RIEŠENIE PROBLÉMOV A PROGRAMOVANIE

JÁN GUNIŠ, VIERA MICHALIČKOVÁ, MARTIN CÁPAY, ĽUBOMÍR ŠNAJDER

EURÓPSKA ÚNIA
Európsky sociálny fond
Európsky fond regionálneho rozvoja

OPERAČNÝ PROGRAM
ĽUDSKÉ ZDROJE

MINISTERSTVO
ŠKOLSTVA, VEDY,
VÝSKUMU A ŠPORTU
SLOVENSKEJ REPUBLIKY

it akadémia

Tento projekt sa realizuje vďaka podpore z Európskeho sociálneho fondu
a Európskeho fondu regionálneho rozvoja v rámci Operačného programu Ľudské zdroje

www.minedu.sk www.employment.gov.sk/sk/esf/ www.itakademia.sk

Riešene problémov a programovanie

Spracované v rámci národného projektu IT Akadémia – vzdelávanie pre 21. storočie

Bratislava 2020

Riešenie problémov a programovanie

Spracované s finančnou podporou národného projektu IT Akadémia – vzdelávanie pre 21. storočie

Autori: JÁN GUNIŠ, VIERA MICHALIČKOVÁ, MARTIN CÁPAY, ĽUBOMÍR ŠNAJDER

Editor: Ján Guniš

Recenzenti:

Jazyková úprava: neprešlo jazykovou úpravou

Grafická úprava:

Vydavateľ: Centrum vedecko-technických informácií SR, Bratislava

Rok vydania: 2020

Vydanie : 1. vydanie

Tlač:

ISBN:

Bratislava 2020

Tento projekt sa realizuje vďaka podpore z Európskeho sociálneho fondu v rámci Operačného programu Ľudské zdroje.

Obsah

Predslov

Milí žiaci,

do rúk sa Vám dostáva učebnica predmetu Riešenie problémov a programovanie. Zámerom nás autorov, je predstaviť vám programovanie ako silný nástroj na riešenie problémov. V jednotlivých kapitolách učebnice vám predstavujeme aj pokročilejšie nástroje a techniky programovania. Robíme to tak ale z dôvodu potreby použitia nástroja pri riešení konkrétneho problému a nie preto, že jazyk Python tento nástroj má a je potrebné ho ovládať.

Pri výbere problémov, ktoré sme do učebnice zaradili sme siahli do viacerých oblastí ľudského poznania. Radi by sme vám ukázali, že programovanie nie je len samoúčelný nástroj hŕstky nadšencov. Programovanie je pre nás, a veríme že bude aj pre vás, jeden zo spôsobov, ako efektívne riešiť rôznorodé problémy z rôznych oblastí.

S programovaním máme bohaté, niekoľkoročné skúsenosti. Aj my sme riešili počiatočné problémy, keď nás počítač neposlúchal a nerozumel nám čo od neho chceme. Postupom času sme ich prekonali a zistili sme, že:

- programovanie je zábavné,
    - čím viac sa programovaniu venujete, tým viac vás táto činnosť napĺňa a uspokojuje,
- ak viete programovať, ste pre zamestnávateľa zvlášť zaujímaví, vedieť programovať je žiadaná zručnosť pre 21. storočie ,
    - dôležité nie sú ani tak vaše vedomosti z konkrétneho programovacieho jazyka ale najmä spôsob myslenia akým pristupujete k problémom a k ich riešeniu,
- programovať = riešiť problémy,
    - programovanie nie je o príkazoch programovacieho jazyka, programovanie je o schopnosti riešiť problémy využitím prostriedkov, ktoré programovací jazyk ponúka,

Veríme, že učebnica bude užitočným pomocníkom pri vašom štúdiu. Držíme vám palce, aby sa vám v budúcnosti podarilo vyriešiť množstvo ďalších, omnoho ťažších problémov ako sú tie, ktoré vám v učebnici predstavujeme.

S pozdravom autori učebnice
Jano, Vierka, Martin a Ľubo

Učebnica bola vyvíjaná v spolupráci s partnerskými strednými školami zapojenými do Národného projektu IT Akadémia – vzdelávanie pre 21. storočie.

Autormi učebnice sú vysokoškolskí učitelia z Univerzity Pavla Jozefa Šafárika v Košiciach a Univerzity Konštantína Filozofa v Nitre, ktorí sa podieľali pri tvorbe jednotlivých kapitol nasledovne: Ján Guniš (Úvod, Výučba predmetu Riešenie problémov a programovanie, 1, 2, 5, 6, 11, 16, 19, 20, 25), Viera Michaličková (8, 9, 17, 18, 21, 22, 23), Martin Cápay (4, 7, 12, 13, 14, 15, 24) a Ľubomír Šnajder (3, 10).

1. Programovanie v jazyku Python

Kľúčové slová

programovanie, Python, IDLE, Shell, IDE, konzola, aritmetické operácie, logické operácie, reťazec, zoznam, výrez, typ hodnoty, pretypovanie, import modulu, modul turtle, modul math, vlastný modul, cyklus for, cyklus while, range, vlastná funkcia, parametre funkcie, návratová hodnota, dokumentačný reťazec, podmienený príkaz, chyba syntaktická, chyba logická, chyba behová, výnimky, ladenie programu, zápis do súboru, čítanie zo súboru, grafické rozhranie, tkinter

Čo sa naučíme a čo si precvičíme

- zopakujeme si základný kurz programovania,
- zopakujeme si základy programovania v jazyku Python.

Jazyk Python

Jazyk Python je multi-paradigmový, multiplatformový a interpretovaný jazyk.

Jazyk Python nenúti programátora, aby používal konkrétny štýl programovania, ale dáva mu slobodu používať viaceré. Preto prívlastok multi-paradigmový. Možno máte skúsenosť so štruktúrovaným programovaním. V tomto predmete si, okrem iného, ukážeme, aké výhody prináša objektový prístup k riešeniu problémov.

V jazyku Python môžeme programovať pod rôznymi operačnými systémami. Programy, ktoré vytvoríme v prostredí MS Windows, budú rovnako dobre fungovať v systéme Linux alebo MAC OS X. Preto prívlastok multiplatformový.

Poznámka na okraj

Interpreter jazyka Python nevytvára žiaden spustiteľný kód (exe alebo com súbor). Pre vykonávanie Python-ovských programov musí byť v počítači inštalovaný interpreter jazyka Python. Preto prívlastok interpretovaný. Jedným z dôsledkov je možnosť interaktívneho režimu, pri ktorom sú výrazy automaticky vyhodnocované a my hneď vidíme výsledok vyhodnotenia.

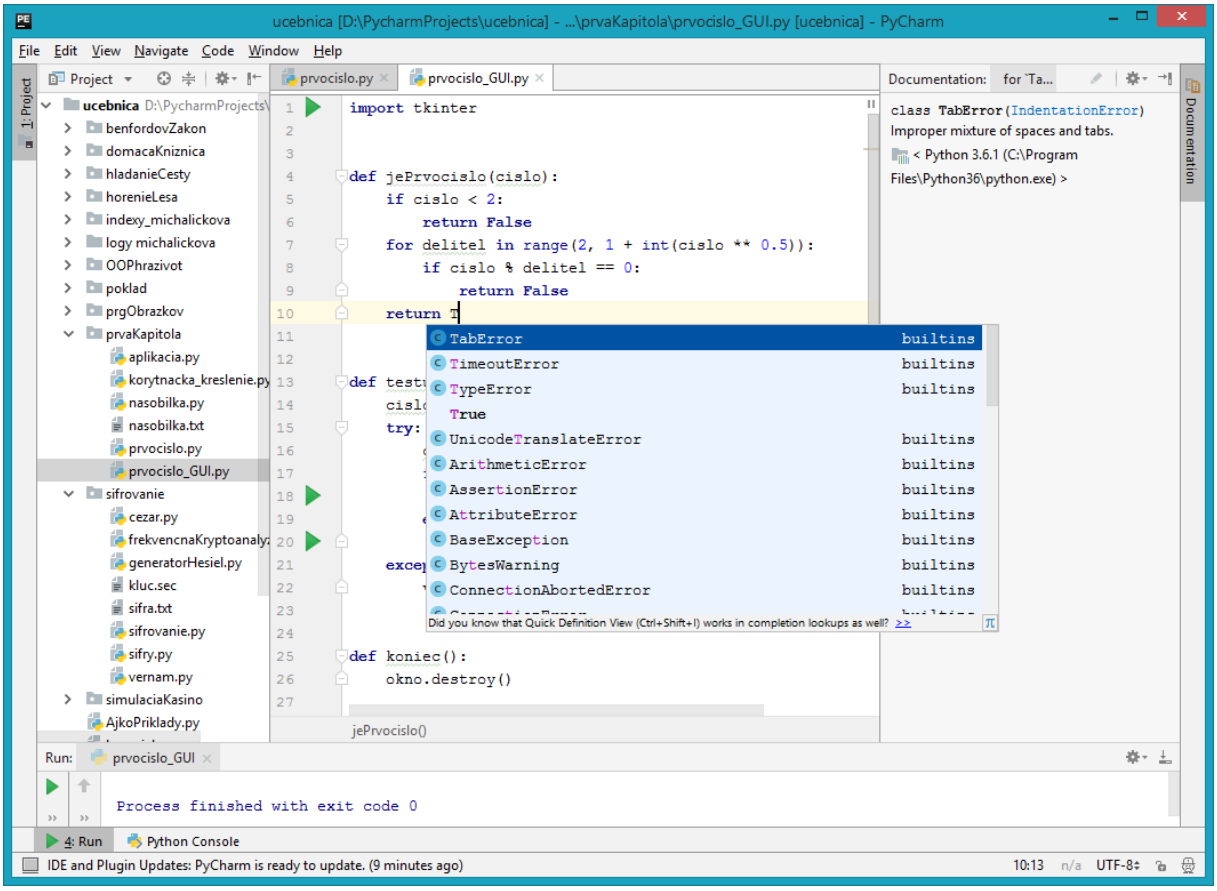
Python (1) je distribuovaný pod otvorenou licenciou (2) kompatibilnou s GPL (GNU General Public License). Samotný jazyk a jednoduché vývojové prostredie si môžeme stiahnuť zo stránky https://www.python.org/downloads/ [20. 5. 2019]. V súčasnosti sú k dispozícii dve hlavné verzie, 2.x a 3.x. Tieto verzie nie sú vzájomne kompatibilné. Dôsledkom je, že programy napísané vo verzii 2.x nebudú fungovať vo verzii 3.x a naopak. V nasledujúcich kapitolách predpokladáme použitie verzie 3.x.

Python si našiel, vďaka svojim vlastnostiam, široké uplatnenie v praxi. Google, Mozilla, NASA, eIBM sú len zlomkom spoločností, ktoré ho aktuálne používajú. Nájdeme ho ako skriptovací jazyk pre iné programy, ako jazyk pre vedecké výpočty, jazyk pre prácu s big data alebo jazyk

bežiaci na pozadí cloudových aplikácií. Môžeme ho teda označiť aj ďalším prívlastkom – moderný.

Vývojové prostredie

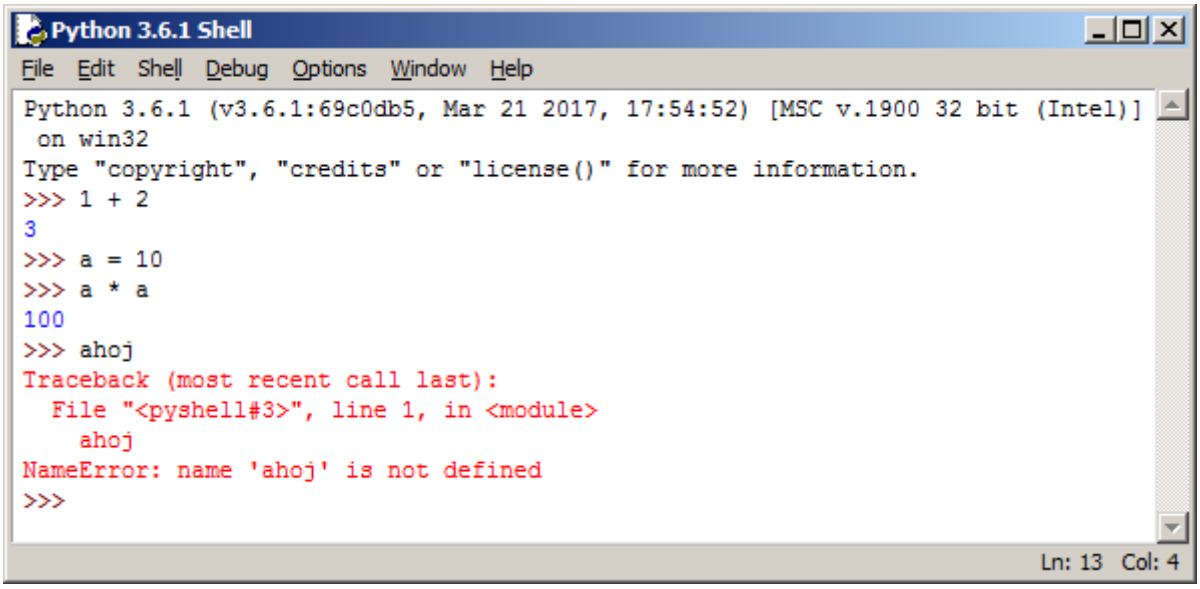
Pri písaní programov môžeme využiť jednoduché prostredie IDLE (Integrated DeveLopment Environment), ktoré je súčasťou základnej inštalácie jazyka. Pre vytváranie rozsiahlejších programov odporúčame využiť pokročilejšie vývojové prostredie (IDE – Integrated Development Environment) (napr. PyCharm Edu (3).



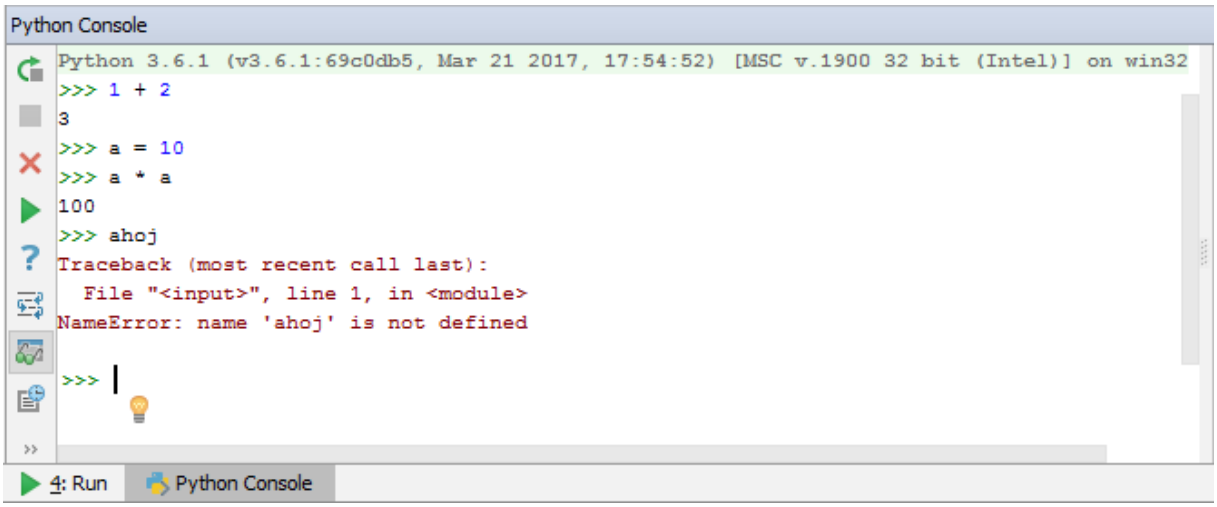
Obrázok 1 Vývojové prostredie PyCharm Edu

Interaktívny režim

V oboch prípadoch (IDLE, PyCharm Edu) máme k dispozícii interaktívny režim (Shell, resp. konzolu jazyka Python). V tomto prostredí môžeme písať príkazy jazyka, ktoré sa automaticky vyhodnocujú a výsledok sa automaticky zobrazí. Prostredie môžeme využiť na riešenie jednoduchých problémov alebo na rýchle testovanie častí programov.



Obrázok 2 Shell - interaktívne prostredie, súčasť základnej inštalácie jazyka Python



Obrázok 3 Konzola jazyka Python, súčasť vývojového prostredia Pycharm Edu

Aritmetické výrazy

V jazyku Python môžeme využiť základné aritmetické operácie:

`x + y`	súčet čísiel x a y
`x - y`	rozdiel čísiel x a y
`x * y`	súčin čísiel x a y
`x / y`	podiel čísiel x a y
`x // y`	celá časť podielu čísla x číslom y
`x % y`	zvyšok po delení čísla x číslom y
`x ** y`	číslo x umocnené na y

Počas výpočtu sa môže stať, že niektorú z hodnôt, ktorú sme vypočítali, potrebujeme použiť viackrát. Riešenie je jednoduché. Python umožňuje túto hodnotu pomenovať pomocou premennej (`nazovPremennej = hodnota`). Premenná je teda meno pre nejakú hodnotu.

```
>>> cena_listka = 3.5
>>> pocet_listkov = 17
```

```
>>> platba = cena_listka * pocet_listkov
>>> platba
59.5
```

Logické výrazy

Logickým výrazom je výraz, ktorého vyhodnotením dostaneme hodnotu pravda (True) alebo nepravda (False). Hodnoty True a False sú hodnoty typu boolean.

Logické výrazy často obsahujú operátory porovnania:

`x < y`	hodnota x je menšia ako hodnota y
`x <= y`	hodnota x je menšia alebo rovná ako hodnota y
`x > y`	hodnota x je väčšia ako hodnota y
`x >= y`	hodnota x je väčšia alebo rovná ako hodnota y
`x == y`	hodnota x je rovná hodnote y
`x != y`	hodnota x je rôzna od hodnoty y

alebo operátor príslušnosti:

`x in y`	hodnota x je prvkom y (napr. znakom reťazca alebo prvkom zoznamu)

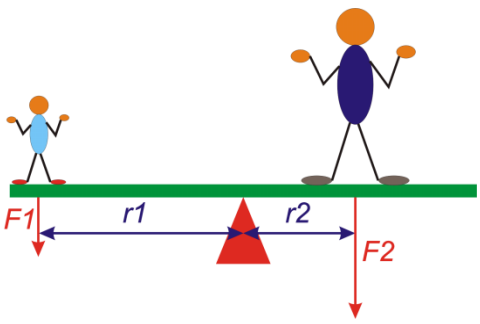
Z logických výrazov môžeme vytvárať zložitejšie logické výrazy pomocou logických operácií `and`, `or` a `not`.

```
>>> 1 < 2
True
>>> 5 > 3 < 8
True
>>> 71 % 7 == 0
False
>>> 'rak' in 'straka'
True
>>> 'Eva' in ['Adam', 'Ivo']
False
>>> heslo = 'Sezam otvor sa'
>>> heslo == 'tajne' or heslo == 'superTajne'
False
```

Úloha 1

Už aj malé deti vedia, že ak sa chcú hojdať na hojdačke, tak ťažšie dieťa si musí sadnúť bližšie ku stredu. Z fyziky vieme, že ak majú byť deti na hojdačke v rovnováhe, musí platiť:

$$F_1 \cdot r_1 = F_2 \cdot r_2$$



Silu páky si uvedomil aj Archimedes a prehlásil: „Dajte mi pevný bod vo vesmíre a pohnem Zemou“.

Pozrime sa na realizáciu podobného pokusu v reálnych podmienkach. Predpokladajme, že máme železnú tyč dlhú 2 m. Kde by sme mali umiestniť onen pevný bod, ak by sme chceli len tiažou svojho tela nadvihnúť osobné auto?

Úloha 2

Pri nákupoch sa často rozhodujeme podľa ceny. Obchodníci by mali pri každom výrobku uvádzať jednotkovú cenu, aby sme si vedeli porovnať výhodnosť rôznych balení. Ak si začneme všímať tieto jednotkové ceny, čoskoro zistíme, že niekedy sú ťažko využiteľné. Niekde obchodník prepočítal cenu na litre a inde na kilogramy. Niekde aj prepočítal cenu na rovnaké jednotky, napr. kusy, ale každý kus má inú veľkosť.

Navrhnite vhodnú jednotku a spôsob prepočítavania pri toaletnom papieri. Niektoré balenia obsahujú viac kotúčov papiera, inde sa predáva len jeden kotúč. Pri niektorých kotúčoch je uvedený počet útržkov, inde zase celková dĺžka v metroch.

Doplňujúca otázka: Ako by ste zohľadnili počet vrstiev papiera?

Reťazec (string), dátová štruktúra zoznam (list)

Reťazec

Reťazec je postupnosť znakov indexovaná od 0 zľava alebo od -1 sprava. Reťazce sú uzatvorené v `"` alebo v `'`. Reťazce sú nemenné, takže výsledkom akejkoľvek operácie s reťazcami je vždy nový reťazec.

```
>>> retazec = ''                                #prázdny reťazec
>>> retazec1 = 'Ahoj'                           #reťazec 'Ahoj'
>>> retazec2 = 'kamoš'                          #reťazec 'kamoš'
>>> pozdrav = retazec1 + ' ' + retazec2         #zreťazenie 'Ahoj kamoš'
>>> pozdrav
'Ahoj kamoš'
>>> pozdrav[5]
'k'
```

Viac o reťazcoch: https://docs.python.org/3.7/library/stdtypes.html#text-sequence-type-str [20. 5. 2019]

Zoznam

Zoznam je dátová štruktúra, ktorá umožňuje uchovávať viacero hodnôt rôzneho typu. Hodnoty sú indexované od 0 zľava alebo od -1 sprava. Zoznamy možno meniť.

```
>>> zoznam = []                                 #prázdny zoznam
>>> dobre_znamky = [1, 2]                       #zoznam [1, 2]
>>> dobre_znamky.append(3)                      #vloženie na koniec [1, 2, 3]
>>> zle_znamky = [4, 5]                         #zoznam [4, 5]
>>> znamky = dobre_znamky + zle_znamky #spojenie [1, 2, 3, 4, 5]
>>> znamky[2]
3
```

Viac o zoznamoch: https://docs.python.org/3.7/tutorial/datastructures.html#more-on-lists [20. 5. 2019]

Výrezy

Z reťazcov aj zo zoznamov vieme vybrať nejakú časť – podreťazce z reťazcov a podzoznamy zo zoznamov. Slúžia nám na to výrezy. Okrem indexovania prvkov od 0 zľava môžeme prvky indexovať aj od -1 sprava.

indexovanie zľava	0	1	2	3	4	5
Reťazec/zoznam	P	y	t	h	o	n
indexovanie sprava	-	-	-	-	-	-

Na výber častí reťazcov/zoznamov používame výrezy. Definícia výrezu môže mať niekoľko tvarov. Vždy však obsahuje jednu alebo dve dvojbodky:

- `[zaciatok:]`
  časť od indexu `zaciatok` až do konca
- `[:koniec]`
  časť od začiatku až do indexu `koniec - 1`
- `[zaciatok:koniec]`
  časť od indexu `zaciatok` až do indexu `koniec - 1`
- `[zaciatok:koniec:krok]`
  časť od indexu `zaciatok` až do indexu `koniec - 1` s krokom `krok`

```
>>> ['P', 'y', 't', 'h', 'o', 'n'][2:]
['t', 'h', 'o', 'n']
>>> ['P', 'y', 't', 'h', 'o', 'n'][:2]
['P', 'y']
>>> 'Python'[2:5]
'tho'
>>> ['P', 'y', 't', 'h', 'o', 'n'][::2]
['P', 't', 'o']
>>> 'Python'[::-1]
'nohtyP'
```

Pretypovanie

Python-u je jedno, aký typ hodnoty premennou pomenujeme. Pri práci s hodnotami však kontroluje, či s hodnotou uvedeného typu možno operáciu vykonať. Preto nám umožňuje hodnoty pretypovať, t.j. z hodnoty jedného typu vytvoriť hodnotu iného typu. Samozrejme, ani pretypovanie nemožno používať ľubovoľne, teda nie každú hodnotu je možné pretypovať na ľubovoľný typ. Na pretypovane používame funkcie: `int()`, `float()`, `str()`, `list()`.

```
>>> str(13)
'13'
>>> float('13')
13.0
>>> int('sedem')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'sedem'
```

```
>>> int(3.14)
3
>>> list('sedem')
['s', 'e', 'd', 'e', 'm']
>>> str(['s', 'e', 'd', 'e', 'm'])
"['s', 'e', 'd', 'e', 'm']"
>>> list(7)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'int' object is not iterable
```

Programy v súboroch

Výpočty a programy zapísané v interaktívnom režime sa po jeho zatvorení stratia. Ak by sme potrebovali výpočet zopakovať, museli by sme všetky programy opakovane napísať. Pre opakované použitie programu je potrebné uložiť ho do súboru. V IDLE, resp. v prostredí PyCharm Edu, si otvoríme nový súbor. Program, ktorý napíšeme a uložíme do súboru, bude prístupný aj po ukončení a opätovnom spustení prostredia.

Všetky vstupy z klávesnice (z konzoly) sú typu reťazec. Ak očakávame hodnotu iného typu ako reťazec, musíme vstupnú hodnotu pretypovať:

```
vstup = input('Zadaj vstup: ')
cele_cislo = int(vstup)
realne_cislo = float(vstup)
```

Rovnako aj pri výstupe na konzolu sa očakáva hodnota typu reťazec. Pri výstupe to Python robí automaticky. Problémom je, ak výstup skladáme z textu a vypočítanej hodnoty. Pri spájaní reťazcov musia byť operandy typu reťazec. Ak sa chceme vyhnúť spájaniu reťazcov, môžeme použiť jeden zo spôsobov pre formátovanie reťazcov: f-string .

```
print('Vypocitana hodnota: ' + str(hodnota))
#alebo
print(f'Vypocitana hodnota: {hodnota}')
```

Poznámka na okraj

V tomto predmete budeme používať spôsob f-string. Pri tomto spôsobe sa reťazcové operácie vyhodnocujú rýchlo a zápis je vo všeobecnosti najkratší.

Úloha 3

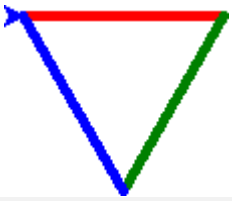
Vytvorte program, ktorý pre zadanú dĺžku tyče a zadané hmotnosti auta a človeka zistí, na ktorom mieste je potrebné podložiť tyč tak, aby človek len tiažou vlastného tela nadvihol auto.

Úloha 4

Vytvorte program, ktorý pre zadané parametre toaletného papiera (cena balenia, počet kotúčov, množstvo papiera v kotúči) zistí, aká je jednotková cena pre 100 ks útržkov.

Korytnačia grafika

Jeden z možných výstupov našich programov môže byť obrázok. Na kreslenie môžeme využiť korytnačiu grafiku (4) a modul turtle.



```
# trojuholnik.py

import turtle                           #[1]
tabula = turtle.Screen()                #[2]
pero = turtle.Turtle()                  #[3]

pero.pensize(5)                         #[4]

pero.color('red')                       #[5]
pero.forward(100)                       #[6]
pero.right(120)                         #[7]

pero.color('green')
pero.forward(100)
pero.right(120)

pero.color('blue')
pero.forward(100)
pero.right(120)

tabula.mainloop()                       #[8]
```

[1] Importujeme modul turtle.

Pri tomto type importu sa všetky volania funkcií alebo tried z modulu turtle začínajú prefixom `turtle`.

Niekde sa môžeme stretnúť aj s importom spôsobom: `form turtle import *`. Pri tomto type importu nemusíme používať prefix turtle a zápisy budú kratšie (napr. `tabula = Screen()`). Tento spôsob, najmä v prípade importu viacerých modulov, neodporúčame používať. V takomto prípade sa narušuje menný priestor. Problém by nastal, ak by rôzne moduly obsahovali rovnako pomenované funkcie alebo ak by sme si definovali funkciu s rovnakým názvom ako má funkcia v module.

Ak by názov importovaného modulu bol príliš dlhý alebo pre nás nezrozumiteľný, vieme si pri importe nastaviť lokálne meno pre importovaný modul: `import turtle as t`. V tomto prípade všetky volania funkcií alebo tried z modulu turtle začínajú prefixom `t`.

Po importe modulu turtle je automaticky k dispozícii kresliace pero a kresliaca plocha. Začať kresliť je teda možné aj spôsobom:

```
import turtle
turtle.pensize(5)
turtle.color('red')
turtle.forward(100)
...
turtle.mainloop()
```

Tento spôsob však nepredpokladá, že použijeme viac kresliacich pier (napr. každé s iným nastavením).

[2] Vytvoríme kresliace plátno a nazveme ho tabula.
[3] Vytvoríme kresliace pero. Toto pero je automaticky umiestnené do kresliacej plochy.
[4] Nastavíme hrúbku kresliaceho pera.
[5] Nastavíme farbu kresliaceho pera.
[6] Vykreslíme čiaru dĺžky 100. Použije sa aktuálne nastavenie kresliaceho pera.
[7] Otočíme kresliace pero o 120° vpravo. Pôvodný smer kresliaceho pera je smerom vpravo.
[8] Spustíme nekonečnú slučku udalostí. V tomto príklade nám slučka zabezpečí, že okno s kresbou sa automaticky nezatvorí, ale čaká, kým ho zatvoríme my.

Viac o module turtle sa dozvieme napr. tu: https://docs.python.org/3.7/library/turtle.html [20. 5. 2019].

Úloha 5

Využitím modulu turtle vykreslite nasledovný obrázok:

Formátovanie kódu

Zvláštnosťou jazyka Python je to, že na určenie blokov príkazov (úrovne vnorenia) nepoužíva žiadne zátvorky alebo kľúčové slová. Blok príkazov je určený samotným odsadením príkazov v bloku. Každú úroveň odsadenia realizujeme pomocou štyroch medzier. Dôsledkom je, že pri vytváraní kódu sa menej napíšeme a zdrojové kódy sú prehľadnejšie. Všimnime si to v nasledujúcich príkladoch.

Príkazy cyklu

Pri riešení problémov sa neraz dostaneme do situácie, keď potrebujeme časť programu niekoľkokrát zopakovať. Môžeme ju viacnásobne nakopírovať, ale tento prístup má niekoľko nevýhod. Ak sme v pôvodnom kóde spravili chybu, kopírovaním ju znásobíme. Kopírovaním rýchlo narastá množstvo kódu a celý program sa tým stáva neprehľadným. A nakoniec sú situácie, keď v čase písania programu ešte nevieme, koľko krát budeme potrebovať túto časť zopakovať. Tieto situácie sa dajú elegantne vyriešiť pomocou cyklov.

Príkaz cyklu for

Príkaz cyklu `for` využijeme v situácii, ak potrebujeme časť programu niekoľko krát zopakovať alebo ak potrebujeme vykonať nejakú činnosť pre každý prvok nejakej iterovateľnej štruktúry (napr. zoznam, reťazec).

```
for pocitadlo in range(100):
    print(f'{pocitadlo}. Na hodine budem dávať pozor a nebudem vyrušovať.')
```

```
0. Na hodine budem dávať pozor a nebudem vyrušovať.
1. Na hodine budem dávať pozor a nebudem vyrušovať.
2. Na hodine budem dávať pozor a nebudem vyrušovať.
...
99. Na hodine budem dávať pozor a nebudem vyrušovať.
```

Vykonaním uvedeného programu sa sto krát vypíše text „Na hodine budem dávať pozor a nebudem vyrušovať.“ aj s poradovým číslom. Cyklus `for` funguje tak, že postupne prechádza prvkami nejakej štruktúry, resp. postupnosti. V tomto prípade (`range(100)`) je to postupnosť čísiel 0 .. 99. Premenná `pocitadlo` pri každom opakovaní cyklu postupne nadobúda hodnoty prvkov z danej postupnosti.

`range()` je funkcia, ktorá pri každom zavolaní vráti jednu z hodnôt postupnosti určenú jej parametrami. Funkcii `range()` môžeme okrem poslednej hodnoty určiť aj prvú hodnotu a krok.

```
for pocitadlo in range(1, 100):
    print(f'{pocitadlo}. Na hodine budem dávať pozor a nebudem vyrušovať.')
```

```
1. Na hodine budem dávať pozor a nebudem vyrušovať.
2. Na hodine budem dávať pozor a nebudem vyrušovať.
...
```

Pozor, v tomto prípade trest zopakujeme len 99 krát (1 .. 99).

Ak potrebujeme určiť krok, s akým generovať postupnosť, určíme to tretím parametrom:

```
print('Násobky 7:')
for cislo in range(7, 71, 7):
    print(cislo)
```

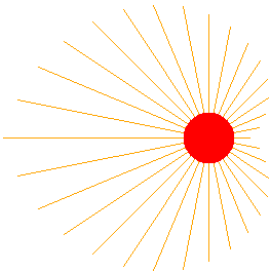
```
Násobky 7:
7
14
21
28
35
42
49
56
63
70
```

Namiesto funkcie `range()` môžeme použiť aj nejakú iterovateľnú štruktúru (štruktúra, cez prvky ktorej možno prechádzať). Zo štruktúr, ktoré už poznáme, sú to reťazce a zoznamy.

```
>>> for znak in 'ahoj':
        print(znak)
a
h
o
j
>>> for cislo in [112, 155, 158, 150]:
        print(cislo)
112
155
158
150
```

Úloha 6

Vytvorte program, ktorý vykreslí nasledovné zubaté slniečko.



Príkaz cyklu while

Príkaz cyklu `while` využijeme v situácii, keď potrebujeme zopakovať nejakú časť programu, ale počet opakovaní nedokážeme určiť. Počet opakovaní je určený platnosťou nejakej podmienky.

```
# heslo.py
heslo = input('Zadaj heslo: ')
while heslo != 'abrakadabra':
    heslo = input('Zadaj heslo: ')

print('Heslo je spravne')
```

Úloha 7

Karol našiel zaujímavú vlastnosť pre najväčšieho spoločného deliteľa (NSD) dvoch prirodzených čísiel.

NSD(a, a) = a
NSD(a, b) = NSD(b, a)
NSD(a, b) = NSD(a, b - a) <=> b > 0

Na základe tohto poznatku vytvoril nasledovný program:

```
# nsd.py
while a != b:
    if a > b:
        a = a - b
    else:
        b = b - a
nsd = a
```

Potom si ale uvedomil, že ak rozdiel čísiel `a` a `b` bude príliš veľký, tak postupné odpočítavanie bude zbytočne zdĺhavé. Napadlo mu, že postupné odpočítavanie by mohol zrýchliť tak, že ho nahradí zvyškom po delení. Naprogramujte jeho vylepšenú verziu.

Vlastné funkcie

Pri riešení rozsiahlejších problémov je výhodné rozdeliť veľký problém na niekoľko menších problémov. Každý z menších problémov vyriešime samostatne a správnosť riešenia otestujeme. Z riešení menších problémov spätne vyskladáme riešenie pôvodného, veľkého problému.

Riešenie menších problémov môžeme umiestniť do samostatných blokov – funkcií. Funkcia pomenováva nejakú časť programu a zavolaním tohto mena v programe sa táto časť programu vykoná. Funkcie majú aj ďalšie výhody. Ich kód je znovu použiteľný a sprehľadňujú celý program. Kód každej funkcie môžeme testovať samostatne. Hľadanie chýb je tak zvyčajne jednoduchšie.

Funkcia `pozdrav()` po jej zavolaní vypíše pozdrav na konzolu.

```
def pozdrav():
    print('Ahoj')

pozdrav() #Ahoj
#po vykonaní časti programu vo funkcii vykonávanie programu
pokračuje ďalej za volaním funkcie
```

Vykonávanie kódu funkcie môžeme ovplyvniť parametrami funkcie. Pomocou parametrov môžeme funkcii poslať konkrétnu hodnotu (alebo hodnoty), s ktorými má pracovať. Zoznam parametrov uvádzame v hlavičke funkcie pri jej definícii.

```
def pozdrav_menom(meno):
    print(f'Ahoj {meno}')

pozdrav_menom('Karol') #Ahoj Karol
```

vrátiť nejakú výslednú hodnotu. S touto hodnotou môžeme ďalej, mimo funkcie, pracovať. pracovať.

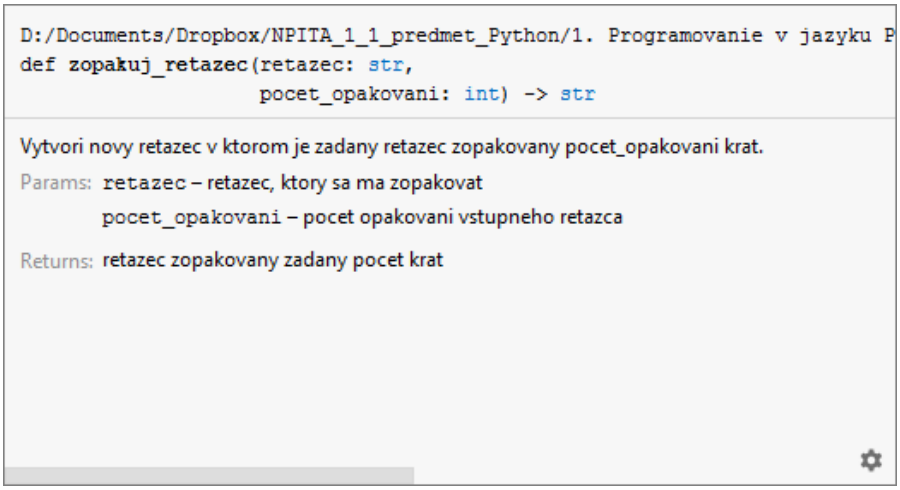
```
def zopakuj_retazec(retazec, pocet_opakovani):
    return retazec * pocet_opakovani

print(zopakuj_retazec('ahoj', 5)) #ahojahojahojahojahoj
```

Príkaz `return` v tele funkcie znamená: vráť hodnotu a ukonči vykonávanie tela funkcie. Príkazov `return` môže byť v tele funkcie viac. Vždy sa však vykoná len jeden z nich, ten, ktorý sa vykoná ako prvý.

Každý dobrý programátor vie, že kód sa píše raz a číta veľakrát. Aby sme aj po čase dobre rozumeli svojmu programu alebo aby mu rozumeli aj iní, je dobrým zvykom kód komentovať. Zvlášť funkcie, ktoré sa často používajú. Dokumentačný reťazec začína a končí trojicou apostrofov a uvádza sa hneď za hlavičkou funkcie. Popisuje čo funkcia robí, aké má vstupy a výstupy.

```
# dokumentacia_funkcie.py
def zopakuj_retazec(retazec, pocet_opakovani):
    ''' Vytvori novy retazec v ktorom je zadany retazec
    zopakovany pocet_opakovani krat.

    :param retazec: retazec, ktory sa ma zopakovat
    :type retazec: str
    :param pocet_opakovani: pocet opakovani vstupneho retazca
    :type pocet_opakovani: int
    :return: retazec zopakovany zadany pocet krat
    :rtype: str
    '''
    return retazec * pocet_opakovani
```

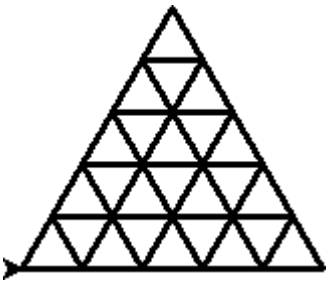
Moderné vývojové prostredia uľahčujú programátorom prácu a na požiadanie zobrazia dokumentáciu k vybranej funkcii. Aj bez toho, aby sme skúmali kód funkcie, vieme presne čo funkcia robí, aké požaduje vstupy a aké sú jej výstupy.



Obrázok 4 Zobrazenie dokumentácie funkcie

Úloha 8

Vytvorte program, ktorý vykreslí nasledovnú pyramídu.



Úloha 9

Vytvorte pomôcku na matematiku pre riešenie kombinatorických úloh. Definujte funkcie pre výpočet počtu variácií, permutácií a kombinácií. Využiť môžete nasledovné vzťahy:

$$V_k(n) = \frac{n!}{(n-k)!} \qquad V'_k(n) = n^k$$

$$P(n) = n! \qquad P_{k_1, k_2, \cdots k_r}(n) = \frac{n!}{k_1! \cdot k_2! \cdot \cdots \cdot k_r!}$$

$$C_k(n) = \frac{n!}{k!(n-k)!} \qquad C'_k(n) = \frac{(n+k-1)!}{(n-1)! \cdot k!}$$

Vlastné moduly

Pri písaní rozsiahlejších programov narážame na dva problémy.

1. Ak by sme celý program písali do jedného súboru, naše programy by boli veľmi rozsiahle, neprehľadné a ťažko by sme v nich hľadali chyby.
2. Ak by sme potrebovali niektoré z našich funkcií použiť v inom programe, museli by sme ich prepísať do nového programu.

Obidva problémy vieme vyriešiť pomocou vlastných modulov. Definície funkcií presunieme do samostatného súboru (alebo súborov) - modulu. Ak potrebujeme v hlavnom programe použiť funkcie z nejakého modulu, pripojíme ho pomocou príkazu `import`.

```
# funcie.py
'''Modul pre zdravenie
'''


def pozdrav():
    ''' Pozdraví používateľa výpisom do konzoly
    '''
    print('Ahoj')
```

```
# hlavny.py
import funkcie

funkcie.pozdrav()   # Ahoj
```

Všimnime si, že súbor, do ktorého sme presunuli naše funkcie, sa volá `funkcie.py`, ale modul sa volá `funkcie`. Ak vytvárame vlastné moduly, dokumentácia je zvlášť dôležitá. Funkcie z modulu používame často bez toho, aby sme videli/skúmali ich definíciu.

Úloha 10

Vytvorte modul `kombinatorika` a vložte doň funkcie pre riešenie kombinatorických úloh z predchádzajúcej úlohy. Doplňte k funkciám a k modulu dokumentačné reťazce. V hlavnom programe otestujte ich použitie.

Podmienený príkaz IF

Programy nemusia byť len sekvenčné, t.j. zoznam príkazov napísaný a vykonávaný jeden za druhým. Program môžeme vetviť a rozhodnúť, ktorou vetvou má vykonávanie programu pokračovať. Príkaz `if` slúži na podmienené vetvenie.

```
if logicky vyraz 1:
    # časť programu, ktorá sa vykoná, ak logický výraz 1 platí
elif logicky vyraz 2:
    # časť programu, ktorá sa vykoná, ak logický výraz 2 platí
…
else:
    # časť programu, ktorá sa vykoná, ak žiaden z predchádzajúcich
výrazov neplatí
```

Príkaz `if` môže mať viac častí `elif`. Časti `elif` a `else` sú nepovinné. Ak sa vykonávanie programu realizuje jednou z vetiev, ostatné vetvy sa už nevykonávajú.

Úloha 11

Vytvorte funkciu BMI (Body Mass Index), ktorá pre zadané hodnoty hmotnosť (kg), výška (m) vráti, do ktorej kategórie človek patrí.

Využiť môžete nasledujúcu klasifikáciu podľa Svetovej zdravotnickej orgasnizácie (5) pre dospelú populáciu:

- bmi < 18,5 podváha,
- 18,5 <= bmi < 25 normálna hmotnosť,
- 25 <= bmi < 30 mierna nadváha,
- 30 <= bmi < 35 obezita 1. triedy,
- 30 <= bmi < 40 obezita 2. triedy,
- 40 <= bmi obezita 3. triedy.

Úloha 12

Pig Latin (6) je jazyková slovná hra pre anglicky hovoriacich. Zjednodušené pravidlá pre preklad do Pig Latin sú nasledovné:

- Ak slovo začína spoluhláskou, presunie sa táto spoluhláska na koniec slova a pridá sa koncovka „ay“. (banana -> ananabay)
- Ak slovo začína samohláskou, pridá sa na koniec slova koncovka „way“. (eat->eatway)

Vytvorte funkciu `slovo_do_pig_latin()`, ktorá vráti preklad zadaného slova do Pig Latin. Pre jednoduchosť uvažujme len písmená malej anglickej abecedy.

Úloha 13

Vytvorte funkciu `jednotkova_cena_toaletného_papiera()`, ktorá pre zadané hodnoty (pozri úlohu 2) vráti cenu za 1000 ks útržkov zaokrúhlenú na dve desatinné miesta. Premyslite si, ako vyriešiť výpočet, ak zadávame počet metrov alebo počet útržkov. Do funkcie vložte dokumentačný reťazec.

Chyby v programoch

Pri programovaní robíme chyby. Nikto nie je neomylný. V programoch môžeme rozpoznať tri druhy chýb:

- **syntaktická chyba** – napr. preklep v kóde, chýbajúca zátvorka alebo úvodzovka,
  na túto chybu a jej lokalizáciu nás upozorní parser (súčasť interpretera jazyka Python, ktorá kontroluje, či v zdrojovom kóde nie je syntaktická chyba) a vieme ju jednoducho opraviť,
- **logická chyba** – program sa dá spustiť, počíta, ale nie to, čo by sme chceli,
  nedostávame očakávané alebo správne výsledky,
  tieto chyby musíme najskôr odhaliť, lokalizovať a potom opraviť, na to nám slúži testovanie a ladenie programov,
- **behová chyba** – objaví sa až počas behu programu, napr. delenie nulou, požiadavka na neexistujúci súbor, nedostatok pamäte,
  tieto chyby možno do istej miery predvídať a čiastočne im predchádzať, problémom je, že ak sa nám to nepodarí, program skončí s chybou.

Syntaktické chyby

Úloha 14

Nájdite a opravte chyby v nasledujúcej časti programu. Program by mal pokračovať len po správne zadanom hesle. Používateľ má len jeden pokus.

```
# vstup_na_heslo.py
heslo = input('Zadaj heslo: '
if heslo <> 'sezam otvor sa'
    prit('Heslo je nesprávne. Program skončil.')
    exit()

# tu bude tajný výpočet
```

Logické chyby

Odhaliť logickú chybu môžeme niekoľkými spôsobmi:

- Pripravíme si testovacie dáta a skontrolujeme, či pre ne dostaneme správne vysledky.
- Analyzujeme program a skontrolujeme, či každá časť robí to, čo má robiť a tak, ako to má robiť.
- Spúšťame program po častiach a kontrolujeme každú časť, uistíme sa, že premenné nadobúdajú požadované hodnoty, že cykly bežia požadovaný počet krát, že funkcie vracajú požadované hodnoty a pod. Jednou z možností, ako kontrolu realizovať, sú kontrolné výpisy. Druhou možnosťou je spustiť ladiaci (debug) režim a kontrolovať krok za krokom, či sa program správa tak, ako to očakávame.

Úloha 15

Funkcia `je_prvocislo()` by mala pre zadané celé číslo zistiť, či je prvočíslo alebo nie. Overte, či sme funkciu definovali správne.

```
#prvocislo.py
def je_prvocislo(cislo):
    '''Zistí, či zadané číslo je prvočíslo

    :param cislo: testované číslo
    :type cislo: int
    :return: odpoveď na otázku, či číslo je prvočíslo
    :rtype: bool
    '''
    for delitel in range(2, int(cislo ** 0.5)):
        if cislo // delitel == 0:
            return False
    return True
```

- Pripravte si testovacie dáta a overte, či dostanete správne výsledky.
- Vytvorte si kontrolné výpisy, aby ste zistili, či sa funkcia správa tak ako očakávame.
- Spustite ladiaci režim a kontrolujte správnosť behu funkcie po krokoch.

Behové chyby

Behová chyba nastane, ak sa vykonávanie programu dostane do výnimočného stavu, v ktorom nemôže ďalej normálne pokračovať. Väčšinou nás na to interpreter upozorní výpisom informácie, kde a aký problém nastal, vyhodí výnimku a vykonávanie programu skončí. Ak pri spustení nasledovného programu:

```
citatel = float(input('Zadaj čitateľa: '))
menovatel = float(input('Zadaj menovateľa: '))
print(citatel / menovatel)
```

zadáme menovateľ rovný 0, dostaneme nasledovný výsledok:

```
Zadaj čitateľa: 1
Zadaj menovateľa: 0
Traceback (most recent call last):
  File C:/skripty/chyba.py", line 3, in <module>
    print(citatel / menovatel)
ZeroDivisionError: float division by zero

Process finished with exit code 1
```

Ak by sme chceli takejto chybe zabrániť, môžeme pred samotným delením otestovať, či môžeme deliť menovateľom.

```
citatel = float(input('Zadaj čitateľa: '))
menovatel = float(input('Zadaj menovateľa: '))
if menovatel != 0:
    print(citatel / menovatel)
else:
    print('Zadal si 0 v menovateli. Nulou sa deliť nedá!')
```

Ak teraz zadáme 0, program neskončí s chybou, ale s upozornením, že používateľ zadal 0, a tou sa deliť nedá.

Čo ale, ak náš program použije používateľ takto:

```
Zadaj čitateľa: jeden
Traceback (most recent call last):
  File "C:/skripty/chyba.py", line 1, in <module>
    citatel = float(input("Zadaj citatela: "))
ValueError: could not convert string to float: 'jeden'

Process finished with exit code 1
```

Opäť by sme mohli doprogramovať test, aby sme overili, či používateľ zadal číslo. Ale mali by sme istotu, že sme obsiahli všetky možné chyby? Asi nie.

Python-e môžeme takéto situácie riešiť trochu inak. Namiesto množstva testov (if if if ...) či akciu možno zrealizovať, zrealizujme akciu a zistime, či pri realizácii nenastala nejaká chyba (výnimočný stav). Ak áno, ošetrime ju. Ešte predtým si všimnime, aké typy chýb spôsobilo nesprávne použitie nášho programu.aké typy chýb spôsobilo nesprávne použitie nášho programu.

```
#podiel1_kontrola_chyb.py
try:
    citatel = float(input('Zadaj čitateľa: '))
    menovatel = float(input('Zadaj menovateľa: '))
    print(citatel / menovatel)
except ValueError:
    print('Nezadal si číslo!')
except ZeroDivisionError:
    print('Nulou sa deliť nedá!')
except:
    print('Neočakávaná chyba!')
```

```
Zadaj citatela: 1
Zadaj menovatela: 0
Nulou sa delit neda.

Process finished with exit code 0
```

alebo

```
Zadaj citatela: sedem
Nezadal si cislo.

Process finished with exit code 0
```

Problematickú časť programu umiestnime do bloku `try`. Python ju bude vykonávať o niečo „opatrnejšie“ ako zvyčajne. Ak pri jej vykonávaní nastane nejaká chyba, Python ukončí vykonávanie tejto časti a bude pokračovať vykonávaním časti programu v tom bloku `except`, v ktorom sa konkrétna výnimka odchytáva. Po vykonaní časti programu v bloku `except` (za predpokladu, že tu nenastane nejaká chyba) pokračuje vo vykonávaní programu za posledným blokom `except` (resp. za celým blokom `try`).

Ak pri vykonávaní nejakej časti programu môže nastať viacero chýb (viac typov výnimiek), musíme ich odchytávať v poradí od najkonkrétnejších po najvšeobecnejšie. Hierarchiu výnimiek nájdeme na https://docs.python.org/3.7/library/exceptions.html#exception-hierarchy [20. 5. 2019]. Posledný blok `except` (bez uvedenia konkrétnej výnimky) sa vykoná, ak nastala výnimka rôzna od všetkých predchádzajúcich.

Ak by uvedená časť programu bola súčasťou rozsiahlejšieho programu a podiel by sme potrebovali vypočítať, docielime to napr. takto:

```
#podiel2_kontrola_chyb.py
while True:
    try:
        citatel = float(input('Zadaj čitateľa: '))
        menovatel = float(input('Zadaj menovateľa: '))
        print(citatel / menovatel)
        break
    except ValueError:
        print('Nezadal si číslo!')
    except ZeroDivisionError:
        print('Nulou sa deliť nedá!')
    except:
```

```
        print('Neočakávaná chyba!')

print('Podiel sme úspešne vypočítali.')
```

Všeobecný tvar riadiacej štruktúry `try` je nasledovný:

```
try:
    #problematická časť programu

except výnimka1:
    #časť programu, ktorá sa vykoná, ak nastane výnimka 1

except výnimka2:
    # časť programu, ktorá sa vykoná, ak nastane výnimka 2

...
except:
    # časť programu, ktorá sa vykoná, ak nastane iná výnimka mimo
predchádzajúcich
else:
    # časť programu, ktorá sa vykoná, ak nenastane žiadna výnimka
finally:
    # časť programu, ktorá sa vykoná vždy
```

Úloha 16

V zozname `znamky` sú známky žiaka. Vytvorte program, ktorý vypočíta priemer známok. Ošetrite situácie, v ktorých dôjde počas výpočtu k chybe.

V istých situáciách sa aj náš program môže dostať do problematickej situácie, v ktorej nemôže vykonať to, čo by mal vykonať. Mohli by sme situáciu mlčky prejsť, ale riskujeme, že naše zamlčanie spôsobí neskôr ďalšiu chybu. A jej dôvod by sa už hľadal ťažšie. Filozofiou Python-u je na takúto situáciu upozorniť čo najskôr – vyhodíme výnimku.

Predpokladajme, že sme definovali funkciu na výpočet obsahu kruhu:

```
import math
def obsah_kruhu(polomer):
    return math.pi * polomer ** 2
```

Čo sa stane, ak zadáme záporný polomer?

```
print(obsah_kruhu(-10))
```

```
314.1592653589793
```

Naša funkcia (alebo Python) samozrejme nevie, že kruh so záporným polomerom neexistuje a výraz vyhodnotí. Dostávame hodnotu, s ktorou môžeme ďalej pracovať. Problém je však v tom, že je to chyba, ktorú sme mlčky ignorovali. V tomto prípade by mala naša funkcia „protestovať“ a vyhodiť výnimku.

```
#obsah_kruhu.py
import math
def obsah_kruhu(polomer):
    if polomer < 0:
```

```
        raise ValueError('Zadaný záporný polomer kruhu!')
    return math.pi * polomer ** 2

print(obsah_kruhu(-10))
```

```
Traceback (most recent call last):
  File " C:/skripty/chyba.py ", line 7, in <module>
    print(obsah_kruhu(-10))
  File " C:/skripty/chyba.py ", line 4, in obsah_kruhu
    raise ValueError("Zadany zaporny polomer kruhu.")
ValueError: Zadaný záporný polomer kruhu.

Process finished with exit code 1
```

Pri vyhadzovaní výnimiek zvoľme taký typ, ktorý zodpovedá chybe, kvôli ktorej výnimku vyhadzujeme. Zoznam výnimiek aj so stručným popisom nájdeme na https://docs.python.org/3.7/library/exceptions.html#concrete-exceptions [20. 5. 2019].

Pri rozsiahlejších programoch je bežnou praxou to, že jedna časť nášho programu výnimky vyhadzuje (lebo nie je schopná požadovanú akciu vykonať) a druhá ich odchytáva a ošetruje (aby program predčasne neskončil).

Úloha 17

Učiteľ informatiky si známky svojich žiakov registruje v zozname takéhoto typu:

```
[['Karol'], ['Imrich', 2, 3], ['Katka', 1, 1]]
```

Pred vysvedčením by si rád uľahčil prácu tým, že si vytvorí program, ktorý mu vygeneruje výpis známok na vysvedčení. Časť problému už vyriešil.

```
# znamky_na_vysvedceni.py
def vysledna_znamka(znamky):
    '''Pre zadaný zoznam známok žiaka vráti výslednú známku žiaka.

    :param znamky: zoznam známok žiaka
    :type znamky: list of int
    :return: výsledná známka
    :rtype: int

    :raise ValueError: ak žiak nemá žiadne známky
    '''

    try:
        return round(sum(znamky) / len(znamky))
    except ZeroDivisionError:
        raise ValueError('Nie sú zadané žiadne známky!')
```

Vytvorte funkciu `vypis_vysvedcenie()`, ktorá pre zadaný zoznam vráti reťazec, v ktorom má každý žiak uvedenú výslednú známku alebo text „neklasifikovaný“ (ak žiak nemá žiadne známky).

Práca so súbormi

Dáta, resp. hodnoty, s ktorými sme doposiaľ pracovali, existovali len dočasne. Prístup k nim nám zabezpečovali ich mená – premenné. Keď vykonávanie programu skončilo, prístup k hodnotám sme stratili. Ak potrebujeme hodnoty uchovať aj po skončení programu, môžeme ich uložiť do súboru a v prípade potreby ich zo súboru prečítať.

Zápis dát do súboru

Zápis dát do súboru môžeme v Python-e realizovať rôznymi spôsobmi. Najjednoduchšie je využiť konštrukciu with:

```
# subor_zapis.py
with open('subor.dat', 'w', encoding='utf-8') as f:                    #[1]
    f.write('Dáta v súbore')                                           #[2]
    f.write('\n')                                                      #[3]
    f.write('Toto bude v novom riadku')
```

subor.dat:

```
Data v subore:
Toto bude v novom riadku
```

[1] Otvoríme súbor `subor.dat` v režime zápisu `w` a pri zápise použijeme kódovanie znakov `utf-8`. K súboru budeme pristupovať cez premennú `f`.

[2] Do súboru sme zapísali reťazec „`Dáta v súbore`“. Do súboru môžeme zapisovať len reťazce. Ak potrebujeme zapísať hodnotu iného typu ako reťazec, musíme ju pretypovať na reťazec.

[3] Ak chceme zapisovať dáta do nového riadku, zapíšeme do súboru znak konca riadku `\n`.

Konštrukcia with má výhodu aj v tom, že po jej ukončení sa súbor automaticky uzatvorí. A to aj v prípade výnimky (chyby) pri práci so súborom.

Úloha 18

Pani učiteľka chce naučiť svojich žiakov malú násobilku. Dochádza jej však tvorivosť pri vymýšlaní prikladov na násobenie. Pomôžte jej a vytvorte program, ktorý po spustení zapíše do súboru 10 náhodných príkladov na násobenie.

Pomôcka: Na vymýšľanie číselných hodnôt môžeme využiť generátor pseudonáhodných čísiel v module random:

```
import random
print(random.randint(1, 10)) #náhodné celé číslo z intervalu 1..10
```

Čítanie dát zo súboru

Na výber máme niekoľko možností ako dáta zo súboru prečítať. Aj v tomto prípade je najjednoduchšie použiť blok with:

```
# subor_citanie.py
with open('subor.dat', 'r', encoding='utf-8') as f:                    #[1]
    for riadok in f.readlines():                                       #[2]
        print(riadok)                                                  #[3]
```

[1] Otvoríme súbor `subor.dat` v režime čítania `r` a pri dekódovaní prečítaného obsahu použijeme kódovanie znakov `utf-8`. K súboru budeme pristupovať cez premennú `f`.
[2] `f.readlines()` prečíta všetky riadky zo súboru a vráti ich v zozname. Každý riadok, vrátane znaku konca riadka, je prečítaný ako samostatný reťazec. Cyklus for tak prechádza postupne všetkými riadkami.
[3] Do konzoly vypíšeme obsah jednotlivých riadkov súboru.

