vytvoríme niekoľko rôznych animovaným obrázkov, ktoré sa budú po ploche pohybovať rôznou rýchlosťou a rôznymi smermi

22.1 Trieda Anim

Animáciu budeme zabezpečovať cyklickým striedaním fáz - grafických objektov `image`. Predpokladáme, že každú fázu animácie máme uloženú v jednom samostatnom súbore:

Všetky obrázky z projektu si môžete stiahnuť zo súboru `anim.zip`

```
import tkinter

class Anim:
    canvas = None

    def __init__(self, meno, n, x, y):
        self.x, self.y = x, y
        self.obr = []
        for i in range(n):
            self.obr.append(tkinter.PhotoImage(file=meno+str(i)+'.png'))
        self.faza = 0
        self.id = self.canvas.create_image(self.x, self.y, image=self.
↪obr[self.faza])

    def posun(self):
        self.faza = (self.faza+1) % len(self.obr)
        self.canvas.itemconfig(self.id, image=self.obr[self.faza])
```

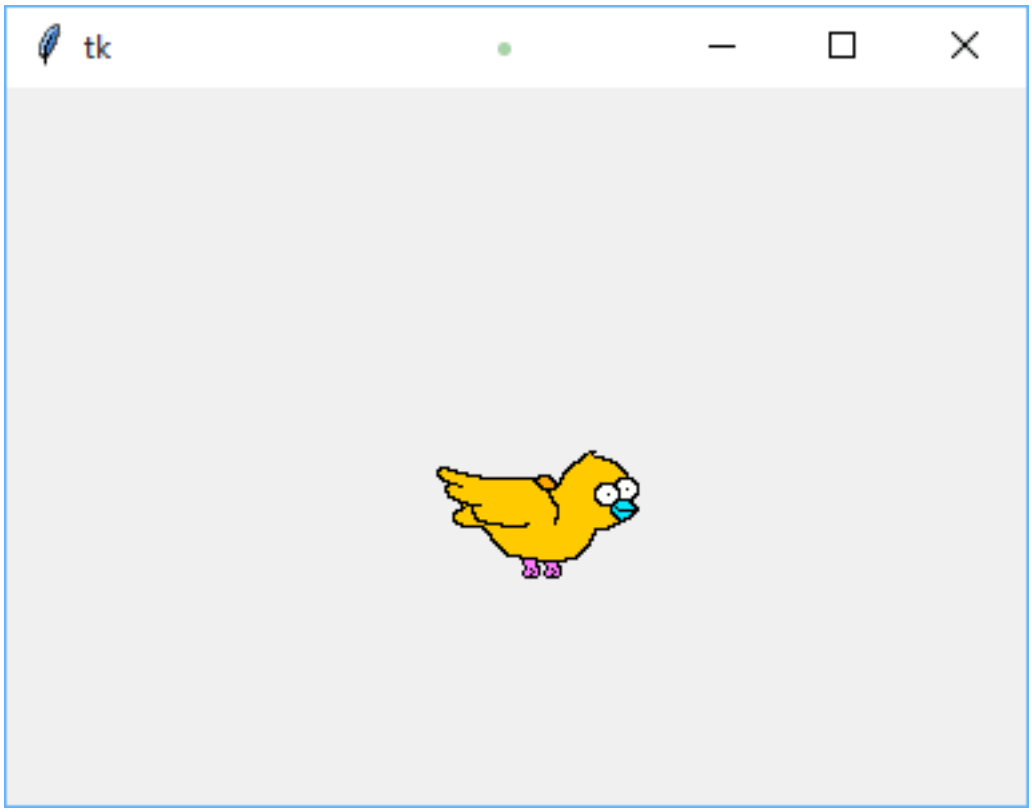
V tejto triede si všimnite:

- trieda obsahuje jeden triedny atribút `canvas`, ktorý bude spoločný pre všetky inštancie vytvorené z tejto triedy; na začiatku má hodnotu `None`, ale keď vytvoríme grafické plochu, nesmieme ju sem zabudnúť priradiť

- do atribútu `self.obr` sme priradili pole všetkých obrázkov, ktoré tvoria fázy animácie
- atribút `self.id` obsahuje identifikačný kód pre `tkinter` obrázkového objektu, vďaka nemu budeme vedieť nielen meniť fázy obrázku (pomocou `canvas.itemconfig()`) ale neskôr tento budeme môcť aj posúvať (pomocou `canvas.coords()`)

Grafická plocha

Pridajme grafickú plochu aj s časovačom:

