Anino BELAN

PYTHON

učebný text pre septimu osemročného gymnázia

BRATISLAVA
2013

Obsah

Úvod

Kedysi v dávnych časoch, keď domáce počítače mali osembitové procesory, keď neexistoval Linux, Mac OS ani Windows a keď grafika počítača zvládla maximálne osem farieb, sa s počítačmi robili prevažne dve veci. Hrali sa na nich hry a programovalo sa. Obe činnosti robili ľudia sami od seba a dobrovoľne. Kto mal prístup k počítaču, užíval si jednak to, že sa mohol stať mocným Pacmanom a žrať strašidlá, jednak to, že sa mohol stať mocným programátorom a niekoľkými šikovnými príkazmi mohol prinútiť počítač, aby robil presne to, čo chce. A keď niekto prišiel s nápadom, že by sa informatika mala učiť v škole, tak sa s týchto dvoch vecí začalo prirodzene učiť práve to programovanie.

Lenže doba pokročila. Počítače majú namiesto osembitových procesorov šesťdesiatštyribitové, medzi operačnými systémami si môžete vyberať a farbami sa nešetrí. Okrem toho sa počítače využívajú na mnohé šikovné veci. Môžete v nich písať typograficky pekne upravené texty, vytvárať tabuľky so zabudovanými výpočtami, počúvať či vytvárať hudbu a video, robiť 3D modely a animácie, skladovať dáta, môžete sa s ich pomocou pripájať do siete a komunikovať s celým svetom.

Tieto zmeny sa samozrejme odrazili aj na vyučovaní informatiky. Na informatike sa učí množstvo vecí, ktoré by sa mohli učiť skôr na slovenčine, hudobnej a výtvarnej výchove a matematike a jediný dôvod, prečo tieto veci pripadli pod predmet informatika je, že sa pri nich používa počítač. Z tohto dôvodu sa na informatike programuje oveľa menej, ako kedysi.

Táto knižka sa vás napriek tomu bude pokúšať naučiť programovať. Dôvody sú viaceré.

Prvý je ten, že vás programovanie môže uživiť. Je to remeslo, ktoré je zaujímavé a okrem toho ešte aj celkom slušne platené. Programátorov je málo, takže nie je problém nájsť si v odbore zamestnanie. Keď sa vydáte touto cestou, môže sa vám stať životným povolaním.

Toto sa samozrejme netýka každého. Aj pre ľudí, ktorí po skončení tohto kurzu už nenapíšu ani riadok kódu ale má zmysel, že ho absolvujú. A to z týchto dôvodov:

- Keď človek aspoň trošku programuje, tak má lepšiu predstavu o tom, čo môže od programov, ktoré pre neho píše niekto iný, čakať. A viac ocení, keď hrá nejakú skvelú hru alebo používa nejaký program, ktorý mu zjednoduší život.
- Ľudia si väčšinou myslia, že vedia veci dobre vysvetliť a väčšinou sa v tejto predstave o sebe žalostne mýlia. Programovať znamená niečo dobre vysvetliť počítaču. Dosť dobre na to, aby robil, čo od neho chceme. A táto schopnosť niečo dobre vysvetliť sa človeku zíde aj inde, než pri práci s počítačom.
- Programovanie má tú výhodu, že keď človek spraví nejakú chybu, príde na to pomerne rýchlo. V živote to tak vždy byť nemusí. Preto je programovanie zaujímavá skúsenosť, vďaka ktorej sa má človek možnosť naučiť vyrovnávať sa s vlastnými chybami a učiť sa ich opravovať.

Programovať sa dá rôznymi spôsobmi a v rôznych jazykoch. Napriek tomu, že na Slovensku sa v školách väčšinou učí Pascal alebo niektorý z jeho klonov (Delphi, Lazarus), rozhodli sme sa zvoliť iný jazyk – Python. Dôvody sú dva.

Prvý je ten, že jazyk Python je pohodlný. Robí sa v ňom príjemne a rýchlo. Na veci, ktoré by ste v Pascale potrebovali štyri riadky, vám bude stačiť jeden, to čo napíšete, bude pravdepodobne

čitateľné aj pre niekoho iného, ako pre vás a namiesto toho, aby ste sa museli sústrediť na množstvo formalít, môžete sa sústrediť na to, čo chcete naprogramovať.

Druhý je ten, že časy, kedy bol Pascal najvhodnejší jazyk na vyučovanie, už dávno pominuli. Keď sa pozriete, ktoré jazyky sa učia v začiatočníckych kurzoch na univerzitách na západe, tak sú to buď jazyky, ktoré sú syntaxou odvodené od jazyka C (teda C, C++ a Java), niektorý funkcionálny jazyk (Haskell), alebo práve Python. Medzi univerzity, ktoré začínajú s Pythonom patrí napríklad Cambridge alebo MIT. Python je totiž jazyk, ktorý samotnou svojou syntaxou núti ľudí, aby písali programy čitateľne. Návyky, ktoré takto získate, oceníte, keď budete programovať v iných jazykoch. K jazyku Python okrem toho existuje mnoho veľmi pekne napísaných knižníc, čo z neho robí rovnako univerzálny jazyk, ako je C++ alebo Java.

V každom prípade sa teším, že ste po tejto knižke siahli a dúfam, že sa vám bude páčiť a bude pre vás užitočná. Želám vám veľa trpezlivosti, pevné nervy, psychickú odolnosť a príjemnú zábavu.

Anino

1. lekcia

Python alebo „Nebezpečné hady“

Python začal vytvárať Holanďan Guido van Rossum v roku 1989 a dodnes je jeho hlavným vývojárom. Názov dal jazyku podľa anglickej humoristickej skupiny Lietajúci cirkus Montyho Pythona. Dnes sa používajú dve jeho verzie – verzia 2 a verzia 3. Tento kurz bude zameraný na verziu 3.

Inštalácia je jednoduchá. Pod Linuxom budete mať Python pravdepodobne rovno nainštalovaný. Ak náhodou nie, stačí použiť váš obľúbený balíčkovací systém a nechať automaticky nainštalovať balíček s názvom `python3`. Ak používate Windows alebo Mac OS X, stačí zájsť na stránku http://www.python.org/getit a stiahnuť si odtiaľ patričný balíček. Dajte si pozor, aby ste stiahli verziu, ktorej číslo začína na 3. Keďže je Python zverejnený pod OpenSource licenciou, nájdete tam aj jeho zdrojové kódy (jadro Pythonu je napísané v Cčku).

Python je teda šťastne nainštalovaný. Čo s ním. Prvá vec, ktorú môžete spraviť, je, že spustíte priamo program `python3`. Keď tak učiníte, ukáže sa vám na obrazovke niečo takéto[1]:

```
Python 3.2.3 (default, Jun  8 2012, 05:36:09)
[GCC 4.7.0 20120507 (Red Hat 4.7.0-5)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Tie zobáky na konci znamenajú, že Python počúva. Môžete mu zadať príkaz a on ho okamžite vykoná. Môže to vyzerať napríklad takto:

```
>>> print(2+3)
5
>>> print(1+1+1+0*9)
3
>>> print("Dobrý deň")
Dobrý deň
>>> print(9/4)
2.25
>>> print(2**5)
32
>>> kr = 5
>>> print(kr)
5

>>> kr = "Ahoj!"
>>> print(kr)
Ahoj!
>>> quit()
```

Ako si môžete všimnúť, Python vie fungovať celkom dobre ako kalkulačka. Funkcia `print` slúži na to, keď potrebujete niečo vypísať. Matematiku Python zvláda celkom dobre, takže keď mu dáte vypísať hodnotu `1+1+1+0*9`, vie, že násobenie má prioritu pred sčítaním, takže najprv

1 To, čo sa vám ukáže, samozrejme závisí od použitého operačného systému. V prípade linuxu budete program pravdepodobne spúšťať v príkazovom riadku, v prípade windows sa po spustení programu objaví terminál. Aj text, ktorý sa objaví, môže byť iný. Dôležité je, že vám Python povie, že sa spustil, uvedie použitú verziu a na konci sa objavia tie tri zobáky.

vypočíta to `0*9` (hviezdička znamená „krát“) a potom k tomu pripočíta tie jedničky. Dobre mu ide aj delenie a umocňovanie (`2**5` znamená 2^5).

Zaujímavý je príkaz `kr = 5`. Znamená totiž „do krabice niekde v pamäti, ktorá sa nazýva `kr` vlož hodnotu 5“. Takéto krabice v pamäti sú veľmi užitočné a nazývajú sa premenné. Meno premennej si môžete vymyslieť aké chcete.[2] Všimnite si, že po vykonaní tohto príkazu Python nič nevypisoval a iba sa objavili ďalšie zobáky. Až keď sme použili funkciu `print`, obsah sme vypísali. Keď potom vykonáme príkaz `kr = "Ahoj!"`, v premennej `kr` sa ocitne nový obsah – reťazec „Ahoj!“ – a predošlá hodnota 5 bude navždy zabudnutá. Keď znovu vypíšeme obsah premennej `kr`, dočkáme sa iného výsledku, ako v predošlom prípade. Funkcia `quit` Python ukončí. Je dôležité, aby funkcia mala za sebou zátvorky. Podľa nich ju vie Python odlíšiť od premennej. Preto má funkcia `quit` za sebou zátvorky, aj keď v nich nemá žiadne parametre.

Toto je jeden spôsob, ako Python používať. Nie je to úplne zlý spôsob – podobne sa používa príkazový riadok. Problém je, že keď chceme tieto všetky veci znovu zopakovať, tak musíme príkazy odznovu napísať.

Nie je ale nič jednoduchšie, ako si zvoliť obľúbený textový editor (napríklad kate, vi, notepad++, TextWrangler ale pokojne aj notepad), všetky príkazy do neho napísať, súbor uložiť s koncovkou `.py` (teda napríklad `01-01-pokus.py` – to prvé číslo je číslo lekcie, druhé je číslo programu v danej lekcii). Súbor teda bude obsahovať nasledujúci text:

```
print(2+3)
print(1+1+1+0*9)
print("Dobrý deň")
print(9/4)
print(2**5)
kr = 5
print(kr)
kr = "Ahoj!"
print(kr)
quit()
```

Teraz už len treba povedať Pythonu, aby ten súbor spustil. Ak pracujete pod linuxom, situácia je jednoduchá. Stačí sa v príkazovom riadku dostať do adresára, v ktorom máte súbor uložený a zadať príkaz

```
python3 01-01-pokus.py
```

Program vám vypíše

```
5
3
Dobrý deň
2.25
32
5
Ahoj!
```

Pod windowsami je situácia trochu zložitejšia. Aby sa súbor s vašimi príkazmi spustil, stačí naň dvojkliknúť. A skutočne. Otvorí sa okno, do neho sa vypíšu výsledky, program skončí a okno sa

2 Dávajte si ale pozor, aby nezačínalo na číslicu a obsahovalo iba písmená, číslice a znak _.

zavrie. Problém je v tom, že toto všetko trvá len zlomok sekundy a vy poriadne nestihnete uvidieť, čo v tom okne vlastne je. Aby ste tomu zamedzili, pred príkaz `quit()` vložte ďalší príkaz

```
input("Stlačte ENTER!")
```

Tento príkaz vypíše `Stlačte ENTER!` a čaká na to, kedy mu používateľ niečo zadá. A aj keď mu nezadáte nič, tak to nevadí. Pre nás je dôležité to čakanie. Kým nestlačíte `ENTER`, okno nezmizne a vy sa môžete kochať tým, čo pekné vám Python vypísal.

Úloha 1: Spustite Python a zadajte mu uvedené príkazy. Vymyslite si nejaké vlastné a tie mu zadajte tiež.

Úloha 2: Čo vypíše Python, keď mu zadáte príkaz `print(10 * "Ahoj ")`

Úloha 3: Vo svojom obľúbenom textovom editore napíšte uvedené príkazy, uložte to pod nejakým rozumným menom s koncovkou `.py` a súbor spustite.

Úloha 4: Čo to spraví, keď do nejakého súboru napíšete príkazy

```
a = "(auto)"
b = "(autobus)"
zapcha = 3*a + 2*b + 4*a + 3*b + a
print(zapcha)
```

a potom to spustíte?

Úloha 5: Nechajte Python vypočítať súčet nepárnych čísel od 1 do 19 a výsledok vypíšte.

Úloha 6: Nechajte Python vypočítať 20! (dvadsať faktoriál) a výsledok vypíšte.

Úloha 7: (Bonusová pre machrov) Spravte zápchu, ktorá nebude vypísaná, ako v úlohe 4, ale vykreslením autíčka nasledujúcim spôsobom:

```
 _/ L\__,
'-o---o-'
```

Ak do reťazca pridáte `"\n"`, znamená to „prejdi na nový riadok“.

2. lekcia

Podmienky a výnimky alebo „Mocné odsadenie“

Prvá lekcia, ktorú ste mali tú česť prednedávnom dokončiť, bola celkom jednoduchá. Pythonu ste voľačo prikázali a on to urobil. Na prvý pohľad sa môže zdať, že kurz splnil svoje poslanie. Počítač vás poslúcha a robí presne to, čo ste mu povedali. Na druhú stranu, skončiť kurz Pythonu prvou lekciou, to predsa len vyvoláva istý pocit nenaplnenia a neúplnosti. Nebojte sa, Python ešte nepovedal svoje posledné slovo.

Na tých programoch, ktoré sme robili minule, je totiž jedna nesympatická vec – zakaždým spravia to isté. Vôbec sa používateľa nespýtajú, čo by chcel a či by to pri ďalšom behu programu náhodou nechcel nejako inak. To sa dá ale ľahko napraviť. Pozrite si nasledujúci program[3]:

```
1   print("Ako sa voláš?")
2   meno = input()
3   vystup = "Ahoj " + meno + ", ja som Python."
4   print(vystup)
```

Všetko to napíšte do súboru s koncovkou `.py` a spustite. Keď program spustíte, najprv sa vás slušne opýta, ako sa voláte. Potom príde zaujímavá časť. To, čo je v tomto programe nové, je funkcia `input`. Tá čaká, kým niečo napíšete a stlačíte `Enter`. Text, ktorý napíšete, vráti funkcia ako svoj výsledok, s ktorým potom môžete robiť, čo uznáte za vhodné. My sme si ho uložili do premennej `meno`.

V treťom riadku vytvoríme premennú `vystup`, do ktorej uložíme za sebou zreťazené texty `"Ahoj "`, meno, ktoré sme načítali zo vstupu a `", ja som Python"`. V štvrtom riadku tento výsledok vypíšeme.

Úloha 1: Pochopte a vyskúšajte si to.

Keď už vieme získať od užívateľa nejakú spätnú väzbu, môžeme skúsiť naprogramovať niečo praktickejšie. Napríklad hrací automat.

```
1   print("Ja som hrací automat.")
2   print("Stav desať eur.")
3   heslo = input("Zadaj heslo: ")
4   if heslo == "krokodil":
5   →   print("Vyhral si.")
6   →   print("Môžeš si zobrať tých desať eur naspäť.")
7   else:
8   →   print("Prehral si.")
9   →   print("Desať eur ti prepadlo.")
10  print("Isto si chceš zahrať ešte raz.")
```

Prvý zaujímavý detail sa vyskytuje v treťom riadku. Funkcia `input` tam totiž má parameter `"Zadaj heslo: "`. To spôsobí, že funkcia najprv napíše `Zadaj heslo:` a až potom čaká na vstup od užívateľa.

3 Riadky sú číslované iba kvôli lepšej orientácii. Keď to budete skúšať, čísla riadkov do programu nepíšte.

Tie dôležité veci sa ale dejú až na riadkoch 4 až 9. Heslo, ktoré užívateľ zadal, máme uložené v premennej `heslo`. Ak užívateľ náhodou trafil správne heslo „krokodil“[4], tak mu chceme napísať, že vyhral a môže si vklad zobrať naspäť. Inak mu chceme napísať, že prehral. Náš program teda potrebujeme rozdeliť na dve nezávislé vetvy, pričom jedna sa vykoná vtedy, keď bude v premennej `heslo` „krokodil“ a druhá sa vykoná vtedy, keď tam bude niečo iné.

Presne na to slúži príkaz `if` (po anglicky „ak“). Konkrétne v našom prípade vyzerá takto:

```
if heslo == "krokodil":
```

Za príkazom `if` nasleduje podmienka. Naša podmienka `heslo == "krokodil"` je pradivá iba vtedy, ak je v premennej `heslo` je uložený text „krokodil“. Za podmienkou treba dať dvojbodku, aby bolo jasné, že už podmienka skončila. Tie znaky „rovná sa“ v tej podmienke musia byť dve, pretože jedno „rovná sa“ už používame, keď chceme niečo vložiť do nejakej premennej.

A teraz sa dostávame k veci, podľa ktorej sa nazýva celá táto kapitola, teda k mocnému odsadeniu. Keď sa pozriete na riadky 5 a 6, teda tie, ktoré sa majú vykonať v prípade, že používateľ ako heslo skutočne zadal `"krokodil"`, tak sú oproti riadku s podmienkou odsadené o jeden tabulátor. (Tabulátor je naznačený šípkou. Tie šípky do svojho programu nepíšte, dajte tam tabulátory!) Python z toho pochopí, že to, čo je odsadené, sa má vykonať len vtedy, ak je podmienka splnená. Keď sa dostane k ďalším neodsadeným riadkom, tak vie, že ten podmienený úsek už skončil a zas má robiť všetko.

Po skončení podmienenej časti nasleduje príkaz

```
else:
```

Tento príkaz sa nikdy nevyskytuje samostatne a vždy sa musí spájať s nejakým predošlým `if`. (Naopak to neplatí – `if` samostatne môže byť a príkaz `else` po ňom nasledovať nemusí.) „Else“ po anglicky znamená „inak“. Riadky, ktoré po `else` nasledujú (a sú odsadené o tabulátor), sa teda vykonajú iba vtedy, keď podmienka v predošlom príkaze `if` nie je splnená. Ak teda používateľ heslo neuhádne, vykonajú sa riadky 8 a 9 a program mu oznámi, že prišiel o peniaze.

Riadok 10 nie je odsadený. Vykoná sa teda bez ohľadu na to, aké heslo užívateľ zadal a vyzve hráča, aby si zahral ešte raz.

Úloha 2: Pochopte, vyskúšajte si to a neprepadnite hráčskej závislosti. Aj tak nevyhráte.

Úloha 3: Pridajte pred riadok 4 príkaz `print(heslo == "krokodil")` Čo bude tento príkaz vypisovať?

Čo robiť, ak chcete napísať nejakú komplikovanejšiu vec a napríklad do jednej vetvy programu vložiť ďalší príkaz `if`? Je to jednoduché. Stačí pridať ďalší tabulátor ako v nasledujúcom programe:

```
1   odpoved = input("Chceš mi povedať číslo? ")
2   if odpoved == "ano":
3   →   cislo = input("Tak povedz: ")
4   →   if cislo != "42":
5   →   →   print("Aha...")
6   →   else:
7   →   →   print("Óóóóó :)")
```

4 Správne heslo musí mať všetky písmenka malé a musí mať krátke i.

Program už nebudeme komentovať tak podrobne, ako predošlý. Na začiatku sa opýtame užívateľa, či sa s nami vôbec hodlá baviť a ďalej sa bude niečo robiť iba ak odpovie `"ano"`. Keďže celá ďalšia časť programu od riadku 3 po riadok 7 je podmienená, všetko je odsadené o tabulátor. V tejto časti sa používateľa program opýta, aké číslo mu to chcel povedať a ak to bude 42[5], patrične to ocení. Riadky 5 a 7 sú podmienené vďaka príkazom `if` a `else` na riadkoch 4 a 6. Keďže sú ale riadky 4 a 6 už o jeden tabulátor odsadené, riadky 5 a 7 musia byť odsadené až o dva.

Všimnite si na tomto programe niekoľko ďalších zaujímavostí. Prvá je dvojica znakov `!=` v riadku 4. Táto dvojica znamená „nerovná sa“. Ak teda užívateľ zadá iné číslo, ako 42, program to veľmi neocení.

Ďalšiu zaujímavú vec uvidíte, keď sa na program pozriete z väčšej diaľky. To odsadzovanie je dobré nie len pre Python, ale aj pre vás. Vďaka nemu je rovno vidieť, ktoré časti kódu patria ku ktorým podmienkam a program sa preto dobre číta. Vidíte napríklad hneď, že `else` na riadku 6 patrí k `if` na riadku 4 a nie k `if` na riadku 2. K podobnému odsadzovaniu nabádajú učitelia aj žiakov, ktorí začínajú programovať v jazykoch Pascal alebo C. Začiatočníci v týchto jazykoch to ale väčšinou k vlastnej škode nerobia.

Úloha 4: Vyskúšajte. Skúste zadať programu také vstupy, aby sa zachoval všetkými možnými spôsobmi.

Posledná zaujímavá črta predošlého programu o ktorej bude reč je drobný detail v podmienke `if cislo != "42":` Ten drobný detail sú tie úvodzovky okolo 42. Ako už bolo povedané vyššie, funkcia `input` vráti ako svoj výsledok text. Hodnota uložená v premennej `cislo` teda nebude číslo, ale text. Ak nerozumiete, v čom je problém, skúste si spustiť nasledujúci minimalistický program, ktorý počíta dvojnásobok zadaného čísla a zadať mu na vstupe napríklad 100.

```
1   cislo = input("Zadaj cislo: ")
2   print("Dvojnasobok toho cisla je", 2 * cislo)
```

Úloha 5: Vyskúšajte.

Neočakávaný výsledok predošlého programu je spôsobený tým, že keď má Python zistiť, aká je hodnota výrazu `3 * 7`, vyjde mu `21`, pretože 7 je číslo a keď má zistiť, aká je hodnota výrazu `3 * "ku"` , vyjde mu `"kukuku"`, pretože `"ku"` je reťazec. No a práve preto, že v premennej `cislo` máme reťazec, ktorý sa na číslo síce podobá, ale číslo to nie je, sa predošlý program správa tak divne.

Ako teda napísať program, ktorý vynásobí zadané číslo dvomi? Musíme reťazec, ktorý nám zistí funkcia `input`, zmeniť na číslo. Na to slúži funkcia `eval`[6]. Táto funkcia vie prerobiť reťazec na číslo. Náš program teda môžeme prerobiť nasledujúcim spôsobom:

```
1   vstup = input("Zadaj cislo: ")
2   cislo = eval(vstup)
3   print("Dvojnasobok toho cisla je", 2 * cislo)
```

Teraz už všetko funguje tak ako má.

Až na detaily.

---

5 http://en.wikipedia.org/wiki/Phrases_from_The_Hitchhiker%27s_Guide_to_the_Galaxy#The_number_42

6 Z anglického evaluate – vyhodnoť. Funkcia vyhodnotí zadaný výraz. Je na nej nebezpečné to, že ak umožníte zlomyseľnému používateľovi zadať na vyhodnotenie výraz obsahujúci príkaz, ktorý zmaže operačný systém, funkcia `eval` vyhodnotí aj ten. Takže pri reálnom programovaní používajte túto funkciu s mierou a nanajvýš opatrne.

Úloha 6: Skúste predošlému programu ako vstup zadať `"ku"`.

Ak ste úlohu 6 splnili, videli ste, že program uhynul v smrteľných kŕčoch a vychrlil na vás chybovú hlášku podobnú tejto:

```
Traceback (most recent call last):
  File "02-04-dvojnasobok.py", line 2, in <module>
    cislo = eval(vstup)
  File "<string>", line 1, in <module>
NameError: name 'ku' is not defined
```

Python vám tam píše, ako sa volá súbor, ktorý ste práve spúšťali (v našom prípade `02-04-dvojnasobok.py`), na ktorom riadku sa stala chyba (na druhom), vypíše chybný riadok, oznámi aká knižnica bola práve volaná (`string`) a nakoniec chybu čo najlepšie popíše (po voľnom preklade z angličtiny niečo v zmysle „netuším, čo je to 'ku'“.

Isto uznáte, že takéto správanie je istá škvrna na dokonalosti nášho programu. Program by sa proti nesprávnemu vstupu mal nejako kultivovane ohradiť. Ako ale rozoznať, ktorý vstup je správny a ktorý nie?

Jedna z možností je, že zavoláme funkciu `eval`, ale budeme dávať pozor, či to, čo bude robiť, dopadne dobre. A ak to skončí chybou, nedovolíme programu, aby užívateľovi vypisoval veci, ktoré sú pre nás zaujímavé, ale jemu by nič nepovedali a namiesto toho mu vypíšeme niečo zrozumiteľnejšie. Spraví sa to takto:

```
1  vstup = input("Zadaj cislo: ")
2  try:
3      cislo = eval(vstup)
4  except:
5      print("Zadali ste nejaký nezmysel")
6      quit()
7  print("Dvojnasobok toho cisla je", 2 * cislo)
```

Kritickú časť programu, v ktorej môže nastať chyba, vložíme do sekcie `try` (po anglicky „skús“). Ak sa v tejto sekcii niečo pokazí – teda ak nastane chyba, ktorá by vygenerovala podobný výpis, aký ste videli pred chvíľou, program bude pokračovať sekciou `except` (po anglicky „urobiť výnimku“). V tejto sekcii na užívateľa zvalíme, že to nefunguje a program opustíme. V prípade, že všetko prebehlo dobre, program pokračuje až za sekciou `except` a vypíše ten dvojnásobok, po ktorom sme celý čas prahli.

Úloha 7: Vyskúšajte. Čo to spraví, keď do vstupu napíšete `(3 * 3 * 3 - 7) / 5` ?

Úloha 8: Čo to spraví, keď z predošlého programu vymažete riadok 6 a na vstupe zadáte programu nejaký nezmysel?

Úloha 9: Ak viete, že podmienka `if a % 2 == 0:` je splnená, práve vtedy, keď je číslo párne, napíšte program, ktorý načíta číslo a vypíše vám, či je párne, alebo nepárne.[7]

Úloha 10: Zmeňte posledný program, aby v prípade, ak užívateľ zadá číslo, vypísal jeho dvojnásobok a ak užívateľ číslo nezadá, program vypíše vstup dvakrát za sebou.

---

7 Znak `%` zistí v Pythone zvyšok po delení. Ak je teda `a % 2` rovné 0, znamená to, že číslo uložené v premennej `a` dá po vydelení dvoma zvyšok nula a teda musí byť párne.

3. lekcia

Zoznamy, n-tice a slovníky alebo „Skladujeme informácie“

V predošlej lekcii sme vylepšili našu schopnosť niečo Pythonu prikázať a to, čo sme robili, sa už celkom solídne začalo podobať na programovanie. V tejto lekcii ale tento sľubne nastúpený trend ešte na chvíľu opustíme. Naučíme sa ale veci, ktoré sa nám budú v budúcnosti veľmi hodiť a ktoré nám programátorský život uľahčia nečakaným spôsobom.

Ide o to, že doteraz sme do jednej premennej uložili vždy iba jednu vec. Niekedy sa ale hodí mať tam tých vecí uložených viacero. Programátori vymysleli viacero fínt ako sa to dá uskutočniť. A v tejto lekcii si povieme o troch z nich.

Prvý spôsob, ako do jednej premennej uložiť veľa vecí je zoznam. Zoznam vyrobíte napríklad tak, že jednotlivé jeho položky napíšete do hranatých zátvoriek a oddelíte čiarkami:

```
a = ["Veľký tresk", 42, "Život, vesmír a vôbec", "Star trek", 47]
```

Do tohto zoznamu sme uložili až päť vecí. Vidíte, že niektoré sú reťazce a niektoré čísla. Do zoznamov môžete vkladať prakticky čokoľvek, dokonca ďalšie zoznamy. Dobre si ale rozmyslite, ak chcete do zoznamu vkladať objekty rôznych typov. Ak to robíte bezdôvodne, môže sa vám to vypomstiť.

Keď už máte nejaký zoznam vytvorený, môžete s ním robiť množstvo vecí. Ak napríklad poznáte poradie prvku v zozname, môžete k nemu pristupovať pomocou hranatých zátvoriek.

Úloha 1: Spustite si Python ako príkazový riadok, vytvorte si rovnaký zoznam, ako tuto vyššie a potom zadajte príkaz `print(a[1])` Čo vám Python vypíše?

Tí, ktorí čakali, že Python bude písať niečo o veľkom tresku sa v predošlej úlohe dočkali sklamania. Prvky v zozname sú totiž číslované od nuly. `a[1]` je až druhý prvok a príkaz z úlohy 1 preto vypíše 42.

Úloha 2: Čo vypíšu nasledujúce príkazy? Najprv skúste uhádnuť a potom si to vyskúšajte.

```
print(a[5])
print(a[-1])
print(a[-5])
```

Nešvindľujte a naozaj si to vyskúšajte prv, ako budete čítať ďalej.

Prvý príkaz skončí chybovou hláškou, ktorá bude obsahovať niečo v zmysle `IndexError: list index out of range`, čo v preklade znamená, že index, ktorý ste použili je mimo rozsah zoznamu. Je to pochopiteľné. Keďže má náš zoznam päť prvkov, sú to prvky `a[0]` až `a[4]`. Prvok `a[5]` by bol až šiesty a taký tam nie je.

Ďalšie dva príkazy prekvapivo chybovou hláškou neskončia. Keď Pythonu uvediete záporný index, začne počítať od konca. `a[-1]` je teda posledný prvok zoznamu. Keďže tých prvkov máme päť, `a[-5]` bude prvý prvok. Keby ste sa ale chceli dostať k prvku `a[-6]`, skončilo by to rovnakou chybovou hláškou, ako v prípade `a[5]`.

Prvky zoznamu nemusíte iba vypisovať, môžete ich aj meniť. Keď napríklad zadáte príkaz `a[4] = "Enterprise"` a necháte si vypísať zoznam `a`, bude vyzerať takto:

```
['Veľký tresk', 42, 'Život, vesmír a vôbec', 'Star trek', 'Enterprise']
```

Úloha 3: Ďalšia séria pokusov so zoznamami. Čo vypíšu nasledujúce príkazy?