Úloha 19

Vytvorte program, ktorý prečíta a vyrieši príklady uložené v súbore z predchádzajúcej úlohy.

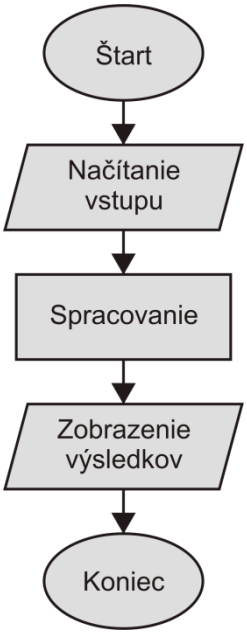
Grafické rozhranie

Grafické rozhranie predstavuje prostredníka medzi behom nášho programu a používateľom nášho programu. Vďaka takémuto prostredníkovi sa programy ľahšie ovládajú a sú aj vzhľadovo príťažlivejšie. Python ponúka niekoľko spôsobov ako grafické rozhranie vytvoriť. Štandardom sa stalo používanie prvkom modulu `tkinter` (7).

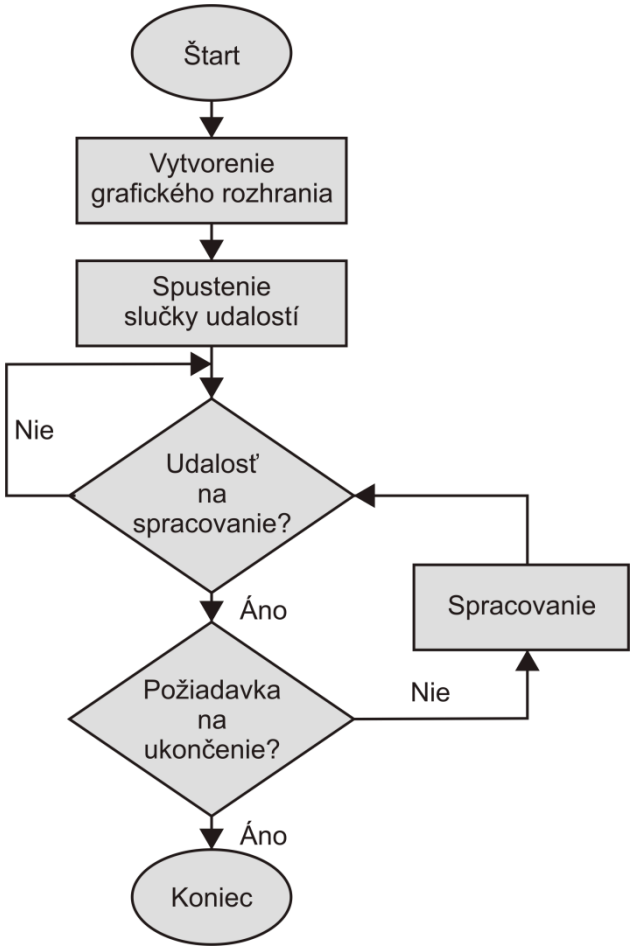
Pomocou modulu tkinter vieme vytvoriť grafické okno, do okna umiestniť prvky potrebné pre prácu s programom (vstupné textové polia, tlačidlá, grafickú plochu atď.) a prepojiť tieto grafické prvky s jednotlivými časťami nášho programu.

Naše doterajšie programy s používateľom komunikovali cez textové rozhranie. Ak počas behu programu nastala chyba, Python vyhodil výnimku a informáciu o chybe zobrazil používateľovi (v okne SHELL, resp. v okne Run). Ak k programu vytvoríme grafické rozhranie, musíme zabezpečiť, aby sa prípadné chyby zobrazili v okne grafického rozhrania, pretože s textovým rozhraním používateľ nepracuje.

Konzolové programy fungovali v sekvencii základných blokov (pozri obrázok nižšie). Grafické programy pracujú odlišne (pozri obrázok nižšie). Po spustení sa vytvorí grafické rozhranie a spustí sa nekonečná slučka udalostí. Zachytáva udalosti a spracováva ich. Jednou z udalostí je ukončenie programu. Vtedy sa slučka preruší, grafické rozhranie sa zatvorí a program skončí.



Obrázok 5 Schéma konzolového programu



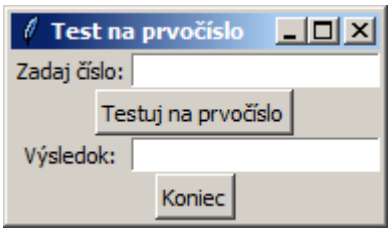
Obrázok 6 Schéma programu s grafickým rozhraním

Ukážeme si to na príklade programu, ktorý testuje, či zadané číslo je prvočíslo. Samotný program vyzerá nasledovne:

```
# prvocislo_textove_rozhranie.py
def je_prvocislo(cislo):
    '''Zistí, či zadané číslo je prvočíslo

    :param cislo: testované číslo
    :type cislo: int
    :return: odpoveď na otázku, či číslo je prvočíslo
    :rtype: bool
    '''
    if cislo < 2:
        return False
    for delitel in range(2, int(cislo ** 0.5) + 1):
        # print(delitel)
        if cislo % delitel == 0:
            return False
    return True


cislo = input('Zadaj číslo: ')
try:
    cislo = int(cislo)
    if je_prvocislo(cislo):
        print(f'Číslo {cislo} je prvočíslo.')
```

```
    else:
        print(f'Číslo {cislo} je nie je prvočíslo.')
except ValueError:
    print('Nezadal si celé číslo.')
```

Naším cieľom je vytvoriť takéto grafické rozhranie:



Obrázok 7 Grafické rozhranie programu na test prvočísiel

```
#prvocislo graficke rozhranie.py
import tkinter                                                  #[1]
def je_prvocislo(cislo):                                        #[2]
    '''Zistí, či zadané číslo je prvočíslo

    :param cislo: testované číslo
    :type cislo: int
    :return: odpoveď na otázku, či číslo je prvočíslo
    :rtype: bool
    '''
    if cislo < 2:
        return False
    for delitel in range(2, int(cislo ** 0.5) + 1):
        # print(delitel)
        if cislo % delitel == 0:
            return False
    return True


def testuj():                                                   #[3]
    cislo = vstup.get()                                         #[4]
    try:
        cislo = int(cislo)
        if je_prvocislo(cislo):
            vystup.set(f'Číslo {cislo} je prvočíslo.')          #[5]
        else:
            vystup.set(f'Číslo {cislo} nie je prvočíslo')       #[6]
    except ValueError:
        vystup.set('Nezadal si celé číslo.')                    #[7]


def koniec():                                                   #[8]
    okno.destroy()                                              #[9]


okno = tkinter.Tk()                                             #[10]
okno.title('Test na prvočíslo')                                 #[11]

tkinter.Label(okno, text='Zadaj číslo:').grid(row=0, column=0)  #[12]
vstup = tkinter.StringVar()                                     #[13]
tkinter.Entry(okno, textvariable=vstup).grid(row=0, column=1)   #[14]
```

```
tkinter.Button(okno, text='Testuj na prvočíslo', command=testuj).\
    grid(row=1, column=0, columnspan=2)                              #[15]

tkinter.Label(okno, text='Výsledok:').grid(row=2, column=0)          #[16]
vystup = tkinter.StringVar()                                         #[17]
tkinter.Entry(okno, textvariable=vystup).grid(row=2, column=1)       #[18]

tkinter.Button(okno, text='Koniec', command=koniec).\
    grid(row=3, column=0, columnspan=2)                              #[19]

okno.mainloop()                                                      #[20]
```

[1] Importujeme modul tkinter.

[2] Pôvodnú funkciu `je_prvocislo()` využijeme na samotný test prvočísla.

[3] Funkcia `testuj()` nahradila pôvodnú časť programu, ktorá čítala vstupy a zobrazovala výstupy. Táto funkcia sa zavolá po stlačení tlačidla „Testuj na prvočíslo“ [15]

[4] Vstup prečítame zo vstupného textového políčka [14] cez premennú vstup [13], ktorá je s textovým políčkom prepojená.

[5] Ak zadané číslo je prvočíslo, informáciu zapíšeme do výstupného textového políčka [18] cez premennú výstup [17], ktorá je s textovým políčkom prepojená.

[6] Ak zadané číslo nie je prvočíslo, informáciu zapíšeme do výstupného textového políčka [18] cez premennú výstup [17], ktorá je s textovým políčkom prepojená.

[7] Ak nebolo zadané celé číslo (pri pokuse o pretypovanie Python vyhodil vynimku), informáciu zapíšeme do výstupného textového políčka [18] cez premennú výstup [17], ktorá je s textovým políčkom prepojená.

[8] Funkcia `koniec()` sa zavolá po stlačení tlačidla „Koniec“ [19].

[9] Príkazom `destroy()` zatvoríme okno aplikácie [10].

[10] Vytvoríme okno aplikácie.

[11] Oknu pridáme titulok „Test na prvočíslo“.

[12] Vytvoríme popisok vstupného okna [14]. Predstavme si neviditeľnú mriežku, do ktorej budeme umiestňovať prvky okna. Popis v tejto mriežke umiestníme do riadku 0 a stĺpca 0.

[13] Vytvoríme špeciálnu premennú `vstup`. Cez túto premennú budeme pristupovať k obsahu vstupného textového políčka [14].

[14] Vytvoríme vstupné textové políčko. Všimnime si, že sme ho prepojili s premennou `vstup` [13] a umiestnili do riadku 0 a stĺpca 1.

[15] Vytvoríme tlačidlo „Testuj“. Po stlačení sa zavolá funkcia `testuj()` [3]. Tlačidlo sme umiestnili do riadku 1. Keďže v tomto riadku máme len jeden prvok, stĺpce sme spojili, a tak v tomto riadku je iba stĺpec 0.

[16] Vytvoríme popisok výstupného okna [18]. Popis umiestníme do riadku 2 a stĺpca 0.

[17] Vytvoríme špeciálnu premennú `vystup`. Cez túto premennú budeme pristupovať k obsahu výstupného textového políčka [18].
[18] Vytvoríme výstupné textové políčko. Všimnime si, že sme ho prepojili s premennou `vystup` [17] a umiestnili do riadku 2 a stĺpca 1.
[19] Vytvoríme tlačidlo „Koniec“. Po stlačení sa zavolá funkcia `koniec()` [8]. Tlačidlo sme umiestnili do riadku 3. Keďže v tomto riadku máme len jeden prvok, stĺpce sme spojili, a tak v tomto riadku je iba stĺpec 0.
[20] Spustili sme nekonečnú slučku pre obsluhu udalostí. Slučka neustále beží a čaká na udalosť (napr. na stlačenie tlačidla), aby vyvolala príslušnú akciu.

Prvky (widgets), ktoré môžeme do okna aplikácie umiestniť, môžu byť rôznych typov.

- Text alebo ScrolledText – viacriadkový text
- Canvas – grafická plocha, do ktorej možno vykresľovať rôzne geometrické útvary
- Checkbutton – zaškrtávacie okienko pre výber viacerých možností
- Listbox – pre zobrazenie zoznamu položiek, z ktorých môže používateľ niektoré označiť
- Radiobutton – prepínač pre výber jednej z niekoľkých možností
- ...

Ďalšie možnosti použitia grafického rozhrania tkinter nájdeme napr. na: https://www.tutorialspoint.com/python/python_gui_programming.htm [20. 5. 2019].

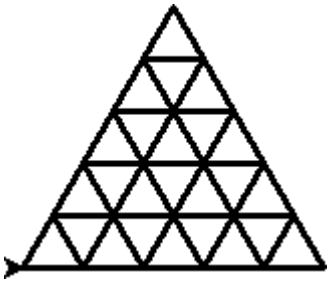
Stratégie riešenia problémov

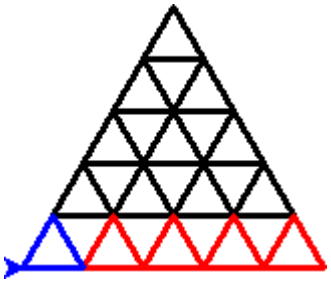
V predchádzajúcej časti sme riešili množstvo problémov. Aj keď to boli rôzne problémy, často sme používali rovnaké postupy pri hľadaní ich riešení. V tejto kapitole si zhrnieme, aké stratégie môžeme použiť pri hľadaní riešenia problémov. Pozrime sa na niektoré z nich bližšie.

Dekompozícia

Dekompozícia problému na podproblémy je jednou zo základných stratégií riešenia problémov pri programovaní. Táto stratégia spočíva v tom, že veľký problém rozdelíme na niekoľko menších, ľahšie riešiteľných. Ak sú aj tieto problémy ťažko riešiteľné, pokračujeme v delení dovtedy, kým nie sme schopní čiastkové problémy vyriešiť. Ak sa nám podarí vyriešiť čiastkové problémy, zložením ich riešení získame riešenie pôvodného problému.

Túto stratégiu sme mohli použiť pri kreslení pyramídy z trojuholníkov. Pyramída pozostáva z radov a jednotlivé rady z trojuholníkov. Vyriešili sme kreslenie trojuholníka. Potom sme naprogramovali kreslenie radu z trojuholníkov a nakoniec kreslenie celej pyramídy z radov.



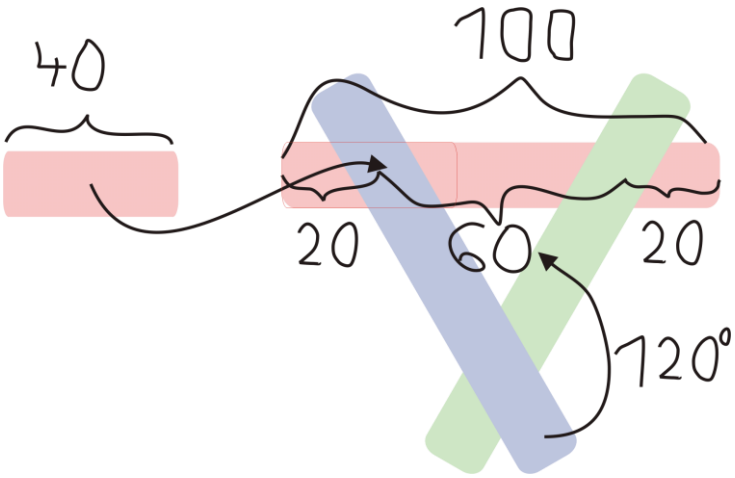


Nakresli si obrázok

Nakresliť si schému alebo obrázok popisujúci problém je jednou z najčastejších stratégií. Obrázok nám pomôže lepšie pochopiť problém, uvedomiť si na prvý pohľad skryté vzťahy.

V jednej z predchádzajúcich úloh sme mali vykresliť nasledovný útvar:

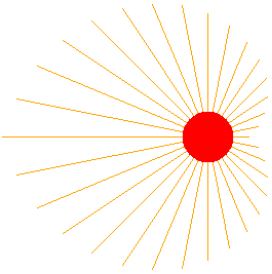
Ak si obrázok nakreslíme a doplníme do neho vzdialenosti a uhly, pomôže nám to jednoduchšie zostaviť výsledný program.



Sprav si zoznam

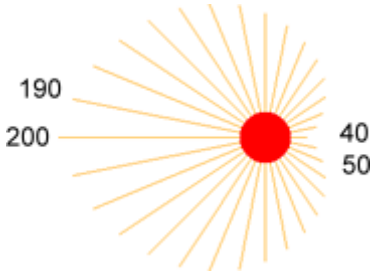
Táto stratégia nám pomôže, ak si potrebujeme nájsť nejaký systém, urobiť vo veciach poriadok.

V jednej z predchádzajúcich úloh sme mali vykresliť takéto slniečko:



Problémom je, že nevieme, ako postupne meniť dĺžky lúčov slniečka. Vidíme, že v jednej časti sa lúče postupne zväčšujú, a v druhej, ak dodržíme rovnaký smer, zase postupne zmenšujú. Prvé, čo nám napadne, je, že začneme lúče vykresľovať buď od najmenšieho alebo od najväčšieho. Vyberme si prvú možnosť a skúsme objaviť závislosť dĺžky lúča v poradí od najmenšieho po najväčší – horná časť. Potom pokračujme od najväčšieho po najmenší – spodná časť.

od najmenšieho	
poradie lúča	dĺžka
1.	40
2.	50
3.	60
...	
16.	190
range(40, 200, 10)	



od najväčšieho	
poradie lúča	dĺžka
1.	200
2.	190
3.	180
...	
16.	50
Range(200, 40, -10)	

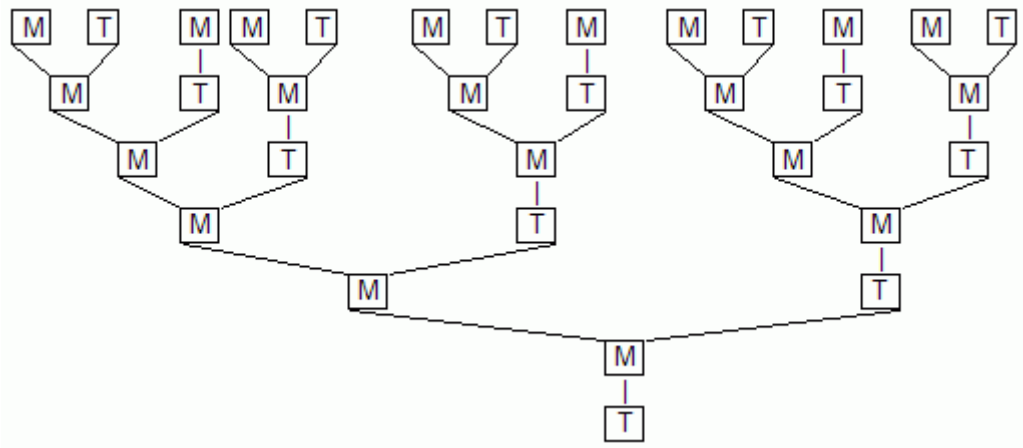
Naprogramovať nájdené riešenie už nie je komplikované.

Nájdi vzor

Táto stratégia spočíva v tom, že sa snažíme nájsť nejaké vzory opakujúce sa v zadanom probléme, a tak problém vyriešiť. Vzorom môže byť časť obrázka opakujúca sa v celom obrázku, pravidlo aplikujúce sa na jednotlivé časti problému apod.

Uvedenú stratégiu môžeme použiť pri riešení nasledovného problému[1]. V spoločenstve včiel sa vyskytuje včelia matka, robotnice a trúdy. Kým včelia matka a robotnice majú dvoch rodičov (včeliu matku a trúda), trúd má len jedného rodiča (včeliu matku). Koľko prapra..prarodičov má trúd?

Ak si nakreslíme rodostrom trúda, dostaneme nasledovný obrázok:



Spočítajme počet predkov v jednotlivých generáciách:

číslo generácie	počet prapra...prarodičov
0 (počet rodičov)	1
1 (počet prarodičov)	2
2 (počet praprarodičov)	3
3 (počet prapraprarodičov)	5
4 (počet prapraprapraprarodičov)	8
5 (počet prapraprapraprapraprarodičov)	13
6 (počet prapraprapraprapraprarodičov)	?

[1] Problém sme prevzali zo súťaže PALMA junior (8). V školskom roku 2005/2006 to bola 4. úloha v 2. kole (https://di.ics.upjs.sk/palmaj/riesenia_komentare.htm [20. 5. 2019]).

Pri pozornejšom preskúmaní počtu predkov (hľadáme vzor, ako sa tieto počty správajú) v jednotlivých generáciách zistíme, že počet predkov v generácii n je rovný súčtu predkov v generáciách n-1 a n-2. Výnimku tvoria generácia predkov 0 a 1. Tam sú počty dané na 1 a 2.

Ak si pozorne všimneme jednotlivé stratégie a ich použitie, zistíme, že vo viacerých prípadoch ich vzájomne kombinujeme. Pri riešení problémov tak často používame viaceré stratégie súčasne. To, čo majú spoločné, je analýza problému, v ktorej identifikujeme vzťahy, a podstatné informácie. Dobrá analýza problému je základom, na ktorom môžeme budovať jeho riešenie.

2. Ako hrať v kasíne a neprehrať

Kľúčové slová

simulácia, náhodné číslo, slovník, hazardná hra, pravdepodobnosť, rozdelenie pravdepodobnosti, systém, model, stavová premenná, pseudonáhodné číslo

Čo sa naučíme a čo si precvičíme

- analyzovať prírodovedné a spoločenské systémy,
- navrhovať modely reálnych systémov,
- overovať hypotézy simulovaním správania sa systému pomocou modelu,
- použiť generátor pseudonáhodných čísiel a pracovať s náhodou,
- pochopiť princíp fungovania generátorov pseudonáhodných čísiel,
- vysvetliť vlastnosti dátovej štruktúry slovník,
- vhodne používať dátovú štruktúru slovník,
- kriticky prijímať a overovať tvrdenia,

Problémová situácia

Predpokladajme takúto stratégiu pri hre v kasíne. Budeme hrať ruletu.

Stavíme 50 € na červenú. Ak padne červená, berieme výhru (100 €) a odchádzame domov so ziskom 50 €. Ak nepadne červená, stavíme v ďalšej hre na červenú toľko, aby sme v prípade výhry mali celkový zisk 50 €. Stavíme teda 100 €. Stávky sú 50 € a 100 €, výhra je 200 €. Ak vyhráme, odchádzame domov so ziskom 50 €. Ak nie, pokračujeme rovnako až do prvej výhry. Červená raz určite padne a zisk 50 € máme každý deň zaručený. Funguje takáto stratégia?

Hľadajme riešenie

V reálnom živote sa musíme veľakrát rozhodovať. Všetci dúfame, že naše rozhodnutia budú správne.

Sú situácie, keď rozhodnutie je pomerne jednoduché, napr. „môžem prejsť cez cestu alebo nie?“ V tomto prípade sa na základe analýzy okolitej situácie (sú na ceste autá?, približujú sa alebo sa vzďaľujú? sú blízko alebo ďaleko“ ...) a na základe našich predchádzajúcich skúseností vieme rozhodnúť, či cez cestu prejdeme alebo nie.

Sú však situácie, keď rozhodnúť sa nie je jednoduché. Buď nemáme dostatok vedomostí, času alebo prostriedkov na to, aby sme vedeli situáciu vyhodnotiť a správne sa rozhodnúť. V takýchto prípadoch môžeme využiť simulácie. V simulácii môžeme niektoré deje urýchliť, spomaliť alebo neriskovať reálny dopad nesprávneho rozhodnutia.

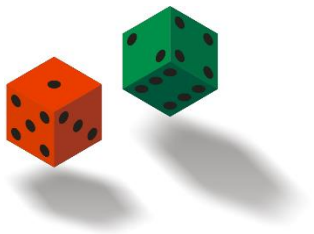
Výkladový text

Časť reálneho sveta, ktorý chceme skúmať nazveme **systém**. Zjednodušené zobrazenie systému, v ktorom zanedbáme jeho nepodstatné časti nazveme **modelom**. Niekedy je reálny systém natoľko komplikovaný, že v modeli zanedbávame aj časti, ktoré na správanie sa systému majú vplyv. Vplyv týchto častí by však mal byť čo najmenší. Imitáciu správania sa

systému v našom modeli nazveme **simuláciou**. Systém sa môže nachádzať v nejakom stave. Tento stav popíšeme pomocou **stavovej premennej**. (8)

Úloha 1

Hádžeme dvoma kockami. Aký súčet bude padať najčastejšie?



Táto úloha nie je náročná a zrejme by sme vedeli odpovedať aj bez toho, aby sme nejakú simuláciu spúšťali. Aj napriek tomu realizujme **simuláciu**, aby sme si ukázali na čo si dávať pozor. **Systém**, ktorý chceme simulovať je hod dvoma kockami. To, aké čísla na kockách padnú je popisom nášho systému. Tento stav si uchováme v **stavovej premennej**. Hod kockou/kockami budeme **modelovať** pomocou generátora pseudonáhodných čísiel. Pre náš model nie je podstatné akú farbu kocky majú, či sú rovnako veľké alebo nie, či kocka ostane stáť na hrane a pod. Toto sú nepodstatné časti a v modeli ich môžeme zanedbať.

Výkladový text

Pseudonáhodné číslo je číslo vygenerované generátorom pseudonáhodných čísiel. Generátory sú navrhnuté tak, že čísla ktoré generujú, sa zdajú byť náhodné. Nie je viditeľná žiadna závislosť medzi nimi. Na generovanie čísiel používa generátor špeciálne navrhnutú funkciu.

Simulujme najskôr hod jednou kockou. Využime generátor pseudonáhodných čísiel. Nájdeme ho v module `random`. Po rýchlom pohľade do dokumentácie modulu `random` nájdeme funkciu `random.randrange(start, stop)`.

Otázka

V reálnom prípade by sme kockami hodili súčasne. V našom modeli „hodíme“ najskôr jednou kockou a potom druhou. Môžeme v modeli simulovať hod dvoma kockami takto bez toho, aby sme dostali skreslené výsledky o počte jednotlivých súčtov?

Túto funkciu by sme mohli využiť na generovanie hodu jednou kockou: `random.randrange(1, 6)`. Spustime ju niekoľkokrát v konzole. Dostali sme nasledovné výsledky:

```
>>> import random
>>> random.randrange(1, 6)
2
>>> random.randrange(1, 6)
5
>>> random.randrange(1, 6)
2
>>> random.randrange(1, 6)
1
...
```

Funkcia naozaj generuje náhodné čísla (teda aspoň na prvý pohľad). Niektoré nám síce nepadli, ale to nemusí byť problém, keďže padajú náhodne. Rovnaká situácia môže nastať aj pri skutočnej kocke. Aby sme sa uistili, že náš simulátor hodov kocky je dobrý (že tento model

zodpovedá reálnemu systému), spustime ho viackrát. V tomto bode by sme však mali vedieť, aký výsledok budeme akceptovať. Počty jednotlivých čísiel, ktoré padnú na kocke by mali byť približne rovnaké. Čim viac hodov zrealizujeme, tým menšie relatívne rozdiely medzi počtami by mali byť. Simulujme napr. 1000 hodov.

```
import random
pocty_cisel = [0, 0, 0, 0, 0, 0, 0]                                    #[1]
for hod in range(1000):                                                #[2]
    padnute_cislo = random.randrange(1, 6)
    pocty_cisel[padnute_cislo] += 1                                    #[3]
print(pocty_cisel)
```

[1] Počty padnutých čísiel si ukladáme do zoznamu. Keďže položky zoznamu sú číslované od 0, zoznam má sedem prvkov. Na mieste 0 by mala ostať 0 aj po simulácii.
[2] Opakujem 1000-krát hod kockou.
[3] Podľa toho aké číslo padlo zvýšime príslušný počet. Zvýšime ho o 1.

Výsledok pre 1000 hodov je nasledujúci:

```
[0, 181, 200, 202, 188, 229, 0]
```

Aj keď pracujeme s náhodou a predpokladáme nejaké odchýlky zarazí nás, že číslo 6 nepadlo ani raz. Pri hľadaní chyby zistíme, že funkcia `randrange(1, 6)` generuje čísla z intervalu $\langle 1, 6)$. Číslo 6 teda nebude vygenerované nikdy.

Túto chybu sme odhalili pomerne ľahko. Predpokladáme, že mnohí z vás ju už odhalili tiež. Priznávame, že sme ju spravili zámerne, aby sme ukázali, ako ľahko sa dá odhaliť pri teste simulácie hodu kockou. Omnoho ťažšie by to bolo, ak by sme v tejto úlohe neriešili model hodu kockou, ale začali riešiť model súčtu hodu dvoch kociek. Tu by nám dokonca hrozilo, že podobnú chybu by sme si ani nevšimli.

Poznámka na okraj

Vždy je lepšie veľký problém rozdeliť na menšie podproblémy, nájsť a otestovať ich riešenia a z týchto riešení spätne vyskladať riešenie pôvodného, veľkého problému. Takejto stratégii riešenia problémov hovoríme **rozklad problému na podproblémy**. Výhodou tohto prístupu je aj fakt, že správnosť riešenia môžeme testovať postupne. Najskôr otestujeme riešenia podproblémov. Lokalizovať a opraviť chybu v riešení podproblému je jednoduchšie ako hľadať chybu v návrhu riešenia celého problému

Oprava je pomerne jednoduchá. Buď upravíme hodnotu parametra `stop` (`stop + 1`) alebo použijeme funkciu `random.randint(start, stop)`, ktorá pri generovaní zahŕňa aj hodnotu stop. Nezabudnime aj tento simulátor otestovať.

Model hodu dvoma kockami môžeme realizovať nasledovne:

```
import random
pocetnosti_suctov = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]            #[1]
for hod in range(1000):                                                #[2]
    padnute_cislo1 = random.randint(1, 6)                              #[3]
```

```
    padnute_cislo2 = random.randint(1, 6)                          #[4]
    pocetnosti_suctov [padnute_cislo1 + padnute_cislo2] += 1       #[5]
print(pocetnosti_suctov)
```

[1] Pri hode dvoma kockami môžu padnúť súčty od 2 do 12. Kvôli číslovaniu položiek zoznamov od 0 bude mať náš zoznam 13 položiek. Pozície 0 a 1 by mali ostať nezmenené.
[2] Hod dvoma kockami opakujeme 1000 krát.
[3] Hod prvou kockou.
[4] Hod druhou kockou.
[5] Podľa súčtu hodnôt upravíme príslušnú početnosť. Zvýšime ju o 1.

Pre 1000 náhodných hodov dvoma kockami sme dostaneme nasledujúce výsledky:

```
[0, 0, 24, 43, 82, 94, 137, 187, 135, 124, 89, 58, 27]
```

Z výsledkov môžeme predpokladať, že súčet 7 padá najčastejšie.

Otázka

Postačuje 1000 hodov na to, aby sme mohli rozhodnúť, ktorý súčet padá najčastejšie? Simulujte väčší počet hodov a porovnajte výsledky.

Poznámka na okraj

V riešení príkladu sme zoznam inicializovali priamo jeho vytvorením. Pri rozsiahlejších zoznamoch by bol tento prístup ťažko použiteľný. Python ponúka aj jednoduchší spôsob ako inicializovať zoznam s rovnakými hodnotami. Krok [1] by sme mohli šikovnejšie zapísať nasledovne:

```
pocetnosti_suctov = [0] * 13
```

V niektorých prípadoch potrebujeme vytvoriť zoznam, pričom jeho počiatočné hodnoty nie sú rovnaké. Aj v tomto prípade nám Python ponúka úsporný zápis Napr.:

```
cisla = [i for i in range (1, 6)]       #[1, 2, 3, 4, 5]
```

alebo

```
mocniny = [i*i for i in range (1, 6)]   #[1, 4, 9, 16, 25]
```

Takémuto zápisu hovoríme generátorová notácia zoznamu (List Comprehension).

Návrhy na vylepšenie:

- Pre záznam súčtu hodu kociek použite namiesto zoznamu slovník.
- Zdôvodnite matematicky, že súčet 7 bude padať najčastejšie.

Výkladový text

Slovník (**dictionary**) je neusporiadaná kolekcia dvojíc: kľúč: hodnota. Napr.: **"auto": "car"** alebo **7: "nepárne"**. Dvojicu kľúč: hodnota musíme do slovníka vložiť naraz. Hodnotu môžeme kedykoľvek zmeniť.

Slovník vytvoríme spôsobom:

```
prazdny_slovnik = {}                                    #[1]
```

alebo:

```
slovnik = {'auto': 'carr'}                              #[2]
```

K hodnotám v slovníku pristupujeme cez ich kľúč.

```
slovnik['auto '] = 'car'                                #[3]
slovnik[1] = 'one'                                      #[4]
print(slovnik['auto'])                                  #[5]
```

Nechcenú dvojicu kľúč: hodnota môžeme zo slovníka vybrať:

```
preloz = slovnik.pop('auto')                            #[6]
```

Slovníkom vieme prechádzať a postupne pristupovať k dvojiciam kľúč-hodnota v ňom:

```
for kluc in slovnik:
    print(kluc, slovnik[kluc])                          #[7]
```

[1] Vytvorili sme prázdny slovník.

[2] Vytvorili sme slovník s jednou položkou. Hodnota je chybná. V nasledujúcom kroku ju opravíme.

[3] Ak priradíme hodnotu k existujúcemu kľúču nahradíme pôvodnú hodnotu novou.

[4] Do slovníka sme vložili novú dvojicu.

[5] K hodnote v slovníku pristupujeme cez jej kľúč.

[6] Vybrali sme hodnotu priradenú ku kľúču. S hodnotou môžeme ďalej pracovať. V slovníku však už nie je.

[7] Cyklom `for` prechádzame cez jednotlivé kľúče uložené v slovníku. Premenná `kluc` postupne nadobúda hodnoty všetkých kľúčov slovníka.

Viac o slovníkoch nájdeme v dokumentácii:
https://docs.python.org/3.6/tutorial/datastructures.html#dictionaries [20. 5. 2019]

Hod dvoma kockami – riešenie využitím slovníka

Ak si pozrieme výsledok predchádzajúceho programu zistíme, že v zozname sme niektoré pozície obsadili zbytočne (pozície 0 a 1). Navyše aj výpis bol trochu neprehľadný. Ručne sme odčítali, ktorý súčet padal najčastejšie. Obidva problémy vieme vyriešiť pomocou slovníka.

Kľúče v slovníku budú predstavovať jednotlivé súčty. Hodnoty prislúchajúce ku kľúčom budú reprezentovať početnosť koľko krát daný súčet padol.

```
import random
pocetnosti_suctov = {}                                          #[1]
for i in range(2, 13):                                          #[2]
    pocetnosti_suctov[i] = 0
for hod in range(1000):
    padnute_cislo1 = random.randint(1, 6)
    padnute_cislo2 = random.randint(1, 6)
    pocetnosti_suctov [padnute_cislo1 + padnute_cislo2] += 1    #[3]
for kluc in pocetnosti_suctov:                                  #[4]
    print(f'{kluc}: {pocetnosti_suctov[kluc]}')
```

[1] Vytvorili sme prázdny slovník.
[2] Do slovníka sme vložili kľúče 2 .. 12. Hodnoty prislúchajúce k kľúčom sme nastavili na 0.
[3] Podľa toho aký súčet čísiel na kockách padol sme zvýšili príslušnú hodnotu v slovníku.
[4] Cyklom `for` sme prešli cez jednotlivé kľúče v slovníku. Pre každý kľúč sme vypísali aj hodnotu k nemu prislúchajúcu.

Pre 1000 hodov dvoma kockami sme dostali nasledovný výsledok:

```
2 : 23
3 : 60
4 : 68
5 : 104
6 : 141
7 : 161
8 : 136
9 : 132
10 : 84
11 : 57
12 : 34
```

Poznámka na okraj

V riešení príkladu sme slovník inicializovali cez cyklus. Python ponúka aj jednoduchší spôsob ako inicializovať hodnoty v slovníku. Krok [2] by sme mohli šikovnejšie zapísať nasledovne:

```
pocetnosti_suctov = {kluc:0 for kluc in range (2, 13)}
```

Takémuto zápisu hovoríme generátorová notácia slovníka (Dictionary Comprehension). Podobne vieme inicializovať aj zoznamy.

Hod dvoma kockami – matematické zdôvodnenie

V úvode sme si povedali, že simulácie používame najmä v prípadoch, keď nemáme dostatok vedomostí alebo prostriedkov na to, aby sme sa vedeli správne rozhodnúť. V tomto prípade však výsledok simulácie vieme aj zdôvodniť (presnejšie povedané, k rovnakému výsledku sa vieme dopracovať aj bez simulácie). Vytvorme si tabuľku, v ktorej uvedieme všetky variácie ako môžu dve kocky padnúť. Pri každej dvojici uveďme súčet čísiel. Potom nám stačí spočítať, ktorý súčet je najčastejší. My sme na riešenie využili tabuľkový kalkulátor.

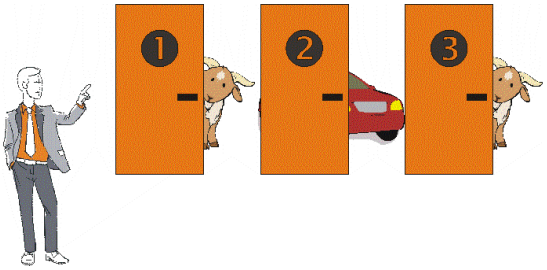
kocka1/kocka2	**1**	**2**	**3**	**4**	**5**	**6**
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

súčet	**početnosť**
2	1
3	2
4	3
5	4
6	5
7	**6**
8	5
9	4
10	3
11	2
12	1

Aj týmto spôsobom vieme nájsť odpoveď na otázku, ktorý súčet bude padať najčastejšie.

Úloha 2

V zábavnej súťaži hrá súťažiaci o auto. Hra prebieha nasledovne (10):



V miestnosti sú troje dverí. Za jednými z dverí je auto. Za zvyšnými dvoma dverami je koza. Súťažiaci nevie nič o tom, čo je za ktorými dverami. Vie len to, že je tam jedno auto a dve kozy. Súťažiaci si vyberie jedny z dverí. Moderátor sa rozhodne pomôcť súťažiacemu a zo zvyšných dvoch dverí mu otvorí tie, kde je koza. Následne vyzve súťažiaceho, či chce zmeniť svoje rozhodnutie. Súťažiaci môže ponechať pôvodné rozhodnutie alebo si vybrať zostávajúce dvere. Mal by súťažiaci zmeniť svoj výber? Vytvorte simuláciu hry a overte, či je výhodnejšie rozhodnutie zmeniť alebo nie.

Problém troch dverí – matematické zdôvodnenie

Ak model systému, ktorý ste navrhli a v ktorom ste simuláciu zrealizovali bol správny, zistili ste že:

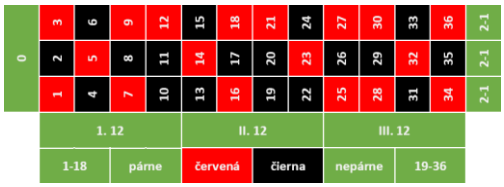
- Ak súťažiaci rozhodnutie nezmení, vyhrá auto približne v 1/3 prípadov
- Ak súťažiaci rozhodnutie zmení, vyhrá auto približne v 2/3 prípadov.

Tento výsledok mnohých z nás prekvapí. Pokúsme sa výsledok zdôvodniť aj matematicky.

- Ak rozhodnutie súťažiaci nezmení, pravdepodobnosť že vyberie dvere s autom je 1/3. Jedny dvere z troch sú tie správne. Skutočnosť, že moderátor otvoril niektoré z dverí nijako neovplyvnila naše počiatočné rozhodnutie.
- Ak rozhodnutie súťažiaci zmení, môžu nastať dve rôzne situácie:
    - Ak súťažiaci pôvodne vybral dvere s autom (1/3 prípadov), zmení svoje rozhodnutie na dvere s kozou.
    - Ak súťažiaci pôvodne vybral dvere s kozou (2/3 prípadov), zmení svoje rozhodnutie na dvere s autom.

    Záver: v 2/3 prípadov si súťažiaci nakoniec zvolí dvere s autom.

Úloha 3



Vráťme sa k stratégii ako hrať v kasíne a neprehrať. Simulujte niekoľko dní a overte, či takáto stratégia funguje alebo nie.

Pomôcka: navrhnite funkciu `jedenDen()`, ktorá bude simulovať priebeh hry v jeden deň. Funkcia môže vrátiť dvojicu: celková suma stávok, výhra. Simulujte niekoľko dní a spočítajte celkový zisk.

Otázky

Ako sa zvyšujú stávky ak v rulete nepadá červená?

Aké čísla môžu v rulete padnúť?

Ktoré sú červené čísla?

Návrhy na vylepšenie:

- Stanovte hodnotu pre maximálny vklad, napr. 1000 €. Simulujte niekoľko dní (napr. rok) hry v kasíne. Aká je ročná bilancia?
- Ukážte matematicky, či takáto stratégia je výhodná alebo nevýhodná.

    Pomôcka: Uvažujte aké sú rôzne scenáre ako môže prebiehať hra počas jedného dňa. Aké sú pravdepodobnosti jednotlivých scenárov. Aká je výhra, resp. prehra pri jednotlivých scenároch.

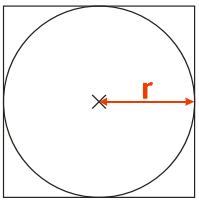
Čo sme sa naučili

- systém je časť reálneho sveta,
- model je zjednodušené zobrazenie systému pomocou pravidiel správania sa,

- simulovať časť reálneho sveta pomocou modelu,
- ak vytvárame nejaký model uistime sa, že každá jeho časť má vlastnosti zodpovedajúce jej reálnemu vzoru – systému,
- niektoré nepodstatné vlastnosti systému môžeme zanedbať a neprenášať ich do modelu,
- overovať hypotézy pomocou simulácie, ale aj matematicky,
- používať dátovú štruktúru slovník,
- kriticky prijímať na prvý pohľad pravdivé tvrdenia.

Ďalšie úlohy na precvičenie a zamyslenie

1. Ako funguje generátor pseudonáhodných čísiel? Ak sa na generovanie čísiel používa funkcia, ako je možné, že dostávame rôzne postupnosti čísiel?
2. Je možné docieliť to, aby sme opakovane vygenerovali rovnakú postupnosť pseudonáhodných čísiel?
    - Ak áno, ako?
    - Ak áno, ako docieliť to, aby ďalšie postupnosti náhodných čísiel už boli rôzne?
3. Máme nasledovnú teóriu ako vypočítať obsah kruhu, ak poznáme jeho polomer. Do štvorca vpíšeme kruh s polomerom r. Obsah štvorca aj obsah kruhu sú nejakou funkciou r^2 (pre zdôvodnenie pozri poznámku na okraji):



$S_{\square} = (2r)^2 = 4r^2$

$S_o = c * r^2$, kde c je nejaká konštanta, ktorú chceme nájsť, aby sme vedeli počítať obsah kruhu.

Ak dáme do pomeru $S_o : S_{\square} = c * r^2 : 4r^2$, odtiaľ pre c platí: $c = 4 S_o / S_{\square}$.

Skúsme teraz vystrihnúť vyššie uvedený obrázok a hádzať do neho náhodne šípky. Zrátajme koľkokrát sme trafili do kruhu (T_o) a koľko krát do štvorca ($T_{\square}$). Uvedomme si, že ak sme trafili kruh, trafili sme aj štvorec. Ak naše hody budú rovnomerne rozmiestnené tak pomer počtu zásahov do kruhu k počtu zásahov do štvorca by mal byť rovnaký ako pomer obsahu kruhu k obsahu štvorca: $T_o : T_{\square} = S_o : S_{\square}$. Ak je to pravda, vieme zistiť hodnotu nami hľadanej konštanty c. Overte simuláciou, či je naša úvaha správna.

Poznámka na okraj

Pri skúmaní systémov, ktoré podliehajú náhodným vplyvom je jednou z možností ich skúmania simulácia týchto systémov. Často sa ako metóda skúmania využíva metóda **Monte Carlo**, ktorá využíva náhodné alebo pseudonáhodné čísla. Princíp metódy je nasledovný:

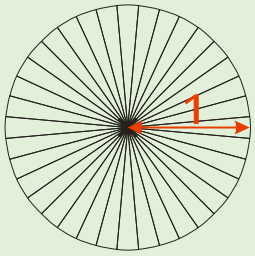
- návrh modelu, ktorý dostatočne presne popisuje skúmaný systém,
- simulácia veľkého množstva experimentov s modelom založená na náhodných alebo pseudonáhodných číslach,
- štatistické vyhodnotenie simulácie.

V prípadoch keď skúmané systémy podliehajú náhodným vplyvom predstavuje metóda Monte Carlo silný nástroj na skúmanie týchto systémov (11).

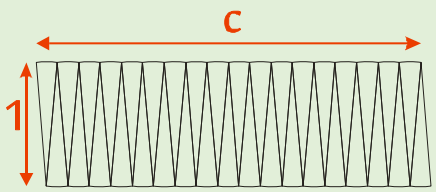
Poznámka na okraj

Prečo môžeme predpokladať, že obsah kruhu je nejakou funkciou r^2?

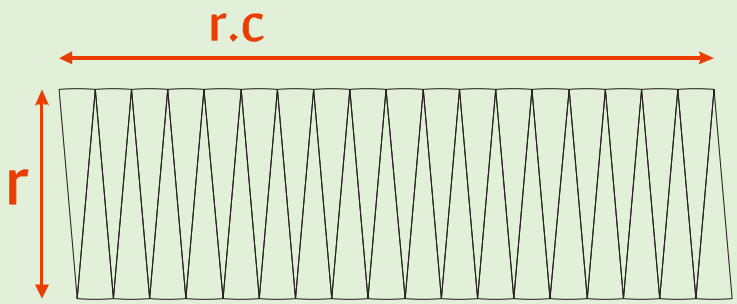
Uvažujme kruh s polomerom 1, ktorý rozdelíme na veľa rovnakých kruhových výsekov.



Preusporiadajme výseky nasledovne:



Pri dostatočne malých výsekoch môžeme oblúčiky zanedbať (predpokladajme, že výsekov máme nekonečne veľa). Obsah útvaru vieme vypočítať ako $1 \cdot c$. Zmeňme teraz polomer z 1 na nejaké r. Ak sa zmení polomer (výška útvaru), musí sa zmeniť aj jeho šírka:

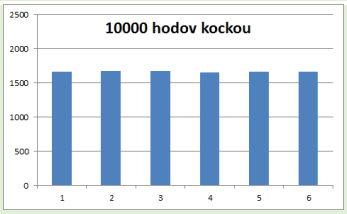


Aj pre tento útvar vieme vypočítať jeho obsah: $c \cdot r^2$. Ak tento útvar rozoberieme a vyskladáme z neho kruh dostaneme kruh s polomerom r. Preto môžeme tvrdiť, že obsah kruhu s polomerom r je $c \cdot r^2$, t.j. obsah kruhu je nejakou funkciou r^2.

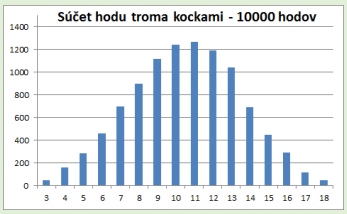
4. Aký súčet bude padať najčastejšie ak budeme hádzať 4 kockami? Simulujte aspoň 1000 hodov.
5. Porovnajte graf početností jednotlivých súčtov ak budeme hádzať 1, 2, 3 a 4 kockami.

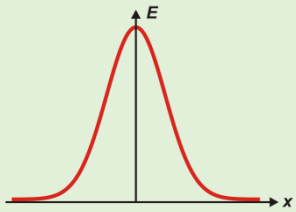
Poznámka na okraj

Početnosti padnutých čísiel pri hode jednou kockou vykazujú **rovnomerné rozdelenie pravdepodobnosti**. Pri rovnomernom rozdelení majú všetky hodnoty náhodnej veličiny rovnakú pravdepodobnosť.



Početnosti súčtu pri hode viacerými kockami sa približujú k **normálnemu** (alebo Gaussovmu) **rozdeleniu pravdepodobnosti**. Čím viac kociek použijeme tým viac sa k normálnemu rozdeleniu priblížime.





Okrem generátora náhodných čísiel, ktorý generuje náhodné čísla pre rovnomerné rozdelenie (napr. random.randint()) existujú aj generátory náhodných čísiel pre normálne rozdelenie (napr. random.gauss()). Ak vieme, ako sa správa náhodná premenná nejakého systému, vieme ju modelovať príslušným generátorom.

6. V súboji sa stretli dvaja pištoľníci. Cieľom každého z nich je súboj vyhrať. Prvý pištoľník má presnú mušku a v priemere 5 zo 6 výstrelov zasiahne terč. Druhý pištoľník je o niečo slabší a terč zasiahne v priemere 4 krát zo 6 výstrelov. Aká je pravdepodobnosť, že v súboji vyhrá slabší pištoľník? Je rozdiel v tom kto má prvý vystrel? Simulujte 1000 duelov.

   V súboji sa stretli traja pištoľníci. Každý z nich má inú úspešnosť zásahu. Prvý trafí terč s pravdepodobnosťou 5/6, druhú s pravdepodobnosťou 4/6 a tretí s pravdepodobnosťou 2/6. V súboji strieľajú podľa vopred dohodnutého poradia a súboj končí ak zostane už len jeden pištoľník. Premyslite si, akú stratégiu by si mal každý z nich zvoliť. Navrhnite, v akom poradí by mali pištoľníci strieľať aby bola šanca najslabšieho na prežitie čo najvyššia. Aká bude šanca najslabšieho prežiť pri tomto poradí? Simulujte 1000 súbojov.

3. Simulácie fyzikálnych pohybov

Kľúčové slová

simulácia, fyzikálne pohyby, trajektória, animácia, analytický model, diferenčný model, grafické rozhranie, generátor pseudonáhodných čísel

Čo sa naučíme a čo si precvičíme

- Analyzovať vybrané fyzikálne javy a zostaviť pre nich analytický model (tiež statický model), resp. diferenčný model (tiež dynamický model) pohybujúceho sa bodu podľa fyzikálnych zákonov (Newtonových pohybových zákonov, Coulombovho zákona),
- naprogramovať simuláciu pohybujúceho sa bodu využívajúcu grafické rozhranie s knižnicou Tkinter s viacerými ovládacími prvkami (Label, Entry, Button, Canvas, OptionMenu),
- použiť simulácie pohybujúceho sa bodu na overenie, resp. vyvrátenie hypotéz,
- aplikovať simulácie pohybujúceho sa bodu pri programovaní počítačových hier,
- pri tvorbe programov použiť (precvičiť) príkazy opakovania, vetvenia a priradenia, tvorbu vlastných funkcií, príkazy karteziánskej grafiky (`tkinter`), príkazy modulov `math` a `random`.

3.1 Rovnomerný priamočiary pohyb

Výkladový text

Podľa 1. Newtonovho pohybového zákona platí, že teleso zotrváva v pokoji alebo v rovnomernom priamočiarom pohybe, kým nie je nútené vonkajšími silami tento svoj stav zmeniť. Pre zjednodušenie simulácií pohybov telesa budeme ďalej zanedbávať rozmery telesa a budeme uvažovať o tzv. **hmotnom bode** (ďalej len bode). **Trajektóriou** rovnomerného priamočiareho pohybu bodu je časť priamky. Pre **dráhu** s rovnomerného priamočiareho pohybu bodu platí vzťah: $s = vt$, kde v je rýchlosť rovnomerného priamočiareho pohybu bodu a t je doba trvania rovnomerného priamočiareho pohybu.

Ak sa bod pohybuje rovnomerným priamočiarym pohybom rýchlosťou $v = (v_x, v_y)$ a v čase $t = 0$ sú jeho súradnice polohy rovné $(0, 0)$, tak pre jeho súradnice polohy x, y v ľubovoľnom čase *t* platí vzťah:

$$(x, y) = (v_x, v_y)t$$

alebo pre každú súradnicu polohy:

$$x = v_x t \qquad y = v_y t$$

Úloha 1

a) Analyzujte a so spolužiakom prediskutujte, čo robí uvedený program `rovnomerny1.py`.

```
# rovnomerny1.py
import tkinter

okno = tkinter.Tk()
platno = tkinter.Canvas(okno, width=300, height=300)
platno.grid()
v_x, v_y = 10, 15
for t in range(0, 21):
    x = v_x * t
    y = v_y * t
    platno.create_oval(x - 3, y - 3, x + 3, y + 3, fill='yellow')

okno.mainloop()
```

b) Upravte program tak, aby zobrazoval trajektóriu pohybu bodu v súradnicovom systéme s osou y nasmerovanou nahor, pričom veľkosť zobrazovaného bodu je väčšia ako predtým (napr. 10) a zobrazená inou farbou (napr. azúrovou, angl. cyan). Výsledný program uložte do súboru `rovnomerny1_R.py`.

Úloha 2

Analyzujte a so spolužiakom prediskutujte, čo robí uvedený program `rovnomerny2.py`.

```
# rovnomerny2.py
import tkinter

okno = tkinter.Tk()
platno = tkinter.Canvas(okno, width=300, height=300)
platno.grid()
x, y = 0, 0
v_x, v_y = 2, 3
bod = platno.create_oval(x - 2, y - 2, x + 2, y + 2, fill='red')
for t in range(0, 100):
    x = v_x * t
    y = v_y * t
    y = 300 - y
    platno.coords(bod, x - 2, y - 2, x + 2, y + 2)
    platno.update()
    platno.after(50)

okno.mainloop()
```

a) **V čom sa líši** uvedený program od programu `rovnomerny1.py` z úlohy 1?
b) Aký význam majú metódy **platno.coords(), platno.update()** a **platno.after()**?
c) Aký význam má pred cyklom uvedené priradenie **bod = platno.create_oval()**?

Výkladový text

Ak chceme na plátne zanechať trajektóriu pohybujúceho sa bodu, stačí opakovane použiť metódu `platno.create_oval()`, ktorá na plátne vytvorí súbor kruhov reprezentujúcich trajektóriu pohybujúceho sa bodu.

Ak chceme animovať pohyb bodu pomocou jedného kruhu, vytvoríme objekt `bod` pomocou priradenia `bod = platno.create_oval()`, ktorého súradnice polohy na plátne meníme metódou `platno.coords()`. Aby sme nevideli len poslednú pozíciu pohybujúceho sa bodu potrebujeme občerstviť obrazovku pri každej zmene súradníc bodu, čo dosiahneme pomocou metódy `platno.update()`. Na časové zdržanie (v milisekundách) použijeme metódu `platno.after()`.

Úloha 3

Vytvorte program `rovnomerny3_R.py` vykresľujúci trajektóriu rovnomerného priamočiareho pohybu bodu, ktorého grafické rozhranie umožňuje zadanie súradníc vektora rýchlosti (v_x, v_y), zmazanie plátna a ukončenie behu programu. (Nápoveď: Na zadanie hodnôt rýchlosti, vykreslenie trajektórie, zmazanie plátna a ukončenie programu použite prvky `Entry`, `Label`, `Button`, `Canvas` a riadiace premenné Tkinteru typu `DoubleVar` a metódu `get()` na ďalšie použitie v programe hodnoty zadanej v prvku `Entry`.)

Výkladový text

Pri tvorbe programov s väčším počtom prvkov grafického rozhrania sa stáva programový kód menej prehľadným. Alternatívou tvorby programu s grafickým rozhraním je použitie objektového prístupu (OOP), ktorý umožňuje napísať kompaktnejší programový kód. V alternatívnom OOP riešení úlohy 3, sme vytvorili triedu `Simulacia` odvodenú od triedy `Tk`, v ktorej sme definovali tri funkcie tejto triedy tzv. **metódy**. V metóde `__init__` nastavujeme počiatočné vlastnosti a spúšťame ďalšie metódy. V metóde `vytvorGUI` vytvárame grafické rozhranie programu a v metóde `kresli()` vykresľujeme na plátno trajektóriu pohybujúceho sa bodu. Metódy a premenné s prefixom `self` patria triede `Simulacia` a sú prístupné v každej jej metóde. Objekt triedy `Simulacia` vytvárame a spúšťame zápisom `Simulacia()`.

```
# rovnomerny4.py
import tkinter

class Simulacia(tkinter.Tk):
    def __init__(self):
        '''Nastavenie počiatočných vlastnosti a spúšťanie ďalších metód triedy'''
        super().__init__()
        self.title('Rovnomerný priamočiary pohyb')
        self.vytvorGUI()
        self.mainloop()

    def vytvorGUI(self):
        '''Vytvorenie grafického rozhrania programu (GUI)'''
        tkinter.Label(self, text='v_x =').grid(row=0, column=0)

        self.v_x = tkinter.DoubleVar()
        self.v_x.set(10)
        tkinter.Entry(self, textvariable=self.v_x).grid(row=0, column=1)

        tkinter.Label(self, text='v_y =').grid(row=0, column=2)

        self.v_y = tkinter.DoubleVar()
        self.v_y.set(15)
```

```
        tkinter.Entry(self, textvariable=self.v_y).grid(row=0, column=3)

        tkinter.Button(self, text='Kresli', command=self.kresli).grid(row=1,
column=0)
        tkinter.Button(self, text='Ukonči', command=self.destroy).grid(row=1,
column=3)
        self.platno = tkinter.Canvas(self, width=400, height=400,
background='#FFF')
        self.platno.grid(row=2, column=0, columnspan=4)

    def kresli(self):
        '''Vykreslenie trajektórie rovnomerného priamočiareho pohybu bodu'''
        v_x = self.v_x.get()
        v_y = self.v_y.get()
        for t in range(0,21):
            x = v_x * t
            y = v_y * t
            y = 400 - y
            self.platno.create_oval(x - 2, y - 2, x + 2, y + 2, fill='yellow')

Simulacia()
```

Úloha 4

Upravte programový kód súboru `rovnomerny4.py` tak, aby program vykreslil väčšie plátno a k nemu odpovedajúci počet bodov trajektórie. Do grafického rozhrania doplňte textové pole na zadanie veľkosti kruhu reprezentujúceho hmotný bod. Výsledný program uložte do súboru `rovnomerny4_R.py`. (Nápoveď: Výšku plátna reprezentuje výraz `int(self.platno['height'])`, resp. výraz `float(self.platno['height'])`.)

3.2. Rovnomerne zrýchlený pohyb

Výkladový text

Pozrime sa teraz na najjednoduchší prípad nerovnomerného pohybu, a to **rovnomerne zrýchleného priamočiareho pohybu**. Pre dráhu s rovnomerne zrýchleného pohybu bodu v ľubovoľnom čase t, ktorý sa pohybuje s konštantným zrýchlením a a v čase $t = 0$ mal počiatočnú rýchlosť v_0 a prejdenú počiatočnú dráhu s_0, platí vzťah:

$$s = s_0 + v_0 t + \frac{1}{2} a t^2$$

Vo všeobecnosti, ak uvažujeme rovnomerne zrýchlený pohyb bodu v rovine s vektorom stáleho zrýchlenia $a = (a_x, a_y)$, vektorom počiatočnej rýchlosti $v_0 = (v_{0x}, v_{0y})$ a počiatočnou polohou (x_0, y_0) v čase $t = 0$, môžeme podľa princípu nezávislosti pohybov uvažovať o tomto pohybe ako o zložení dvoch pohybov, napr. v smere súradnicových osí x a y. Potom pre súradnice polohy bodu x a y v ľubovoľnom čase t platia vzťahy:

$$x = x_0 + v_{0x} t + \frac{1}{2} a_x t^2$$

$$y = y_0 + v_{0y} t + \frac{1}{2} a_y t^2$$

kde x_0 a y_0 sú počiatočné súradnice polohy bodu, v_{0x} a v_{0y} sú súradnice vektora počiatočnej rýchlosti, a_x a a_y sú súradnice vektora konštantného zrýchlenia.