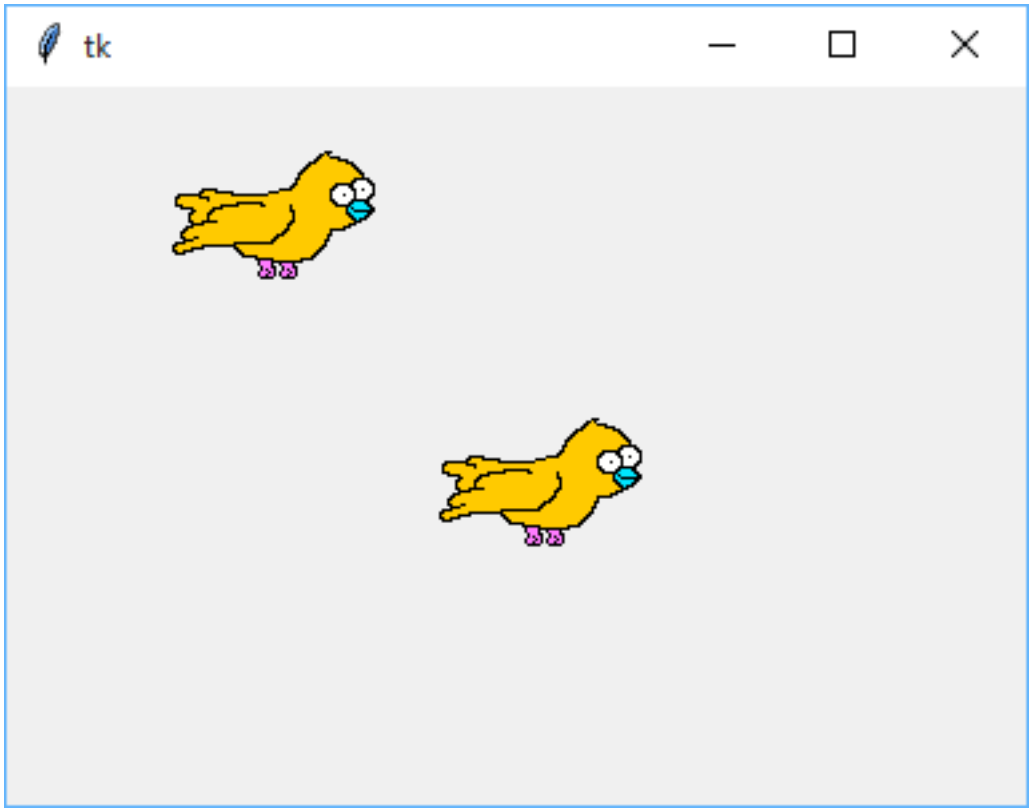



```
canvas = tkinter.Canvas()
canvas.pack()
Anim.canvas = canvas
a = Anim('a1/vtak', 8, 200, 150)

def timer():
    a.posun()
    canvas.after(100, timer)

timer()
```

Všimnite si:

- pomocou `Anim.canvas = canvas` sme do triedneho atribútu `canvas` priradili referenciu na grafickú plochu, vďaka čomu hneď ďalší príkaz `a = Anim(...)` vytvorí animovaný objekt a hneď ho aj zobrazí v grafickej ploche
- časovač volá pre pre animovaný objekt `a` metódu `posun()` každých 100 milisekúnd, t.j. 10-krát za sekundu

Pridáme ďalší animovaný objekt, oba objekty sú teraz uložené v premennej `pole` a časovač teraz hýbe oba objekty naraz:




```
canvas = tkinter.Canvas()
canvas.pack()
Anim.canvas = canvas
pole = [Anim('a1/vtak', 8, 200, 150),
        Anim('a1/vtak', 8, 100, 50)]

def timer():
    for a in pole:
        a.posun()
    canvas.after(100, timer)

timer()
```

Pridajme nejaký obrázok pozadia a vytváranie animovaných objektov kliknutím do plochy:


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

im = tkinter.PhotoImage(file='jazero.png')
canvas['width'] = im.width()
canvas['height'] = im.height()
canvas.create_image(0, 0, image=im, anchor="nw")

pole =[]

def klik(event):
    pole.append(Anim('a1/vtak', 8, event.x, event.y))

def timer():
    for a in pole:
        a.posun()
    canvas.after(100, timer)

canvas.bind('<Button-1>', klik)
timer()
```

Všimnite si, koľko nám tu teraz vzniklo **globálnych** premenných. Okrem toho:

- najprv sme vytvorili `canvas = tkinter.Canvas()` ešte neznámej veľkosti
- potom sme prečítali obrázok pozadia zo súboru `'jazero.png'`, zistili sme jeho rozmery a na základe nich sme zmenili aj rozmery grafickej plochy

- pri vykresľovaní pozadia pomocou `create_image()` sme využili pomenovaný parameter `anchor`, vďaka čomu nemusíme zadávať súradnice stredu obrázku, ale ľavý horný roh (`'nw'` označuje "severo-západ", t.j. ľavý horný roh)
- udalosť kliknutie do grafickej plochy zavolá funkciu `klik()` s parametrom `event`, v ktorom sa dozvieme súradnice kliknutého bodu

22.2 Trieda Plocha

Spôsob tvorby programov, pri ktorom máme všetko dôležité vo veľa globálnych premenných, je veľmi amatérsky - všetky tieto globálne premenné a funkcie **zapuzrime** do novej triedy `Plocha`:

```
class Plocha:
    def __init__(self):
        self.canvas = tkinter.Canvas()
        self.canvas.pack()
        Anim.canvas = self.canvas

        self.im = tkinter.PhotoImage(file='jazero.png')
        self.canvas['width'] = self.im.width()
        self.canvas['height'] = self.im.height()
        self.canvas.create_image(0, 0, image=self.im, anchor="nw")

        self.pole =[]
        self.canvas.bind('<Button-1>', self.klik)
        self.timer()

    def klik(self, event):
        self.pole.append(Anim('a1/vtak', 8, event.x, event.y))

    def timer(self):
        for a in self.pole:
            a.posun()
        self.canvas.after(100, self.timer)

p = Plocha()
```

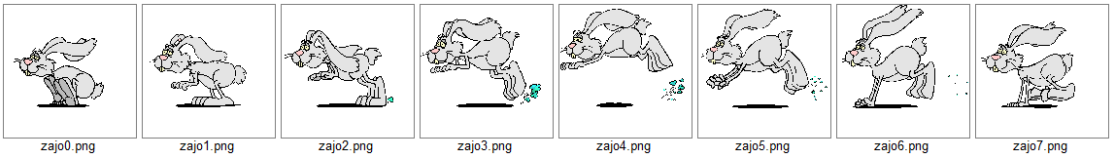
Funkčnosť tohto programu je úplne rovnaká, ako predchádzajúceho s globálnymi premennými. V tejto novej verzii máme jedinú globálnu premennú, ktorou je inštancia `p` celého projektu.

Vyskúšajte po spustení projektu a naklikaní niekoľkých objektov z shelli zadať:

```
>>> del p.im
>>>
```

Takto sa ukáže dôležitá vlastnosť `tkinter`: všetky obrázky, ktoré sa zobrazili v grafickej ploche (napr. pomocou `canvas.create_image()`), sa musia nachádzať aj v nejakej premennej (musí na ne odkazovať nejaká referencia), lebo ak sa obrázok (referencia) zruší, `tkinter` ho ďalej nedokáže zobrazovať. Treba si dať pozor, aby takéto obrázky neboli len v lokálnych premenných funkcií (metód), ale aby sa uchovali buď v globálnych alebo atribútoch objektu.

Pridajme ďalší animovaný obrázok:



Táto séria obrázkov sa tiež skladá z 8 fáz, preto vytvorenie obrázku bude veľmi podobné predchádzajúcemu. Do metódy `klik()` pridáme náhodné rozhodovanie sa, či sa vytvorí animovaný objekt s vtáčikom alebo zajacom:

```
from random import randint as ri

...

class Plocha:
    def __init__(self):
        Anim.canvas = self.canvas = tkinter.Canvas()
        self.canvas.pack()

        self.im = tkinter.PhotoImage(file='jazero.png')
        self.canvas['width'] = self.im.width()
        self.canvas['height'] = self.im.height()
        self.canvas.create_image(0, 0, image=self.im, anchor="nw")

        self.pole = []
        self.canvas.bind('<Button-1>', self.klik)
        self.timer()

    def klik(self, event):
        if ri(0, 1):
            self.pole.append(Anim('a1/vtak', 8, event.x, event.y))
        else:
            self.pole.append(Anim('a2/zajo', 8, event.x, event.y))

    def timer(self):
        for a in self.pole:
            a.posun()
        self.canvas.after(100, self.timer)

p = Plocha()
```

Vďaka tomuto vieme vytvoriť veľké množstvo animovaných objektov, ktoré sa animujú v grafickej ploche. Len si treba uvedomiť, že každý jeden takto vytvorený objekt sa skladá z 8 obrázkov a teda každý objekt si v svojom atribúte `self.obr` pamätá 8 rôznych obrázkov (objektov `PhotoImage`). Lenže takto môžeme veľmi rýchlo minúť pamäť systému určenú na obrázky a systém sa buď spomalí alebo spadne.

Lenže všetky objekty s vtáčikom majú tých istých 8 obrázkov (podobne aj zajace). Prerobíme program tak, že nie každý objekt `Anim` si bude vyrábať svoje vlastné pole `self.obr`, ale v triede `Plocha` si dopredu vytvoríme dve polia pre fázy vtáčika aj zajaca. Samotný Objekt `Anim` potom dostane len referenciu na toto pole:

```
import tkinter
from random import randint as ri

class Anim:
    canvas = None

    def __init__(self, obr, x, y):
        self.x, self.y = x, y
        self.obr = obr
```

```
        self.faza = 0
        self.id = self.canvas.create_image(self.x, self.y, image=self.
↪obr[self.faza])

    def posun(self):
        self.faza = (self.faza+1) % len(self.obr)
        self.canvas.itemconfig(self.id, image=self.obr[self.faza])

class Plocha:
    def __init__(self):
        Anim.canvas = self.canvas = tkinter.Canvas()
        self.canvas.pack()

        self.im = tkinter.PhotoImage(file='jazero.png')
        self.canvas['width'] = self.im.width()
        self.canvas['height'] = self.im.height()
        self.canvas.create_image(0, 0, image=self.im, anchor="nw")

        self.obr1 = []
        for i in range(8):
            self.obr1.append(tkinter.PhotoImage(file='a1/vtak'+str(i)+'.png
↪'))

        self.obr2 = []
        for i in range(8):
            self.obr2.append(tkinter.PhotoImage(file='a2/zajo'+str(i)+'.png
↪'))

        self.pole = []
        self.canvas.bind('<Button-1>', self.klik)
        self.timer()

    def klik(self, event):
        if ri(0, 1):
            self.pole.append(Anim(self.obr1, event.x, event.y))
        else:
            self.pole.append(Anim(self.obr2, event.x, event.y))

    def timer(self):
        for a in self.pole:
            a.posun()
        self.canvas.after(100, self.timer)