```
print(a[2:4])
print(a[2:7])
print(a[6:8])
print(a[2:])
print(a[:3])
print(a[:2]+a[3:])
```

Prvý príkaz vám vypíše kus zoznamu, ktorý začína prvkom číslo 2 (to je tretí) a končí pred prvkom číslo 4 (to je piaty), takže vypíše `['Život, vesmír a vôbec', 'Star trek']`. Je dôležité si pamätať, že ak sa v Pythone zadávajú rozsahy, vždy to funguje tak, že ľavý okraj tam patrí a pravý nie.

Druhý príkaz prekvapivo chybu nevypíše. Ak rozsah končí za koncom aktuálneho zoznamu, zoberú sa všetky prvky do konca zoznamu, takže príkaz vypíše `['Život, vesmír a vôbec', 'Star trek', 'Enterprise']`. Chybou neskončí ani tretí príkaz. Ak je rozsah úplne mimo, výsledkom bude prázdny zoznam a príkaz vypíše `[]`.

Štvrtý príkaz vypíše časť zoznamu od prvku číslo 2 do konca, teda opäť `['Život, vesmír a vôbec', 'Star trek', 'Enterprise']`. Piaty príkaz vypíše časť zoznamu od začiatku po prvok číslo 2 (prvok číslo 3 tam už nebude!), takže výsledok bude `['Veľký tresk', 42, 'Život, vesmír a vôbec']`.

Posledný príkaz ukazuje, že zoznamy môžeme spájať obyčajným `+`. Keď spojíme dva podzoznamy uvedené v príkaze, dostaneme `['Veľký tresk', 42, 'Star trek', 'Enterprise']`.

Podobne, ako pri jednotlivých prvkoch, môžeme do zoznamu priradzovať a nahrádzať aj celé úseky. Ak napríklad zadáme príkaz

```
a[1:3] = ["Han Solo", "Star wars", "Yoda", "Leia"]
```

tak sa prvok č.1 a prvok č.2 (teda druhý a tretí) zmažú a nahradia sa uvedenými štyrmi prvkami. Zoznam `a` teda bude vyzerať takto:

```
['Veľký tresk', 'Han Solo', 'Star wars', 'Yoda', 'Leia', 'Star trek',
'Enterprise']
```

Aby sa vám so zoznamami robilo pohodlnejšie, máte k dispozícii mnoho príjemných funkcií. Funkcie používate tak, že napíšete meno zoznamu, potom bodku a potom meno funkcie. Ak by sme napríklad chceli náš zoznam utriediť, napíšeme `a.sort()` a keď si ho potom necháme vypísať, bude v ňom

```
['Enterprise', 'Han Solo', 'Leia', 'Star trek', 'Star wars', 'Veľký
tresk', 'Yoda']
```

Pri volaní tejto funkcie si musíte dať pozor, aby v zozname boli veci, ktoré sa dajú navzájom porovnávať. Keby ste tam stále mali textové reťazce aj čísla, funkcia by vyhlásila chybu.

Ak by sme chceli náš zoznam otočiť, zadáme príkaz `a.reverse()`. Ak by sme chceli na koniec zoznamu pridať prvok `"Ender"`, zavoláme príkaz `a.append("Ender")`, ak chceme zo

zoznamu zmazať Star trek, robí sa to príkazom `a.remove("Star trek")` a ak chceme zmazať posledný prvok zoznamu a pritom ho vložiť do premennej `e`, spravíme to príkazom `e = a.pop()`

Úloha 4: Čo bude v zozname `a`, keď s ním toto všetko spravíte?

Okrem týchto funkcií existujú ešte niektoré ďalšie. Prípadných záujemcov odkazujeme na Google a manuály.

Okrem funkcií s pomocou ktorých môžeme zoznamy upravovať existujú aj funkcie, ktoré priamo zoznamy vytvárajú. Spomeňme si najdôležitejšiu z nich – funkciu `range`.

`range(10)` vytvorí zoznam `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]` Rovnako, ako pri indexoch začína zoznam nulou a končí pred zadanou hranicou. `range(5,10)` vyrobí zoznam `[5, 6, 7, 8, 9]` – začína sa päťkou a končí pred desiatkou. `range(5,10,2)` vyrobí zoznam `[5, 7, 9]` – začína sa päťkou, končí sa pred desiatkou a veľkosť kroku je 2. Takéto zoznamy budeme používať často.[8]

Spomeňme ešte krátko ďalšie dva spôsoby, ako napchať viac údajov do jednej premennej. Prvý z nich je n-tica. Veci v n-nici sa oddeľujú čiarkou, okolo n-tice sa môžu dať okrúhle zátvorky, ale nie sú povinné. N-ticu môžeme vyrobiť príkazom

```
k = 2, 3, 5
```

Tiež môžeme k jednotlivým prvkom pristupovať s pomocou indexov, teda napríklad `k[0]` má hodnotu `2`. Na rozdiel od zoznamu sa ale jednotlivé prvky nedajú meniť. N-tica má ale tú výhodu, že sa dá ľahko rozbiť naspäť na prvky. Môžeme teda spraviť priradenie

```
p, q, r = k
```

a v `p` bude `2`, v `q` bude `3` a v `r` bude `5`.

Toto sa občas hodí, keď potrebujeme vymeniť obsah dvoch premenných. Nemusíme ani robiť okľuku cez ďalšiu premennú. Výmena premenných môže prebehnúť priamo takto:

```
a = 5
b = 3
a, b = b, a
```

Po vykonaní týchto príkazov bude v `a` trojka a v `b` päťka.

Úloha 5: Vyskúšajte.

Posledný spôsob na masové ukladanie údajov je slovník. Rovno ho predvedieme na príklade:

```
slovnik = {"ahoj":"hello", "svet":"world"}
print( slovnik["ahoj"], slovnik["svet"] )
slovnik["dovidenia"] = "goodbye"
print( slovnik["dovidenia"] )
```

Ako ste si už určite všimli, slovník funguje podobne, ako zoznam, ale ako index sa nepoužíva číslo, ale v podstate hocičo. V našej ukážke sú to reťazce. Ako môžete vidieť v treťom riadku, do slovníka môžete kedykoľvek dodávať nové položky.

Úloha 6: Vyskúšajte a tvorivo rozviňte.

---

8 Zoznamy vytvorené pomocou `range` sú istým spôsobom zvláštne. Nemôžete napríklad sčítať `range(3)+range(4)`, pretože výsledok by už nebol typu `range`. Ak zoznam `[0, 1, 2, 0, 1, 2, 3]` naozaj potrebujete vyrobiť, musíte tie zoznamy zmeniť na obyčajné príkazom `list`. Hľadaný zoznam vám teda vyrobí príkaz `list(range(3)) + list(range(4))`

Pri dátových štruktúrach v Pythone si treba dať pozor na jednu vec. Pekne ju ilustruje nasledujúca postupnosť príkazov:

```
a = [ 2, 4, 6 ]
b = a
a[1] = 100
print(b[1])
```

Čo myslíte, akú hodnotu Python vypíše? Štyri alebo sto?

Vyhrali tí, čo stavili na sto. Veci totiž fungujú tak, že priradenie

```
a = [ 2, 4, 6 ]
```

v skutočnosti znamená „urob zoznam, zapamätaj si ho a daj mu meno `a`“ a priradenie

```
b = a
```

znamená „tomu istému zoznamu daj ešte aj meno `b`“. Keď teda zmeníme druhý prvok v zozname `a`, zmení sa aj druhý prvok v zozname `b`. Ak nám toto správanie nevyhovuje a v premennej `b` by sme chceli mať síce kópiu zoznamu `a`, ale bola by to iná kopia a zmeny `a`-čka by sa na nej neprejavovali, priradenie v druhom riadku je treba urobiť takto:

```
b = list(a)
```

Funkcia `list` zabezpečí, že sa zo zoznamu, ktorý je v `a` vytvorí kópia. Tej dáme meno `b`, takže sa `a` s `b` už ovplyvňovať nebudú. Podobne ak chceme vytvoriť kópiu slovníka, musíme použiť `dict`:

```
dictionary = dict(slovnik)
```

Keby sme spravili iba `dictionary = slovnik`, mali by sme stále iba jeden slovník, ten by mal ale dve rôzne mená: `slovnik` a `dictionary`.

Úloha 7: Pochopte a vyskúšajte.

Na záver tejto lekcie ešte jedna poznámka: aj obyčajné textové reťazce sú v podstate zoznamy a dá sa s nimi tak narábať. Keď vykonáte nasledujúce príkazy:

```
a = "najneobhospodárovávateľnejšieho"
print(a[7:15])
```

Python vám napíše `hospodár`.

Úloha 8: Iba s pomocou premennej `a` z predošlej ukážky a jej podreťazcov vypíšte čo najviac rôznych slov. Napríklad `print(a[7:9]+a[18:20]+a[12:10:-1])` Mimochodom – čo presne spraví tá posledná vec v tom súčte?

4. lekcia

Cykly
alebo „opakovanie je matka múdrosti“

V tejto lekcii budeme podobne ako v druhej lekcii vytvárať programy, takže spravíte dobre, ak si jednotlivé príkazy budete ukladať do súboru a potom ich s pomocou Pythonu budete spúšťať. Oproti programom z druhej lekcie sa ale programy z tejto lekcie budú líšiť v jednom podstatnom detaile. V druhej lekcii sa totiž každý príkaz, ktorý ste do programu napísali, vykonal maximálne raz. A v prípade, že mal tú smolu a vyskytoval sa vo vnútri nejakej podmienenej časti a podmienka nebola splnená, nemusel sa vyskytovať ani raz. A v tejto lekcii sa naučíme, ako to zariadiť, aby sa nejaká časť programu vykonala viackrát.

To, aby sa nejaký kus programu opakoval, nám môžu zabezpečiť práve cykly. Python pozná dva druhy cyklov: cyklus `for` (po anglicky „pre“) a cyklus `while` (po anglicky „kým“). Pozrime sa najprv na ten prvý z nich.

Predstavte si, že máte zoznam, v ktorom máte uložené mená hrdinov:

```
hrdinovia = ["Indiana Jones", "Spiderman", "Captain America", "Hulk", "Iron Man", "Thor"]
```

a chcete napísať program, ktorý o každom z uvedených chlapov napíše, že je hrdina. (Aby o tom náhodou niekto nepochyboval…) Jeden zo spôsobov, ako to urobiť, je napísať nasledujúci program:

```
print(hrdinovia[0],"je hrdina!")
print(hrdinovia[1],"je hrdina!")
print(hrdinovia[2],"je hrdina!")
print(hrdinovia[3],"je hrdina!")
print(hrdinovia[4],"je hrdina!")
print(hrdinovia[5],"je hrdina!")
```

Iste ale uznáte, že takýto prístup má svoje nedostatky. Jednak si človek pripadá ako cvičená opica, ktorá mačká stále dokola Ctrl-C a Ctrl-V, pretože každý z tých riadkov je až na jeden drobný detail prakticky rovnaký, jednak je zrejmé, že ak by si niekto spomenul na nejakého ďalšieho superhrdinu, bolo by treba program opraviť a doplniť ďalší riadok.

Cyklus typu `for` umožňuje vykonať nejaký kus kódu pre každý prvok z určeného zoznamu. Celú tú strašnosť tuto vyššie môžeme s jeho pomocou nahradiť týmito dvoma riadkami:

```
for chlap in hrdinovia:
    print(chlap,"je hrdina!")
```

Program, ktorý tu vidíte, vykoná nasledujúcu vec: Postupne zoberie všetky prvky zo zoznamu `hrdinovia`, vloží ich do premennej `chlap` a vykoná blok kódu, ktorý je odsadený o tabulátor. Predošlú vetu si prečítajte ešte raz, pretože je dôležitá!

Rozmenené na drobné: Program najprv vloží do premennej `chlap` reťazec `"Indiana Jones"` a vykoná ten `print`, potom vloží do premennej `chlap` reťazec `"Spiderman"` a vykoná `print`, potom vloží do premennej `chlap` reťazec `"Captain America"` a vykoná `print` atď. až kým neprejde celý zoznam.

Úloha 1: Pochopte a vyskúšajte. Nezabudnite v programe nastaviť premennú `hrdinovia`. Zmeňte program tak, aby ste nezmenili zoznam `hrdinovia`, ale aby vypisoval iba tých hrdinov, ktorí patria medzi Avengerov. (To sú tí od Kapitána America ďalej.)

Aby bolo jasnejšie, ako príkaz for funguje, ukážeme niekoľko rôznych programov, ktoré budú mať rovnaký for cyklus. Cyklus bude prechádzať zoznam `range(1,11)`, teda zoznam čísel od 1 do 10. Každý program bude ale robiť niečo trochu iné:

```
for i in range(1,11):
    print(i)
```

```
sucet = 0
for i in range(1,11):
    sucet = sucet + i
print(sucet)
```

```
sucin = 1
for i in range(1,11):
    sucin = sucin * i
print(sucin)
```

```
for i in range(1,11):
    s = ""
    for j in range(i):
        s = s + "*"
    print(s)
```

Prvý program čísla od 1 do 10 postupne dosadí do premennej `i` a iba ich vypíše.

Druhý program si vyrobí premennú `sucet`, do ktorej v cykle postupne pripočíta najprv jednotku, potom dvojku, potom trojku atď. až po desiatku, takže v premennej `sucet` bude nakoniec súčet všetkých čísel od 1 do 10. Tretí program funguje rovnako, ako druhý, len číslami od 1 do 10 budeme postupne násobiť premennú súčin, takže v nej nakoniec bude uložené číslo 10! (desať faktoriál).

Štvrtý program je trošku komplikovanejší. Odsadená časť, ktorá sa vykoná postupne so všetkými hodnotami, má až štyri riadky. V prvom sa vytvorí prázdny reťazec `s`. Potom nasleduje ďalší cyklus, ktorý je do prvého vnorený. Na tom cykle je zaujímavé to, že sa bude opakovať presne `i` krát. Teda keď bude v `i` jednotka, zopakuje sa raz, keď bude v `i` dvojka, zopakuje sa dvakrát atď. Vo vnútornom cykle sa k reťazcu `s` prilepí vždy jedna hviezdička, takže ak je napríklad hodnota `i` päť, bude po skončení vnútorného cyklu v premennej `s` reťazec `"*****"`. Na záver cyklu sa reťazec vypíše.

Úloha 2: Uhádnite, čo spraví štvrtý program ešte predtým, ako ho spustíte. Nešvindľujte! Vyskúšajte a pochopte.

Úloha 3: Vypočítajte súčet čísel od 1 do 1000.

Úloha 4: Vypočítajte súčin čísel od 1 do 1000 (teda číslo 1000!).

Úloha 5: Skúste napísať program, ktorý bude robiť presne to isté, čo štvrtý program z ukážky, ale nebude mať v sebe vnorené dva cykly. (Dá sa to!)

Úloha 6: Napíšte program, ktorý si vypýta od používateľa reťazec, a vypíše ho tak, že medzi každé dva znaky vloží hviezdičku. Teda napr. pre vstup `"Ahoj"` vypíše `"A*h*o*j"`. Žiadna hviezdička nesmie prečnievať.

Druhý typ cyklu je cyklus `while`. Ten nie je viazaný na žiadny zoznam, ale na podmienku. Cyklus sa bude opakovať dovtedy, kým je podmienka splnená. Týmto spôsobom môžeme napríklad kontrolovať, či je vstup od používateľa správny. Pozrime si opäť ukážkový program:

```
1   vstup = ""
2   while vstup != "A":
3       vstup = input("Napíš A: ")
4       if vstup != "A":
5           print("Hovoril som A!!!")
6   print("Hurá, je to A!")
```

Riadky 3, 4 a 5 sa budú opakovať dovtedy, kým je v premennej `vstup` niečo iné, ako `A`. Na riadku 3 načítame vstup. Riadky 4 a 5 slúžia iba na to, aby sme používateľovi vynadali, ak to zase nezvládol. Keď sa mu konečne podarí zadať `A`, cyklus skončí (pretože podmienka na riadku 2 prestane byť splnená) a vykoná sa riadok č. 6, v ktorom sa program poteší, že sa to konečne podarilo.

Podmienka na druhom riadku sa kontroluje vždy predtým, než sa telo cyklu spustí. Preto sme museli nastaviť premennú `vstup` ešte pred cyklom. Ak by sme ju nenastavili, dostali by sme chybovú hlášku, lebo by Python nevedel otestovať podmienku na riadku 2. Ak by sme premennú `vstup` nastavili na hodnotu `A`, cyklus by sa nevykonal ani raz, lebo podmienka by hneď na začiatku nebola splnená.

Úloha 7: Načítavajte čísla až dovtedy, kým používateľ nezadá nulu. Potom vypíšte súčet všetkých zadaných čísel.

Úloha 8: Nájdite najmenšie číslo, ktoré je väčšie ako 10 000 a je deliteľné 31. (Rada: Vložte do premennej 31 a pripočítavajte k nej 31 dovtedy, kým bude výsledok menší alebo rovný, ako `10000`.)

S cyklami sa viažu ešte dva užitočné príkazy, ktoré sa dajú použiť v oboch typoch cyklov. Prvý je príkaz `break` (po anglicky „zlom“), ktorý použijeme, keď nejaký cyklus potrebujeme rýchlo ukončiť. Napríklad keby sme chceli náš predošlý program upraviť tak, že po troch zlých pokusoch usúdime, že to používateľ nezvládne a skončíme:

```
1   vstup = ""
2   pocetZlych = 0
3
4   while vstup != "A":
5       vstup = input("Napíš A: ")
6       if vstup != "A":
7           print("Hovoril som A!!!")
8           pocetZlych = pocetZlych + 1
9       if pocetZlych == 3:
10          break
11
12  if pocetZlych == 3:
13      print("Nevadí! Niekedy nabudúce.")
14  else:
15      print("Hurá, je to A!")
```

Zriadili sme si novú premennú `pocetZlych`, v ktorej si budeme počítať počet zlých pokusov. Ak sa používateľ pomýli, na riadku 8 zväčšíme jej hodnotu. Ak premenná `pocetZlych` dosiahne hodnotu 3, znamená to, že sa používateľ pomýlil už trikrát a tak zavoláme príkaz `break`. Ten spôsobí, že cyklus skončí a program bude pokračovať za ním. Tam musíme pozrieť, či sa používateľ pomýlil trikrát (vtedy sa tešiť nebudeme, iba ho poľutujeme alebo povzbudíme), ale ak sa nepomýlil trikrát, vtedy ho chváliť môžeme.

V prípade, že použijeme príkaz `break` v nejakom cykle, ktorý je vnorený do iného cyklu, zruší sa iba ten najvnútornejší.

Úloha 9: Pochopte a vyskúšajte!

Úloha 10: Môže sa stať, že na riadku 12 bude hodnota premennej `pocetZlych` 4? Ak áno, ako? Ak nie, prečo?

Druhé užitočné slovo je `continue` (po anglicky „pokračuj“). Znamená „na tento beh cyklu sa vykašli a sprav ďalší“. Predstavte si, že chceme spočítať všetky nepárne čísla od 1 do 20. Mohli by sme to spraviť napríklad takto:

```
1   sucet = 0
2   for i in range(1,21):
3       if i % 2 == 0:
4           continue
5       sucet = sucet + i
6   print(sucet)
```

Cyklus na riadku 2 zabezpečí, že riadky 3 až 5 sa vykonajú dvadsaťkrát, pričom v `i` budú postupne všetky čísla od 1 do 20. Riadkami 3 a 4 si ale zabezpečíme, že ak bude v `i` párne číslo, program sa na neho vykašle. To znamená, že riadok 5 sa vykoná iba pre nepárne hodnoty `i` a iba tie sa dostanú do celkového súčtu.

Rozdiel medzi `break` a `continue` je ten, že keby sme na riadku 4 použili `break`, cyklus by skončil už pri `i` rovnom 2 a v premennej `sucet` by bola hodnota 1. Keďže sme ale použili `continue`, cyklus pre `i` rovné 2 neskončil, iba nepokračoval ďalej na riadok 5, ale spustil sa znovu s ďalšou hodnotou `i`.

Úloha 11: Pochopte a vyskúšajte.

Úloha 12: Spravte to jednoduchšie bez `continue`.

5. lekcia

Moduly alebo „nevymýšľajte koleso“

Žiadny jazyk určený na vývoj softvéru si dnes nevystačí len sám o sebe. Časy, kedy si každý programátor musel všetko spraviť od základov sám, sú nenávratne preč. Programovanie sa rozrástlo do mnohých smerov a zahŕňa mnohé oblasti ľudskej činnosti. Niektorí ľudia programujú hry, niektorí robia veľké databázové systémy, niektorí programujú mikročipy priemyselných robotov, niektorí sieťové komunikačné protokoly a každý z nich potrebuje trochu inú sadu nástrojov. A takéto rôzne sady nástrojov poskytujú práve moduly.

Moduly sú knižnice objektov a funkcií, ktoré pre vás urobili iní programátori, aby sa vám dobre robilo. Keď si nainštalujete Python, niektoré moduly už máte priamo k dispozícii. Okrem nich existuje ale mnoho ďalších, takže skôr, než sa pustíte do nejakého väčšieho projektu, oplatí sa stráviť nejaký čas googlením, aby ste zistili, či už niekto nenaprogramoval buď rovno to, čo potrebujete, alebo aspoň nejakú vec, ktorá vám prácu výrazne uľahčí. Traduje sa, že Python má moduly úplne na všetko. A je to takmer pravda.

Poďme sa ale pozrieť na veci, ktoré máme k dispozícii rovno. Aj tých je ale relatívne veľa. Ako sa dá dozvedieť, ktoré to sú a čo obsahujú? Python v sebe obsahuje pomerne rozsiahlu dokumentáciu. Tá je písaná po anglicky, ale ak to myslíte s programovaním vážne, tak čím skôr začnete čítať anglickú dokumentáciu, tým lepšie.

Keď sa chcete dovolať Pythonovskej pomoci, treba spustiť interpreter Pythonu a keď sa vám objavia tri zobáky, ktoré hovoria, že Python počúva, treba napísať príkaz `help()`. Tým sa prepnete do prostredia na čítanie dokumentácie. Keď budete chcieť ísť naspäť do Pythonu, treba napísať `quit` alebo stlačiť Ctrl-D. Keď ale chcete niečo vedieť, môžete sa spýtať buď na kľúčové slová Pythonu (vtedy treba napísať `keywords` – znamená to „kľúčové slová“), alebo sa môžete spýtať na nejaké veci, ktoré sú priamo súčasťou jazyka (treba napísať `topics` – znamená to „témy“), alebo napíšete `modules` a vtedy sa niečo dozviete práve o moduloch. Python na vás vychrlí zoznam modulov a vy si môžete vyberať.

Je zrejmé, že celým tým množstvom modulov sa v jednej lekcii prehrabať nedá. Preto spomeňme aspoň dva z nich. Prvá je knižnica `random`. Táto knižnica obsahuje množstvo vecí, ktoré sa týkajú náhody, náhodných čísel a štatistiky.

Úloha 1: Spustite si interpreter Pythonu, zadajte príkaz `help()`, potom napíšte `random` a pozrite si dokumentáciu ku knižnici. Potom použite vyhľadávač a pozrite si dokumentáciu k tej istej knižnici na internete.

Pravdepodobne vás to množstvo funkcií, ktoré sú k dispozícii, zmiatlo. Srdce štatistika by síce plesalo, ale nám zatiaľ budú stačiť iba nejaké základné veci: funkcia `random()`, ktorá vyrobí náhodné reálne číslo z intervalu $\langle 0,1)$, funkcia `randint(a,b)`, ktorá vyrobí náhodné celé číslo z intervalu $\langle a, b \rangle$ (`randint(1,6)` bude teda fungovať rovnako, ako hracia kocka), funkcia `choice`, ktorá dostane na vstupe zoznam a vyberie z neho náhodný prvok a funkcia `shuffle`, ktorá dostane na vstupe zoznam a náhodne v ňom poprehadzuje prvky.

Radi by sme tieto funkcie používali v našom programe. Ako presvedčiť Python, aby nám to dovolil? Je viacero možností.

Prvá je tá, že do príkazového riadku, alebo do programu napíšeme

```
import random
```

To spôsobí, že sa k vášmu programu pripojí celý modul `random` a vy môžete používať jeho funkcie tak, že napíšete meno modulu, bodku a meno funkcie. Kus programu teda môže vyzerať napríklad takto:

```
import random

a = ['a','b','c','d','e']
random.shuffle(a)
print(a)

print(random.choice(a))
print(random.random())
print(random.randint(1,6))
```

Úloha 2: Vyskúšajte si to.

Druhá možnosť je nenačítať celý modul, ale iba tie funkcie, ktoré budete potrebovať. Ak by sme chceli načítať iba tie funkcie, ktoré sme použili v predošlej ukážke, spravili by sme to príkazom

```
from random import shuffle, choice, random, randint
```

Tento zápis má oproti predošlému dve výhody. Prvá je tá, že nemusíte načítavať milión funkcií, keď vám stačia štyri. Druhá je tá, že potom v programe nemusíte vypisovať pred meno každej funkcie to `random` s bodkou. V tejto druhej podobe by teda predošlý program vyzeral takto:

```
from random import shuffle, choice, random, randint

a = ['a','b','c','d','e']
shuffle(a)
print(a)

print(choice(a))
print(random())
print(randint(1,6))
```

To, že nemusíme vypisovať meno modulu, môže byť ale dvojsečná zbraň. Ak totiž vyrobíte funkciu, ktorá sa volá `choice` aj vo vašom programe[9], bude sa vám s tou funkciou z modulu pliesť. Ak toto riziko hrozí, odporúča sa držať skôr toho prvého prístupu.

Úloha 3: Vyskúšajte si aj tento druhý postup. Dalo to rovnaké výsledky, ako v predošlej verzii?

Tretia možnosť je, že z modulu načítate všetky funkcie. Robí sa to príkazom

```
from random import *
```

Tento spôsob ale používajte čo najmenej, pretože takto si do svojho programu naťaháte množstvo nových funkcií a ak ste si neprečítali dokumentáciu poriadne, tak o polovici z nich ani netušíte, že ako sa volajú. A riziko, že sa vám pomotajú mená funkcií z modulu s vašimi funkciami, je naozaj veľké.

Úloha 4: Toto radšej ani neskúšajte.

9 O tom, ako si vyrobiť vlastné funkcie, ešte bude reč.

Ďalšia knižnica, ktorú v tejto lekcii spomenieme, je knižnica `datetime`. Ako už názov napovedá, s jej pomocou sa dá dobre pracovať s dátumami a časmi.

Úloha 5: Pozrite si manuál ku knižnici `datetime`, buď v nápovednom systéme alebo na internete – podľa toho, čo vám viac vyhovuje. Zvlášť dobre si pozrite, ako fungujú objekty `date` a `timedelta`.

Úloha 6: Napíšte program, ktorý vypočíta, koľko máte dnes dní.

Úloha 7: Napíšte program, ktorý vypočíta, aký dátum bude o sto dní odteraz.

Úloha 8: Napíšte program, ktorý vypočíta, na aký deň v týždni padnú Vianoce v roku 3000.

Úloha 9: Napíšte program, ktorý šesťstokrát hodí kockou a vypíše vám, koľkokrát padla šestka.

Úloha 10: Napíšte program, ktorý predošlý pokus päťdesiatkrát zopakuje, výsledky uloží do zoznamu a ten potom utriedi od najmenších hodnôt po najväčšie a vypíše.

Úloha 11: Napíšte program, ktorý z čísel 1 až 49 vyberie sedem náhodných rôznych čísel.

Úloha 12: Načítajte modul `antigravity` (príkaz `import antigravity`) a pozrite sa, čo to spraví.

6. lekcia
Funkcie
alebo „ako funguje evolúcia“

Učiť sa programovať je v istom zmysle podobné, ako učiť sa jazyk. Najprv sa naučíte nejaké slovíčka, pochytíte základy gramatiky, potom sa učíte hovoriť niečo zmysluplné a nakoniec pochopíte logiku jazyka a ste schopní v ňom myslieť. Predošlé lekcie boli o tých prvých slovíčkach a gramatike. V tejto lekcii sa už ale pokúsime naprogramovať niečo zmysluplnejšie. Samozrejme sa popri tom aj niečo nové naučíme.

To niečo nové budú funkcie. Funkcia je kus kódu, ktorý má svoj osobitý zmysel a oplatí sa ho kvôli prehľadnosti pomenovať. Takýto kus kódu môžeme použiť na viacerých miestach v ďalšom programe iba tak, že zavoláme meno funkcie a nemusíme tam prepisovať celý ten kód. Okrem toho niektoré funkcie môžu vrátiť výsledok, ktorý môžeme neskôr použiť. Po dočítaní tejto lekcie by malo byť zrejmé, že funkcie nám môžu značne spríjemniť život a spôsobiť, že celý program bude pôsobiť zmysluplnejším dojmom, než bolo doteraz zvykom.

V tejto lekcií sa totiž pokúsime naprogramovať evolúciu. Bude to evolúcia úplne umelá, ktorá s tým, čo sa deje v prírode, nebude mať veľa spoločného, namiesto štyroch typov báz, ktoré sa vyskytujú v DNA budeme používať čísla od 0 do 9 a namiesto zhruba troch miliárd báz od každého z rodičov, ako to majú ľudia, ich naše virtuálne potvory budú mať iba päť.

Najprv si teda spravíme funkciu, ktorá nám náhodnú virtuálnu potvoru vyrobí. Keďže sa budeme často hrať s náhodou, budeme potrebovať nejaké funkcie z modulu `random`:

```
1   from random import random, randint, choice
2
3   def potvora():
4       p = []
5       for i in range(5):
6           p.append(randint(0,9))
7       return p
```

Na riadku 3 začína definícia našej prvej funkcie `potvora`. Funkciu vytvárame kľúčovým slovom `def` po ktorom nasleduje meno funkcie, zátvorky, v ktorých môžu byť uvedené parametre a dvojbodka. Vytvoríme si prázdny zoznam `p` a potom do neho vložíme päť náhodných čísel od 0 do 9. Tento zoznam potom na riadku 7 funkcia vyhlási pomocou príkazu `return` za svoj výsledok. Funkcia sa ukončí buď príkazom `return`, alebo tak, že príde na koniec.

Keď chceme našu novú funkciu vyskúšať, môžeme po funkcii vložiť testovací kód, v ktorom ju použijeme. Ten už samozrejme nebude odsadený o tabulátor. Keď teda chceme vidieť výsledok, ktorý funkcia `potvora()` vráti, môžeme použiť príkaz

```
print(potvora())
```

Úloha 1: Vyskúšajte. Vyskúšajte viackrát, nech vidíte, aké pekné potvory naša funkcia generuje. Potom testovací kód vymažte a v programe si nechávajte len hotové funkcie – nabudúce budeme testovať zas niečo ďalšie.

Potvory sa nám generujú pekne. Lenže nie je potvora ako potvora. Niektoré sú krajšie ako iné. Totiž – najkrajšia potvora, aká môže existovať, je potvora `[1, 2, 3, 4, 5]`. Nepýtajte sa,

prečo. Potvory skrátka majú taký vkus. No a potvora je taká pekná, na koľkých miestach sa zhoduje s najkrajšou potvorou. Najkrajšia potvora má teda krásu 5, potvora `[1, 4, 7, 9, 5]` má krásu 2, pretože sa s najkrajšou potvorou zhoduje na prvom a poslednom mieste (to že má niekde v sebe ešte aj štvorku jej nepomôže – nemá ju na správnom mieste) a potvora `[9, 3, 6, 0, 1]` má krásu 0. Chudinka.

Teraz si urobíme funkciu `krasa(p)`, ktorá bude mať za úlohu zistiť, aká krásna je zadaná potvora:

```
 8   def krasa(p):
 9       najkrajsiaPotvora = [1, 2, 3, 4, 5]
10       k = 0
11       for i in range(5):
12           if p[i] == najkrajsiaPotvora[i]:
13               k += 1
14       return k
```

Táto funkcia má vstupný parameter – premennú `p`. Do tejto premennej si uloží potvoru, ktorú hodlá testovať. Na riadku 9 si zadefinuje najkrajšiu potvoru a na riadku 10 nastaví premennú `k`, v ktorej bude počítať, aká je potvora krásna, na nulu. V cykle na riadkoch 11 až 13 funkcia prejde všetky pozície v genetickom kóde a ak sa testovaná potvora na danej pozícii zhoduje s najkrajšou potvorou, zvýši krásu o 1.[10] Keď prebehne celý cyklus, krása je vypočítaná a funkcia ju vráti ako svoju hodnotu.

Funkciu môžete otestovať nasledujúcim testovacím kódom:

```
a = potvora()
print(a, krasa(a))
```

Vyrobenú potvoru si v ňom uložíme do premennej `a` a vypíšeme ju spolu s jej krásou. Všimnite si, že pri volaní funkcie `krasa` sa hodnota z premennej `a` vložila do premennej `p`. Funkcie vo všeobecnosti majú svoje súkromné premenné a keď môžu, používajú tie. Premenné zvonku môžu použiť tiež, ale iba vtedy, keď nemajú inú možnosť.

Úloha 2: Vyskúšajte to. Akú najkrajšiu potvoru sa vám podarilo vygenerovať?

Potvory sú spoločenské živočíchy, ktoré žijú v svorkách. Každá svorka pozostáva z desiatich potvor. Nasledujúca procedúra nám takú svorku vyrobí:

```
15   def svorka():
16       s = []
17       for i in range(10):
18           s.append(potvora())
19       return s
```

Úloha 3: Pochopte, ako funkcia funguje a otestujte ju. Testovací kód si vymyslite.

Keď ste v predošlej úlohe vypisovali svorku, výpis bol pravdepodobne neveľmi čitateľný. Preto by bolo fajn urobiť si funkciu, ktorá vypíše svorku krajšie – každého člena do jedného riadku a podľa možností k nemu aj napíše, aký je krásny.

---

10 Tu sme si dovolili použiť programátorskú skratku: Namiesto príkazu `k = k + 1`, ktorý do premennej `k` vloží hodnotu o 1 väčšiu, než tam bola predtým, sme použili príkaz `k += 1`, ktorý premennú vľavo zväčší o hodnotu vpravo, čiže spraví to isté, len musíme menej písať.

```
20  def vypis(s):
21      for pot in s:
22          print(pot, krasa(pot))
```

Táto funkcia dostane na vstupe svorku a pre každého jej člena vypíše patričnú potvoru aj jej krásu. Otestujte si ju, kým budete čítať ďalej. Však je výpis krajší, ako keď svorku vypíšete s pomocou jediného `print`?

Keďže sa jedná o evolúciu, patrilo by sa povedať viac o tom, ako to majú potvory zariadené s potomstvom. Veci sa majú tak, že ak chce mat potvora potomstvo, môže ale nemusí si nájsť partnera. Ak si partnera nenájde, má potomkov sama so sebou.[11] Potomok zdedí každú bázu po jednom z rodičov a iba náhoda rozhoduje, že po ktorom. Okrem toho ale ešte môže dochádzať k mutáciám. S pravdepodobnosťou 8% môže každá báza nadobudnúť úplne náhodnú hodnotu.

Potomka dvom potvorám vyrobí nasledujúca funkcia:

```
23  def potomok(a,b):
24      mlade = []
25      for i in range(5):
26          if randint(0,1) == 0:
27              mlade.append(a[i])
28          else:
29              mlade.append(b[i])
30      for i in range(5):
31          if random() < 0.08:
32              mlade[i] = randint(0,9)
33      return mlade
```

Funkcia si najprv vyrobí prázdny zoznam `mlade`. Potom si na riadkoch 25 až 29 päťkrát hodí mincou (vygeneruje náhodné číslo 0 alebo 1) a ak padne 0, prilepí mladému bázu od rodiča `a` a ak padne 1, prilepí mladému bázu od rodiča `b`.[12] V cykle na riadkoch 30 až 32 prebehnú mutácie. Ak generátor náhodných čísel vygeneruje číslo, ktoré je menšie, ako 0,08 (pravdepodobnosť, že sa to stane, je práve 8%), tak sa báza zmení na nejakú úplne náhodnú hodnotu. Hotové mladé potom funkcia vráti ako hodnotu.

Úloha 4: Otestujte. Odporúčaný testovací kód je

```
print(potomok([0, 0, 0, 0, 0],[1, 1, 1, 1, 1]))
```

Ako často dochádza k mutáciám? Častejšie alebo zriedkavejšie, než ste čakali?

Spoločenský život svorky funguje tak, že potvory, ktoré patria do krajšej polovice svorky si vyberú partnera (vyberú si náhodne – láska krásnych potvor je slepá) a s ním majú dvoch potomkov. Ak si vyberú niekoho z krajšej polovice, tak ten si potom tiež ešte vyberá svojho partnera. Do ďalšej generácie tak prejde presne toľko potvor, koľko bolo v predošlej. Funkcia `novasvorka(s)` dostane na vstup svorku a vyrobí z nej podľa týchto pravidiel ďalšiu generáciu:

```
34  def novasvorka(s):
35      s = sorted(s, key = krasa, reverse = True)
36      nova = []
37      for i in range(int(len(s) / 2)):
38          partner = choice(s)
39          nova.append(potomok(s[i],partner))
40          nova.append(potomok(s[i],partner))
41      return nova
```

11 Volá sa to partenogenéza a v biologickom svete to zvládajú napríklad vošky.

12 Podobný mechanizmus sa v prírode volá crossing over a deje sa pri vzniku pohlavných buniek.

Na riadku 35 funkcia zoradí svorku podľa krásy. Keďže potrebujeme mať svorku zoradenú od najväčšej krásy po najmenšiu, treba pri triedení nastaviť opačné poradie. Na riadku 36 vyrobíme premennú pre novú svorku.

V cykle na riadkoch 37 až 40 pridávame do novej svorky potomkov potvor zo starej svorky. Funkcia `len(s)` nám prezradí, koľko členov svorka má. Túto hodnotu vydelíme dvomi a zmeníme na celé číslo. Takto prejdeme len krajšiu polovicu svorky. Každému členovi vyberieme na riadku 38 náhodného partnera a do novej svorky pridáme ich dvoch potomkov. Výslednú svorku funkcia vráti ako svoju hodnotu.

A môžeme spustiť evolúciu. Na začiatku vygenerujeme náhodnú svorku a kým si to nerozmyslíme, môžeme počítať ďalšie a ďalšie generácie. Keď sme už v tejto lekcii všetko dávali do funkcií, dajme tam aj samotnú evolúciu:

```
42  def evolucia():
43      s = svorka()
44      generacia = 0
45      koniec = False
46      while not koniec:
47          print("Generácia: ",generacia)
48          vypis(sorted(s, key=krasa, reverse = True))
49          vs = input(":")
50          if vs == 'x':
51              koniec = True
52          s = novasvorka(s)
53          generacia += 1
```

V premennej `generacia` si budeme počítať, ku koľkej generácii sme sa dopracovali. Na riadku 48 vypíšeme zoradenú svorku, na riadku 49 sa vždy opýtame, či už chce používateľ končiť, ak zadá čokoľvek okrem `x` (teda napríklad aj keď iba stlačí `Enter`), vyrobíme novú svorku a zvýšime hodnotu v premennej `generacia` o 1.

Na záver ešte túto funkciu treba spustiť. Nezabudnite vložiť riadok

```
54  evolucia()
```

Úloha 5: Pochopte a otestujte. Ako sa svorka mení? Objaví sa niekedy potvora s krásou 4? Objaví sa niekedy potvora s krásou 5?

Úloha 6: Vyrobte funkciu `priemernaKrasa(s)`, ktorá dostane ako vstupný parameter svorku a vypočíta jej priemernú krásu. Upravte funkciu `evolucia` tak, aby si do zoznamu ukladala priemernú krásu v každej generácii. Nakoniec ten zoznam vypíšte.

Úloha 7: Upravte funkciu `novasvorka` tak, aby boli dve najkrajšie potvory obdarené dlhovekosťou a automaticky prešli do novej svorky. Partnera si budú potom vyberať iba štyri najkrajšie potvory (vrátane tých dlhovekých). Ako sa bude vyvíjať svorka v takomto prípade?

Úloha 8: Vymyslite si nejakú zaujímavú úpravu celého procesu a uskutočnite ju.

Náš príklad evolučného algoritmu bol netypický tým, že bolo dopredu jasné, ako má vyzerať skvelý jedinec a k čomu to všetko má smerovať. Ak sa ale podobné algoritmy spoja s nejakou fyzikálnou simuláciou, situácia sa stáva ešte oveľa zaujímavejšou. Svedčia o tom aj nasledujúce tri videá:

http://www.youtube.com/watch?v=Xe_euHneE0I
http://www.youtube.com/watch?v=JBgG_VSP7f8
http://www.youtube.com/watch?v=cP035M_w82s

7. lekcia

Objekty
alebo „dlhá lekcia o sardinkách a žralokoch“

V predošlej lekcii sme si ukázali, ako si veľký program rozdeliť na menšie úlohy a s pomocou funkcií sme tak zvládli napísať niečo, čo by sa v jednom obrovskom programe robilo naozaj zle. Funkcie sú z tohto hľadiska náramne užitočná vec.

V tejto lekcii si ale ukážeme prístup, ktorý sa v niektorých situáciách osvedčil ešte oveľa viac. Namiesto toho, aby sme vytvárali funkcie, budeme vytvárať objekty. Objekty budú mať nejaké svoje vlastnosti aj nejaké svoje funkcie, s pomocou ktorých budú vedieť medzi sebou komunikovať. A keď to všetko spustíme, bude to robiť presne to, čo potrebujeme.

Poďme ale na vec postupne. Začnime klasickým príkladom. Predstavte si, že idete programovať veľkú databázu zamestnancov pre nejakú firmu. Začneme niečim skromným. Vytvoríme si triedu `Zamestnanec`, ktorá bude obsahovať tri údaje: meno, priezvisko a plat. Keď je trieda hotová, môžeme si začať vytvárať objekty danej triedy a s nimi robiť, čo uznáme za vhodné. Dobre to vidno na nasledujúcom príklade:

```
 1  class Zamestnanec:
 2      meno = ""
 3      priezvisko = ""
 4      plat = 0
 5
 6  jozo = Zamestnanec()
 7  jozo.meno = "Jozef"
 8  jozo.priezvisko = "Mrkvička"
 9  jozo.plat = 753.20
10
11  print("%s %s má plat %.2f euro" %
12            (jozo.meno, jozo.priezvisko, jozo.plat))
```

Na riadkoch 1 až 4 sme si vytvorili triedu `Zamestnanec` a jej jednotlivé atribúty sme zatiaľ nastavili na prázdne reťazce a na nulu. („Atribút“ je také učené pomenovanie pre premennú nejakého objektu.) Na riadku 6 sme si vytvorili objekt `jozo`, ktorý je triedy `Zamestnanec`, tým pádom už má vytvorené vnútorné premenné `jozo.meno`, `jozo.priezvisko` a `jozo.plat`. Tie majú ale zatiaľ štandardné hodnoty nastavené na riadkoch 2 až 4. Na riadkoch 7 až 9 tieto hodnoty prepíšeme a na riadkoch 11 a 12 vypíšeme o Jožovi dôverné informácie.

Všimnite si percentovú fintu, ktorú sme použili pri výpise. Ak v reťazci použijeme špeciálne sekvencie, môžeme Pythonu povedať, nech ich nahradí hodnotami z n-tice za reťazcom. Operácia, ktorá má to nahradenie na svedomí má znak `%`. Je na riadku 11 na konci. V našom prípade sme použili dve sekvencie `%s`, ktoré budú nahradené reťazcom (z anglického `string`) a sekvenciu `%.2f`, ktorá bude nahradená desatinným číslom (z anglického `float`), ktoré sa vypíše s presnosťou dve miesta za desatinnou čiarkou, takže Janov plat vypíšeme s presnosťou na centy.[13]

Úloha 1: Vyskúšajte. Vytvorte si ďalších dvoch zamestnancov a všetkých vypíšte.

13 Tí, ktorí programovali v jazyku C, si iste spomínajú, že podobné špeciálne sekvencie používal príkaz `printf`.

Prvou výhodou objektov je, že ich môžete ľahko upravovať a pridávať im funkcie podľa vlastnej úvahy. Napríklad je celkom nešikovné, že keď chceme vypísať informácie o plate zamestnanca, zakaždým musíme napísať pomerne veľký príkaz. Oveľa šikovnejšie by bolo, keby objekt zamestnanec mal metódu `vypis()`, ktorá sa o všetko postará sama. („Metóda“ je také učené pomenovanie pre funkciu nejakého objektu.) Zariadiť to ale nie je problém.

```
 1   class Zamestnanec:
 2       meno = ""
 3       priezvisko = ""
 4       plat = 0
 5       def vypis(self):
 6           print("%s %s má plat %.2f euro" %
 7                   (self.meno, self.priezvisko, self.plat))
 8
 9   jozo = Zamestnanec()
10   jozo.meno = "Jozef"
11   jozo.priezvisko = "Mrkvička"
12   jozo.plat = 753.20
13   jozo.vypis()
```

Funkciu sme definovali vo vnútri triedy `Zamestnanec` na riadkoch 5 až 7. Použitá je na riadku 13. Všimnite si, že funkcia má jeden parameter s názvom `self`. Ak definujeme funkciu vo vnútri triedy, prvý parameter tejto funkcie bude vždy objekt danej triedy, s ktorým sa práve pracuje a až prípadné ďalšie parametre sa dajú nastaviť pri volaní funkcie. Ten prvý parameter si môžete nazvať, ako chcete, ale použiť `self` patrí k dobrému Pythonovskému programátorskému vychovaniu a keď bude po vás niekto program čítať, bude hneď vedieť, o čo sa jedná. Tento parameter sme využili na riadku 7. Ak bude teda funkciu volať objekt `jozo`, vypíše to údaje o Jožovi, ak máte vytvorený objekt `fero`, vypíše to údaje o Ferovi.

Ďalšia nešikovná vec je, že hneď ako na riadku 9 vytvoríte nového zamestnanca, musíte na ďalších troch riadkoch nastavovať nové hodnoty. Keby sa tieto dva procesy dali spojiť, bolo by to stručnejšie aj prehľadnejšie.

Naše problémy vyrieši skvelá funkcia `__init__`. (Názov je „dva podtržníky, init a zase dva podtržníky“.) Keď totiž nejaká trieda má funkciu s týmto menom, zavolá ju automaticky vždy, keď sa vytvára nový objekt tejto triedy. Takáto funkcia sa nazýva konštruktor. Použije sa nasledujúcim spôsobom:

```
 1   class Zamestnanec:
 2
 3       def vypis(self):
 4           print("%s %s má plat %.2f euro" %
 5                (self.meno, self.priezvisko, self.plat))
 6
 7       def __init__(self, meno, priezvisko, plat = 337.70):
 8           self.meno = meno
 9           self.priezvisko = priezvisko
10           self.plat = plat
11
12   jozo = Zamestnanec("Jozef", "Mrkvička", 753.20)
13   fero = Zamestnanec("František", "Otruba")
14   jozo.vypis()
15   fero.vypis()
```

Konštruktor sme vytvorili na riadkoch 7 až 10. Má štyri parametre. Prvý je objekt, s ktorým sa práve pracuje. Ďalšie sú meno, priezvisko a plat, ktorý chceme objektu nastaviť. Na riadkoch 8 až 10 vytvoríme objektu atribúty `meno`, `priezvisko` a `plat` a nastavíme ich na hodnoty, ktoré dostane funkcia ako parametre.

Keď teraz vyrábate nový objekt, parametre, ktoré použijete pri jeho výrobe, sa dostanú priamo konštruktoru. To, že sú tie parametre iba tri a konštruktor ich má až štyri, nevadí. Prvý parameter konštruktora je odkaz na objekt, takže hodnoty sa dosadia až od druhého parametra ďalej.

Ako si môžete všimnúť, posledný parameter konštruktora má nastavenú štandardnú hodnotu 337,70 eura (čo je minimálna mzda). To znamená, že ak sa posledný parameter neuvedie, použije sa táto hodnota. Vyskúšali sme to, keď sme vyrábali objekt `fero`.

Úloha 2: Vyskúšajte a pochopte.

Naša trieda `Zamestnanec` by sa samozrejme ešte dala vylepšovať mnohými spôsobmi a dali by sa jej pridávať ďalšie a ďalšie postupnosti a funkcie. Toto vylepšovanie je jedna z vecí, ktoré sú na objektovom prístupe k programovaniu fajn. Zanechajme ju ale svojmu osudu a skúsme naprogramovať zase niečo inšpirované biológiou.

Namiesto programovania ale začneme popisom toho, čo chceme programovať. Popis budeme stále spresňovať, až kým nebude zrejmé, aké objekty budeme v našom programe potrebovať, aké majú mať atribúty a čo budú mať za úlohu a až potom sa pustíme do programovania.

Budeme chcieť naprogramovať more. V tom mori budú žiť dva druhy rýb: sardinky a žraloky.[14] Sardinky žerú planktón a množia sa. Keďže planktónu je dosť, sardinky by časom zaplnili celé more. Žraloky žerú sardinky. Ak sú nažraté, tiež sa množia, ale ak nažraté nie sú, tak uhynú.

Keď si pozorne prečítate predošlý odsek, zistíte, že sa v ňom vyskytujú tieto podstatné mená: more, ryby, sardinky, žraloky, planktón. Planktónu sa náš program venovať nebude. Použili sme ho iba na vysvetlenie toho, že nám sardinky pribúdajú len tak. Ostatné štyri veci sú horúci kandidáti na to, aby sme z nich spravili triedy nášho programu. Aby sme situáciu trochu zjednodušili, nebudeme vytvárať ani triedu ryba, ktorá by v sebe zahŕňala všetko, čo majú sardinky a žraloky spoločné.[15] Ostali nám teda tri triedy, ktoré bude treba vytvoriť: sardinka, žralok a more.

More bude reprezentované štvorčekovým papierom a na každom políčku sa buď bude nachádzať sardinka, žralok, alebo tam nebude nič. Poďme sa pozrieť podrobnejšie na to, čo budú musieť jednotlivé triedy zvládnuť.

Trieda `Sardinka` si bude musieť o sebe pamätať svoj typ – bude to písmenko `'s'`, aby bola odlíšiteľná od žraloka, bude si pamätať objekt `more`, do ktorého patrí, bude si pamätať súradnice, na ktorých sa práve nachádza, svoj vek, čas, v ktorom dosiahne dospelosť (pri sardinke to budú tri kolá), interval, v akom sa môže množiť (raz za dve kolá) a čas, pred akým sa naposledy množila.

Ďalšia vec, nad ktorou sa bude treba zamyslieť, sú veci, ktoré bude musieť sardinka dokázať. Bude sa musieť vedieť pohnúť na niektoré susedné políčko (metóda `pohyb`) a bude sa musieť vedieť rozmnožiť (metóda `potomstvo`).

14 Žraloky v skutočnosti nie sú ryby, ale paryby.

15 Vytvoriť takýto objekt ale môže mať zmysel. Keď dokončíte túto lekciu, môžete sa na toto miesto vrátiť a premyslieť si, ako by objekt `Ryba` vyzeral a aké by bolo jeho miesto v programe. Prerobiť program týmto spôsobom by dokonca mohlo byť v istom zmysle prirodzenejšie.

Keď sme si toto všetko rozmysleli, môžeme sa pustiť do programovania sardinky. Pri písaní kódu budeme využívať aj funkcie triedy `More` o ktorej zatiaľ veľmi nebola reč. Pekné na tom je, že ani nie je nutné, aby bola. Na to, že sa nám nejaká metóda triedy `More` hodí, môžeme prísť počas vytvárania sardinky a pokojne ju môžeme použiť. Len ju potom nesmieme v triede `More` zabudnúť vytvoriť. Kód sardinky teda bude vyzerať takto:

```
 1   from random import choice
 2
 3   class Sardinka:
 4        """Trieda reprezentujúca obeť"""
 5        def __init__(self, more, x = -1, y = -1):
 6             self.more = more
 7             if x == -1 and y == -1:
 8                  volnemiesta = more.volneMiesta()
 9                  if volnemiesta == []:
10                       raise Exception("Plné more!!!")
11                  x,y = choice(volnemiesta)
12             self.x = x
13             self.y = y
14             self.interval = 2
15             self.posmnoz = 0
16             self.dospelost = 3
17             self.vek = 0
18             self.typ = 's'
19        def _mlade(self,x,y):
20             s = Sardinka(self.more,x,y)
21             return s
22        def pohyb(self):
23             volne = self.more.blizkeVolno(self.x,self.y)
24             if volne != []:
25                  x,y = choice(volne)
26                  self.more.presun(self, x, y)
27             self.vek += 1
28             self.posmnoz += 1
29        def potomstvo(self):
30             if self.vek >= self.dospelost and self.posmnoz >= self.interval:
31                  volne = self.more.blizkeVolno(self.x,self.y)
32                  if volne != []:
33                       x,y = choice(volne)
34                       novaRyba = self._mlade(x,y)
35                       self.more.pridajRybu(novaRyba)
36                       self.posmnoz = 0
```

Konštruktor má za úlohu nastaviť sardinke všetky atribúty. Má okrem parametra `self` ďalšie tri parametre. Prvý označuje more, do ktorého bude sardinka patriť, ďalšie dva označujú súradnice, kde sa má vytvoriť. V prípade, že pri volaní funkcie neboli parametre zadané, alebo majú obe súradnice hodnotu `-1`, vypýtame si na riadku 8 od mora súradnice všetkých voľných miest (predpokladáme, že metóda triedy `More` `volneMiesta` vráti zoznam párov čísel) a jedno z nich náhodne vyberieme (riadok 11). Ak je zoznam prázdny a teda už v mori žiadne miesto nezostalo, vyhlásime chybu (riadok 10). Ak chybu nezachytíme pomocou `try`, ako to bolo popísané v druhej lekcii, program sa ukončí. Ak nejaké voľné miesto máme, vyberieme z voľných miest náhodné pomocou funkcie `choice` (riadok 11). Atribút `self.interval` určuje, ako často sa môže sardinka množiť, atribút `self.posmnoz` hovorí, kedy sa sardinka naposledy množila, sardinka dosiahne dospelosť vo veku `self.dospelost` a jej aktuálny vek je `self.vek`.

Úloha 3: Metóda `pohyb` je dobre čitateľná aj bez komentára. Pokúste sa pochopiť, čo robí, ešte pred tým, ako budete čítať ďalej.

Na riadku 23 sme zavolali metódu `blizkeVolno`, v ktorej nám more vráti zoznam všetkých voľných políčok v okolí zadaných súradníc. Ak ten zoznam nie je prázdny, vyberieme z neho náhodný prvok a požiadame more, aby rybu na tú pozíciu presunulo. Na záver zväčšíme hodnoty atribútov `vek` a `posmnoz` o 1, pretože pre sardinku uplynulo jedno kolo.

Metóda `potomstvo` funguje podobne. Najprv skontroluje, že či je ryba už dospelá a či nemala potomstvo príliš nedávno (riadok 30). Potom zistí, či je naokolo nejaké voľné miesto, kam by sa dalo rozmnožiť (riadky 31 a 32). Ak voľné miesta sú, tak z nich jedno náhodne vyberie a pomocou metódy `_mlade` vytvorí novú sardinku so zadanými súradnicami. Zavolá metódu mora `pridajRybu`, s pomocou ktorej si more všimne, že pribudla nová ryba. Na záver nastaví sardinke čas posledného množenia na 0.

Všimnite si metódu `_mlade`, ktorú sme volali z metódy `potomstvo`. Táto metóda nerobí nič iné, len to, že vytvorí novú sardinku. Nepredpokladá sa, že by ju používal niekto mimo triedy `Sardinka`. Takéto metódy sa podľa konvencie označujú tak, že ich meno začína podtržníkom. Python vám nebude brániť takúto metódu zavolať, ale autori triedy tak dávajú najavo, že táto metóda nie je určená na verejné používanie. Načo sme vlastne túto metódu vytvárali, sa dozviete o chvíľočku.

Poďme sa teraz venovať žralokom. Všetky veci, ktoré fungovali pri sardinke, budú fungovať aj pri žralokovi. Žralokovi bude trvať inú dobu, kým dospeje a bude mať iný interval množenia, ale základnú funkčnosť bude mať rovnakú ako sardinka. Až na detaily.

Jednak si bude treba zapamätať, aký je žralok momentálne nažratý, pričom treba dbať na to, že keď toho žralok zožerie priveľa, už sa mu nažratosť ďalej nezvyšuje. Ďalej si treba uvedomiť, že sa žralok hýbe trochu inak, ako sardinka. Najprv sa pozrie, či sa v jeho okolí nachádzajú nejaké sardinky a ak áno, tak žerie. Ak tam žiadne sardinky nie sú, tak sa presunie na susedné voľné miesto. Poďme sa pozrieť na kód žraloka:

```
37  class Zralok(Sardinka):
38      """Trieda reprezentujúca predátora"""
39      def __init__(self,more, x = -1, y = -1):
40          Sardinka.__init__(self,more,x,y)
41          self.interval = 3
42          self.dospelost = 4
43          self.typ = 'Z'
44          self.nazratost = 1
45          self.maxnazratost = 2
46      def _mlade(self,x,y):
47          s = Zralok(self.more,x,y)
48          return s
49      def pohyb(self):
50          sardinka = self.more.blizkaSardinka(self.x,self.y)
51          if sardinka == []:
52              self.nazratost -= 1
53              if self.nazratost < 0:
54                  self.more.uhyn(self)
55              else:
56                  Sardinka.pohyb(self)
57          else:
58              x,y = choice(sardinka)
59              self.more.zozer(self,x,y)
60              self.vek += 1
61              self.posmnoz += 1
```

Sympatický je hneď začiatok celého kódu triedy. Tým, že napíšeme `class Zralok(Sardinka)` zabezpečíme, že trieda `Zralok` zdedí od triedy `Sardinka` celú jej funkcionalitu, takže ju nemusíme robiť nanovo. (Trieda `Zralok` sa preto zvykne nazývať potomok triedy `Sardinka`. Vo veľkých projektoch sa takto dajú vytvoriť celé hierarchie objektov.) Znamená to napríklad, že trieda `Zralok` bude mať automaticky metódu `potomstvo` a my pre to nemusíme ani prstom pohnúť.

Niečo ale nanovo budeme musieť spraviť. V prvom rade bude treba napísať nový konštruktor pre žraloka. Na riadku 40 zavoláme konštruktor zo sardinky, ktorý nám ponastavuje súradnice a ostatné veci. Všimnite si jeden detail – ak voláme metódu nejakého iného objektu, hodnotu prvého parametra neuvádzame, lebo sa automaticky nahradí objektom, ktorý voláme. Ak voláme metódu predka našej triedy (ako v tomto prípade `__init__` zo sardinky), musíme uviesť všetky parametre, teda aj ten `self` na začiatku.

Po tom, ako sme zavolali konštruktor zo sardinky, ešte prestavíme veci, ktoré má žralok inak – typ, dobu dospievania a interval, v ktorom sa žraloky množia. Nakoniec nastavíme počiatočnú nažratosť žraloka a maximálnu nažratosť, ktorú môže dosiahnuť.

Úloha 4: Poriadne si prečítajte žralokovu metódu `pohyb` a príďte na to, čo robí.

Vráťme sa k metóde `_mlade`, ktorú sme spomínali už pri sardinke. Pri žralokovi sme ju zmenili tak, že namiesto novej sardinky vyrobí nového žraloka. Načo nám tá metóda presne je? Pointa je v tom, že sardinka aj žralok majú tú istú metódu `potomstvo`. A keď tam na riadku 34 generujeme nové mláďa, tak nevieme, či to má byť sardinka, alebo žralok. Preto zavoláme metódu `_mlade`. Tá nám v prípade, že ju voláme zo sardinky vyrobí sardinku a ak ju voláme zo žraloka, vyrobí nám žraloka.

Pred nami je posledná trieda `More`. Pripomeňme si metódy, ktoré sme použili v doterajšom kóde a ktoré budeme musieť naprogramovať:

- `volneMiesta(self)` má vrátiť zoznam všetkých voľných miest
- `blizkeVolno(self,x,y)` má vrátiť zoznam voľných miest v okolí pozície `[x,y]`
- `presun(self,ryba,x,y)` má presunúť rybu (teda sardinku alebo žraloka) určeného parametrom `ryba` na pozíciu `[x,y]`
- `pridajRybu(self,ryba)` pridá rybu (sardinku alebo žraloka) do mora
- `blizkaSardinka(self,x,y)` má vrátiť zoznam miest, na ktorých sú sardinky, v okolí pozície `[x,y]`
- `zozer(self,zralok,x,y)` má presunúť žraloka na pozíciu `[x,y]` a zrušiť rybu, ktorá sa na tej pozícii nachádzala.

Okrem týchto funkcií by sa hodili ešte nejaké ďalšie, ktoré na začiatku vedia pridať do mora počiatočné sardinky a žraloky, funkcia na výpis mora a funkcia, ktorá pohne každou sardinkou a žralokom a zariadi, že sa budú množiť.

Aké atribúty bude more obsahovať, ak chceme aby toto všetko mohlo robiť? V prvom rade to bude veľkosť. Keďže more bude štvorcové stačí nám jedno číslo. Potom budeme potrebovať štvorcovú tabuľku, v ktorej si budeme uchovávať, kde sú ryby, kde žraloky a kde je voľno. Takúto tabuľku vytvoríme ako zoznam, ktorého každý prvok bude zase zoznam. Ak teda budeme chcieť vedieť, čo sa nachádza na súradniciach `[2,6]`, pozrieme sa, akú hodnotu má `self.more[2][6]`.

V premennej `self.more[2]` je totiž uložený riadok číslo 2 a `self.more[2][6]` je jeho prvok číslo 6. Okrem toho budeme mať dva zoznamy, jeden na sardinky a druhý na žraloky.

Úloha 5: Nasleduje zdrojový kód triedy more. Pomaly a poriadne si ho prečítajte a pokúste sa prísť na to, čo to robí. Potom si svoje závery porovnajte s komentárom.