Úloha 5

a) Vytvorte program `zrychleny1_R.py` vykresľujúci trajektóriu bodu pohybujúceho sa rovnomerne zrýchlene, ktorý umožňuje zadanie súradníc počiatočnej polohy bodu (x_0, y_0), vektora zrýchlenia (a_x, a_y), vektora počiatočnej rýchlosti (v_{0x}, v_{0y}), celkového času pohybu $t_{celkový}$, zmazanie plátna a ukončenie behu programu.

b) Preskúmajte nastavením ktorých hodnôt dosiahnete vykreslenie trajektórie rovnomerne zrýchleného pohybu (napr. rovnomerne zrýchleného priamočiareho pohybu, vodorovného či šikmého vrhu).

3.3. Pohyb bodu, na ktorý pôsobia viaceré sily

Výkladový text

V predchádzajúcich simuláciách rovnomerného priamočiareho a rovnomerne zrýchleného pohybu bodu sme použili tzv. **analytický model,** resp. **analytické vyjadrenie súradníc** pohybujúceho sa bodu.

Pri simuláciách pohybu bodu, na ktorý pôsobia viaceré sily, môže byť analytické vyjadrenie súradníc veľmi náročné. Preto sa v praxi používa aj tzv. **diferenčný model**, resp. **diferenčné vyjadrenie súradníc** pohybujúceho sa bodu. Pri **diferenčnom modelovaní** pohybu bodu budeme uvažovať, že v každom patrične malom časovom intervale Δt sa bod sa pohybuje rovnomerným priamočiarym pohybom a ďalšie zmeny pohybového stavu môžu nastať až v nasledovnom časovom intervale Δt. Toto zjednodušenie umožňuje veľmi ľahko simulovať pohyb bodu, na ktorý pôsobia viaceré sily, no na druhej strane ide len o približné výpočty. Čím menší je časový interval Δt, tým presnejšie je modelovanie a bližšie k skutočným výpočtom. Pri diferenčnom modelovaní budeme postupne v rámci každého časového intervalu Δt meniť súradnice vektora rýchlosti v závislosti od aktuálnych súradníc vektora zrýchlenia (udeleného výslednicou síl) a následne meniť súradnice polohy pohybujúceho sa bodu v závislosti od aktuálnych súradníc vektora rýchlosti, čo vyjadríme pomocou nasledovných vzťahov:

$$(v_x, v_y) \leftarrow (v_x, v_y) + (a_x, a_y)\Delta t$$

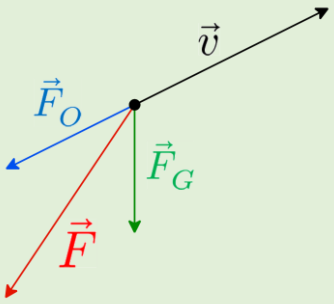
$$(x, y) \leftarrow (x, y) + (v_x, v_y)\Delta t$$

Uvedené vzťahy použijeme pri niekoľkých nasledujúcich úlohách na vykreslenie trajektórie pohybujúceho bodu, na ktorý pôsobia viaceré sily.

3.3.1. Balistická krivka

Výkladový text

Trajektóriou šikmého vrhu bodu v hmotnom prostredí je **balistická krivka.** Na bod pohybujúci sa rýchlosťou v pôsobia dve sily: gravitačná sila $\vec{F}_G$ a odporová sila $\vec{F}_O$.



Obrázok 11 Výsledná sila F pôsobiaca na pohybujúci sa bod ako vektorový súčet odporovej $\vec{\mathrm{F}}_{\mathrm{O}}$ a gravitačnej sily $\vec{\mathrm{F}}_{\mathrm{G}}$

Pri výpočte výsledného zrýchlenia, ktoré udeľuje bodu pôsobiaca výsledná sila, budeme vychádzať z 2. Newtonovho zákona, ktorý je reprezentovaný vzťahom: $\mathrm{a} = \frac{\vec{\mathrm{F}}}{\mathrm{m}}$.

Gravitačná sila $\vec{\mathrm{F}}_{\mathrm{G}}$ udeľuje pohybujúcemu sa bodu zrýchleníe $\mathrm{a} = (0, \text{-}\mathrm{g})$, kde v našich zemepisných šírkach je $\mathrm{g} \cong 9{,}81\ \mathrm{m.s^{-2}}$. Odporová sila $\vec{\mathrm{F}}_{\mathrm{O}} = \text{-}\mathrm{kv}$ udeľuje bodu pohybujúceho sa rýchlosťou v zrýchleníe $\mathrm{a}_{\mathrm{O}} = (\text{-}\frac{\mathrm{k}}{\mathrm{m}}\mathrm{v}_{\mathrm{x}}, \text{-}\frac{\mathrm{k}}{\mathrm{m}}\mathrm{v}_{\mathrm{y}})$.

Výslednica všetkých síl pôsobiacich na pohybujúci sa bod $\vec{\mathrm{F}} = \vec{\mathrm{F}}_{\mathrm{G}} + \vec{\mathrm{F}}_{\mathrm{O}}$ mu udeľuje výsledné zrýchlenie $\mathrm{a} = (\text{-}\frac{\mathrm{k}}{\mathrm{m}}\mathrm{v}_{\mathrm{x}}, \text{-}\frac{\mathrm{k}}{\mathrm{m}}\mathrm{v}_{\mathrm{y}}\text{-}\mathrm{g})$. (12) (13)

Úloha 6

a) Vytvorte program `balistika1_R.py` vykresľujúci trajektóriu bodu vystreleného počiatočnou rýchlosťou v_0 pod uhlom α pohybujúceho sa v gravitačnom poli Zeme v hmotnom prostredí (napr. vzduchu) so silou odporu prostredia $\vec{\mathrm{F}}_{\mathrm{O}} = \text{-}\mathrm{kv}$. Predpokladáme, že počiatočné súradnice polohy bodu $(\mathrm{x}_0, \mathrm{y}_0)$ majú hodnotu $(0, 0)$ a hmotnosť bodu $\mathrm{m} = 1\ \mathrm{kg}$. Program má umožniť zadanie hodnoty počiatočnej rýchlosti v_0, uhla šikmého vrhu α, koeficientu trenia k, časového intervalu $\Delta\mathrm{t}$, výpis dĺžky doletu $\mathrm{x}_{\max}$ a času doletu $\mathrm{t}_{\max}$, zmazanie plátna a ukončenie behu programu.

b) Po vytvorení programu experimentujte s rôznymi hodnotami časového intervalu $\Delta\mathrm{t}$, napr. $\Delta\mathrm{t} \in \{1; 0{,}1; 0{,}01; 0{,}001; 0{,}0001\}$ a zisťujte presnosť výpočtu dĺžky doletu a času doletu šikmého vrhu bez odporu prostredia. Tieto hodnoty porovnajte s teoretickými hodnotami $\mathrm{x}_{\max} = \frac{{\mathrm{v}_0}^2 \sin 2\alpha}{\mathrm{g}}$ a $\mathrm{t}_{\max} = \frac{2\mathrm{v}_0 \sin\alpha}{\mathrm{g}}$.

c) Pri akom uhle je dĺžka doletu vo vákuu maximálna? Ako je to s maximálnym doletom v hmotnom prostredí, platí rovnaký uhol ako vo vákuu?

d) Experimentujte s hodnotami koeficientu trenia k, napr. $\mathrm{k} \in \{0{,}0; 0{,}1; 0{,}2; 0{,}3; 0{,}4; 0{,}5\}$ a zistite ako sa mení dĺžka doletu $\mathrm{x}_{\max}$ v závislosti od zmeny koeficientu trenia k.

(Nápoveď: Pri výpočte hodnôt súradníc vektora rýchlosti $(\mathrm{v}_{\mathrm{x}}, \mathrm{v}_{\mathrm{y}})$ zo zadanej počiatočnej rýchlosti v_0 a zadaného uhla $\propto$ môžeme použiť vzťah $(\mathrm{v}_{\mathrm{x}}, \mathrm{v}_{\mathrm{y}}) = (\mathrm{v}_0 \cos\alpha\,, \mathrm{v}_0 \sin\alpha)$. Pri ich programovaní môžeme použiť modul `math` a jeho funkcie `math.cos()`, `math.sin()`, `math.radians()`.

Pri programovaní zadávania časového intervalu $\Delta\mathrm{t}$ zo zadanej množiny hodnôt môžeme použiť ovládací prvok výberový zoznam `OptionMenu` s uvedeným zdrojovým kódom:

```
tkinter.OptionMenu(self, self.dt, '0.0001', '0.001', '0.01', '0.1', '1.0')
```

Pri programovaní výpisu hodnôt odporúčame použiť f-reťazce.)

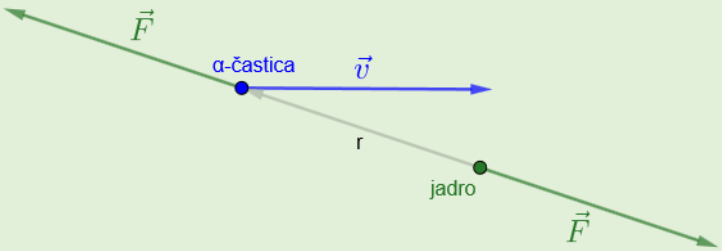
3.3.2. Rutherfordov experiment

Historická poznámka

V roku 1909 skúmali na návrh Ernesta Rutherforda jeho asistenti Hans Geiger a Ernest Marsden prechod častíc (katiónov $_2^4\mathrm{He}^{2+}$) zlatou fóliou. Pozorovali, že väčšina častíc prechádza fóliou. Neskôr zistili, že menší počet častíc sa veľmi odchyľuje od pôvodného smeru pohybu častíc alebo sa dokonca odráža pred fóliu, čo bolo v rozpore s **Thomsonovým modelom atómu**, podľa ktorého mala byť kladný náboj atómu rovnomerne rozmiestnený v celom objeme atómu. Po tomto experimente bol prijatý nový tzv. **Rutherfordov model atómu**, podľa ktorého, je kladný náboj atómu sústredený v jeho malej časti, v tzv. jadre atómu. (14)

Výkladový text

Poďme vytvoriť program, ktorý bude simulovať **Rutherfordov experiment**, t.j. pohyb kladných α-častíc v blízkosti kladné nabitého jadra atómu zlata. Na túto simuláciu použijeme diferenčný model. V našom prípade uvažujeme o elektrostatickej sile pôsobiacej medzi dvomi bodovými nábojmi – kladnou α-časticou a kladným jadrom zlata. Sila, ktorou pôsobí α-častica na jadro zlata je rovnako veľká ako sila, ktorou pôsobí jadro zlata na α-časticu. Vzhľadom na oveľa väčšiu hmotnosť jadra ako α-častice, budeme pre zjednodušenie predpokladať, že jadro sa prakticky nebude pohybovať a ďalej sa budeme zaoberať len simuláciou pohybu α-častice.



Obrázok 13 Na pohybujúcu sa α-časticu pôsobí jadro elektrostatickou silou F v smere polohového vektora od jadra atómu k α-častici

Pre elektrostatickú silu medzi dvomi bodovými nábojmi $\mathrm{q_1}$ a $\mathrm{q_2}$ vo vákuu so vzdialenosťou r medzi sebou platí Coulombov zákon $\vec{\mathrm{F}} = \frac{1}{4\pi\varepsilon_0}\frac{\mathrm{q_1 q_2}}{\mathrm{r^3}}\mathrm{r}$. Využitím 2. Newtonovho zákona dostaneme vzťah $\mathrm{ma} = \frac{1}{4\pi\varepsilon_0}\frac{\mathrm{q_1 q_2}}{\mathrm{r^3}}\mathrm{r}$, z ktorého vyjadríme zrýchlenie $\mathrm{a} = \frac{\mathrm{kr}}{\mathrm{r^3}}$, kde $\mathrm{k} = \frac{\mathrm{q_1 q_2}}{4\pi\varepsilon_0 \mathrm{m}}$ je výraz tvorený veličinami, ktoré sa nebudú meniť počas simulácie. Vektor $\mathrm{r} = (\mathrm{r_x}, \mathrm{r_y})$ je určený počiatočným bodom – jadrom zlata $(\mathrm{x_0}, \mathrm{y_0})$ a koncovým bodom – pohybujúcou sa α-časticou (x, y), t. j. $\mathrm{r} = (\mathrm{r_x}, \mathrm{r_y}) = (\mathrm{x\text{-}x_0}, \mathrm{y\text{-}y_0})$. Pre aktuálne súradnice vektora zrýchlenia platí:

$$(\mathrm{a_x}, \mathrm{a_y}) = \left(\frac{\mathrm{kr_x}}{\mathrm{r^3}}, \frac{\mathrm{kr_y}}{\mathrm{r^3}}\right) = (\mathrm{x\text{-}x_0}, \mathrm{y\text{-}y_0})\frac{\mathrm{k}}{\mathrm{r^3}}, \mathrm{kde}\ \mathrm{r} = \sqrt{(\mathrm{x\text{-}x_0})^2 + (\mathrm{y\text{-}y_0})^2}$$

Pri diferečnom modelovaní pohybu α-častíc v blízkosti jadra atómu zlata využijeme aktuálne súradnice vektora zrýchlenia na výpočet aktuálnych súradníc vektora rýchlosti a tie následne na výpočet aktuálnych súradníc polohy pohybujúcej sa α-častice podľa už známych vzťahov (uvedené v 3.3):

$$(v_x, v_y) \leftarrow (v_x, v_y) + (a_x, a_y)\Delta t$$

$$(x, y) \leftarrow (x, y) + (v_x, v_y)\Delta t$$

Zdrojový kód metódy **kresli()** vykresľujúcej zadaný počet letiacich α-častíc so zadaným časovým intervalom a vypisujúcej počet odrazených α-častíc môže vyzerať nasledovne:

```
def kresli(self):
    '''Vykreslenie trajektórie alfa-častíc letiacich okolo jadra zlata'''
    pocet = self.pocet.get()    # počet lúčov
    dt = self.dt.get()          # časový interval
    k = 300                     # výraz q1 * q2 / (4 * pi * epsilon0 * m)
    x0, y0 = 200, 300           # počiatočné súradnice polohy jadra
    odrazene = 0                # počet odrazených alfa častíc
    self.platno.create_oval(x0 - 9, y0 - 9, x0 + 9, y0 + 9, fill='gold')
    for i in range(pocet):
        x, y = 1, random.randrange(1, 600)
        v_x, v_y = 5, 0
        while 0 < x < 800 and 0 < y < 600:
            koeficient = k / (((x - x0) ** 2 + (y - y0) ** 2) ** (3/2))
            a_x = (x - x0) * koeficient  # aktuálne súradnice zrýchlenia
            a_y = (y - y0) * koeficient
            v_x = v_x + a_x * dt         # aktuálne súradnice rýchlosti
            v_y = v_y + a_y * dt
            x = x + v_x * dt             # aktuálne súradnice polohy
            y = y + v_y * dt
            self.platno.create_oval(x - 1, y - 1, x + 1, y + 1, outline = 'blue')
            self.platno.update()
            if x < 1:
                odrazene +=1
    self.vypis.set(f'{odrazene} odrazených alfa častíc')
```

Kompletný zdrojový kód programu je uložený v súbore `rutherford.py`.

Úloha 7

Po vytvorení programu `rutherford.py` simulujte Rutherfordov experiment pre rôzne hodnoty parametrov. Pozorujte a prediskutujte so spolužiakmi nasledovné otázky:

- Aký počet odrazených α-častíc od jadra atómu zlata ste zaregistrovali?
- Aké rôzne typy trajektórií α-častíc ste zaregistrovali?
- Čo tvorí množinu všetkých trajektórií α-častíc?

3.3.3. Lissajousove krivky

Výkladový text

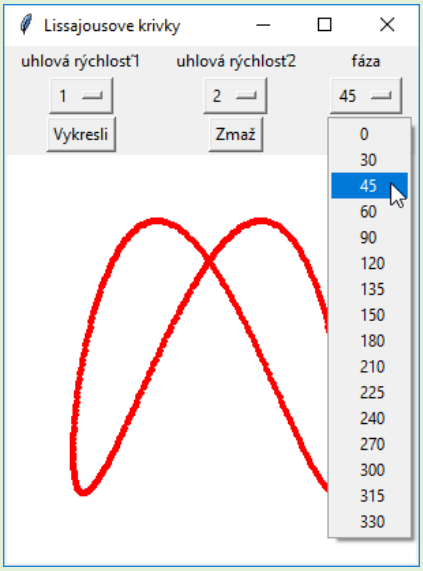
Lissajousove krivky (pomenované podľa francúzskeho fyzika Julesa Antoine Lissajousa) popisujú trajektóriu bodu pohybujúceho sa zloženým pohybom, ktorý je tvorený dvomi na seba kolmými harmonickými pohybmi. Trajektória takého pohybu je popísaná rovnicami:

$$x(t) = a_1 \sin \omega_1 t \qquad y(t) = a_2 \sin \omega_2 t + \varphi$$

kde a_1 je amplitúda, ω_1 je uhlová rýchlosť 1. harmonického pohybu a a_2 je amplitúda, ω_2 je uhlová rýchlosť a φ je fázový posun 2. harmonického pohybu vzhľadom k 1. harmonickému pohybu.

Pre lepšiu predstavu vykreslenia Lissajousových kriviek sa môžeme pozrieť na video s ich mechanickou simuláciou https://www.youtube.com/watch?v=4CbPksEl51Q.

Poďme naprogramovať simuláciu vykresľovania Lissajousových kriviek. Grafické rozhranie programu môže obsahovať tlačidlá a rozbaľovacie ponuky pre parametre ω_1, ω_2 a φ:



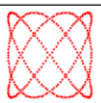
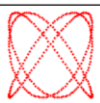
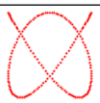
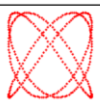
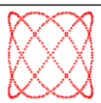
Obrázok 15 Grafické rozhranie programu simulujúceho vykresľovanie Lissajousových kriviek

Zdrojový kód metódy `kresli()` vykresľujúci Lissajousove krivky pre rôzne uhlové rýchlosti ω_1 a ω_2 a rôznu fázu φ môže vyzerať nasledovne:

```
def kresli(self):
    '''Vykreslenie Lissajousovej krivky'''
    o1 = self.o1.get()
    o2 = self.o2.get()
    fi = self.fi.get()
    for t in range(360):
        x =  100 * math.sin(math.radians(t * o1 + 0))  + 150
        y = -100 * math.sin(math.radians(t * o2 + fi)) + 150
        self.platno.create_oval(x - 1, y - 1, x + 1, y + 1, outline='red',
fill='red')
```

Kompletný zdrojový kód programu je uložený v súbore `lissajouse.py`.

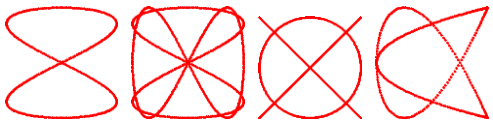
Nastavením rôznych hodnôt uhlových rýchlostí ω_1 a ω_2 a fázy φ dostaneme veľmi zaujímavé obrázky, ako napr.:

ϕ $\omega_1 : \omega_2$	0°	45°	90°	135°	180°
1 : 1					
1 : 2					
2 : 3					
3 : 4					

Obrázok 16 *Lissajousove krivky pre rôzne pomery uhlových rýchlosti a rôzne fázy*

Úloha 8

Spustite program `lissajouse.py` a nastavením parametrov vykreslite rôzne obrázky, napr.:



3.3.4. Ďalšie zaujímavé krivky

Napokon si ukážeme ďalšie zaujímavé krivky, ktoré by sme mohli ľahko naprogramovať. Pri kotúľaní kružnice s polomerom r zvonku po inej kružnici s polomerom R trajektória jej pevného bodu vytvára krivku – **epicykloidu** (15), ktorej parametrické rovnice sú:

$$x\ \theta\ =\ R+r\ \cos\theta - r\cos\left(\frac{R+r}{r}\theta\right)$$

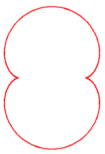
$$y\ \theta\ =\ R+r\ \sin\theta - r\sin\left(\frac{R+r}{r}\theta\right)$$

Ak označíme $k = \frac{R}{r}$ dostaneme nasledovné rovnice:

$$x\ \theta\ = r\ k+1\ \cos\theta - r\cos\ (k+1)\theta$$

$$y\ \theta\ = r\ k+1\ \sin\theta - r\sin\ (k+1)\theta$$

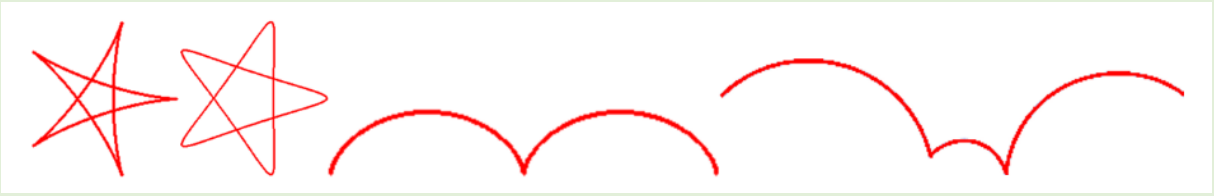
Ak je pomer polomerov k racionálnym číslom, epicykloida je uzavretá.

				
k = 1 (kardioida)	k = 2 (nefroida)	k = 3	k = 5 / 3	k = 19 / 5

Obrázok 17 *Ukážky epicykloid pre vybrané hodnoty parametra* k

Medzi ďalšie zaujímavé krivky patria:

- **hypocykloidy** (16) – krivky tvorené trajektóriou pevného bodu kružnice, ktorá sa kotúľa vnútri inej kružnice bez kĺzania,
- **hypotrochoidy** (17) – krivky tvorené trajektóriou pevného bodu ležiaceho na predĺženom polomere kružnice, ktorá sa kotúľa vnútri inej kružnice bez kĺzania,
- **cykloidy** (18) – krivky tvorené trajektóriou pevného bodu kruhu, ktorý sa kotúľa po priamke bez kĺzania,
- **cyklogóny** (19) – krivky tvorené trajektóriou pevného bodu mnohouholníka, ktorý sa prevaľuje po priamke bez kĺzania.



Obrázok 18 *Ukážky hypocykloidy, hypotrochoidy, cykloidy a cyklogóny*

Čo sme sa naučili

- Zostaviť analytický, resp. diferenčný model pohybujúceho sa bodu podľa fyzikálnych zákonov,
- naprogramovať simuláciu pohybujúceho sa bodu využívajúcu grafické rozhranie s knižnicou Tkinter s ovládacími prvkami,
- použiť simulácie pohybujúceho sa bodu na overenie, resp. vyvrátenie hypotéz.

Ďalšie úlohy na precvičenie a zamyslenie

1. **Hra zasiahni terč.** Vo zvislom smere sa po úsečke pohybuje terč rovnomerným priamočiarym pohybom, pričom na okrajoch úsečky terč zmení smer pohybu na opačný. Vo vodorovnom smere je akustické delo, pomocou ktorého môžeme zasiahnuť pohybujúci sa terč. Navrhnite a naprogramujte hru, v ktorej budete zasahovať kolmo sa pohybujúci terč.
2. **Simulácia pohybu biliardovej gule na obdĺžnikovom stole.** Vytvorte program vykresľujúci trajektóriu biliardovej gule pohybujúcej sa rovnomerne priamočiaro na uzavretom obdĺžnikovom stole. Predpokladáme, že biliardová guľa sa pohybuje rovnomerným priamočiarym pohybom bez rotácie, pričom jej odrazy od okrajov stola sú pružné (platí, že uhol dopadu na okraj stola je rovný uhlu odrazu od okraja stola).

Ako by sa zmenilo riešenie úlohy, ak by stôl nebol obdĺžnikový, ale kruhový alebo eliptický?

3. **Hod lopty do basketbalového koša.** Vytvorte program, ktorý pre náhodne vygenerovanú vzdialenosť hráča od koša a zadanú výšku hráča umožní hráčovi čo najmenším počtom hodov trafiť loptu do koša. Hráč pri každom hode zadáva rýchlosť lopty a uhol, pod ktorým hádže loptu do koša.
4. **Krivka sledovania.** Korisť sa pohybuje rovnomerným priamočiarym pohybom. Lovec ju prenasleduje rovnomernou rýchlosťou, pričom je na ňu stále nasmerovaný. Spresnite špecifikáciu tohto problému a naprogramujte trajektóriu pohybu koristi a lovca, ktorý ju prenasleduje. (20)
5. **Pristátie na Mesiaci.** Pilot padá s pristávacím modulom z určitej výšky na povrch Mesiaca. Voľný pád modulu môže pilot brzdiť zapnutím motorom proti smeru voľného pádu. Pilot má k dispozícii len obmedzené množstvo paliva. Podľa rýchlosti, ktorú dosiahne modul na povrchu Mesiaca, môžeme konštatovať mäkké pristátie, tvrdé pristátie, resp. deštrukciu modulu.
6. **Jednoduchý letecký simulátor.** Naprogramujte jednoduchý letecký simulátor, ktorý bude zohľadňovať gravitačnú silu, silu odporu prostredia, silu vetra zo zadaného smeru, vztlakovú silu a tiež ťah motora lietadla.

4. Hra Život

Kľúčové slová

simulácia, okolie bodu, indexovanie okolia , iterácia , zoznam zoznamov, umelý život

Čo sa naučíme a čo si precvičíme

- vysvetlíme si ako zakódovať dvojrozmerné hracie pole (šachovnicu),
- vysvetlíme ako indexovať okolie bodov, ako adresovať susedné bunky, v dvojrozmernom prostredí,
- vysvetlíme si, čo je to bunkový automat,
- zadefinujeme si pojem umelý život,
- naučíme sa identifikovať vzory správania sa.

Problémová situácia

Umelý život (Artificial life, Alife) je mladá vedecká disciplína. V priebehu posledného storočia sa táto téma vplyvom rapídneho rozvoja techniky preniesla z mýtov a sci-fi literatúry aj do sveta vedy. Umelý život simuluje prírodné procesy a kopíruje ich správanie v oblastiach, ktoré sú pre človeka užitočné. Rovnako zaujímavé sú aj bunkové automaty, ktoré sa často využívajú na simuláciu a modelovanie fyzikálnych a chemických procesov, v ekológii ako model predátor-korisť, v informatike na budovanie paralelných počítačov a v poslednom desaťročí sú populárne aj v behaviorálnych a sociálnych vedách (21). Cieľom našej hodiny je navrhnúť a vytvoriť model bunkového automatu (definícia neskôr) a následne v ňom pozorovať a popísať javy, ktoré reprezentujú princípy umelého života.

Poznámka na okraj

Umelý život – Alife - dostala svoje pomenovanie od Christophera Langtona v roku 1986.

Hľadajme riešenie

Pri vytváraní umelých „živých“ systémov nie je podmienkou, aby spĺňali kritériá našej definície života. Cieľom je napodobniť alebo naformulovať správanie sa systému na úrovni primitív a potom pozorovať jeho vývoj a vlastnosti na globálnej úrovni.

V roku 1970 predstavil John Horton Conway doteraz najpopulárnejší bunkový automat a nazval ho ***Hra Život*** (z anglického Game of Life). Hra ŽIVOT simuluje život spoločenstva mikroorganizmov v prostredí, kde zdanlivo nič nebráni ich úspešnému vývoju. Postupne si budeme popisovať pravidlá fungovania tejto simulácie umelého života . (22)

Poznámka na okraj

Zaujímavosťou je, že v čase vynájdenia hry, ju ešte nebolo možné simulovať počítačom a preto sa v originálnom článku uvádza, že na hranie je najvhodnejšia doska s kamienkami alebo štvorcový papier a ceruzka.

Formálne je hra Život veľmi jednoduchý dvojrozmerný bunkový automat, s dvoma stavmi a jednoduchou lokálnou prechodovou funkciou.

Výkladový text

Bunkový automat (CA, celulárny automat) je tvorený je N-rozmernou pravidelnou **štruktúrou buniek** (jednorozmerné pole, dvojrozmerná mriežka môže byť trojuholníková, štvorcová alebo šesťuholníková). Bunka nadobúda jeden z konečnej množiny **stavov** (najčastejšie 0 - mŕtva, 1 - živá). Každý stav je charakterizovaný určitými parametrami, ktoré potom nesie aj bunka v tomto stave. Množina všetkých dvojíc buniek a ich stavov v čase `t` sa nazýva **konfigurácia**. Konfigurácia v čase `t+1` sa vypočíta z konfigurácie v čase `t` na základe lokálnej prechodovej funkcie. (23)

Lokálna prechodová funkcia určuje stav bunky v ďalšej generácii na základe súčasného stavu bunky a stavov jej susedov. (24)

Úloha 1

Navrhnite dvojrozmerný bunkový automat o veľkosti NxN a umožnite, aby sa bunky mohli náhodne nastaviť do jedného z dvoch stavov. Na zadávanie využite udalosť stlačenie tlačidla myši a na vykreslenie grafické rozhranie `tkinter`.

Otvorme súbor `GameOfLife_uloha_1z.py`. Našou úlohou je doprogramovať funkcie `mriezka` a `vytvorGUI`.

```
import tkinter

class GameOfLife(tkinter.Tk):

    def __init__(self, velkost, n):
        super().__init__()
        self.title("Hra Život")
        self.velkost = velkost
        self.n = n
        self.vytvorGUI()
        self.mainloop()

    def mriezka(self):
        pass

    def vytvorGUI(self):
        pass

GameOfLife(500,50)
```

Otázka pre žiaka

Experimentujte so vstupnými parametrami funkcie `GameOfLife`. Ako sa bude meniť počet buniek a veľkosť plátna?

Najskôr teda vytvoríme grafické rozhranie, plátno, na ktorom sa nám bude vykresľovať umelý život. Naše kresliace plátno bude mať meno `self.plocha`.

```
def vytvorGUI(self):
      self.plocha = tkinter.Canvas(width = self.velkost, height =
      self.velkost)
      self.plocha.grid(row=0, column=0)
```

Bunkový automat (CA, celulárny automat) je tvorený N-rozmernou plochou. Aby sme presnejšie vedeli meniť stav konkrétnej bunky priamo klikaním do grafického plátna, môžeme si vykresliť mriežku. Keďže predpokladáme, že sa počet buniek môže meniť (aj keď my používame fixný počet 50x50), vykreslíme mriežku vzhľadom na aktuálne vypočítanú šírku bunky (`sirka`).

```
def mriezka(self):
      sirka = self.velkost // self.n
      for i in range(self.n):
        self.plocha.create_line(i*sirka,0,i*sirka,self.velkost)
        self.plocha.create_line(0,i*sirka,self.velkost,i*sirka)
```

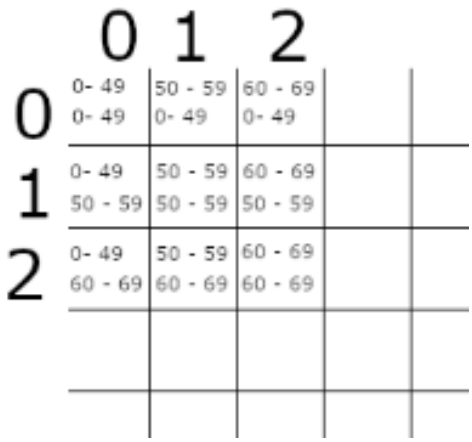
Mriežku teraz vykreslíme a pridáme ako posledný príkaz vo funkcii `VytvorGUI`.

```
self.mriezka()
```

Teraz nám už nič nebráni v tom, aby sme previazali udalosť kliknutia s metódou, ktorá vypočíta index označenej bunky. Pridáme príkaz vo funkcii `VytvorGUI`.

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

V momente keď klikneme na plátno, nastane udalosť `<Button-1>` a zároveň sa zapamätáva skupina informácií popisujúcich danú udalosť. Okrem iného máme informáciu o presnom mieste kliknutia. Tieto informácie sa automaticky prinesú ako parameter do metódy `klik`. Ako však indexovať bunky v mriežke? Každá bunka bude mať dva indexy `[x][y]`, x – stĺpcový, y – riadkový (pozri obrázok nižšie).



Obrázok 19 Ako sa indexujú bunky mriežky?

```
def klik(self,p):
    sirka = self.velkost // self.n
    x = p.x // sirka                                          [1]
```

```
    y = p.y // sirka                                          [2]
    self.plocha.create_oval (x*sirka,y*sirka,(x+1)*sirka,
    y+1)*sirka, fill = 'blue')                                [3]
```

[1] `p.x` nesie informáciu o tom, v ktorom stĺpci grafického plátna sme klikli. Ak chceme vedieť v ktorej časti mriežky sme, musíme túto informáciu vydeliť šírkou jednej bunky. Prvá bunka v poradí je tak definovaná rozsahom 0-49, druhá rozsahom 50 -59, tretia 60 – 69 atď.

[2] `p.y` nesie informáciu o tom, v ktorom riadku grafického plátna sme klikli. Ak chceme vedieť v ktorej časti mriežky to je, musíme túto informáciu opäť vydeliť šírkou jednej bunky. Prvá bunka v poradí je tak opäť definovaná rozsahom 0-49, druhá rozsahom 50 -59, tretia 60 – 69 atď.

[3] Po nájdení pozície v dvojrozmernej mriežke, môžeme v bunke vykresliť organizmus – modrý kruh.

Naše aktuálne riešenie umožňuje v mriežke označiť ktorúkoľvek bunku. Ak však bunku raz označíme, nevieme ju vrátiť do pôvodného prázdneho stavu. Funkcia klik totiž netestuje v akom stave sa bunka nachádzala pred kliknutím.

Úloha 2

Upravte úlohu 1 tak, aby sme vedeli bunky v automate zapínať a vypínať. Uchovávajte informáciu o tom, v akom stave sa bunkový automat momentálne nachádza.

Otvorme súbor `GameOfLife_uloha_2z.py`. Našou úlohou je doprogramovať prázdne funkcie.

V predchádzajúcom riešení sme si neuchovávali žiadne informácie o stave bunkového automatu. Vieme však, že bunky sú usporiadané v tvare mriežky, pričom každú z nich môžeme jednoznačne identifikovať podľa stĺpcového a riadkového indexu. Preto sa nám ako najvhodnejšia štruktúra javí použitie zoznamu zoznamov. A keďže každá bunka môže nadobúdať len jeden z dvoch stavov, využijeme dátový typ `bool`.

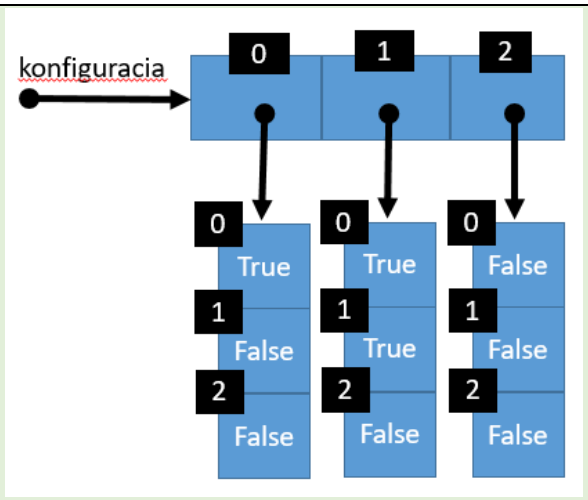
Výkladový text

Zoznam môže obsahovať ľubovoľné prvky a teda aj ďalšie zoznamy. Výsledok môže byť napríklad takýto:

```
konfiguracia = [[True, False, False],[True, True, False], [False,
False, False]]
```

Aká je hodnota prvku `konfiguracia[0]`? Je ním zoznam `[True, False, False]`. Aj tento zoznam má však indexovatelné prvky. Ak by sme sa teda spýtali na hodnotu prvku `konfiguracia[0][0]`, tak dostaneme `True`. Ako vidíme, dostali sme štruktúru, ktorá má dva indexy a to je to čo potrebujeme.

Bunkový automat teda budeme ukladať ako zoznam stĺpcov.



Po spustení simulácie neexistuje žiadny organizmus, preto má každá bunka nastavený stav na `False`.

```
def prazdna_konfiguracia(self)
     self.konfiguracia = []
     for i in range(self.n):
         self.konfiguracia.append([False]*self.n)
```

Prázdnu konfiguráciu je potrebné vytvoriť hneď po spustení, preto do funkcie `__init__` musíme funkciu zavolať `self.prazdna_konfiguracia()`. Tento stav môže zmeniť používateľ kliknutím na mriežku (alebo simulácia na základe pravidiel, ktoré si stanovíme o chvíľu). Ešte pred tým, však vyriešme, ako upraviť metódu `klik`, tak aby brala do úvahy aj aktuálny stav bunky a to nasledovne. Ak sme klikli na bunku:

- kde bol organizmus, tak organizmus zrušíme,
- kde organizmus nebol, tak ho vytvoríme.

Na vyriešenie konfigurácie nám stačí použiť negáciu (**not**) súčasného stavu.

```
def zmen_organizmus(self, x, y):
  self.konfiguracia[x][y] = not self.konfiguracia[x][y]
```

Výkladový text

Situáciu by sme mohli riešiť aj s použitím vetvenia

```
if self.konfiguracia[x][y] == True:
    self.konfiguracia[x][y] = False
else:
    self.konfiguracia[x][y] = True
```

Nás však vôbec nezaujíma, aký bol stav pred kliknutím. Jediné, čo chceme je tento stav zmeniť. A tak použitie negácie `not`, ktorá len preklopí stav `True` na `False`, alebo `False` na `True` je z programátorského pohľadu hodnotnejšie.

Vykreslenie organizmu je pre nás zaujímavé len vtedy, keď je v konfigurácii uvedená hodnota `True`. Prejdeme teda celú konfiguráciu a vykreslíme len tie organizmy, ktoré v bunkách žijú.

Musíme prejsť všetkými hodnotami v zozname konfigurácia. Preto je nutné použiť vnorený cyklus.

```
def vykresli_organizmus(self):
        sirka = self.velkost // self.n
        for x in range(self.n):
            for y in range(self.n):
                if self.konfiguracia[x][y]:
                  self.plocha.create_oval(x*sirka,y*sirka,
                  (x+1)*sirka,(y+1)*sirka, fill = 'blue')
```

Posledným problémom, ktorý treba vyriešiť je, ako priebežne vykresľovať alebo mazať organizmy v mriežke na grafickom plátne.

Najjednoduchším spôsobom, ako reagovať na zmenu, je vymazať plátno `self.plocha.delete('all')` a prekresliť ho s aktuálnymi hodnotami.

```
def klik(self,p):
    sirka = self.velkost // self.n
    x = p.x // sirka
    y = p.y // sirka
    self.zmen_organizmus(x, y)
    self.plocha.delete('all')
    self.mriezka()
    self.vykresli_organizmus()
```

Výkladový text

Po kliknutí na formulár sa vykoná sekvencia:

1) Zistia sa súradnice bunky, v ktorej sme klikli.

2) Zmení sa stav organizmu v bunke v aktuálnej konfigurácii.

3) Zmaže sa plocha.

4) Vykreslí sa mriežka.

5) Vykreslia sa všetky živé organizmy.

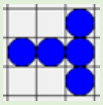
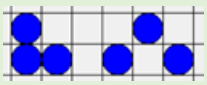
Úloha 3

Spustite simuláciu umelého života, ktorého podstata bude založená na týchto pravidlách reprodukcie a umierania:

1. ak je v bunke organizmus a ten má práve 2 alebo 3 susedov, tak táto bunka prežije aj do ďalšej generácie
2. ak je v bunke organizmus a má menej ako 2 susedov, organizmus do ďalšej generácie neprežije (umiera na samotu)
3. ak je v bunke organizmus a má viac ako 3 susedov, organizmus do ďalšej generácie neprežije (umiera na premnoženie)
4. ak v bunke nie je organizmus a zároveň má za susedov práve tri organizmy, tak sa tu v ďalšej generácii narodí nový organizmus.

Otázka pre žiaka

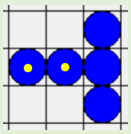
Ako bude vyzerať nasledujúca generácia pre tri modelové situácie?

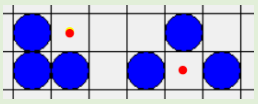
A)  B)  C) 

Poznámka pre učiteľa

V materiáloch je pripravená prezentácia s pravidlami, na základe ktorých funguje hra Život.

 Každý organizmus má dvoch susedov. Všetky prežijú do ďalšej generácie.

 Označené organizmy nespĺňajú kritéria prežitia.

 Vo vyznačených bunkách vznikne v ďalšej generácii nový organizmus.

Otvorme súbor `GameOfLife_uloha_3z.py`. Našou úlohou je doprogramovať prázdne funkcie.

Každá simulácia začína s **inicializačnou konfiguráciou**. To je tá, ktorú sme sa naučili zadávať v predchádzajúcich úlohách. Simulácia ďalej prebieha v iteráciách. V každej iterácii sa pre každú bunku vyhodnotia všetky pravidlá. Počet organizmov sa počas vyhodnotenia nikdy nemení. Zmena sa prejaví až v nasledujúcej iterácii.

V každej iterácii je potrebné pre každú bunku otestovať počet živých susedov. Okrajové bunky však nemajú všetkých susedov. Pri testovaní by sme na to mali myslieť.

```
def otestuj(x,y):
    if (x>=0 and x<self.n) and (y>=0 and y<self.n)
    and self.konfiguracia[x][y]:
        return 1
    else:
        return 0
```

Funkcia `otestuj(x,y)` zisťuje, či sa v bunke nachádza organizmus a zároveň chráni hranice mriežky.

Výkladový text

Stav bunkového automatu kódujeme v zozname zoznamov. Prvky zoznamu sú indexované v rozsahu `0 .. n-1`

Testovať, či v bunke žije organizmus má preto význam len v prípade, že sme pri kontrole nevyšli zo žiadnej strany z mriežky `(x>=0 and x<n) and (y>=0 and y<n)`.

Z popisu prechodovej funkcie je jasné, že potrebujeme pracovať s okolím bunky, musíme vedieť, v akom stave sú susedia. V našom prípade má každá bunka maximálne 8 susedov.

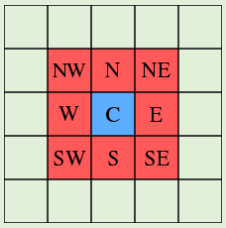
Okolie bodu [x][y] adresujeme zmenou stĺpcového a riadkového indexu (pozri obrázok nižšie). Keďže nás zaujíma len bezprostredné okolie, tak zmena bude maximálne o jeden riadok (stĺpec).

[x-1] [y-1]	[x] [y-1]	[x+1] [y-1]
[x-1] [y]	[x] [y]	[x+1] [y]
[x-1] [y+1]	[x] [y+1]	[x+1] [y+1]

Obrázok 20 Adresovanie susedných buniek

Poznámka na okraj

Toto susedstvo je tzv. Moorovho typu so vzdialenosťou 1 (25).



Zistime teda pre aktuálnu bunku, koľko živých susedov sa nachádza v jeho bezprostrednom okolí.

```
def susedia(self,x,y):
    pocet = 0

    pocet += self.otestuj(x-1, y-1)
    pocet += self.otestuj(x, y-1)
    pocet += self.otestuj(x+1, y-1)
    pocet += self.otestuj(x-1, y)
    pocet += self.otestuj(x+1, y)
    pocet += self.otestuj(x-1, y+1)
    pocet += self.otestuj(x, y+1)
    pocet += self.otestuj(x+1, y+1)

    return pocet
```

Výkladový text

Na testovanie môžeme použiť aj zápis s využitím vnoreného cyklu. Funkcia susedia môže vyzerať aj nasledovne.

```
for i in range(x-1, x+2):
    for j in range(y-1,y+2):
        pocet += self.otestuj(i, j)
```

V tejto verzii sa v skutočnosti kontroluje až 9 „susedov“. V rozsahu sa totiž nachádza aj bunka na pozícii [x][y]. Ak sa v danej bunke nachádza organizmus, môže to skresliť výsledok. Preto je potrebný krok, ktorý výsledok upraví.

```
if self.konfiguracia[x][y]: pocet -= 1
```

Teraz, keď už vieme zistiť počet živých susedov, môžeme to aplikovať pri výpočte novej generácie.

```
# V každej iterácii nás zaujímajú len bunky, v ktorých organizmus prežije
alebo organizmus vzniká.
def novaKonfiguracia(self):
    nova = []                                                    [1]
    for i in range(self.n):
        nova.append([False]*self.n)

    for i in range(self.n):                                      [2]
        for j in range(self.n):
            pocet = self.susedia(i,j)

            if self.konfiguracia[i][j] == True:                  [3]
                if pocet == 2 or pocet == 3:
                        nova[i][j] = True
            if self.konfiguracia[i][j] == False:                 [4]
                if pocet == 3:
                        nova[i][j] = True
    return nova
```

[1] Na začiatku každej iterácie vytvoríme novú, prázdnu generáciu buniek.
[2] Pre každú bunku aktuálnej konfigurácie vypočítame počet živých susedov.
[3] Ak sa v testovanej bunke nachádza organizmus a zároveň spĺňa podmienky na prežitie, zapíšeme ho do novej generácie.
[4] Ak sa v testovanej bunke nenachádza organizmus a zároveň spĺňa podmienky pre vznik nového, zapíšeme ho do novej generácie.

Aby sme mohli regulovať prechod medzi iteráciami, pridáme jedno tlačidlo, ktoré bude spúšťačom pre výpočet a vykreslenie novej generácie organizmov.

```
def dalsia_konfiguracia(self):
        self.konfiguracia[:] = self.novaKonfiguracia()
        self.plocha.delete('all')
        self.mriezka()
        self.vykresli_organizmus()
```

Výkladový text - zopakujme si

V metóde ďalšia sme použili zápis `konfiguracia[:] = novaKonfiguracia(konfiguracia)`, vďaka ktorému môžeme starú generáciu prepísať novou bez straty pôvodného ukazovateľa na zoznam. Použitím rezu sme všetky hodnoty vlastne prepísali hodnotami novými.

Keby sme použili zápis `konfiguracia = novaKonfiguracia(konfiguracia)` tak by vznikla nová lokálna premenná s názvom `konfiguracia`, do ktorej by sme sa snažili priradiť výsledok aktuálnej iterácie.

Zápis `konfiguracia[:]` sa nazýva rez (slice). Rez `[zaciatok:koniec]` určuje indexy prvkov, ktoré sa majú skopírovať z pôvodného zoznamu. Výsledkom rezu je úplne nový zoznam, pôvodný ostáva nezmenený.

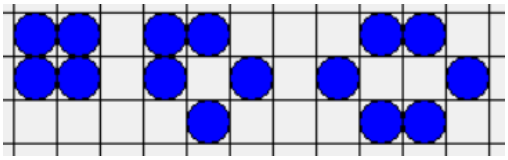
```
n = [1,2,3,4,5,6,7,8,9,10]
n[0:3] >> [1, 2, 3]
n[4:5] >> [5]
n[5:]  >> [6, 7, 8, 9, 10]
n[:5]  >> [1, 2, 3, 4, 5]
```

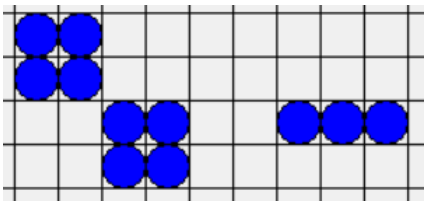
Rez je možné použiť aj na ľavej strane príkazu priradenia. V tomto prípade hovoríme, ktorá časť zoznamu sa má nahradiť hodnotou na pravej strane príkazu priradenia.

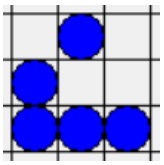
```
n[2:5] = [] >> [1, 2, 6, 7, 8, 9, 10]
n[:5] = [0] >>  [0, 9, 10]
```

Úloha 4

Experimentujte s inicializačnou generáciou a skúste identifikovať správanie sa nasledujúcich konfigurácií. Skúste ich pomenovať, prípadne hľadať štruktúry s rovnakým správaním.

a) 

b) 

c) 

Ďalšie úlohy na precvičenie a zamyslenie

1. Umožnite uloženie prvých N generácií aktuálnej konfigurácie do textového súboru.
2. Navrhnite vlastnú prechodovú funkciu a experimentujte s rôznymi inicializačnými konfiguráciami. Dokážete v novej v simulácii identifikovať stabilné spoločenstvo, oscilátory alebo lode?

5. Hra Život alebo každý sám za seba

Kľúčové slová

objektovo orientované programovanie, objekt, trieda, metóda, atribút, zapuzdrenie

Čo sa naučíme a čo si precvičíme

- objektovo pristupovať k riešeniu problémov,
- aplikovať základy objektovo orientovaného programovania pri riešení problémov,
- analyzovať objekty reálneho sveta a navrhovať zodpovedajúce triedy.

Problémová situácia

V predchádzajúcej kapitole sme simulovali život buniek (26), ktoré žijú v obmedzenom priestore a ich život sa riadi jednoduchými pravidlami. Keď sa zamyslíme nad tým, ako sme jednotlivé bunky, ich vlastnosti a správanie simulovali vo virtuálnom svete, uvedomíme si, že bunky boli dosť pasívne:

- hlavný program im povedal, kde sa majú zobraziť, resp. ich tam priamo zobrazil,
- hlavný program im na začiatku povedal, či sú živé alebo nie,
- hlavný program zistil, koľko živých susedov bunka má,
- a nakoniec hlavný program rozhodol, či sa stav bunky v nasledujúcej generácii zmení alebo nie.

Pre potreby zobrazenia takto jednoduchej simulácie to možno stačí. Ak ale začneme uvažovať nejaké náročnejšie simulácie zistíme, že riešiť to týmto spôsobom je prakticky nemožné. Ak by sme sa pokúsili na jednom mieste zabezpečiť celú funkcionalitu simulácie, vytvoríme obrovský kus ťažko zrozumiteľného kódu náchylného na chyby. Takto sa to v skutočnosti nerobí.

Uvažujme trochu inak. Bunky v reálnom svete tvoria relatívne samostatné entity. Bunka existuje v nejakom životnom priestore, dokáže komunikovať/interagovať s bunkami v jej okolí. Ak jej dôjde zdroj energie zahynie a ak sa vyskytnú vhodné životné podmienky, „narodí“ sa nová bunka.

Poznámka na okraj

Samozrejme vieme, že nové bunky sa nerodia, ale vznikajú delením. V našom modeli sme si umieranie a rodenie nových buniek trochu zjednodušili.

Pozrime sa na výsledný program ako na systém objektov, ktoré vzájomne komunikujú. Implementujme ekvivalentný model správania sa skutočných buniek aj do virtuálneho sveta. Dajme im viac samostatnosti.

Objektovo orientovaný prístup k riešeniu problémov

Výzvu, ktorú sme si dali na konci predchádzajúceho odseku, vieme dosiahnuť pomocou objektovo orientovaného prístupu. Pri tomto prístupe sa snažíme objekty reálneho sveta nahradiť objektmi virtuálneho sveta. Vytvárame virtuálne objekty - modely, ktorých vlastnosti

(hodnoty a správanie sa) sú ekvivalentom vlastností (hodnoty a správanie sa) reálnych objektov. Napriek tomu, že predchádzajúca veta znie príliš komplikovane, skutočnosť je jednoduchšia.

To, aké vlastnosti má objekt mať, aké hodnoty si bude uchovávať a ako sa má správať, definujeme pomocou tried. Triedu si môžeme predstaviť ako nejakú formičku alebo plánik, pomocou ktorého objekty vytvárame. Napriek tomu, že trieda je len „formičkou“, poskytuje nám pomerne dobrú predstavu o tom, ako budú objekty vytvorené pomocou nej vyzerať alebo ako sa budú správať. Pre lepšie pochopenie si pozrime nasledujúce príklady:

<table>
<tr><th>trieda</th><th>pomocou tried vytvárame objekty</th><th>objekt</th></tr>
<tr><td>

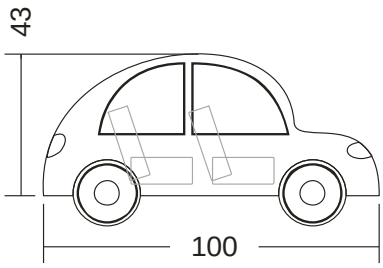
```
trieda Formička:
```

</td><td>Pomocou formičiek vytvárame koláčiky z piesku. Aj keď možno nebudú všetky koláčiky z jednej formičky rovnaké, vieme si vopred predstaviť ako budú vyzerať.</td><td>

```
koláčik
```

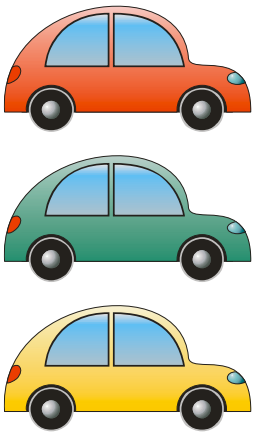
</td></tr>
<tr><td>

```
trieda OsobneAuto:
```



</td><td>Pomocou plánu auta, vieme auto zostrojiť. Zostrojené autá majú niečo spoločné: napr. 4 kolesá a v niečom sa líšia: napr. farba karosérie.</td><td>

```
auto
```

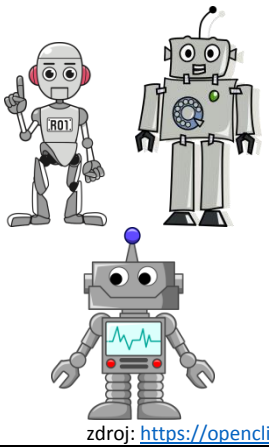


</td></tr>
<tr><td>

```
trieda Robot:
    má:
        hlavu
        trup
        dve ruky
        dve nohy
    vie:
        pozdraviť mávaním
        chodiť
```

</td><td>Pomocou popisu vieme robota zostrojiť. Roboti majú niečo spoločné, napr. počet a typ končatín, a v niečom sa líšia, napr. dĺžkou končatín, rýchlosťou chôdze.</td><td>

```
robot
```



zdroj: https://openclipart.org

</td></tr>
</table>

S objektmi, ktoré niekto na základe definovaných tried vytvoril, už vieme manipulovať. Koláčik z piesku vieme niekam premiestniť. Auto má kolesá – môžeme ho postrčiť a ono prejde nejakú dráhu. Robot vie odzdraviť zamávaním rukami a odkráčať niekam preč. Prínosom objektového prístupu je fakt, že ak už sú objekty vytvorené, môžeme využívať ich funkcionalitu a nemusíme sa zapodievať tým, ako je realizovaná. Netrápi nás, ako si auto pamätá, že je červené, ako otáčať kolesami auta, alebo ako prepletať nohami robota, aby kráčal. Všimnime si aj to, že pomocou jednej triedy vieme vyrobiť viac objektov.

Objekty a triedy v Pythone

Pozrime sa teraz na to, ako v Pythone definovať triedy a ako v Pythone vytvárať objekty.

trieda	pomocou triedy vytvárame objekty	objekt
`class Trieda():` `  pass`	`objekt1 = Trieda()` `objekt2 = Trieda()`	`objekt1` `objekt2`

Výkladový text

Na vytvorenie triedy používame kľúčové slovo `class`. Za ním uvádzame názov triedy so zátvorkami. Názvy tried píšeme so začiatočným veľkým písmenom (taká je všeobecná dohoda). Príkaz `pass` sme použili len preto, aby sme mohli vytvoriť prázdnu definíciu triedy.

Pri vytváraní objektov je výhodné zapamätať si referenciu na novovytvorený objekt (`objekt1 = Trieda()`), aby sme neskôr dokázali tento objekt osloviť a manipulovať s ním. Objekt chápeme ako kontajner, ktorý v sebe uchováva hodnoty a nástroje, ako s týmito hodnotami pracovať.

V ďalšom texte budeme:

triedu chápať ako šablónu pre vytváranie objektov,

objekt ako konkrétny prípad typu, ktorý je definovaný triedou.

Nikoho asi neprekvapí, že objekty nami definovanej triedy `Trieda()` toho veľa nevedia. Prakticky sú nám takéto objekty nanič.

Metódy a atribúty objektov

Navrhnime triedu pre zmysluplnejšie objekty. Definujme triedu, ktorá nám umožní vytvárať virtuálnych škriatkov, ktorí si niečo pamätajú a majú nejakú funkcionalitu. Na začiatok stačí, nech škriatok vie pozdraviť.

```
# skriatok1.py
class Skriatok():
    '''Reprezentácia virtuálneho škriatka'''
    def pozdrav_sa(self):                                    #[1]
        ''' Škriatok vypíše pozdrav'''
        print('Ahoj')
```

```
mato = Skriatok()
mato.pozdrav_sa()  # Ahoj                                    #[2]
```

Výkladový text

Ak chceme objektu pridať nejakú funkcionalitu, pridáme do definície triedy zodpovedajúcu funkciu. Takejto funkcii vravíme **metóda**. Všimnime si parameter `self` (jeho názov je výsledkom všeobecnej dohody a mali by sme túto konvenciu dodržiavať). Tento parameter je povinný pri definícii metódy [1]. Pri volaní metódy tento argument neuvádzame [2]. Doplní ho Python automaticky. Jeho hodnotou je referencia na objekt, ktorého metódu sme zavolali. Ako to využiť si ukážeme v ďalšej časti tejto kapitoly.

Ak chceme, aby objekt metódu vykonal, za menom objektu uvedieme jej názov [2]. Názov objektu a názov metódy oddelíme znakom bodka.

Všimnime si aj ďalší dôležitý prvok. Okrem metód aj triedy majú svoje dokumentačné reťazce.

Z predchádzajúceho textu vieme, že pomocou jednej triedy vieme vytvoriť viac objektov. Vytvorme dvoch škriatkov tak, aby sa každý z nich vedel pozdraviť a predstaviť svojim menom. Každý škriatok si teda musí pamätať svoje meno. Najšikovnejšie bude, ak škriatkovi jeho meno povieme v okamihu, keď ho vytvárame.

```
# skriatok2.py
class Skriatok():
    '''Reprezentácia virtuálneho škriatka'''
    def __init__(self, meno):                                #[1]
        '''Vytvorí škriatka a nastaví mu meno.

        :param meno: meno škriatka
        :type meno: str
        '''
        self.meno = meno

    def pozdrav_sa(self):
        ''' Škriatok sa pozdraví a predstaví svojim menom'''
        print(f'Ahoj, ja som škriatok {self.meno}.')


martin = Skriatok('Maťo')                                    #[2]
martin.pozdrav_sa()  # Ahoj, ja som škriatok Maťo.
amalia = Skriatok('Amálka')                                  #[3]
amalia.pozdrav_sa()  # Ahoj, ja som škriatok Amálka.
```

Výkladový text

Do definície triedy pribudla špeciálna metóda `__init__()`. Túto metódu nemusíme volať. Vďaka jej menu sa o jej volanie postará Python automaticky ihneď po vytvorení objektu. Ak má metóda `__init__()` nejaké parametre [1] (okrem `self`), ich hodnoty uvedieme pri vytváraní objektu [2, 3]. Keďže táto metóda slúži na počiatočné nastavenie hodnôt niektorých

premenných objektu (v tomto prípade mena škriatka), nazýva sa aj inicializačná metóda. V metóde `__init__()` neuvádzame príkaz `return`.