p = Plocha()
```

Tento projekt má rovnakú funkčnosť ako predchádzajúca verzia, len sa výrazne znížila pamäťová náročnosť ukladaných obrázkov.

22.3 Pohyb objektov

Ďalším krokom vo vylepšovaní tohto projektu bude pohyb obrázkov po ploche. Aby pobyhujúce sa objekty neušli z grafickej plochy, zabezpečíme nejaký mechanizmus správania sa na obvode plochy. Každý objekt bude mať svoj vektor pohybu `(dx,dy)`, ktorý bude označovať zmenu `x`-ovej aj `y`-ovej súradnice pri každom tiknutí časovača. Do každá inštancia `Anim` dostane ešte dva nové atribúty `dx` a `dy` a tiež do triedy `Anim` pridáme dva nové triedne atribúty (budú spoločné pre všetky inštancie) `sirka` a `vyska`, aby sme zabezpečili hraničné správanie pohybujúcich sa objektov:

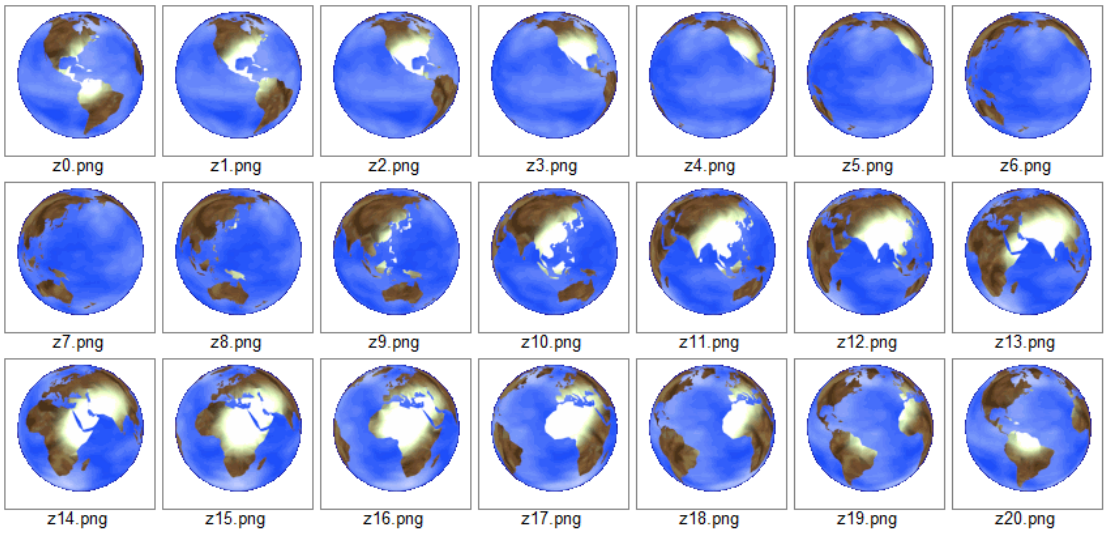
ak nejaký objekt prejde cez niektorú hranicu grafickej plochy, objaví sa na opačnom konci plochy a ďalej pokračuje vo svojom pohybe.

Musíme zmeniť aj metódu `posun()`, ktorá teraz zabezpečí aj zmenu súradníc objektu. Okrem toho aj pri vytváraní objektu vygenerujeme pre každý objekt nejaké `dx` a `dy`:

```
import tkinter
from random import randint as ri

class Anim:
    canvas = None
    sirka = None
    vyska = None

    def __init__(self, obr, x, y, dx=0, dy=0):
        self.x, self.y = x, y
        self.dx, self.dy = dx, dy
        self.obr = obr
        self.faza = 0
        self.id = self.canvas.create_image(self.x, self.y, image=self.
↪obr[self.faza])

    def posun(self):
        self.faza = (self.faza+1) % len(self.obr)
        self.canvas.itemconfig(self.id, image=self.obr[self.faza])
        self.x += self.dx
        if self.x < 0:
            self.x += self.sirka
        if self.x >= self.sirka:
            self.x -= self.sirka
        self.y += self.dy
        if self.y < 0:
            self.y += self.vyska
        if self.y >= self.vyska:
            self.y -= self.vyska
        self.canvas.coords(self.id, self.x, self.y)

class Plocha:
    def __init__(self):
        Anim.canvas = self.canvas = tkinter.Canvas()
        self.canvas.pack()

        self.im = tkinter.PhotoImage(file='jazero.png')
        Anim.sirka = self.canvas['width'] = self.im.width()
        Anim.vyska = self.canvas['height'] = self.im.height()
        self.canvas.create_image(0, 0, image=self.im, anchor="nw")

        self.obr1 = []
        for i in range(8):
            self.obr1.append(tkinter.PhotoImage(file='a1/vtak'+str(i)+'.png
↪'))

        self.obr2 = []
        for i in range(8):
            self.obr2.append(tkinter.PhotoImage(file='a2/zajo'+str(i)+'.png
↪'))

        self.pole = []
        self.canvas.bind('<Button-1>', self.klik)
```

```
        self.timer()

    def klik(self, event):
        if ri(0, 1):
            self.pole.append(Anim(self.obr1, event.x, event.y, 6, -1))
        else:
            self.pole.append(Anim(self.obr2, event.x, event.y, -4, 1))

    def timer(self):
        for a in self.pole:
            a.posun()
        self.canvas.after(100, self.timer)

p = Plocha()
```

Na to, aby sme pridali ďalší animovaný objekt zemeguľu s 21 fázami, stačí do plochy pridať pole `self.obr3` s prečítanými obrázkami a do metódy `klik()` pridáme náhodné generovanie tejto animácie:




```
class Plocha:
    def __init__(self):
        ...

        self.obr3 = []
        for i in range(21):
            self.obr3.append(tkinter.PhotoImage(file='a3/z'+str(i)+'.png'))

        ...

    def klik(self, event):
        t = ri(0, 2)
        if t == 0:
            self.pole.append(Anim(self.obr1, event.x, event.y, 6, -1))
        elif t == 1:
            self.pole.append(Anim(self.obr2, event.x, event.y, -4, 1))
        else:
            self.pole.append(Anim(self.obr3, event.x, event.y))
```

Vidíme, že objekt zemeguľa sa krúti na mieste (`dx` aj `dy` sme nastavili na 0).

22.4 Ďalší typ pohybu

Predpokladajme, že chceme, aby sa objekt zemeguľa hýbal podľa iných pravidiel: chceme, aby sa na hranici grafickej plochy odrážal, ako gulečníková guľa. Zabezpečíme to tak, že na okraji plochy nebudeme meniť aktuálnu polohu `x` a `y`, ale zmeníme vektor `(dx,dy)`. Keďže chceme, aby sa takto správal iba jeden z objektov, zmeníme iba jemu metódu `posun()`, ostatné objekty budú mať svoju pôvodnú metódu `posun()`. Najjednoduchšie to urobíme, tak, že Zemeguľa nebude odvodená z triedy `Anim`, ale z nejakej inej, napr. `AnimOdraz`. Keďže obe tieto triedy sú veľmi podobné a majú rovnakú inicializáciu `__init__()`, túto novú triedu odvodíme z `Anim`, t.j. stane sa potomkom tejto triedy a zdedí všetko, čo neprekryjeme novou verziou (metódu `posun()`):