```
 62  class More:
 63      """Prostredie pre predátorov a obete"""
 64      def __init__(self,velkost = 20):
 65          self.more = []
 66          self.velkost = velkost
 67          for i in range(velkost):
 68              riadok = []
 69              for j in range(velkost):
 70                  riadok.append('.')
 71              self.more.append(riadok)
 72          self.sardinky = []
 73          self.zraloky = []
 74      def pridajRybu(self,ryba):
 75          if self.more[ryba.x][ryba.y] == '.':
 76              self.more[ryba.x][ryba.y] = ryba.typ
 77              if ryba.typ == 's':
 78                  self.sardinky.append(ryba)
 79              else:
 80                  self.zraloky.append(ryba)
 81      def pridajSardinku(self):
 82          self.pridajRybu(Sardinka(self))
 83      def pridajZraloka(self):
 84          self.pridajRybu(Zralok(self))
 85      def volneMiesta(self):
 86          vm = []
 87          for i in range(self.velkost):
 88              for j in range(self.velkost):
 89                  if self.more[i][j] == '.':
 90                      vm.append((i,j))
 91          return vm
 92      def _najdiBlizko(self,co,x,y):
 93          zoz = []
 94          for i in range(-1,2):
 95              for j in range(-1,2):
 96                  if i == 0 and j == 0:
 97                      continue
 98                  nx = (x+i) % self.velkost
 99                  ny = (y+j) % self.velkost
100                  if self.more[nx][ny] == co:
101                      zoz.append((nx,ny))
102          return zoz
103      def blizkeVolno(self,x,y):
104          return self._najdiBlizko('.',x,y)
105      def blizkaSardinka(self,x,y):
106          return self._najdiBlizko('s',x,y)
107      def presun(self,ryba,x,y):
108          self.more[ryba.x][ryba.y] = "."
109          self.more[x][y] = ryba.typ
110          ryba.x, ryba.y = x,y
```

```
111         def zozer(self,zralok,x,y):
112             for i in range(len(self.sardinky)):
113                 if self.sardinky[i].x == x and self.sardinky[i].y == y:
114                     self.sardinky.pop(i)
115                     break
116             self.more[zralok.x][zralok.y] = '.'
117             self.more[x][y] = zralok.typ
118             zralok.x, zralok.y = x,y
119         def uhyn(self,zralok):
120             self.zraloky.remove(zralok)
121             self.more[zralok.x][zralok.y] = "."
122         def vypis(self):
123             for i in range(self.velkost):
124                 s = ""
125                 for j in range(self.velkost):
126                     s += self.more[i][j] + " "
127                 print(s)
128         def pohyb(self):
129             for i in self.sardinky:
130                 i.pohyb()
131                 i.potomstvo()
132             for i in self.zraloky:
133                 i.pohyb()
134                 i.potomstvo()
```

V konštruktore nastavíme základné atribúty. Najviac priestoru zaberá nastavenie prázdneho mora (riadky 67 až 71). Všimnite si, že políčko, ktoré je prázdne reprezentuje znak `'.'`.

Metóda `pridajRybu` skontroluje, či je políčko, kam chceme pridať rybu prázdne a ak áno, bodku prepíše typom ryby, teda buď `'s'` alebo `'Z'`. Podľa typu ryby pridá rybu do správneho zoznamu.

Metódy `pridajSardinku` a `pridajZraloka` slúžia na počiatočné pridanie sardiniek a žralokov. Keďže voláme konštruktory `Sardinka` a `Zralok` iba s parametrom, ktorý určuje, do ktorého mora budú patriť a nezadávame súradnice, súradnice sa vygenerujú náhodne.

Metóda `volneMiesta` prehľadá celé more (s každou rozumnou hodnotou premennej `i` skontroluje všetky rozumné hodnoty premennej `j`) a ak je patričné políčko mora voľné, dvojicu `(i,j)` pridá do zoznamu `vm`. Nakoniec tento zoznam vráti ako výsledok.

Metódy `blizkeVolno` a `blizkaSardinka` robia v podstate to isté – niečo hľadajú v blízkosti zadaných súradníc. Aby sme to nemuseli programovať dvakrát, spravíme si pomocnú metódu, ktorá vyhľadá okolo bodu to, čo jej povieme a tú potom len zavoláme s rôznymi parametrami. Táto pomocná metóda je `_najdiBlizko` a okrem `self` má tri parametre: čo hľadáme a súradnice, okolo ktorých to hľadáme. Súradnice okolitých políčok sa od daného políčka líšia o `-1`, `0` alebo o `1`. Premenné `i` a `j` nadobudnú postupne všetky možné kombinácie hodnôt `-1`, `0` a `1`. V prípade, že sú obe premenné nulové, necháme pomocou `continue` veci tak, lebo vtedy to nie je susedné políčko, ale samotné políčko, ktorého susedov máme kontrolovať. Inak vypočítame súradnice susedného políčka `nx` a `ny`. Pri výpočte použijeme fintu

```
nx = (x+i) % self.velkost
```

(a jej proťajšok pre `y`), ktorá spôsobí, že sa za susedné budú považovať aj políčka, ktoré sa nachádzajú na opačných koncoch mapy. Pripomíname, že znak `%` je zvyšok po delení. A ak by veľkosť mapy bola napríklad `10`, tak `(9 + 1) % 10` by bolo `0` a `(0 + (-1)) % 10` by bolo `9`.

Súradnice všetkých miest, na ktorých nájdeme požadovaný objekt si budeme ukladať do zoznamu a ten zoznam na riadku 102 vrátime ako výsledok metódy.

Metóda `posun` nastaví v zozname `more` pôvodnú pozíciu ryby na `'.'`, na jej novú pozíciu vloží typ ryby a samotnej rybe zmení súradnice.

Metóda `zozer` prehľadáva zoznam sardiniek až kým nenájde takú, ktorá má zadané súradnice. Vyhodí ju zo zoznamu sardiniek a na jej v zozname `more` miesto presunie žraloka.

Metóda `uhyn` vyhodí určeného žraloka zo zoznamu žralokov a jeho miesto v mori označí, ako voľné.

Metóda `vypis` vypíše po riadkoch celé more.

Metóda `pohyb` najprv pohne a nechá sa rozmnožiť každú sardinku a potom pohne a nechá sa rozmnožiť každého žraloka.

Keď máme teda naše skvelé triedy hotové, poďme ich použiť. Napríklad takto:

```
135   m = More(30)
136   for i in range(60):
137       m.pridajSardinku()
138   for i in range(6):
139       m.pridajZraloka()
140   m.vypis()
141   while (input(':') != 'q'):
142       m.pohyb()
143       m.vypis()
```

Vytvoríme more 30 × 30 políčok a nasadíme do neho 60 sardiniek a 6 žralokov. Necháme more vypísať. Potom môžeme stláčať enter a more samo sa bude starať o to, aby sa všetko v ňom pohlo a aby nám vypísalo, čo potrebujeme. Keď nás to prestane baviť, zadáme `q`.

Úloha 6: Vyskúšajte to.

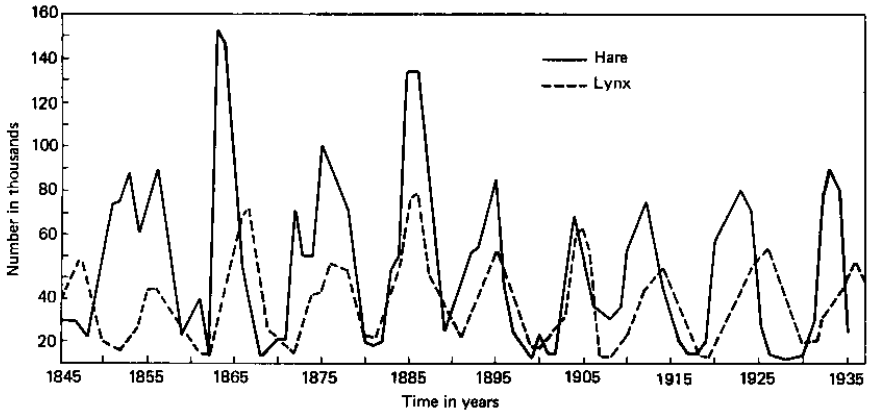
Úloha 7: Ako to vo vašom mori dopadlo? Všetko uhynulo? Ostali iba sardinky? Ani jedno, ani druhé? Spravte viacero pokusov. Pokúste sa upraviť program tak, aby zaznamenal do nejakých nových zoznamov počty sardiniek a žralokov v jednotlivých kolách. Skúste tie zoznamy vypísať, dostať ich do tabuľkového kalkulátora a nakresliť graf.

Úloha 8: Krehká ekologická rovnováha je v mori vtedy, keď nie je ani prázdne, ani úplne zasardinkované. Skúste meniť parametre – rozmery mora, úvodné počty sardiniek a žralokov, vlastnosti sardiniek a žralokov tak, aby sa rovnováha udržala čo najdlhšie.

Úloha 9: Vráťme sa k našej úvahe na strane 30. Predstavte si, že by ste mali spraviť triedu `Ryba` a triedy `Sardinka` a `Zralok` by boli jej potomkami. Ktoré atribúty a ktoré metódy by ste dali do Ryby? Ktoré z nich by ste prepísali v sardinke a ktoré v žralokovi?

Úloha 10: Vyskúšajte si fintu, ktorú sme použili na strane 30, keď sme vymýšľali, čo bude objekt a čo nie, na inom príklade. Predstavte si, že idete programovať softvér na predaj leteniek. Popíšte slovne, čo to má robiť, podčiarknite podstatné mená a zamyslite sa, ako by vyzerali ako triedy v Pythone. (Úloha pre pokročilých: naprogramujte to a predajte nejakej leteckej spoločnosti.)

Ešte drobná poznámka pre tých, ktorí by chceli vedieť, nakoľko náš model ekosystému zodpovedá skutočnosti. Na obrázku 1 môžete vidieť graf, ktorý zachytáva údaje kanadskej kožiarskej spoločnosti Hudson Bay Company o počte králičích a rysích koží, ktoré spoločnosť nakúpila od lovcov.[16] Záznamy pokrývajú takmer storočie. Ak vám ten graf v niečom pripomína ten váš graf, hovorí to niečo o tom, že ten model je spravený dobre a aspoň do istej miery opisuje realitu.



Obrázok 1: Rysy a králiky

16 Citované z Eugene Pleasants Odum: *Fundamentals of Ecology*, Saunders, 1953

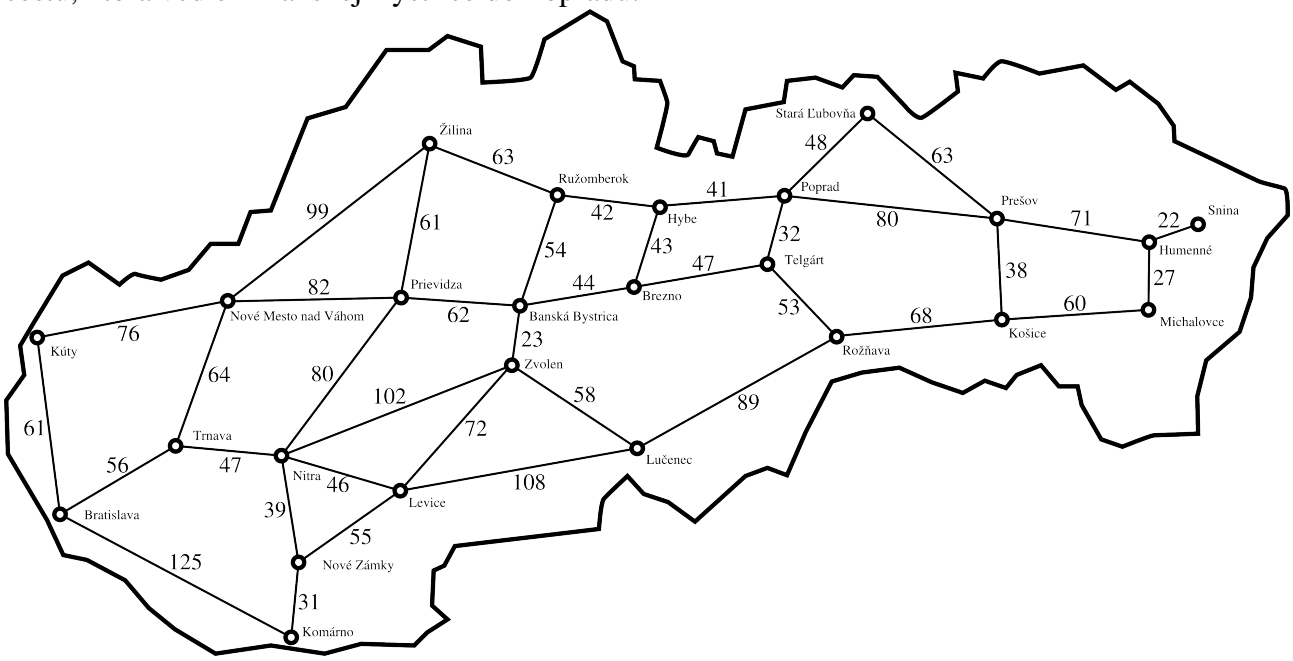
8. lekcia

Dijkstrov algoritmus alebo „kade je to najkratšie“

Podobne, ako majú chemici svoje zlúčeniny, s ktorými vedia pracovať, ako majú strojári materiály, ktorých vlastnosti poznajú, ako majú matematici svoje vety a definície, majú aj informatici svoj základný pracovný nástroj – algoritmy a dátové štruktúry. Algoritmus je finta, ako niečo zistiť či vypočítať.[17] Dátová štruktúra je finta, ako niečo uložiť v pamäti počítača tak, aby sa s tým potom dobre pracovalo.

Niektoré algoritmy sú naozaj slávne a nazývajú sa podľa svojich tvorcov – možno ste už počuli napríklad o Euklidovom algoritme. Táto lekcia bude pojednávať o algoritme, ktorý vymyslel Edsger Wybe Dijkstra. Dijkstra bol holandský počítačový vedec, ktorý sa nezmazateľne vpísal do histórie informatiky – mimo iného tým, že sa spolupodieľal na vzniku jazyka ALGOL 60[18], z ktorého boli neskôr odvodené jazyky C a Pascal a niektoré jeho črty podedil aj Python. Okrem toho je Dijkstra ten človek, ktorý programátorov uhovoril, že používať v programovaní skoky je škodlivé, takže vy už ani poriadne neviete, čo to skoky sú a dozviete sa to iba ak budete chcieť programovať v assembleri.

Dijkstrov algoritmus rieši nasledujúci problém: Predstavte si, že máte mapu Slovenska a na nej vyznačené cestné vzdialenosti tak, ako to môžete vidieť na obrázku 2. Chcete zistiť najkratšiu cestu, ktorá vedie z Banskej Bystrice do Popradu.



Obrázok 2: Cestné vzdialenosti na Slovensku

Keď sa človek na tú mapu pozrie, tak vidí, že do úvahy pripadajú tri možnosti – cez Ružomberok, cez Čertovicu (trasa cez Brezno a Hybe) alebo cez Telgárt. Stačí spočítať jednotlivé dĺžky a vybrať najkratšiu cestu. Keby ste ale mali hľadať najkratšiu cestu z Bratislavy do Sniny, tých

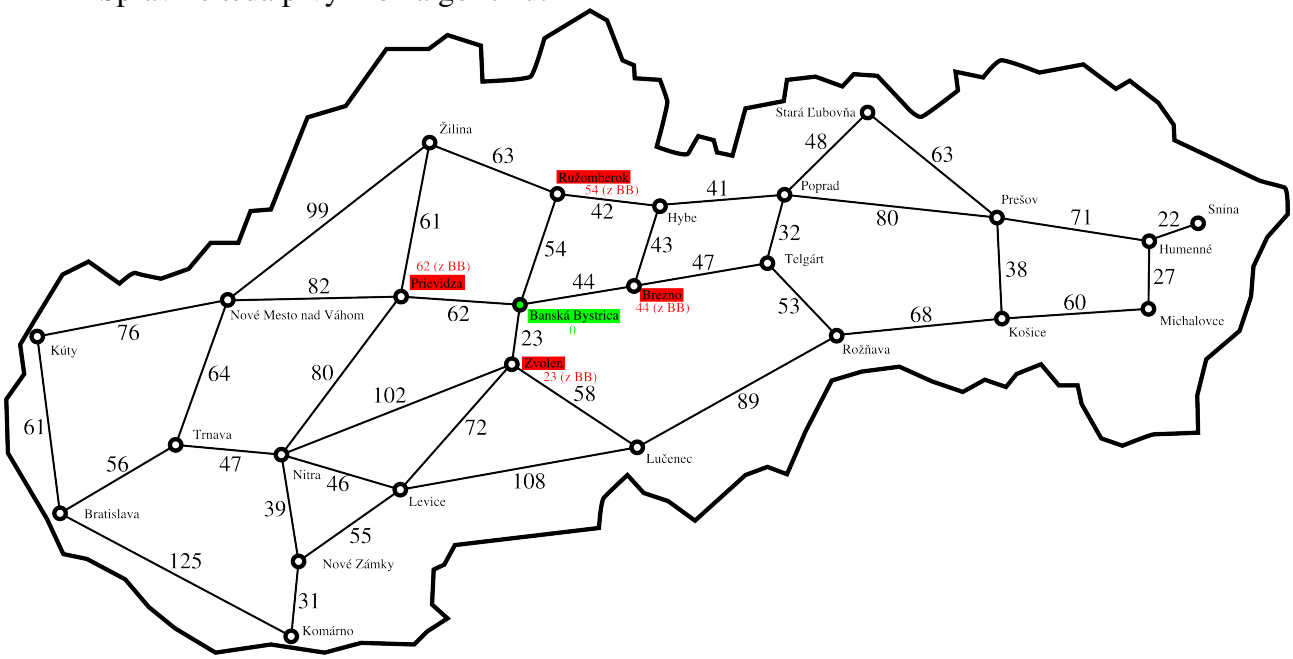
17 Slovo algoritmus pochádza z mena perzského matematika al-Chwarizmiho, ktorý ako prvý vedel úplne riešiť kvadratické rovnice.

18 Dijkstra s kolegom Zonneveldom sľúbili, že sa neoholia, kým kompilátor ALGOLu nedokončia. Sľub dodržali.

možností zrazu bude viacej a je reálne nebezpečenstvo, že by ste na niečo zabudli. Preto je dobré zveriť takéto vyhľadávanie počítaču. A na to, ako presne to má počítač rátať, aby mu to dlho netrvalo a aby zaručene tú najkratšiu cestu našiel, práve prišiel pán Dijkstra.

Ako to funguje, predvedieme najskôr na mape. Ako to naprogramovať, budeme rozmýšľať až potom. Predpokladajme, že hľadáme najkratšiu cestu z Banskej Bystrice do Popradu. Keďže máme počítač, nemusíme byť trochári – nájdeme najkratšiu cestu z Banskej Bystrice do každého mesta a potom sa pozrieme, koľko to vyšlo pre Poprad.

Spravme teda prvý krok algoritmu:



Obrázok 3: Cesty z Banskej Bystrice

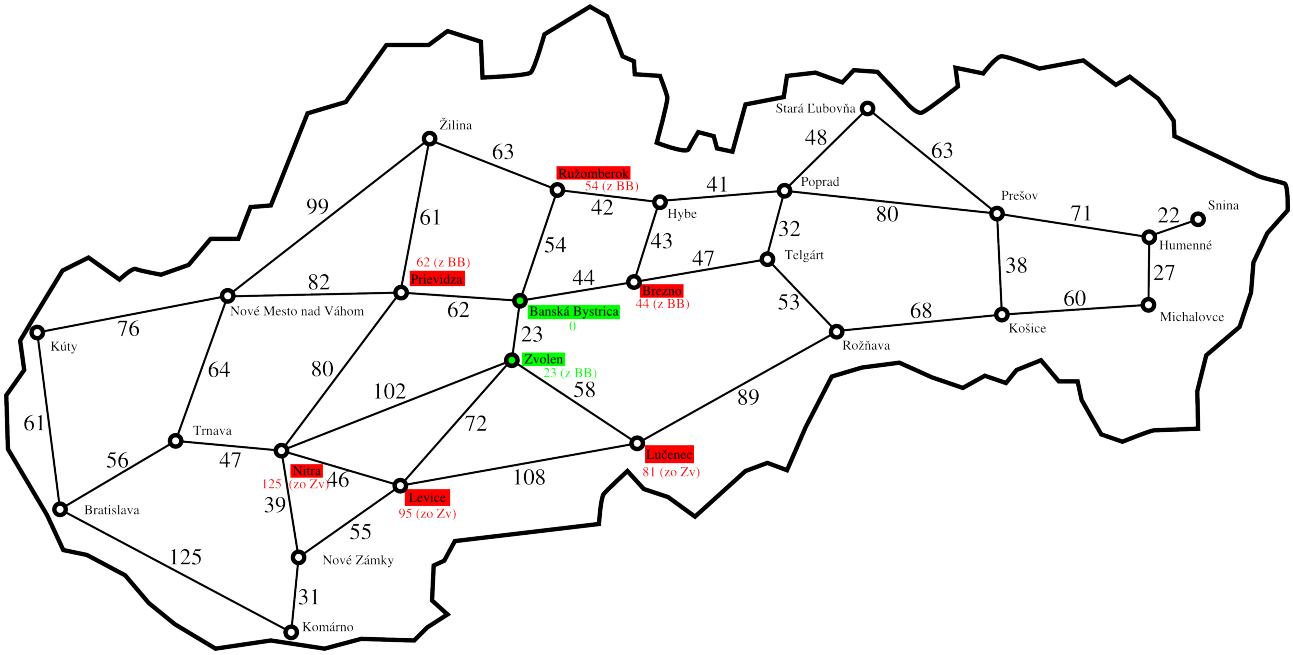
Na začiatku je zrejmé iba jedno: Že do Banskej Bystrice sa z Banskej Bystrice dostaneme najlepšie tak, že nebudeme nikam chodiť. Preto si Banskú Bystricu označíme zelenou a napíšeme si k nej nulu – je to minimálna vzdialenosť, ktorú treba prejsť, aby sme sa tam dostali.

Druhá vec, ktorú v tomto prvom kroku spravíme je, že sa pozrieme na mestá, do ktorých sa vieme z Bystrice dostať a ak to vieme spraviť lepšie, než doteraz[19], zapíšeme si, odkiaľ sme tam prišli a koľko sme museli prejsť. Mestá máme zatiaľ vyznačené červenou. Nie je totiž zatiaľ isté, že nájdené vzdialenosti sú najkratšie. Napríklad do Prievidze môže existovať skratka cez Zvolen, ktorú sme zatiaľ nemali šancu objaviť.

Druhý krok algoritmu začne v momente, keď si uvedomíme, že spomedzi červených miest môžeme jedno prefarbiť na zeleno. Je to konkrétne Zvolen. Spomedzi doteraz navštívených miest má najkratšiu dosiahnutú vzdialenosť. To znamená, že keby sme hľadali obchádzku cez ktorékoľvek zo zvyšných červených miest, tak to lepšie ako na 23 kilometrov nemáme šancu stihnúť. Zvolen teda môžeme prefarbiť na zeleno.

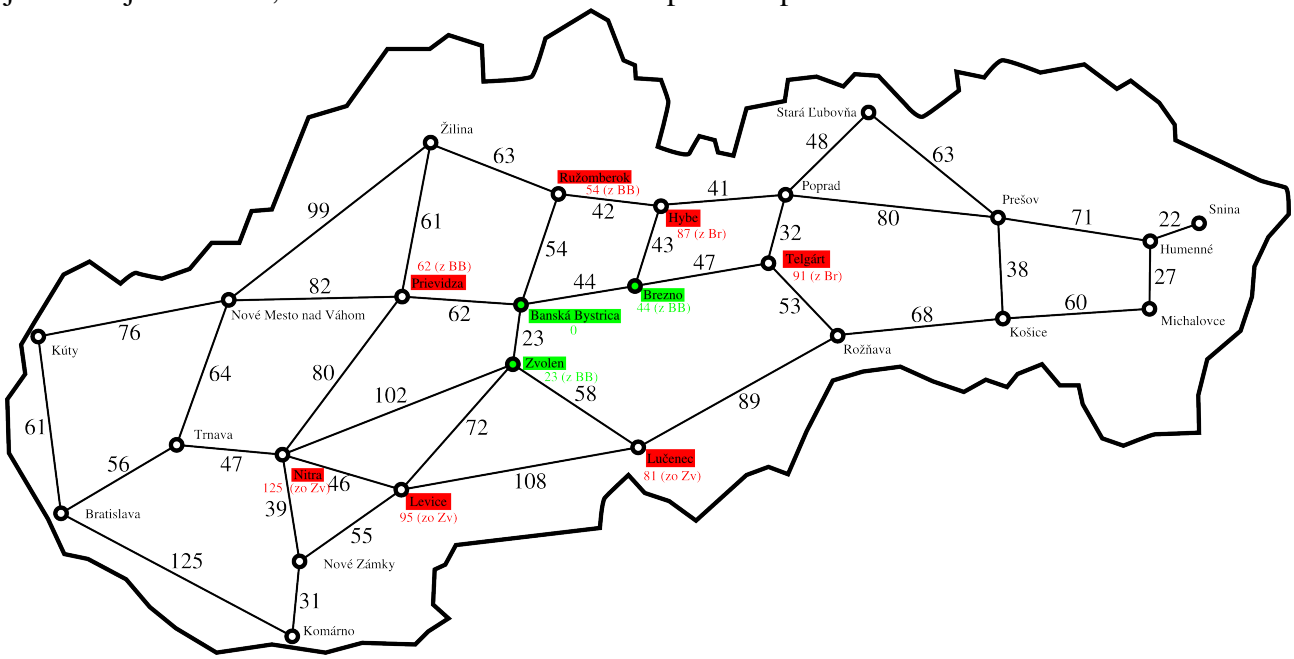
Teraz sa opäť pozrieme, kam sa dá dostať zo Zvolena. Nesmieme pri tom ale zabudnúť, že cestujeme z Bystrice, takže keď budeme rátať, ako ďaleko je to do Lučenca, musíme k vzdialenosti Zvolen – Lučenec pripočítať najrýchlejšiu cestu, ktorou sa vieme z Bystrice dostať do Zvolena (to už vieme, že je 23). Opäť si zapamätáme, odkiaľ sme sa do Lučenca, Levíc a Nitry dostali. Výsledok môžete vidieť na obrázku 4.

19 Doteraz sme sa do žiadneho z okolitých miest nedostali, takže zapíšeme rovno vzdialenosti od Bystrice. V ďalšej fáze algoritmu to ale bude komplikovanejšie.



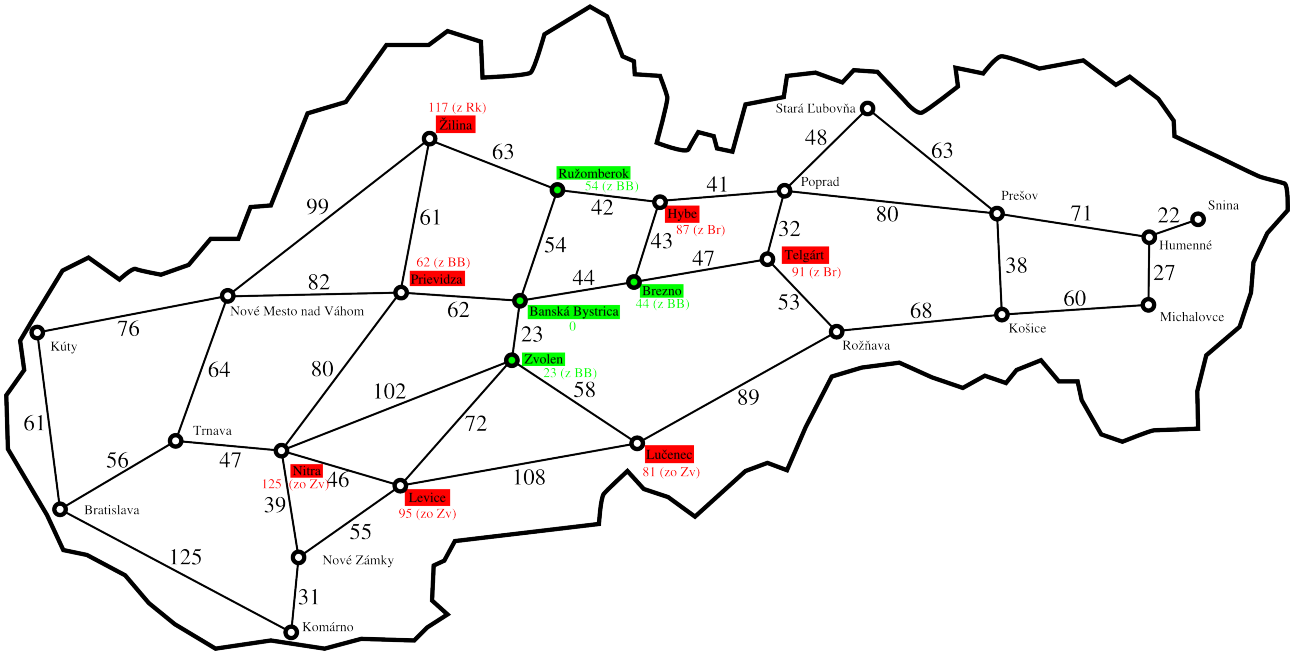
Obrázok 4: Pridané cesty cez Zvolen

A môžeme ísť na tretí krok. Opäť prezrieme všetky mestá, ktoré sú zafarbené na červeno a vyberieme z nich to, ktoré má pri sebe najmenšiu vzdialenosť. V tomto prípade to bude Brezno. Vzdialenosť sa zaručene nebude dať zlepšiť. Kratšia cesta do Brezna by totiž musela viesť buď zo zeleného mesta a to by sme pri Brezne to menšie číslo získali, už keď sme zafarbovali dotyčné mesto na zeleno, alebo cez nejaké červené mesto a to by bola okľuka, lebo cez všetky červené mestá je to ďalej. Pozrieme, kam sa dá dostať z Brezna a upravíme patričné vzdialenosti:



Obrázok 5: Brezno a susedia

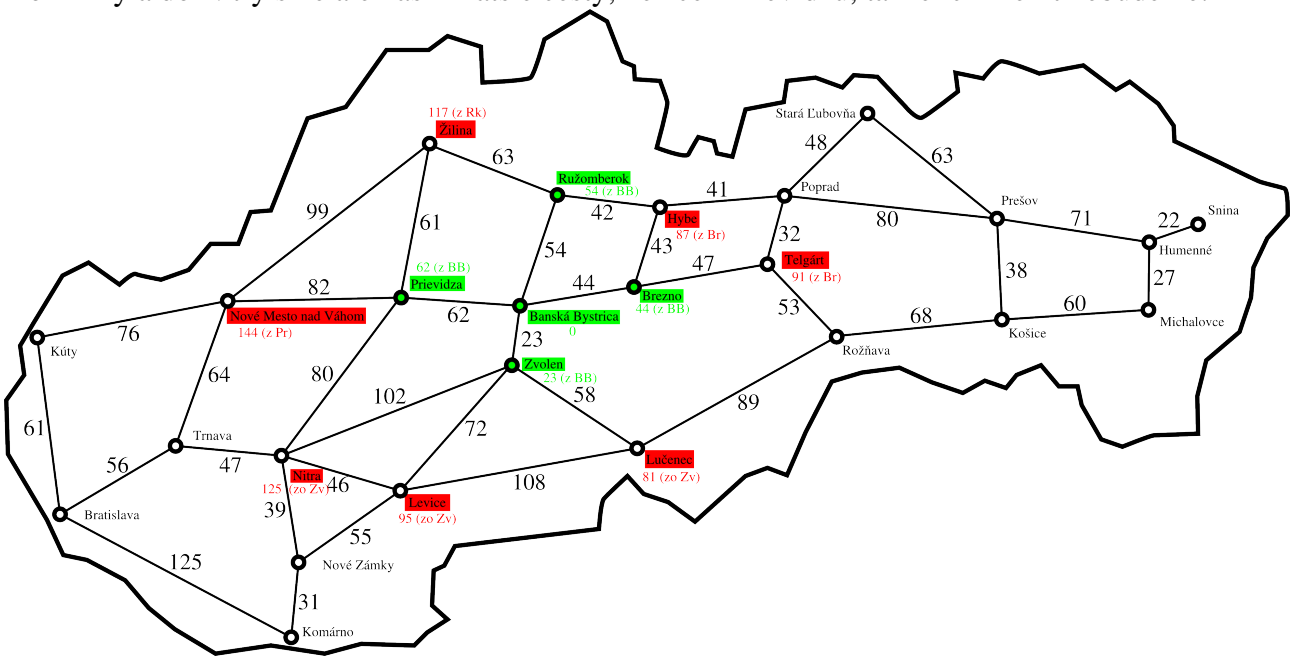
Úloha 1: Ktoré mesto zmeníme na zelené v ďalšom kroku?



Obrázok 6: Ružomberok a okolie

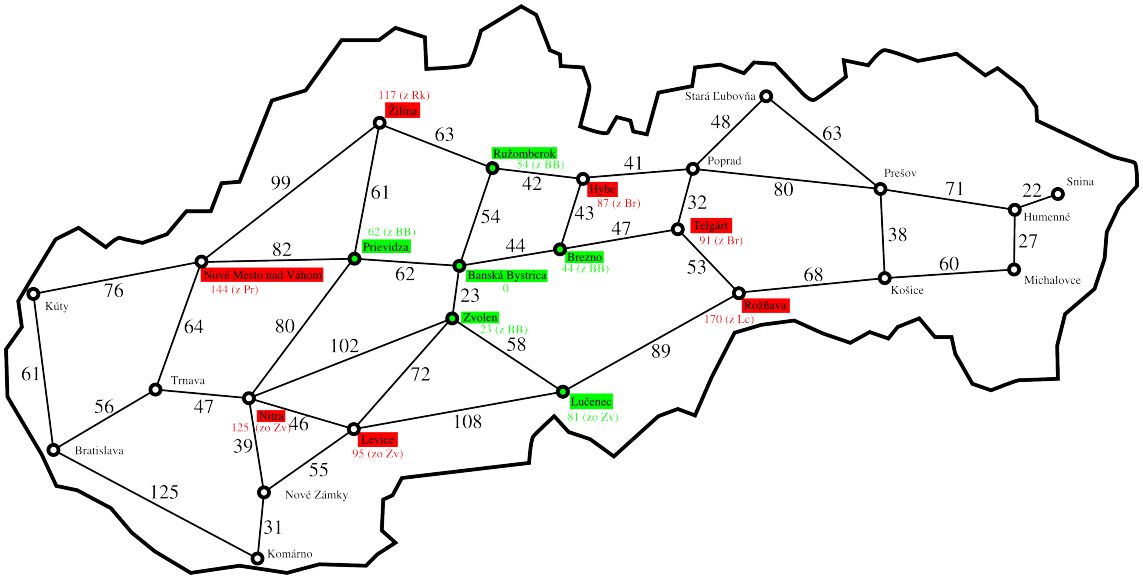
Áno, riešenie úlohy 1 je mesto Ružomberok. Všimnite si, že má dvoch ešte nevybavených susedov: Žilinu a Hybe. Keď sme počítali cestu do Hýb cez Ružomberok, vyšlo nám 54 + 42 = 96. Keďže sme sa už predtým dostali do Hýb kratšou cestou, pri Hybiach nič nemeníme. Keby sa ukázalo, že cesta cez Ružomberok je kratšia, k Hybiam by sme priradili menšie číslo a zapamätali by sme si, že sme sa tam dostali z Ružomberka.

A pokračujeme ďalej. Červené mesto s najmenšou vzdialenosťou od Bystrice je Prievidza. Zmeníme ju na zelenú. Vieme sa z nej dostať do Žiliny, do Nového Mesta nad Váhom a do Nitry. Do Žiliny a do Nitry sme ale našli kratšie cesty, než cez Prievidzu, takže ich meniť nebudeme.

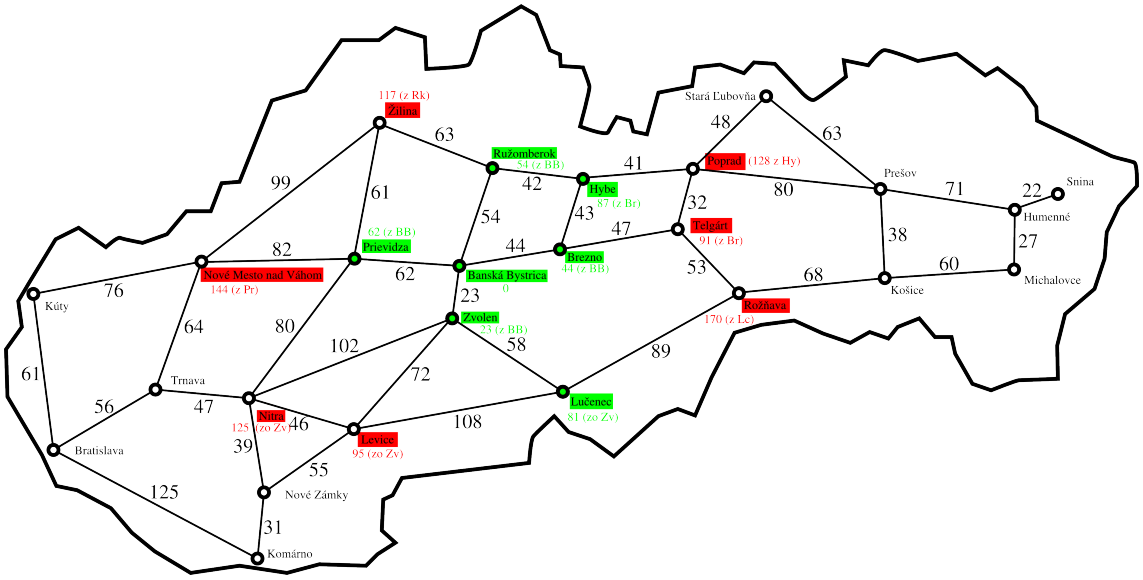


Obrázok 7: Prievidza

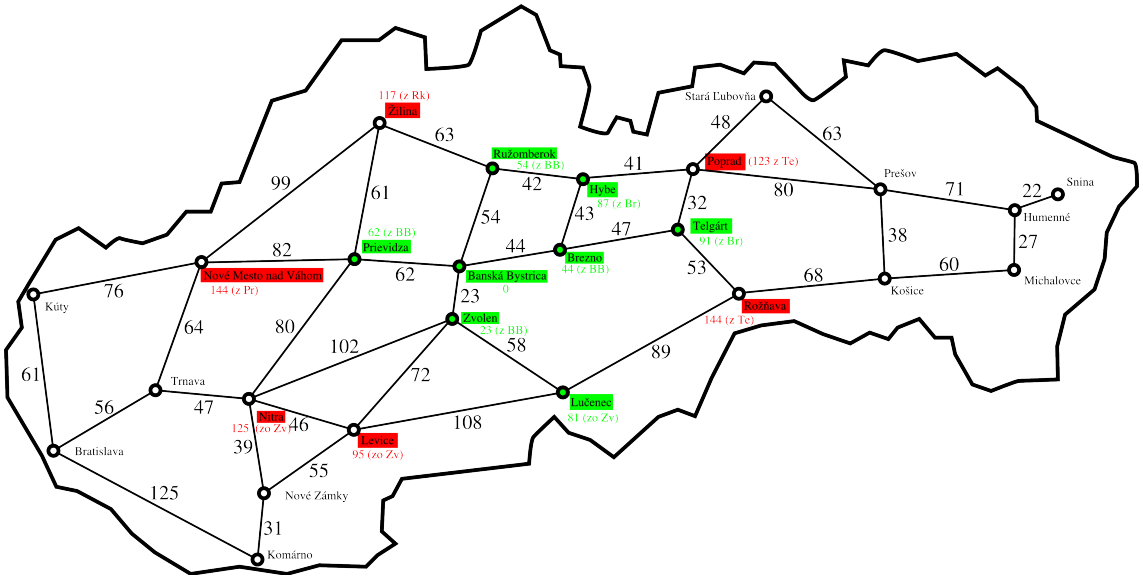
Teraz už by malo byť zrejmé, ako budeme pokračovať. Na ďalšej strane sú spravené ďalšie tri kroky algoritmu. Skúste si ich najprv spraviť sami a potom porovnajte, či vám to vyšlo rovnako. Skontrolujte, či ste vždy vybrali správne zelené mesto aj či ste správne upravili jeho susedov.



Obrázok 8: Lučenec



Obrázok 9: Hybe



Obrázok 10: Telgárt

Všimnite si, že keď sme v poslednom kroku kontrolovali susedov Telgártu, tak sme aj Popradu aj Rožňave menili zatiaľ nájdenú najkratšiu cestu, pretože cez Telgárt to bolo bližšie.

Úloha 2: Vytlačte si mapu, zoberte červené a zelené pero a kalkulačku a dopočítajte najkratšie cesty z Banskej Bystrice do ostatných miest Slovenska.

Úloha 3: Skúste porozmýšľať, ako by sa to dalo naprogramovať.

Skôr, než začneme programovať, povedzme si najprv niečo o tom, ako preniesť mapu Slovenska, s ktorou sme doteraz pracovali, do počítača. Mapa – tak ako sme ju nakreslili – je z matematického hľadiska graf. Tým nie je myslený graf nejakej funkcie, ale štruktúra, ktorá pozostáva z vrcholov (v našom prípade miest) a hrán (v našom prípade ciest). Každá hrana súvisí s dvoma vrcholmi.[20] Keďže v našom grafe je pri každej hrane nejaké číslo (ktoré označuje dĺžku cesty), je to ohodnotený graf.

Grafy sa v pamäti uchovávajú dvoma spôsobmi. Ktorý je výhodnejší, závisí od toho, či má graf veľa alebo málo hrán. Ak je hrán veľa, je dobré si zriadiť dvojrozmerné pole, v ktorom si pri každej dvojici vrcholov budete pamätať, či tam je alebo nie je hrana a akú má hodnotu. V Pythone sa to dá riešiť napríklad slovníkom, ktorého každý prvok bude opäť slovník. V nasledujúcej ukážke predvedieme, ako by taká matica vyzerala pre tri mestá: Bratislavu, Trnavu a Nitru.