Po vytvorení objektu si objekt uloží zadané meno do premennej `self.meno`. Prefixom `self` hovoríme, že toto je premenná objektu – inštančná premenná. Premennú objektu nazývame **atribút**.. Keďže `self` odkazuje na objekt samotný, každý objekt má v tejto premennej uloženú svoju vlastnú hodnotu. Inštančné premenné majú aj ďalšiu užitočnú vlastnosť. Existujú počas celej existencie objektu a nie len počas behu metódy, kde boli vytvorené.

Úloha 1

Navrhnite triedu `Ziak()`, ktorej objekty môžeme využiť na reprezentáciu žiaka. Každý žiak si pamätá svoje meno a vek a vie sa predstaviť uvedením svojho mena a veku.

Vytvorte dva objekty reprezentujúce žiakov: Kamil, 15 rokov a Marcela, 16 rokov. Overte funkčnosť triedy tak, že sa obaja predstavia.

Poznámka na okraj

Dokumentačné reťazce, okrem tých, ktoré sa vyskytnú prvý krát, v ďalších ukážkach kódov neuvádzame. Zápisy by sa zbytočne predlžovali a funkciu dokumentačných reťazcov plní samotný výkladový text. V zdrojových (*.py) súboroch je dokumentácia uvedená.

Aj keď si to neuvedomujeme, s objektmi v Pythone pracujeme už od začiatku. Všetky hodnoty, s ktorými sme doposiaľ pracovali, boli uchovávané v objektoch nejakých tried. Presvedčme sa o tom:

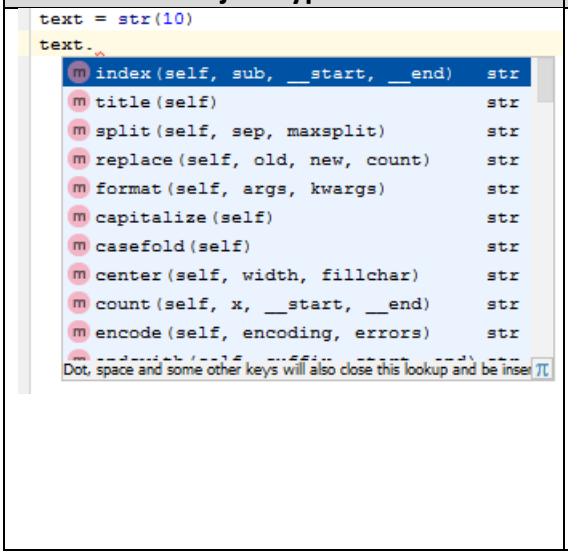
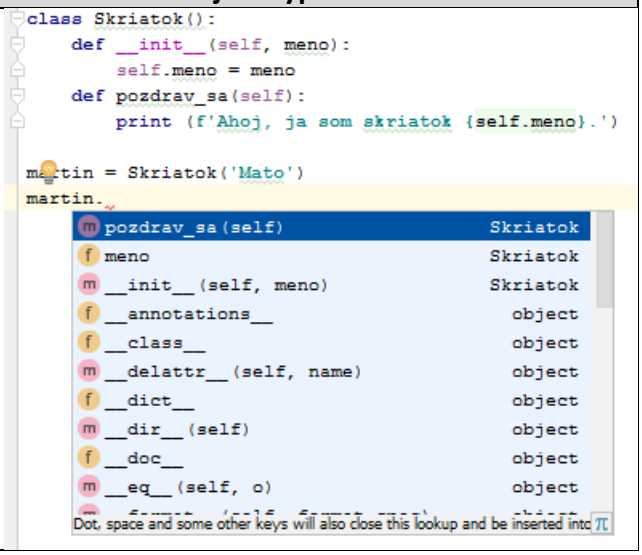
```
#skriatok3.py
class Skriatok():
    def __init__(self, meno):
        self.meno = meno

    def pozdrav_sa(self):
        print(f'Ahoj, ja som škriatok {self.meno}.')

martin = Skriatok('Mato')
print(type(martin))          #<class '__main__.Skriatok'>          #[1]

text = 'martin'
print(type(text))            #<class 'str'>                        #[2]
```

[1] Ak sa spýtame, akého typu je `martin`, odpoveď môžeme interpretovať aj tak, že je to objekt, ktorý vznikol z triedy `Skriatok()`.
[2] Ak sa spýtame, akého typu je `text`, odpoveď môžeme interpretovať aj tak, že je to objekt, ktorý vznikol z triedy `str()`.

Ak aplikujeme objektový prístup k riešeniu problémov, zmení sa aj naše uvažovanie. Pri procedurálnom programovaní (keď nepoužívame objektový prístup) sme uvažovali v zmysle, čo budeme s dátami robiť. Pri objektovo orientovanom prístupe uvažujeme v zmysle, čo by

mohol objekt urobiť pre nás. V Pythone sme z faktu, že všetky hodnoty sú objekty nejakých tried, doteraz len profitovali.

Pozrime sa, čo pre nás môže urobiť reťazec a čo pre nás môže urobiť náš škriatok. Niektoré vývojové prostredia (napr. PyCharm) nám vedia pri písaní kódu ponúknuť dostupné metódy a premenné objektu a uľahčiť nám tým písanie kódu.

objekt typu str	objekt typu Skriatok
	

Kým reťazec má pre nás širokú ponuku metód, ktoré môžeme použiť, škriatok je na tom omnoho skromnejšie.

Poznámka na okraj

Metódy/premenné začínajúce a končiace dvoma podčiarkovníkmi si zatiaľ nevšímajme. Sú to špeciálne metódy/premenné a o ich volanie sa stará Python. O jednej z nich, metóde `__init__()`, už niečo vieme.

Napriek tomu, že sme tieto metódy nedefinovali, objekty ich majú k dispozícii. Prečo je to tak, si povieme v niektorej z nasledujúcich kapitol.

Všimnime si, že škriatok `martin` má okrem metódy `pozdrav_sa()` dostupnú aj premennú `meno`. Využiť to môžeme napr. takto:

```
#skriatok4.py
class Skriatok():
    def __init__(self, meno):
        self.meno = meno

    def pozdrav_sa(self):
        print(f'Ahoj, ja som škriatok {self.meno}.')

martin = Skriatok('Maťo')
martin.pozdrav_sa()          #Ahoj, ja som škriatok Maťo.
print(martin.meno)           #Maťo                          #[1]
martin.meno = 'Martinko'                                    #[2]
martin.pozdrav_sa()          #Ahoj, ja som škriatok Martinko.
```

[1] K hodnote premennej objektu vieme pristupovať. Stačí ak vieme jej názov.
[2] Hodnotu premennej objektu vieme zmeniť tak, že jej priradíme novú hodnotu.

Úloha 2

Bez toho, aby ste menili už napísaný kód programu z úlohy 1, doplňte kód nasledovne:

- Marcela by sa rada predstavovala menom Marcelka.
- Kamil už má 16 rokov

Zapúzdrenie (encapsulation)

Priamy prístup k premennej objektu môže byť rizikový v tom, že objektovej premennej priradíme nevhodnú hodnotu. Je preto výhodnejšie, aby sme v triede definovali funkciu, pomocou ktorej budeme vedieť hodnotu objektovej premennej priradiť. Táto funkcia by mala okrem samotnej zmeny hodnoty aj skontrolovať, či nová hodnota je platnou hodnotou pre túto premennú.

Upravme triedu `Skriatok()` tak, že doplníme metódu na zmenu mena. Určime si jednoduchú podmienku pre meno: meno škriatka by nemalo byť kratšie ako 3 znaky.

```
# skriatok5.py
class Skriatok():
    def __init__(self, meno):
        self.set_meno(meno)

    def set_meno(self, meno):
        '''Nastaví meno škriatkovi. Dĺžka mena sú aspoň 3 znaky

        :param meno: meno škriatka
        :type meno: str
        :raise ValueError: ak dĺžka mena je menej 3 znaky
        '''
        if len(meno) < 3:
            raise ValueError("Príliš krátke meno")
        self.meno = meno

    def pozdrav_sa(self):
        ''' Škriatok sa pozdraví a predstaví svojim menom'''
        print(f'Ahoj, ja som škriatok {self.meno}.')
```

`martin = Skriatok('Mato')`
`martin.set_meno('Au')`
`martin.pozdrav_sa()`
	`martin = Skriatok('Mato')`
	`martin.set_meno('Martinko')`
	`martin.pozdrav_sa()`

ValueError: Príliš krátke meno: Au Process finished with exit code 1	Ahoj, ja som škriatok Martinko. Process finished with exit code 0

Všimnime si, že aj v metóde `__init__()` sme použili metódu `set_meno()` na nastavenie mena škriatka. Takto docielime, že už pri vytváraní objektu sa kontroluje, či meno má platnú hodnotu.

Výkladový text

Pre funkcie, ktoré nastavujú hodnotu niektorej z objektových premenných, sa používa názov zložený z prefixu `set_` a názvu premennej. Už z názvu funkcie vieme rýchlo zistiť, čo robí. Takejto metóde hovoríme **setter** .

Všimnime si aj reakciu metódy `set_meno()` na nekorektnú hodnotu mena. Funkcia generuje výnimku. Týmto sme zabezpečili, že nemôže vzniknúť nekorektný objekt – škriatok s príliš krátkym menom.

Úloha 3

Triedu `Ziak()` z predchádzajúcej úlohy chceme použiť v 1. B na evidenciu žiakov. Pridajte do triedy `Ziak()` metódy pre zmenu mena a veku žiaka. Aby sme náhodou nevytvorili objekt s chybnými hodnotami, dodržme tieto podmienky.

- Meno žiaka môže byť len niektoré zo zoznamu žiakov 1. B: Alexandra, Karina, Daniela, Drahoslav, Andrea, Antónia, Bohuslava, Severín, Alexej, Dáša, Malvína, Ernest, Rastislav, Radovan.
- Vek žiaka je minimálne 15 a maximálne 16 rokov.

Tým, že sme definovali metódu `set_meno()` nezabránime tomu, aby niekto zmenil meno spôsobom `martin.meno = 'Au'`.

Tvorcovia objektov v Pythone sa spoliehajú na to, že iní programátori využívajúci ich triedy, budú využívať dostupné metódy na zmeny hodnôt objektových premenných.

Programátori sa zase spoliehajú na to, že dostupné metódy objektov ustrážia nevhodné hodnoty. Ak objekt umožňuje priamu zmenu, malo by to byť bezpečné.

Jedni aj druhí však robia chyby. Ak môžeme chybám vopred zabrániť, urobme to.

Výkladový text

Ak z nejakého dôvodu potrebujeme zabezpečiť, aby sa k nejakej vlastnosti objektu (atribút alebo metóda) nedalo pristupovať zvonka objektu, zabezpečíme to tak, že názov identifikátora tejto vlastnosti začne dvojitým podčiarkovníkom `__`. Podľa konvencie sa takéto vlastnosti chápu ak súkromné (privátne) vlastnosti objektu a nemalo by sa k nim pristupovať mimo objektu.

```
#skriatok6
class Skriatok():
    def __init__(self, meno):
        self.set_meno(meno)

    def set_meno(self, meno):
        if len(meno) < 3:
            raise ValueError(f'Príliš krátke meno: {meno}')
```

```
        self.__meno = meno

    def pozdrav_sa(self):
        print (f'Ahoj, ja som škriatok {self.__meno}.')
```

`martin = Skriatok('Maťo')`
`martin.set_meno('Au')`
`martin.pozdrav_sa()`
	`martin = Skriatok('Maťo')`
	`martin.__meno = 'Au'`
	`martin.pozdrav_sa()`

ValueError: Príliš krátke meno: Au Process finished with exit code 1	Ahoj, ja som škriatok Maťo Process finished with exit code 0

Úloha 4

Upravte triedu `Ziak()` z úlohy 3 tak, aby nebolo možné zmeniť hodnotu mena a veku priamym prístupom k premennej objektu.

Spravili sme niekoľko úprav triedy `Skriatok()`.

Na začiatku sme mohli pohodlne pristupovať k menu škriatka a mohli sme ho meniť. Malo to však nevýhodu v tom, že sme mohli škriatkovi priradiť aj nevhodné meno:

```
martin = Skriatok('Maťo')
print(martin.meno)      #Maťo
martin.meno = 'Au'
print(martin.meno)      #Au
```

Potom sme triedu `Skriatok()` upravili tak, aby nebolo možné priradiť nevhodnú hodnotu do mena škriatka. Už sme však nemohli k menu pristupovať tak jednoducho:

```
martin = Skriatok('Mato')
martin.__meno = 'Au'          #premenna __meno nie je zvonka viditeľná
martin.set_meno('Martinko')
martin.pozdrav_sa()           #Ahoj, ja som škriatok Martinko.
```

Python našťastie ponúka možnosť, ako spojiť výhody oboch prístupov a zároveň obísť ich nevýhody.

```
# skriatok7.py
class Skriatok():
    def __init__(self, meno):
        self.meno = meno                                        #[1]

    def set_meno(self, meno):                                   #[2]
        if len(meno) < 3:
            raise ValueError(f'Príliš krátke meno: {meno}')
        self.__meno = meno

    def get_meno(self):                                         #[3]
        '''Vráti meno škriatka

        :return: meno škriatka
```

```
        :rtype: str
        '''
        return self.__meno

    meno = property(get_meno, set_meno)                    #[4]

    def pozdrav_sa(self):
        print(f'Ahoj, ja som škriatok {self.meno}.')       #[5]
```

```
martin = Skriatok('Maťo')
martin.meno = 'Martinko'    #[6]
martin.pozdrav_sa()
print(martin.meno)
```

```
Ahoj, ja som škriatok Martinko.
Martinko

Process finished with exit code 0
```

```
martin = Skriatok('Maťo')
martin.meno = 'Au'
#[6]
martin.pozdrav_sa()
print(martin.meno)
```

```
ValueError: Príliš krátke meno: Au

Process finished with exit code 1
```

Meno škriatka je uložené v súkromnej objektovej premennej `__meno`. Na zmenu hodnoty mena sme definovali metódu `set_meno()` [2], ktorá zároveň kontroluje, či zadané meno je korektné. Ak potrebujeme zistiť meno škriatka, využijeme na to metódu `get_meno()` [3]. Nikde inde k tejto premennej takto priamo nepristupujeme.

Výkladový text

Pre funkcie, ktoré vracajú hodnotu niektorej z objektových premenných, sa používa názov zložený s prefixu get_ a názvu premennej. Už z názvu funkcie vieme rýchlo zistiť, čo robí. Takejto metóde hovoríme **getter**. (27)

Výkladový text

Najzaujímavejšia časť kódu je v riadku [4]. Týmto riadkom zabezpečíme:

- pri požiadavke na získanie hodnoty premennej `meno` [5] Python zavolá funkciu `get_meno()`,
- pri požiadavke na zmenu hodnoty premennej `meno` [6] Python zavolá funkciu `set_meno()`,
- všimnime si, že aj v inicializačnej metóde [1] a v metóde `pozdrav_sa ()` [5] sme na prístup k menu použili krátky zápis `self.meno`.

Príkazom `meno = property(get_meno, set_meno)` (28) sme vytvorili nový atribút `meno`. Keď k nemu chce niekto pristupovať, Python zavolá príslušnú funkciu. Prvým parametrom v zátvorke je meno funkcie, ktorá sa zavolá pri požiadavke na prístup k hodnote

tohto atribútu. Druhým parametrom je meno funkcie, ktorá sa zavolá pri požiadavke na zmenu hodnoty atribútu. Ak nechceme poskytnúť možnosť zmeny niektorého z atribútov, môžeme druhý parameter vynechať.

S uvedenou úpravou by sme mohli byť spokojní. Vždy keď pristupujeme k atribútu `meno`, zavolá sa príslušná funkcia. Teraz však máme dva spôsoby ako pristupovať k objektovej premennej `__meno`:

```
martin.meno = 'Martinko'
```

alebo

```
martin.set_meno('Martinko')
```

a

```
print(martin.meno)
```

alebo

```
print(martin.get_meno())
```

Tento fakt nie je v súlade s filozofiou Pythonu (29): *Mal by existovať jeden - a najlepšie len jeden - zrejmý spôsob, ako niečo urobiť.* Navyše takýto prístup mätie ostatných programátorov, ktorí by sa rozhodli našu triedu používať: *„Ak sú dva spôsoby ako meniť meno, asi na to bude dôvod. A ak sú dva, v čom sa líšia? Kedy mám ktorý z nich použiť?“*

Upravme teda našu triedu tak, aby poskytovala len jeden spôsob ako k menu pristupovať. Schovajme obslužné funkcie, aby boli z vonka objektu neviditeľné.

```
#skriatok8.py
class Skriatok():
    def __init__(self, meno):
        self.meno = meno

    def __set_meno(self, meno):
        if len(meno) < 3:
            raise ValueError(f'Príliš krátke meno: {meno}')
        self.__meno = meno

    def __get_meno(self):
        return self.__meno

    meno = property(__get_meno, __set_meno)

    def pozdrav_sa(self):
        print(f'Ahoj, ja som škriatok {self.meno}.')
```

Výkladový text

Postupnými úvahami a úpravami sme navrhli triedu, ktorá zabalila atribúty a metódy do jedného celku. K manipulácii s atribútmi sme vytvorili obslužné metódy. Pomocou nich sme zamedzili tomu, aby niektorý z atribútov nadobudol neplatnú hodnotu. Zároveň sme

znemožnili priamy prístup k atribútom. Dobre navrhnutá trieda by nemala umožniť vytvoriť chybný objekt (objekt s chybnou hodnotou) a narušiť tak jeho vnútornú konzistenciu.

Tieto vlastnosti patria medzi základné vlastnosti objektovo orientovaného programovania a súhrne sa označujú pojmom **zapuzdrenie** (Encapsulation).

Úloha 5

Upravte triedu `Ziak()` z predchádzajúcej úlohy tak, aby sme:

- meno žiaka mohli zistiť len skráteným zápisom,
- meno žiaka mohli nastaviť len na začiatku (pri vytváraní objektu) a neskôr ho už nemohli meniť,
- vek žiaka vedeli zistiť a zmeniť len skráteným zápisom,
- pre meno a vek žiaka dodržali rovnaké obmedzenia ako v predchádzajúcej úlohe.

Úloha 7

Využitím OOP implementujte Hru život. Predpokladá sa grafický výstup s možnosťou krokovania na nasledujúcu generáciu buniek.

Pomôcka:

- Navrhnite vhodnú triedu `Bunka()`, ktorá bude modelom bunky z Hry život.
- Objekty triedy `Bunka()` by mali byť relatívne samostatné, t. j. mali by vedieť:
  - sa zobraziť v nejakej grafickej ploche, teda musia vedieť kde žijú,
  - či sú živé alebo nie a podľa toho meniť svoje zobrazenie,
  - zistiť, ako sa zmení ich stav v nasledujúcej generácii, teda musia vedieť osloviť svojich susedov aby zistili, koľkí z nich sú živé bunky,
  - zmeniť svoj stav, ak kolónia buniek prechádza do ďalšej generácie.
- Metódy grafického rozhrania by mali:
  - vytvoriť priestor pre život buniek,
  - vytvoriť počiatočnú generáciu buniek,
  - generovať nasledujúcu generáciu buniek.

Úloha 8

Premyslite si, ako efektívnejšie implementovať metódu `pocet_zivych_susedov()` v triede `Bunka()`.

Čo sme sa naučili

- využívať objektový prístup k riešeniu problémov,
- navrhovať jednoduché triedy, ktoré môžu byť modelmi reálnych objektov,
- navrhovať triedy tak, aby sme nedovolili vytvoriť chybný objekt,
- vytvárať objekty a pomocou ich metód meniť hodnoty ich atribútov.

Ďalšie úlohy na precvičenie a zamyslenie

1. Navrhnite triedu `BankovyUcet()`, ktorej objekty môžeme využiť na správu účtu v banke. Každý účet by mal vedieť povedať, kto je jeho majiteľom a aký je zostatok na účte. Okrem toho by mal účet umožniť vklad na účet a výber z účtu. Nie je možné vybrať viac, ako je na účte a meno majiteľa musí byť neprázdny reťazec znakov. Meno majiteľa účtu nie je možné meniť. Správnosť návrhu triedy si overte.

2. Navrhnite triedu `RodneCislo()`, ktorá bude reprezentovať rodné číslo človeka. Pre rodné číslo platia isté pravidlá (malo by byť deliteľné 11, korektný dátum, ..). Trieda by teda nemala umožniť vytvoriť objekt s nekorektným rodným číslom. Z rodného čísla dokážeme zistiť deň, mesiac a rok narodenia a pohlavie človeka. Doplňte triedu tak, aby jej objekty dokázali tieto informácie poskytnúť.

   Pomôcka1: Skontrolovať korektnosť zadaného dátumu je možné využitím modulu `datetime` a jeho triedy `datetime()`. Ak sa pokúsime vytvoriť neexistujúci dátum, trieda vyhodí výnimku.

   Pomôcka2: Užitočnou by bola metóda, ktorá vráti celé rodné číslo, napr. pre potreby výpisu. Toto vieme zabezpečiť metódou `__str__()`, ktorá vracia textovú reprezentáciu objektu, v tomto prípade rodného čísla. O jej volanie sa nemusíme starať. Python ju zavolá automaticky v situáciu, ak sa požaduje textová reprezentácia objektu. To čo musíme spraviť je, definovať jej telo.

6. Ako horí les

Kľúčové slová

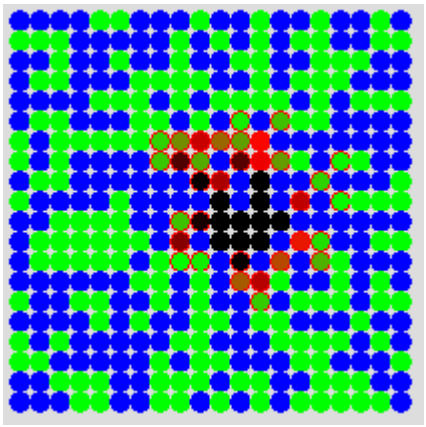
model, simulácia, celulárny automat, prechodová funkcia, horenie lesa, požiar, analýza reálnej situácie,

Čo sa naučíme a čo si precvičíme

- navrhovať objekty, ich metódy a atribúty,
- navrhovať metódy tak, aby objekty vedeli vzájomne komunikovať,
- analyzovať reálne situácie a deje,
- určiť podstatné a zanedbať nepodstatné faktory resp. prvky pre potreby simulácie,
- navrhnúť model reálnej situácie, resp. deja,
- definovať pravidlá správania sa elementov modelu,
- implementovať model a simulovať jeho správanie.

Problémová situácia

V predchádzajúcej kapitole sme si vytvorili jednoduchú simuláciu Hry život. Bunky sa v tejto simulácii správali podľa vopred jasných a jednoduchých pravidiel. Ich stav sa dal vopred predikovať. Vedeli by sme simulovať aj komplexnejšie deje?



Vytvorme simuláciu, v ktorej budeme sledovať, ako sa v lese šíri požiar. Na rozdiel od predchádzajúcej simulácie si definujeme vlastné pravidlá správania sa požiaru v lese. Tieto pravidlá by mali zodpovedať tomu, ako sa správa oheň, resp. horiaci objekt.

Les je pomerne komplikovaná štruktúra a tak sa nevyhneme tomu, že niektoré faktory budeme musieť redukovať, prípadne úplne zanedbať.

Naše uvažovanie smerujme k aplikácii, v ktorej vygenerujeme počiatočnú konfiguráciu lesa. Aké parametre lesa budeme môcť vopred nastaviť záleží len od nás, resp. od návrhu nášho modelu. Po vygenerovaní nultej generácie lesa zapálime jeden alebo niekoľko vybraných stromov (buniek). Následne budeme sledovať ako sa požiar v lese šíri.

Návrh modelu horenia lesa

Prvky modelu šírenia sa požiaru v lese

Poznámka na okraj

Postupne budeme navrhovať model lesa a šírenia požiaru v lese. Takýchto modelov môže byť viac a môžu byť navzájom rôzne. Ako si nakoniec výsledný model upravíte, záleží len na vás. Inšpirovať sa môžete napr. na (30) (31) (32).

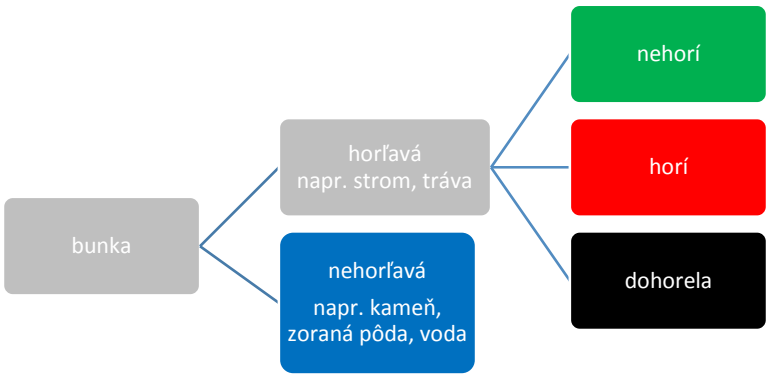
V lese nájdeme množstvo objektov (kamene, stromy, tráva, zvieratá, voda, pôda, počasie, terén ...) medzi ktorými navyše existujú vzájomné vzťahy. Niektoré objekty významne ovplyvňujú spôsob šírenia požiaru, iné mierne a niektoré vôbec.

Zamyslime sa

Ktoré faktory a objekty vplývajú na šírenie ohňa v lese. Ktoré sú podstatné a ktoré naopak nie? Ktoré zanedbáme a ktoré budeme brať do úvahy?

Pre naše potreby vytvoríme jednoduchý model lesa, v ktorom budeme uvažovať **horľavé prvky** (stromy, tráva ..) ako zdroj paliva pre požiar a **nehorľavé prvky** (kamene, voda ...) ktoré nehoria a neumožnia prechod požiaru. Rovnako uvažujme aj **vietor a jeho smer**, ktorý ovplyvňuje smer a rýchlosť šírenia požiaru.

Model lesa budeme simulovať celulárnym automatom. Každá bunka v našom modeli predstavuje jednu časť lesa. V akom stave sa táto bunka môže nachádzať? Jedna z možných odpovedí je na nasledujúcom obrázku.



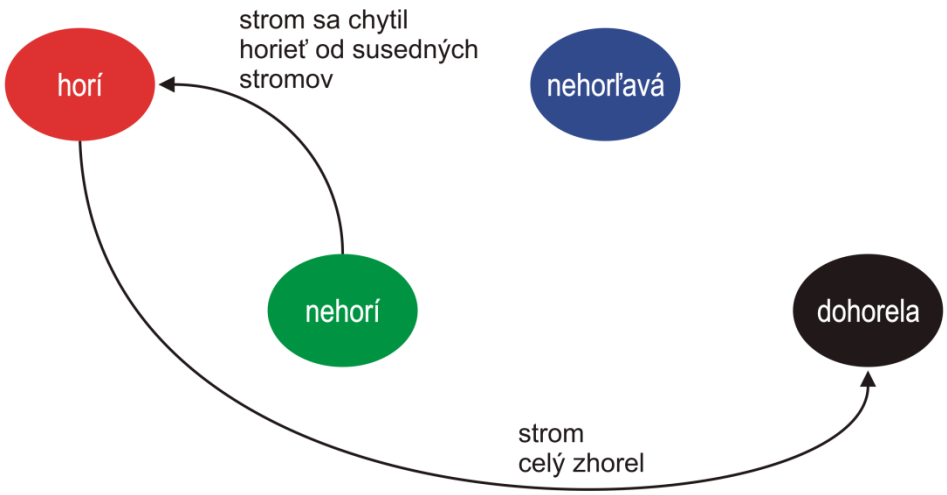
Obrázok 21 Možné stavy buniek v modeli lesa

Správanie sa prvkov modelu šírenia sa požiaru v lese

Definovali sme základné stavy bunky: **nehorľavá**, **nehorí**, **horí** a **dohorela**. V našom modeli budeme simulovať prechody medzi týmito stavmi.

Medzi ktorými stavmi je možný prechod? Ako by sme tento prechod interpretovali v reálnom lese? Uvažujme prechody **prirodzené**, ktoré môžu nastať bez zásahu človeka a prechody **umelé**, pri ktorých musí zasiahnuť človek.

Prirodzené prechody (pozri obrázok nižšie) sa budú realizovať na základe nejakej pravdepodobnosti alebo na základe definovanej prechodovej funkcie.

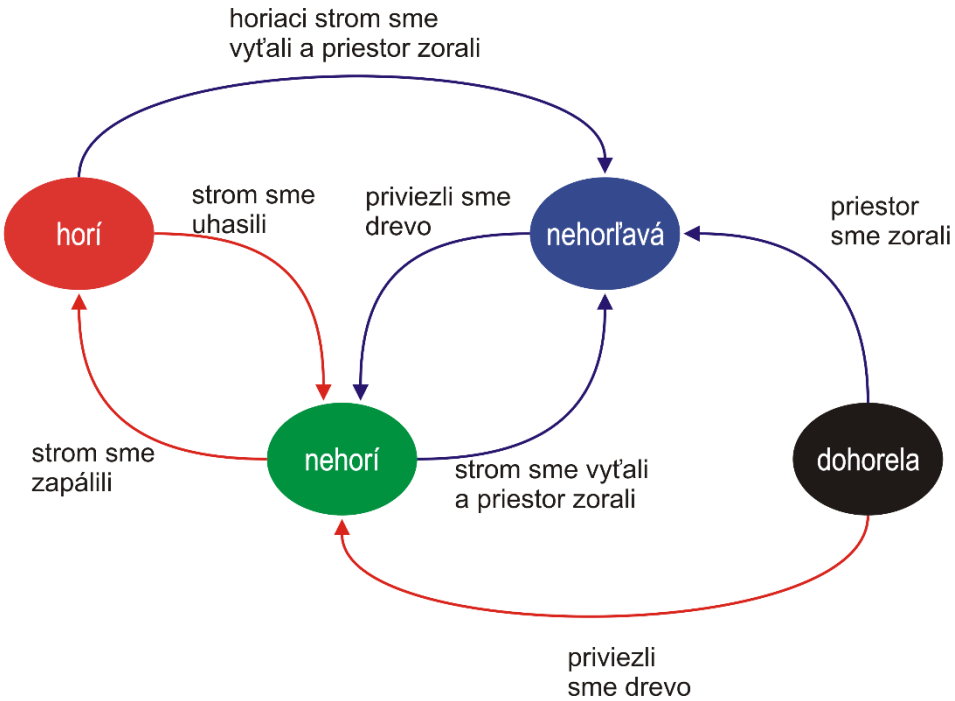


Obrázok 22 Prirodzené prechody medzi stavmi buniek

Všimnime si niekoľko dôsledkov, ak by sme aplikovali len prirodzené prechody:

- ak bunka na začiatku bola nehorľavá, svoj stav prirodzeným prechodom nezmení,
- ak v modeli nebude na začiatku žiadna horiaca bunka, oheň na žiadnom mieste nevznikne
- horiaca bunka skončí v stave dohorela.

Umelé prechody (pozri obrázok nižšie) medzi stavmi budeme realizovať priamym zásahom používateľa do prostredia virtuálneho lesa. Ak necháme simuláciu bežať bez zásahu používateľa, žiaden z umelých prechodov sa neuskutoční.



Obrázok 23 Umelé prechody medzi stavmi buniek

Umelé prechody podstatne zvyšujú variabilitu správania sa požiaru v lese. Požiar môžeme inicializovať, zahasiť alebo zabrániť jeho šíreniu. Na mieste kadiaľ už požiar prešiel (a všetko zhorelo) môžeme založiť nový požiar apod.

Všimnime si dva rôzne typy prechodov:

- prechody buniek medzi rôznymi horľavými stavmi – červené šípky
- prechody buniek medzi niektorým z horľavých stavov a nehorľavým stavom - modré šípky.

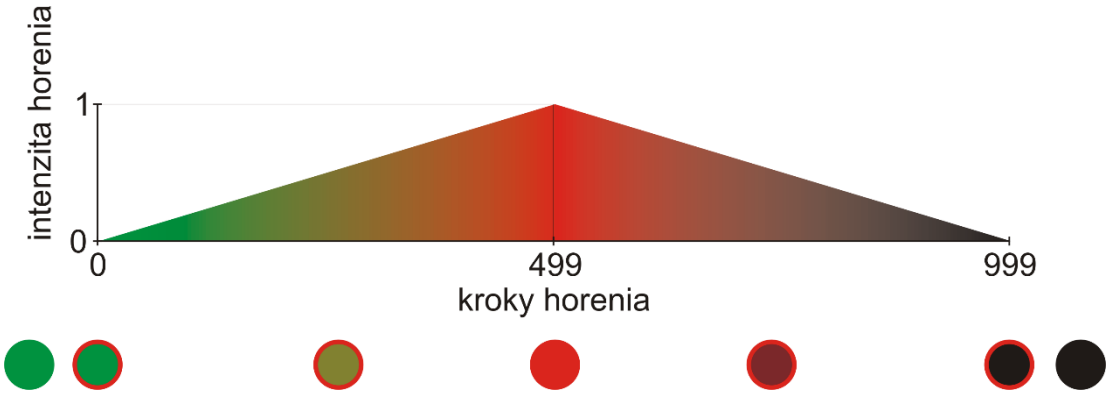
Dôvodom prečo sme takto prechody vzájomne odlíšili je fakt, že ak existuje z jedného stavu viac prechodov do iných stavov (viac vychádzajúcich šípok), bunka musí vedieť odlíšiť, ktorý z prechodov má zrealizovať. Napr. pri kliknutí **ľavým tlačidlom myšky** na bunku sa realizujú **červené prechody** a pri kliknutí **pravým tlačidlom myšky** na bunku sa realizujú **modré prechody**.

Ako horí strom

Pozrime sa bližšie na postupnosť stavov bunky nehorí → horí → dohorela. Táto zmena sa zrejme neudeje v troch jednoduchých krokoch. Ak strom začne horieť, oheň postupne naberá na intenzite až dosiahne maximálnu úroveň. Potom sa intenzita postupne znižuje až kým nezhorí všetko drevo. Predpokladajme pre náš model **jednotkové množstvo dreva** a **1000 krokov** na to, aby strom bez vonkajších zásahov úplne zhorel. Rovnako predpokladajme, že jednotkový strom dosiahne **maximálnu úroveň horenia rovnú 1**. Priebeh horenia môžeme simulovať nasledujúcim spôsobom (pozri obrázok nižšie):

Poznámka na okraj

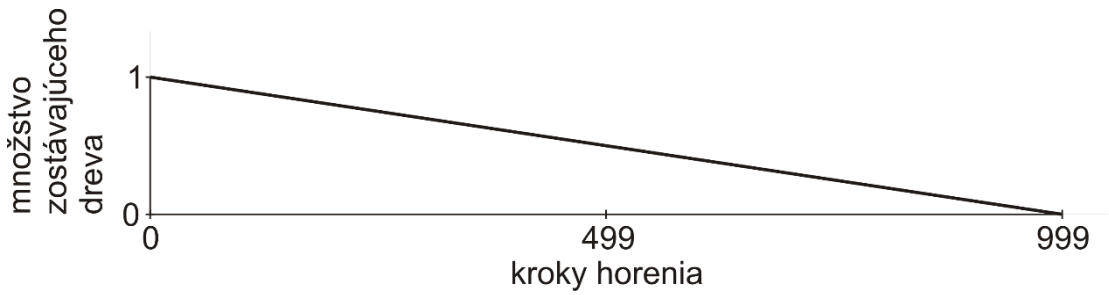
1000 krokov horenia a jednotkové množstvo dreva a horenia uvažujme len pre jednoduchosť modelu. Neskôr môžeme model pozmeniť a uvažovať o iných hodnotách.



Obrázok 24 Priebeh horenia jednotkového stromu

Bunky v modeli lesa budú svoj stav vyjadrovať farbou obrysu a farbou výplne (pozri obrázok vyššie). Ak bunka horí, bude mať krúžok červený obrys. Výplň horiacej bunky bude prechádzať od zelenej (bunka práve začala horieť) cez červenú (teraz je oheň najintenzívnejší) až po čiernu (všetko drevo zhorelo).

Postupne ako strom horí ubúda aj z množstvo dreva, ktoré ešte môže zhorieť. Množstvo nezhoreného dreva môžeme simulovať nasledovne:



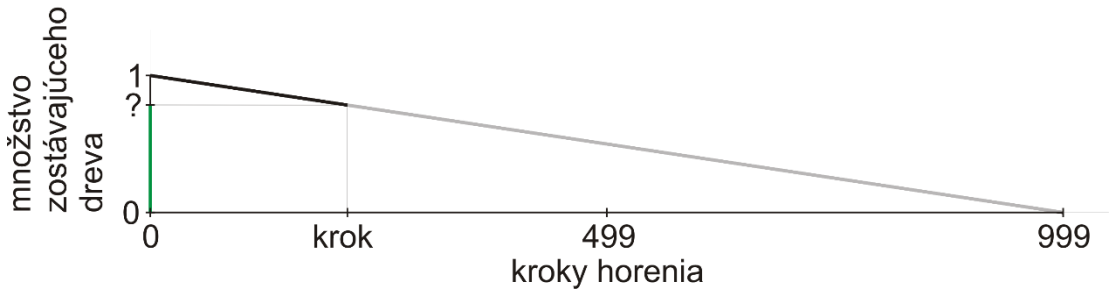
Obrázok 25 Množstvo zostávajúceho dreva pri horení jednotkového stromu

Poznámka na okraj

Takýto priebeh úbytku dreva počas horenia je značne zjednodušený. Ak sa pozrieme na priebeh horenia (Obrázok 24), tak pri najväčšej intenzite horenia by drevo malo ubúdať najrýchlejšie. V našom modeli na začiatku uvažujme takýto zjednodušený priebeh. Neskôr ho môžeme nahradiť reálnejším priebehom.

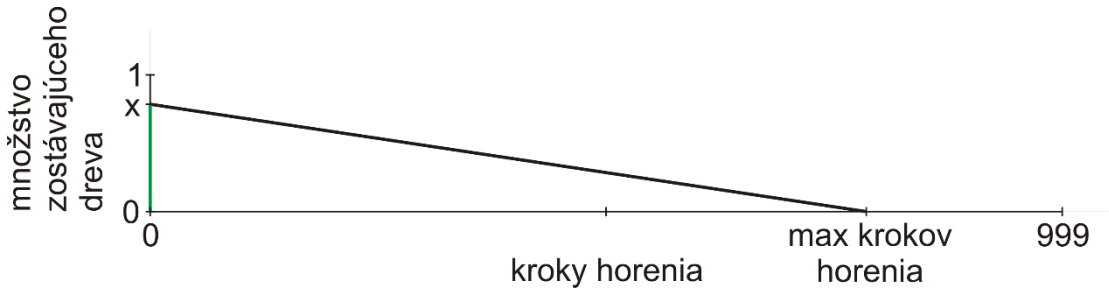
Uvažovať o množstve zostávajúceho dreva je potrebné z dôvodu, že ak strom uhasíme a on znova začne horieť, môže zhorieť len drevo, ktoré nezhorelo v predchádzajúcom horení. Podobne aj maximálna intenzita horenia už bude menšia.

Predpokladajme, že horiaci strom uhasíme v kroku `krok` (pozri obrázok nižšie).



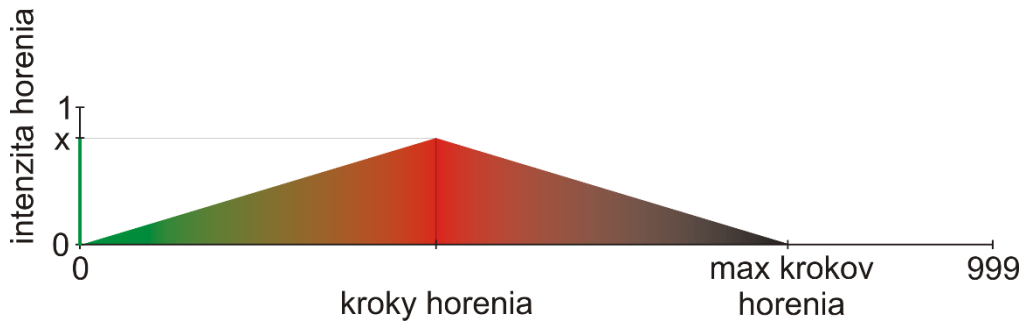
Obrázok 26 Uhasenie horiaceho stromu v kroku krok

Po jeho opätovnom zapálení bude horieť len tá časť dreva, ktorá nezhorela predtým (pozri obrázok nižšie).



Obrázok 27 Množstvo zostávajúceho dreva po znovu zapálení stromu

Rovnako ani maximálna možná intenzita požiaru nedosiahne pôvodnú úroveň (pozri obrázok nižšie).



Obrázok 28 Priebeh horenia stromu po znovu zapálení stromu

Implementácia modelu horenia lesa

V súbore **horenie_lesa.py** je pripravená časť programu pre simuláciu horenia lesa. Súbor obsahuje dve triedy:

- `HorenieLesa(tkinter.Tk)` – Objekt tejto triedy reprezentuje grafické prostredie pre simuláciu. Vygeneruje počiatočnú konfiguráciu lesa. Počiatočná generácia lesa je určená rozmerom lesa a náhodným rozmiestnením stromov v lese. Počet stromov je určený hustotou stromov. Bunky zatiaľ nevedia meniť svoj stav.
- `Bunka()` – Predstavuje jednu bunku. Objekt tejto triedy má pri vzniku nasledujúce informácie:
  - Kde je v lese umiestnený (`riadok`, `stlpec`). Všimnime si, že bunka sa zobrazí ako krúžok s priemerom 10 a umiestňujeme ju (resp. jej stred) na pozíciu `riadok * 10`, `stlpec * 10`.
  - Kresliacu plochu (`svet`) v ktorej sa má zobraziť.
  - Zoznam všetkých buniek v lese (`susedia`).
  - Odkiaľ fúka vietor (`smer_vetra`).
  - Stav (`stav`), v ktorom sa bunka nachádza. V čase vytvorenia bunky je stav bunky `nehori` alebo `nehorlava`.

Poznámka na okraj

V triede `Bunka()` je pomocná metóda `pis_info()`. Táto metóda sa zavolá, ak stredným tlačidlom myšky kliknete na bunku. Môžete ju využiť na kontrolné výpisy hodnôt inštančných premenných.

Úloha 1

Spustite program **horenie_lesa.py** a generujte rôzne počiatočné konfigurácie lesa.

Preskúmajte kód v programe **horenie_lesa.py**. Identifikujte funkcionalitu jednotlivých metód.

Úloha 2

Koľko krokov potrvá, kým zhorí jednotkový strom?

Koľko krokov potrvá, kým zhorí strom, ktorého množstvo nezhoreného dreva bolo pred začiatkom horenia `mnozstvo_dreva` (0 <= mnozstvo_dreva <= 1)?

Úloha 3

Metóda `priprav_na_horenie(self, mnozstvo_dreva)` sa zavolá vždy, keď sa bunka dostane do stavu `nehori`. Jej úlohou je pripraviť bunku na horenie, t. j. nastaviť počiatočné hodnoty premenných, ktoré charakterizujú samotné horenie:

- `self.mnozstvo_dreva`
- `self.max_krokov_horenia`
- `self.krok`

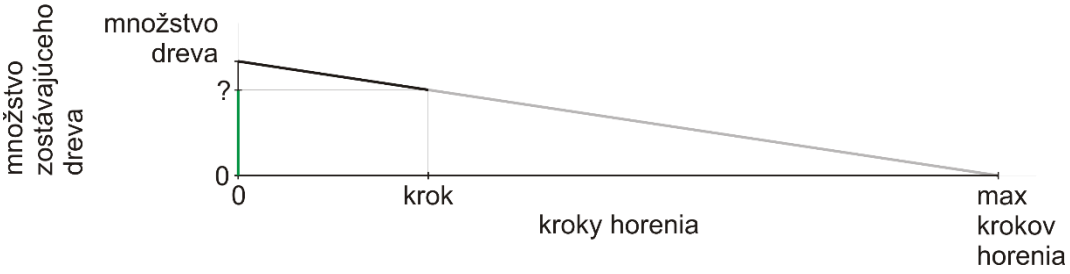
Preskúmajte program a nájdite miesto, kde sa táto metóda volá.

Preskúmajte prechody (Obrázok 22, Obrázok 23) medzi stavmi a identifikujte prechody, kedy by sa mala táto metóda zavolať.

Definujte telo tejto metódy tak, aby sa nastavili správne počiatočné hodnoty pre vyššie uvedené premenné.

Úloha 4

Predpokladajme, že sme horiaci strom uhasili v kroku krok (pozri obrázok nižšie). Ak poznáme počiatočné množstvo dreva a počet krokov horenia, koľko nezhoreného dreva zostalo?



Obrázok 29 Množstvo nezhoreného dreva po uhasení stromu

Doplňte telo metódy `get_zostatok_dreva(self)`.

Úloha 5

Definujte telá metód, ktoré umožnia realizovať umelé prechody medzi stavmi bunky.

Metóda `horlava_nehorlava(self, event)` realizuje modré prechody (Obrázok 23). Táto metóda sa zavolá pri kliknutí **pravým tlačidlom** myšky na bunku.

Metóda `zapal_zahas(self, event)` realizuje červené prechody (Obrázok 23). Táto metóda sa zavolá pri kliknutí **ľavým tlačidlom** myšky na bunku.

Poznámka: *Po zmene stavu bunky je vhodné strom nanovo vykresliť. Na to slúži metóda* `nakresli_strom(self)`.

Stav bunky sa uchováva v premennej `self.stav`. *Táto premenná nadobúda hodnoty* `'nehori'`, `'hori'`, `'dohorela'` *a* `'nehorlava'`.

Na začiatku sme si definovali, že ak strom horí bez vonkajších zásahov, tak každá ďalšia iterácia horenia ho posunie o jeden krok horenia. Ak však horia aj stromy okolo (vonkajší zásah), tak

bude strom zrejme horieť rýchlejšie (v jednej generácii horenia sa posunie o viac krokov). Aby sme mohli simulovať aj vplyv susedných stromov na priebeh horenia stromu, potrebujeme nasledovné:

- vedieť ktoré stromy (bunky) sú najbližší susedia,
- poznať intenzitu horenia susedov,
- definovať ako vplýva intenzita horenia okolitých buniek na bunku.

Premenná `self.susedia` je referenciou na zoznam všetkých stromov v lese. Niektoré z nich sú najbližší susedia. Aby sme nemuseli vždy prehľadávať všetky stromy v lese, upravme túto premennú tak, aby referencovala len zoznam blízkych susedov. Vytvorme pre bunku nový zoznam, do ktorého vložíme len tie bunky, ktoré s ňou bezprostredne susedia.

Úloha 6 (najbližší susedia)

Kedy najskôr môžeme bunku požiadať, aby si aktualizovala svoj zoznam susedov?

Ako bunka zistí, ktorá z buniek zo zoznamu `self.susedia` je jej blízkym susedom?

Môže bunka modifikovať zoznam `self.susedia`?

Doplňte telo metódy `zisti_blizkych_susedov(self)` tak, aby po jej volaní premenná `self.susedia` referencovala len zoznam blízkych susedov.

Úloha 7

Intenzita horenia postupne narastá až do polovice maximálneho počtu krokov horenia. Potom postupne klesá až k 0 (Obrázok 28).

Ak poznáte krok horenia, maximum krokov horenia a množstvo dreva ktoré môže zhorieť vyjadrite intenzitu horenia.

Doplňte telo metódy `get_intenzita_horenia(self)`.

Pomôcka: Uvažujte aj stav, v ktorom sa bunka nachádza.

Uvažujme o tom, aký bude priebeh horenia, ak pripustíme aj vonkajšie vplyvy – vplyv horenia susediacich buniek. Čím je intenzita horenia susedných buniek väčšia, tým rýchlejšie bude postupovať aj horenie bunky. V našom modeli definujeme nasledovný vplyv okolo horiacich buniek na bunku:

- ak bunka horí
  - ak žiadna z okolitých buniek nehorí, zmení sa krok horenia bunky o 1,
  - ak niektoré z okolitých buniek horia, zmení sa krok horenia bunky o 1 + intenzita horenia okolitých buniek,
- ak bunka nehorí
  - ak niektoré z okolitých buniek horia, bunka začne horieť s nejakou pravdepodobnosťou.

Aby sme mohli aplikovať nasledujúce pravidlá potrebujeme vedieť, aká je súhrnná intenzita horenia susediacich buniek a ako rozhodnúť, či bunka začne horieť alebo nie.

Úloha 8 (intenzita horenia susedov)

Definujte telo metódy `get_intenzita_horenia_okolo()`. Metóda vráti, aká je súhrnná intenzita horenia buniek okolo.

Úloha 9

Uvažujme nejaký jav, ktorý nastáva s nejakou pravdepodobnosťou. Táto pravdepodobnosť záleží od hodnoty premennej x, ktorá nadobúda hodnoty z intervalu ⟨0, max⟩, nasledovne:

Obrázok 30 Závislosť pravdepodobnosti, že jav nastane od hodnoty premennej x

Z obrázka vidno, že ak hodnota premennej x je 0, jav nenastane. Ak premenná x má hodnotu max, jav určite nastane. Čím vyššia je hodnota premennej x, tým vyššia je pravdepodobnosť, že jav nastane.

Vytvorte funkciu `nastane_jav(x, max)` ktorá vráti `True`, ak jav nastane a `False` ak jav nenastane.

Pre nehoriacu bunku sme si definovali pravidlo, že bunka začne horieť s nejakou pravdepodobnosťou. Čím je intenzita horenia susediacich buniek vyššia, tým bude táto pravdepodobnosť vyššia. Ak nehorí žiadna zo susediacich buniek, bunka horieť nezačne. Ak všetky susediace bunky horia maximálnou možnou intenzitou, bunka začne určite horieť.

Úloha 10 (vplyv intenzity horenia okolitých buniek na zapálenie bunky)

Aká je maximálna možná súhrnná intenzita horenia susediacich buniek?

Definujte telo metódy `zacne_horiet(self, aktualna_intenzita, max_intenzita)` na základe ktorej sa bunka rozhodne, či začne horieť alebo nie. Metóda by mala vrátiť hodnotu `True` alebo `False`.

Pomôcka: inšpirujte sa funkciou `nastane_jav()` z predchádzajúcej úlohy.

V tomto bode máme pripravené všetko pre to, aby sme vedeli implementovať zmenu stavu bunky do ďalšej generácie. V tomto prípade pôjde o prirodzené prechody (Obrázok 22). Umelé prechody sme už definovali v metódach `horlava_nehorlava()` a `zapal_zahas()`.

Pripomeňme si, ako pracuje celulárny automat. Je to dynamický systém, ktorý sa mení v krokoch. Najskôr sa pre každú bunku vypočíta, aký bude jej stav v nasledujúcej generácii. Potom všetky bunky svoj stav naraz zmenia a celý systém prejde do nasledujúcej generácie. Toto sa opakuje až do ukončenia celej simulácie.

Úloha 11

Definujte telo metódy `zisti_stav_nasledujuci(self)`.

Definujte telo metódy `nastav_stav_nasledujuci(self)`.

Pomôcka: Uvedomte si, ktoré premenné charakterizujú stav bunky.

Vytvorte si pomocné premenné pre pamätanie si nasledujúceho stavu bunky.

Ak bunka svoj stav nemení, jej nasledujúci stav bude rovnaký ako aktuálny.

Ak sme správne implementovali všetky uvedené metódy, mali by sme mať k dispozícii funkčnú aplikáciu, ktorá simuluje šírenie požiaru v lese.

Z uvažovaných faktorov, ktoré vplývajú na šírenie požiaru v lese sme zatiaľ nezohľadnili vietor a jeho smer.

Smer vetra vie každá bunka zistiť z premennej `self.smer_vetra` spôsobom `self.smer_vetra.get()`. Hodnota, ktorú takto získame, je jedna z nasledovných: 'S', 'SV', 'V', 'JV', 'J', 'JZ', 'Z', 'SZ', 'bezvetrie'. Ak fúka vietor smerom od horiacej bunky, vplyv horenia tejto bunky bude väčší. V metóde `__init__()` sme nastavili koeficient vplyvu vetra (`self.koeficient_vplyvu_vetra`). Intenzitu horenia bunky, od ktorej fúka vietor vynásobíme týmto koeficientom.

Výkladový text

Smer blízkeho suseda vieme charakterizovať zmenou čísla riadku a stĺpca z aktuálnej pozície. Táto zmena pre každého suseda predstavuje dvojicu, kde každý prvok je -1, 0 alebo 1. Ak by sme ku každému smeru vetra priradili takúto dvojicu, vieme pre každého blízkeho suseda zistiť, či vietor fúka od neho alebo nie. Stačí porovnať dvojicu reprezentujúcu smer suseda s dvojicou reprezentujúcou smer vetra.

N-ticu vytvoríme uzatvorením hodnôt oddelených čiarkami do zátvoriek, napr.:

```
ntica = (-1, 0)
```

Raz vytvorená n-tica je nemenná. Môžeme však pristupovať k jej prvkom, ktoré sú číslované od 0:

```
print(ntica[0])                    #-1
```

Prvkami n-tice môžeme prechádzať rovnako, ako prvkami iných štruktúrovaných premenných (reťazec, zoznam ..):

```
for prvok in ntica:
    print(prvok)                   #-1    0
```

Použitie n-tíc je výhodné (efektívne) v prípadoch, ak potrebujeme reprezentovať postupnosť hodnôt, ktorú nemeníme.

Úloha 12

Zohľadnite vplyv vetra a upravte metódu `get_intenzita_horenia_okolo()`.

Je potrebné ešte niekde zohľadniť vplyv smeru vetra?

Záver

Naprogramovali sme aplikáciu pomocou ktorej vieme simulovať horenie lesa. Do nášho modelu sme zakomponovali niektoré faktory, ktoré majú vplyv na šírenie požiaru v lese. Vplyv niektorých faktorov sme zjednodušili alebo zanedbali úplne. Do priebehu simulácie vieme umelo zasiahnuť, napr. zapáliť niektoré bunky alebo horiace bunky uhasiť. Môžeme simulovať stret viacerých požiarov alebo počas simulácie meniť smer vetra. Vieme skúmať, ako vplývajú nehoriace úseky a ich rozmiestnenie na šírenie požiaru alebo na možnosť jeho uhasenia človekom. Napriek tomu náš model predstavuje len zjednodušený pohľad na šírenie požiaru v lese.

Čo sme sa naučili

- analyzovať reálne situácie a deje,
- určiť podstatné a zanedbať nepodstatné faktory resp. prvky pre potreby simulácie,
- navrhnúť model reálnej situácie, resp. deja,
- definovať pravidlá správania sa elementov modelu,
- implementovať model a simulovať jeho správanie.

Ďalšie úlohy na precvičenie a zamyslenie

1. Bunky zobrazujeme ako krúžky rovnakej veľkosti. Doplňte model tak, aby veľkosť krúžku korešpondovala s množstvom horiaceho dreva.

   Pomôcka: Definujte si metódu `get_velkost()`, ktorá vráti veľkosť bunky. Túto hodnotu využite v metóde `nakresli_strom()`.

   Zmeniť veľkosť bunky je možné pomocou metódy `coords()` triedy `Canvas`.

2. V triede `Bunka()` sú všetky atribúty a metódy verejné, z vonka viditelné. Analyzujte, ktoré z nich nemusia byť verejné. Atribúty a metódy, ktoré nemusia byť verejné spravte z vonka triedy neviditeľné. Pre atribúty vytvorte get a set metódy (ak je to potrebné) a sprístupnite ich cez property.

3. V implementácií neuvažujeme chybné vstupy od používateľa grafického rozhrania. Analyzujte, ktoré vstupy pre simuláciu horenia lesa môže používateľa zadať chybne a ošetrite ich. Kedy je vhodné vstupy kontrolovať? Ak je vstup chybný, zobrazte chybovú správu v sekundárnom okne.

Poznámka na okraj

Chybové správy je vhodné zobraziť tak, aby si ich používateľ všimol. Môžeme ich zobraziť v novom okne napr. takto:

```
import tkinter.messagebox
....
def zobraz_chybu(self, chyba):
    tkinter.messagebox.showerror('Chyba', chyba)
```

4. V našom modeli sme neuvažovali situáciu, keď horiaci strom prirodzene prestane horieť. Uvažujte a implementujte aj tento prirodzený prechod. Tento prechod môže nastať s nejakou pravdepodobnosťou, prípadne v závislosti od intenzity horenia susedných buniek.
5. V našom modeli sme niektoré hodnoty nastavili ako konštanty (maximálne jednotkové množstvo dreva, maximálna jednotková intenzita horenia, maximálne 1000 krokov horenia, koeficient vplyvu vetra). Ktoré z týchto hodnôt je vhodné spraviť premenlivé? Do grafického rozhrania implementujte možnosť pre zmenu týchto hodnôt.
6. V triede `HorenieLesa()` sme všetky novovytvárané bunky umiestňovali do zoznamu `self.les`. Referenciu na tento zoznam dostala každá bunka (aby vedela osloviť svojich susedov) pri jej vytváraní. Ak by bunka tento zoznam modifikovala, zmena sa prejaví v každej bunke (všetky referencujú ten istý zoznam). Upravte triedu `HorenieLesa()` tak, aby sme bunke neposlali zoznam buniek ale n-ticu buniek.