```
import tkinter
from random import randint as ri

class Anim:
    canvas = None
    sirka = None
    vyska = None

    def __init__(self, obr, x, y, dx=0, dy=0):
        self.x, self.y = x, y
        self.dx, self.dy = dx, dy
        self.obr = obr
        self.faza = 0
        self.id = self.canvas.create_image(self.x, self.y, image=self.
↪obr[self.faza])

    def posun(self):
        self.faza = (self.faza+1) % len(self.obr)
        self.canvas.itemconfig(self.id, image=self.obr[self.faza])
        self.x += self.dx
        if self.x < 0:
            self.x += self.sirka
        if self.x >= self.sirka:
            self.x -= self.sirka
        self.y += self.dy
        if self.y < 0:
            self.y += self.vyska
        if self.y >= self.vyska:
            self.y -= self.vyska
        self.canvas.coords(self.id, self.x, self.y)

class AnimOdraz(Anim):
    def posun(self):
        self.faza = (self.faza+1) % len(self.obr)
        self.canvas.itemconfig(self.id, image=self.obr[self.faza])
        self.x += self.dx
        if self.x < 50:
            self.dx = abs(self.dx)
        if self.x >= self.sirka-50:
            self.dx = -abs(self.dx)
        self.y += self.dy
        if self.y < 50:
            self.dy = abs(self.dy)
        if self.y >= self.vyska-50:
            self.dy = -abs(self.dy)
        self.canvas.coords(self.id, self.x, self.y)
```

```
class Plocha:
    def __init__(self):
        Anim.canvas = self.canvas = tkinter.Canvas()
        self.canvas.pack()

        self.im = tkinter.PhotoImage(file='jazero.png')
        Anim.sirka = self.canvas['width'] = self.im.width()
        Anim.vyska = self.canvas['height'] = self.im.height()
        self.canvas.create_image(0, 0, image=self.im, anchor="nw")

        self.obr1 = []
        for i in range(8):
            self.obr1.append(tkinter.PhotoImage(file='a1/vtak'+str(i)+'.png
↪'))

        self.obr2 = []
        for i in range(8):
            self.obr2.append(tkinter.PhotoImage(file='a2/zajo'+str(i)+'.png
↪'))

        self.obr3 = []
        for i in range(21):
            self.obr3.append(tkinter.PhotoImage(file='a3/z'+str(i)+'.png'))

        self.pole = []
        self.canvas.bind('<Button-1>', self.klik)
        self.timer()

    def klik(self, event):
        t = ri(0, 2)
        if t == 0:
            self.pole.append(Anim(self.obr1, event.x, event.y,
                                  ri(0, 6), ri(-3, 3)))
        elif t == 1:
            self.pole.append(Anim(self.obr2, event.x, event.y,
                                  ri(-6, -1), ri(-1, 1)))
        else:
            self.pole.append(AnimOdraz(self.obr3, event.x, event.y,
                                       ri(-6, 6), ri(-6, 6)))

    def timer(self):
        for a in self.pole:
            a.posun()
        self.canvas.after(100, self.timer)

p = Plocha()
```

Všimnite si, že oproti predchádzajúcej verzii sa skoro nič nezmenilo, len

- pridali sme novú triedu `AnimOdraz`, ktorá je odvodená od `Anim`, v tejto triede sme zadefinovali metódu `posun()`, ktorá týmto prekryla pôvodnú metódu: objekty typu `Anim` majú teda pôvodnú `posun()`, objekty typu `AnimOdraz` majú už novú verziu
- všetkým typom objektov sme nastavili nejaké náhodné hodnoty pre `dx` a `dy`

22.5 Plánovač

Teraz vylepšíme rýchlosť animácií rôznych objektov. Doterajšia verzia programu animovala všetky objekty rovnakou rýchlosťou (100 milisekúnd), pričom my by sme možno chceli, aby sa zemeguľa krútila najrýchlejšie (napr. s intervalom časovača 10), vtáčiky by sa mohli animovať rýchlosťou medzi 30 a 130 (niektoré budú veľmi rýchle, iné budú pomalšie) a zajace by mohli byť najpomalšie (ich interval medzi 100 a 160). Tak ako je to naprogramované teraz, to fungovať nebude. Asi by sme pre každý objekt mohli vytvoriť samostatný časovač s konkrétnym jeho intervalom milisekúnd - toto je ale veľmi neefektívne a pre systém zaťažujúce (100 časovačov sú pre systém náročné).

Využijeme inú ideu:

- budeme mať jediný časovač, ktorý ale bude tikať rýchlejšie ako doteraz (napr. 10 milisekúnd)
- časovač sa pozrie na všetky animované objekty, či už im neuplynul čas, kedy malo nastať ich animovanie (každý objekt má svoj vlastný interval, budeme ho volať `tik`)
- všetky objekty, ktorým už nastal ich čas, zanimuje (zavolá ich `posun()`) a znovu im **naplánuje**, kedy by sa mali animovať nabudúce
- aby sme nemuseli pri každom tiknutí časovača kontrolovať všetky objekty (môžu ich byť tisíce), zoradíme si ich do poradia, v ktorom na začiatku sú tie, ktoré treba vybaviť čo najskôr a na konci tie neskoršie = tomuto hovoríme **plánovač** (niekedy aj plánovací kalendár) a využijeme nám známu štruktúru rad (queue), ktorú mierne vylepšíme:
- do takéhoto radu pridáme novú metódu `insert()`, ktorá zaradí nový objekt na správne miesto, t.j. zaradí za tie objekty, ktoré sa majú animovať skôr a predbehne všetky tie, ktoré majú neskorší čas animácie
- takémuto radu hovoríme **prioritný rad** (zaraďujeme podľa nejakej priority)

Pôvodná verzia `Queue` (z 20. prednášky):