```
matica = {}
matica['Bratislava'] = {}
matica['Bratislava']['Bratislava'] = -1
matica['Bratislava']['Trnava'] = 56
matica['Bratislava']['Nitra'] = -1
matica['Trnava'] = {}
matica['Trnava']['Bratislava'] = 56
matica['Trnava']['Trnava'] = -1
matica['Trnava']['Nitra'] = 47
matica['Nitra'] = {}
matica['Nitra']['Bratislava'] = -1
matica['Nitra']['Trnava'] = 47
matica['Nitra']['Nitra'] = -1
```

Ak chceme teraz zistiť, aká je dĺžka cesty z Nitry do Trnavy, stačí sa pozrieť na hodnotu `matica['Nitra']['Trnava']`. Ak je hodnota nezáporné číslo, je to vzdialenosť. Ak je hodnota `-1`, znamená to, že medzi danými dvoma mestami cesta nevedie.

Úloha 4: Vyskúšajte zadať pythonu uvedené príkazy a potom si nechajte vypísať celý objekt `matica`. Potom sa na ten výpis chvíľu pozerajte a pokúste sa v ňom zorientovať.

Druhý spôsob, ako si zapamätať graf, je vhodnejšie použiť, keď je hrán menej a prevažnú väčšinu matice by tvorili mínus jednotky. Vtedy si pri každom vrchole budeme pamätať iba zoznam vrcholov, do ktorých z neho vedie hrana a dĺžky týchto hrán. Takúto reprezentáciu grafu si môžeme v Pythone urobiť nasledujúcim spôsobom:

```
graf = {}
graf['Bratislava'] = [['Trnava',56]]
graf['Trnava'] = [['Bratislava',56],['Nitra',47]]
graf['Nitra'] = [['Trnava',47]]
```

20 http://sk.wikipedia.org/wiki/Graf_%28matematika%29

Úloha 5: Vyskúšajte zadať pythonu uvedené príkazy a potom si nechajte vypísať celý objekt `graf`. Potom sa na ten výpis chvíľu pozerajte a pokúste sa v ňom zorientovať.

V našom prípade mapy Slovenska je hrán relatívne málo, preto použijeme tento druhý prístup. Problém ale je, že celú mapu Slovenska máme uloženú vo formáte CSV.[21] Každá cesta je tam určená svojimi krajnými mestami a dĺžkou. Začiatok súboru vyzerá takto:

```
Bratislava,Trnava,56
Bratislava,Kúty,61
Bratislava,Komárno,125
Kúty,Nové Mesto nad Váhom,76
Trnava,Nové Mesto nad Váhom,64
Trnava,Nitra,47
```

Výhodou toho, že dáta skladujeme v súbore a nie priamo v programe je, že keď budeme chcieť mapu vylepšiť, stačí do súboru pridať nejaké riadky. Nevýhodou je, že bude treba napísať funkciu, ktorá ten súbor prečíta a urobí nám dátovú štruktúru. Tá nevýhoda je ale relatívne malá, pretože spraviť si takú funkciu nie je problém:

```
 1   def citajSubor(meno):
 2       subor = open(meno,"r")
 3       riadky = subor.readlines()
 4       cesty = {}
 5       for i in riadky:
 6           a = i.split(",")
 7           for j in range(2):
 8               if not a[j] in cesty:
 9                   cesty[a[j]] = []
10           cesty[a[0]].append([a[1],float(a[2])])
11           cesty[a[1]].append([a[0],float(a[2])])
12       subor.close()
13       return cesty
```

Funkcia má jediný parameter – meno súboru, ktorý chceme čítať. Na riadku 2 súbor otvoríme. Znamená to, že pekne poprosíme operačný systém, aby nás k súboru pustil. Ten druhý parameter `"r"` hovorí, že chceme zo súboru čítať. Keby ste tam namiesto `r` dali `w`, znamenalo by to, že chcete do súboru zapisovať a operačný systém by súbor, do ktorého idete zapisovať, najprv zmazal, takže sa nepomýľte a `w` tam nedávajte.

Na riadku 3 voláme funkciu `readlines`, ktorá celý súbor prečíta a ako výsledok vráti zoznam jeho riadkov. Teda vráti niečo, čo bude začínať takto:

```
['Bratislava,Trnava,56\n', 'Bratislava,Kúty,61\n', 'Bratislava,Komárno,
125\n', 'Kúty,Nové Mesto nad Váhom,76\n', ...
```

Na riadku 4 si vyrobíme slovník `cesty`, do ktorého uložíme náš graf. Zatiaľ je prázdny.

V cykle na riadkoch 5 až 11 teraz jednotlivé riadky spracujeme. Riadok uložený do premennej `i` najprv rozdelíme na jednotlivé časti pomocou funkcie `split`. Táto funkcia rozdelí reťazec na časti, ktoré sú oddelené čiarkou. Tú čiarku sme dali funkcii ako parameter. (Ak by sme tam nedali nič, funkcia by reťazec rozdelila podľa tzv. blank znakov – medzery, tabulátoru a prechodu na nový riadok.) Pri spracovaní prvého riadku teda v premennej `a` budeme mať hodnotu

```
['Bratislava', 'Trnava', '56\n']
```

21 Comma-separated values alebo po našom „hodnoty oddelené čiarkou“. Takéto súbory majú tú výhodu, že s nimi vedia pracovať aj tabuľkové kalkulátory a iný bežný softvér.

Na riadkoch 7 až 9 sa pozrieme, či naše dve mestá už máme v slovníku. Ak nejaké nemáme, tak ho tam pridáme a ako hodnotu mu zatiaľ vložíme prázdny zoznam. Na riadku 10 pridáme prvému mestu do zoznamu druhé (ako zoznam aj so vzdialenosťou). Funkcia `float` nám zmení reťazec `'56\n'` na číslo `56.0`. Na riadku 11 pridáme druhému mestu do zoznamu prvé s tou istou vzdialenosťou, pretože naspäť je to rovnako ďaleko.[22]

Na riadku 12 súbor zatvoríme – povieme operačnému systému, že už ho pre nás nemusí držať otvorený. Na riadku 13 vrátime to, čo sme načítali, ako výsledok funkcie.

Úloha 6: Vyrobte si súbor `cesty.csv` a pokúste sa načítať ho s pomocou našej funkcie. Nechajte si vypísať výsledok, potom sa na ten výpis chvíľu pozerajte a pokúste sa v ňom zorientovať.

Úloha 7: (Pre machrov.) V tomto momente prestaňte čítať a pokúste sa urobiť to sami. Keď to urobíte, prečítajte si aj našu verziu.

Ak už to máte hotové, alebo ak (nebodaj!) nie ste machri, poďte sa pozrieť, ako to naprogramoval autor učebnice.

Ako ste si všimli, pri každom meste, s ktorým sme pracovali, sme si uchovávali tri údaje: mesto, z ktorého sme prišli, minimálnu vzdialenosť, ktorú sa nám zatiaľ podarilo dosiahnuť a farbu – teda informáciu o tom, či už je dosiahnutá vzdialenosť minimálna možná. Aby sa nám s týmito údajmi dobre pracovalo, vytvoríme si triedu `Smerovka`, ktorá bude tieto tri údaje obsahovať.

```
14  class Smerovka:
15      def __init__(this, vzdialenost=float("infinity"),
16                   odkialsmeprisli="odnikial", hotovo=False):
17          this.vzdialenost = vzdialenost
18          this.odkialsmeprisli = odkialsmeprisli
19          this.hotovo = hotovo
```

Všimnite si štandardné hodnoty, ktoré bude mať smerovka nastavené, ak neurčíme inak. Vzdialenosť bude nastavená na `float("infinity")`, po našom nekonečno. Áno, Python vie počítať aj s nekonečnom. Je to číslo, ktoré je väčšie, ako každé iné reálne číslo. Ak vás na matematike učili, že nekonečno nie je číslo, nenechajte sa tým zviklať. Keďže na začiatku nepoznáme cestu do žiadneho z miest, hodnota „nekonečno“ je ako vzdialenosť presne to, čo potrebujeme. Hodnotu `odkialsmeprisli` zatiaľ nastavíme na `odnikial`. Atribút `hotovo` bude označovať, či už je mesto zelené. Na začiatku bude mať hodnotu `False`.

Môžeme začať programovať Dijkstrov algoritmus. Vytvoríme pre neho samostatnú funkciu `najkratsiaCesta`. Tá dostane na vstupe dva parametre. Prvý bude cestná sieť, ako sme ju načítali zo súboru. Druhý bude mesto, z ktorého budeme chcieť cestovať. Mesto, do ktorého budeme cestovať, určovať nemusíme, lebo Dijkstrov algoritmus nám aj tak nájde najkratšiu cestu do všetkých.

Ako výsledok vráti funkcia slovník. Každému mestu bude priradená smerovka, ktorá bude hovoriť, kadiaľ je to do neho zo zadaného mesta najbližšie a ako je to v prípade najkratšej cesty ďaleko.

Naša funkcia môže vyzerať napríklad takto:

22 Toto v prípade jednosmerných ciest nemusí byť vždy pravda. Dijkstrovmu algoritmu by to ale nevadilo. Fungoval by aj tak.

```
20  def najkratsiaCesta(cesty,odkial):
21      kade = {}
22      for i in cesty:
23          kade[i] = Smerovka()
24      kade[odkial] = Smerovka(0, '', True)
25      posledne = odkial
26      minimum = 0
27      while minimum < float("infinity"):
28          for i in cesty[posledne]:
29              if kade[i[0]].vzdialenost > kade[posledne].vzdialenost + i[1]:
30                  kade[i[0]].vzdialenost = kade[posledne].vzdialenost + i[1]
31                  kade[i[0]].odkialsmeprisli = posledne
32          minimum = float("infinity")
33          nova = ''
34          for i in kade:
35              if (not kade[i].hotovo) and kade[i].vzdialenost < minimum:
36                  nova = i
37                  minimum = kade[i].vzdialenost
38          posledne = nova
39          if posledne != '':
40              kade[posledne].hotovo = True
41      return kade
```

Slovník, v ktorom budeme ukladať medzivýsledky aj finálny výsledok Dijkstrovho algoritmu sa bude nazývať `kade`. Vytvoríme ho na riadku 21. Na riadkoch 22 a 23 využijeme to, že keď spravíme `for` cyklus, ktorý bude prechádzať cez slovník `cesty`, tak sa do riadiacej premennej cyklu budú postupne vkladať všetky kľúče, v našom prípade mestá. Pre každé mesto teda vytvoríme v slovníku `kade` novú smerovku. Každá smerovka bude mať zatiaľ nekonečnú vzdialenosť. Na riadku 24 nastavíme mestu, z ktorého hľadáme najkratšie cesty smerovky, ktorá hovorí, že najkratšia cesta do neho meria 0 kilometrov a že mesto už je hotové.

V premennej `posledne` budeme skladovať posledné mesto, ktoré sme zatiaľ prefarbili na zeleno. Zatiaľ do neho vložíme mesto, z ktorého začíname. V premennej `minimum` budeme skladovať vzdialenosť posledne uzavretého mesta od východzieho. Zatiaľ to bude 0.

Hlavný cyklus funkcie sa odohrá na riadkoch 27 až 40. Cyklus bude bežať dovtedy, kým bude v premennej `minimum` hodnota menšia ako nekonečno. Na riadkoch 28 až 31 prezrieme všetky cesty, ktoré vedú z posledne uzavretého mesta a skontrolujeme, či do susedných miest nevedie cez posledne uzavreté lepšia cesta. Ak sme lepšiu našli, tak ju na riadkoch 30 a 31 mestu nastavíme. (Ak do mesta doteraz žiadna cesta neviedla, má nastavenú vzdialenosť nekonečno a lepšie bude čokoľvek, než to.) V cykle na riadkoch 34 až 37 nájdeme spomedzi doteraz neuzavretých miest to, ktoré je východziemu najbližšie. To vyhlásime na riadku 40 za hotové. Dobre si pozrite podmienku na riadku 35.

Ak už žiadne neuzavreté mesto, do ktorého sa vieme dostať, neostalo, hodnota premennej `minimum`, ktorú sme nastavili na riadku 32 ostane nekonečno a hlavný cyklus sa skončí. Zoznam `kade` vráti funkcia ako svoj výsledok.

Najkratšiu cestu do každého mesta máme zistenú. Teraz už len ostáva tú cestu vypísať. Na to si spravíme samostatnú funkciu. Tá bude mať dva vstupné parametre: slovník s najkratšími cestami a mesto, do ktorého chceme prísť. Môže vyzerať napríklad takto:

```
42  def vypisTrasu(najkratsie,kam):
43      if najkratsie[kam].vzdialenost == float("infinity"):
44          print("Nedá sa tam dostať. :(")
45          return
46      cesta = [kam]
47      kdesme = kam
48      while najkratsie[kdesme].vzdialenost != 0:
49          kdesme = najkratsie[kdesme].odkialsmeprisli
50          cesta.append(kdesme)
51      print("Najkratšia cesta vedie cez mestá:")
52      for i in reversed(cesta):
53          print(i)
54      print("Vzdialenosť je %d kilometrov." % najkratsie[kam].vzdialenost)
```

Na riadku 43 skontrolujeme, či sa do cieľového mesta vôbec dá dostať. Keď je totiž mapa rozdelená na viacero samostatných častí, nemusí sa to vôbec dať. Rovnako by mohli nastať problémy, keby sme pracovali s jednosmernými cestami. Ak sa to teda nedá, podáme o tom správu a výpis ukončíme.

Ak sa do cieľového mesta dostať dá, spravíme si zoznam `cesta`. Do neho si uložíme cieľové mesto. Rovnako si ho uložíme aj do premennej `kdesme`. V cykle na riadkoch 48 až 50 sa vždy z mesta, kde práve sme presunieme do mesta, odkiaľ sme do neho prišli a toto mesto pridáme do cesty. To robíme dovtedy, kým sa nedostaneme do mesta, ktoré má od východzieho vzdialenosť 0, teda do východzieho. Keďže máme mestá v zozname `cesta` uložené v opačnom poradí – od cieľového k východziemu – musíme pri výpise na riadkoch 52 a 53 zoznam otočiť.

A teraz už len ostáva naše skvelé funkcie použiť, napríklad takto:

```
55  siet = citajSubor("cesty.csv")
56  oznam = "V databáze máme tieto mestá:\n"
57  for i in siet:
58      oznam += i + "   "
59  print(oznam)
60  odkial = input("Z ktorého mesta si prajete cestovať? ")
61  kam = input("Kam si prajete ísť? ")
62  najkratsia = najkratsiaCesta(siet,odkial)
63  vypisTrasu(najkratsia,kam)
```

Dijkstrov algoritmus sa používa na mnohých miestach. Stačí napríklad zameniť kilometre za potrebný čas a namiesto najkratšej cesty bude počítať najrýchlejšiu. Podobné algoritmy (aj keď trochu rafinovanejšie) sa používajú na vyhľadávanie najrýchlejšieho spojenia v cestovných poriadkoch.

Úloha 8: Pochopte a ak ste si to nespravili po svojom, tak vyskúšajte.

9. lekcia
Programujeme hru alebo „mimozemšťania útočia“

Táto kapitola bude o tom, ako naprogramovať v Pythone hru. O programovaní hier existuje samozrejme rozsiahla literatúra.[23] V tejto kapitole sa ale dozviete všetky základné finty, ktoré potrebujete na to, aby ste boli schopní vytvoriť v Pythone hru vo vlastnej réžii.

V prvom rade budete potrebovať nejakú knižnicu, ktorá vie pracovať s obrázkami a zvukmi, vie spracovať vstup od myši a z klávesnice. Takýchto knižníc určených na tvorbu hier existuje viacero, väčšina ale zatiaľ funguje len pre Python 2. Existuje ale svetlá výnimka – knižnica `pyglet`. Tá behá hladko aj pod Pythonom 3. Musíte si ju ale najprv stiahnuť a nainštalovať. V čase písania tejto kapitoly ale táto procedúra pozostáva z dvoch krokov[24]:

Najprv si treba stiahnuť a nainštalovať softvér zvaný `mercurial`. Je to softvér, ktorý je veľmi užitočný, keď viacerí programátori pracujú na tom istom projekte. S jeho pomocou sú schopní posielať zmeny, ktoré práve urobili alebo si stiahnuť najnovšiu verziu projektu. Tento softvér používajú aj programátori `pyglet`. Stiahnite si ho z adresy http://mercurial.selenic.com/wiki/Download a nainštalujte.

Keď bude `mercurial` nainštalovaný, spustite si príkazový riadok a do neho zadajte príkaz

```
hg clone https://pyglet.googlecode.com/hg/ pyglet
```

Ten spôsobí, že sa celá knižnica, ktorú chcete inštalovať, stiahne do adresára `pyglet`. Keď sa sťahovanie dokončí, zadajte príkaz

```
cd pyglet
```

ktorým sa prepnete do práve vytvoreného adresára a v ňom zadajte príkaz

```
python3 setup.py install
```

ktorý obstará kompletnú inštaláciu knižnice `pyglet` pre Python 3. Tu uvedený postup by mal fungovať pre každý z bežných operačných systémov (Linux, Os X, Windows). Inštalácia tiež chvíľku trvá a keď sa skončí, môžete začať knižnicu `pyglet` používať.

Keď máme úspešne knižnicu nainštalovanú, môžeme sa konečne venovať programovaniu. Predtým, než sa začneme zaoberať samotnou knižnicou, povieme ešte pár slov o niektorých črtách Pythonu, o ktorých zatiaľ nebola reč a ktoré budeme potrebovať.

Prvá zaujímavá vec je, že do premenných sa dajú priradiť aj funkcie. Jednoduchú ukážku môžete vidieť tu:

```
def vypis():
    print("Myslím, teda som. Aspoň myslím.")

hu = vypis
hu()
```

23 Prípadným záujemcom odporúčame knižku Core Techniques and Algorithms in Game Programming, ktorej autorom je Daniel Sanchez-Crespo. Je to naozaj skvelá kniha.

24 Problém je v tom, že pod Pythonom 3 momentálne funguje iba úplne najnovšia verzia, na ktorej vývojári ešte len pracujú. Táto verzia bude určite časom zverejnená na inštaláciu klasickým spôsobom. Ale momentálne ešte zverejnená nie je a tie verzie, ktoré zverejnené sú, s Pythonom 3 nefungujú.

Program vypíše výrok seržanta Navážku z Ank-Morporkskej nočnej hliadky napriek tomu, že nevoláme funkciu `vypis`, ale funkciu `hu`.

To, že sa funkcia dá vložiť do premennej, sa dá využiť napríklad na to, že sa funkcia dá dať ako parameter inej funkcii. To sa niekedy celkom môže hodiť. Pozrite si napríklad toto:

```
def kluc(a):
    return len(a)

mena = ["Janíčko","Bob","Alice","Eve","Arabela","Marienka"]
mena.sort()
print(mena)
mena.sort(key=kluc)
print(mena)
```

Zoznam mien `mena` utriedime pomocou funkcie sort. Funkcia ich zotriedi podľa abecedy. To ale nie je jediný spôsob, ako sa mená dajú utriediť. Ak chceme utriediť mená napríklad podľa dĺžky, spravíme si funkciu, ktorá ako svoj výsledok vráti dĺžku parametra. A túto funkciu môžeme dať ako parameter `key` sortu, aby vedel, že má triediť podľa nej a nie podľa abecedy. Funkciou `kluc` môžete úplne ovládať, podľa čoho sa bude zoznam triediť.

Úloha 1: Vyskúšajte si oba príklady. Prepíšte funkciu `kluc` tak, aby ste zoznam utriedili abecedne podľa druhého písmena slova.

Drobná poznámka – podľa dĺžky slova sa zoznam dal utriediť aj príkazom `mena.sort(key=len)` Ak existuje funkcia, ktorú potrebujeme, môžeme ju rovno použiť. Nie je treba vytvárať vlastnú.

Aby bol zmätok ešte väčší, existujú nielen funkcie, ktoré ako parameter dostanú nejakú funkciu, ale aj funkcie, ktoré funkciu vrátia ako svoj výsledok. Môže sa to spraviť napríklad takto:

```
import time

def obal(f):
    def nova():
        t = time.time()
        f()
        print("Funkcia",f.__name__,"sa už skončila.")
        print("Trvala %.5f sekúnd." % (time.time() - t))
    return nova

def vypis():
    print("Myslím, teda som. Aspoň myslím.")

vypis = obal(vypis)
vypis()
```

Funkcia `obal` dostane na vstupe funkciu a vráti inú funkciu, ktorá vykoná pôvodnú funkciu a zistí, ako dlho bežala. To sa môže hodiť, keď napríklad potrebujete zistiť, ktorá funkcia váš program zdržuje. Príkaz `vypis = obal(vypis)` potom zmení pôvodnú funkciu `vypis` na funkciu `vypis` so vstavanými stopkami.

Táto finta sa dá použiť aj v podobe tzv. dekorátoru. Predošlý program môžeme prepísať do takejto podoby:

```
import time

def obal(f):
    def nova():
        t = time.time()
        f()
        print("Funkcia",f.__name__,"sa už skončila.")
        print("Trvala %.5f sekúnd." % (time.time() - t))
    return nova

@obal
def vypis():
    print("Myslím, teda som. Aspoň myslím.")

vypis()
```

Funkcia `obal`, ktorou k iným funkciám vieme pridať stopky, je rovnaká, ako v predošlom prípade. Keď chceme pridať stopky, k funkcii `vypis`, stačí v riadku nad ňou uviesť `@obal`. Robí to to isté, ako keby sme po funkcii napísali `vypis = obal(vypis)`.

Úloha 2: Pochopte a vyskúšajte.

Dekorátory sa občas používajú v pythonovských knižniciach, keď chceme, aby používateľ mohol napísať iba podstatnú časť nejakej funkcie a nemusel do nej prepisovať spústu vecí, ktoré mu tam pridá knižnica samotná. Podobne to robí aj `pyglet`.

Pozrime sa na prvý jednoduchý program, v ktorom `pyglet` použijeme:

```
 1  import pyglet
 2
 3  window = pyglet.window.Window()
 4  label = pyglet.text.Label('Hello, world',
 5                    font_size=36,
 6                    x=window.width//2, y=window.height//2,
 7                    anchor_x='center')
 8
 9  @window.event
10  def on_draw():
11      window.clear()
12      label.draw()
13
14  pyglet.app.run()
```

Na riadku 3 vytvoríme okno, v ktorom sa bude všetko diať. Parametrami môžeme určiť, aké má byť veľké, kde má byť umiestnené alebo či chceme zapnúť celoobrazovkový režim. Na riadkoch 4 až 7 vytvoríme ceduľu (po anglicky label). Ceduľa je obrázok s textom. Zadáme text, ktorý chceme na ceduli mať, veľkosť písma (`font_size`) a súradnice v okne (`x` a `y` – nastavujeme ich tak, aby bol nápis v strede okna. Dve lomítka znamenajú celočíselné delenie. Pozor! Súradnice sú spravené tak, že bod `[0,0]` je vľavo dole. Ak ste navyknutí na súradnice z matematiky, tak vám to zvláštne nepríde, ale ak ste už programovali nejakú inú grafiku, mohli ste sa stretnúť aj s iným prístupom.) Ďalej nastavíme to, že v horizontálnom smere chceme nápis vycentrovať (`anchor_x`). Ceduli sa dajú určiť aj nejaké ďalšie parametre, napríklad použitý font.

A dostávame sa k funkcii s dekorátorom. Funkcia sa musí volať `on_draw`. Zavolá ju samotná knižnica, takže do nej treba napísať, ako sa má vykresliť scéna. Dekorátor

`@window.event` hovorí, že sa jedná o obsluhu udalosti. To, kedy sa prekresľovanie udeje, nemáme celkom pod kontrolou – udeje sa to vtedy, keď si knižnica zmyslí. Ale potrebujeme mať funkciu, ktorá to urobí, keď je treba.

Samotná funkcia okno premaže a vykreslí ceduľu.

Na riadku 14 sa spustí samotná aplikácia. Otvorí sa okno, program bude dávať pozor, čo používateľ robí, udalosti sa budú posielať všetkým obslužným funkciám, ktoré sme urobili, bude sa vykresľovať, čo sa má vykresľovať a prípadne sa budú robiť aj nejaké ďalšie veci. Ak stlačíte `ESC` alebo zavriete okno, program sa skončí.

Úloha 3: Vyskúšajte.

Ďalší ukážkový program v sebe zahŕňa takmer všetko, čo budeme potrebovať, keď budeme robiť našu hru:

```
 1  import pyglet
 2  from pyglet.window import key
 3  from pyglet.window import mouse
 4
 5  ufon = pyglet.resource.image('09-alien.png')
 6  cas = 0
 7
 8  window = pyglet.window.Window()
 9
10  @window.event
11  def on_draw():
12      window.clear()
13      ufon.blit((window.width - ufon.width)//2, 20)
14      label = pyglet.text.Label(str(cas), font_size=36,
15                    x=window.width//2, y=window.height//2,
16                    anchor_x='center')
17      label.draw()
18
19  @window.event
20  def on_key_press(symbol, modifiers):
21      if symbol == key.M:
22          print("Stlačil si M")
23      elif symbol == key.LEFT:
24          print("Stlačil si šípku vľavo.")
25
26  @window.event
27  def on_mouse_press(x, y, button, modifiers):
28      if button == mouse.LEFT:
29          print("Klikol si ľavým na súradniciach [%d,%d]" % (x,y))
30
31
32  def sekunda(dt):
33      global cas
34      cas += 1
35
36  pyglet.clock.schedule_interval(sekunda,1)
37
38  pyglet.app.run()
```

Na prvých troch riadkoch načítame knižnicu `pyglet` a aby sme nemuseli zakaždým vypisovať milióny odkazov, načítame zvlášť aj objekty `key` a `mouse`. Tie sa nám zídu, keď budeme potrebovať zistiť, aký kláves bol stlačený a kde sa udiali kliknutia myšou.

Do premennej `ufon` načítame obrázok zo súboru. Súbor by mal byť umiestnený v tom istom adresári, ako samotný program, inak ho Python nenájde. Premennú `cas` vynulujeme.

Na riadku 8 vyrobíme okno, v ktorom sa bude program odohrávať. Konštruktoru sa dajú zadať parametre, ktoré určujú veľkosť okna, nadpis okna alebo ikonu.

Poďme sa pozrieť na obsluhu udalostí. Ako prvú vytvoríme udalosť `on_draw` (riadky 10 až 17). Plochu vymažeme (riadok 12). Do stredu dole nakreslíme ufóna (riadok 13). Pozrite sa, ako sa počíta umiestnenie ufóna. Operátor `//` znamená delenie v celých číslach, takže napríklad `14 // 3` bude `4`. Vytvoríme ceduľu, do ktorej zapíšeme hodnotu premennej `cas` (riadky 14 až 16) a vykreslíme ju (riadok 17). Pred funkciu nezabudneme pridať dekorátor.

Ďalšia funkcia má na svedomí správu udalostí z klávesnice a nazýva sa `on_key_press`. Má dva parametre. Parameter `symbol` určuje, aký kláves bol stlačený, parameter `modifiers` zas určuje, aké modifikátory (napríklad `ctrl`, `alt` alebo `shift`) boli použité. Každý kláves má svoje meno, napríklad kláves M má meno `key.M`. Naša funkcia iba napíše niečo do terminálu, keď stlačíte kláves M, alebo šípku vľavo.

Funkcia, ktorá má na starosti reagovať na kliknutie myšou, sa nazýva `on_mouse_press`. Má štyri parametre. Parametre `x` a `y` určujú súradnice miesta, kde sa kliknutie udialo. Parameter `button` určuje, ktoré tlačidlo na myši bolo stlačené. Parameter `modifiers` má rovnaký význam, ako v predošlej funkcii. Naša funkcia vypíše súradnice miesta, na ktoré sa kliklo, ale iba v prípade, že sa kliklo ľavým tlačidlom myši.

Na riadkoch 32 až 36 sa deje zaujímavá vec. Riadok 36 nastavuje, že sa pravidelne raz za sekundu má zavolať funkcia `sekunda`. (Meno funkcie je prvý parameter, ako často sa má funkcia volať, je druhý parameter.) Funkcia `sekunda` nerobí nič iné, iba zväčší globálnu premennú `cas` o jedna. Výsledný efekt teda bude, že premenná `cas` bude plniť funkciu stopiek.

Funkcia, ktorú týmto spôsobom voláme, musí mať parameter. V našom prípade je to parameter `dt`. Do tohto parametra dostane funkcia skutočný čas, ktorý uplynul od posledného volania. Niekedy sa totiž stane, že sa niečo zdrží (napríklad sa udeje nejaké komplikované vykresľovanie) a funkcia sa nespustí po sekunde, ale po jeden a pol sekunde. Ak chceme, aby hra reagovala aj na takéto náhodné zdržania, je dobré túto informáciu zapracovať. V tejto ukážke to ale nerobíme.

Dobre. Máme nastavené, ako sa má okno vykresľovať, máme nastavené, čo sa má diať, keď dôjde k nejakej udalosti od myši či klávesnice a máme nastavené, že každú sekundu sa má zmeniť čas. Môžeme to celé spustiť. Udeje sa tak na riadku 38.

Úloha 4: Pochopte a vyskúšajte.

Predošlá ukážka popisuje všetky veci, s pomocou ktorých sme schopní naprogramovať jednoduchú hru. Ale ako ste si isto všimli, táto ukážka niektoré veci iba naznačuje a celkom isto neposkytuje odpovede na mnoho podstatných otázok, napríklad „Aký je kód klávesy `medzera`?“ alebo „Ako zistiť, či je niektorý kláves práve stlačený?“ Odpoveď je – čítajte manuál. Ak to myslíte s programovaním hier vážne, aj tak sa tomu nevyhnete. Manuál ku knižnici Pyglet nájdete na adrese http://pyglet.org/doc/programming_guide.pdf Na stránke http://pyglet.org nájdete aj ďalšie zaujímavé informácie.

Môžeme sa teda pustiť do skutočnej hry. Bude to remake prastarej hry, ktorá kedysi dávno behávala na počítačoch PMD-85. Potrebujeme na to ale tri obrázky. Obrázok ufóna, rozmery 11 × 8 pixelov, bude uložený v súbore `09-alien.png`, obrázok rakety, rozmery 27 × 13 pixelov, bude uložený v súbore `09-rocket.png` a obrázok strely, rozmery 1 × 3 pixely v súbore `09-missile.png`. Okrem toho budeme potrebovať aj nejaký randál. Obrátime sa na niektorú databanku zvukov, napríklad http://soundbible.com a stiahneme si nejaký pekný výbuch – napríklad súbor `BombExplosion.wav` z adresy http://soundbible.com/107-Bomb-Explosion-1.html Všetky tieto súbory umiestnime v adresári, v ktorom budeme písať program a môžeme začať pracovať.

Celú hru si môžete pozrieť v nasledujúcej ukážke. Pokúste sa najprv poriadne prečítať a pochopiť kód. Až potom si prečítajte komentár a overte si, či ste správne pochopili, čo sa kde deje:

```
 1  import pyglet
 2  from pyglet.window import key
 3  import random
 4
 5  window = pyglet.window.Window(320,200,fullscreen = True)
 6  window.set_mouse_visible(False)
 7  pyglet.gl.glEnable(pyglet.gl.GL_BLEND)
 8  pyglet.gl.glBlendFunc(pyglet.gl.GL_SRC_ALPHA,
 9                        pyglet.gl.GL_ONE_MINUS_SRC_ALPHA)
10
11  class Alien:
12      alien_img = pyglet.resource.image('09-alien.png')
13
14      def __init__(self,hra):
15          self.hra = hra
16          self.x = random.randint(0,28)
17          self.y = 192
18          self.rychlost = 3
19          if random.random() < 0.08:
20              self.rychlost = 10
21
22      def pohni(self):
23          self.y -= random.randint(0,self.rychlost)
24          if self.y <= 0:
25              self.hra.dosiel(self)
26
27      def kresli(self):
28          Alien.alien_img.blit(11*self.x,self.y)
29
30  class Raketa:
31      def __init__(self,hra):
32          self.image = pyglet.resource.image('09-rocket.png')
33          self.x = 14
34
35      def kresli(self):
36          self.image.blit(11*self.x - 8,0)
37
38      def vlavo(self):
39          if self.x > 0:
40              self.x -= 1
41
42      def vpravo(self):
43          if self.x < 28:
44              self.x += 1
```

```
45  class Strela:
46      missile_img = pyglet.resource.image('09-missile.png')
47
48      def __init__(self,hra,x):
49          self.hra = hra
50          self.x = x
51          self.y = 0
52          self.rychlost = 10
53
54      def pohni(self):
55          self.y += self.rychlost
56          self.hra.strielam(self)
57
58      def kresli(self):
59          Strela.missile_img.blit(11*self.x + 6,self.y)
60
61  class Hra:
62      vybuch = pyglet.resource.media('BombExplosion.wav',streaming=False)
63
64      def __init__(self):
65          self.zivoty = 3
66          self.vypustit = 100
67          self.aliens = []
68          self.raketa = Raketa(self)
69          self.strely = []
70
71      def kresli(self):
72          if self.zivoty < 0:
73              prehra = pyglet.text.Label("Prehral si!!!",
74                          font_size=30,
75                          x=window.width // 2,
76                          y=window.height // 2,
77                          anchor_x='center')
78              prehra.draw()
79              return
80          if self.vypustit == 0 and len(self.aliens) == 0:
81              vyhra = pyglet.text.Label("Vyhral si!!!",
82                          font_size=30,
83                          x=window.width // 2,
84                          y=window.height // 2,
85                          anchor_x='center')
86              vyhra.draw()
87              return
88          zivoty = pyglet.text.Label(str(self.zivoty),
89                          font_size=18,
90                          x=10, y=window.height - 18)
91          zivoty.draw()
92          pocet = pyglet.text.Label(str(self.vypustit),
93                          font_size=18,
94                          x=window.width - 50,
95                          y=window.height - 18)
96          pocet.draw()
97          for i in self.aliens:
98              i.kresli()
99          for i in self.strely:
100             i.kresli()
101         self.raketa.kresli()
```

```
102       def spracujKlaves(self,klavesa):
103           if klavesa == key.LEFT:
104               self.raketa.vlavo()
105           elif klavesa == key.RIGHT:
106               self.raketa.vpravo()
107           elif klavesa == key.SPACE:
108               self.strely.append(Strela(self,self.raketa.x))
109
110       def krok(self,dt):
111           if random.random() < 0.05 and self.vypustit > 0:
112               self.aliens.append(Alien(self))
113               self.vypustit -= 1
114           for i in range(int(dt/0.1)):
115               for i in self.strely:
116                   i.pohni()
117               for i in self.aliens:
118                   i.pohni()
119
120       def strielam(self,strela):
121           trafeni = []
122           for alien in self.aliens:
123               if alien.x == strela.x and alien.y < strela.y:
124                   trafeni.append(alien)
125           if trafeni == []:
126               if strela.y > 200:
127                   self.strely.remove(strela)
128               return
129           najnizsi = trafeni[0]
130           for alien in trafeni:
131               if alien.x < najnizsi.x:
132                   najnizsi = alien
133           self.aliens.remove(najnizsi)
134           self.strely.remove(strela)
135           self.vybuch.play()
136
137       def dosiel(self,alien):
138           self.zivoty -= 1
139           self.aliens.remove(alien)
140
141   h = Hra()
142
143   @window.event
144   def on_key_press(symbol, modifiers):
145       h.spracujKlaves(symbol)
146
147   @window.event
148   def on_draw():
149       window.clear()
150       h.kresli()
151
152   pyglet.clock.schedule_interval(h.krok,0.1)
153   pyglet.app.run()
```

Úloha 5: Naozaj ten program poriadne prečítajte!!!

Tak ste práve prečítali necelé tri strany kódu. Čo sa z neho dá vidieť na prvý pohľad?

V prvom rade to, že v celej hre sa vyskytujú štyri objekty: `Alien`, `Raketa`, `Strela` a `Hra`. Trieda `Alien` reprezentuje ufónov, ktorí útočia na matičku Zem. Objektov tejto triedy môže byť v hre viacero naraz. Objekt triedy `Raketa` bude iba jeden. Napriek tomu, že je to raketa, bude pobiehať po zemi a ostreľovať ufónov. Objektov typu `Strela`, ktoré bude raketa vypúšťať, môže byť viacero. Objekt `Hra` to bude všetko riadiť.

Ďalšie zaujímavé veci si zaslúžia pozornejšie čítanie. Spomeňme najprv drobné detaily – na riadku 5 bolo okno otvorené v režime `fullscreen` – teda na celú obrazovku. Túto voľbu je dobré mať počas ladenia vypnutú a zapnúť ju až keď všetko funguje. Na riadku 6 skryjeme myš a na riadkoch 7 až 9 nastavíme `pyglet` tak, aby priesvitné `png` obrázky vykresľoval ako priesvitné.

Ďalej si treba všimnúť, ako fungujú súradnice. Na riadku 5 sa vytvorí okno s rozmermi 320 × 200 bodov. Jeden ufón má šírku obrázka 11 bodov, takže sa ich na šírku obrazovky vmestí vedľa seba 320 // 11 teda 29. Ufóni, raketa aj strely budú mať teda namiesto súradnice `x` číslo od 0 do 28. Keď si pozriete metódu `kresli` v jednotlivých objektoch, vidíte, že toto číslo sa vždy násobí jedenástimi (šírkou ufóna), aby sa objekty dostali na správne miesto. Raketa aj strela sa ešte o kúsok posunú, aby ich stred bol vždy pod stredom patričného ufóna.

Keby sme predošlú ukážku programovali poriadne, namiesto konkrétnych čísel by sa patrilo používať premenné a vlastnosti obrázkov. To umožňuje program jednoduchšie meniť a prispôsobovať, keby to bolo treba.

Ďalej si všimnite rozdiel, akým načítavame obrázok pre triedu `Alien` a pre triedu `Raketa`. V prípade rakety sme si spravili v konštruktore (vo funkcii `__init__`) premennú `self.image` a do nej sme obrázok načítali. Keby sme to tak ale spravili aj v prípade triedy `Alien`, tak by sa musel obrázok načítať pre každého mimozemšťana zvlášť a zbytočne by to v počítači zaberalo veľa pamäte, pretože všetci ufóni sú rovnakí a obrázok pre nich stačí načítať pre celú triedu raz. Ako to spraviť šikovnejšie, vidíte v našom programe. Mimo všetkých metód si vytvoríme premennú `alien_img` a do nej načítame obrázok (riadok 12). Táto premenná je spoločná pre všetky objekty triedy `Alien`. Keď ju chceme použiť, pristupujeme k nej ako k `Alien.alien_img` (riadok 28).

Poďme sa teraz pozrieť na jednotlivé triedy. Začnime triedou `Alien`. V konštruktore sa ufónovi nastaví jeho hra, vyberie sa stĺpec, v ktorom bude ufón padať, súradnica `y` sa nastaví na vrch obrazovky a nastaví sa rýchlosť. Tá je väčšinou 3, ale s pravdepodobnosťou 8% môže byť aj 10, pretože niektorí ufóni sú výsadkári.

Okrem konštruktora má `Alien` ešte metódu `pohni`, ktorá ním pohne o náhodný počet bodov (od nula do jeho rýchlosti) smerom nadol a zavolá metódu hry `dosiel`, ak sa ufónovi podarí prísť až na zem a metódu `kresli`, ktorá ufóna nakreslí tam, kde práve je. To, že pri pohybe použijeme náhodné čísla, spôsobí, že sa ufón nepohybuje rovnomerne, ale trhane. Lepšie to tak vyzerá.

Trieda `Raketa` má konštruktor `__init__`, ktorý načíta obrázok a nastaví raketu do stredu, metódu `kresli`, ktorá ju nakreslí a metódy `vlavo` a `vpravo`, ktoré pohnú raketou doľava a doprava.

Trieda `Strela` má konštruktor, ktorý jej nastaví určenú pozíciu a rýchlosť, metódu `pohni`, ktorá strelou pohne a zavolá metódu hry `strielam`, aby hra zistila, či strela náhodou niečo nezasiahla.

Konečne sa dostávame k samotnej triede `Hra`. Táto trieda bude mať svoju vlastnú premennú `vybuch`, do ktorej načítame zvuk výbuchu, ktorý bude hra spúšťať vždy, keď nejaká strela zničí

ufóna (riadok 62). Súbor, v ktorom máme zvuk uložený, sa nazýva `BombExplosion.wav`. Parameter `streaming` udáva, či sa má zvuk prehrávať z disku. Ak je jeho hodnota `False`, zvuk si program načíta do pamäte. To je vhodné pri krátkych a často sa opakujúcich zvukoch. Ak chcete prehrávať hudbu v pozadí, je vhodnejšie ju celú do pamäte neťahať, prehrávať to z disku a použiť štandardnú hodnotu `True`.

V konštruktore `__init__` nastavíme počet životov hráča na 3, počet ufónov, ktorí majú byť vypustení na 100, vyrobíme raketu a spravíme si zoznamy ufónov a striel, ktoré sú práve na obrazovke. Oba tieto zoznamy sú zatiaľ prázdne.

Okrem konštruktora bude mat samotná hra päť metód. Metóda `kresli` bude mať na starosti vykreslenie celej scény. Túto metódu budeme volať, keď nastane udalosť `on_draw`. Metóda `spracujKlaves` bude mať na starosti spracovanie udalosti z klávesnice. Budeme ju volať, keď nastane udalosť `on_key_press`. Metóda `krok` sa bude volať každú desatinu sekundy a bude hýbať všetkými vecami, ktoré nezávisia od klávesnice – teda strelami a ufónmi. Okrem toho bude mať na starosti aj vypúšťanie nových ufónov. Metódu `strielam` sme už spomínali pri strele. Táto metóda pre zadanú strelu zistí, či strela niekoho nezasiahla a vybaví prípadnú likvidáciu ufónov a nepotrebných striel. Posledná metóda `dosiel` sa zavolá, keď sa nejakému ufónovi podarí doraziť až na zem.

Poďme sa jednotlivým metódam venovať podrobnejšie. V metóde `kresli` sa najprv pozrieme, či už hráč náhodou neprehral. Ak je počet životov menší ako nula, vypíše sa tragická správa a viac sa nekreslí nič (riadky 72 – 79). Ak sa nám už medzičasom podarilo vyhubiť všetkých ufónov (počet tých, ktorých ešte treba vypustiť aj počet tých, ktorí sú na obrazovke je nulový), vypíšeme správu o tom, že hráč vyhral a tiež s vykresľovaním skončíme (riadky 80 – 87). Inak vypíšeme počet zostávajúcich životov (riadky 88 – 91), počet ufónov, ktorí ešte čakajú na vypustenie (riadky 92 – 96) a na riadkoch 97 až 101 vykreslíme všetkých aktívnych ufónov, strely a raketu.

Metóda `spracujKlaves` vie rozlíšiť tri klávesy. V prípade stlačenia šípky vpravo alebo vľavo zavolá patričnú funkciu rakety a pohne ňou doprava alebo doľava. V prípade stlačenia medzery vyrobí na mieste, na ktorom sa práve raketa nachádza strelu.

Metóda `krok` sa najprv pozrie, či ešte treba nejakých ufónov vypustiť a ak áno, tak to s pravdepodobnosťou 5% urobí. Potom si zistí, o koľko sa volanie funkcie zdržalo oproti predpísanej desatine sekundy (presný čas má uložený v premennej `dt`) a koľkokrát je táto hodnota väčšia ako desatina sekundy, toľkokrát pohne ufónmi aj raketami.

Metóda `strielam` je pomerne zložitá. Najprv si vyrobíme zoznam `trafeni` a na riadkoch 122 až 124 do neho uložíme všetkých ufónov, cez ktorých strela preletela. Ak zoznam ostal prázdny skontrolujeme, či už strela nepreletela cez okraj a ak áno, tak ju zrušíme a skončíme (riadky 125 až 128). Ak zoznam prázdny nie je, tak na riadkoch 129 až 132 nájdeme ufóna, ktorý je spomedzi zasiahnutých najnižšie (dobre si pozrite, ako to hľadanie presne funguje!). Najnižšieho ufóna aj strelu zrušíme a prehráme zvuk výbuchu.

Pri metóde `dosiel` iba znížime počet životov o 1 a zrušíme ufóna, ktorému sa podarilo pristáť.

Teraz to ešte treba spustiť. Na riadku 141 vyrobíme novú hru `h`. Nastavíme udalosti prekresľovania a spracovania klávesnice na patričné funkcie hry `h`. Na riadku 152 nastavíme, že sa má každú desatinu sekundy volať funkcia `h.krok` a na riadku 153 to celé spustíme.

Úloha 6: Pochopte a vyskúšajte.

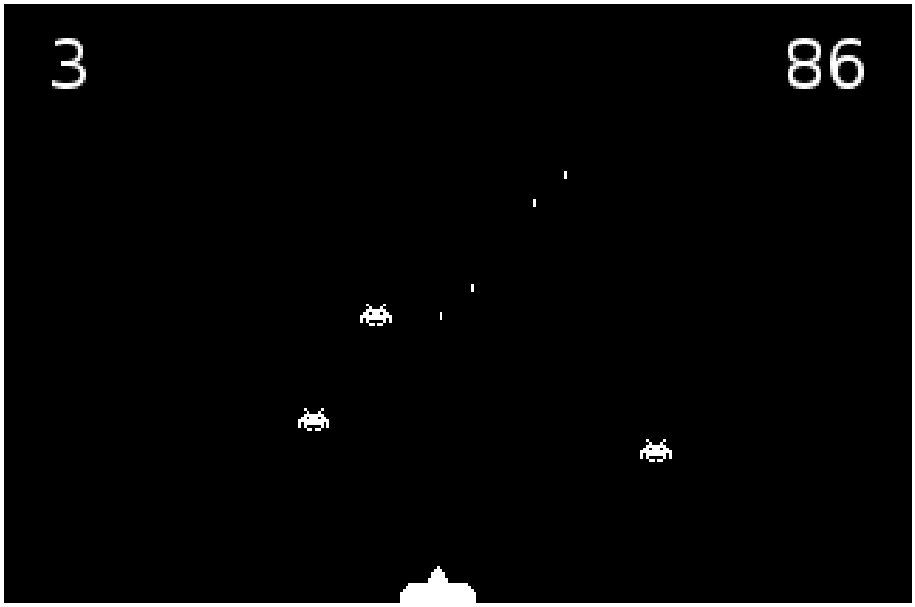
Úloha 7: Pridajte ďalšiu rýchlosť – ufónov, ktorí sú rýchlejší, než bežní, ale pomalší, než výsadkári.

Úloha 8: Zabezpečte, aby mohli byť naraz vypustené najviac tri strely.

Úloha 9: Zmeňte nastavenia a obtiažnosť hry. Pohrajte sa s rýchlosťou ufónov, s pravdepodobnosťou, s akou sú vypúšťaní, s pravdepodobnosťou, s akou sa spúšťajú výsadkári. Dosiahnite stav, aby bola hra uhrateľná, ale náročná.

Úloha 10: Pridajte kláves, ktorým sa po skončení hry bude dať spustiť nová hra.

Úloha 11: Naprogramujte nejakú inú hru.



Obrázok 11: Screenshot z hry

10. lekcia

Popoluška a zlatá priadka alebo „vlákna, triedenie a zložitosť“

Iste si všetci pamätáte príbeh o tom, ako zlá macocha pomiešala hrach so šošovicou a nechala to Popoluške triediť. Triedenie vtedy prevzali namiesto Popolušky holúbky, aby mohla ísť na bál (podľa Grimmovcov) prípadne do kostola (podľa Dobšinského) a dať sa tam dokopy s krásnym princom.[25] Holúbky všetko správne potriedili, princ si zobral Popolušku a žili šťastne dokiaľ nepomreli.

Už táto pradávna rozprávka naznačuje, aké je triedenie dôležité. V tejto lekcii sa ním budeme zaoberať podrobnejšie. A to triedením v istom zmysle komplikovanejším, než bolo to Popoluškine. Nebudeme totiž rozdeľovať jednu zmes na dve hromádky, ale v našom prípade nám zlá macocha zverí mericu hrachu a povie: „Zoraďte hrášky od najmenšieho po najväčší!“

Keď to budeme programovať, nebudeme mať samozrejme k dispozícii kopu hrachu, ale pole s nejakými náhodnými číslami. Zoradiť čísla (alebo iné objekty) od najmenšieho po najväčšie sa dá viacerými spôsobmi. A tu sa dostávame k ďalšiemu zaujímavému aspektu programovania: nie každý spôsob je rovnako dobrý. V tejto lekcii si ukážeme niektoré z nich a povieme si niečo o tom, ako sa určuje zložitosť algoritmov, aby sme vedeli porovnať, ktorý z nich bude lepší a ktorý horší.

Len tak naprogramovať jednotlivé triediace algoritmy a porozprávať o nich by bolo ale príliš jednoduché vzhľadom na to, akí dobrí už pri práci s Pythonom ste. Aby bolo lepšie vidno, ako jednotlivé triediace algoritmy fungujú, rozhodli sme sa spraviť program tak, aby bolo graficky znázornené, čo sa v našom poli deje. Celé pole budeme vykresľovať ako stĺpcový graf a na vykresľovanie použijeme knižnicu `pyglet`, s ktorou ste sa mali možnosť zoznámiť v predošlej lekcii.

Aby sme ale mohli tento veľkolepý projekt zavŕšiť, treba spraviť dve veci. V prvom rade si potrebujeme vytvoriť objekt, ktorý bude mať vlastnosti poľa, okrem toho ale bude mať aj nejaké ďalšie – bude sa vedieť nakresliť a bude počítať, koľko zmien sa v ňom udialo. V druhom rade, aby sme sa vrámci nášho triediaceho algoritmu nemuseli starať o vykresľovanie, potrebujeme si náš program rozdeliť na dve vlákna. Každé vlákno bude bežať samostatne – ak máte počítač s viacerými jadrami, môže každé dokonca bežať na inom procesore. Prvé vlákno bude triediť pole a o nič iné sa nebude starať. Druhé vlákno bude mať na starosti vykresľovanie a nebude nijak zasahovať do triedenia.

A tak sme sa od Popolušky nenápadne prepracovali k zlatej priadke, ktorá je v našej ľudovej slovesnosti odborníčka na vlákna. Kým však vlákna začneme používať, musíme sa o tom, ako s nimi Python pracuje, čo-to naučiť.

Rovnako ako na iné veci má Python na prácu s vláknami knižnicu. Tá sa volá `threading` a pre naše potreby bude stačiť, keď si z nej importujeme objekt `Thread`, s pomocou ktorého vieme vlákno vytvoriť.

Predstavte si teraz, že máme nejakú dôležitú funkciu, ktorú potrebujeme spustiť trikrát, od výsledkov jedného spustenia nezávisí žiadne ďalšie a buď funkcia procesor príliš nezaťažuje, alebo

25 Dobšinského variant dával princovi príležitosť k rôznym rafinovaným úskokom, ako napríklad že nechal natrieť kostolnú lavicu smolou, nech sa tam Popoluška prilepí, ale na jeho veľké sklamanie to nedržalo dosť pevne…

máme k dispozícii dostatok jadier, aby mohli bežať viaceré funkcie naraz. V nasledujúcom príklade bude touto funkciou funkcia `funkcia`. Táto iba o sebe podá správu, že začala, počká tri sekundy a podá správu, že skončí.