7. Genetické algoritmy

Kľúčové slová

Genetický algoritmus, populácia, kríženie, mutácia, fitness, evolúcia

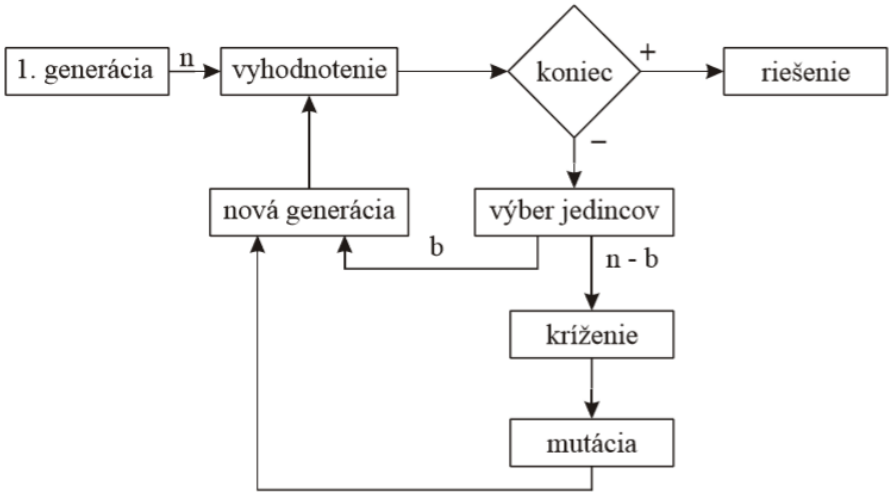
Čo sa naučíme a čo si precvičíme

- naučíme sa čo je to evolučný proces vo svete algoritmov,
- vysvetlíme si ako ohodnotiť riešenie,
- naučíme sa, čo je to selekcia, kríženie a mutácia,
- vysvetlíme si ako hľadať riešenie iným ako deterministickým spôsobom.

Problémová situácia

Optimalizačné problémy sa vyskytujú v každej oblasti ľudskej činnosti. Denne riešime problémy, ako niečo urobiť čo najlepšie, ako procesy optimalizovať. Optimalizačný problém vznikne v situácii, keď je nutné vybrať nejaké riešenie, ktoré je pre nás najvýhodnejšie. Zvyčajne tu nastupujú optimalizačné metódy. Problém matematicky formulovať, je treba zostaviť matematický model situácie. Výber najlepšieho riešenia prináša určité úskalia. Pre matematickú formuláciu optimalizačného problému volíme kritérium, podľa ktorého vyberáme najlepší variant riešenia. Výber kritéria optima je problematický a v mnohých aplikáciách podlieha často subjektívnym požiadavkám (33). Evolučné algoritmy v súčasnosti patria medzi základné nástroje modernej informatiky v prípadoch hľadania riešení v zložitých situáciách, kedy použitie štandardných metód založených na technikách úplného prehľadávania nemožno aplikovať. Evolučné algoritmy (EA) sú veľmi účinné optimalizačné algoritmy, ktoré vychádzajú zo znalosti zákonu prírodnej genetiky (34). Základnou myšlienkou evolučných algoritmov je napodobniť vývoj a učenie živočíšneho druhu a takto vzniknutý algoritmus použiť pri riešení úloh umiestnených do zložitého prostredia, v ktorom tvorca algoritmu nie je schopný dopredu definovať všetky vzniknuté prípady a správne reakcie na tieto prípady (35). Evolučné algoritmy sa zvyčajne delia na genetické algoritmy, genetické programovanie a evolučné stratégie.

Genetický algoritmus je programovacia technika, ktorá sa na základe biologickej evolúcie zaoberá stratégiou riešenia danej problematiky. Úlohou genetického algoritmu je pomocou správne navrhnutej **účelovej funkcie** (fitness) spracovať vstupné údaje, ktoré vyhodnotí, ako je každý prvok schopný sa ďalej vyvíjať. Vo všeobecnosti sú možné riešenia generované náhodne a úlohou genetického algoritmu je vybrať čo najlepšie riešenie a prípadne ho ešte vylepšiť Kandidátom, ktorí boli vyhodnotení ako málo perspektívni v oblasti ďalšieho požadovaného reprodukovania, nebude umožnené ďalšie rozmnožovanie, pretože predpoklad na ich silu a schopnosť nie je prijateľný na úkor silnejších a schopnejších jedincov. Tento proces sa neustále opakuje s tým, že sa očakáva postupné zvyšovanie kvality výsledného produktu evolúcie na základe fitness (36). Jedná sa vždy o algoritmus, ktorý je problémovo závislý, viac či menej vhodný pre konkrétny účel. Aplikáciou troch základných operácií: selekcie, kríženia a mutácie sa realizuje optimalizačný mechanizmus tak, aby sa hodnota účelovej funkcie minimalizovala, prípadne maximalizovala (pozri obrázok nižšie).



Obrázok 31 Genetický algoritmus pre n jedincov (33)

Problémová úloha

Navrhnite riešenie problému obchodného cestujúceho (The salesman problem). Problém môžeme popísať nasledovne: „Majme zoznam miest s ich polohou. Aká je najkratšia možná cesta navštívenia všetkých miest? Každé mesto môžeme navštíviť len raz a cesta musí skončiť v tom istom meste, v ktorom sme začali.

Pre jasnejšie pochopenie procesov, ktoré budeme používať, je potrebné uviesť niektoré pojmy, aby sme vôbec vedeli, o čo ide. Sú to základné pojmy, ktorých význam je u niektorých zrejmý už z názvu, iným dala špeciálny názov biológia, alebo genetika. V konečnom dôsledku si ich menovanie osvojíme a bude vhodné pre našu prácu.

Gén	základný nositeľ informácie, reťazec kódujúci jednu vlastnosť. V našom prípade **Mesto** reprezentované súradnicou **(x , y)**.
Jedinec	základná žijúca jednotka, označovaný aj ako chromozóm. V našom prípade pôjde o jednu **cestu spĺňajúcu kritériá**.
Populácia	množina jedincov, ktoré sa vyvíjajú z generácie na generáciu. V našom prípade **zoznam rôznych ciest**.
Generácia	jednoduchý prechod od aktuálnej populácie k nasledujúcej.
Fitness	hodnotenie (parameter), kvalita jedinca vo vzťahu k danému problému. V našom prípade, **ako krátka cesta to je**.

Úloha 1

Vygenerujte množinu bodov (určených súradnicou X a Y) predstavujúcich mestá, ktoré má navštíviť obchodný cestujúci. Umožnite vypočítať vzdialenosť medzi všetkými mestami (dĺžku trasy).

Pre lepšiu správu problému si vytvoríme triedu Mesto. Atribútmi mesta budú koordináty v 2D priestore.

```
class Mesto:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def vzdialenost(self, mesto):
        vzdialenostX = abs(self.x - mesto.x)
        vzdialenostY = abs(self.y - mesto.y)
        v = math.sqrt((vzdialenostX ** 2) + (vzdialenostY ** 2))
        return v

    def __repr__(self):
        return "(" + str(self.x) + "," + str(self.y) + ")"
```

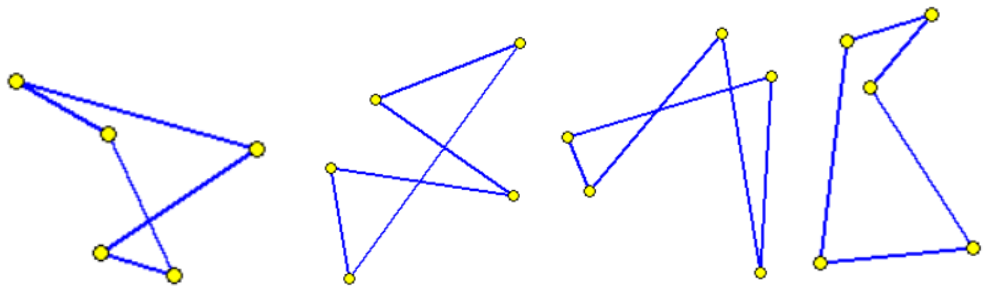
Dôležitou metódou triedy Mesto je `vzdialenost(mesto)`, ktorá slúži na výpočet vzdialenosti dvoch miest v rovine.

$$\sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$$

Pomocou triedy Mesto môžeme vyrobiť zoznam ľubovoľného počtu miest, pričom súradnice budeme generovať náhodne.

```
mesta = []
for i in range(0,5):
    m = Mesto(random.randrange(200), random.randrange(200))
    mesta.append(m)
```

Pre 5 miest by sme mohli vygenerovať rôzne mestské usporiadania. Na obrázku (pozri obrázok nižšie) je vykreslená vždy len jedna trasa (cesta), ktorou by náš obchodný cestujúci mohol ísť pre rôzne usporiadania miest.



Obrázok 32 Vizualizácia ciest pre päticu rozdielnych miest v priestore

Úloha 2

Umožnite ohodnotiť kvalitu trasy. Čím kratšia trasa, tým kvalitnejšie hodnotenie.

Vygenerujte množinu bodov (určených súradnicou X a Y) predstavujúcich mestá, ktoré má navštíviť obchodný cestujúci. Umožnite vypočítať vzdialenosť medzi mestami. Keď už máme vytvorené mestá, pozrime sa na spojenia medzi nimi. Vytvorme účelovú funkciu, fitness, ktorá ohodnotí kvalitu cesty.

$$\text{fitness} = \frac{1}{\text{dlzkaTrasy}}$$

```
class Fitness:
    def __init__(self, trasa):
        self.trasa = trasa
        self.vzdialenost = 0
        self.fitness= 0.0

    def vypocitajVzdialenost(self):
        self.vzdialenost = 0
        # ak som este vziadelonst    nevypocital
        if self.vzdialenost ==0:
            drahaVzdialenost = 0 #prejdi vsetky mesta trasy
            for i in range(0, len(self.trasa)):
                zMesta = self.trasa[i]           #idem z mesta [i]
                doMesta =self.trasa[i+1] if i+1 < len(self.trasa)
                else self.trasa[0]
                #do mesta [i+1] - ak ešte nejaké mesto je, ak uz som
                 presiel mesta,
                vraciam sa do prvého [0]
                print(zMesta.vzdialenost(doMesta))
                #vypocitam vzdialenost zMesta doMesta
                drahaVzdialenost += zMesta.vzdialenost(doMesta)
                #vypocitanu vzdialenost ulozim ako atribut triedy
                self.vzdialenost = drahaVzdialenost
        return self.vzdialenost

    def trasaFitness(self):
        if self.fitness == 0:
            self.fitness = 1 / float(self.vypocitajVzdialenost())
        return self.fitness
```

Trieda `Fitness` bude uchovávať `vzdialenost` vypočítanú metódou `vypociatajVzdialenost()` ako aj a `fitness` pre konkrétnu trasu.

Poznámka

Výpis pre jednu konkrétnu trasu bude vyzerať nasledovne. Toto je zatiaľ len trasa v takom poradí, v akom boli mestá generované do pola.

```
f = Fitness(mesta)
print('Trasa:', f.trasa)
print(f'Vzdialenost: {f.vypocitajVzdialenost():3.2f}')
print(f'Ohodnotenie (fitness):{f.trasaFitness():3.5f}')

#.....................................................

Trasa: [(120,64), (105,169), (14,73), (161,187), (109,8)]
Vzdialenost: 667.84
Ohodnotenie (fitness):0.00150
```

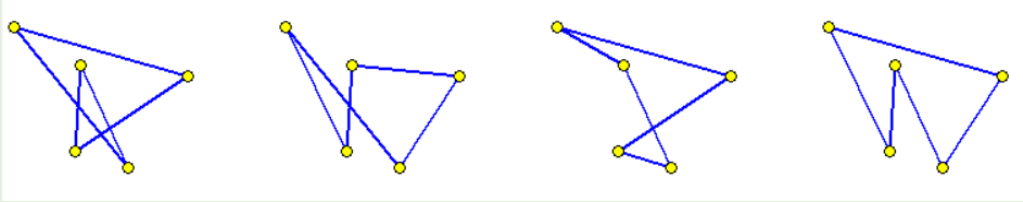
Úloha 3

Vytvorte populáciu jedincov. Populáciu v našom prípade bude predstavovať zoznam rôznych trás, ktoré treba ohodnotiť.

V predchádzajúcom príklade sme vygenerovali zoznam miest v určitom poradí. Teraz potrebujeme poradie miest postupne zmeniť.

Poznámka na okraj

Vygenerovaný zoznam miest predstavuje len jednu trasu, jedného jedinca celej populácie. Preusporiadaním miest dostane obchodný cestujúci inú trasu, iného jedinca a teda aj ich dĺžka a ohodnotenie sa bude odlišovať.



Ako vznikne ďalší jedinec? Stačí premiešať zoznam miest, čím vznikne nová trasa pre obchodného cestujúceho.

```
def vytvorTrasu(mesta):
    trasa = random.sample(mesta, len(mesta))
    return trasa
```

Poznámka

Metóda sample vyberie náhodné elementy zo vzorky hodnôt Ich počet je určený parametrom.

```
>>> import random
>>> c = list(range(0, 15))
>>> c
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

>>> random.sample(c, 5)
[9, 2, 3, 14, 11]
```

Ak je parameter rovnako veľký ako je veľkosť zoznamu, tak sa vlastne v náhodnom poradí vyberie každá položka práve raz.

Celý proces začína počiatočnou populáciou (initial population). Zavolaním metódy `vytvorTrasu(mesta)` vznikne len nový jedinec. My však potrebujeme vygenerovať celú populáciu jedincov. Budeme opakovane vytvárať toľko jedincov, koľko pre danú populáciu budeme potrebovať.

```
def pociatocnaPopulacia(popVelkost, mesta):
    populacia = []

    for i in range(0, popVelkost):
        populacia.append(vytvorTrasu(mesta))
        #vytvorime novu trasu, noveho jedinca a pridame do zoznamu

    return populacia
    #populacia je vlastne zoznam tras vytvorenych z povodneho zoznamu
populacia = pociatocnaPopulacia(10, mesta)
```

Vytvorili sme prvotnú generáciu jedincov. Tento proces vykonáme len raz. Ďalšie generácie budú vytvárané na základe selekcie, mutácie a kríženia.

Úloha 4

Simulujte proces evolúcie, kde prežijú len niektorí jedinci. Na generáciu aplikujte pravidlá prirodzeného výberu, selekcie. Výberom získajte jedincov vhodných reprodukcie.

Skôr ako budeme môcť rozhodnúť, ktorý jedinec je ten najsilnejší (ktorá trasa je zatiaľ najkratšia), potrebujeme pre každú trasu vypočítať hodnotenie a následne ich usporiadať. Potrebujeme uchovávať dve hodnoty - hodnotenie (fitness) a identifikáciu príslušného mesta. Ako najlepšie riešenie je pracovať s dátovou štruktúrou `dict` a z nej vytvoriť zoznam obsahujúci `tuple`.

```
def ohodnotTrasy(populacia):
    hodnotenie = {} #slovník bude obsahovať dvojicu poradieTrasy:fitnessTrasy
    for i in range(0,len(populacia)):
        hodnotenie[i] = Fitness(populacia[i]).trasaFitness()

    return sorted(hodnotenie.items(), key = operator.itemgetter(1),
                  reverse = True)
```

Poznámka

Hodnotením vytvoríme slovník, v ktorom je poradie trasy kľúčom a fitness trasy je hodnotou.

```
{0: 0.002191581344527877, 1: 0.0020960893400294852, 2: 0.001993791365400247,
3: 0.0021225941817017042, 4: 0.0020960893400294852}
```

Vo funkcii sorted používame tri parametre:

`hodnotenie.items` – položky v slovníku sa vrátia ako zoznam hodnôt typu `tuple`. `0: 0.002191581344527877 sa premapuje na (0, 0.002191581344527877)`

`key = operator.itemgetter(1)` – parameter `key` určuje, čo kritérium usporiadania. V našom prípade usporiadame vzhľadom na ohodnotenie, fitness.

`reverse = True` – usporiadanie bude zostupné, od najväčšieho fitnes po najmenšie

Funkcia `ohodnotTrasy` teda vráti usporiadaný zoznam, kde na prvom mieste sa nachádza tá trasa populácie, ktorá ma najlepšie ohodnotenie.

```
[(0, 0.002191581344527877), (3, 0.0021225941817017042), (1, 0.0020960893400294852),
(4, 0.0020960893400294852), (2, 0.001993791365400247)]
```

Počas reprodukčnej fázy genetického algoritmu sú z populácie vybraní jedinci rekombinovaní a vytvorí sa potomstvo, ktoré bude tvoriť nasledujúcu generáciu. Rodičia sú vybraní náhodne, avšak kvalitní jedinci sú zvýhodnení – je pravdepodobné, že sa budú rozmnožovať niekoľkokrát za generáciu, zatiaľ čo slabí jedinci ani raz. Ako však rozhodnúť, ktorí jedinci budú vybratí do ďalšej generácie? Pri reprodukcii sa používajú rôzne metódy selekcie. My využijeme dve a to elitárstvo a ruletový výber.

Elitárstvo (Elitism) Na reprodukciu sú vždy zvolení najvhodnejší rodičia. Tí tak majú istotu, že budú existovať aj v ďalšej populácii (36). Implementovať elitárstvo je jednoduché. Stačí vybrať určený počet jedincov s najlepším hodnotením.

```
def selekcia(aktualnaGeneracia, elita):
    popHodnotena = ohodnotTrasy(aktualnaGeneracia)

    vyber = []
    for i in range(0, elita): #vyber elitnych jedincov
        vyber.append(popHodnotena[i])
```

Ruleta (Roulette Wheel Selection) Rodičia sú vyberaní na základe ich fitness. Čím lepší fitness, tým je pravdepodobnosť rozmnožovanie väčšia. Nech by sme mali hodnoty jedincov napríklad takéto 80, 66, 35, 22, 15 (pozri obrázok nižšie). Spolu by mali hodnotu 218.

Obrázok 33 Vizualizácia hodnoty (fitness) pre piatich jedincov

```
total_fitness = 0
for i in range(len(popHodnotena)):
    total_fitness += popHodnotena[i][1]
```

Každý jedinec, ktorý je spracovávaný v procese reprodukcie má na pomyselnom kole rulety určitý výrez, ktorého veľkosť je závislá od jeho vhodnosti pre reprodukciu, inými slovami, čím je jeho vhodnosť väčšia, tým má väčší výrez. Túto hodnotu môžeme prepočítať v percentách v pomere k súčtu všetkých hodnôt (relatívne fitness, pozri obrázok nižšie) (37).

```
rel_fitness =
[100*popHodnotena[i][1]/total_fitness for i in
range(len(popHodnotena))]
```

Obrázok 34 Vizualizácia hodnoty pre piatich jedincov vyjadrená percentuálne (relatívne fitness)

Potom sa náhodným výberom (hodením guľky do kola rulety) zvolí jedinec. Šancu, aj keď menšiu, majú aj menej vhodní jedinci. To zrealizujeme tak, že pre každý úsek zvolíme kumulatívne súčty pre percentuálne hodnotenie jedincov (pozri obrázok nižšie).

Obrázok 35 Vizualizácia hodnoty pre piatich jedincov vyjadrená kumulatívne (kumulatívne fitness)

```
kum_fitness = [sum(rel_fitness[:i+1]) for i in
range(len(rel_fitness))]
```

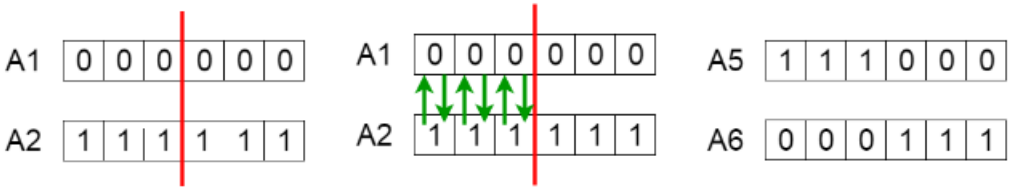
Ako bude prebiehať ruletový výber? Náhodne vyberieme číslo od 0-100. A podľa toho, do ktorého kumulatívneho intervalu padlo, podľa toho vyberieme jedinca. Niektorý jedinec sa môže vo výbere vyskytnúť častejšie, keďže zaberá väčší priestor na pomyselnej rulete.

```
while len(vyber) < len(popHodnotena):
      pom = 100*random.random()
      for i in range(0, len(kumu_fitness)):
          if kum_fitness[i]<= pom < kum_fitness[i+1]:
            vyber.append(aktualnaGeneracia[i])
            break

return vyber
vybrataPopulacia = selekcia(populaciaHodnotena, elita = 20)
```

Úloha 5

Simulujte proces evolúcie, kde jedinci vznikajú reprodukciou, počas ktorej dochádza ku kríženiu (crossover).

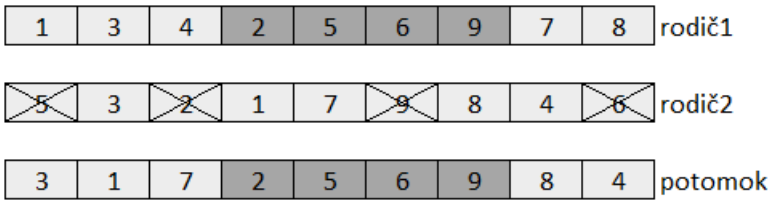
Kríženie je súčasťou reprodukčného procesu a je to vlastne výmena génov od rodičov. Počet zdedených génov môže byť náhodný (pozri obrázok nižšie).



Obrázok 36 Reprodukcia jedincov krížením

Aby sme mohli tento proces požiť aj v našom prípade musíme dodržať pravidlo, že každé mesto môže byť navštívené len raz. Kríženie preto musíme mať pod kontrolou, bude prebiehať nasledovne (pozri obrázok nižšie):

1. Od prvého rodiča náhodne vyberieme interval génov (miest), ktorý potomok zdedí.
2. Od druhého rodiča zoberieme všetky nepoužité gény (mestá) v rovnakom poradí, ako sa nachádzali u rodiča.



Obrázok 37Reprodukcia jedincov krížením pre dva zoznamy miest

```
def krizenie(rodic1, rodic2):
    potomok = []
    #vyber hranic intervalu dedených génov
    genA = random.randrange(len(rodic1))
    genB = random.randrange(len(rodic1))
    #zistime, ktory gen je vporadi vlavo a ktory vpravo
    zacGen = min(genA, genB)
    konGen = max(genA, genB)

    #ak sme v intervale berieme rodica1 inak rodica2
    for i in range(len(rodic1)):
```

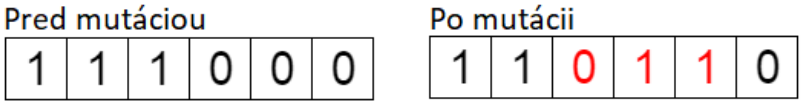
```
        potomok.append(rodic1[i] if zacGen <= i <=konGen else
        rodic2[i])

    return potomok
```

Teraz už len potrebujeme aplikovať kríženie na celú populáciu, opäť budem myslieť na elitárstvo najlepších jedincov.

```
def krizeniePopulacie(pop, elita):
    potomkovia = []

    #najskôr zachovám elitu
    for i in range(0,elita):
        potomkovia.append(pop[i])

    #pomiešam jedincov
    pom = random.sample(populacia, len(populacia))

    #skrížim prvého jedinca populácie s posledným, druhého s
predposledným atď.
    for i in range(0, (len(pop) - elita)//2):
        potomok = krizenie(pom[i], pom[len(populacia)-i-1])
        potomkovia.append(potomok)
    return potomkovia
```

Úloha 6

Simulujte proces evolúcie, kde jedinci vznikajú mutáciou. Každý z potomkov je náhodným spôsobom s určitou (spravidla veľmi malou) pravdepodobnosťou mierne pozmenený.



Obrázok 38 Vznik jedinca procesom mutácie zmenou génu

Užitočnosť používania mutácie je v tom, že sa vlastne stará o to, aby sa v novo vytvorenej populácii mohli vyskytnúť u jedincov aj gény, ktoré v predošlej populácii neboli. V našom prípade opäť nemôžeme použiť zmenu génu, lebo by sme stratili dôležitú informáciu, mesto v trase. Mutáciu si preto definujeme *ako výmenu pozície dvoch miest v jednej trase.* Stačí ak náhodne vyberiem dve mestá trasy a vymeníme ich (Obrázok 38).

```
def mutacia(jedinec, mieraMutacie):
    #pre kazde mesto jedinca
    for vymena1 in range(len(jedinec)):
        # vygenerujeme nahodné číslo od 0 po 1
        # porovname ci je nahodne cislo mensie ako miera mutacie
        # miera mutacie určuje pravdepodobnosť mutácie
        if(random.random() < mieraMutacie):
            # náhodne vyberiem index druhého mesta
            vymena2 = random.randrange(len(jedinec))

            mesto1 = jedinec[vymena1]
            mesto2 = jedinec[vymena2]
```

```
            jedinec[vymena1] = mesto2
            jedinec[vymena2] = mesto1
    return jedinec
```

Novú populáciu mutáciou vytvárame tak, že najskôr vyberieme elitných jedincov (trasy) a potom urobíme mutáciu pre zostávajúcich.

```
def mutaciaPopulacie(pop, mieraMutacie, elita):
    mutPop = []

    #najskor zachovam elitu
    for i in range(0,elita):
        mutPop.append(pop[i])

    # pre každého jedinca
    for index in range(elita, len(pop)):
        # s istou pravdepodobnosťou urobím mutáciu trasy
        mut = mutacia(pop[index], mieraMutacie)
        mutPop.append(mut)

    return mutPop
```

Dokončili sme funkcie pre všetky potrebné časti genetického algoritmu. Teraz ich už len potrebujeme vhodne skomprimovať. Najlepšie proces vystihuje pseudokód.

```
Vygeneruj počiatočnú populáciu
Vypočítaj fitness
Opakuj
    Výber
    Kríženie
    Mutácia
    Vypočítaj fitness
```

Nasledujúca generácia sa teda vypočíta tak, že na aktuálnu generáciu aplikujeme proces výberu, kríženia a mutácie.

```
def dalsiaGeneracia(aktualnaGeneracia, el, mieraMutacie):
    vybrataPopulacia = selekcia(aktualnaGeneracia, elita = el)
    krizenaPopulacia = krizeniePopulacie(vybrataPopulacia, elita =
el)
    mutaciaPopulacia = mutaciaPopulacie (krizenaPopulacia,
mieraMutacie)
    return mutaciaPopulacia
```

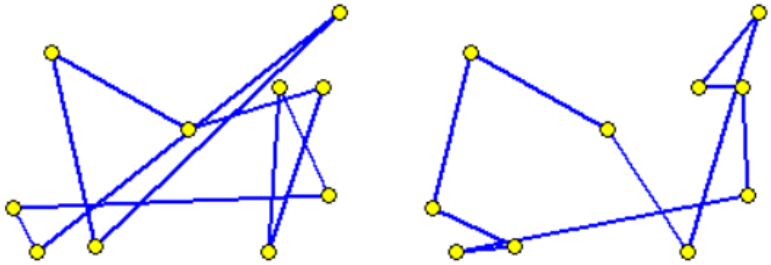
Samotný genetický algoritmus vygeneruje počiatočnú populáciu a aplikuje proces hľadania ďalších generácií.

```
def GA(mesta, velkost, elita, mieraMutacie, generacie):

    pop = pociatocnaPopulacia(velkost, mesta)
    h = ohodnotTrasy(pop)
    najkratsia = h[0][1]
    print("Počiatočná veľkosť trasy: " + str(1 / najkratsia))

    for i in range(0, generacie):
        pop = dalsiaGeneracia(pop, elita, mieraMutacie)
```

```
    h = ohodnotTrasy(pop)
    najkratsia = h[0][1]
    print("Konečná veľkosť trasy: " + str(1 / najkratsia))

    najTrasaIndex = ohodnotTrasy(pop)[0][0]
    najTrasa = pop[najTrasaIndex]
    return najTrasa
```

Ďalšie úlohy na precvičenie a zamyslenie

1. Doplňte grafické rozhranie na zobrazenie miest a ciest medzi mestami.



8. Chcem byť ako Mozart

Kľúčové slová

digitalizácia zvuku, zvukový formát *wav*, modul *wave*, náhoda, simulácia

Čo sa naučíme a čo si precvičíme

- pripomenieme si, ako sa digitalizuje zvuková informácia,
- preskúmame podrobnejšie zvukový formát *wav*,
- zoznámime sa s modulom *wave*,
- precvičíme si prácu so zoznamom zoznamov („dvojrozmerným poľom“),
- s využitím náhody vytvoríme originálne hudobné dielo.

Problémová situácia

V roku 1787 *Wolfgang Amadeus Mozart* vytvoril hru s hracími kockami, pri ktorej hráč skomponuje krátky valčík tým, že náhodne vyberie a spojí **32 z 272 predpripravených taktov**. Takty sú zložené tak, aby na seba plynulo nadväzovali. Náhodne generované skladby tak znejú prekvapivo dobre („mozartovsky“), podobne ako valčíky skladané v období vzniku hry (38).

Valčík, ktorý je výsledkom hry, je tvorený 2 časťami s názvom *menuet* a *trio*. Každá z častí pozostáva zo **16 taktov**. Pre ich výber sú definované presné pravidlá:

Menuet

Vyberá sa zo 176 možností. Pre každý takt hodíme **dvoma hracími kockami** a z tabuľky vyberieme tú možnosť, ktorá zodpovedá hodenému súčtu. Pri hádzaní 2 hracími kockami môžeme získať 11 rôznych súčtov (2, 3, 4, ..., 12):

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	96	22	141	41	105	122	11	30	70	121	26	9	112	49	109	14
3	32	6	128	63	146	46	134	81	117	39	126	56	174	18	116	83
4	69	95	158	13	153	55	110	24	66	139	15	132	73	58	145	79
5	40	17	113	85	161	2	159	100	90	176	7	34	67	160	52	170
6	148	74	163	45	80	97	36	107	25	143	64	125	76	136	1	93
7	104	157	27	167	154	68	118	91	138	71	150	29	101	162	23	151
8	152	60	171	53	99	133	21	127	16	155	57	175	43	168	89	172
9	119	84	114	50	140	86	169	94	120	88	48	166	51	115	72	111
10	98	142	42	156	75	129	62	123	65	77	19	82	137	38	149	8
11	3	87	165	61	135	47	147	33	102	4	31	164	144	59	173	78
12	54	130	10	103	28	37	106	5	35	20	108	92	12	124	44	131

Príklad: Ak pre 5. takt v poradí padne na kockách súčet 8, do vytváranej skladby pripojíme takt č. 99.

Trio

Vyberá sa z 96 možností. Hádže sa len **jednou kockou**. Pre každý takt hodíme hracou kockou a z tabuľky vyberieme tú možnosť, ktorá zodpovedá padnutej hodnote:

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

Príklad: Ak pre 16. takt v poradí padne na kocke hodnota 3, do vytváranej skladby pripojíme takt č. 21.

Poznámka na okraj

W. A. Mozart (1756-1791) bol rakúsky hudobný skladateľ v období klasicizmu. Prečítajte si o Mozartovi vo *Wikipédii* alebo si vypočujte niektoré z jeho diel na *Youtube*.

Autorstvo hry *Musikalisches Würfelspiel* sa tradične pripisuje Mozartovi, hoci prvý raz bola publikovaná až po jeho smrti. Myšlienka, na ktorej je komponovanie bez nutnosti vedieť komponovať založené, bola známa už skôr. Je preto možné aj to, že sa Mozart inšpiroval niekým iným (39).

Hľadajme riešenie

V Mozartových časoch by sme museli jednotlivé takty valčíka ručne prepisovať na papier podľa vzorov zobrazených v tabuľkách a vytvoriť tak notový zápis novej skladby. Ak by sme si chceli valčík vypočuť či zatancovať, niekto by ho musel podľa nôt zahrať, napr. na klavíri.

Keď máme k dispozícii zvukové záznamy jednotlivých taktov vo formáte *wav*, nebude nutné, aby sme poznali noty. Predpokladajme, že súbory taktov pre časť menuet majú názvy *M1.wav, M2.wav, ..., M176.wav*. Súbory taktov pre časť trio budú pomenované *T1.wav, T2.wav, ..., T96.wav*.

Naprogramujeme počítačovú simuláciu pôvodnej Mozartovej hry. Hádzanie kockami nahradíme generovaním pseudonáhodných čísel. **Podľa údajov z tabuliek vyberieme pre každý z 32 taktov správny zvukový súbor a spojíme ich do výslednej skladby**. Tú si ľahko prehráme na počítači, či v inom obľúbenom zariadení. Na prácu so zvukovými súbormi použijeme modul *wave* zo štandardnej knižnice Pythonu.

Poznámka na okraj

Valčík je spoločenský tanec v ¾ rytme. Už ste ho tancovali? Ak nie, asi budete – na stužkovej, na plese, na svadbe. Oplatí sa ho naučiť.

Úloha 1

Pre každý zo 16 taktov menuetu máme na výber 11 možností. Pre každý zo 16 taktov tria je 6 rôznych možností. Koľko rôznych valčíkov môžeme pri hraní Mozartovej hry skomponovať?

Porovnajte si výsledok svojho výpočtu: Výstupom hry môže byť 11^{16} rôznych menuetov a 6^{16} rôznych verzií tria. Ich kombinovaním môže vzniknúť až $11^{16}. 6^{16}$ = 129 629 238 163 050 258 624

287 932 416 rôznych valčíkov. Niektoré sa budú líšiť len v jednom alebo málo taktoch a budú znieť takmer rovnako. Z veľkého počtu možných výsledkov však určite vzíde aj veľa originálnych skladieb.

Úloha 2

a) Tabuľku s riadkami a stĺpcami v Pythone implementujeme ako *zoznam zoznamov*. Každý prvok *hlavného zoznamu* je opäť zoznam predstavujúci 1 riadok tabuľky. Každý z riadkov má rovnaký počet prvkov, preto sú tieto zoznamy rovnako dlhé. Inicializujte v programe premenné `menuet` a `trio` reprezentujúce vyššie uvedené tabuľky pravidiel Mozartovej hry.

b) Predstavte si túto situáciu: Vyberáme zvukovú nahrávku pre 12. takt menuetu. Vypíšte na obrazovku *názov súboru*, ktorý sa má pripojiť do vytváranej skladby.

Výkladový text

Digitalizácia zvukovej informácie

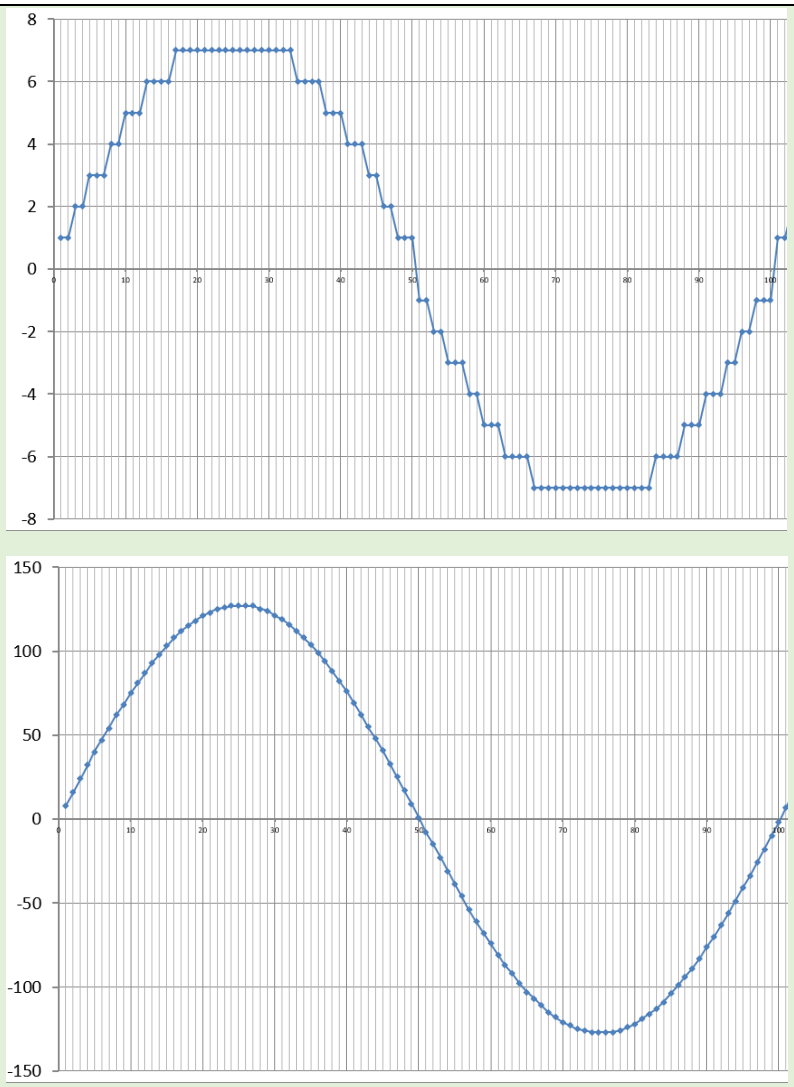
Z fyziky vieme, že zvuk je mechanické vlnenie. Analógový (spojitý) zvukový signál sa pri nahrávaní do počítača digitalizuje **procesom vzorkovania** (v angl. *sampling*).

Pri vzorkovaní **s frekvenciou** 44100 Hz sa **amplitúda** zvukovej vlny odmeria 44100 krát za sekundu (časový interval v trvaní 1 sekundy sa rozdelí na 44100 rovnakých častí). Namerané číselné hodnoty sa kódujú ako postupnosť núl a jednotiek a zapisujú sa do binárneho súboru. Vzniká digitálna (číslicová) reprezentácia zvukovej informácie. Pri nižšej vzorkovacej frekvencii získame o pôvodnej zvukovej vlne menej informácií.

Dôležitým parametrom pri nahrávaní zvuku je **rozlíšenie vzorky (bitová hĺbka)**. Pri 16 bitovom rozlíšení máme pri kódovaní k dispozícii až 2^{16} = 65536 rôznych úrovní (-32768..32767). Pri 8 bitovom rozlíšení môžeme nameraným hodnotám priradiť len niektorú z 2^8 = 256 hodnôt (-128..127).

Príklad:

Na obrázkoch nižšie vidíme výsledok vzorkovania s frekvenciou 22050 Hz pre prvých 100 vzoriek rovnakej zvukovej vlny (v skutočnosti mala táto zvuková vlna tvar sínusoidy).



Obrázok 39 Porovnanie výsledku vzorkovania pri rôznych bitových hĺbkach

V prvom prípade sme použili rozlíšenie 4 bity, v druhom prípade 8 bitov.

Rozdiel je očividný. Prvá digitalizovaná vlna je viac „schodovitá“, keďže rozsah hodnôt na osi *y* je menší (2^4 = 16 hodnôt). Pri 8 bitovom rozlíšení je to až 2^8 = 256 úrovní.

Pri nahrávaní zvuku tiež volíme počet **kanálov** (1 kanál = mono, 2 kanály = stereo). Pre každý z kanálov sa hodnoty ukladajú osobitne. Stereo záznam má preto dvojnásobnú veľkosť.

Bezstratový zvukový formát, v ktorom je zvuková informácia uložená v podobe, v akej sa pri nahrávaní zakódovala, sa nazýva **WAV/WAVE** (*wave = vlna*). Keďže nedochádza ku kompresii, súbory *wav* s dlhším nahrávkami sú veľké. Pre porovnanie: skomprimovaním do *MP3* formátu zmenšíme veľkosť zvukového súboru cca na 1/10 pôvodnej veľkosti. Výhodou *wav* súborov je však možnosť pracovať so „surovými“ dátami zaznamenanými pri nahrávaní. Pomocou softvéru na spracovanie zvuku môžeme *wav* súbor ľahko upravovať (40).

Poznámka na okraj

O digitalizáciu zvuku sa pri nahrávaní do počítača postará analógovo-digitálny prevodník zvukovej karty.

Príklady typických hodnôt parametrov:

telefónna kvalita	11 025 Hz, 8 b, mono
rozhlasová kvalita	22 050 Hz, 8 b, mono
CD kvalita	44 100 Hz, 16 b, stereo
DVD kvalita	192 000 Hz, 24 b, 5.1 surround (5 reproduktorov a subwoofer)

Otázka

Akú veľkosť má 1 minúta stereo nahrávky vyhotovenej pri vzorkovacej frekvencii 44100 Hz a rozlíšení 16 b?

Úloha 3

Napíšte skript, v ktorom otvoríte nahrávku *M1.wav* a zobrazíte všetky informácie z hlavičky *wav* súboru. Potom vypočítajte *dĺžku nahrávky v sekundách* a celkovú *veľkosť súboru* s nahrávkou (hlavička + dáta zvuku).

Výkladový text

Súbor *wav* je binárny súbor. Začína hlavičkou (44 B) s informáciami o parametroch použitých pri nahrávaní, za ktorou nasleduje blok dát reprezentujúci samotný zvuk. So súbormi vo formáte *wav* môžeme v Pythone pracovať pomocou štandardného modulu `wave`:

```
import wave

# otvorenie súboru s nahrávkou na čítanie
nahravka = wave.open('M1.wav', 'rb')

# výpis hodnôt jednotlivých parametrov
print(nahravka.getnchannels()) # 1          počet kanálov
print(nahravka.getsampwidth()) # 2          rozlíšenie
print(nahravka.getframerate()) # 44100      frekvencia
print(nahravka.getnframes())   # 82588      počet vzoriek
print(nahravka.getcompname())  # 'NONE'
print(nahravka.getcomptype())  # 'not compressed'

# všetky parametre v jednej n-tici
print(nahravka.getparams())
```

Funkcia `open` vracia ako výsledok špeciálny objekt typu `Wave_read`, pre ktorý sú definované rôzne užitočné metódy. Zatiaľ sme potrebovali len metódy na prístup k informáciám uloženým v hlavičke otvoreného súboru.

Úloha 4

Preštudujte skript, v ktorom nahrávky 8 tónov stupnice C dur (*c.wav, d.wav, e.wav, f.wav, g.wav, a.wav, h.wav* a *c2.wav*) spojíme za sebou a vytvoríme jeden zvukový súbor (celú stupnicu):

```
import wave

# pripravíme si názvy súborov, ktoré budeme otvárať
nahravky = ['tony/'+ ton +'.wav' for ton in 'cdefgah']
nahravky.append('tony/c2.wav')                                    #[1]

nahravka1 = wave.open('tony/c.wav', 'rb')

# otvoríme súbor, do ktorého budeme zapisovať
vystup = wave.open('stupnica_cdur.wav', 'wb')                     #[2]

# parametre v hlavičke môžeme skopírovať z prvej nahrávky
vystup.setparams(nahravka1.getparams())

# pre každý tón stupnice
for i in range(len(nahravky)):
    # otvoríme súbor s nahrávkou tónu
    nahravka = wave.open(nahravky[i], 'rb')
    # zapíšeme jeho dáta do vytváraného súboru
    vystup.writeframes(nahravka.readframes(nahravka.getnframes()))   #[3]


# súbor, do ktorého sme zapisovali, zatvoríme
vystup.close()
```

[1] Vytvoríme zoznam s názvami zvukových súborov s tónmi. Predpokladáme, že sú všetky uložené v priečinku *tony*. V zápise vytvárania zoznamu vidíme generátorovú notáciu – každý znak (`ton`) z reťazca `'cdefgah'` využijeme na vytvorenie jedného prvku zoznamu. Posledný tón stupnice pridáme do zoznamu osobitne.

[2] Ideme **vytvárať nový zvukový súbor**, pri otváraní súboru preto použijeme režim `'wb'` (*w = write, b = binárne*). Funkcia `wave.open` vráti špeciálny objekt typu `Wave_write`, ktorého metódy používame v ďalších riadkoch skriptu.

[3] Metóda `writeframes()` zapíše binárne dáta uvedené v parametri do výstupného súboru. Metóda `readframes()` má 1 parameter – počet vzoriek (*frames*), ktoré zo súboru chceme prečítať. My sme chceli prečítať celý vstupný súbor, ako parameter preto uvádzame počet všetkých vzoriek príslušného tónu.

Úloha 5

Dokončite simuláciu Mozartovej hry. Napíšte funkciu `generuj_valcik` s jedným parametrom – názvom súboru *wav*, do ktorého sa výsledná, náhodne vytvorená skladba uloží.

Čo sme sa naučili

- Zvukový formát WAV/WAVE je bezstratový spôsob kódovania zvukovej informácie. Binárny súbor *wav* pozostáva z hlavičky nasledovanej dátami získanými procesom vzorkovania analógového zvukového signálu. Kvalitu a veľkosť zvukového záznamu ovplyvňuje vzorkovacia frekvencia, rozlíšenie vzorky aj počet použitých kanálov. Čím viac a čím presnejších čísel vzorkovaním získame, tým vernejšie vieme zvuk uchovať.
- V Pythone je pre prácu so zvukovými súbormi k dispozícii modul *wave*. Obsahuje funkcie a objekty s metódami, vďaka ktorým vieme zistiť základné charakteristiky zvukového súboru, čítať zvukové dáta a zapisovať ich.

Ďalšie úlohy na precvičenie a zamyslenie

1. Prezrite si jednotlivé súbory *wav* vo zvukovom editore, napr. *Audacity* (https://www.audacityteam.org/). Uvidíte časový priebeh zvukovej vlny a aj jej parametre. Pri vhodnom priblížení tiež hodnoty amplitúdy získané vzorkovaním.

2. Skonvertujte výsledný *wav* súbor s valčíkom do formátu *mp3*. Môžete použiť niektorý z online nástrojov, napr.: https://convertio.co/wav-mp3/.

9. Tvorba a spracovanie zvukov

Kľúčové slová

digitalizácia zvuku, zvukový formát *wav*, modul *wave*, syntéza zvuku, úprava zvuku

Čo sa naučíme a čo si precvičíme

- ešte raz sa pozrieme na zvukový formát *wav*,
- vysvetlíme si, ako Python pracuje s binárnymi dátami,
- precvičíme si načítavanie a zapisovanie binárnych dát,
- umelo (programom) vytvoríme zvukové súbory s tónmi zadaných frekvencií,
- upravíme binárne dáta zvukového súboru aplikovaním jednoduchých efektov.

Problémová situácia

V predchádzajúcej kapitole o Mozartovej hre sme viaceré zvukové súbory spájali do jedného celku. Naučili sme sa prečítať obsah vstupného súboru *wav* a zapísať ho do výstupného súboru *wav*. Používali sme funkcie a metódy z modulu *wave*.

V tejto kapitole nechceme spracúvať nahrávky zvukov, ale **chceme zvuk vytvoriť umelo – programom**. Najprv vygenerujeme jeden tón, potom celú stupnicu či kúsok melódie.

Poznámka na okraj

Vyskúšajte si online generátor tónov:

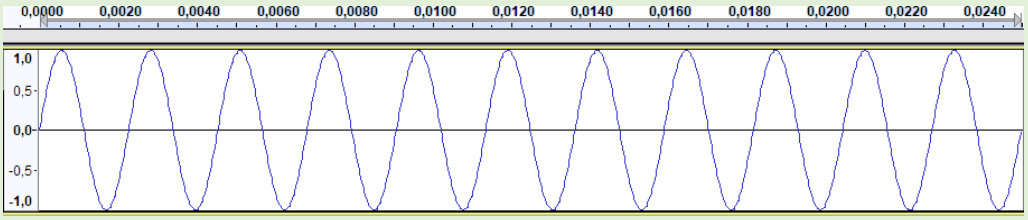
http://www.szynalski.com/tone-generator/ [30. 6. 2020]

Ktoré frekvencie tónov ste schopní počuť?

Hľadajme riešenie

Výkladový text

Hudobný tón charakterizuje frekvencia jeho zvukovej vlny. Napr. tón A je sínusoida, ktorá sa opakuje 440 krát za sekundu (má frekvenciu 440 Hz). Na obrázku nižšie vidíme 0,025 = 1/40 sekundy tónu A. Sínusoida sa zopakovala 11 krát:



Obrázok 40 Sínusoida reprezentujúca zvukovú vlnu

Funkcia sínus je periodická s periódou 2π a nadobúda hodnoty z intervalu <-1, 1>. V ukážke vyššie sme zvolili maximálnu amplitúdu, t. j. 1,0. Pri menšej hodnote (napr. 0,8 by bol tón pre poslucháča tichší).

Sínusovú vlnu tónu vieme vypočítať podľa vzorca (42):

$$y(t) = a.\sin(2.\pi.f.t),$$

kde a je amplitúda, π je Ludolfovo číslo, f je frekvencia tónu, t je čas.

Pri generovaní zvuku sa musíme rozhodnúť, pre ktoré časové okamihy budeme hodnoty funkcie sínus počítať a ukladať. Ak pracujeme so vzorkovacou frekvenciou 44100 Hz, tak na 1 sekundu pripadá 44100 vzoriek zvukovej vlny. V programe vytvárame digitalizovanú vlnu, hodnotu *t* preto musíme chápať ako podiel poradového čísla vzorky a vzorkovacej frekvencie.

Podobne ako pri digitalizovaní nahrávky, aj pri generovaní zvukovej vlny musíme uvažovať o rozlíšení vzorky. Hodnota, ktorú vypočítame, je reálne číslo. Do súboru *wav* sa však zapisuje celé číslo. Pri 16 bitovom rozlíšení ide o hodnoty z intervalu -32768..32767. Amplitúde 1,0 teda zodpovedá hodnota 32767. Ak by sme požadovali približne polovičnú hlasitosť, mohli by sme zvoliť $a = 16000$.

Úloha 1

Zapíšte časť skriptu s otvorením zvukového súboru na zápis a výpočtom sínusovej vlny pre tón zadanej frekvencie a dĺžky. Uvažujte nasledovné parametre zvuku: mono, 16 bit, 44100 Hz. Hodnoty, ktoré vypočítate, uložte do zoznamu.

Riešenie:

```
import wave
import math

vzorkovacia_f = 44100   # Hz
f_tonu = 440            # Hz (tón A)
rozlisenie = 2          # B
kanaly = 1              # mono
amplituda = 32767
dlzka_tonu = 2          # sek

ton = wave.open('ton.wav', 'wb')
ton.setparams((kanaly, rozlisenie, vzorkovacia_f, 0,
               'NONE', 'Not Compressed'))


sin_vlna = [math.sin(2 * math.pi * f_tonu * (x/vzorkovacia_f))
            for x in range(int(vzorkovacia_f * dlzka_tonu))]
```

Funkcia `math.sin` vypočíta hodnotu funkcie sínus pre uhol zadaný v radiánoch.

Podiel `x / vzorkovacia_f` predstavuje časový okamih, pre ktorý generujeme hodnotu funkcie sínus. Po vyhodnotení celého výrazu v argumente funkcie sínus získame hodnotu zodpovedajúceho uhla v radiánoch.

Zoznam `sin_vlna` má po skončení výpočtu 88200 prvkov (premenná *x* postupne nadobudla hodnoty od 0 po 88199). V našom príklade sme vytvárali 2 sekundy tónu pri vzorkovacej frekvencii 44100 Hz.

Výsledok generovania zvuku (bajty reprezentujúce jednotlivé vzorky) potrebujeme zapísať do binárneho súboru (súboru *wav*) – v našom prípade má ísť o **16 bitové znamienkové čísla** (43)**.** Ako to urobiť správne?

Úloha 2

V binárnom súbore *data.bin* je zapísaných týchto 10 bajtov:

```
A3 ED 41 77 FF C1 10 20 30 40
```

Otvorte ho v režime na čítanie a prečítajte prvé 2 bajty. Vypíšte **znamienkové celé číslo**, ktoré tieto 2 bajty reprezentujú.

Riešenie:

Analyzujme najskôr výstup tohto krátkeho skriptu:

```
fr = open('data.bin','rb')                                    #[1]
dva_bajty = fr.read(2)                                        #[2]
print(dva_bajty)                        # b'\xa3\xed'
print(dva_bajty.hex())                  # a3ed
print(int(dva_bajty.hex(), 16))         # 41965
```

[1] Binárny súbor sme otvorili na čítanie.
[2] Zo súboru `fr` sme prečítali 2 bajty.

Z prvého výpisu vidíme, čo je výsledkom metódy `read`. Pri čítaní z binárneho súboru Python vracia objekt typu *bytes*. Ide o nemennú postupnosť bajtov, ktorú reprezentuje reťazcový literál s prefixom `b`. Z reťazca `b'\xa3\xed'` vidíme, že sme zo súboru prečítali bajty `a3` a `ed`. Zápis `\x` znamená, že v reťazci nasleduje hexadecimálne číslo.

V ďalšom riadku sme z „bajtového“ objektu urobili hexadecimálny reťazec. Následne sme ho pomocou metódy `int` skonvertovali na celé desiatkové číslo. Výsledok vrátený funkciou `int` si ľahko overíme aj vlastným výpočtom:

$$(\mathrm{a3ed})_{16} = 10.16^3 + 3.16^2 + 14.16^1 + 13.16^0 = (41965)_{10}$$

Zdalo by sa, že všetko je v poriadku. Ale len do chvíle, keď si uvedomíme, aké dáta čítame. Podobne ako v tejto úlohe, aj pri čítaní zvukových dát predsa pracujeme so **znamienkovými celými číslami**. Dva bajty, ktoré sme prečítali v našom skripte, musia určite reprezentovať niektorú z celočíselných hodnôt z intervalu **-32768..32767**. Výsledok 41965 je preto zjavne nesprávny. Pri prevode čísla *a2ed* zo šestnástkovej do desiatkovej sústavy sme sa naň dívali ako na neznamienkové.

Výkladový text

Pri čítaní binárnych dát musíme vedieť, ako sa bajty požadovaného údajového typu ukladajú v pamäti.

Ak sa na najnižšiu adresu zapisuje najmenej významný bajt a zaň sa ukladajú ostatné bajty až po najvýznamnejší, hovoríme o formáte *„little endian“*. Ide o spôsob bežný pre väčšinu počítačov. Bajty nášho 16 bitového celého čísla sú v tomto prípade v binárnom súbore zapísané opačne. Pri výpočte hodnoty, ktorú kódujú, sa spracúvajú odzadu.

Poznámka na okraj

Porovnajme, ako sa bajty čísla $(2015)_{10} = (07DF)_{16}$ ukladajú v pamäti na platformách uplatňujúcich odlišnú endianitu:

Little endian DF 07
Big endian 07 DF

Pozrime sa teraz na bajty prečítané v riadku [2] ešte raz: `a3 ed`. Obráťme ich poradie: `ed a3`. Zapíšme ich binárne: `11101101 10100011`. Vidíme, že najľavejší bit je 1. Znamená to, že toto číslo **je záporné**. Ak chceme zistiť jeho absolútnu hodnotu, musíme jednotlivé bity invertovať a k výsledku pripočítať 1. Po inverzii bitov dostaneme: `00010010 01011100`. Po pripočítaní jednotky: `00010010 01011101`. Ručným výpočtom alebo pomocou kalkulačky rýchlo zistíme, že ide o číslo **-4701**.

Poznámka na okraj

Ak by išlo o kladné celé číslo (najľavejší bit by bol vtedy nulový), jeho hodnotou je pôvodné binárne číslo a žiadny ďalší výpočet nie je potrebný.

Vysvetlili sme si, čo sa musí udiať, aby sme pri čítaní bajtov z binárneho súboru získavali také hodnoty, ktoré očakávame. Python obsahuje podporu aj pre túto situáciu (44). Vyššie opísanú výmenu bajtov, kontrolu znamienka a výpočet absolútnej hodnoty za nás urobí pomocná funkcia z modulu `struct`:

```
import struct
print(struct.unpack('h', dva_bajty))        # (-4701,)
```

Funkcia `struct.unpack` má dva parametre: reťazec 'h' špecifikuje, že 2 bajty v druhom parametri potrebujeme „rozbaliť” ako 16 bitové znamienkové celé číslo. Vracia n-ticu, v našom prípade jednoprvkovú. Z nej si hodnotu -4701 prečítame pomocou indexu:

```
hodnota = struct.unpack('h', dva_bajty)[0]
print(hodnota)                              # -4701
```

Poznámka na okraj

O ďalších možnostiach funkcie `struct.unpack` sa dozviete v dokumentácii:

https://docs.python.org/3/library/struct.html [30. 6. 2020]

Naučme sa ešte, ako budeme do binárneho súboru **zapisovať 16 bitové znamienkové celé čísla**:

Úloha 3

Vytvorte binárny súbor, do ktorého zapíšete celé číslo -4701.

Riešenie:

Celočíselnú hodnotu pripravíme na zápis do binárneho súboru pomocou funkcie `struct.pack`:

```
import struct

fw = open('data2.bin','wb')
hodnota = -4701
dva_bajty = struct.pack('h', hodnota)
print(dva_bajty)
fw.write(dva_bajty)
fw.close()
```

Úloha 4

Dokončite skript z **Úlohy 1**. Vygenerovanú zvukovú vlnu jedného tónu uložte do súboru *wav*.

Úloha 5

Napíšte skript, ktorým vytvoríte celú stupnicu C dur. Frekvencie tónov uvádzame v tabuľke:

c	d	e	f	g	a	h	c
261.63	293.66	329.63	349.23	392.00	440.00	493.88	523.25

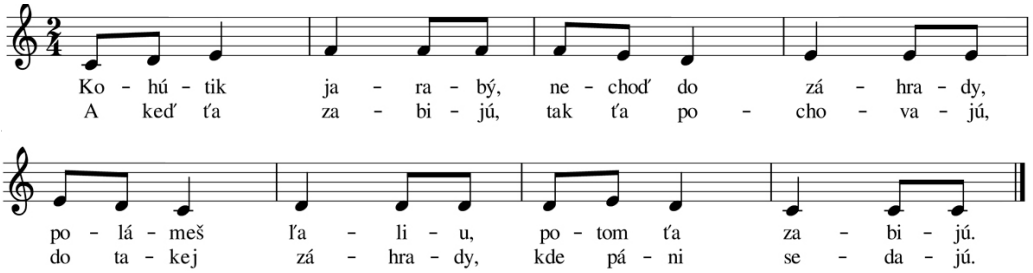
Poznámka na okraj

Frekvencie ostatných tónov nájdete napr. tu:

https://pages.mtu.edu/~suits/notefreqs.html [30. 6. 2020]

Úloha 6

Vygenerujte melódiu ľudovej piesne *Kohútik jarabý*:



Notový zápis piesne uložte v Pythone ako zoznam dvojíc:

```
kohutik = [('c',1),('d',1),('e',2),('f',2),('f',1),('f',1),
           ('f',1),('e',1),('d',2),('e',2),('e',1),('e',1),
           ('e',1),('d',1),('c',2),('d',2),('d',1),('d',1),
           ('d',1),('e',1),('d',2),('c',2),('c',1),('c',1)]
```

Kódovanie dĺžky tónu: 1 = „kratší tón“ (osminová nota), 2 = „dlhší tón“ (štvrťová nota)

Pri generovaní piesne je vhodné medzi tóny vkladať krátke pauzy (sekvencie nulových bajtov), podobne ako to prirodzene robíme pri hraní piesne na hudobnom nástroji.