```
class EmptyError(Exception): pass

class Queue:
    def __init__(self):
        self.pole = []

    def is_empty(self):
        return self.pole == []

    def enqueue(self, data):
        self.pole.append(data)

    def dequeue(self):
        if self.is_empty():
            raise EmptyError('rad je prazdny')
        return self.pole.pop(0)

    def front(self):
        if self.is_empty():
            raise EmptyError('rad je prazdny')
        return self.pole[0]
```

Táto realizácia pridáva (metóda `enqueue()`) na koniec poľa a prvky odoberá (metóda `dequeue()`) zo začiatku.

Keďže pridávať nebudeme pomocou metódy `enqueue()`, ale pomocou novej metódy `insert()`, vytvoríme novú odvodenú triedu `PriorityQueue`, ktorá bude obsahovať túto novú metódu a zároveň "zablokuje" pôvodnú metódu `enqueue()`

Túto definíciu uložíme do súboru `mojequeue.py`:

```
import time

class EmptyError(Exception): pass

class Queue:
    def __init__(self):
        self.pole = []

    def is_empty(self):
        return self.pole == []

    def enqueue(self, data):
        self.pole.append(data)

    def insert(self, tik, data):
        cas = time.time() + tik/1000
        self.pole.append(0)
        i = len(self.pole)-2
        while i>= 0 and self.pole[i][0] > cas:
            self.pole[i+1] = self.pole[i]
            i -= 1
        self.pole[i+1] = (cas, data)

    def dequeue(self):
        if self.is_empty():
            raise EmptyError('rad je prazdny')
        return self.pole.pop(0)

    def front(self):
        if self.is_empty():
            raise EmptyError('rad je prazdny')
        return self.pole[0]

class PriorityQueue(Queue):
    def enqueue(self, data):
        raise AttributeError('pre prioritny rad nefunguje metoda enqueue')

    def insert(self, tik, data):
        cas = time.time() + tik/1000
        self.pole.append(0)
        i = len(self.pole)-2
        while i>= 0 and self.pole[i][0] > cas:
            self.pole[i+1] = self.pole[i]
            i -= 1
        self.pole[i+1] = (cas, data)
```

Vysvetlenie k tejto novej metóde:

- najprv sa zistí, kedy presne sa bude najbližšie animovať tento objekt: k momentálnemu času `time.time()` sa pripočíta počet milisekúnd (keďže čas `time.time()` je v sekundách, `tik` treba previesť na sekundy delením 1000)
- pole sa nafúkne o jeden prvok
- ďalej prechádzame toto pole od konca: kým sú časy objektov v poli väčšie (neskoršie), posúvame ich v poli o jeden prvok ďalej
- na takto vzniknuté voľné miesto vložíme nový objekt aj s jeho plánovaným časom

Do samotného modulu urobíme tieto zmeny:

- pridali sme `import` pre moduly `time` a `mojequeue`
- inicializácia objektu `Anim` - pridali sme parameter aj atribút `self.tik`, ktorý označuje počet milisekúnd pri animovaní fáz
- v inicializácii triedy `Plocha` sme vytvorili (zatiaľ prázdny) prioritný rad `self.queue`
- keďže už ďalej nebudeme potrebovať pole všetkých objektov (`self.pole`) môžeme ho zrušiť (hoci, ak plánujete posúvať objekty ťahaním myšou, toto pole sa vám môže zísť)
- do metódy `klik()`, v ktorej vyrábame všetky animované objekty, pridáme náhodné generovanie frekvencie animovania objektu (posledný parameter `tik` inicializácie objektu) a tiež prvé vloženie práve vytvoreného objektu do prioritného radu `self.queue`
- zmeníme časovač, t.j. metódu `timer()`:
- pozrieme v prioritnom rade prvý prvok a ak je jeho čas, kedy sa mal spracovať menší ako momentálny čas, vyberieme ho (`dequeue()`) a zavoláme jeho `posun()`
- zároveň naplánujeme jeho znovu posúvanie v budúcnosti (zaradíme ho na správne miesto do `self.queue`)
- toto opakujeme pre všetky objekty zo začiatku radu, ktorým už uplynul ich čas
- ešte nesmieme zabudnúť zmeniť interval časovača, t.j. konštantu pri volaní `after()`

Výsledný projekt:

```
import tkinter
from random import randint as ri
import time
import mojequeue

class Anim:
    canvas = None
    sirka = None
    vyska = None

    def __init__(self, obr, x, y, dx=0, dy=0, tik=100):
        self.x, self.y = x, y
        self.dx, self.dy = dx, dy
        self.tik = tik
        self.obr = obr
        self.faza = 0
        self.id = self.canvas.create_image(self.x, self.y, image=self.
↪obr[self.faza])

    def posun(self):
        self.faza = (self.faza+1) % len(self.obr)
        self.canvas.itemconfig(self.id, image=self.obr[self.faza])
        self.x += self.dx
        if self.x < 0:
            self.x += self.sirka
        if self.x >= self.sirka:
            self.x -= self.sirka
        self.y += self.dy
        if self.y < 0:
            self.y += self.vyska
        if self.y >= self.vyska:
            self.y -= self.vyska
        self.canvas.coords(self.id, self.x, self.y)