```
154   import time
155   from threading import Thread
156
157   def funkcia(i):
158       print("Štartuje dôležitá funkcia",i)
159       time.sleep(3)
160       print("Končí dôležitá funkcia",i)
161
162   for i in range(3):
163       funkcia(i)
164
165   print("\n\n")
166
167   for i in range(3):
168       v = Thread(target=funkcia, args=(i,))
169       v.start()
```

Prvú možnosť, ako funkciu trikrát spustiť, môžete vidieť na riadkoch 9 a 10. Je to klasická metóda, na akú ste boli navyknutí doteraz. Celý ten cyklus bude trvať o niečo málo viac, ako deväť sekúnd.

Dá sa to ale spraviť aj šikovnejšie. Na riadkoch 14 až 16 vytvoríme tri vlákna. Každému vláknu povieme, ktorú funkciu má spustiť (parameter `target`) a s akými parametrami (parameter `args`). Akonáhle vlákno spustíme, náš hlavný program sa oň viacej nestará a môže si ďalej robiť vlastné veci, teda napríklad vytvoriť ďalšie vlákno. Výsledný efekt bude ten, že všetky tri funkcie odštartujú v tesnom závese, o tri sekundy všetky tri viac-menej naraz skončia a teda tie tri vykonania funkcie zaberú dokopy iba niečo viac ako tri sekundy. Oproti predošlému spôsobu je to o šesť sekúnd lepšie.

Vzhľadom na to, že väčšina funkcií nefunguje tak, že väčšinu svojej činnosti prespí, ale reálne zaťažuje procesor, je rozumné vlákna využívať buď vtedy, keď potrebujete rozdeľovať prácu medzi viacero procesorov, alebo vtedy, keď treba čakať na nejaké vstupy alebo odosielať nejaké výstupy. Pozrime sa na nasledujúci príklad.

Predstavte si, že máte rýchle pripojenie na internet (ak ho naozaj máte, nemusíte si predstavovať) a píšete program, v ktorom potrebujete stiahnuť tri stránky. Jednoduchá verzia takého programu môže vyzerať takto:

```
 1   import urllib.request
 2   import time
 3
 4   def stiahni(url):
 5       poziadavka = urllib.request.Request(url)
 6       return urllib.request.urlopen(poziadavka).read()
 7
 8   vystup = ["","",""]
 9
```

```
10   start = time.time()
11
12   vystup[0] = stiahni("http://www.smnd.sk")
13   vystup[1] = stiahni("http://www.sme.sk")
14   vystup[2] = stiahni("http://www.nrsr.sk")
15
16   print("Trvalo to %.2f s" % (time.time() - start))
```

Na prácu s internetom potrebujeme knižnicu `urllib`. Na riadkoch 4 až 6 máme krátku funkciu, ktorá zo zadanej adresy vytvorí objekt `poziadavka`, otvorí ju (funkcia `urlopen`) a vráti dáta, ktoré odtiaľ príjme (funkcia `read`). Táto funkcia je skutočne stručná – keby sme robili seriózny aplikáciu, patrilo by sa celé to načítanie dať do bloku `try` a správne ošetriť výnimky. Na riadkoch 12 až 14 načítame tri stránky, uložíme si ich do zoznamu `vstup` a nakoniec vypíšeme čas potrebný na stiahnutie.

Vráťme sa teraz k predpokladu, že vaša linka je naozaj rýchla. V tom prípade situáciu zdržujú linky na strane serverov. Ku vám by mohli pokojne prúdiť dáta od všetkých troch serverov naraz, ale keďže je program spravený tak, že sa jednotlivé stránky sťahujú postupne, je to celé zbytočne pomalé.

V takejto situácii je vhodné použiť vlákna. Keďže ale chceme zmerať čas sťahovania, potrebujeme zistiť, kedy všetky tri vlákna skončia. Ony totiž vôbec nemusia skončiť naraz, každé žije svojim vlastným životom. Na synchronizáciu vecí ohľadom vlákien má Python štruktúru nazývanú fronta, po anglicky `Queue`. Fronta je objekt s množstvom funkcií o ktorých sa dočítate v manuáli[26], tu si však spomenieme iba štyri z nich. S pomocou `put` môžete do fronty niečo vložiť – to niečo väčšinou predstavuje popis nejakej úlohy. Metódu `get` použije vlákno, keď si chce niektorú úlohu z fronty vyzdvihnúť a spracovať. Ak vo fronte nič nie je, `get` čaká, kým sa tam niečo neobjaví. Keď sa vlákno k úlohe dostane, vykoná ju a s pomocou príkazu `task_done` dá fronte vedieť, že tá jeho úloha je hotová. Metóda `join` nerobí nič iné, len čaká, kým budú všetky úlohy vo fronte spracované. Náš program na sťahovanie stránok môžeme s pomocou vlákien a fronty prepísať takto:

```
 1   import threading
 2   import queue
 3   import urllib.request
 4   import time
 5
 6   fronta = queue.Queue()
 7   vystup = ["","",""]
 8
 9   def stiahni(url):
10       poziadavka = urllib.request.Request(url)
11       return urllib.request.urlopen(poziadavka).read()
12
13   def sosadlo():
14       vstup = fronta.get()
15       vystup[vstup[0]] = stiahni(vstup[1])
16       print("%s hotovo." % vstup[1])
17       fronta.task_done()
18
19   start = time.time()
```

26 http://docs.python.org/3/library/queue.html

```
20  for i in range(3):
21      t = threading.Thread(target = sosadlo)
22      t.daemon = True
23      t.start()
24
25  for stranka in [(0,"http://www.smnd.sk"),
26                  (1,"http://www.sme.sk"),
27                  (2,"http://www.nrsr.sk")]:
28      fronta.put(stranka)
29
30  fronta.join()
31
32  print("Trvalo to %.2f s" % (time.time() - start))
```

Na začiatku naimportujeme všetky potrebné knižnice, vytvoríme frontu `fronta` a zoznam do ktorého budeme ukladať načítané dáta `vystup`. Funkcia `stiahni` je úplne rovnaká, ako v predošlom príklade. Na vytvorenie jednotlivých vláken použijeme funkciu `sosadlo`. Táto funkcia počká, kým sa pre ňu vo fronte nájde úloha (riadok 14), Každá úloha je dvojica pozostávajúca z čísla a adresy. Číslo určuje, do ktorej položky zoznamu `vystup` sa má uložiť výsledok, adresa určuje, aká stránka sa má sťahovať. Na riadku 15 sa do patričnej položky uloží patričný text, na riadku 16 sa vypíše oznam a na riadku 17 vlákno fronte oznámi, že je všetko hotové.

Na riadkoch 20 až 23 vytvoríme tri vlákna, ktoré spustia funkciu `sosadlo`. Na riadku 22 nastavíme jednotlivým vláknam, že sú to démoni – znamená to, že hlavný program skončí až vtedy, keď skončia aj tieto vlákna. Ak by sme to nespravili, vlákna môžu ďalej bežať aj po skončení hlavného programu. Vlákna naštartujeme (riadok 23), ale každé z nich ostane visieť na riadku 14, pretože fronta je zatiaľ prázdna.

Na riadkoch 25 až 28 do fronty napcháme všetky stránky, ktoré chceme stiahnuť spolu s poradovými číslami, aby vlákna vedeli, kam majú uložiť výstup. Akonáhle sa fronta naplní, vlákna si jednotlivé stránky rozoberú a začnú sťahovať. Ak nemáte práve veľmi zaneprázdnenú linku, ľahko sa môže stať, že sa načítavanie neskončí v tom poradí, v akom ste ho zadali. Niektoré stránky sú kratšie, niektoré dlhšie a na strane niektorého servera môžu byť voľnejšie linky.

Na riadku 30 program počká, kým všetky vlákna neohlásia fronte, že sú hotové. Môžete porovnať dosiahnutý čas sťahovania s predošlým prístupom. V niektorých našich pokusoch sa časy príliš nelíšili, v niektorých bol druhý rýchlejší až trojnásobne. Záleží na stave liniek na vašej strane a na strane serverov, z ktorých sťahujete.

Úloha 1: Vyskúšajte a pochopte. Vyskúšajte zadať do vstupu aj stránky na horšie prístupných serveroch a porovnajte časy dosiahnuté metódou s vláknami a bez vlákien.

Teraz, keď už vieme, ako fungujú vlákna, poďme sa pozrieť na to, ako vytvoriť zoznam, ktorý bude sám seba vedieť vykresliť a bude si pamätať, koľko zmien sa v ňom odohralo.

Budeme sa samozrejme snažiť vytvoriť novú triedu. Keď ale vytvoríme objekt tejto triedy (nech sa bude volať napríklad `p`), budeme s ním musieť vedieť spraviť nasledujúce operácie:

- `print(p[2])` (teda zistiť, čo sa nachádza na treťom mieste)
- `p[1] = 17` (teda nastaviť druhý prvok na 17)
- `print(len(p))` (teda vedieť o sebe povedať, aký je dlhý)

Takéto veci sa v Pythone riešia pomocou špeciálnych metód, ktoré majú dopredu určené meno väčšinou začínajúce a končiace dvoma podtržníkmi. Zatiaľ sme sa stretli s dvomi takýmito metódami – s metódou `__init__`, ktorá sa spúšťa automaticky pri vytvorení objektu a s metódou `__name__`, s ktorou sme sa stretli, keď sme hovorili o dekorátoroch a s pomocou ktorej sme zisťovali meno funkcie. Teraz si povieme o ďalších troch takýchto metódach. Aby objekt vedel fungovať ako zoznam alebo slovník, treba mu vytvoriť každú z týchto metód.

- Metóda `__getitem__(self,key)` má ako výsledok vrátiť obsah prvku označeného kľúčom `key`. Keď teda budete používať `p[2]`, je to to isté, ako keby ste používali `p.__getitem__(2)`
- Metóda `__setitem__(self,key,value)` nastaví prvok označený kľúčom `key` na hodnotu `value`. Keď teda použijete príkaz `p[1] = 17`, v skutočnosti sa vykoná `p.__setitem__(1,17)`
- Metóda `__len__(self)` vráti veľkosť zoznamu. Funkcia `len(p)` vráti teda tú istú hodnotu, ako `p.__len__()`

Vyzbrojení týmito vedomosťami môžeme začať vytvárať naše pole. Nasledujúci program rozdelíme na dve časti. V prvej spravíme všetko, čo sa týka novej triedy, v druhej pridáme samotné triedenie a ostatné veci potrebné na spoluprácu s knižnicou `pyglet`. Takže najprv tá prvá časť:

```
 1  import pyglet
 2  from pyglet.window import key
 3  import random
 4  import time
 5  from threading import Thread
 6
 7  window_width = 800
 8  window_height = 600
 9
10  def obdlznik(x1, y1, x2, y2):
11      pyglet.graphics.draw(4,
12                           pyglet.gl.GL_QUADS,
13                           ('v2f', (x1, y1, x1, y2,
14                                    x2, y2, x2, y1)))
15
16
17  class Pole:
18      def __init__(self,length,maximum):
19          self.dx = window_width // length
20          self.dy = window_height // (maximum + 1)
21          self.changes = 0
22          self.data = []
23          for i in range(length):
24              self.data.append(random.randint(1,maximum))
25          self.size = max(2,self.dx - 2)
26          size1 = max(2,self.dy - 2)
27          self.size = min(self.size,size1)
28      def __getitem__(self,key):
29          return self.data[key]
30      def __setitem__(self,key,value):
31          self.data[key] = value
32          self.changes += 1
33      def __len__(self):
34          return len(self.data)
```

```
35      def draw(self):
36          for i in range(len(self.data)):
37              x = i * self.dx + (self.dx - self.size) // 2
38              y = self.data[i] * self.dy
39              obdlznik(x,y,x+self.size,y+self.size)
40              obdlznik(x + self.size // 2,0,
41                       x + self.size // 2 + 1,y)
42          pocet = pyglet.text.Label(str(self.changes),
43                       font_size=30,
44                       x=100,
45                       y=window_height - 50,
46                       anchor_x='center')
47          pocet.draw()
```

Na začiatku sme importovali potrebné knižnice, na riadku 7 a 8 sme nastavili premenné určujúce šírku a výšku okna. Na riadkoch 10 až 14 sme vytvorili funkciu, ktorá nakreslí obdĺžnik. Obdĺžnik je určený súradnicami protiľahlých rohov `x1`, `y1` a `x2`, `y2`. Funkcia `draw` z knižnice `pyglet` sa odvoláva na knižnicu OpenGL, s ktorou `pyglet` spolupracuje. Jej podrobný popis je ale nad rámec týchto skrípt.

Dostávame sa k samotnej triede `Pole`. Jej konštruktor má dva parametre – dĺžku poľa a veľkosť maximálneho člena. Najprv sa v ňom nastavia premenné `dx` a `dy`, ktoré určujú, koľko priestoru ostane pre jeden prvok pri vykresľovaní v smere osi x a v smere osi y. V atribúte `changes` si budeme pamätať, koľko zmien sa v poli udialo. Atribút `data` bude obsahovať samotné dáta poľa. Na riadkoch 23 a 24 ho naplníme daným počtom náhodných čísel v určenom rozsahu. Na riadkoch 25 až 27 vypočítame veľkosť štvorčeka, ktorý sa bude vykresľovať namiesto hodnoty. Nemal by presiahnuť ani `dx - 2` ani `dy - 2`, ale mal by byť aspoň 2.

Na riadkoch 28 až 34 vyrobíme metódy, o ktorých bola reč vyššie. Všimnite si metódu `__setitem__`, ktorá je spravená tak, že vždy, keď sa nastaví niektorej položke poľa nová hodnota, zvýši atribút `changes` o 1.

Na riadkoch 37 až 47 je metóda, ktorá má na starosti vykresľovanie. Pre každý prvok zoznamu `data` sa vypočítajú súradnice `x` a `y` na ktorých sa má vykresliť. Na určených súradniciach sa vykreslí štvorček a k nemu dlhá stopka až na spodok obrazovky. Na záver sa vypíše počet zmien, ktoré sa zatiaľ v poli udiali.

Skvelé. Naše pole, ktoré je schopné sa vykresliť a viesť o sebe záznamy by sme mali hotové. Môžeme začať triediť. Samotné triedenie sa odohrá vo funkcii `bubblesort`. Toto triedenie zatiaľ nebudeme komentovať, poďme sa najprv pozrieť na zvyšok programu:

```
48  p = Pole(70,50)
49
50  def bubblesort():
51      for i in range(1,len(p)):
52          for j in range(i,0,-1):
53              if p[j] < p[j-1]:
54                  p[j],p[j-1] = p[j-1],p[j]
55                  time.sleep(0.25)
56              else:
57                  break
58
59  window = pyglet.window.Window(window_width,window_height,
60                                  fullscreen = False)
61  window.set_mouse_visible(False)
```

```
62  @window.event
63  def on_draw():
64      window.clear()
65      p.draw()
66
67  def update(dt):
68      pass
69
70  t = Thread(target = bubblesort)
71  t.setDaemon(True)
72  t.start()
73
74  pyglet.clock.schedule_interval(update,0.04)
75  pyglet.app.run()
```

Na riadku 48 vytvoríme pole so sedemdesiatimi prvkami, pričom hodnoty budú od 1 do 50. Na riadkoch 50 až 57 je už spomínaná procedúra `bubblesort`, ktorá bude pole triediť. Na riadkoch 59 až 61 vytvoríme okno, do ktorého budeme kresliť a nastavíme, aby v ňom nebol viditeľný kurzor myši. Na riadkoch 62 až 65 je vykresľovacia funkcia `on_draw`. Nedeje sa v nej nič zvláštne, len to, že sa všetko zmaže a nakreslí sa pole.

Zaujímavá je funkcia `update`. Funkcia totiž obsahuje jediné kľúčové slovo `pass`, ktoré znamená, že funkcia vôbec nič nerobí. Prirodzene sa natíska otázka, na čo je nám taká funkcia dobrá. Vec sa má totiž tak, že funkcia `on_draw` sa volá iba vtedy, keď sa knižnica `pyglet` dozvie, že sa niečo zmenilo. Ale naše triedenie sa bude odohrávať v úplne inom vlákne, takže sa knižnica nemá odkiaľ o zmene dozvedieť.[27] Preto sme si urobili funkciu `update` a voláme ju dvadsaťpäťkrát za sekundu. Funkcia síce nič nerobí, ale vďaka tomu, že sa zavolá, `pyglet` vie, že má prekresliť scénu.

Na riadkoch 70 až 72 naštartujeme vlákno s triedením a na riadkoch 74 a 75 spustíme volanie funkcie `update` a naštartujeme veci týkajúce sa knižnice `pyglet`.

Úloha 2: Pochopte a vyskúšajte. Funkciu `bubblesort` zatiaľ podrobne čítať nemusíte, skúste iba z toho, čo vám program ukazuje, prísť na to, ako to funguje.

Konečne máme za sebou všetky podporné práce a môžeme sa venovať samotnému triedeniu. Ako prvé sme zvolili takzvané bublinkové triedenie. Pripomeňme ešte raz jeho kód:

```
1   def bubblesort():
2       for i in range(1,len(p)):
3           for j in range(i,0,-1):
4               if p[j] < p[j-1]:
5                   p[j],p[j-1] = p[j-1],p[j]
6                   time.sleep(0.25)
7               else:
8                   break
```

Pre každý prvok v poli (nazvime ho dočasne Hubert) spravíme nasledujúcu operáciu: Pozrieme sa, či je Hubert menší, než prvok od neho vľavo a ak je, vymeníme ich. Toto opakujeme, až kým nenarazíme na prvok, ktorý už nie je väčší, než Hubert alebo kým Hubert nedorazí na úplný začiatok. (Pripomíname, že premenná `j` môže v cykle na riadku 3 nadobudnúť všetky hodnoty od `i`

27 To, že sa jedno vlákno nemá odkiaľ dozvedieť, čo sa deje v inom, nie je celkom pravda. Vlákna môžu medzi sebou komunikovať, dočasne si pre seba vyhradzovať premenné a pomocou semaforov zabezpečiť, aby nejaký kus kódu vykonávalo iba jedno vlákno. Detaily nájdete v manuáli ku knižnici `threading`.

do 1. Nulu už nenadobudne.) Keď to postupne spravíme so všetkými prvkami v poli, každý prvok takto prebublá na svoje miesto a nakoniec bude celé pole utriedené.

Bubblesort sme ako naše úvodné triedenie zvolili z dvoch dôvodov: Jednak preto, lebo sa ľahko programuje. V druhom rade preto, že je to jedno z najhorších známych triedení. Aby sme ho ale len tak naprázdno neohovárali, bude sa treba aspoň trochu zaoberať matematikou a zistiť, koľko priradení bude treba vykonať, keď chceme pole týmto spôsobom triediť. Tej matematiky sa tu až do konca lekcie bude vyskytovať viac. Ak sa vám počítanie bridí, ale chcete vedieť, ako fungujú ďalšie typy triedenia, pokojne tú matematiku preskočte.

Predstavme si, že máme desaťprvkové pole, ktoré je zoradené presne opačne, než by sme chceli, teda vyzerá takto:

10 9 8 7 6 5 4 3 2 1

V prvom kroku vymeníme 9 s 10. Na to potrebujeme spraviť *dve* priradenia do poľa. Pole bude vyzerať takto:

9 10 8 7 6 5 4 3 2 1

Keď budeme posúvať dopredu osmičku, budeme ju musieť vymeniť najprv s desiatkou a potom s deviatkou a budeme potrebovať urobiť *štyri* priradenia do poľa.

Takto to postupne urobíme so všetkými prvkami. Nakoniec budeme dostávať dopredu jednotku. Budeme ju musieť vymeniť až s deviatimi číslami a vyjde nás to teda na *osemnásť* priradení. Dokopy sme tých priradení vykonali

$$2+4+6+8+10+12+14+16+18=90$$

Vo všeobecnom prípade, kedy prvkov nemáme 10, ale n, by tých výmen bolo

$$2+4+6+\ldots+2(n-1)=2[1+2+3+\ldots+(n-1)]=2\frac{(n-1)n}{2}=n^2-n$$

Ako ste si isto všimli, v predošlom riadku sa už vyskytovala matematika – konkrétne súčet aritmetickej postupnosti a úprava výrazov. Vyšlo nám, že vo všeobecnom prípade budeme potrebovať maximálne n^2-n výmen. Keď autorom knihy neveríte (čo je prístup, ku ktorému povzbudzujeme), tak sa na ten riadok ešte chvíľu pozerajte a pochopte ho.

Znalci vedia, že n^2-n je kvadratická funkcia. Keď sa programátori bavia o tom, aký je algoritmus zložitý, tak im často netreba vedieť, koľko operácií je to presne. Bez ohľadu na to, či je výsledok n^2-n alebo $20n^2+4n+55$, algoritmus má pre nich kvadratickú zložitosť (zapisuje sa to aj, že má zložitosť $O(n^2)$), pretože si v počte operácií všímajú iba tú jeho časť, ktorá má sklony pri zväčšujúcom sa n rásť najrýchlejšie a aj pri nej ignorujú konštantu, ktorou je vynásobená. Ak by úlohu algoritmus vo všeobecnom prípade zvládal na $27n+16$ operácii, mal by zložitosť $O(n)$ (tzv. lineárna zložitosť) a ak by to bolo 2^n+n^5, mal by zložitosť $O(2^n)$ (tzv. exponenciálna zložitosť). Rýchlosti algoritmov sa v prvom rade porovnávajú podľa zložitosti. Algoritmus, ktorý má zložitosť $O(n^2)$ je pokladaný za výrazne lepší, ako algoritmus, ktorý má zložitosť $O(2^n)$, pretože napr. 2^{1000} je nepredstaviteľne väčšie, než 1000^2 – v celom viditeľnom vesmíre je len asi 2^{177} atómov. [28]

28 Samozrejme je nutné si pri takomto rozlišovaní zachovať zdravý rozum a zvážiť napríklad, aké veľké hodnoty n pripadajú do úvahy. Ak by sme mali dva algoritmy, prvý by potreboval vykonať $1000n$ operácií, druhý n^2 operácií a vedeli by sme, že n môže byť maximálne 10, je určite lepšie použiť druhý algoritmus, aj keď má kvadratickú zložitosť a prvý má lineárnu. Keby ale malo byť n milión, prvý algoritmus bude výhodnejší.

Pokúsme sa teraz náš algoritmus vylepšiť. Prvé vylepšenie, ktoré človeku môže napadnúť je, že zakaždým vymieňať iba susedné prvky je zbytočné obmedzenie. Keď niektorý prvok potrebujeme dostať na jeho miesto, bolo by oveľa jednoduchšie posunúť všetky väčšie prvky o jedno miesto vpravo a zatrieďovaný prvok rovno priradiť tam, kam patrí. Takto nemusí byť zatrieďovaný prvok postupne priradený do všetkých pozícií a veľa priradení tak ušetríme. Algoritmus (nazýva sa triedenie vsúvaním) by mohol po vylepšení vyzerať napríklad takto:

```
 1   def insertsort():
 2       for i in range(1,len(p)):
 3           a = p[i]
 4           n = i - 1
 5           while n >= 0 and p[n] > a:
 6               p[n+1] = p[n]
 7               time.sleep(0.625)
 8               n -= 1
 9           p[n+1] = a
10           time.sleep(0.625)
```

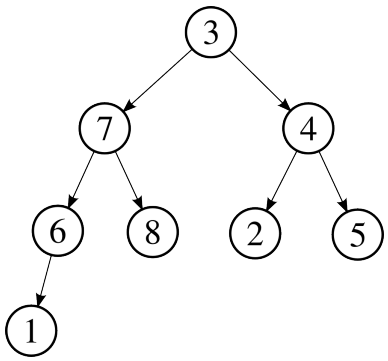
Prvok zoznamu, ktorý ideme upratať, si uložíme do premennej `a`. Prvky zoznamu posúvame doprava, kým sú väčšie než prvok v premennej `a`, alebo kým neprídeme na začiatok zoznamu (riadky 5 až 8). Zdržanie na riadku 7 má polovičnú hodnotu oproti zdržaniu v bubblesorte, pretože robíme iba jedno priradenie namiesto dvoch. Keď sú už všetky potrebné prvky presunuté, prvok z premennej `a` sa uloží na správne miesto.

Úloha 3: Pochopte a vyskúšajte. Nezabudnite pri vytváraní vlákna zmeniť funkciu, ktorú má vlákno spustiť. Aká je zložitosť triedenia `insertsort`?

Ak ste správne vyriešili problém zložitosti, zistili ste, že v najhoršom prípade (ak by bolo pole zoradené naopak) bude funkcia presúvať $1+2+3+\ldots+(n-1)=\frac{(n-1)n}{2}=\frac{n^2}{2}-\frac{n}{2}$ prvkov. Na jednu stranu je to dvakrát menej, ako pri bublinkovom triedení, čo je skvelé. Ak niečo predtým trvalo hodinu, teraz to bude trvať iba pol hodiny. Na druhú stranu, zložitosť je stále $O(n^2)$. Ak budeme triediť milión čísel, budeme musieť spraviť namiesto bilióna operácií iba pol bilióna a to sme si teda veľmi nepomohli, lebo v rozumnom čase nestihneme ani jedno, ani druhé. Existuje vôbec triedenie, ktoré by bolo natoľko lepšie, že by malo aj lepšiu zložitosť?

Existuje. Dokonca viacero. Z viacerých kandidátov tu predvedieme haldové triedenie alias `heapsort`. Kým sa ale dostaneme k samotnému triedeniu, musíme si povedať niečo viac o dátových štruktúrach, ktoré budeme používať.

V kapitole, v ktorej sme hľadali najkratšiu cestu medzi dvoma mestami, ste sa stretli s dátovou štruktúrou nazývanou graf. Teraz budeme potrebovať istý špeciálny graf, ktorý sa nazýva binárny strom. Binárny strom môžete vidieť na obrázku 12.

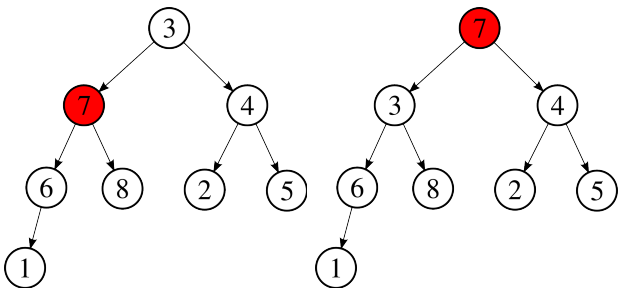


Obrázok 12: Binárny strom

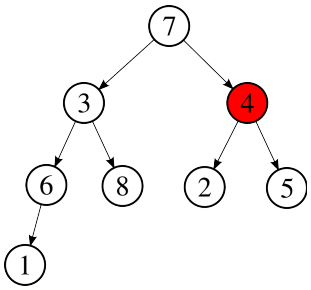
Binárny strom sa skladá z viacerých úrovní. Vrchol na každej úrovni je spojený s jedným vrcholom na vyššej úrovni. Výnimku tvorí vrchol na najvyššej úrovni, ktorý sa nazýva koreň. (Áno, je to zvláštne, ale binárne stromy majú koreň hore.) Každý vrchol môže byť spojený hranami s vrcholmi na nižších úrovniach, ale tie hrany môžu byť maximálne dve. (Kvôli tomu sa ten strom volá binárny.) Od nášho binárneho stromu budeme ešte navyše chcieť, aby boli jeho vrcholy umiestnené čo najvyššie a v poslednom riadku čo najviac vľavo, takže ak má mať taký strom osem vrcholov, tak musí vyzerať presne tak, ako na obrázku 12. Do tohto stromu teraz prepíšeme (pekne po riadkoch) čísla z nášho poľa a môžeme začať triediť.

V prvom rade budeme chcieť, aby sa náš binárny strom stal haldou. Halda je strom, ktorý má tú peknú vlastnosť, že číslo, ktoré je zapísané v niektorom vrchole, musí byť väčšie, než všetky čísla pod ním.[29] Náš strom evidentne takou haldou zatiaľ nie je, pretože napríklad číslo 3, ktoré je zapísané v koreni stromu je menšie, než obe čísla vo vrcholoch pod ním.

Ako z nášho stromu takú haldu urobiť? Jednoducho. Prejdeme všetky pozície v strome pekne od vrchu a ak zistíme, že číslo na niektorej pozícii je väčšie, než číslo nad ním, čísla vymeníme a naše číslo budeme ďalej posúvať hore až dovtedy, kým nebude mať nad sebou väčšie alebo kým nebude v koreni. Udeje sa teda toto:

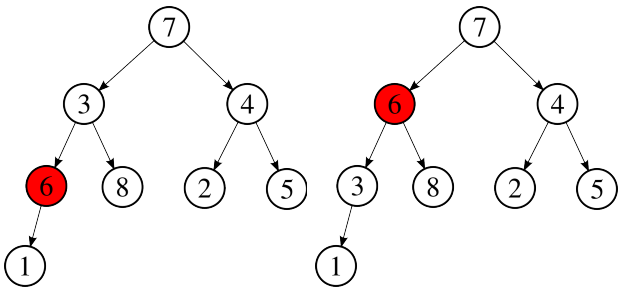


Sedmičku sme presunuli vyššie.

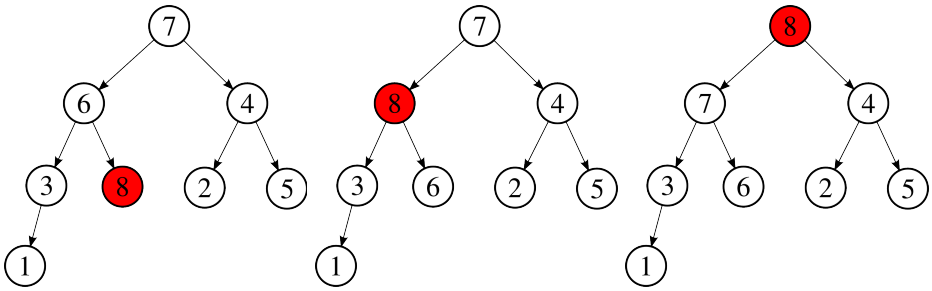


Štvorka má nad sebou vyššie číslo, tak s ňou nehýbeme.

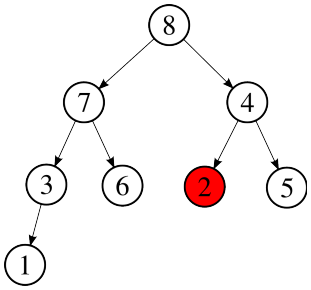
29 Binárny strom a halda sú celkom názorne nakreslené tu: http://xkcd.com/835/



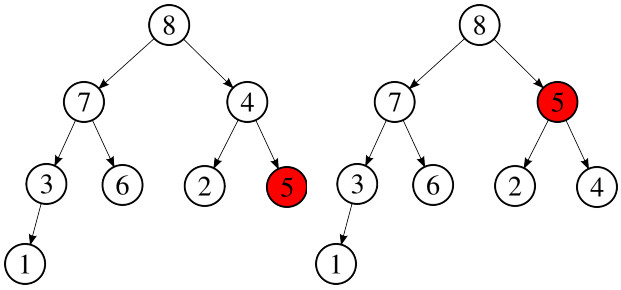
Šestku sme presunuli vyššie.



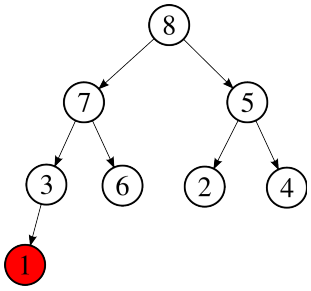
Osmičku sme na správne miesto dostali až na dva kroky.



S dvojkou hýbať netreba.



Päťku sme presunuli vyššie.



S jednotkou hýbať netreba.

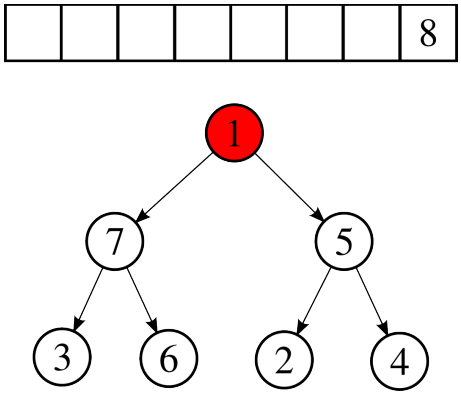
Teraz je už z nášho binárneho stromu halda. Sú v nej tie isté čísla, ako predtým, ale každé má pod sebou len nižšie čísla.

Skôr, než sa pustíme ďalej, pokúsme sa vypočítať, koľko najviac priradení nás stálo spraviť zo stromu haldu. Každá výmena sú dve priradenia. Začali sme presúvať na $n-1$ pozíciach (na všetkých okrem koreňa) a číslo z každej pozície sme vždy presúvali vyššie, takže sme určite nespravili viac priradení, než je $2nh$, kde h je výška stromu.[30]

30 Tento odhad by sa samozrejme dal zlepšiť. Veď sme napríklad namiesto n – 1 použili n. Ale takto je zápis jednoduchší a krajší a ukáže sa, že takýto odhad je dostatočný.

Fajn. Haldu máme. Ako z nej získať utriedené pole? Na to použijeme nasledujúci trik:

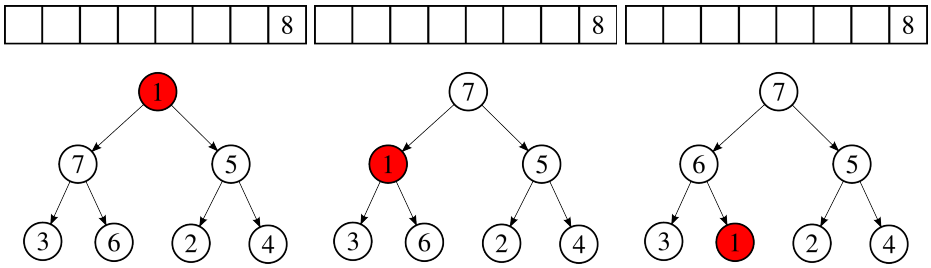
Zatiaľ vieme iba jedno – najväčší prvok poľa (osmička) je zaručene v koreni. Pokojne ho môžeme odložiť na koniec nášho poľa. Musíme ho ale niečím nahradiť. A aby si náš strom zachoval tú peknú vlastnosť, že je odvrchu a zľava plne obsadený, musíme ho nahradiť tou jednotkou vľavo dole.



Obrázok 13: Zaradená osmička

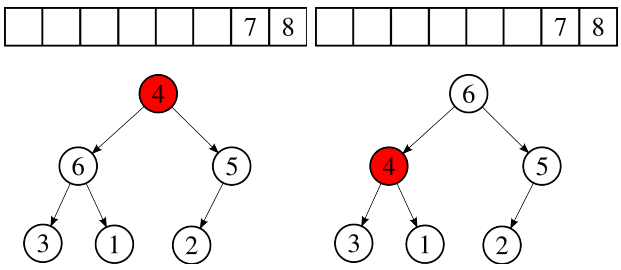
Lenže nastal problém. V momente, keď sme jednotku zapísali do koreňa, náš skvelý strom prestal byť haldou. (Pozri obrázok 13.) Treba to nejako opraviť.

Teraz sme ale v trochu inej situácii, ako keď sme haldu vytvárali. Tú jednotku musíme s niečím vymeniť. Keby sme jednotku vymenili s päťkou, päťka by síce už nemala nad sebou menšie číslo, ale dostala by sa nad sedmičku, čo nemôže byť. Preto sa potrebujeme pozrieť, ktorý z dolných susedov jednotky je väčší a vymeniť jednotku s ním. Toto opakujeme, až kým znovu nemáme haldu:

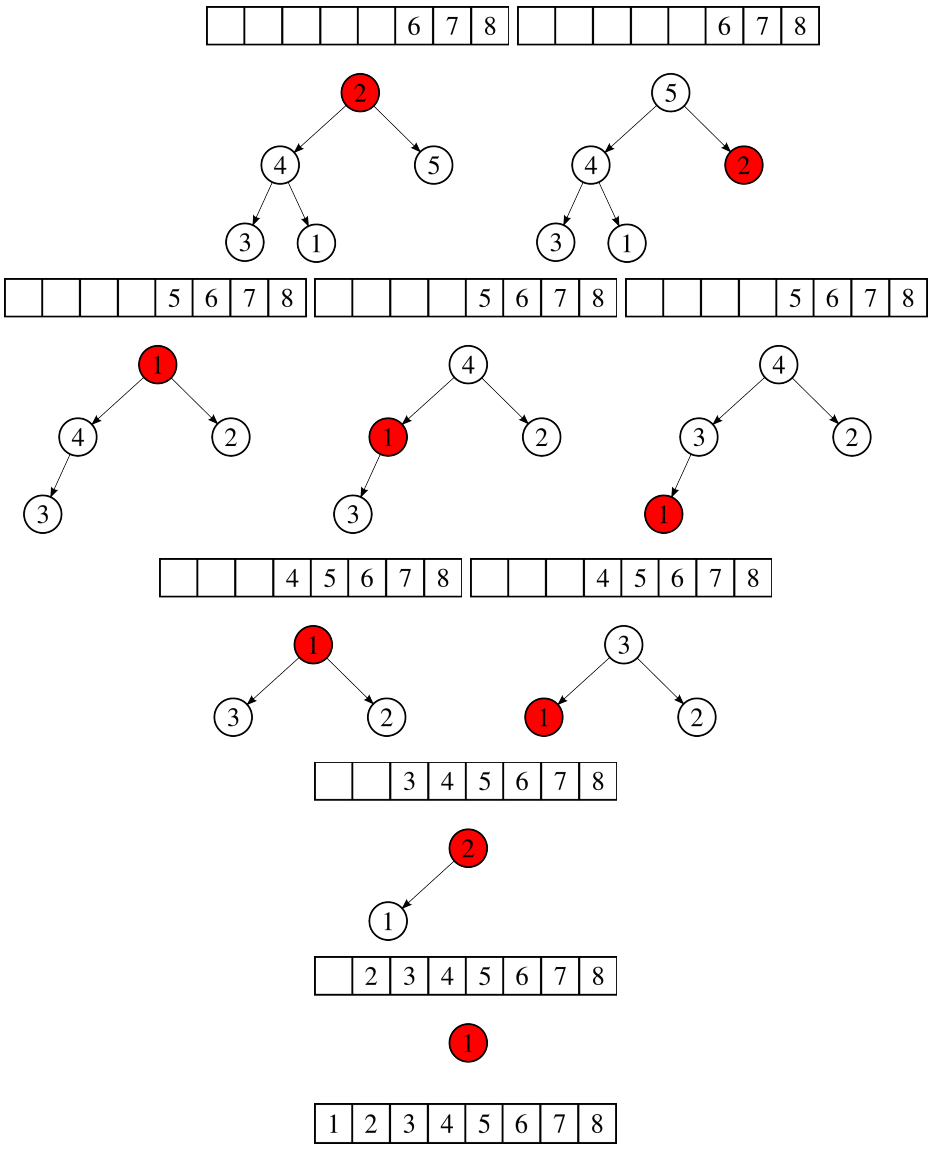


Všimnite si, že pri prechode z druhého na tretie poschodie sa jednotka musela vymeniť so šestkou a nie s trojkou, lebo inak by sa trojka dostala nad šestku.

Opäť teda máme haldu a najväčší prvok máme v koreni. Zaradíme ho teda do už utriedených čísel a nahradíme ho číslom na poslednej pozícii, teda štvorkou a opäť upravíme strom, aby bol haldou.



Toto sa bude opakovať, až kým nebude pole utriedené.



A je utriedené.

Koľko operácií nás stála táto fáza? Opäť sa $n-1$ krát dostalo nejaké číslo do koreňa a bolo ho treba dopraviť na správne miesto. Zaručene sme teda nepoužili viac, než $2nh$ priradení, kde h je výška stromu.

Poďme sa teraz na chvíľu venovať tajomnému písmenu h. Potrebovali by sme vedieť, koľko poschodí bude mať binárny strom, ktorý vznikne z poľa s n prvkami. Toto je jedna z úloh, na ktoré je lepšie pozrieť sa „z druhej strany“ a opýtať sa – keď mám strom s h poschodiami, koľko čísel sa mi doňho zmestí?

V prvom riadku je jedno číslo. V druhom dve. V treťom štyri, v štvrtom osem atď. Vždy v ďalšom riadku je dvakrát viac čísel, ako v predošlom. Počty čísel v riadkoch sú tým pádom mocniny dvojky. Výnimku tvorí posledný riadok, ktorý nemusí byť celkom plný. V minimálnom prípade je v poslednom riadku jediné číslo. Všetkých čísel v strome je tým pádom

$$1+2+4+8+\ldots+2^{h-2}+1$$

(tá jednotka na konci je to jedno číslo v poslednom riadku). Tí, čo vedia sčítať geometrickú postupnosť vedia, že je to dokopy 2^{h-1}. Môžete si to overiť na našom prvom strome – mal štyri riadky, v poslednom bolo len jedno číslo a dokopy sa do neho vošlo 2^3 teda 8 čísel.

V maximálnom prípade je posledný riadok plný. Všetkých čísel v strome bude teda

$$1+2+4+8+\ldots+2^{h-2}+2^{h-1}=2^h-1$$

pre počet čísel *n* teda platí, že

$$2^{h-1}\leq n<2^h$$

Teraz si spomenieme, ako fungoval logaritmus (alebo sa opýtame staršieho brata) a z predošlej nerovnosti dostaneme

$$h-1\leq \log_2 n<h$$

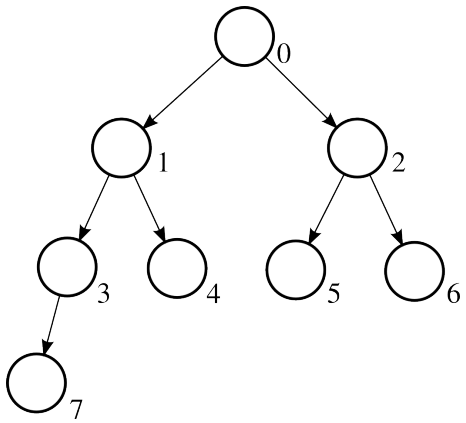
Vyhrali sme. Vieme, že h sa líši od $\log_2 n$ o menej ako 1. Počet priradení, ktoré sme použili počas celého triedenia, je teda menší, než $2nh+2nh=4nh\approx 4n\log_2 n$.

Porovnajme to s triedením vsúvaním. Predstavte si, že by ste museli triediť milión čísel. V prípade vsúvania v najhoršom prípade použijete $\frac{999\,999.1\,000\,000}{2}=499\,999\,500\,000$ teda asi päťsto miliárd priradení. V prípade haldového triedenia budete potrebovať maximálne $4\,000\,000\,.\,\log_2 1\,000\,000=4\,000\,000\,.\,19{,}9\approx 79\,726\,274$ čiže asi osemdesiat miliónov priradení. To je asi šesťtisíckrát menej. A dôležité je to, že na bežnom počítači to prvé nestihnete a to druhé hej.

Je zrejmé, že dvojkový logaritmus z milióna je číslo podstatne menšie, než milión – je to o niečo menej, ako 20. Zložitosť haldového triedenia nebude teda $O(n^2)$, ale $O(n\log n)$.[31]

Všetci, ktorí sa prehrýzli predošlou horou teórie, si zaslúžia nehynúcu česť a úctu. Môžeme začať programovať. Najprv sa ale oplatí zamyslieť, ako budeme čísla do haldy ukladať. Mohli by sme si spraviť graf podobne ako v ôsmej lekcii. Ale to by sme museli najprv presúvať čísla zo zoznamu do grafu a potom naspäť. Omnoho lepšie by bolo udržať čísla nejako v poli.

Môžeme spraviť napríklad vec, ktorú vidíte na obrázku 14. Za koreň stromu vyhlásime nultý prvok zoznamu, druhý riadok budú tvoriť prvý a druhý prvok, tretí riadok bude tvoriť tretí až šiesty prvok zoznamu atď.



Obrázok 14: Očíslovaný strom

Potrebovali by sme ale nejako rýchlo zistiť, ktorý prvok je s ktorým spojený. A na to sa nám bude hodiť elegantná matematická finta. Všimnite si, že vrchol s číslom n má pod sebou vždy

31 Všimnite si, že v zápise zložitosti je použitý zápis $\log n$, ktorým matematici označujú logaritmus so základom 10 namiesto $\log_2 n$. To má dve príčiny. Prvá je tá, že v informatike sa vyskytuje dvojkový logaritmus častejšie, ako desiatkový a tak v niektorých článkoch informatici používajú zápis $\log n$ pre dvojkový logaritmus. Druhý dôvod je ten, že dvojkový logaritmus nejakého čísla je vždy asi 3,32-krát väčší, než desiatkový a konštanty sa v zápise zložitosti zanedbávajú, takže je ten zápis dobre aj s desiatkovým logaritmom.

vrcholy s číslami $2n+1$ a $2n+2$ a nad sebou vrchol $(n-1)//2$ (pripomeňme, že s pomocou `//` sa v Pythone 3 zapisuje celočíselné delenie – teda napr. $14//3=4$).

Vyzbrojení touto fintou môžeme naprogramovať haldové triedenie:

```
 1   def heapsort():
 2       for i in range(1,len(p)):
 3           n = i
 4           while n > 0 and p[n] > p[(n-1)//2]:
 5               p[n],p[(n-1)//2] = p[(n-1)//2],p[n]
 6               n = (n-1)//2
 7               time.sleep(1.25)
 8       for i in range(len(p)-1,0,-1):
 9           p[i],p[0] = p[0],p[i]
10           time.sleep(1.25)
11           n = 0
12           while 2*n + 1 < i:
13               a = 2*n + 1
14               if 2*n + 2 < i and p[2*n + 2] > p[2*n + 1]:
15                   a = 2*n + 2
16               if p[a] > p[n]:
17                   p[a],p[n] = p[n],p[a]
18                   n = a
19                   time.sleep(1.25)
20               else:
21                   break
```

Na riadkoch 2 – 7 vytvoríme haldu. Postupne prejdeme všetky prvky poľa (hlavný cyklus `for`) a každý posúvame v strome nahor dovtedy, kým nad ním nie je väčšie číslo, alebo kým nie je úplne navrchu (cyklus `while`, v premennej `n` je číslo vrchola, s ktorým práve pracujeme). To, že sme úplne hore, spoznáme podľa toho, že `n == 0`.

Na riadkoch 8 – 21 sa udeje samotné triedenie. Hodnota `i` predstavuje posledný prvok, ktorý ešte v strome máme, zvyšok už nie je strom, ale utriedené pole. Na riadku 9 vymeníme koreň stromu (v ktorom je aktuálne najväčšie číslo poľa) s posledným prvkom a tým ho zaradíme na správne miesto. V cykle na riadkoch 12 – 21 dostaneme číslo, ktoré sa ocitlo v koreni na také miesto, aby strom opäť tvoril haldu. Na riadkoch 13 až 15 zistíme, či je väčšia hodnota pod aktuálnym vrcholom vľavo alebo vpravo – číslo vrchola s väčšou hodnotou si uchovávame v premennej `a`. Dávame pri tom pozor, aby sme nevliezli do už utriedenej časti, ktorá začína `i`-tym prvkom. Ak je hodnota vo vrchole `a` väčšia, ako hodnota vo vrchole `n`, tak ich vymeníme a pokračujeme ďalej (riadky 16 – 19). Ak nie je, prvok je už na mieste a môžeme vnútorný cyklus skončiť.

Úloha 4: Pochopte, naprogramujte a kochajte sa. Nezabudnite zase upraviť spustenie vlákna na nové triedenie.

Na záver lekcie spomeňme niečo jednoduché.

Dá sa dokázať, že vo všeobecnosti je $O(n \log n)$ najlepšia zložitosť, ktorá sa dá pri triedení dosiahnuť. Ale ak nastanú isté špecifické okolnosti, situácia sa dá ešte zlepšiť.

Predstavte si napríklad, že dopredu vieme, že v našom poli budú zaručene iba prirodzené čísla od 1 do 50. Táto podmienka je zhodou okolností splnená – pripomeňme si, ako bolo pole

inicializované. Nič nám nebráni pole prezrieť a zapamätať si, koľko v ňom bolo jednotiek, koľko dvojok, koľko trojok… a potom podľa tohto záznamu pole prepísať. Na začiatok dať správny počet jednotiek, po ňom správny počet dvojok… Táto finta sa nazýva počítacie triedenie alias `countingsort`. Naprogramované to vyzerá takto:

```
 1   def countingsort():
 2       pocty = 51 * [0]
 3       for i in p:
 4           pocty[i] += 1
 5           time.sleep(0.625)
 6       pozicia = 0
 7       for i in range(1,51):
 8           for j in range(pocty[i]):
 9               p[pozicia + j] = i
10               time.sleep(0.625)
11           pozicia += pocty[i]
```

Najprv si vytvoríme zoznam `pocty`, ktorý bude mať 51 prvkov – samé nuly. V cykle na riadkoch 3 – 5 sa prezrie celé pole `p` a pre každú hodnotu, ktorá sa v ňom nájde, zväčšíme patričnú položku v zozname `pocty` o jedna. Potom v cykle na riadkoch 7 – 11 do pôvodného poľa zapíšeme vždy toľko kusov danej hodnoty, koľko nájdeme v zozname `pocty`. Pomocná premenná `pozicia` nám hovorí, odkiaľ máme začať zapisovať. Ak má pole n prvkov, pri čítaní spravíme n zápisov do zoznamu `pocty` a pri zápise spravíme n zápisov do poľa `p`. Dokopy je to $2n$ zápisov. Algoritmus má teda zložitosť $O(n)$. Ak teda máte triediť pole, o ktorom viete, že sa v ňom nachádza iba niekoľko typov položiek, zvážte, či nie je vhodné použiť počítacie triedenie.

Úloha 5: Vyskúšajte.

Úloha 6: Skúste prerobiť metódu `draw` tak, aby sa pole vykresľovalo do štvrtiny obrazovky, ktorú určíte v konštruktore. Spustite potom štyri vlákna tak, aby v každom bežalo jedno triedenie a všetky by sa ukazovali na obrazovke súčasne, každé v inej štvrtine.

Zaujímavých triedení je ešte niekoľko. Spomeňme napríklad `quicksort`, ktorý má veľmi elegantný algoritmus alebo `timsort`, ktorý vymyslel v roku 2002 Tim Petersen a ktorý ako triediaci algoritmus používa vo svojich procedúrach Python. Detaily zvedavý čitateľ nájde na wikipedii.

Úloha 7: Pozrite si videá týkajúce sa triedenia, na ktoré nájdete odkazy nižšie.

http://www.youtube.com/watch?v=ywWBy6J5gz8

http://www.youtube.com/watch?v=kPRA0W1kECg

Záver

Šťastne ste sa dostali na koniec skrípt k jazyku Python. Čo teraz? Nuž čo. Programujte. Čokoľvek. Naštudujte poriadne knižnicu `pyglet` a urobte hru. Naštudujte si kúsok nejakej vedy (matematiku, fyziku, chémiu, biológiu, čokoľvek …) a skúste naprogramovať niečo v nej. Píšte údržbové skripty pre linux. Skúste sa s pomocou pythonu hrať s obrázkami. Pustite sa do súťaženia. Navštívte stránky http://www.ksp.sk a http://oi.sk, skúste vyriešiť niečo na http://projecteuler.net, prihláste sa do http://www.smnd.sk/main/spongia Alebo si len tak naprogramujte nejaký jednoduchý skript pre radosť a tešte sa z neho. Mnoho šťastia prajem.

Anino