Čo sme sa naučili

- Keď chceme zvuk vytvárať programom, musíme ho vedieť popísať matematickým modelom. Zvuková vlna tónu má tvar sínusoidy. Okrem frekvencie tónu a jeho dĺžky ovplyvňujú kvalitu výsledku aj ďalšie parametre: vzorkovacia frekvencia, rozlíšenie vzorky a počet kanálov. Pri generovaní zvuku postupne vypočítame hodnoty všetkých vzoriek, ktoré by vznikli procesom digitalizácie pri nahrávaní zvuku pomocou mikrofónu.
- Pri práci s binárnymi dátami musíme mať istotu, či postupnosti bajtov, ktoré zo súboru čítame alebo ktoré doň zapisujeme, reprezentujú požadovaný údajový typ. Postupnosť 2 bajtov (vypočítanú hodnotu vzorky) vieme do súboru *wav* zapísať s využitím pomocnej funkcie `struct.pack`.
- Pri úpravách zvuku môžeme využiť funkcie z modulu `audioop`.

Ďalšie úlohy na precvičenie a zamyslenie

1. Knižnica Pythonu ponúka aj modul `audioop` obsahujúci užitočné funkcie pre priamu manipuláciu so zvukovými dátami. Vyhľadajte v dokumentácii funkciu, pomocou ktorej vieme
   a. zložiť (sčítať) 2 zvukové vlny,
   b. obrátiť poradie vzoriek vo vstupnom súbore *wav*,
   c. upraviť hlasitosť zvuku.

2. Vytvorte nahrávku svojho hlasu (aspoň jednu vetu). Potom z pôvodného súboru *wav* vytvorte nový, v ktorom bude vaša nahrávka uložená odzadu. Pri testovaní môžete využiť aj také vety, ktoré znejú odpredu aj odzadu rovnako (palindromy, napr.: „Jeleňovi pivo nelej.“)

3. Vytvorte nahrávku svojho hlasu (nahrajte napr. krátku detskú básničku). Potom z pôvodného súboru *wav* vytvorte nový, v ktorom k vstupnej zvukovej vlne pridáte efekt echa (ozvenu) .

   Pomôcka: Ak sa rozhodnete pre echo s dozvukom s dĺžkou napr. 1/10 sekundy, uchovávajte si v pomocnom zozname toľko starších vzoriek spracúvanej zvukovej vlny, koľko zodpovedá tomuto časovému intervalu. Pri zápise zvuku do výstupného súboru k vzorke v čase *t* pridajte vzorku z času *t-1/10*.

10. Špecifiká floating point aritmetiky

Kľúčové slová

Reprezentácia čísel, čísla s plávajúcou rádovou čiarkou, mantisa, exponent, platné číslice, pretečenie, podtečenie, množina presne uložených desatinných čísel, IEEE 754-2008 formát.

Čo sa naučíme a čo si precvičíme

- Popísať špecifiká dátového typu `float` (formátu IEEE 754-2008) – množina presne uložených desatinných čísel, obmedzenosť rozsahu hodnôt, obmedzenosť počtu platných číslic, podtečenie (strojová nula), pretečenie.
- Vysvetliť principiálnu nepresnosť pri numerickom prístupe k riešeniu problémov s využitím dátového typu `float`.
- Aplikovať poznatky o špecifikách floating point aritmetiky pri programovaní riešení výpočtových problémov.

Zapojenie

Úloha 1

Do tabuľky doplňte k jednotlivým tvrdeniam či platia alebo nie, a tiež váš komentár.

Por.	Tvrdenie	Platí	Komentár

1.	Uvedený program **sa zacyklí** (neskončí).
	`n = 1`
	`while n > 0:`
	`  n = n / 2`
		áno – nie	

2.	Každé **reálne číslo** sa dá **presne uložiť** do pamäti pomocou niektorého neceločíselného dátového typu (napr. `float`).	áno – nie	
3.	Každé **desatinné číslo** s konečným desatinným rozvojom sa dá **presne uložiť** v pamäti pomocou niektorého neceločíselného dátového typu (napr. `float`).	áno – nie	
4.	**Desatinné čísla** uložené v dátovom type `float` **sú obmedzené** len na určitý **interval** hodnôt.	áno – nie	
5.	Niektoré **nenulové desatinné čísla** (`float`) sa uložia v pamäti ako **nula.**	áno – nie	
6.	**Rozdiely hodnôt** ľubovoľných dvojíc susedných čísel uložených v type `float` **sú rovnaké.**	áno – nie	
7.	Môže nastať situácia, že pre nejaké hodnoty desatinných čísel `a`, `b≠0` vráti príkaz `print(a + b == a)` hodnotu **`True`**?	áno – nie	
8.	Môže nastať situácia, že pre nejaké hodnoty desatinných čísel `a`, `b`, `c` vráti príkaz	áno – nie	

	`print((a + b) + c == a + (b + c))` hodnotu **`False`**?		

V nasledovných úlohách si zopakujeme, čo si pamätáte zo zápisov desatinných čísel v počítači a tiež prevody zápisov čísel medzi dvojkovou a desiatkovou sústavou.

Úloha 2

Priraďte číslam z druhého stĺpca čísla s rovnakou hodnotou v poslednom stĺpci:

A	1.234		1.234e-3
B	123.4		1.234e-2
C	0.01234		1.234e0
D	1.234×10^{-2}		1.234e1
E	1.234×10^{2}		1.234e2
F	1.234×10^{0}		1.234e3

Úloha 3

Preveďte čísla z desiatkovej do dvojkovej sústavy: 37 = 1.5 = .

Úloha 4

Preveďte čísla z dvojkovej do desiatkovej sústavy: 11011_2 = 0.1_2 = .

Úloha 5

Uveďte, ktoré obmedzenia poznáte pri programovaní výpočtov pomocou neceločíselných dátových typov (napr. `float`, `double`, `real`)?

Úloha 6

Za akú považujete problematiku floating point aritmetiky pri štúdiu programovania?

- *za nepodstatnú,*
- *za málo dôležitú,*
- *za priemerne dôležitú,*
- *za veľmi dôležitú,*
- *neviem posúdiť.*

Úloha 7

Prečítajte si a prediskutujte vybrané prípady nešťastí spôsobených numerickými chybami v počítačových systémoch:

- (25. 2. 1991) Zlyhanie systému protiraketovej obrany Patriot počas vojny v Golfskom zálive a nezostrelenie rakety, ktorá usmrtila 28 ľudí. (45)

- (4. 6. 1996) Neúspešný 37 sekundový let zavŕšený explóziou rakety Arian 5 vypustenej z ESA, ktorá znamenala premárnené 10-ročné úsilie vedcov a škodu cca 500 miliónov dolárov. (46)

Ďalšie podobné nešťastné prípady nájdete na webove stránke (47) po zadaní kľúčového slova *"rounding"* do vyhľadávacieho poľa. Ako sa teraz zmenil váš pohľad na problematiku floating point aritmetiky pri štúdiu programovania? Pokladáte ju:

- *za nepodstatnú,*
- *za málo dôležitú,*
- *za priemerne dôležitú,*
- *za veľmi dôležitú,*
- *neviem posúdiť.*

Skúmanie

Úloha 8

Do priloženej tabuľky so 4 aritmetickými výrazmi uveďte a zdôvodnite svoje predpovede výsledkov týchto výrazov. Ak ste to urobili, overte si svoje predpovede pomocou konzoly nainštalovaného interpretera Pythonu, resp. online konzoly (48) a zapíšte a zhodnoťte ich do posledných dvoch stĺpcov tabuľky.

Výraz	Predpoveď výsledku	Zdôvodnenie vašej predpovede	Skutočný výsledok	Bola správna vaša predpoveď?
0.1 + 0.2 - 0.3				áno / nie
(0.1 + 0.2) + 0.3				áno / nie
0.1 + (0.2 + 0.3)				áno / nie
1 + 1e16 > 1e16				áno / nie

Očakávali ste takéto výsledky? Stručne zdôvodnite rozdiely medzi vašimi očakávaniami a skutočnosťou.

Poďme teraz preskúmať špecifiká floating point aritmetiky v interpreteri Pythonu.

Úloha 9

Zadajte v konzole nasledovné výrazy. Výsledky výrazov spolu so zdôvodnením zapíšte do tabuľky.

Výraz	Výsledok	Komentár / zdôvodnenie
1.7e308		
1.8e308		
2*1.7e308		
2**1024		
2.0**1024		

Zo zistených výsledkov výrazov vyslovte svoj záver k **rozsahu desatinných čísel**:

Úloha 10

Zadajte v konzole nasledovné výrazy. Výsledky výrazov spolu so zdôvodnením zapíšte do tabuľky.

Výraz	Výsledok	Komentár / zdôvodnenie
5e-324		
3e-324		
2e-324		
3e-324/2		
1/1.7e308		
1/1.8e308		
1e-309		
1/1e309		

Zo zistených výsledkov výrazov vyslovte svoj záver k **najmenším nenulovým hodnotám desatinných čísel**:

Úloha 11

Zadajte v konzole nasledovné výrazy. Výsledky výrazov spolu so zdôvodnením zapíšte do tabuľky.

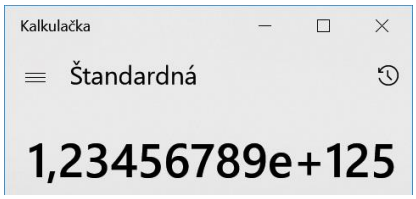
Výraz	Výsledok	Komentár / zdôvodnenie
9007199254740990.0 + 1		
9007199254740991.0 + 1		
9007199254740992.0 + 1		
9007199254740993.0 + 1		
9007199254740993.0		
2**(-52)		
2**(0) +2**(-52)		
2**(-53)		
2**(0) +2**(-53)		
2**(10) + 2**(-42)		
2**(10) + 2**(-43)		

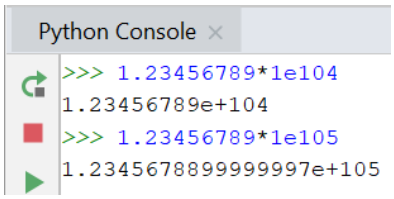
Zo zistených výsledkov výrazov vyslovte svoj záver k **počtu platných číslic desatinných čísel**:

Vysvetlenie

Úloha 12

Prediskutujte v skupinách čo ste zistili pri riešení úloh 8 až 11 a uveďte, ku ktorým novým poznatkom ste dospeli.
Poďme sa spolu pozrieť na desatinné čísla ako sú uložené v pamäti počítača.





V programe Kalkulačka aj v konzole jazyka Python používame pri výpočtoch s desatinnými číslami tzv. **zápis s plávajúcou rádovou čiarkou** (alebo tiež **semilogaritmický zápis**). Tento zápis čísla $1.23456789 \times 10^{125}$ pozostáva z 2 častí, tzv. **mantisy** (1.23456789) a **exponenta** (125).

V starších verziách Pythonu sa rozlišovali 2 typy desatinných čísel: `float` (32-bitový) a `double` (64-bitový). Čísla typu `float` boli v pamäti uložené podľa schémy (49):

$$\underbrace{\mathrm{S}}_{\substack{\text{znamienko}\\1}}\ \underbrace{\mathrm{EEEEEEEE}}_{\substack{\text{exponent}\\8}}\underbrace{\mathrm{MMMMMMMMMMMMMMMMMMMMMMM}}_{\substack{\text{mantisa}\\23}}$$

V aktuálnych verziách Pythonu sa pod typom `float` označujú desatinné čísla s dvojitou presnosťou (64-bitové), ktoré sú v pamäti uložené podľa schémy:

$$\underbrace{\mathrm{S}}_{\substack{\text{znamienko}\\1}}\ \underbrace{\mathrm{EEEEEEEEEEE}}_{\substack{\text{exponent}\\11}}\underbrace{\mathrm{MMMMMMMMMM\ldots MMMMMMMMMM}}_{\substack{\text{mantisa}\\52}}$$

Teraz sa spoločne pozrime na to, ako je v pamäti uložené to „nešťastné“ číslo **0.1**, ktoré súviselo so zlyhaním protiraketového systému Patriot v roku 1991 (50). Pozrime sa ako vyzerá zápis tohto čísla v dvojkovej sústave. Pri prevode desatinných čísel z desiatkovej do dvojkovej sústavy budeme postupovať tak, že budeme postupne vynásobovať dvojkou zlomkovú časť čísla. V každom novom riadku budeme vynásobovať číslo menšie ako 1. Vedľa vynásobeného čísla zapíšeme 0, ak neprevyšuje 1, alebo zapíšeme 1, ak ju prevyšuje. Postupnosť týchto binárnych čísel 0 a 1 počínajúcu prvým riadkom bude tvoriť desatinnú časť dvojkového čísla.

$$\begin{aligned}
0.1 \times 2 &= 0.2 + 0\\
0.2 \times 2 &= 0.4 + 0\\
0.4 \times 2 &= 0.8 + 0\\
0.8 \times 2 &= 0.6 + 1\\
0.6 \times 2 &= 0.2 + 1
\end{aligned}$$

Posledné 4 riadky sa budú cyklicky opakovať. Spísaním číslic v pravom stĺpci zhora nadol dostaneme:

$$0.1_{10} = 0.\overline{00011}_2 \text{ (resp. } 0.1_{10} = 0.000\overline{1100}_2 \text{ s periódou začínajúcou } 1)$$

Nastala podobná situácia ako s číslom 1/3 či 2/3, ktoré majú v desiatkovej sústave nekonečný periodický rozvoj. Na displejoch kalkulačiek sa zvyknú tieto čísla odrezať alebo zaokrúhliť na určitý počet desatinných miest. V našom prípade budeme musieť dvojkový nekonečný rozvoj čísla 0.1 (ktoré má konečný desiatkový rozvoj) zaokrúhliť na niektorom desatinnom mieste. Znamená to, že číslo 0.1 sa nedá presne uložiť v dvojkovej sústave, namiesto neho sa v pamäti uloží jeho zaokrúhlená hodnota.

Tu si treba uvedomiť, že nasledovné tri čísla sa v pamäti uložia do jedného čísla, ktoré je totožné s tretím z uvedených čísel, ktoré sa dá presne uložiť v pamäti počítača v dvojkovej sústave (ako hodnota typu `float`):

```
0.1
0.10000000000000001
0.1000000000000000055511151231257827021181583404541015625
```

Celkovo môžeme povedať, že len niektoré čísla sa uložia presne v pamäti v dátovom type `float`. Preto pri uložení reálnych čísel treba uvažovať podmnožinu racionálnych čísel M s presne uloženými hodnotami v dátovom type `float`.

Evidentne $0.1 \notin \mathrm{M}$. Zaujímavé je, že pri výpise čísla z tejto množiny môžeme dostať číslo mimo tejto množiny, v našom prípade pri uložení hodnoty 0.1 do pamäti sa uloží ako najbližšie presne uložené číslo $0.1000000000000000055511151231257827021181583404541015625$, ktoré sa pri výpise zobrazí zaokrúhlene ako 0.1).

Poďme sa bližšie pozrieť ako je číslo 0.1 uložené v pamäti počítača.

$$0.1_{10} = 0.000\overline{1100}_2 = \frac{1100110011001100110011001100110011001100110011001101_2}{10000000000000000000000000000000000000000000000000000_2}$$

Číslo 0.1 je vo (64-bitovom) `float` formáte uložené nasledovne:

0**0111111**, **10111001**, **10011001**, **10011001**, **10011001**, **10011001**, **10011001**, **10011010**

Znamienko: 0 (je to kladné číslo)

Exponent: 011 1111 1011_2 = 1019
(hodnotu dvojkového exponentu dostaneme odpočítaním čísla 1023)

1019 – 1023 = –4

Mantisa: 1**1001**, **10011001**, **10011001**, **10011001**, **10011001**, **10011001**, **10011010**

Každé nenulové číslo v dvojkovej sústave sa začína číslicou 1. V mantise sa táto číslica neuvádza. Posledné číslice ...01 sa vzhľadom na nasledujúcu číslic **1** mimo pamäť (...01**1**) zaokrúhlia na ...10.

Vo všeobecnosti je desatinné číslo uložené v pamäti ako 64-bitový `float` nasledovne:

$$(-1)^{\text{znamienko}} \times 2^{\text{exponent-1023}} \times 1.\,\text{mantisa}$$

Exponent veľkosti 11 bitov môže nadobúdať hodnoty od 00000000000 až po 11111111111, čo je v desiatkovej sústave 0 až 2047. Tieto hraničné hodnoty exponenta sa používajú len pre špeciálne hodnoty dátového typu `float`. Pozrime sa na nich bližšie. Užitočnou pomôckou pre nás je vlastný program `float_format.py`.

Aký výsledok vráti funkcia `bin2float()` pre nasledovné zoznamy reprezentujúce 64-bitový `float`?

```
['01111111','11110000','00000000','00000000','00000000','00000000','00000000','00000000']
['11111111','11110000','00000000','00000000','00000000','00000000','00000000','00000000']
['01111111','11110001','00000000','00000000','00000000','00000000','00000000','00000000']
['11111111','11110001','00000000','00000000','00000000','00000000','00000000','00000000']
['00000000','00000000','00000000','00000000','00000000','00000000','00000000','00000000']
['10000000','00000000','00000000','00000000','00000000','00000000','00000000','00000000']
['00000000','00000000','00000000','00000000','00000000','00000000','00000000','00000001']
```

Riešenie:

Pre maximálnu hodnotu exponenta (11 číslic 1), nulovú hodnotu mantisy (52 číslic 0) a kladné znamienko (hodnota 0) dostaneme výsledok `inf` (nekonečno). Pri zápornom znamienku (hodnota 1), dostaneme výsledok `-inf` (-nekonečno).

Pre maximálnu hodnotu exponenta (11 číslic 1) a nenulovú hodnotu mantisy dostaneme výsledok `NaN` (Not a Number). Táto špeciálna hodnota je rezervovaná pre výsledky nedefinovaných operácií, napr. delenie nulou, odmocninu záporného čísla, rozdiel ∞-∞ (čo zapíšeme v konzole ako: `float ('inf') - float ('inf')`) atď.

Pre nulovú hodnotu exponenta (11 číslic 0), nulovú hodnotu mantisy (52 číslic 0) a kladné znamienko (hodnota 0) dostaneme výsledok `0` (tento výsledok dostaneme aj po zápise do konzoly výrazu: `1/float('inf')`).

Pre nulovú hodnotu exponenta (11 číslic 0), nulovú hodnotu mantisy (52 číslic 0) a záporné znamienko (hodnota 1) dostaneme výsledok `-0` (tento výsledok dostaneme aj po zápise do konzoly výrazu: `1/float('-inf')`).

Pre nulovú hodnotu exponenta (11 číslic 0) a nenulovú hodnotu mantisy dostaneme tzv. **subnormálne čísla**, ktorých hodnota je menšia ako hodnota normálnych čísel (t.j. s exponentom neobsahujúcim samé 0, ani samé 1).

Subnormálne čísla sú v pamäti uložené ako 64-bitový `float` nasledovne:

$$(-1)^{\text{znamienko}} \times 2^{1-1023} \times 0.\text{mantisa}$$

Rozpracovanie

Pri riešení nasledovných úloh môžete:

- použiť hotové programy `float_format.py`, `odcitavanie1.py`, `rovnost_vyrazov.py`, `odcitavanie2.py`, `pocetnosti_číslic.py`,
- použiť niektorý z online appletov (51) (52) (53) (54)
- prípadne vytvoriť vlastné programy.

Úloha 13

Ktoré z uvedených reálnych čísel patria do množiny presne uložených čísel v 64-bitovom type `float`:

číslo	**odhadnite** či je presne uložené v type `float`	pomocou programu/ appletu **overte** presnosť uloženia tohto čísla	**zdôvodnite** zistený výsledok
0.1	je – nie je	je – nie je	
0.2	je – nie je	je – nie je	
0.5	je – nie je	je – nie je	
0.6	je – nie je	je – nie je	
0.75	je – nie je	je – nie je	
0.8	je – nie je	je – nie je	
0.9375	je – nie je	je – nie je	

Úloha 14

1. Zistite:
   a. Aké **najväčšie číslo** sa dá uložiť v 64-bitovom dátovom type `float` ?
   b. Aké **najmenšie normálne číslo** sa dá uložiť v 64-bitovom dátovom type `float`?
   c. *Aké **najmenšie subnormálne číslo** sa dá uložiť v 64-bitovom dátovom type `float`?
   d. Aké sú **najväčšie rozdiely** medzi dvomi **susednými číslami** uloženými v 64-bitovom dátovom type `float`?
   e. Aké sú **najmenšie rozdiely** medzi dvomi **susednými normálnymi číslami** uloženými v 64-bitovom dátovom type `float`?
   f. *Aké sú **najmenšie rozdiely** medzi dvomi **susednými subnormálnymi číslami** uloženými v 64-bitovom dátovom type `float`?

Úloha 15

Reálne čísla v matematike môžeme zobraziť ako body na priamke.

Nakreslite množinu s hodnotami dátového typu `float`.

Úloha 16

Čo robí uvedený program (uložený v súbore `odcitavanie1.py`)? Prediskutujte ako vylepšiť tento program.

```
number = 1.0
while number != 0:
    number = number - 0.1
    print(number)
```

Úloha 17

Otvorte súbor `odcitavanie2.py`, porovnajte v ňom funkcie `generate_adding()`, `generate_multiplying()` a `generate_division()` a prediskutujte ich využitie pri iných podobných úlohách.

```
def generate_adding():
    numbers = []
    number = 0.0
    for i in range(1, 11):
        number = number + 0.1
        numbers.append(number)
    return numbers


def generate_multiplying():
    return [i * 0.1 for i in range(1, 11)]


def generate_division():
    return [i / 10 for i in range(1, 11)]


print(generate_adding())
print(generate_multiplying())
print(generate_division())
```

Úloha 18

(Riešenie lineárnej rovnice)

a. Vytvorte program pre nájdenie riešenia lineárnej rovnice tvaru $ax = b$.
b. Nájdite také hodnoty koeficientov a, b, aby výpočet na počítači dával iný (nesprávny) výsledok ako presný matematický výpočet.

Úloha 19

Otvorte súbor `rovnost_vyrazov.py`, porovnajte v ňom funkcie `verify_float()`, `verify_decimal()` a `verify_decimal2()` a prediskutujte ich využitie pri iných úlohách.

```
import decimal


def verify_float(a, x, b):
    return a * x == b


def d(value):
    return decimal.Decimal(str(value))


def verify_decimal(a, x, b):
    return d(a) * d(x) == d(b)


def verify_decimal2(places, a, x, b):
    return int((10 ** places) * a) * int((10 ** places) * x) /
((10 ** places)) == int((10 ** places) * b)


print(verify_float(1.12, 3, 3.36))
print(verify_decimal(1.12, 3, 3.36))
print(verify_decimal2(2, 1.12, 3, 3.36))
print(verify_float(0.15, 3, 0.45))
print(verify_decimal(0.15, 3, 0.45))
print(verify_decimal2(2, 0.15, 3, 0.45))
print(verify_decimal2(1, 0.15, 3, 0.45))
```

Úloha 20

(Určenie kolmosti dvoch vektorov v rovine)

a. Vytvorte program, ktorý pre dva zadané vektory $u = (u_x, u_y)$ a $v = (v_x, v_y)$ určí či sú na seba kolmé alebo nie.
   (Poznámka: Pre dva kolmé vektory (u_x, u_y) a (v_x, v_y) platí rovnosť $u_x v_x + u_x v_y = 0$).
b. Nájdite také hodnoty súradníc vektorov, pre ktoré by program uviedol, že sú kolmé, pričom v skutočnosti by neboli kolmé. Rovnako nájdite súradnice, pre ktoré by program uviedol, že nie sú kolmé, pričom by v skutočnosti boli kolmé.
c. Upravte program na základe riešenia predchádzajúcej úlohy, aby vracal správne výsledky.

Úloha 21

2. Otvorte súbor `pocetnosti_číslic.py` a na základe experimentovania s ním odpovedzte na nasledovné otázky:
   a. Aký je rozdiel medzi hodnotami `2.0 ** 1023` a `2 ** 1023`?
   b. Aké sú obmedzenia výpočtu vzhľadom k veľkosti čísla uloženého v premennej `number`?
   c. Ak máme celé číslo uložené v zozname, aké jeho charakteristiky vieme pomerne ľahko vypočítať?
   d. *Prečo sme na výpočet frekvenčnej analýzy jednotlivých číslic použili dátovú štruktúru slovník?

Úloha 22

* Okrem numerických výpočtov existuje aj prístup využívajúci symbolickú manipuláciu, napr. pri riešení rovníc, zjednodušení výrazov, vyhodnotení výrazov s iracionálnymi číslami atď. Symbolickú manipuláciu umožňujú tzv. Computer Algebra Systems (napr. Geogebra). V Pythone na symbolickú manipuláciu slúži modul SymPy. V tutoriále SymPy (55) nájdite a prediskutujte aké typy úloh sa dajú riešiť pomocou symbolickej manipulácie.

Vyhodnotenie

Úloha 23

Do tabuľky doplňte k jednotlivým tvrdeniam či platia alebo nie, a tiež váš komentár.

Por.	Tvrdenie	Platí	Komentár

1.	Uvedený program **sa zacyklí** (neskončí).
	`n = 1`
	`while n > 0:`
	`    n = n / 2`
		áno – nie	

2.	Každé **reálne číslo** sa dá **presne uložiť** do pamäti pomocou niektorého neceločíselného dátového typu (napr. `float`).	áno – nie	
3.	Každé **desatinné číslo** s konečným desatinným rozvojom sa dá **presne uložiť** v pamäti pomocou niektorého neceločíselného dátového typu (napr. `float`).	áno – nie	
4.	**Desatinné čísla** uložené v dátovom type `float` **sú obmedzené** len na určitý **interval** hodnôt.	áno – nie	
5.	Niektoré **nenulové desatinné čísla** (`float`) sa uložia v pamäti ako **nula.**	áno – nie	

6.	**Rozdiely hodnôt** ľubovoľných dvojíc susedných čísel uložených v type `float` **sú rovnaké.**	áno – nie	
7.	Môže nastať situácia, že pre nejaké hodnoty desatinných čísel `a`, `b≠0` vráti príkaz `print(a + b == a)` hodnotu **`True`**?	áno – nie	
8.	Môže nastať situácia, že pre nejaké hodnoty desatinných čísel `a`, `b`, `c` vráti príkaz `print((a + b) + c == a + (b + c))` hodnotu **`False`**?	áno – nie	

Úloha 24

Zosumarizujte cca 10 vetami svoje súčasné poznanie špecifík floating point aritmetiky.
Uveďte tiež ako budete odteraz riešiť výpočtové úlohy pomocou programovacieho jazyka?

Úloha 25

Uveďte, čomu ešte nerozumiete z problematiky špecifík floating point aritmetiky.

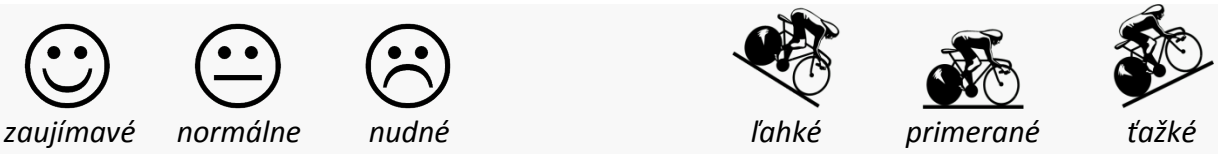
Úloha 26

Posunul vás dopredu táto kapitola v problematike špecifík floating point aritmetiky?

- *áno, veľmi,*
- *áno, ale len trochu,*
- *neviem posúdiť,*
- *ani nie,*
- *vôbec nie.*

Úloha 27

Aké boli pre vás tieto úlohy? Zaujímavé? Ľahké? Zafarbite/zakrúžkujte niektorú z uvedených možností:



Čo sme sa naučili:

- V dátovom type `int` sú uložené celé čísla prakticky bez obmedzenia rozsahu, čo umožňuje presné výpočty s nimi.
- V 64-bitovom dátovom type `float` sú presne uložené len niektoré desatinné čísla, a to tie, ktoré sa dajú presne vyjadriť pomocou 64-bitového zápisu. Takúto množinu presne uložených desatinných čísel označujeme ako M.
- Ak ukladáme do pamäti čísla, ktoré nie sú z množiny M (napr. $0.1_{10} = 0.000\overline{1100}_2$), tak tie sa pri ukladaní nahradia čo najbližšími desatinnými číslami z množiny M (napr. $0.1000000000000000055511151231257827021181583404541015625$). Pri výpise hodnôt čísel z množiny M sa tieto hodnoty môžu zaokrúhliť na určitý počet desatinných miest čím môžeme dostať výsledok, ktorý nie je z množiny M.
- Desatinné čísla s konečným desatinným rozvojom v desiatkovej sústave nemusia mať konečný desatinný rozvoj v dvojkovej sústave (napr. $0.1_{10} = 0.000\overline{1100}_2$), čo znamená, že nebudú v type `float` uložené presne a ani výpočty s nimi nebudú presné.
- Desatinné číslo uložené v 64-bitovom dátovom type `float` ma v 1 bite uložené svoje znamienko, v 11 bitoch exponent a v 52 bitoch má uloženú mantisu. Hodnota (normálneho) desatinného čísla sa z jeho binárneho zápisu vypočíta ako:
$$(\text{-}1)^{\text{znamienko}} \times 2^{\text{exponent-1023}} \times 1.\,\text{mantisa}.$$
- Číslo s 52-bitovou mantisou a uvádzajúcou 1 má spolu 53 bitov čomu odpovedá 16-ciferné číslo v desiatkovej sústave ($2^{53} = 10^{\log_{10} 2^{53}} = 10^{53\log_{10} 2} \approx 10^{15.95}$). Dôsledkom toho je, že pri výpočtoch s číslami typu `float` môžeme rátať nanajvýš so 16 platnými číslicami. Ďalším dôsledkom toho je, že pri sčitovaní čísel, ktorých rády sa líšia o 16 a viac dekadických miest menšie číslo sa nepričíta k väčšiemu, t.j. súčet týchto dvoch čísel je rovný väčšiemu z týchto čísel.
- Pre desatinné čísla sa používa exponent v rozsahu 1 až 2046. Potom číslo s najväčším exponentom $2^{2046\text{-}1023} = 2^{1023}$ a najväčšou mantisou $1.1111\ldots \approx 2$ je približne rovné $2^{1024} = 10^{1024\log_{10} 2} \approx 10^{308.25472} = 10^{0.25472} \times 10^{308} \approx 1.7977 \times 10^{308}$. Číslo s najmenším exponentom je $2^{1\text{-}1023} = 2^{\text{-}1022} = 10^{\text{-}1022\log_{10} 2} \approx 10^{\text{-}307.65266} = 10^{0.34734} \times 10^{\text{-}308} \approx 2.22507 \times 10^{\text{-}308}$. Ak pri výpočtoch dostaneme väčšie ako najväčšie uložené desatinné číslo dôjde k tzv. pretečeniu rozsahu.
- Pri výpočtoch s desatinnými číslami sa môžeme stretnúť aj s číslami menšími ako $2.22507 \times 10^{\text{-}308}$, tzv. subnormálnymi číslami, ktorých hodnota sa z ich binárneho zápisu vypočíta ako: $(\text{-}1)^{\text{znamienko}} \times 2^{1\text{-}1023} \times 0.\,\text{mantisa}$. Najmenšie subnormálne číslo má exponent -1022 a mantisu $2^{\text{-}52}$. Jeho hodnota je $2^{1\text{-}1023} \times 2^{\text{-}52} = 2^{\text{-}1074} = 10^{\text{-}1074\log_{10} 2} \approx 10^{\text{-}323.306215} = 10^{0.693785} \times 10^{\text{-}324} \approx 4.94066 \times 10^{\text{-}324}$. Ak pri

výpočtoch dostaneme menšie ako najmenšie uložené desatinné číslo dôjde k tzv. podtečeniu rozsahu a výsledkom bude 0.

- Špeciálnymi hodnotami sú `inf` (∞, nekonečno) resp. `-inf` (-∞, -nekonečno) pre pretečený rozsah, ktoré majú v type `float` v exponente uložené samé 1 (t.j. exponent 2047) a v mantise samé 0. Ďalšou špeciálnou hodnotou je `NaN` (Not a Number), ktorá je rezervovaná pre výsledky nedefinovaných operácií, napr. deleniu nulou. V dátovom type `float` má `NaN` v exponente uložené samé 1 (t.j. exponent 2047) a v mantise je nenulová.
- Pri výpočtoch s desatinnými číslami musíme rátať s obmedzeným počtom platných číslic a obmedzeným číselným rozsahom a tým aj s nepresnými a nesprávnymi výsledkami vzhľadom k matematickej aritmetike (napr. neplatnosť asociatívneho zákona, súčet dvoch kladných čísel je menší ako jeden z jeho sčítancov, podielom dvoch kladných čísel je 0).
- Pri programovaní výpočtových problémov s desatinnými číslami uloženými v type `float` treba byť obozretní a zvažovať, či nám bude postačovať rozsah čísel a počet platných číslic. Pri podmienkach namiesto rovnosti odporúčame použiť nerovnosť (pre vyvarovanie sa zacykleniu). Obzvlášť pri rozhodovacích úlohách s binárnou odpoveďou (pravda/nepravda) odporúčame namiesto dátového typu `float` ďalšie alternatívne prístupy ako napr. použitie dátového typu `int`, použitie modulu decimal či fractions, prípadne použitie symbolickej manipulácie.

11. Šifry a tajné správy

Kľúčové slová

šifra, šifrovací algoritmus, kľúč, kryptografia, kryptológia, kryptoanalýza, Cézarova šifra, šifrovanie s posunom, Vigenèrova šifra, Vernamova šifra, Cardanova šifra, frekvenčná analýza

Čo sa naučíme a čo si precvičíme

- vysvetliť a implementovať jednoduché šifrovacie algoritmy,
- analyzovať slabé miesta vybraných šifrovacích algoritmov,
- teoreticky navrhnúť spôsoby ako prelomiť konkrétne šifry,
- pri vybraných algoritmoch demonštrovať prelomenie šifry,
- transformovať štruktúrované dáta do podoby, ktorá zjednoduší manipuláciu s nimi,
- používať správne postupy generovania pseudonáhodných hodnôt pre potreby kryptografie.

Problémová situácia

Slovné spojenie „Informácie hýbu svetom“ sme už čítali alebo počuli nespočetne veľakrát. Ak začneme skúmať, kedy sa informácie stali takými dôležitými, zistíme, že to bolo už veľmi, veľmi dávno. Ak informácie hýbu svetom, potom je užitočné informácie mať a nedeliť sa o ne s ostatnými. Veď, kto by sa rád delil o moc? Napriek tomu potrebujeme nejakých „spojencov“ a potrebujeme si s nimi informácie vymieňať. Dostávame sa tak do situácie, kedy by sme chceli dosiahnuť na prvý pohľad dva protichodné ciele:

- informácie si chceme s priateľmi vymieňať,
- nechceme, aby sa počas tejto výmeny informácie dozvedeli aj ostatní.

Ak si odmyslíme situáciu, že spojencovi tajnú informáciu šepkáme do ucha, musíme vymyslieť spôsob ako informáciu preniesť tak, aby sa o nej niekto nepovolaný nedozvedel. Touto problematikou sa ľudstvo zaoberá tak dlho, až z toho vznikla zaujímavá vedná disciplína – kryptológia.

Výkladový text

Kryptológia je veda o utajení obsahu správ. Ak by sme mali túto vedu niekam zaradiť, zrejme by sme ju našli niekde medzi matematikou a informatikou. K matematike patrí preto, lebo spôsoby utajenia správ využívajú matematické výpočty a matematické dôkazy, a k informatike preto, lebo tieto výpočty sú tak náročné, že ľudská sila na to nestačí. V kryptológii nás budú zaujímať dve z jej oblastí – kryptografia a kryptoanalýza

Kryptografia skúma a navrhuje spôsoby ako správy utajiť – zašifrovať a overiť ich originalitu.

Kryptoanalýza skúma spôsoby ako utajené správy dešifrovať a preniknúť k ich nezašifrovanému obsahu aj v prípade, že nie sú určené pre nás.

Základné pojmy z tejto oblasti, príklady a kódy programov niektorých šifier môžete nájsť v (56).

V tejto kapitole sa pozrieme na niektoré spôsoby šifrovania a vytvoríme si nástroje, pomocou ktorých budeme vedieť správy šifrovať aj dešifrovať. Ukážeme si aj slabé stránky našich šifier a implementujeme šifrovanie, ktoré dnes vedci považujú principiálne za neprelomiteľné.

Otázky pre žiaka

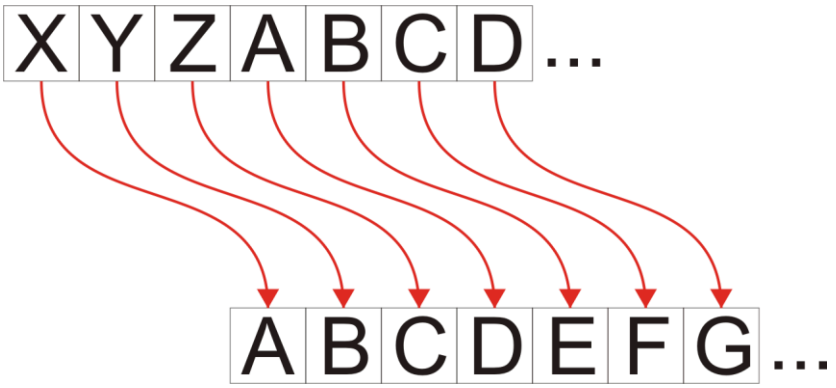
Aké spôsoby šifrovania správ poznáte?

Čo majú jednotlivé spôsoby spoločné a v čom sa líšia?

Používate šifrovanie? Kde a ako?

Cézarova šifra

Cézarova šifra (57) je jednoduchá substitučná šifra. Spočíva v tom, že každé písmeno správy nahradíme písmenom, ktoré je v abecede o tri miesta ďalej. Ak by sme sa dostali za posledné písmeno abecedy, budeme pokračovať od jej začiatku.



Obrázok 41 Princíp Cézarovej šifry - posun písmen v abecede

Úloha 1

Vytvorme trochu prepracovanejšiu verziu Cézarovej šifry. Neuvažujme len našu abecedu, ale aj ďalšie dostupné znaky utf-8 (58).

Výkladový text

V kódovacej tabuľke sú znaky umiestnené v určenom poradí. Ak chceme zistiť, aké poradové číslo prislúcha danému znaku, využijeme funkciu `ord()`. Naopak, ak potrebujeme zistiť, aký znak sa nachádza na konkrétnej pozícii, využijeme funkciu `chr()`. Prípustné hodnoty pre funkciu `chr()` sú z intervalu <0, 1114111).

```
>>> ord('č')
269
>>> chr(269)
'č'
```

Nie všetky kódy ktoré sú zahrnuté v UTF-8 budú pre potreby šifrovania vhodné. Niektoré kódy nie sú určené pre kódovanie znakov. Budeme preto uvažovať len znaky, ktorých ordinálne hodnoty sú z intervalu <0, 55296).

Na tomto mieste je potrebné uvedomiť si, že šifrovaný text nie je určený na zobrazenie – čítanie používateľom. Sú to dáta, ktoré chceme primárne uchovávať v počítači.

Viac na https://docs.python.org/3/library/functions.html [28. 5. 2019].

Vytvorte funkcie `cezar_sifruj()` a `cezar_desifruj()`, ktoré zašifrujú a dešifrujú zadaný text Cézarovou šifrou.

Vylepšená Cézarova šifra

Tento spôsob šifrovania je zaujímavý tým, že tajný je samotný algoritmus šifrovania (posúvaj o 3 znaky). Ak sa tento algoritmus prezradí, musíme vymyslieť iný spôsob. To je dosť nepríjemné, lebo vymýšľať stále nové algoritmy nie je jednoduchá vec. Môžeme preto spraviť na prvý pohľad nelogickú vec, zverejníme algoritmus: posúvame znaky v abecede. To, čo utajíme, bude kľúč: o koľko tie znaky posúvame. Samozrejme, že neostaneme len pri čísle 3, ale budeme voliť rôzne posuny. Šifry, v ktorých sa rovnaký kľúč používa pre šifrovanie aj pre dešifrovanie, sa nazývajú symetrické.

Zverejniť šifrovací algoritmus má aj ďalšiu výhodu. Môžeme ho vystaviť verejnej diskusii a tak rýchlejšie odhaliť jeho slabiny.

Úloha 2

Úpravou funkcií `cezar_sifruj()` a `cezar_desifruj()` vytvorte nové funkcie `sifruj_s_posunom()` a `desifruj_s_posunom()`. Okrem textu teraz budeme zadávať aj veľkosť posunu.

Poznámka na okraj

Jednoduchšiu verziu šifrovania s posunom môžeme realizovať aj mimo počítača pomocou šifrovacieho kotúča. Jednu z jeho verzií nájdeme napr. na https://di.ics.upjs.sk/palmaj/zadania/2006_2007/4/riesenie_komentare/sifrator.pdf [28. 5. 2019].

Prelamujeme vylepšenú Cézarovu šifru

Pozrime sa na bezpečnosť tejto šifry. Algoritmus je verejný. Ak chceme správu dešifrovať potrebujeme vedieť aký kľúč sa pri jej šifrovaní použil. Ako ho nájsť? Máme niekoľko možností:

- Použijeme hrubú silu a vyskúšame všetky možné kľúče. Tých je toľko, koľko je znakov v našej množine znakov, t.j. 55396. Toto by sa zrejme dalo, ale bolo by to príliš zdĺhavé. Zašifrovanú správu by sme mali pre každý kľúč dešifrovať a zistiť, či dáva zmysel.
- Môžeme k tomu pristúpiť trochu sofistikovanejšie. Využijeme frekvenčnú analýzu. Táto myšlienka spočíva v nasledovnej úvahe:

  V danom jazyku sa hlásky (resp. znaky) nevyskytujú s rovnakou pravdepodobnosťou. Niektoré sa vyskytujú častejšie (napr. „a“), iné menej často (napr. „w“). Ak poznáme frekvencie výskytov hlások v danom jazyku, môžeme hľadať, či v šifre majú nejaké hlásky podobnú frekvenciu výskytu. Ak áno, overme, či tieto dva znaky neurčujú posun. Samozrejme že frekvencie v danom jazyku nebudú presne zodpovedať frekvenciám

v šifrovanom texte. Budú sa len približovať. Čím dlhší je zašifrovaný text, tým viac sa budú frekvencie približovať.

Úloha 3

V súbore sifra.txt je výsledok šifrovania nejakého slovenského textu – správy. Vieme, že na šifrovanie autor použil algoritmus šifrovania s posunom. Aký je záver zašifrovanej správy?

Predpokladáme, že správu sa vám podarilo dešifrovať a poznáte aj jej záver. V skutočnosti kryptoanalýza nie je taká jednoduchá. My sme si ju zjednodušili tým, že vieme aký šifrovací algoritmus autor použil, a vieme aj to, že komunikuje v slovenčine.

Vigenèrova šifra

Pozrime sa na to, či by sme vedeli predchádzajúcu šifru ešte vylepšiť. Jej slabinou bolo, že percentá výskytu znakov sa nezmenili. Zmenili sa len znaky. Čo ak by sme znaky v tabuľke posúvali, ale každý o iný počet znakov.

Napr. pre kľúč 10, 5, 13 by sme šifrovali nasledovne:

- nultý znak posunieme o 10 miest
- prvý znak posunieme o 5 miest
- druhý znak posunieme o 13 mieste
- tretí znak posunieme o 10 miest
- štvrtý znak posunieme o 5 miest
- ...

Pamätanie kľúča si môžeme uľahčiť tak, že si nebudeme pamätať čísla ale reťazec - heslo. Znaky hesla, resp. ich ordinálne hodnoty, budú definovať kľúč.

Napr. heslo „tajné“ definuje kľúč „116, 97, 106, 110, 233“

Podobné vylepšenie používa Vigenèrova šifra (59). Tá však pracuje len s 26 znakmi abecedy a kľúč je definovaný len pomocou znakov abecedy.

Úloha 4

Vytvorte funkcie `sifruj_s_heslom()` a `desifruj_s_heslom()`, pomocou ktorých budeme vedieť zašifrovať a dešifrovať text tak, že znaky hesla určujú posuny znakov v texte.

Zamyslime sa, ako sme touto úpravou vylepšili bezpečnosť šifrovania. Ak by sme chceli šifru prelomiť hrubou silou, potrebovali by sme teoreticky overiť $55296^{\text{dĺžka hesla}}$ rôznych kombinácií. Na druhej strane existuje možnosť, že používateľ si ako heslo zvolil nejaké existujúce slovo (slovníkové heslo), takže by sme mohli použiť slovníkový útok. Vyskúšali by sme všetky slová zo slovníka použiť ako heslo. To ale nie je problém metódy ako takej, ale jej nesprávneho použitia.

Veľmi nám nepomôže ani to, ak by sme použili frekvenčnú analýzu tak, ako v predchádzajúcom prípade. Jeden a ten istý znak sa mohol nahradiť rôznymi znakmi.

Existujú však možnosti, ako šifru prelomiť. Ak sme napr. zvolili heslo dĺžky 5, tak znaky pôvodnej správy môžeme rozdeliť do pätíc:

- znaky v poradí 0, 5, 10, 15 ...
- znaky v poradí 1, 6, 11, 16 ...
- znaky v poradí 2, 7, 12, 17 ...
- znaky v poradí 3, 8, 13, 18 ...
- znaky v poradí 4, 9, 14, 19 ...

V každej pätici sa znaky posúvajú o rovnaký posun. Ak tieto pätice lokalizujeme v zašifrovanej správe, tak na každú päticu môžeme použiť frekvenčnú analýzu podobne ako v predchádzajúcom príklade.

Keďže heslo sa pri šifrovaní používa opakovane, môže sa stať, že rovnaké slová (postupnosti znakov) sa zašifrujú rovnako. Napr. ak máme heslo dĺžky 5 znakov, reťazec „áno“ sa môže zašifrovať len 5 rôznymi spôsobmi. Ak by sme v šifrovanom texte našli opakujúce sa postupnosti znakov, pravdepodobne by zodpovedali rovnakému reťazcu z pôvodného textu. Z tohto vieme odhadnúť dĺžku hesla a postupovať tak, ako v predchádzajúcom odseku.

Otázka pre žiaka

Ak by sme mali počítač, ktorý by vedel otestovať 1 000 000 000 hesiel za sekundu, ako dlho by nám trvalo prelomenie šifry, ak by sme použili heslo dĺžky 8 znakov?

Otázka pre žiaka

Ak by sme mali počítač, ktorý by vedel otestovať 1 000 000 000 hesiel za sekundu, ako dlho by nám trvalo prelomenie šifry, ak by sme použili slovníkový útok? Máme istotu, že by sme šifru v tomto prípade prelomili?

Vernamova šifra

Pozrime sa na slabé miesta Vigenèrovej šifry a pokúsme sa ich odstrániť:

- možnosť odhaliť dĺžku hesla,
- možnosť použiť frekvenčnú analýzu na vybrané množiny znakov,
- ak bolo použité slovníkové heslo, môžeme použiť slovníkový útok,
- ak sa rôzne správy šifrovali tým istým heslom, dáva to útočníkovi výhodu, lebo čím dlhší šifrovaný text má k dispozícii, tým lepšie sa dá využiť frekvenčná analýza.

Vytvorme šifrovací systém, kde:

- heslo bude náhodne zvolené $\Rightarrow$ slovníkový útok sa nedá použiť,
- dĺžka hesla bude rovná dĺžke samotnej správy $\Rightarrow$ použitie hesla pri šifrovaní sa nebude opakovať, takže rovnaké slová sa zašifrujú rôzne,
- heslo použijeme len raz, resp. raz na zašifrovanie a raz na dešifrovanie správy $\Rightarrow$ ak by útočník získal viacero šifrovaných správ nebude môcť použiť frekvenčnú analýzu,
- heslo po použití zničíme $\Rightarrow$ ak by sa aj niekto dostal k nášmu heslu, nebude môcť dešifrovať žiadnu z predchádzajúcich správ (po zašifrovaní heslo zničíme).

Túto šifru navrhol v roku 1917 Gilbert S. Vernam (60), po ktorom je pomenovaná.

Aby sme vyriešili problém odovzdávania si kľúčov medzi odosielateľom a prijímateľom (pre každú správu potrebujeme iné heslo), vygenerujme si heslá do zásoby. Vygenerujeme dostatočne dlhý reťazec znakov, ktorého časti použijeme ako heslo pre šifrovanie jednotlivých správ.

Pri šifrovaní správy použime len toľko znakov z tohto reťazca aká je dĺžka šifrovaného textu.

Znaky reťazca, ktoré sme už raz ako heslo použili, zničíme.

Napr.:

vygenerovaný reťazec pre heslo:	5 a A s 8 Z + - X b f t g f / g 8 4 j k
1. text na zašifrovanie:	s p r á v a 1
heslo:	5 a A s 8 Z + - X b f t g f / g 8 4 j k
zašifrovaná správa	¨ Ñ ³ Ŕ ® » K ^
2. text na zašifrovanie:	s p r á v a 2
heslo:	X b f t g f / g 8 4 j k
zašifrovaná správa	Ë Ò Ø ŕ Ý Ç O «
3. text na zašifrovanie:	s p r á v a 3
heslo:	8 4 j k
zašifrovaná správa	správa sa nedá zašifrovať, nedostatočne dlhé heslo, je potrebné vygenerovať nové heslo

Ak dodržíme všetky podmienky tak, ako sme ich uviedli vyššie, tak táto šifra je dokázateľne neprelomiteľná.

Úloha 5

Vytvorte modul vernam, ktorý poskytne všetky potrebné nástroje na praktické použitie Vernamovej šifry. Analyzujte, akú funkcionalitu by mal modul poskytovať a definujte zodpovedajúce funkcie.

Otázka pre žiaka

Môžeme pre generovanie reťazca pre heslo použiť modul random? Ak nie, prečo?

Výkladový text

Modul `secrets` sa používa pre generovanie kryptograficky silných náhodných čísiel, ktoré sa dajú použiť pre tvorbu hesiel alebo kľúčov. Modul `random` bol vytvorený pre použitie v simuláciách a v modelovaní. Pre potreby kryptografie sa modul `random` nehodí, lebo je možné opakovane generovať rovnaké náhodné postupnosti.

Pre generovanie silných pseudonáhodných čísiel môžeme použiť funkciu `secrets.randbelow()`. Modul ponúka aj ďalšie zaujímavé funkcie a ak nás problematika bezpečnosti a šifrovania zaujíma, stojí za to funkcie modulu preskúmať.

Modul `secrets` bol pridaný v Python-e 3.6 a viac informácií o ňom nájdeme na: https://docs.python.org/3/library/secrets.html [28. 5. 2019].

Čo sme sa naučili:

- vysvetliť jednoduché šifrovacie algoritmy a implementovať ich v jazyku Python,
- analyzovať slabé miesta vybraných algoritmov a navrhnúť postupy ako šifry prelomiť,
- navrhnúť nástroje na prelomenie vybraných šifier.

Ďalšie úlohy na precvičenie a zamyslenie

1. Pri šifrovaní Vernamovou šifrou nahradzujeme jeden znak pôvodnej správy jedným znakom šifrovanej správy. Takmer vždy je však súbor so šifrovanou správou väčší ako súbor s pôvodnou správou. Prečo je tomu tak?

2. Predstavme si, že máme k dispozícii neobmedzený výpočtový výkon. Ak použijeme útok hrubou silou a vyskúšame všetky možné kľúče, budeme schopní správy zašifrované Vernamovou šifrou dešifrovať bez znalosti použitého kľúča?

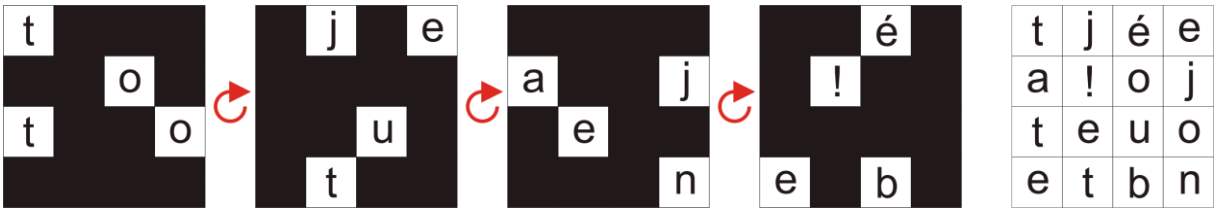
3. Akú nevýhodu má Vernamova šifra?

4. Vytvorte generátor silných hesiel. Za silné heslo budeme považovať reťazec, ktorý:
   - má aspoň 15 znakov,
   - obsahuje veľké aj malé písmená,
   - obsahuje číslice,
   - obsahuje špeciálne symboly, napr.: +-*/_<>?!#$%^&

   Generovanie hesla by malo byť neopakovateľné.

   Pomôcka: preskúmajte modul `secrets` a použite vhodné funkcie.

5. Cardanova šifra využíva mriežku (=kľúč), v ktorej sú niektoré okienka vystrihnuté. Cez tieto okienka je možné na podložený papier písať znaky tajnej správy. Ak mriežku pootočíme o 90°, vyrezané okienka sa dostanú na novú pozíciu, a môžeme v písaní správy pokračovať ďalej. Celkovo môžeme použiť 4 rôzne natočenia mriežky. Výsledkom je zhluk písmen, ktoré na prvý pohľad nedávajú zmysel. Do okienok, ktoré nevyužijeme, môžeme vpísať náhodné znaky.



   - Navrhnite vhodný spôsob (vhodnú dátovú štruktúru) ako mriežku s vyrezanými okienkami reprezentovať v počítači.
   - Navrhnite a implementujte postup ako vygenerovať, ktoré okienka na mriežke vyrezať.
   - Navrhnite ako zadanú správu (jej znaky) zapísať do štvorcovej matice tak, aby správa bola čitateľná po priložení mriežky.
   - Navrhnite ako vygenerovanú mriežku vytlačiť tak, aby sa dalo šifrovať pomocou ceruzky a papiera.

12. Šifrovačka

Kľúčové slová

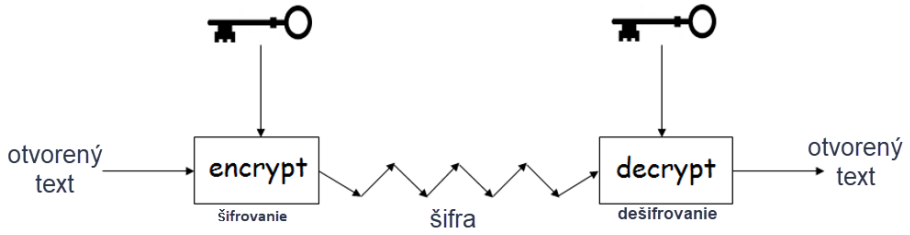
šifra, šifrovací algoritmus, kľúč, substitučná šifra, transpozičná šifra

Čo sa naučíme a čo si precvičíme

- zopakujeme si pojem šifrovanie,
- vysvetlíme rozdiel medzi šifrovaním a kódovaním,
- vysvetlíme rozdiely v spôsoboch šifrovania,
- ukážeme si substitučné a transpozičné šifry.

Problémová situácia

Na úvod si trochu zopakujme, čo už vieme. Informácie okolo nás uchovávame rôznymi formami – obraz, zvuk, text. Často je však výhodné uchovávať informácie v upravenej podobe ako napríklad piktogramy, morzeovka, Braillovo písmo, čiarový kód, QR kód, či kódovacie tabuľky (ASCII, UTF-8). Takéto všadeprítomné formy ukladania dát nazývame **kódovanie.** V tomto prípade ide len o transformáciu informácie z jednej formy na druhú pomocou postupu, ktorý je väčšinou verejne známy. Účelom kódovania nie je utajenie informácie, ale len iná forma zápisu vybraná tak, aby sa informácia dala čo najúspornejšie uchovať alebo šíriť. Ak chceme správu skryť, musíme do komunikácie zaviesť nejaký tajný aspekt. Vtedy hovoríme, že informácie šifrujeme. **Šifrovanie** (Encryption) je transformácia informácie z jednej formy na druhú za účelom skrytia skutočného obsahu informácie pred očami nesprávnych osôb (61). Šifrovanie ako metóda skrývania správ je veľmi stará a dnes veľmi dôležitá najmä pri komunikácii cez internet. O metódach šifrovania a návrhoch šifrovacích systémov pojednáva **kryptografia**. Lúštením šifier, štúdiom možností prelomenia šifry a odhaľovaním slabín v šifrovacích systémoch sa venuje **kryptoanalýza** (62). Základný princíp šifrovania a dešifrovania je na obrázku.



Obrázok 42 Základný princíp šifrovania a dešifrovania

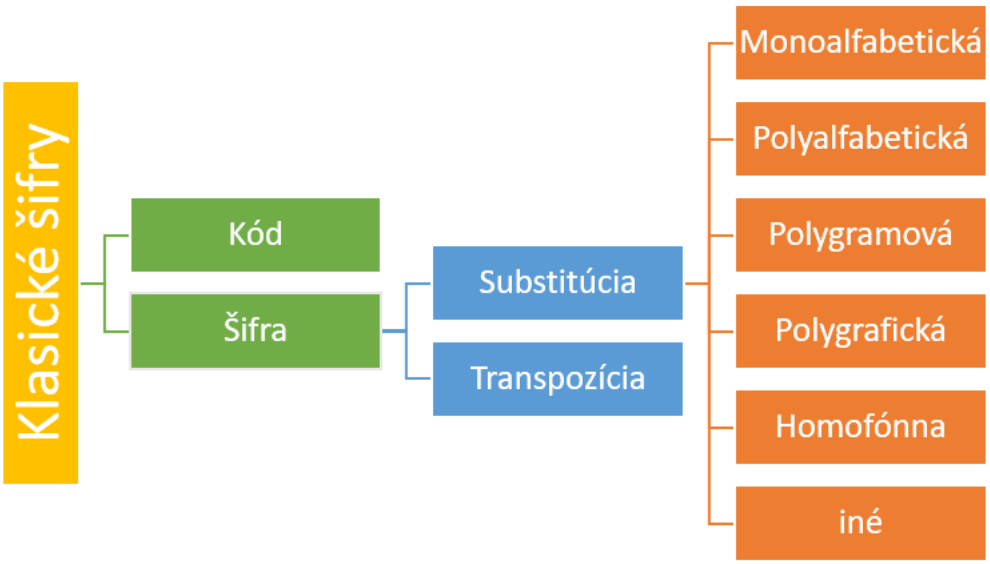
Šifrovacia funkcia (kryptografický algoritmus, šifrovač) je postup (matematická funkcia) pomocou ktorej sa transformuje otvorený text na šifrovaný a opačne. **Šifrovací kľúč** je utajovaný prvok bezpečnej komunikácie, ktorý vstupuje do šifrovanej funkcie (Obrázok 42).