class AnimOdraz(Anim):
```

```
    def posun(self):
        self.faza = (self.faza+1) % len(self.obr)
        self.canvas.itemconfig(self.id, image=self.obr[self.faza])
        self.x += self.dx
        if self.x < 50:
            self.dx = abs(self.dx)
        if self.x >= self.sirka-50:
            self.dx = -abs(self.dx)
        self.y += self.dy
        if self.y < 50:
            self.dy = abs(self.dy)
        if self.y >= self.vyska-50:
            self.dy = -abs(self.dy)
        self.canvas.coords(self.id, self.x, self.y)

class Plocha:
    def __init__(self):
        Anim.canvas = self.canvas = tkinter.Canvas()
        self.canvas.pack()

        self.im = tkinter.PhotoImage(file='jazero.png')
        Anim.sirka = self.canvas['width'] = self.im.width()
        Anim.vyska = self.canvas['height'] = self.im.height()
        self.canvas.create_image(0, 0, image=self.im, anchor="nw")

        self.obr1 = []
        for i in range(8):
            self.obr1.append(tkinter.PhotoImage(file='a1/vtak'+str(i)+'.png
↪'))

        self.obr2 = []
        for i in range(8):
            self.obr2.append(tkinter.PhotoImage(file='a2/zajo'+str(i)+'.png
↪'))

        self.obr3 = []
        for i in range(21):
            self.obr3.append(tkinter.PhotoImage(file='a3/z'+str(i)+'.png'))

        self.queue = mojequeue.PriorityQueue()
        self.canvas.bind('<Button-1>', self.klik)
        self.timer()

    def klik(self, event):
        t = ri(0, 2)
        if t == 0:
            a = Anim(self.obr1, event.x, event.y,
                     ri(0, 6), ri(-3, 3), ri(30, 130))
        elif t == 1:
            a = Anim(self.obr2, event.x, event.y,
                     ri(-6, -1), ri(-1, 1), ri(100, 160))
        else:
            a = AnimOdraz(self.obr3, event.x, event.y,
                          ri(-6, 6), ri(-6, 6), ri(10, 20))
        self.queue.insert(a.tik, a)

    def timer(self):
        while not self.queue.is_empty() and self.queue.front()[0] <= time.
↪time():
```

```
            a = self.queue.dequeue()[1]
            a.posun()
            self.queue.insert(a.tik, a)
        self.canvas.after(10, self.timer)

p = Plocha()
```

Všimnite si, že teraz sa rôzne objekty hýbu aj animujú rôznou rýchlosťou: najrýchlejšie sa krútia zemegule, niektoré vtáčiky mávajú krídlami výrazne rýchlejšie ako iné, zajace skáču najpomalšie.

KAPITOLA 23

Turingov stroj

Alan Turing (http://en.wikipedia.org/wiki/Alan_Turing) (1912-1954) sa považuje za zakladateľa modernej informatiky - rok 2012 sa na celom svete oslavoval ako Turingov rok.

Turingov stroj (http://en.wikipedia.org/wiki/Turing_machine) je veľmi zjednodušený model počítača, vďaka ktorému sa informatika dokáže zaoberať zložitosťou problémov - Turing ho vymyslel v roku 1936.

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)
- 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 páske nachádza nejaký vstupný reťazec (postupnosť symbolov), stroj sa nachádza v počiatočnom stave (pre nás je to stav napr. 0), celý zvyšok nekonečnej pásky obsahuje prázdne symboly (pre nás je to symbol `'_'`)
- činnosť stroja (program) sa definuje špeciálnou dvojrozmernou tabuľkou, tzv. **pravidlami**
- každé pravidlo popisuje:
  - keď si v konkrétnom stave a na páske je tento symbol, tak ho prepíš týmto novým symbolom, posuň sa na páske o (-1, 0, 1) políčko a zmeň stav na tento nový stav
  - napr. pravidlo **(0,a) -> (b,>,1)** označuje: v stave **0** so symbolom **'a'** na páske ho zmeň na **'b'**, posuň sa o jedno políčko vpravo a zmeň svoj stav na 1
- 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 (http://www.google.com/doodles/alan-turings-100th-birthday)

- LEGO Turing Machine (http://vimeo.com/44202270)
- Turing Machine-IA Lego Mindstorms (http://www.youtube.com/watch?v=x92lEFpAG40)
- A Turing Machine - Overview (http://www.youtube.com/watch?v=E3keLeMwfHY)

Zostavme náš prvý program pre Turingov stroj:

(0, a)	->	(A, >, 0)
(0, _)	->	(_, =, 1)

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_
   ^ 1
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

23.1 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, ktorá bude popisovať Turingov stroj a jej metódy budú realizovať výpočet takéhoto stroja. Začneme jednoduchou verziou:

```
class Turing:
    def __init__(self):
        self.program = {}
```

```
        self.paska = ['_']
        self.stav = '0'
        self.konc = {'1'}
        self.poz = 0

    def rob(self):
        ...
```

Atribúty sú asi zrejmé:

- `program` 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 asociatívneho poľa (typ `dict`) tak, že kľúčom bude dvojica **(stav, symbol)** a hodnotou pre tento kľúč bude trojica **(nový_symbol, smer_posunu, nový_stav)**
- `paska` bude "nekonečná" postupnosť symbolov, budeme ju automaticky nafukovať, podľa toho, ako sa nad ňou pohybuje čítacia hlava - na začiatku obsahuje jediný prázdny symbol
- `stav` označuje, momentálne číslo stavu (na začiatku bude napr. 0)
- `konc` označuje, množina koncových stavov
- `poz` je pozícia na páske, zrejme by vždy mala byť v intervale <0, len(paska)-1>

Metódy

- `__init__()` inicializuje atribúty
- `rob()` riadiaca jednotka bude postupne vykonávať kroky programu, kým sa nezastaví
  - metóda vráti `True`, ak program akceptoval symboly na páske, inak vráti `False`

Aby sme nemuseli nejako komplikovane nastavovať atribúty stroja, napr. `program`, `paska`, ..., uvedieme ich ako parametre inicializácie `__init__()`:

- parameter `program` očakáva ako vstup viacriadkový znakový reťazec: každý riadok musí obsahovať jedno pravidlo ako päticu hodnôt

Naprogramujme tieto metódy:

```
class Turing:
    def __init__(self, program='', paska='', stav='0', konc={'1'}):
        self.program = {}
        for riadok in program.split('\n'):
            if riadok:
                a,b,c,d,e = riadok.split()
                self.program[a, b] = (c, d, e)
        self.paska = list(paska or '_')
        self.stav = stav
        self.konc = konc
        self.poz = 0

    def rob(self, vypis=True):
        while self.stav not in self.konc:
            if vypis:
                print(''.join(self.paska))
                print(' '*self.poz+'^', self.stav)
            try:
```

```
                symb, smer, stav = self.program[self.stav, self.paska[self.
↪poz]]
            except:
                return False
            self.paska[self.poz] = symb
            if smer == '>':
                self.poz += 1
                if self.poz >= len(self.paska):
                    self.paska.append('_')
            elif smer == '<':
                if self.poz > 0:
                    self.poz -= 1
                else:
                    self.paska.insert(0, '_')
            self.stav = stav
        if vypis:
            print(''.join(self.paska))
            print(' '*self.poz+'^', self.stav)
        return True
```

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 _ _ = 1', 'aaa')
print(t.rob())
```

```
aaa
^ 0
Aaa
 ^ 0
AAa
  ^ 0
AAA_
   ^ 0
AAA_
   ^ 1
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', '1', {'stop'}).rob())
```

Všimnite si, že stavy tohto programu sú **1**, **2**, **3**, **4**, **5** a **stop**, pričom **1** je počiatočný stav a **stop** je jediný koncový stav.

Program zadaný vstup `'ahoj'` akceptuje ale napr. `'hello'` nie. Doplňme program tak, aby akceptoval obe slová `'ahoj'` aj `'hello'`:

```
print(Turing('''
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
''', 'hello', '1', {'stop'}).rob())
```

Podobný tomuto je program, ktorý akceptuje vstup len vtedy, keď sa skladá z ľubovoľného počtu dvojíc znakov `'ok'`:

```
print(Turing('''
1 o o > 2
1 _ _ = ok
2 k k > 1
''', 'ok'*100, '1', {'ok'}).rob(False))
```

Zostavme ešte takýto program:

- predpokladáme, že na páske je postupnosť 0 a 1, ktorá reprezentuje nejaké dvojkové číslo, napr. '1011' označuje 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
  - 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, =, konc)
(s2, _)	->	(1, =, konc)

Otestujeme naším programom:

```
print(Turing('''
s1 0 0 > s1
s1 1 1 > s1
s1 _ _ < s2

s2 1 0 < s2
s2 0 1 = konc
s2 _ 1 = konc
''', '1011', 's1', {'konc'}).rob())
```

po spustení:

```
1011
^ s1
1011
 ^ s1
1011
  ^ s1
1011
   ^ s1
1011_
    ^ s1
1011_
   ^ s2
1010_
  ^ s2
1000_
 ^ s2
1100_
 ^ konc
True
```

Na záver program, ktorý zistí, či vstup zložený len z písmen `'a'` a `'b'` je palindrom:

```
print(Turing('''
s1 a _ > sa
s1 b _ > sb
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
''', 'aba', 's1', {'end'}).rob())
```

```
aba
^ s1
_ba
 ^ sa
_ba
  ^ sa
_ba_
   ^ sa
_ba_
  ^ saa
_b__
 ^ s2
_b__
^ s2
_b__
 ^ s1
____
  ^ sb
____
 ^ sbb
____
^ end
True
```

Riešenie minuloročnej skúšky

Ukážeme riešenie jedného `zadania minuloročnej skúšky`.

Stiahnite si všetky dátové súbory, ktoré boli k dispozícii na skúške: `subor.zip`.

24.1 Riešenie:

```
class Program:
    def __init__(self, meno_suboru):
        with open(meno_suboru) as subor:
            pr, ps = subor.readline().split()
            pr, ps = int(pr), int(ps)
            self.pole = [['*']*ps]
            for i in range(pr-2):
                self.pole.append(list('*'+' '*(ps-2)+'*'))
            self.pole.append(['*']*ps)
            for riadok in subor:
                p, r, s = riadok.split()
                self.pole[int(r)][int(s)] = p

    def __repr__(self):
        vysl = []
        for r in self.pole:
            vysl.append(''.join(r))
        return '\n'.join(vysl)

    def robot(self, riadok, stlpec):
        self.r, self.s = riadok, stlpec

    def povely(self, postupnost):
        vysl = ''
        r, s = self.r, self.s
        for pocet, smer in postupnost:
            for i in range(pocet):
                rr, ss = [(r-1,s),(r,s+1),(r+1,s),(r,s-1)]['svjz'.
↪index(smer)]
                p = self.pole[rr][ss]
                if p == '*':
                    break
                if p != ' ':
                    vysl += p
                    self.pole[rr][ss] = '*'
```

```
                r, s = rr, ss
        self.r, self.s = r, s
        return vysl

if __name__ == '__main__':
    p = Program('subor.txt')
    print(p)
    p.robot(2,2)
    s = p.povely([(1,'s'),(1,'v'),(2,'j'),(2,'z'),(2,'s')])
    print('zozbieral:', s)
    print(p)

if __name__ == '__main__x':
    p = Program('subor1.txt')
    p.robot(4,1)
    print(p)
    print('prvy:', p.povely([(1,'v'),(3,'s')]))
    print('druhy:', p.povely([(1,'v'),(3,'j')]))
    print(p)
```