Otázka pre žiaka

Kerckhoffov princíp (62) hovorí, že šifrovací algoritmus má byť verejne známy. Neznámy má byť jedine kľúč. Prečo?

Ak majú komunikovať dva vzdialené subjekty, ktoré zdieľajú rovnaký **tajný kľúč**, je potrebné ho utajiť. Tento kľúč je totiž ako kľúč od trezora - každý, kto má kľúč, ho vie otvoriť. Preto je najdôležitejšia časť komunikácie práve výmena tajného kľúča, ktorá musí prebehnúť v utajení. V minulosti banky vynaložili nemalé prostriedky na zabezpečené zasielanie kľúčov do svojich pobočiek. Obe strany totiž musia mať kópiu toho istého kľúča, ktorý navzájom zdieľajú. Tento typ šifrovanej komunikácie, kde všetci účastníci používajú rovnaký, zdieľaný kľúč, sa nazýva **symetrické šifrovanie**.

Práca kryptografa, či kryptoanalytika, sa v dejinách šifrovania výrazne menila, pričom najväčšiu zmenu priniesli počítače. Keď sa na šifrovanie a dešifrovanie správ nepoužívali počítače, môžeme hovoriť o ére **klasických (konvenčných) šifier** (pozri obrázok nižšie). Počítače preniesli šifrovanie do modernej doby. Klasické šifry môžeme rozdeliť na **substitučnú** (zachováva pôvodnú pozíciu znaku pri zmene jeho identity, už je nám známa Cézarova a Vigénerova šifra) a **transpozičnú**, ktorá zachováva pôvodnú identitu znaku, nastane len zmena jeho pozície. My si priblížime šifrovanie cez symetrické konvenčné šifry (63).



Obrázok 43 Rozdelenie klasických šifier (63)

Substitučné (transkripčné) šifry

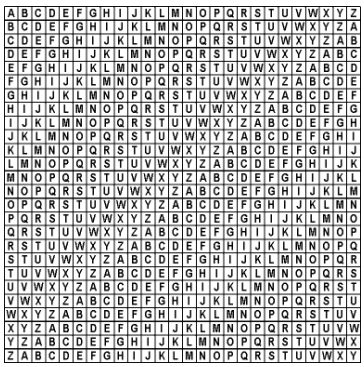
Šifrovanie, ktorého základ je zachovanie pôvodnej pozície znaku pri zmene jeho identity, sme si už vyskúšali v kapitole Šifry a tajné správy. Doplňme už len posledné informácie a ukážme si niektoré zaujímavé šifry, ktoré boli reálne navrhnuté a využívané v klasickej kryptografickej ére. Podľa toho, aký typ tajného kľúča sa využíval rozdeľujeme substitučné šifry na monoalfabetické a polyalfabetické (63).

Monoalfabetická šifra. Pri tomto spôsobe šifrovania sa celý text šifruje jednou šifrovou abecedou. Spomeňme si na Cézarovu šifru. Tu bol tajným kľúčom usporiadanie písmen v abecede (pozri obrázok nižšie).

Otvorený znak	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Šifrovaný znak	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

Obrázok 44 Tajným kľúčom v Cézarovej šifre bola posunutá abeceda

My už vieme, že monoalfabetická substitučná šifra sa dala ľahko prelomiť po objavení významu frekvenčnej analýzy pri dešifrovaní. Pri tomto type šifrovania sa znaky síce zamenia, ale početnosť výskytu šifrovaného znaku ostáva rovnaká. A tak sa stále dajú ľahko identifikovať znaky, ktoré sa v texte, v danom jazyku, vyskytujú najčastejšie a pod. Preto sa začali pre niektoré znaky používať viaceré šifrovacie symboly. Čím bola početnosť výskytu znaku väčšia, tým viac rozdielnych znakov bolo písmenu priradených. Tento typ šifry sa nazýva **homofónna** (63).



Spomeňme si na Vigenèrovu šifru, kde sa na šifrovanie používalo 26 rôznych abecied, každá posunutá o jedno miesto vpravo. Tajným cyklickým kľúčom bolo slovo, ktoré určovalo, ktoré konkrétne abecedy sa majú v danom momente použiť (dĺžka slova predstavovala periódu opakovania).

Keďže sa text šifruje viacerými abecedami podľa určitého kľúča išlo o **polyalfabetickú šifru.**

Ukážme si ešte zástupcu šifier, ktorá skupinu písmen z textu nahradí inou skupinou písmen šifrového textu s rovnakým počtom písmen. Patrí do skupiny tzv. **polygramových** (bigramová, trigramová, polygramová...) **šifier**.

Úloha 1

Naprogramujte **Polybiovu šifru**. Základom bola tabuľka 5x5 políčok, do ktorej vpísal 25 písmen abecedy (pozri obrázok nižšie). Keďže abeceda, ktorú používame, má 26 písmen, môžeme jedno z písmen vynechať alebo dať dve do jedného políčka. Zvyčajne sa tak robí s písmenami Q a W, ktoré sa u nás veľmi často nevyskytujú a z kontextu je vždy jasné, o ktoré písmeno ide (v angličtine sa napríklad zlučujú písmená I a J). Týmto usporiadaním vieme charakterizovať každé písmeno usporiadanou dvojicou riadok-stĺpec (64).

	1	2	3	4	5
1	A	B	C	D	E
2	F	G	H	I	J
3	K	L	M	N	O
4	P	Q	R	S	T
5	U	V	X	Y	Z

	1	2	3	4	5
1	R	G	X	O	D
2	V	A	B	C	N
3	Z	F	P	H	U
4	L	M	Q	S	T
5	J	K	I	Y	E

Obrázok 45 Tajným kľúčom v Polybiovej šifre bola abeceda usporiadaná do štvorca

Poznámka na okraj

Grécky historik Polybios (210 – 120 p. n. l.) vymyslel spôsob, ako zasielať správy na veľké vzdialenosti pomocou svetelnej signalizácie, preto túto šifru môžete nájsť v literatúre aj pod menom fakľový ďalekopis. Sám autor odporúčal, aby sa správa podávala ďalej tak, že počet faklí v ľavej ruke určuje číslo riadka a počet faklí v pravej ruke číslo stĺpca daného písmena.

Ako by sme zašifrovali text s kľúčom uvedeným na obrázku 4? Zoberme napríklad otvorený text `OKAMZITE USTUP`. Každý znak nahradíme jeho pozíciou v tabuľke. Zašifrovaný text bude vyzerať `35 31 11 33 55 24 45 54 51 44 45 51 41`.

Šifrovací kľúč si nemusíme uchovávať vo forme tabuľky. Abecedu môžeme spojiť do súvislého reťazca a polohu písmena dopočítame celočíselným delením a zvyškom po delení. Napríklad písmeno `k` je v šifrovacom kľúči `'abcdefghijklmnoprstuvwxyz'` na pozícii 11. Z toho vieme, že v tabuľke by mu prislúchala pozícia `11//5` a `11%5`, čiže `21`.

```
abc = 'abcdefghijklmnoprstuvwxyz'

def polybius_crypt(text, kluc = abc):
    text = text.lower()                                          [1]
    sifra = ""

    for pismeno in text:                                         [2]
       riadok = str((kluc.index(pismeno) // 5) + 1)              [3]
       stlpec = str((kluc.index(pismeno) % 5) + 1)               [4]
       sifra += riadok + stlpec                                  [5]
    return sifra

print(polybius_crypt('OKAMZITEUSTUP'))
```

[1] Šifrovací kľúč obsahuje len malé písmená. V riešení neuvažujeme nad situáciou, že do šifrovania pošleme neplatné znaky.
[2] Prejdeme všetky znaky otvoreného textu.
[3] Dopočítame pozíciu riadka vo virtuálnej tabuľke.
[4] Dopočítame pozíciu stĺpca vo virtuálnej tabuľke.
[5] Spojíme oba indexy a pripojíme ich do šifrovanej správy.

Jednou z možností, ako zvýšiť kvalitu šifry, je zvoliť si náhodné usporiadanie písmen v tajnom kľúči (Obrázok 45).

Úloha 2

Upravte algoritmus Polybiovej šifry tak, aby sa dal šifrovací kľúč tvoriť podľa kľúčového slova, ktoré napíšeme do prvých riadkov štvorca, pričom použité písmená už neopakujeme. Ostatné písmená sa potom doplnia tak, ako idú v abecede, pričom písmená, ktoré už sú v štvorci napísané, vynechávame. Ako vzorové heslo použijeme SPIONKA.

	1	2	3	4	5
1	S	P	I	O	N
2	K	A	B	C	D
3	E	F	G	H	J
4	L	M	Q	R	T
5	U	V	X	Y	Z

Zadefinujme najskôr funkciu, ktorá vytvorí nový šifrovací kľúč podľa kľúčového hesla.

```
abc = 'abcdefghijklmnoprstuvwxyz'
```

```
def novy_kluc(heslo):
    kluc = heslo.lower()                                    [6]

    for pismeno in abc:                                     [7]
        if pismeno not in kluc:                             [8]
            kluc += pismeno                                 [9]
    return kluc
```

[6] Parameter `kluc` bude zaciatok hesla. Napr. spionka.
[7] Prejdem všetky písmena neposunutej originálnej abecedy.
[8] Ak sa písmeno z abecedy ešte nenachádza vo vytváranom kľúči,
[9] tak rozšírim `kluc` o nové písmeno.

Novovytvorený kľúč môžeme poslať aj ako parameter šifrovacej funkcie.

```
def polybius_crypt(text, kluc=abc):
      ...
print(polybius_crypt('AHOJ', kluc=novy_kluc('spionka')))
```

Úloha 3

Čo ak zvolíme kľúčové slovo, ktoré obsahuje rovnaké písmená? Napríklad slovo `mamut` obsahuje dvakrát písmeno m. Šifrovacia tabuľka by bola bez úpravy tohto slova zhotovená nesprávne. Vynechajte z kľúča opakujúce sa písmená.

```
def odstran_duplicity(kluc):
    novykluc = ""
    for znak in kluc:
        if znak not in novykluc:
            novykluc += znak

def novy_kluc(heslo):
    kluc = heslo.lower()
    kluc = odstran_duplicity(kluc) ...                      [10]
```

[10] Pred použitím odstránim duplicity.

Poznámka

Teraz môžeme všetko spojiť do jednej funkcie. Spýtame sa používateľa, či chce použiť abecedu (kľúč) s heslom. Ak nechce, zoberie sa základná abeceda. Ak chce, vytvorí sa nová abeceda podľa zadaného kľúča.

```
def generuj_kluc():
    pridat_heslo = input("Chcete zadať tajne heslo? A/N ")
    if pridat_heslo.lower() == "a":
       return novy_kluc (input("Zadajte heslo: "))
    elif pridat_heslo.lower() == "n":
       return abc
```

Úloha 4

Teraz, keď už máme doladený algoritmus šifrovania, bolo by vhodné vedieť správu aj odtajniť. Vytvorte funkciu, ktorá dešifruje správu zašifrovanú Polybiovou šifrou.

```
def polybius_decrypt(text, kluc=abc):
    otvoreny_text = ""
    for i in range(0,len(text),2):                          [11]
        riadok = int(text[i])                               [12]
        stlpec = int(text[i+1])                             [13]
        index = (riadok-1)*5+ (stlpec-1)                    [14]
        otvoreny_text += kluc[index]                        [15]
    return otvoreny_text
```

[11] V šifrovanom texte je jedno písmeno zakódované dvoma znakmi, preto v cykle prechádzame šifru krokom 2.

[12] Na každej prvej pozícii z dvojice je kód pre riadok.

[13] Na každej druhej pozícii z dvojice je kód pre stĺpec.

[14] Kľúčom k dešifrovaniu je rovnako ako pri šifrovaní lineárny reťazec s abecedou. Pozíciu znaku, ktorý sa má pridať do dešifrovanej správy je preto nutné dopočítať inverznou operáciou ako pri šifrovaní, miesto delenia použijeme násobenie a pripočítame zvyšok.

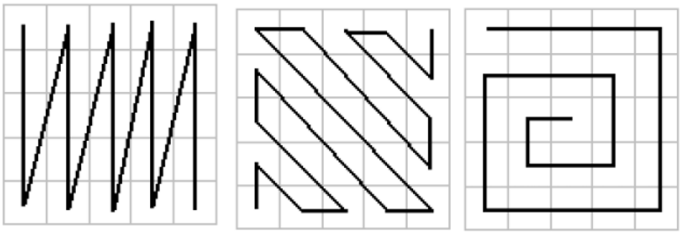
[15] Ak použijeme `kluc = "spionkabcdefghjlmrtuvwxyz"`, tak znak zašifrovaný kódom `41`, bude v abecede umiestnený na pozícii `4*5+1`. `kluc[21]` je písmeno `l`.

Dešifrovanie môžeme zavolať so šifrovaným textom a konkrétnou abecedou.

```
polybius_decrypt('22411323222322', kluc=novy_kluc('SPIONKA'))
```

Transpozičná šifra

Transpozícia je šifra, ktorá zachováva pôvodnú identitu znakov, nastane len zmena ich pozície (65). Pod jednoduchou transpozíciou si môžeme predstaviť napríklad vypísanie písmen pospiatky, ale prekladanie písmen nejakým iný slovom. Zaujímavou technikou bolo vypísať texty do kríža. Písmená sa rozdelia do skupín po štyroch alebo po ôsmych a tie sa potom zapíšu okolo kríža. Potom nasleduje prepis správy po riadkoch. Častou technikou bolo prepísanie znakov do tabuľky, ktorá sa potom rozvinie do riadka na základe nejakého pravidla (napr. špirála, pozri obrázok nižšie, (63)).



Obrázok 46 Ukážky možných pravidiel usporiadania znakov pre transpozičnú šifru

Úloha 5

Naprogramujte jednoduchú transpozičnú tabuľkovú šifru podľa stĺpca s heslom. Šifrovacia funkcia prepíše otvorený text do tabuľky, pričom počet stĺpcov bude daný na základe dĺžky zadaného šifrovacieho kľúča (pozri obrázok nižšie).

Ako budeme postupovať? Nech je kľúčom slovo `SPIONKA` a otvorený text `STRETNEME SA NA POLUDNIE`. Kľúč by sme mohli postupne zapísať nad otvorený text bez medzier.

S	P	I	O	N	K	A	S	P	I	O	N	K	A	S	P	I	O	N	K	A
S	T	R	E	T	N	E	M	E	S	A	N	A	P	O	L	U	D	N	I	E

Obrázok 47 K otvorenému textu cyklicky priradíme písmená tajného hesla

Heslo potom určuje poradie v akom sa majú znaky „miešať“. Najskôr by išli všetky znaky na pozíciách `S`, potom všetky na pozíciách `P`, ... a pod (pozri obrázok nižšie).

S	P	I	O	N	K	A	S	P	I	O	N	K	A	S	P	I	O	N	K	A
S	T	R	E	T	N	E	M	E	S	A	N	A	P	O	L	U	D	N	I	E
0	1	2	3	4	5	6	0	1	2	3	4	5	6	0	1	2	3	4	5	6

Obrázok 48 Tajné heslo určuje poradie miešania znakov otvoreného textu

Situácia by mohla byť jasnejšia, keď zmeníme usporiadanie textu do tabuľky pod seba podľa dĺžky textu (pozri obrázok nižšie).

S	P	I	O	N	K	A
S	T	R	E	T	N	E
M	E	S	A	N	A	P
O	L	U	D	N	I	E

Obrázok 49 Otvorený text môžeme podľa hesla prepásať aj do tabuľky

V skutočnosti je to to isté, ako keby sme znakom priradili poradové čísla, ktoré by sa opakovali s periódou podľa dĺžky kľúča (pozri obrázok nižšie).

S	P	I	O	N	K	A
S	T	R	E	T	N	E
0+0=0	1+0=1	2+0=2	3+0=3	4+0=4	5+0=5	6+0=6
M	E	S	A	N	A	P
0+7=7	1+7=8	2+7=9	3+7=10	4+7=11	5+7=12	6+7=13
O	L	U	D	N	I	E
0+14=14	1+14=15	2+14=16	3+14=17	4+14=18	5+14=19	6+14=20

Obrázok 50 Každému znaku otvoreného textu môžeme priradiť poradové číslo, periódu následne určuje dĺžka kľúča

Postupným spájaním znakov s rovnakým poradovým číslom vytvoríme hľadanú transpozíciu.

STRETNEMESANAPOLUDNIE

Pri realizácii využijeme zoznam, v ktorom sa budú postupne vytvárať čiastkové reťazce. Zavedieme si teda skokana, ktorý bude poskakovať po indexoch otvoreného textu, podľa presných pravidiel:

1) Postav sa na prvý ešte neskopírovaný znak reťazca.
2) Skopíruj znak do šifrovaného textu.
3) Keď sa dá posuň sa doprava o dĺžku kľúča a pokračuj bodom 2 inak sa vráť do bodu 1.

```
def column_transposition_crypt(text, kluc):
    dlzka_kluca = len(kluc)
    sifra = [''] * dlzka_kluca                              [16]

    for stlpec in range(dlzka_kluca):                       [17]

        skokan = stlpec                                     [18]
        while skokan < len(text):                           [19]
            sifra[stlpec] += text[skokan]                   [20]
            skokan += dlzka_kluca                           [21]

    return sifra                                            [22]

#.............................................................
sifra = "".join(column_transposition_crypt(text, kluc))
```

[16] Počet hodnôt zoznamu obsahujúceho čiastkové reťazce šifrovaného textu bude určovaný dĺžkou reťazca.
[17] Postupne prejdeme všetky stĺpce (všetky poradové čísla).
[18] Skokana ukazujúceho na aktuálne zapisovanú hodnotu nastavíme na začiatok stĺpca.
[19] Pokiaľ skokan nepresiahol dĺžku reťazca,
[20] tak do čiastkového reťazca príslušného stĺpca pridáme aktuálne čítanú hodnotu,
[21] a posunieme sa o celú dĺžku kľúča doprava.
[22] Čiastkové reťazce vrátime ako zoznam a mimo metódy ich spojíme.

Poznámka

Tesne pred spojením šifrovaného textu bude zoznam sifra vyzerať nasledovne:

```
['smo', 'tel', 'rsu', 'ead', 'tnn', 'nai', 'epe']
```

Po spojení dostaneme pre náš vzorový príklad šifru `smotelrsueadtnnnaiepe`.

Miesto cyklu `while` sme mohli použiť aj zápis s cyklom `for`.

```
for skokan in range(stlpec,len(text),dlzka_kluca):
                        ...
```

Úloha 6

Dešifrujte reťazec znakov vytvorený transpozičnou šifrou podľa stĺpca tabuľky s heslom.

Ako už vieme z kryptografickej teórie, tak proces dešifrovania bude vyzerať podobne, šifrovaciu funkciu sme nezmenili. Opäť budeme poskakovať po reťazci a vyberať potrebné prvky. Pri dešifrovaní však potrebujeme postupne zložiť pôvodné riadky. Koľko tých riadkov vlastne bolo? To sa dá zistiť vzťahom medzi dĺžkou zašifrovaného textu a dĺžkou kľúča.

Ak je šifra `smotelrsueadtnnnaiepe` (dĺžka 21 znakov) a kľúč `spionka` (dĺžka 7 znakov), tak v pôvodnej tabuľke boli tri riadky (21/7 = 3) a to znamená, že každý stĺpec obsahoval presne tri znaky. Pri dešifrovaní preto musíme rozdeliť text do trojíc, očíslovať príslušné znaky poradovými číslami a zopakovať postup ako pri procese šifrovania (Obrázok 51).

S	M	O	T	E	L	R	S	U	E	A	D	T	N	N	N	A	I	E	P	E
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3

Obrázok 51 Znaky šifrovaného textu vieme spätne „rozvinúť“ do tabuľky (obrázok 9) podľa pozície šifrovaného znaku

```
def column_transposition_decrypt(sifra, kluc):

    pocet = len(sifra) // len(kluc)                          [23]
    text = [''] * pocet

    for riadok in range(pocet):
            text += sifra[riadok::pocet] #každý pocet-tý znka šifry

    return text
#.............................................................
text = "".join(column_transposition_decrypt(sifra, kluc))
```

[23] Najdôležitejšou časťou je identifikovať počet znakov tvoriacich jeden blok zašifrovaného textu. Potom je už proces identický ako v bodoch [16]-[22].

Úloha 7

Skúste experimentovať so šifrovaním a dešifrovaním pričom použijete kľúče rôznej dĺžky. Diskutujte o funkčnosti riešenia.

Ak za heslo zvolíme text `SPION`, zašifrovanie textu prebehne úplne v poriadku. Jeho dešifrovanie však už nebude možné. Prečo?

Ak je dĺžka textu násobkom dĺžky hesla, všetok je v poriadku. Ak toto pravidlo nie je splnené, tak nastanú problémy pri dešifrovaní. Napr. po aplikovaní hesla `SPION` zostanú v poslednom riadku prázdne miesta. Čiastkové reťazce tak majú rôznu dĺžku `['snale', 'tenu', 'rmad', 'eepn', 'tsoi']` a ich spojením túto informáciu strácame (pozri obrázok nižšie).

S	P	I	O	N
S	T	R	E	T
N	E	M	E	S
A	N	A	P	O
L	U	D	N	I
E				

S	P	I	O	N
S	T	R	E	T
N	E	M	E	S
A	N	A	P	O
L	U	D	N	I
E	X	G	R	Q

Obrázok 52 Ak dĺžka otvoreného textu nie je násobkom dĺžky hesla, musíme do otvoreného textu doplniť náhodné znaky

Takáto situácia sa riešila pridaním ľubovoľných znakov, ktorých úlohou bolo len doplniť tabuľku. Preto vždy pred tým, ako text zašifrujeme zavoláme metódu `text = dopln(text, dlzka_kluca)`.

```
def dopln(text, dlzka):
    if len(text) % dlzka == 0:
        return text.lower()                                    [24]
    else:
        for i in range(dlzka - len(text) % dlzka):             [25]
            text += random.choice('abcdefghijklmnoprstuvwxyz') [26]
        return text.lower()
```

[24] V prípade, ak je dĺžka textu násobkom dĺžky hesla, tak sa nič nemení a vráti sa len text, kde všetky jeho znaku budú malé.
[25] V prípade, že by vznikli prázdne miesta v tabuľke, vypočítame koľko znakov treba pridať.
[26] Opakovane to textu pridáme náhodný znak abecedy.

Úloha 10

Naprogramujte Richelieuovu šifru, ktorej pravidlá sú popísané nižšie (64).

Poznámka na okraj

Armand Jean du Plessis - Kardinál Richelieu prvý minister kráľa Ludvíka XIII. V 14. storočí vo Francúzku bola používaná jednoduchá transpozičná šifra. Jej autorom nebol nikto iný než známy kardinál Richelieu - nechválene známi ako „Šedá eminencia.“ Richelieu bol prvým ministrom kráľa Ludvíka XIII. Za jeho éry sa Francúzsko pozdvihlo na úroveň poprednej európskej mocnosti. Kardinál Richelieu údajne založil šifrovacie oddelenie, ktoré sa zaoberalo šifrovaním posielaných informácii. Všetkým známe nároky o kráľovský trón, ta jné dohody s nepriateľmi a intrigy na dvore kráľa Ludvíka boli samozrejme taktiež vykonávané doručovaním tajných správ, súbojov mušketierov a členov kardinálovej gardy (66).

Richelieuova šifra používa vopred zvolený kľúč konštantnej dĺžky. Ide o transpozičnú šifru, kde sa využíva periodické heslo. Zašifrujme text `STRETNEME SA NA POLUDNIE`, pričom použijeme kľúč `SPIONKA`. Keďže ide o cyklický kľúč, budeme opäť kľúč postupne vpisovať nad otvorený text, alebo text budem postupne vpisovať pod heslo a zalamovať ho. Šifrovanú správu si teda napíšeme do riadkov tabuľky a nad tabuľku si napíšeme kľúčové slovo (pozri obrázok nižšie).

S	P	I	O	N	K	A
S	T	R	E	T	N	E
M	E	S	A	N	A	P
O	L	U	D	N	I	E

Obrázok 53 V prvej fáze šifry Richelieu zalomíme otvorený text podľa hesla

Prvá časť problému je rovnaká. Preto je možné použiť už existujúcu šifrovaciu funkciu, ktorá rozdelí text do stĺpcov podľa dĺžky hesla.

```
def richelieu_crypt(text, kluc):
    sifra = column_transposition_crypt(text, kluc)

#.....................................................
['smo', 'tel', 'rsu', 'ead', 'tnn', 'nai', 'epe']
```

Potom stĺpce zoradíme podľa poradia písmen v kľúčovom slove (pozri obrázok nižšie). Šifrový text sa vypíše po stĺpcoch (alebo riadkoch). Výsledný text prepíšeme po stĺpcoch `EPERSUNAITNNEADTELSMO`.

A	I	K	N	O	P	S
E	R	N	T	E	T	S
P	S	A	N	A	E	M
E	U	I	N	D	L	O

Obrázok 54 V druhej fáze šifry Richelieu zmeníme poradie stĺpcov, usporiadame ich podľa písmen v hesle

Ďalším krokom je teda zamiešanie stĺpcov podľa upraveného kľúča, ktorého znaky sú usporiadané abecedne vzostupne. Stačí, keď kľúč pretypujeme na zoznam a utriedime ho.

```
usporiadany_kluc = sorted(list(kluc))

#....................................
['a', 'i', 'k', 'n', 'o', 'p', 's']
```

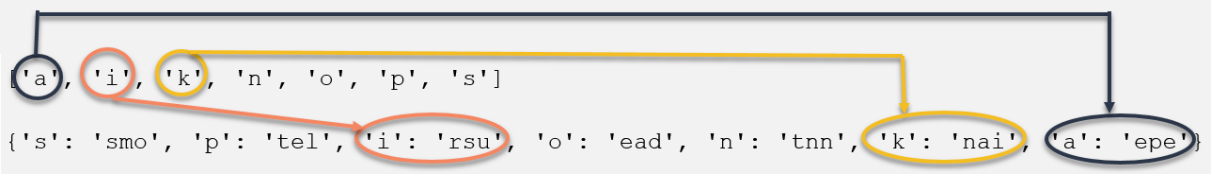
Aby sa nám dobre pracovalo so stĺpcami bolo by dobré spraviť prepojenie medzi hlavičkou, v ktorej je vpísané heslo a hodnotou samotných stĺpcov tabuľky. Použijeme zviazanie pomocou štruktúry `dict`.

```
tabulka = { kluc[i]:sifra[i]  for i in range(len(kluc))}

#.............................................................
{'s': 'smo', 'p': 'tel', 'i': 'rsu', 'o': 'ead', 'n': 'tnn', 'k': 'nai', 'a':
'epe'}
```

Posledným krokom je vyskladať reťazec (pozri obrázok nižšie). Zoberieme prvý prvok usporiadaného kľúča (`'a'`) a vyhľadáme v tabuľke k nemu prislúchajúci stĺpec (`'epe'`). Zoberieme druhý prvok …

```
sifra = ''
for index in usporiadany_kluc:
     sifra += tabulka[index]
```

```
return sifra

#.....................................................
'epersunaitnneadtelsmo'
```



Obrázok 55 Logické prepojenie zoznamu s utriedeným heslom a slovníka so šifrovanými stĺpcami

Úloha 11

Dešifrujte text zašifrovaný šifrou Richelieu.

Pri dešifrovaní je nutné najskôr vrátiť stĺpce do pôvodného tvaru a potom môžeme použiť už vytvorenú dešifrovaciu funkciu `column_transposition_decrypt(text, kluc)`. Najskôr zistíme koľko znakov tvorí jeden stĺpec.

```
pocet = len(sifra) // len(kluc)
```

Vytvoríme si rovnaký usporiadaný kľúč ako sme mali pri šifrovaní.

```
kluce = sorted(list(kluc))
```

A rozsekáme šifru na čiastkové reťazce podľa vypočítanej dĺžky podreťazca.

```
stlpce =  [sifra[i:i + pocet] for i in range(0, len(sifra), pocet)]
```

Teraz už môžeme urobiť previazanie medzi znakmi kľúča a stĺpcami. Vytvoríme si pôvodnú tabuľku použitú pri šifrovaní textu, opäť cez štruktúru slovník.

```
tabulka = {kluce[i]:stlpce[i] for i in range(len(kluc))}
```

Informáciu o tom, ako boli usporiadané stĺpce na začiatku nám hovorí samotný kľúč. Prejdeme preto všetky znaky pôvodného, neupraveného kľúča, a budeme v slovníku hľadať k nim prislúchajúce položky. Tie následne spojíme do dešifrovaného reťazca.

```
text = ''
for k in kluc:
    text += tabulka[k] #v slovníku tabuľka hľadáme reťazec s kľúčom k

return column_transposition_decrypt(text, kluc)
```

Ďalšie úlohy na precvičenie a zamyslenie

1. Ako sa zmení správanie šifry Richelieu, ak za heslo dáme reťazec obsahujúci rovnaké písmená? Navrhnite ako vyriešiť takýto problém s kľúčom.

2. Pôvodné šifry často pracovali s písmenami malej abecedy, preto sú aj naše riešenia zamerané len na tieto znaky. Aby sme sa neorientovali len na malú abecedu, implementujte rozšírenú verziu aj pre ostatné znaky.

3. Francis Bacon (1561-1626) bol anglický filozof a štátnik. Baconova šifra, patrí medzi substitučné šifry, kde každý znak z pôvodného textu je nahradený skupinou piatich znakov (pozri obrázok nižšie). Naprogramujte buď originálnu verziu alebo nahraďte A za 0 a B za 1 (67).

a:	AAAAA	h:	AABBB	o:	ABBBA	v:	BABAB
b:	AAAAB	i:	ABAAA	p:	ABBBB	w:	BABBA
c:	AAABA	j:	ABAAB	q:	BAAAA	x:	BABBB
d:	AAABB	k:	ABABA	r:	BAAAB	y:	BBAAA
e:	AABAA	l:	ABABB	s:	BAABA	z:	BBAAB
f:	AABAB	m:	ABBAA	t:	BAABB		
g:	AABBA	n:	ABBAB	u:	BABAA		

Obrázok 56 Pôvodná šifrovacia tabuľka Francisa Bacona

13. Steganografia

Kľúčové slová

tajná správa, steganografia, kľúč, kryptológia, skrývanie dát v iných dátach

Čo sa naučíme a čo si precvičíme

- vysvetlíme si rozdiel medzi kryptografiou a steganografiou,
- ako skrývať dáta v iných dátach,
- čo je to najmenej významný bit,
- ako konvertovať text na čiernobiely obrázok,
- čo je to podprahová informácia.

Problémová situácia

V súčasnom svete prebieha veľké množstvo našej komunikácie, vyjadrení a jeho zdieľaní so širokým okolím v digitálnej podobe. Na jednej strane je takáto forma komunikácie rýchla, lacná a dostupná širokým masám ľudí, čo je vítané. Na druhej strane je však často zneužívaná aj pre nelegálnu činnosť, kde komunikujú skupiny ľudí s úmyslom akéhokoľvek poškodenia. Ochrana informácií a bezpečnosť ich prenosu je celosvetovým problémom. Tejto téme sme sa už venovali v časti Šifry a tajné správy. Tam sme sa dáta snažili zmeniť tak, aby ich nepovolaná osoba nevedela prečítať. Hovorili sme tomu, že sme ich zašifrovali. Ak sme dátam chceli vrátiť ich pôvodnú formu, hovorili sme tomu, že sme ich dešifrovali. Stali sme sa kryptografmi. Kryptografia ako veda, ktorá skúma a navrhuje spôsoby utajenia správ je väčšinou založená na matematickom princípe. Do skrývania sa zapája často zložitá matematická funkcia a tajný kľúč. Ak zašifrovanú správu odchytí osoba, ktorej nemala byť určená, bez znalosti tajného kľúča by ju nemala vedieť prečítať (dešifrovať).

V tejto kapitole sa však pozrieme na iný spôsob utajovania správ, na steganografiu. Počiatky steganografie siahajú ku koreňom našej civilizácie. Počas jej vývoja sa používali rôzne techniky ukrývania správ v závislosti od stupňa vývoja civilizácie a jej možností.

Výkladový text

Steganografia je vednou disciplínou, ktorej úlohou je preniesť tajnú správu v pozadí neutajenej komunikácie. Je to veda o utajovaní komunikácie prostredníctvom ukrytia tajnej informácie v inej správe. Úspešný steganografický systém nedáva nepovolaným osobám vôbec vedieť, že je nejaká dodatočná (tajná) informácia prítomná. V skutočnosti ide o istú formu kamufláže. Steganografia sa od kryptografie líši tým, že kladie dôraz na zneprístupnenie komunikácie a nie na utajenie správy (68).

Steganografia a kryptografia sa teda navzájom dopĺňajú. Kryptografia urobí správu nečitateľnou, zatiaľ čo steganografia ju robí neviditeľnou.

V gréčtine slovo *stegos* znamená skrytý, *graphos* znamená zobrazovanie, písanie. Spolu je to teda ukrývanie písma alebo obrazov (69).

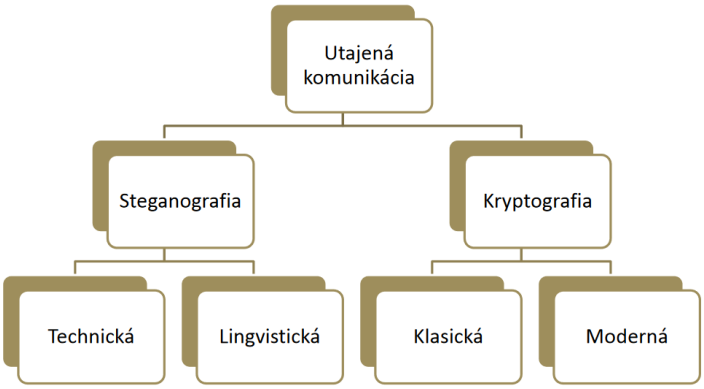
Otázky pre žiaka

V steganografii je správa skrytá tak, aby si pozorovateľ ani neuvedomil, že nejaká komunikácia vôbec prebieha. Cieľom steganografie je zakryť skutočnosť, že sa prenáša niečo tajné. Jej podstatou je snaha ukryť správu tam, kde ju nikto nebude hľadať.

Ako by ste skryli správu tak, aby nikto nevedel o tom, že sa práve „pozerá“ na nejakú komunikáciu?

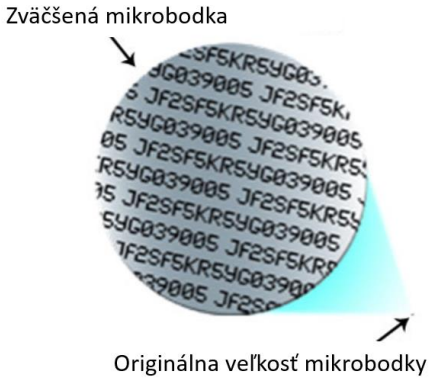
Čo majú jednotlivé spôsoby spoločné a v čom sa líšia?

Utajenú komunikáciu by sme teda mohli rozdeliť na dve základné oblasti, kryptografia (tej sme sa už venovali v iných kapitolách) a steganografia.



Obrázok 57 Rozdelenie utajenej komunikácie (70)

Lingvistická steganografia funguje napríklad tak, že cez média sa zverejní obyčajne vyzerajúca správa, v ktorej je však na základe vopred dohodnutých pravidiel (slová, či znaky textu) skrytý odkaz. Technická steganografia využíva rôzne technologické procesy fyzické ukrytie správy (70). Neviditeľný atrament (ocot, mlieko, rôzne ovocné šťavy), tetovanie na pokožku (pod vlasy), drevené doštičky zaliate vo vosku, písanie na škrupinku natvrdo uvareného vajíčka, či zmenšenie textu fotografickými technikami do tzv. mikrobodky sú techniky, ktoré sú známe skôr pred príchodom počítačov (71).



Obrázok 58 Mikrobodka - ukážka technickej steganografie (72)

Digitálna steganografia

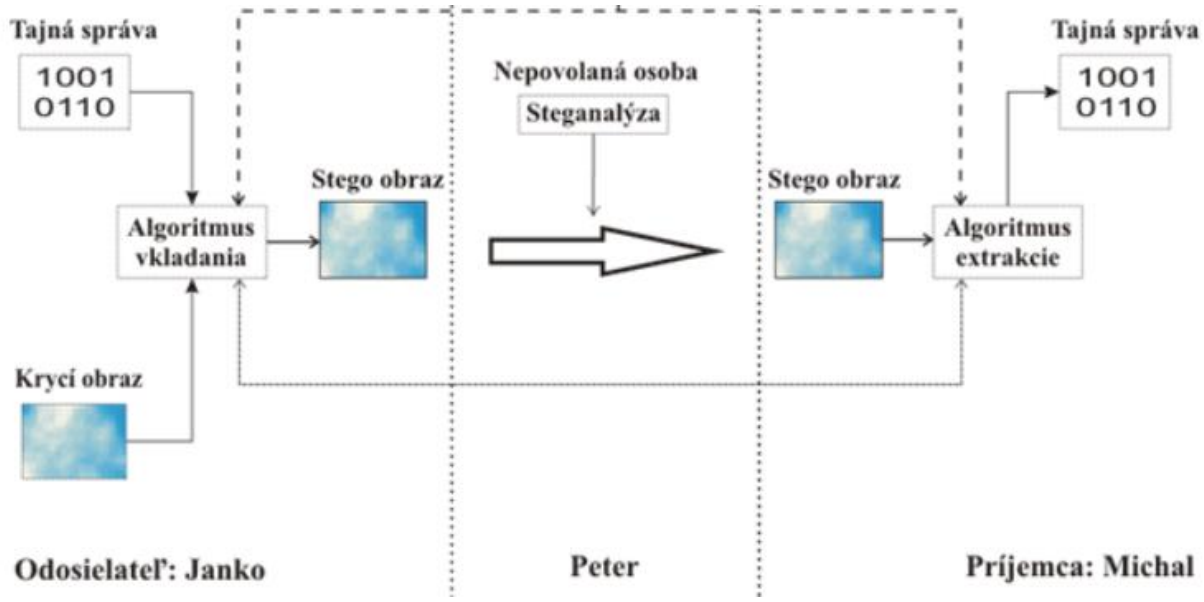
S príchodom digitalizácie dát sa používajú moderné metódy skrývania správ. **Digitálna steganografia** umožňuje pribaliť tajnú správu do textových súborov, obrázkov, či dokonca zvukov. Pri skrývaní tajných správ sa využíva vlastnosť, že dnešné dátové formáty môžu byť nosičmi istého šumu, a to bez toho, aby boli znehodnotené.

Poznámka na okraj

Dodnes sa používajú klasické techniky steganografie napríklad v bankovách, kde sú isté vlákna vidieť len v určitom druhu svetla. Digitálna steganografia sa zase používa na vkladanie ochranných prvkov do samotných digitálnych dát. Napríklad vloženie autorského práva (copyrightu) priamo do obrázka (68).

Zovšeobecnený model digitálnej steganografie:

- výber krycieho média (cover medium), nosič tajnej správy,
- výber vhodného algoritmu pre ukrytie dát do krycieho média,
- vloženie tajnej informácie do krycieho média, vzniká stegoobjekt (stego médium),
- extrahovanie tajnej správy zo stegoobjektu na strane príjemcu.



Obrázok 59 Model digitálnej steganografie (68)

Do digitálneho obsahu sa vkladajú bezpečnostné informácie nazývané ako autentizačné bitové vzorky. Vzorky sú pevnou súčasťou zdrojových dát a môžu byť pre koncového užívateľa obsahu nevnímateľné, ale je možné z nich zistiť prítomnosť správy, prípadne ich extrahovať a porovnať originalitu. Tieto autentizačné vzorky sa nazývajú **digitálne vodoznaky**. Vodoznak môže obsahovať informácie o príjemcovi, alebo autorovi pôvodných dát, autorské práva vo forme textových dát, alebo obrazu. Cieľom vodoznakov je chrániť médium proti akýmkoľvek modifikáciám s dôrazom na utajenie (73).

Výkladový text

Vedný odbor zaoberajúci sa metódami detekcie prítomnosti steganograficky ukrytých informácií sa nazýva **stegoanalýza**. Zaoberá sa odhalením tajnej správy v multimédiách a detekciou podprahovej komunikácie prebiehajúcej medzi dvoma stranami. Dnes existujú spôsoby ako tajnú informáciu dekódovať. Žiadna z metód skrývania správ tak nezaručuje stopercentné utajenie.

Úloha na rozohriatie

Otvorte si dokument `Digitálna steganografia.doc`. Na prvý pohľad vyzerá dokument úplne obyčajne. V texte je však ukrytá tajná správa, vedeli by ste ju nájsť?

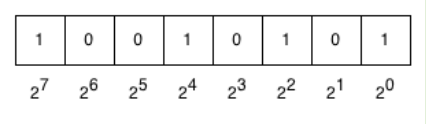
Úloha 1

Navrhnite, ako by sme mohli zakódovať text do obrázka. Uvažujme na úrovni bitovej reprezentácie každého pixela. Berme do úvahy, že spojením obrázka a textu, musia byť zmeny spôsobené spojením nepozorovateľné.

Výkladový text

Každý pixel obrázka nadobúda nejakú farbu, kódovanú pod číslom. Každý obrázok si tak môžeme predstaviť ako maticu (tabuľku čísel) čísel. Čiernobielym obrázkom stačí na zakódovanie len jedna takáto tabuľka čísel. Farebný obrázok ich má tri, každú pre jeden farebný kanál RGB (Red Green Blue).

Každý bod sa tak skladá zo samostatných farebných zložiek červená (R), zelená (G) a modrá(B), pričom každý z nich môže nadobúdať hodnoty od 0-255. Každá zložka obrazového bodu sa dá zapísať v 8 bitovom binárnom kóde. Ak máme napríklad farbu s číslom 149 vieme ju binárne zapísať ako:



$$1 \cdot 2^7+0 \cdot 2^6+0 \cdot 2^5+1 \cdot 2^4+0 \cdot 2^3+1 \cdot 2^2+0 \cdot 2^1+1 \cdot 2^0=149$$

Prevod binárneho číslia do desiatkového si môžeme otestovať priamo v Pythone:

```
>>> print(0b10010101)
149
>>> print(0b00010101)
21
>>> print(0b10010100)
148
```

Pri pohľade na testované čísla je jasné, že zmenou bitu úplne naľavo sa číslo výrazne zmenilo. Tento bit označujeme ako najviac významný bit - **most significant bit**. Zmenou bitu najviac

vpravo sa číslo zmenilo len o jednotku. Aj preto sa tento bit nazýva najmenej významný bit - **least significant bit** (74).

Úloha 2

Overte, ako sa zmenou najmenej významného bitu mení farba, tzn. Jeden z farebných kanálov sa zmení o jednotku. Napríklad RGB(150,50,0) sa zmení na RGB(151,50,0). Využite napr. https://www.colorspire.com/

V ďalšej časti využijeme fakt, že dve farby, ktoré sú v palete farieb v tesnej blízkosti, to znamená, že ich číselná hodnota sa líši o 1, sú odtiene tej istej farby s takým minimálnych rozdielom, že človek si neuvedomí ich zámenu.

Úloha 3

Z obrázka `stegoobraz.png` treba extrahovať zakódované informácie. Pri kódovaní bola využitá technika Least Significant Bit, pričom text bol zakódovaný do červenej vrstvy RGB kanála. Na prácu s obrázkami využite balíček `Pillow`.

Výkladový text

Modul `Pillow` treba doinštalovať dodatočne. Zoznam všetkým externých balíčkov je možné nájsť na stránke https://pypi.org/ (Python Package Index). Na uvedenej stránke vyhľadajte informácie o balíčku `Pillow`.

Inštalácia je veľmi jednoduchá. Stačí otvoriť príkazový riadok a napísať:

```
> pip install Pillow
```

V priečinku `Scripts` sa automaticky vyhľadá a spustí program **pip**, ktorý si všetky potrebné a aktuálne dáta vyhľadá na internete. Výsledok inštalácie môže vyzerať napríklad takto:

```
C:\Users\Ucitel>pip install Pillow
Collecting Pillow
  Downloading https://files.pythonhosted.org/packages/72/22/a96d7b
    100% |████████████████████████████████| 1.4MB 787kB/s
Installing collected packages: Pillow
Successfully installed Pillow-5.2.0
```

Importovaním knižnice Image z balíka PIL `from PIL import Image` získame množstvo funkcií na prácu s obrázkami.

Výkladový text:

Funkcie z balíka `PIL Image`, ktoré použijeme :

`open(cesta)` – súbor uvedený v parameter `cesta` je otvorený a pripravený na prácu, môžeme ho čítať ale aj vytvoriť nový.

`new(mód, rozmer, farba)` - vytvorí nový obrázok vo zvolenom móde, rozmeru a farbách (štandardne je nastavená čierna). My budeme používať mód „`RGB`".

`load()` - v pamäti alokuje priestor pre toľko pixelov, aký je rozmer obrázka. Pri otváraní obrázku sa metóda `load()` volá automaticky.

`save(meno, format=None)` – uloží obrázok pod vybraným menom vo zvolenom formáte. Pokiaľ nezadáme formát, tak sa automaticky nastaví podľa zvolenej prípony v mene súboru.

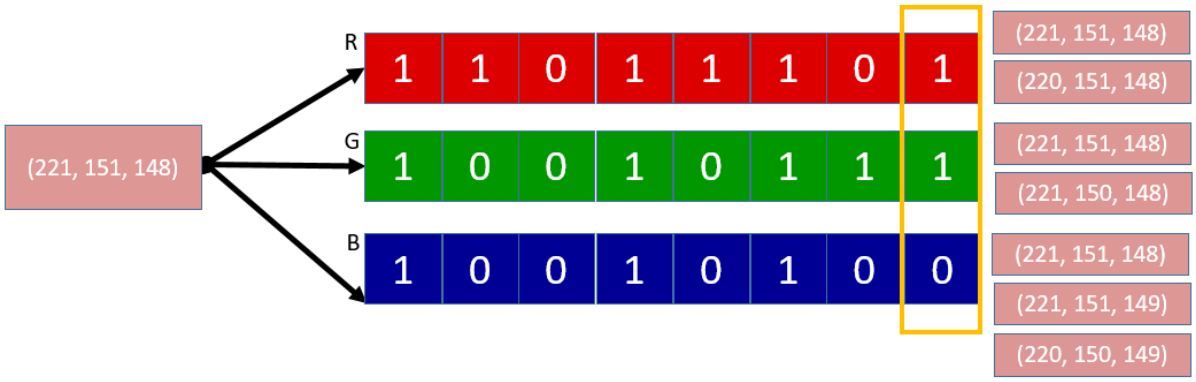
`split()` - funkcia vráti zoznam individuálnych vrstiev (kanálov) obrázka. Pri RGB sa vytvoria tri nové obrázky, kde každý z nich obsahuje vyextrahovaný jeden z farebných kanálov, `split()[0]` vyextrahuje červený kanál z pôvodného obrázka.

`size` - zoznam (tuple), v ktorom je definovaný rozmer obrázka, teda šírka a výška `(width, height)`.

`getpixel(xy)` – vráti konkrétny pixel na zadaných súradniciach.

Technika Least Significant Bit (LSB)

LSB je najstaršou a bežne používanou technikou na vkladanie informácií do obrázkov. Vodoznak vloží do najmenej významného bitu z pixelových dát. V každom pixely máme možnosť vkladať tajnú správu do ľubovoľného z troch farebných kanálov (RGB). Na obrázku nižšie je vidieť, že ak pre farbu RGB(221,151,148) zmeníme posledný bit na jednom z farebných kanálov, farba zostane na pohľad nezmenená (73).



Obrázok 60 Zmenou hodnoty najmenej významného bitu vo farebných kanáloch RBG

Ako vzniká skrytá tajná správa:

1. Zoberieme obrázok (tzv. cover, originál, nosič), do ktorého chceme skryť tajnú správu.
2. Vytvoríme nový čiernobiely obrázok, o veľkosti obrázka nosiča, do ktorého napíšeme, alebo skôr zakreslíme správu.
3. Následne pixel po pixely kombinujeme oba obrázky:
    a. Zoberieme `pixel[x,y]` z originálneho obrázka aj z tajnej správy zároveň.
    b. Ak je v tajnej správe pixel bielej farby, zamení sa najmenej významný bit (LSB) v originálnom obrázku za nulu.
    c. Ak je v tajnej správe pixel čiernej farby, zamení sa najmenej významný bit (LSB) v originálnom obrázku za jednotku.

Originál[2]	Stegoobraz
	

Obrázok 61 Porovnanie originálneho obrázka so stegoobrazom (75)

Otázky pre žiaka

V priečinku si otvorte obrázok `priatelia.png` a `priatelia+tajnytext.png`. Vidíte nejaký rozdiel? Skontrolujte aj veľkosť samotných obrázkov. V jednom z nich je skutočne zakódovaný text ako šum na pozadí, poďme ho z obrázka dekódovať.

Pri dekódovaní tajného textu budeme postupovať opačne, ako vo vyššie popísanom postupe. Prejdeme všetky body v stego obraze a z neho vyskladáme obraz s tajným textom. Otvoríme si súbor `Stego_uloha_3z.py`. Našou úlohou je doprogramovať funkciu `dekoduj`.

```
def dekoduj(self,zakodovany_cesta):                [1]
  zakodovany = Image.open(zakodovany_cesta)        [2]
  cerveny_kanal = zakodovany.split()[0]            [3]

  sirka = zakodovany.size[0]                       [4]
  vyska = zakodovany.size[1]
```

[1] Ako parameter funkcie prinesieme cestu k stegoobrázku, pokiaľ vo volaní funkcie uvedieme len názov, bude sa obrázok hľadať v tom istom priečinku, kde je uložený script.

[2] V pamäti alokujeme miesto a načítame doň obrázok na uvedenej ceste.

[3] Z načítaného obrázka vyextrahujeme len červený kanál, v ktorom je podprahovo ukrytá tajná správa.

[4] Zistíme rozmer stegoobrazu.

Pripravme si nový obrázok. Bude mať tri farebné vrstvy a veľkosť rovnakú ako náš zakódovaný stego obraz. V zozname zoznamov evidujeme množinu všetkých pixelov nového obrázka.

```
dekodovany = Image.new("RGB", zakodovany.size)
pixely = dekodovany.load()
```

V poslednej fáze prejdeme pixel po pixely stegoobraza, pričom nás zaujíma len najmenej významný bit každého z nich.

```
for i in range(sirka):
  for j in range(vyska):
    if bin(cerveny_kanal.getpixel((i, j)))[-1] == '0':
      pixely[i,j] = (255, 255, 255)
    else:
      pixely[i,j] = (0,0,0)
```

Výkladový text

V riešení sme na získanie informácie o LSB využili funkciu **bin**, ktorá číselnú informáciu o farbe získanú funkciou `getpixel()` prevedie do binárneho tvaru. Keďže výsledok prevodu je typu `str`, môžeme k bitu pristúpiť cez index. Index bitu najviac vpravo je `[-1]`.

```
bin(cerveny_kanal.getpixel((i, j)))[-1]
```

Ak je v stegoobraze na pozícii LSB nula zapíšeme do nového obrázka biely pixel `(255, 255, 255)`, inak tam bude čierny pixel `(0,0,0)`.

Funkcia `getpixel()` vyžaduje zadať len jeden parameter a preto musíme súradnice zadať ako dvojicu hodnôt pomocou `tuple`.

Dekódovaný tajný text uložíme do samostatného súboru.

```
dekodovany.save("dekodovany.png")
```

Otázka pre žiaka

Čo je zakódované v súbore `stegoobraz.png`?

Úloha 4

Do obrázka `priatelia.png` vložte technikou LSB ľubovoľný text. Využite podklad pripravený v súbore `Stego_uloha_4z.py`.

Princíp fungovania techniky LSB sme si vlastne už vysvetlili, poďme teda priamo k popisu algoritmu kódovania.

```
def koduj(self, text, nosic):                [5]

  nosic = Image.open(nosic)                  [6]

  cerveny_kanal = nosic.split()[0]           [7]
  zeleny_kanal = nosic.split()[1]
  modry_kanal = nosic.split()[2]

  sirka = nosic.size[0]                      [8]
  vyska = nosic.size[1]
```

[5] Ako parameter funkcie koduj prinesieme text, ktorý chceme utajiť a cestu k obrázku, do ktorého tajnú správu chceme vložiť, volajme ho nosič.

[6] Alokujme priestor v pamäti a načítajme doň obrázok na uvedenej ceste.
[7] V procese kódovania potrebujeme extrahovať všetky tri farebné kanály, neskôr ich totiž budeme všetky potrebovať pri opätovnom skladaní farby pixelu. Ktorýkoľvek kanál môžeme následne použiť ako nosič tajnej informácie.
[8] Potrebujeme vedieť šírku a výšku obrázka.

V druhom kroku potrebujeme vytvoriť obrázok s bielym podkladom, do ktorého prepíšeme tajnú správu. Skonvertujme text typu `str` na typ `image`. Výsledný obrázok musí mať rovnaký rozmer ako nosič, do ktorého ho budeme vkladať.

Výkladový text

Pre konverziu textu budeme potrebovať bitmapový font `ImageFont`, ktorým budeme kresliť na plochu obrázka.

`PIL.ImageFont.load(nazov)` – metóda načíta zvolený bitmapový font.

`PIL.ImageFont.load_default()` – metóda načíta základný bitmapový font.

Na samotné kreslenie potrebujeme modul `ImageDraw`. Môžeme si to predstaviť ako pečiatku, ktorej vzor sa bude postupne odtláčať na plátno obrázka.

```
def konvertuj_text(self,text, rozmer_obrazka):
  obrazok_text = Image.new("RGB", rozmer_obrazka)
  font = ImageFont.load_default().font                [9]
  peciatka = ImageDraw.Draw(obrazok_text)             [10]
  okraj = posun = 10                                  [11]
```

[9] Vytvoríme nový objekt typu `PIL.ImageFont`, načítame základný bitmapový font.
[10] Vytvoríme objekt typu `PIL.ImageDraw.Draw`, ktorý umožní kresliť (odtláčať, pečiatkovať) na obrázok zadaný ako parameter funkcie. V našom prípade dokážeme kresliť na novovytvorený `obrazok_text` typu RGB.
[11] Premenná `okraj` určuje vzdialenosť textu od ľavého okraja obrázka a `posun` určuje v ktorom riadku aktuálne sme (určuje posun od horného okraja obrázka).

Na samotné kreslenie potrebujeme modul `ImageDraw`. Môžeme si to predstaviť ako pečiatku, ktorej textový vzor sa bude postupne odtláčať na plátno obrázka na presne adresované miesto.

Výkladový text:

Dĺžka utajovaného textu môže presiahnuť šírku nosiča, čo by malo veľmi negatívne následky. Preto je vhodnejšie text po vopred určených znakoch zalomiť na nový riadok. Preto potrebujeme importovať posledný balíček `import textwrap`. Nás bude zaujímať metóda `wrap`, ktorá zo vstupného textu vytvorí zoznam riadkov, rozdelených podľa vopred definovanej dĺžky (štandardne je nastavená na 70 znakov).

```
>>> s = 'Toto je text, ktory sa pokusime zalomit podla vopred urceneho poctu znakov.'

>>> textwrap.wrap(s, width = 20)
```

```
['Toto je text, ktory', 'sa pokusime zalomit', 'podla vopred',
'urceneho poctu', 'znakov.']

>>> textwrap.wrap(s,width = 5)

['Toto', 'je', 'text,', 'ktory', 'sa po', 'kusim', 'e zal',
'omit', 'podla', 'vopre', 'd urc', 'eneho', 'poctu', 'znako',
'v.']
```

Môžeme si všimnúť, že pokiaľ je šírka zalomenia dostatočne veľká a vo vete sú slová oddelené čiarkou, tak funkcia `wrap` vráti zoznam reťazcov, kde žiaden z nich nie je dlhší ako parameter `width` ale zároveň sa veta rozdelí po celých slovách. Ak je však šírka zalomenia veľmi krátka metóda vráti zoznam reťazcov, kde každý reťazec má vopred presne definovanú dĺžku, bez ohľadu na dĺžku slov.

```
for riadok in textwrap.wrap(text, width=60):          [12]
    peciatka.text((okraj,posun), riadok, font=font)   [13]
    posun += 10                                       [14]
```

[12] Rozdelíme text na riadky s dĺžkou maximálne 60 znakov. Pre každý riadok zo zonamu vykonáme krok [13] a [14].

[13] Nakreslíme aktuálny riadok do obrázka. Súradnica ľavého horného rohu určuje dvojica `(okraj, posun)`.

[14] V ďalšom kroku posunieme ľavý horný roh o 10px nižšie.

Výkladový text

Trieda `ImageDraw.Draw` nám ponúka funkcie na kreslenie čiar, elíps, pravouholníkov, obrázkov ale aj textov.

Metóda `PIL.ImageDraw.Draw.text(xy, text, fill, font)` „kreslí text“, kde `xy` predstavuje súradnicu ľavého horného rohu textu v určenej farbe a zvolenom rastrovom fonte.

Funkcia `konvertuj_text` vráti obrázok s vloženým textom **`return`** `obrazok_text`.

Vráťme sa do funkcie **`koduj`**, kde vytvoríme nový obrázok s tajnou správou zavolaním funkcie `konvertuj_text`.

```
obrazok_text = self.konvertuj_text(text, nosic.size)
bw_obrazok_text = obrazok_text.convert('1')
```

Výkladový text

V metóde `konvertuj_text` sme vytvárali obrázok v RGB móde. Aby sme s pixelmi mohli narábať binárne (biela = 0, čierna = 1) treba ho ešte prekonvertovať. Funkcia `convert(mode)` slúži na konverziu do rôznych módov, napríklad `L` (8bit pixel, odtiene šedej), `RGB` (24 bit pixel, true color), `CMYK` (32 bit pixel) alebo `1` (1 bit pixel, čiernobiely). Pre nás je najzaujímavejší mód 1, teda čiernobiely obrázok (black and white, BW).

V poslednom kroku potrebujeme technikou LSB spojiť oba obrázky do jedného. Postupujeme podobne ako v úlohe č. 1. Vytvoríme nový obrázok `kodovany`, pre ktorý budeme pixel po pixely vypočítavať výslednú farbu.

```
kodovany = Image.new("RGB", (sirka, vyska))
pixely = kodovany.load()

for i in range(sirka):
  for j in range(vyska):
   cerveny_kanal_pix = bin(cerveny_kanal.getpixel((i,j)))          [15]
   bw_obrazok_text_pix = bin(bw_obrazok_text.getpixel((i,j)))      [16]

   if bw_obrazok_text_pix[-1] == '1':                              [17]
    cerveny_kanal_pix = cerveny_kanal_pix[:-1] + '1'               [18]
   else:
    cerveny_kanal_pix = cerveny_kanal_pix[:-1] + '0'

    pixely[i, j] = (int(cerveny_kanal_pix, 2),                     [19]
            zeleny_kanal.getpixel((i,j)),
            modry_kanal.getpixel((i,j)))

kodovany.save("nosic+text.png")                                    [20]
```

[15] Pre extrahovaný červený kanál urobíme prevod pixelu na pozícii `xy` do binárneho zápisu.
[16] Pre obrázok s tajnou správu urobíme prevod pixelu na pozícii `xy` do binárneho zápisu.
[17] Ak na poslednej pozícii binárneho zápisu je jednotka (našli sme čierny pixel), tak ...
[18] tak na najmenej významný bit v červenom kanále uložíme jednotku, inak nulu.
[19] Výsledná farba pixelu v stegoobraze sa skladá z troch zložiek (R,G,B), pričom. Červený kanál sme upravili, preto zoberieme upravenú hodnotu. Zelený a modrý kanál ostali bezo zmeny a preto použijeme hodnoty, ktoré sme získali v bode [7].

Ďalšie úlohy na precvičenie a zamyslenie

1. Ako by sa zmenila kvalita obrázka, keby sme do každého z RGB kanálov chceli zakódovať tri rozdielne textové správy? Upravte program tak, aby funkcie `koduj` a `dekoduj` tak, aby sme mali tri textové parametre predstavujúce tri rôzne tajné správy.
2. Čo sa stane, ak obrázok, v ktorom je skrytá tajná správa zväčšíme (zmenšíme), orežeme alebo ho uložíme v inom formáte?
3. Ako sa zmení obrázok, ak by sme na kódovanie nepoužil najmenej významný bit, ale až dva najmenej významné bity?
4. V úlohe sme pracovali s čiernobielou tajnou správnou. Spoločne navrhnite spôsob, ako by ste do farebného obrázka skryli iný farebný obrázok.

14. Rekurzia

Kľúčové slová

rekurzia, zásobník, triviálna vetva, zásobníková pamäť, návratová adresa

- vysvetlíme si pojem rekurzia, ako vytvárať rekurzívne riešenia úloh,
- naučíme sa ako funguje zásobníková pamäť,
- priblížime problematiku rekurzívneho volania a vykonávania podprogramov na úrovni triviálnych matematických príkladov,
- naučíme sa ako rozložiť problém na menšie podproblémy s nižšou zložitosťou,
- vysvetlíme si stratégiu riešenia problémov s názvom rozdeľuj a panuj,
- vysvetlíme si pojem fraktál a nakreslíme jednoduché fraktálové útvary.

Aktivita na úvod

Zoberte dve zrkadlá a nejaký objekt (váza, kniha, desiata ..). Nastavme zrkadlá tak, aby boli kolmo oproti sebe. Objekt umiestnime medzi zrkadlá. Z rôznych uhlov pozorujte odraz v zrkadlách.

Otázky pre žiakov

Opíšte, čo sa deje s odrazom v zrkadlách? Vedeli by ste to vyjadriť aj nejakým formálnym zápisom?

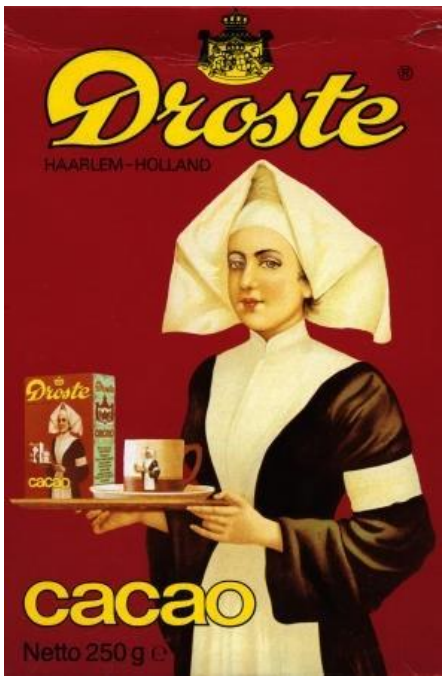
Čo sa stalo? V zrkadle uvidíme objekt a za objektom zrkadlo, v ktorom je objekt a a za ním zrkadlo ... a takto to pokračuje až do momentu, kedy nie sme viac schopní rozpoznať detaily vo vnorenom obraze.

Tento jav voláme rekurzia. **Rekurzia** je teda spôsob definovania objektu pomocou seba samého.

Obrazová rekurzia - Ide o efekt, kde sa obraz rekurzívne objavuje sám v sebe, a to na mieste, kde by sa podobný obraz realisticky najviac očakával (76). Tento efekt sa označuje aj ako „droste effect“, podľa výrobcu kakaa, ktorý túto techniku použil prvýkrát v roku 1904 (pozri obrázok nižšie). Najjednoduchšie získame rekurzívny objekt postavením dvoch zrkadiel oproti sebe.

Úloha 1

Skúste na internete nájsť najzaujímavejšie efekty založené na obrazovej rekurzii. Odhadnite, ktoré zábery sú len efektom vytvoreným v grafickom editore, a ktoré je možné reálne dosiahnuť priamo, či už zrkadlami, fotoaparátom, či kamerou.



Obrázok 62 Ukážky obrazovej rekurzie dosiahnuté reálnou situáciou a umelo vytvorené v grafickom editore (77)

Poznámka na okraj

Technika nekonečného odrazu sa využíva aj pri vytvorení efektu nekonečnej hĺbky pri pohľade na obyčajné zrkadlo, tzv. *nekonečné zrkadlo* (infinity mirror). Napriek tomu, že je zrkadlo umiestnené v úzkom ráme, pri pohľade doň vzniká pocit, že za zrkadlom je nekonečný priestor. Tento efekt vzniká nekonečným odrazom LED svetielok v dvoch proti sebe umiestnených zrkadlách. Rozdiel oproti predchádzajúcemu pokusu je ten, že jedno zo zrkadiel je polopriepustné – z jednej strany sa správa ako zrkadlo, z druhej strany ako priehľadné sklo.


(78)

Slovná rekurzia – ide o formu rozprávania, kde sa nedopovedaný príbeh opäť vráti na začiatok (76).

- Pri ohni sedelo 40 zbojníkov a zbojnícky kapitán začal rozprávať:
  - Pri ohni sedelo 40 zbojníkov a zbojnícky kapitán začal rozprávať:
    - Pri ohni sedelo 40 zbojníkov a zbojnícky kapitán začal rozprávať:
      - Pri ohni sedelo 40 zbojníkov a zbojnícky kapitán začal rozprávať:

Úloha 2

Skúste do vyhľadávača Google zadať kľúčové slovo rekurzia (recursion). Spoločnosť Google vtipne myslela na túto situáciu.

Typickou vlastnosťou rekurzií zo života je ich nekonečnosť (obrazová rekurzia pre nás skončí vtedy, keď nasledujúci obraz pre jeho veľkosť nedokážeme rozpoznať, slovná vtedy, keď jej rozprávačovi dôjdu sily a pod.).

Rekurzia v matematike – najčastejšie sa rekurzia používa na vyjadrenie nasledujúcej hodnoty pomocou predchádzajúcej. Napríklad na definíciu hodnoty člena číselnej postupnosti prostredníctvom predchádzajúceho člena. Aby bola definícia korektná a dala sa podľa nej určiť hodnota ľubovoľného člena postupnosti, musí sa uviesť aspoň jeden prípad, ktorý nie je definovaný rekurzívne a dá sa preň určiť hodnota bez určenia hodnôt iných členov (76). *Aritmetická postupnosť* by mohla mať nasledujúci rekurzívny zápis (v matematike označovaný ako rekurentný vzorec).

$$a_n \begin{cases} a_{n-1} + d & n>1 \\ 3 & n=1 \end{cases}$$

Nech máme `d = 2`. Pre a_4 potrebujem vypočítať a_3. Tam ale potrebujeme najskôr poznať hodnotu a_2. Pre a_2 potrebujeme poznať hodnotu a_1, ktorú už konečne vieme bez ďalších výpočtov.

Zhrňme si teda, čo už vieme o rekurzii.

Výkladový text

Rekurzia (po latinsky: recurrere = bežať znovu) vo všeobecnosti rieši problém tak, že vytvára akúsi kópiu seba s jednoduchším a menším problémom, tzv. **rekurzívny prípad**. Je dôležité zabezpečiť, aby sa rekurzia skončila, teda nastala situácia, keď je objekt vytváraný pre najjednoduchší problém - tzv. základný, **triviálny prípad**. Triviálny prípad je vlastne situácia, kedy už jednoznačne vieme povedať výsledok aktuálne riešeného problému. V triviálnom prípade je výsledná hodnota pevne daná, napr. číslom. Sekvencia menších problémov musí vždy smerovať k triviálnemu prípadu. Cieľom rekurzie je vlastne na základe sady pravidiel, zredukovať všetky ostatné hodnoty na základný prípad (76).

Rekurziu vo všeobecnosti možno použiť, ak riešenie nejakého problému vedie k riešeniu analogických problémov smerujúcich až k triviálnemu problému (79).

Rekurzia z pohľadu informatiky

V informatike je **rekurzia** definícia metódy (funkcie) pomocou samej seba. Pod rekurzívnym vyvolaním rozumieme také vyvolanie, ktoré nastáva skôr, než bolo predchádzajúce vyvolanie ukončené. Metódy, ktoré obsahujú rekurzívne vyvolanie, sa nazývajú rekurzívne.

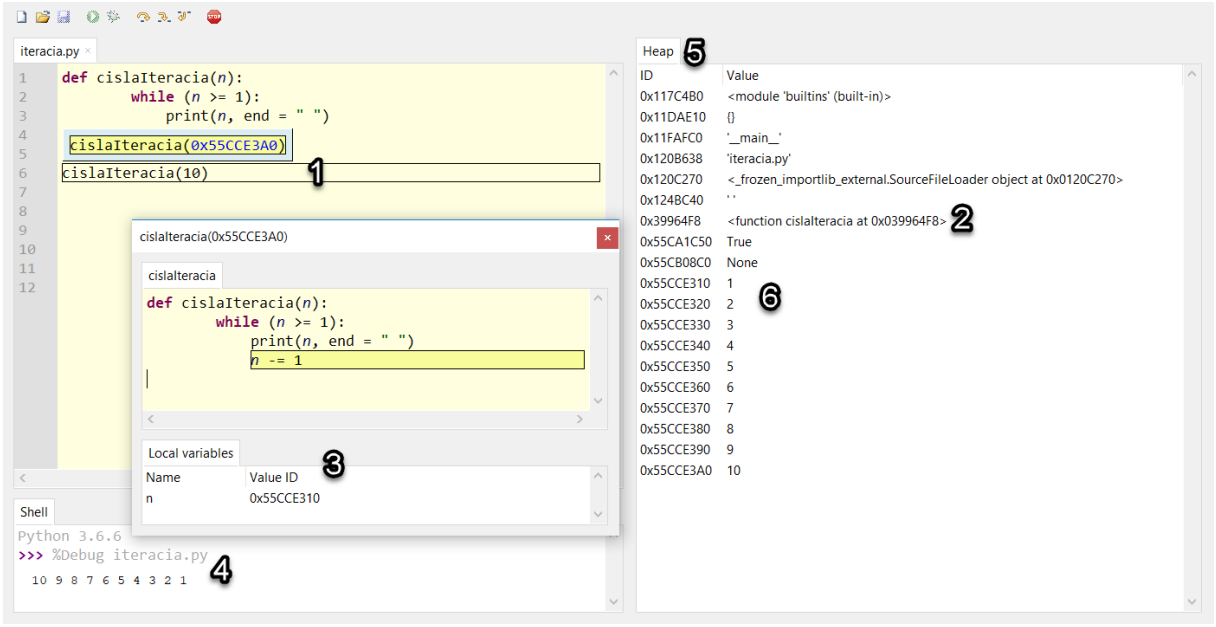
Úloha 3

Vytvorte funkciu pre výpis všetkých čísel od N po 1. Výpis bude v jednom riadku. Skôr, ako prejdeme k rekurzii, skúste navrhnúť jednoduché iteratívne riešenie. Iteratívne riešenie vyžaduje použitie cyklu, zamerajme sa na cyklus `while`.

Otvorme súbor `Rekurzia_uloha_3-4-5_z.py`. Našou úlohou je postupne doprogramovať všetky funkcie.

```
def cislaIteracia(n):
    while (n >= 1):
        print(n, end = " ")
        n -= 1
```

Aby sme neskôr pochopili, ako funguje rekurzia, zopakujme si mechanizmus volania podprogramu (pozri obrázok nižšie):



Obrázok 63 Ukážka mechanizmu volania podprogramu

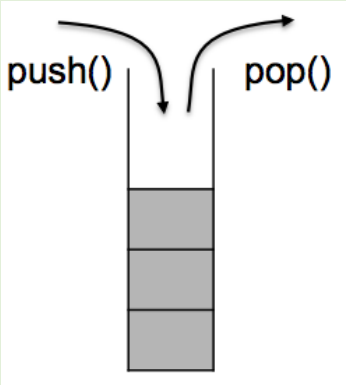
Zavolá sa podprogram (1), zapamätá sa návratová adresa (2) a vyhradí sa pamäť pre lokálne premenné, inicializujú sa parametre (3). Vykoná sa telo podprogramu (4). V zásobníku (5) sa potupne menia hodnoty lokálnych premenných (6). Po vykonaní tela programu sa uvoľní pamäť po lokálnych premenných a parametroch, program pokračuje na zapamätanej návratovej adrese. Krátku ukážku mechanizmu volania iteratívnej funkcie je možné vidieť aj vo videu `ITA-iteracia.mp4`.

Čo je to ten **zásobník**? Do zásobníkovej pamäte sa ukladajú všetky lokálne premenné volaných funkcií. Po ukončení funkcie, keď už sú nepotrebné, sa hodnoty z vrchu zásobníka odstránia

a „stratia sa“. Zásobník je teda špeciálne vyhradené miesto v pamäti, kde sa ukladajú informácie, ktoré nie sú potrebné pre celý program a po ich použití je možné sa ich jednoducho zbaviť. Ich platnosť je len lokálna, sú dočasné.

Výkladový text

Zásobník je v informatike dátová štruktúra, ktorá sa používa na dočasné uloženie dát. Označuje sa aj ako `stack` alebo aj `LIFO` (Last In First Out – posledný príde, prvý odíde). Pre zásobník je totiž typické to, že dáta, ktoré do zásobníka prišli ako posledné, odídu z neho ako prvé. Je to dátová štruktúra, ku ktorej sa pristupuje len zvrchu. Predstavme si kopu kníh. Keď chceme na kopu pridať ďalšiu knihu pridáme ju navrch – `push()`. Keď chceme odobrať, je logické zobrať knihu zhora – `pop()`. (80)



Zásobník sa používa na uchovávanie menných priestorov: každé ďalšie volanie funkcie, vytvorí nový menný priestor, ktorý sa uloží na koniec doteraz vytvorených menných priestorov. Keď ale príde ukončenie volania funkcie, z tohto zásobníka sa odstráni naposledy pridávaný menný priestor.

Úloha 4

Vytvorte funkciu pre výpis všetkých čísel od N po 1. Výpis bude realizovaný v jednom riadku. Skúsme teraz prerobiť riešenie na rekurzívne. Identifikujte ako sa dá problém rozložiť na triviálnu a rekurzívnu vetvu.

Zložitý problém môžeme riešiť rekurzívnym spôsobom tak, že:

1. Daný problém rozložíme na elementárne podproblémy, ktoré dokážeme jednoduchšie vyriešiť. Tieto podproblémy musia byť rovnakého typu. V našom prípade:
   a. hlavný problém *je výpis čísla N,*
   b. *podproblém, je výpis čísla N-1.* Čiže opäť riešime výpis, ktorý je jednoduchší.
2. Riešenie spočíva v opakovanom (resp. rekurzívnom) vykonávaní funkcie riešiacej daný problém. *Ak teda budeme funkciu volať s parametrom N, v opakovanom vykonaní ju budeme volať s parametrom N-1.*
3. Dôležitým bodom je určiť podmienku, kedy sa ukončí riešenie úlohy. *Úlohu ukončíme keď parameter N má hodnotu 1.*
4. Konečný výsledok je zlúčenie všetkých čiastkových výsledkov.

```
def cislaRekurzia(n):
    if (n == 1):                                    [1]
        print(n, end = " ")
    else:                                           [2]
        print(n, end = " ")
        cislaRekurzia(n-1)                          [3]
```

[1] V prípade ak n je rovné číslu 1 našli sme **triviálny prípad.** Stačí ak vypíšeme hodnotu n a nič viac.
[2] Ak je hodnota iná ako 1 našli sme **rekurzívny prípad**. Musíme vypísať hodnotu n ...
[3] ... a následne opakovane volať tú istú funkciu s parametrom zmenšeným o jedna.

Môžeme si všimnúť, že v našom prípade sa v triviálnom, ako aj rekurzívnom prípade používa ten istý príkaz výpisu, preto môžeme rekurziu zapísať aj inak.

```
def cislaRekurziaUprava(n):
    print(n, end = " ")
    if (n>1):
        cislaRekurzia(n-1)
```

Uvedený príklad prezentuje tzv. **chvostovú rekurziu**, kde sa rekurzívny prípad nachádza ako posledný príkaz vo funkcii (je na úplnom chvoste) (81).

Aký je rozdiel medzi mechanizmom volania rekurzívnej a nerekurzívnej metódy? Zatiaľ, čo v prípade iteratívnej verzie bola funkcia volaná len raz, v prípade rekurzívnej verzie bude funkcia volaná opakovane, vždy s rozdielnym parametrom. Aktuálne hodnoty premenných sa pred vnorením do ďalšej procedúry odkladajú do zásobníka. Rekurzívne riešenie úlohy je aj preto v porovnaní s iteračným pamäťovo náročnejšie. Po ukončení aktuálnej úrovne sa všetky nepotrebné informácie zo zásobníka zmažú a vykonávanie sa vráti o úroveň nižšie. Postupným návratom k uloženým úrovniam sa zo zásobníka vymažú všetky dočasne uložené informácie. Postupným riešením problémov s nižšou zložitosťou sa vyrieši pôvodný zložitejší problém. Krátku ukážku mechanizmu volania rekurzívnej funkcie je možné vidieť aj vo videu `ITA-rekurzia.mp4`.

Úloha 5

Skúsme výsledok v úlohe 4 upraviť tak, že miesto znižovania budeme parameter v rekurzívnej vetve zvyšovať, teda `cislaRekurzia(n+1)`. Diskutujte o tom, čo sa stalo na výpise?

Výkladový text

Program najskôr začne vypisovať čísla: `2 3 4 5 6 7 8 9 10 ...` Postupne sa vypisované číslo bude zvyšovať až sa vypíše chybové hlásenie `RecursionError: maximum recursion depth exceeded`, čo znamená, že sme prekročili maximálnu hranicu počtu volaní rekurzívnej funkcie. Došlo k pretečeniu zásobníka, alebo tzv. `stack overflow`, čo znamená, že sme prekročili kapacitu zásobníkovej pamäte.

Našim nesprávnym riešením sme vytvorili nikdy nekončiace rekurzívne volanie. V skutočnosti sa však proces sám ukončil po naplnení zásobníka.

Otvorme súbor `Rekurzia_uloha_6-7-8_z.py`. Našou úlohou je doprogramovať `faktorial` a `NSD`.

Úloha 6

Rekurentná definícia výpočtu `n!` (čítame ako n-faktoriál) má takýto tvar:

$$n! \begin{cases} n * (n-1)! & n>0 \\ 1 & n=0 \end{cases}$$

Identifikujte triviálny prípad (pre ktoré n vieme jednoznačne povedať výsledok) a navrhnite rekurzívny prípad (volanie rovnakej funkcie s nižšou zložitosťou).

V zmysle rekurentnej definície možno napr. `4!` vypočítať ako: `4*3!=4*(3*2!)=4*(3*(2*1!))=4*(3*(2*(1*0!)))=4*(3*(2*(1*1))=120`.

V predchádzajúcom riešení boli volania funkcií na sebe „nezávislé“. Každá funkcia si vykonala svoju činnosť (výpis čísla príkazom `print`), zavolala novú funkciu s novým parametrom a koniec.

V tomto prípade sú však volania na sebe závislé, pretože výpočet na jednej úrovni nie je možné ukončiť bez hodnoty, ktorá sa vypočíta pri vnorenom volaní. Výpočet `n!` vedie k analogickému problému, ktorým je výpočet `(n-1)!`.

Je nutné aby rekurzívna vetva vrátila hodnotu späť do úrovne, v ktorej bola volaná.

```
def faktorial(n):
    if n == 0:
        return 1                                        [1]
    else:
        return n * faktorial(n-1)                       [2]

# ......  alternatívne riešenie  ..............................

def faktorial_uprava(n):
    if n == 0:
        return 1
    return n * faktorial(n - 1) # do tejto časti sa program dostane
len ak neplatí podmienka v triviálnej vetve
```

[1] Ak metódu volám s parametrom 0 (počítam hodnotu 0!), tak návratová hodnota bude 1.

[2] Ak metódu volám nenulovým parametrom, tak návratová hodnota bude `n*[hodnota vypočítaná vo vnorenej funcii]`.

Rekurzívna funkcia `faktorial(n)` vráti celočíselnú hodnotu, ktorú je nutné ešte spracovať, vypísať alebo uložiť v nejakej premennej.

```
print(faktorial(4))
```

Alebo ak chceme vypísať tabuľku všetkých faktoriálov od 0 po n, musíme použiť cyklus.

```
for i in range(15):
    print('{:2}! = {}'.format(i, faktorial(i)))
```

Krátku ukážku mechanizmu volania rekurzívnej funkcie s návratovou hodnotou je možné vidieť aj vo videu `ITA-faktorial.mp4`.

Úloha 7

Naprogramujte Euklidov algoritmus, ktorý slúži na výpočet najväčšieho spoločného deliteľa (NSD) dvoch kladných čísel. Algoritmus je založený na postupnom odčítaní, respektíve na opakovanom použití nasledujúcich pravidiel.

$$NSD(a,b) \begin{cases} a \text{ alebo } b & a == b \\ NSD(a,b-a) & a<b \\ NSD(a-b,b) & a>b \end{cases}$$

Výkladový text

Najväčší spoločný deliteľ (NSD) z dvoch kladných celých čísel, je najväčšie prirodzené číslo, ktoré delí obe čísla bezo zvyšku. `NSD(20,90) = 10; NSD(14,42) = 14; NSD(3,19) = 1`

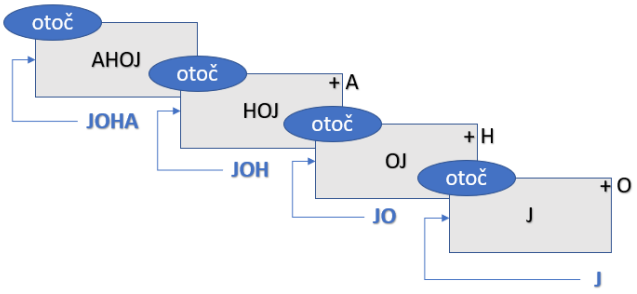
Algoritmus vlastne v každom kroku kontroluje, či sú čísla rovnaké. Ak nie sú, nahradí väčšie číslo ich rozdielom. Triviálna vetva je jednoznačne identifikovateľná. Ak do funkcie pošleme dva rovnaké parametre, môžeme jeden z nich vyhlásiť za riešenie. V opačnom prípade máme dve možnosti: dve rekurzívne vetvy závislé od aktuálnych hodnôt parametrov.

```
def NSD(a, b):
    if a == b:
        return a
    elif a < b:
        return NSD(a, b-a)
    elif a > b:
        return NSD(a-b,b)
```

Úloha 8

Naprogramujte rekurzívnu funkciu, ktorá obráti textový reťazec, teda vytvorí jeho zrkadlový obraz. Postup v rekurzívnej funkcii bude mať takýto priebeh (pozri obrázok nižšie):

1. Ak má reťazec aspoň dva znaky, tak prvý znak reťazca odtrhnem, inak ukončím volanie.
2. Skrátim pôvodný reťazec o prvý znak. K odtrhnutému znaku pripojím výsledok, ktorý sa vráti z opätovne zavolanej funkcie s parametrom so skráteným reťazcom.



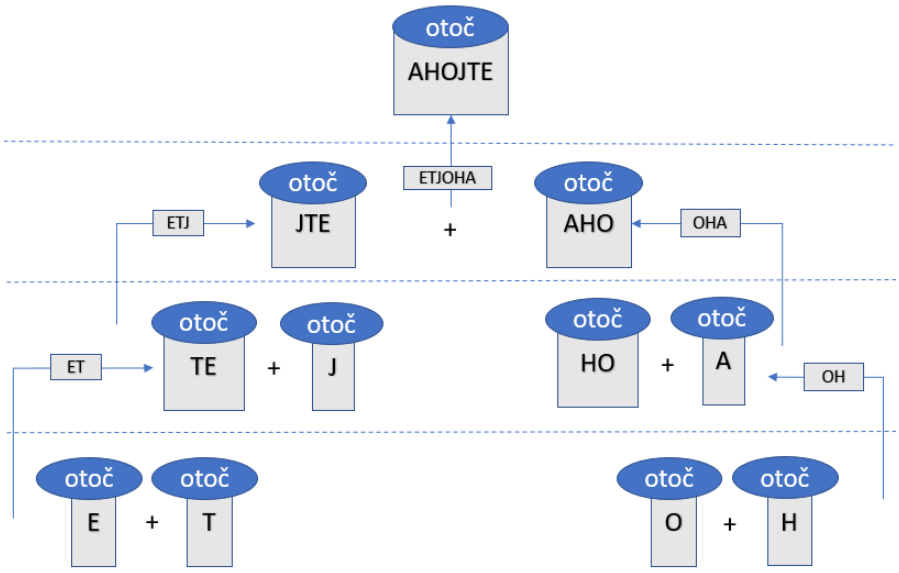
Obrázok 64 Vizualizácia úrovní volania rekurzívneho algoritmu na otočenie reťazca (podrobnejšie vo videu ITA-otocRetazec.mp4)

```
def otocRetazec(ret):
    if (len(ret)<=1):
        return ret
    return otocRetazec(ret[1:]) + ret[0]
```

Krátku ukážku rekurzívneho otočenia reťazca je možné vidieť aj vo videu `ITA-otocRetazec.mp4`. Experimentujte s volaním. Skúste, ako sa správa algoritmus pre rôzne vstupy.

```
print(otocRetazec("abc"))
print(otocRetazec("Python"))
print(otocRetazec(500*"abc"))
```

Pri vstupnom parametre dlhšom ako 1000 znakov nám však algoritmus prestane fungovať. Dosiahli sme totiž najväčší povolený počet rekurzívnych volaní. Existuje však spôsob, ako to vyriešiť. Môžeme si problém rozdeliť na menšie časti, ktoré budeme otáčať oddelene a až po otočení ich spojíme. Takto by sme jeden veľký problém rozdelili na množstvo maličkých problémov, ktoré sú riešiteľné rýchlo a zároveň sa nepresiahne najvyšší povolený počet rekurzívnych vnorení (pozri obrázok nižšie). Ako je to možné? Po vyriešení každého malého podproblému sa časť nepotrebných informácií zo zásobníkovej pamäti jednoducho odstráni, čím sa uvoľní miesto pre nové hodnoty.



Obrázok 65 Vizualizácia úrovní volania rekurzívneho algoritmu na otočenie reťazca metódou Rozdeľuj a panuj

```
def otocRetazecRozdel(ret):
    if (len(ret)<=1):
        return ret
    else:
        lstrana = otocRetazecRozdel(ret[:len(ret)//2])      [3]
        pstrana = otocRetazecRozdel(ret[len(ret)//2:])      [4]
        return pstrana + lstrana                            [5]
```

[3] obrátim všetko od začiatku po stred,

[4] obrátim všetko od stredu po koniec,

[5] po otočení oboch polovíc textu pošlem spojený reťazec o úroveň vyššie.

Výkladový text

Metóda, ktorú sme použili sa nazýva Rozdeľuj a panuj. Jej postup môžeme popísať v troch krokoch.

Rozdeľuj: Rozdeľ problém na niekoľko menších podproblémov.

Panuj: Každý podproblém vyrieš samostatne rekurzívnym volaním. Ak sú podproblémy dostatočne malé, vyrieš ich priamočiaro.

Kombinuj: Skombinuj riešenia menších podproblémov do riešenia pôvodného veľkého problému.

Túto metódu je možné použiť pri triedení (Quicsort), či násobení veľkých čísel. (82)

Úloha 9

Pre prvky v množine vytvorte ich permutácie. Inak povedané, musíte nájsť všetky odlišné usporiadania prvkov množiny,

Skúsme si najskôr predstaviť ako má také usporiadanie vyzerať.

Pre množinu {1} je permutácia jednoduchá, tá istá množina.

```
P[1] = 1
```

Pre množinu {1, 2} sú dve odlišné permutácie.

```
P[1 2] = 12; 21
```

Pre množinu {1, 2, 3} je permutácii už šesť.

```
P[1 2 3] = 123; 132; 213; 231; 312; 321
```

Sledujme výsledné permutácie troch prvkov a všimnime si „vzorec“. Najskôr vyberieme jeden prvok množiny. A pripojíme k nemu všetky permutácie pre zvyšné prvky dva prvky množiny. Ak teda chcem postup zovšeobecniť, môžeme napísať, že pre množinu {1, 2, 3, … n} budú permutácie vyzerať nasledovne.

```
P[1 2 3 … N] = 1P[2 3 … N]; 2P[1 3 … N]; … ; NP[1 2 … N-1]
```

Vidíme, že aj tu je problém riešený rekurzívne. Na zistenie jednej permutácie potrebujeme vybrať jeden prvok a vedieť permutáciu zvyšných prvkov. Triviálnym prípadom funkcie `permutacie` bude permutáciajednoprvkovej množiny.

```
def permutacie(lst):
# Ak je v zozname lst len jeden prvok, je možná len jediná
permutácia
   if len(lst) == 1:
       return [lst]
```

Ak má zoznam viac ako jeden prvok, potrebujeme vytvoriť nový zoznam na ukladanie aktuálnej permutácie, kde ku každému prvku zoznamu musíme pripojiť permutáciu zvyšných prvkov

```
  else:
    out = []                                                    [6]
    for i in range(len(lst)):                                   [7]
       prvok = lst[i]
       skratenyLst = lst[:i] + lst[i+1:]                        [8]
       for perm in permutacie(skratenyList]):                   [9]
           out += [prvok + perm]
  return out
```

[6] Potrebujeme prázdny zoznam na ukladanie aktuálnej permutácie.

[7] Ku každému prvku zoznamu musíme pripojiť permutáciu zvyšných prvkov. Preto vyberieme i-ty prvok zoznamu.

[8] Vytvoríme nový zoznam, ktorý obsahuje všetky prvky okrem aktuálneho.

[9] Vygenerujme všetky permutácie, kde prvok vybraný v bode 7 je prvým prvkom zoznamu.

Ako parameter funkcie môžeme poslať ľubovoľný zoznam znakov.

```
l = permutacie(list('AHOJ'))
print(l)
```

Rekurzia vs. Iterácia

Ako už bolo spomínané na začiatku kapitoly, v niektorých prípadoch je síce rekurzívna implementácia riešenia problému krajšia a čitateľnejšia, avšak môže byť obrovsky neefektívna. Ako vzorovú ukážku si predstavíme výpočet členov Fibonacciho postupnosti. Aké číslo nasleduje v postupnosti 0, 1, 1, 2, 3, 5, 8, 13 ...? Na základe čoho ste ho vypočítali? Napíšte rekurzívnu a nerekurzívnu funkciu pre výpočet čísel tzv. Fibonacciho postupnosti.

$$fib(n) \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & a>b \end{cases}$$

Spustite program `fibonacci.py` a diskutujte o výpočtových zložitostiach oboch verzií algoritmu.

Iteratívne

```
def fib_iterativne(self, n):
    if (n == 0) or (n == 1):
        return 1
    else:
        fib0 = 0
        fib1 = 1
        for i in range(2,n+1,1):
            fib0, fib1 = fib1, fib0 + fib1
        return fib1
```

Rekurzívne

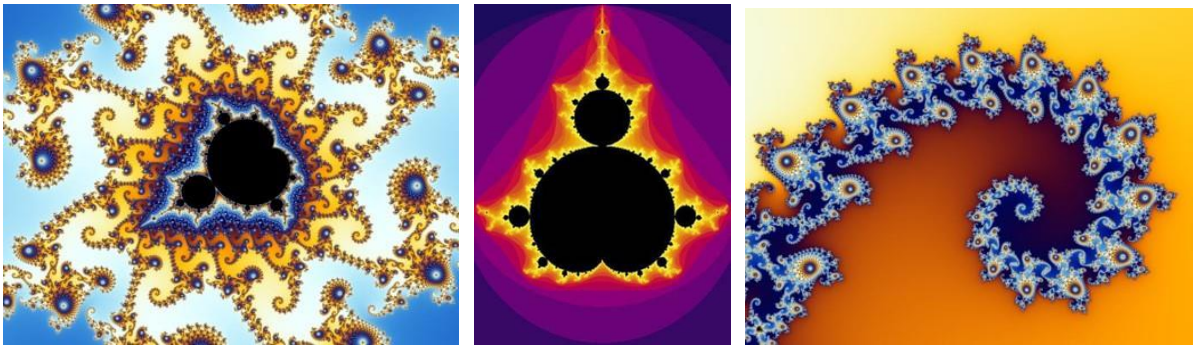
```
def fib(self, n):
    if  n== 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Rekurzívne krivky, fraktály

Väčšina vyššie riešených úloh by sa dala zhrnúť v jednom odporúčaní: pokiaľ je možné daný problém vyriešiť pomocou iterácie, vyhýbajme sa rekurzii. Existujú však situácie, kedy použitie rekurzívneho prístupu je viac než žiadané. Jednou z nich sú rekurzívne kresby, **rekurzívne krivky**, niekedy označované aj ako **fraktály** (pozri obrázok nižšie). Mnohé krivky sú také známe, že sú jednoznačne identifikovateľné podľa mena, napríklad. C-krivka, Dračia krivka, Kochova vločka, Sierpinskeho koberec. V ďalšej kapitole budeme používať knižnicu `turtle`.

Výkladový text

Fraktály sú objekty, ktoré poskytujú rovnakú úroveň detailov bez ohľadu na rozlíšenie, ktoré používame. Fraktály je možné nekonečne zväčšovať tak, že pri každej novej zväčšenine sú zreteľné niektoré detaily, ktoré pred zväčšením neboli viditeľné a množstvo nových detailov je vždy približne rovnaké. Hoci sú tieto detaily podobné, pozostávajú z menších verzií seba samých, sú príliš nepravidelné na to, aby boli opísané jednoduchou geometriou. Autorom tohto pojmu je matematik Benoit Mandelbrot. (83)



Obrázok 66 Počítačom generované fraktálové krivky

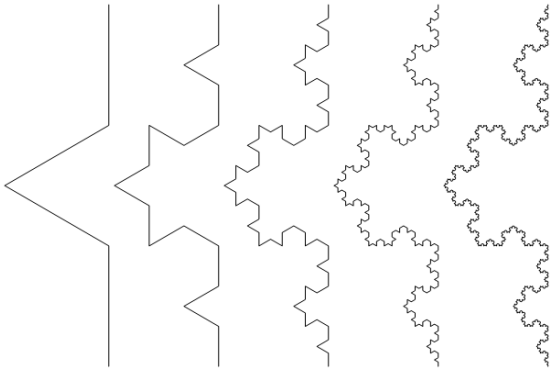
Úloha 10

Na obrázku (Obrázok 66) sú počítačom generované krivky. Vedeli by ste podobné útvary nájsť aj v bežnej prírode?

Otvorme súbor `Rekurzivne_krivky.py`.

Úloha 11

Princíp kreslenia rekurzívnych kriviek si vysvetlíme na známej Kochovej krivke. Navrhnite rekurzívny algoritmus, ktorý na základe zadaných parametrov určujúcich dĺžku a počet úrovní vykreslí krivku podľa obrázku (pozri obrázok nižšie).



Obrázok 67 Kochová krivka s piatimi rozdielnymi úrovňami

Skôr, ako pristúpime k samotnému programovaniu analyzujme, z čoho sa Kochová krivka (Obrázok 67) skladá. Za základný tvar môžeme považovať čiaru zadanej dĺžky (pozri obrázok nižšie, (84)).

Obrázok 68 Základ Kochovej krivky je čiara konkrétnej dĺžky

Základná čiara sa na ďalšej úrovni rozdelí na tri samostatné segmenty, ktoré sú spolu tvorené štyrmi čiarami (pozri obrázok nižšie):

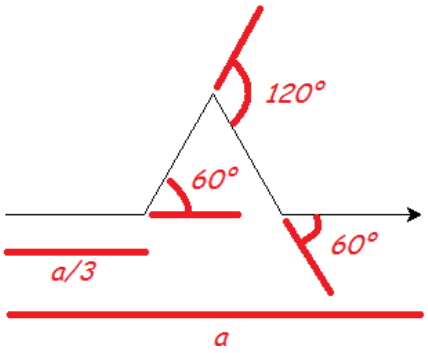
- Segment 1: Čiara tretinovej dĺžky.
- Segment 2: Lomená čiara (ramená rovnostranného trojuholníka) tretinovej dĺžky.
- Segment 3: Čiara tretinovej dĺžky.

Obrázok 69 Základná čiara v Kochovej krivke sa rozdelí na tri segmenty

Proces delenia čiar sa v ďalšej úrovni zopakuje podľa rovnakého postupu (pozri obrázok nižšie). Každá z čiar, sa rozdelí na tri samostatné segmenty.

Obrázok 70 Kochova krivka

Prvé, čo si musíme uvedomiť, že v každej úrovni sa veľkosť čiary, z ktorej sa krivka skladá skracuje. Ak je teda dĺžka základnej čiary `a = 300`, tak na prvej úrovni má každý segment veľkosť `100` a na druhej približne `33`. Matematicky by sme pohyb korytnačky mohli zapísať nasledovne (pozri obrázok nižšie).



Obrázok 71 Matematický zápis pohybu korytnačky pre Kochovu krivku (84)

Funkcia `ciara1` vykreslí Kochovu krivku nultej alebo prvej úrovne. Korytnačka sa v každom kroku prvej úrovne posunie o tretinu dĺžky strany a po vykreslení sa natočí správnym smerom.

```
def ciara1(uroven, a):
    if (uroven == 0):
        forward(a)
    else:
        forward(a/3)
        left(60)
        forward(a/3)
        right(120)
        forward(a/3)
        left(60)
        forward(a/3)
```

Aby sme kód skrátili uvažujme, že korytnačka sa otáča len doľava. Miesto príkazu `right 120` môžeme použiť príkaz `left -120`. Vznikne nám opakujúca sa sekvenciu príkazov s rôznymi parametrami, a tak čiaru môžeme nakresliť aj v cykle.

```
def ciara2(uroven, a):
    if uroven == 0:
        forward(a)
    else:
        for t in [60, -120, 60, 0]:
            forward(a//3)
            left(t)
```

My však potrebujeme, aby **sa každý segment stal samostatnou kochovou krivkou**. Štyri čiary sa ďalej rozdelia na šestnásť menších a tie sa v ďalšej úrovni rozdelia na ešte menšie. Úprava na rekurzívnu verziu, kde sa každý kreslený segment stáva novou krivkou, je jednoduchá. Príkaz `forward` nahradíme volaním funkcie `koch` (funkcia volá samú seba).

```
def koch(uroven, a):
    if uroven == 0:
        forward(a)
    else:
        for t in [60, -120, 60, 0]:
                koch(uroven - 1, a/3)                    [4]
                left(t)
```

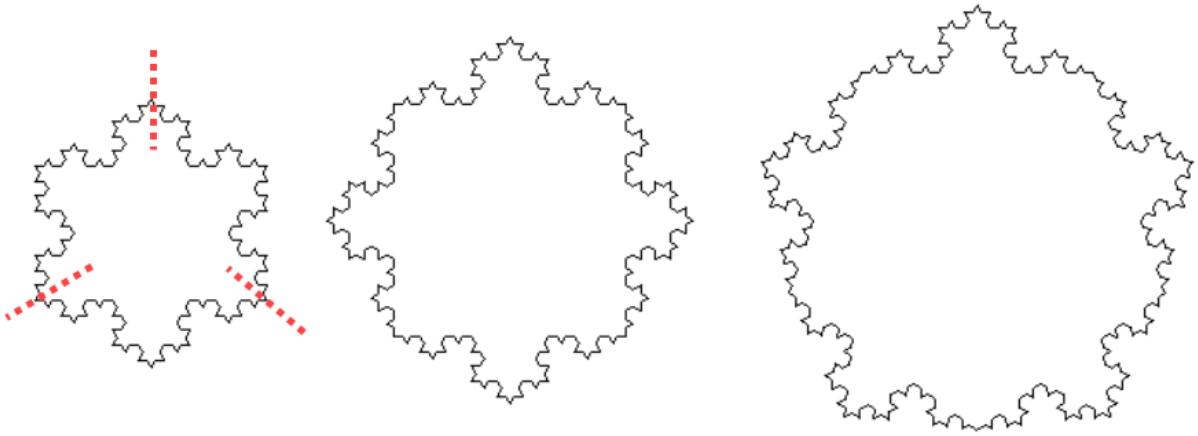
Funkcia `koch` zavolá funkciu `koch` s upravenými parametrami. Rekurzia si v zásobníku pamätá, ktorú úroveň práve „počíta“. V každej úrovni je v porovnaní s predchádzajúcou úrovňou vykresľovaná čiara tretinovej veľkosti. Zavolajme rekurzívnu funkciu.

```
setscreen(1500,500)
koch(3,200)
```

Skúsme meniť hodnotu parametra `uroven` a sledujme ako sa ten istý 3-segmentovy vzor opakuje pri rôznych veľkostiach a uhloch. Celá krivka obsahuje miniatúrne verzie, menšie a menšie, až do „neviditeľnosti“. Hovoríme, že krivka je samo-podobná, čo je základná vlastnosť fraktálov.

Úloha 12

Použite Kochovu krivku na vykreslenie „snehovej vločky“. Počet strán a veľkosť vločky zadajte cez parameter.



Obrázok 72 Opakovaným volaním Kochovej krivky s pootočením korytnačky môžeme dosiahnuť efekt snehovej vločky

Keď si uvedomíme, že základný tvar kochovej krivky je čiara, stačí zadanie transformovať na vykreslenie n-uholníka, kde sa miesto každej strany vykreslí Kochova krivka s danou úrovňou (Obrázok 72).

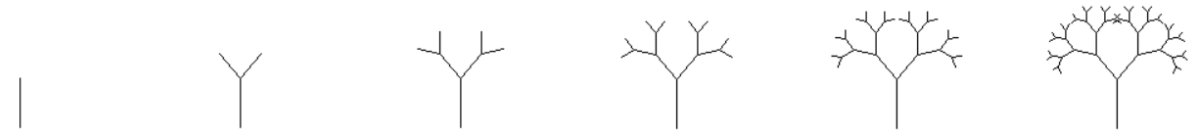
```
def kochovaVlocka(pocet, velkost):
    for i in range(pocet):
        koch(pocet,velkost)
        right(360/pocet)

kochovaVlocka(3, 150)
```

Použitím príkazov `begin_fill` a `end_fill` môžeme pre vločku definovať aj farbu výplne.

```
def kochovaVlockaFarebna(pocet, velkost):
    fillcolor("blue")
    begin_fill()
    kochovaVlocka(pocet, velkost)
    end_fill()

kochovaVlockaFarebna(3, 150)
```

Úloha 13

Vytvorte funkciu `stromPravidelny(n, dlzka)` na vykreslenie binárneho stromu. Úroveň stromu nám podobne, ako pri iných rekurzívnych krivkách hovorí o počte rekurzívnych vnorení pri kreslení. Dĺžka nám hovorí o veľkosti kmeňa a vetiev.



Obrázok 73 Binárny strom s úrovňou 0 až 5

Vzhľad stromu závisí od úrovne, na ktorej ho vykresľujeme. Ak je **úroveň = 0**, nakreslí sa len čiara nejakej dĺžky (kmeň). Pri kreslení Kochovej krivky sme vždy išli smerom vpred. Binárny strom sa však v jednom bode rozvetvuje, a preto je potrebné sa vždy po nakreslenej čiare vrátiť do bodu rozvetvenia.

```
if n == 0:
    forward(dlzka)
    back(dlzka)
```

Pre vyššiu úroveň, sa najprv nakreslí čiara, potom sa na jej konci najprv nakreslí celý ľavý binárny strom úrovne (n-1), a potom opäť binárny strom úrovne (n-1) na pravej strane (Obrázok 73).

```
    else:
        forward(dlzka)
        left(40)                                    [10]
        strom(n - 1, dlzka*2/3)                     [11]
        right(80)                                   [12]
        strom(n - 1, dlzka*2/3)                     [13]
        left(40)                                    [14]
        back(dlzka)
```

[10] Keď je korytnačka v bode, kde sa strom začína rozvetvovať, je potrebné natočiť korytnačku o nejaký uhol doľava.

[11] V novom smere vykreslíme ľavý podstrom s kratšou dĺžkou (dve tretiny pôvodnej).

[12] Po vykreslení ľavého podstromu sme sa vrátili do pôvodného bodu. Teraz potrebujeme korytnačku natočiť novým smerom.

[13] V novom smere vykreslíme pravý podstrom s kratšou dĺžkou (dve tretiny pôvodnej).

[14] Po vykreslení pravého podstromu sme sa vrátili do pôvodného bodu. Potrebujeme korytnačku vrátiť do smeru, aký mala na začiatku v momente, keď sme začali kresliť rozvetvovanie. Následne sa korytnačka vráti späť.

Úloha 14

Vytvorte funkciu `stromNepravidelny(n, dlzka)` tak, aby vykresľovaný strom nebol symetrický, ale aby sa svojou nevyrovnanou korunou čo najviac podobal reálnemu stromu (pozri obrázok nižšie).

Obrázok 74 Nepravidelný binárny strom s úrovňou 1 až 5

Riešením úlohy je dať do rekurzívnej vetvy prvok náhody pre dĺžku čiar ako aj uhol natočenia.

```
else:
    forward(dlzka)

    nova_dlzka_lavy = dlzka * random.randint(40, 80) / 100    [10]
    nova_dlzka_pravy = dlzka * random.randint(40, 80) / 100
    uhol_lavy = random.randint(20, 40)                        [11]
    uhol_pravy = random.randint(20, 40)

    left(uhol_lavy)                                           [12]
    stromNepravidelny(n - 1, nova_dlzka_lavy)
    right(uhol_lavy + uhol_pravy)                             [13]
    stromNepravidelny (n - 1, nova_dlzka_pravy)
    left(uhol_pravy)                                          [14]

    back(dlzka)
```

[10] Výpočet dĺžky čiary pre nasledujúce volanie. V našom prípade sa náhodne vyberie dĺžka v intervale 40% až 80% dĺžky v aktuálnej úrovni. Ľavý aj pravý podstrom budú mať rozdielne dĺžky.

[11] Výpočet uhla otočenia pre podstromy. V našom prípade sa náhodne vyberie uhol z intervalu 20° až 40°. Ľavý aj pravý podstrom budú mať rozdielny náklon.

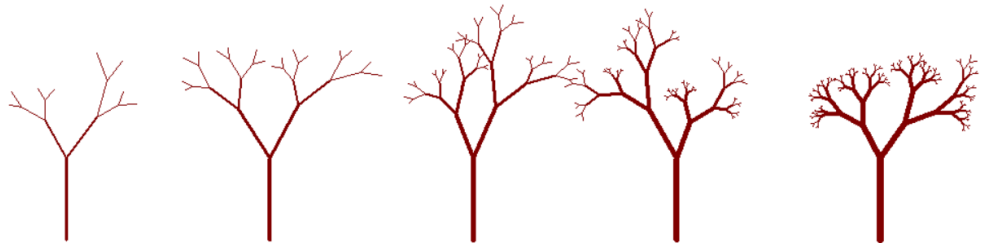
[12] Natočíme korytnačku na kreslenie ľavej časti.

[13] Vrátime korytnačku na pôvodný smer a zároveň ju natočíme na kreslenie pravej časti.

[14] Vrátime korytnačku na pôvodný smer.

Úloha 15

Upravte funkciu `stromNepravidelny(n, dlzka)` tak, aby sa s pribúdajúcimi úrovňami hrúbka vetiev stenčovala. Zmeňme farbu vetiev (pozri obrázok nižšie).



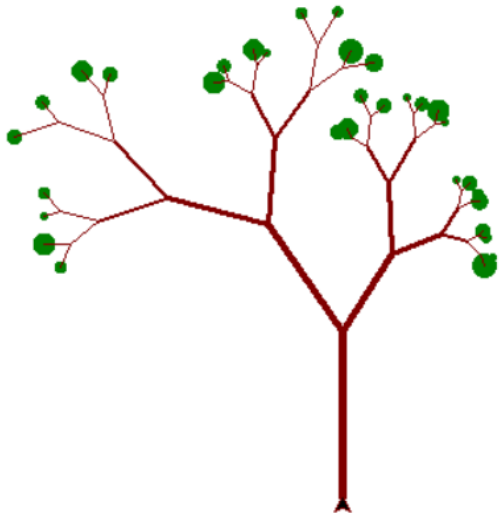
Obrázok 75 Nepravidelný binárny strom s meniacou sa hrúbkou vetiev

Úlohu môžeme elegantne vyriešiť tak, že hrúbka kreslenej čiary bude závisieť od toho, v akej úrovni sa nachádzame.

```
def stromNepravidelny(n, dlzka):
    pencolor("maroon")
    pensize(n)
    if n == 0:
        ...
```

Úloha 16

Aktuálne stromy vyzerajú akoby už bola jeseň a opadli všetky listy. Nakreslime preto na vrcholových vetvách listy (pozri obrázok nižšie).



Obrázok 76 Nepravidelný binárny strom s meniacou sa hrúbkou vetiev, s listami v korune stromov

Keďže listy chceme kresliť len na najnižšej úrovni, teda na vrchole stromu, stačí nám upraviť triviálnu vetvu programu.

```
if n == 0:
    forward(dlzka)
    velkost = random.randint(5,15)                    [15]
    dot(velkost, 'green')
    back(dlzka)
```

[15] Vyberieme priemer kresleného listu a vykreslíme ho ako bod zelenej farby.

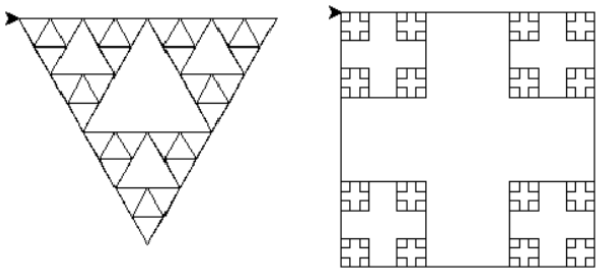
Ďalšie úlohy na precvičenie a zamyslenie

1. Naprogramujte funkciu `mocnina(x,n)`, ktorá vypočíta `n-tú` mocninu čísla `x`. Rekurentný zápis problému je nasledovný:

$$x^n \begin{cases} 1 & n=0 \\ x * x^{n-1} & n>0 \end{cases}$$

2. Naprogramujte funkciu `sucin(a, b)`, ktorá vypočíta bez použitia cyklov súčin dvoch nezáporných celých čísel a to len pomocou sčitovania.

3. Nakreslite nasledujúce rekurzívne obrázky.



15. Backtracking: Hľadanie riešenia s možným návratom

Kľúčové slová

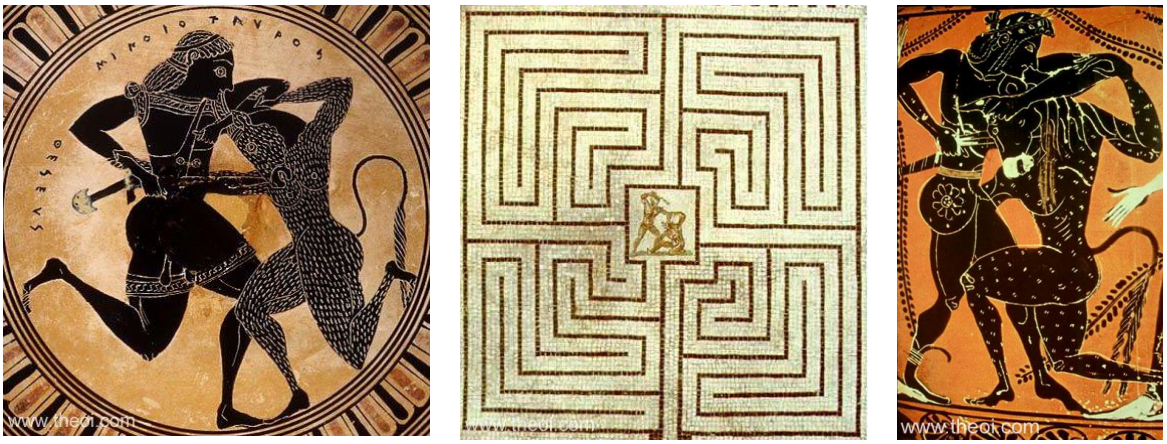
backtracking, rekurzia, hľadanie všetkých riešení, zásobníková pamäť, prehľadávanie s návratom

Čo sa naučíme a čo si precvičíme

- naučíme sa ako na hľadať riešenia rôznych problémov s možným návratom,
- naučíme sa spôsob ako hľadať všetky riešenie pre vybrané typy problémov,
- precvičíme si ako funguje zásobníková pamäť,
- naučíme sa orientovať v bludisku na základe vopred definovaných pravidiel.

Problémová situácia

Možno ste už počuli o Minotaurovi, o postave s telom človeka a hlavou býka. Celú legendu je možné prečítať si v knihe zaoberajúcou sa gréckou mytológiou. V skratke si len priblížme, že Minotaurus bol uväznený v labyrinte a obyvatelia Atén mu museli z času na čas obetovať mládencov a dievčatá. Ale len do doby, kedy sa Théseus rozhodol s Minotaurom skoncovať (pozri obrázok nižšie). Aby sa však Théseus v labyrinte nestratil mal Ariadninu niť (dar od bohyne Afrodity), ktorú celú cestu odvíjal a s pomocou ktorej našiel cestu v bludisku a von z bludiska. Ktovie, ako by to dopadlo, keby Théseus nemal možnosť vrátiť sa späť zo slepej uličky a pokračovať inou cestou (85).



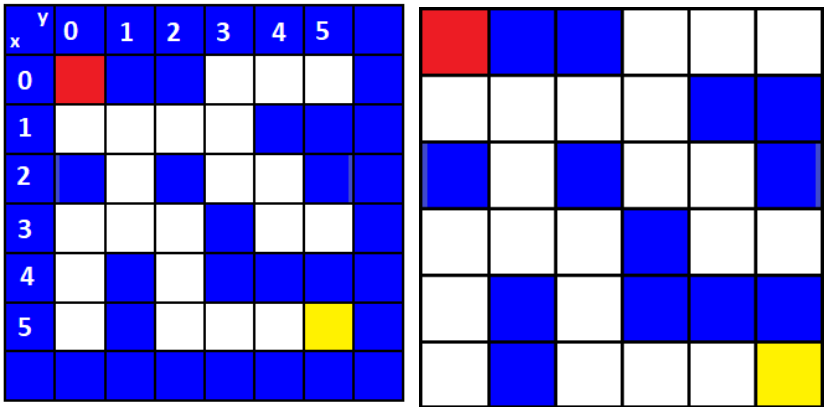
Toledo Museum of Art, Toledo | Villa de la via Cadolini (in situ), Cremona | Gregorian Etruscan Museum, Vatican City

Obrázok 77 Rôzne zobrazenia súboja Mínotaura a Thésea

V tejto kapitole si ukážeme ako vyriešiť problém hľadania cesty v neznámom prostredí, v našom prípade hlavne v bludisku.

Úloha 1

Skúsme sa zahrať na Thésea a Minotaura. Théseus vojde do labyrintu v políčku [0][0] (červená bunka). Napíšte, ako sa má dostať k Minotaurovi (žltá bunka), ak platí, že na jedno miesto môže stúpiť len raz. Modré bunky predstavujú steny a biele prestavujú miesto, kam je možné vkročiť (pozri obrázok nižšie). Na ktoré bunky Theseus vkročí? Očíslujte ich v poradí, v akom na políčka stúpi.



Obrázok 78 Labyrint s Minotaurom

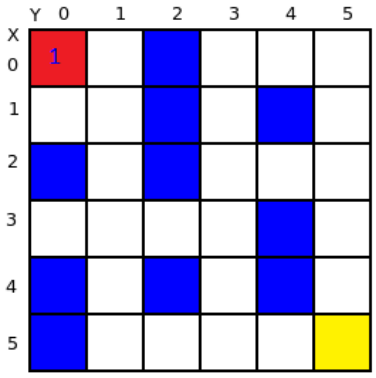
Ďalej už budeme pracovať s labyrintami, kde obvodové múry nebudeme vykresľovať, ale ich existenciu automaticky predpokladať.

Otázky pre žiakov

Koľko krokov musel Théseus vykonať? Vedel v každom kroku kde sa má pohnúť? Théseus totiž nevidel celý labyrint ale videl len bezprostredne okolo seba.

Úloha 2

Skúsme sa ešte raz zahrať na Thésea a Minotaura. Zadanie je rovnaké ako v prvej úlohe, len zameníme bludisko (pozri obrázok nižšie).



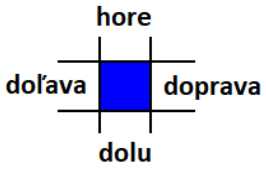
Obrázok 79 Labyrint s Minotaurom v inom prevedení

Otázky pre žiakov

Spýtajme sa tie isté otázky. Koľko krokov musel Théseus vykonať? Vedel v každom kroku kade sa má pohnúť?

Úloha 3

Skúsme sa posledný raz zahrať na Thésea a Minotaura. Théseus vojde do labyrintu v políčku [0][0] (červená bunka). Skúsme Théseovi nastaviť presné pravidlá pohybu, pričom budeme uvažovať o smeroch hore, doľava, doprava a dolu (pozri obrázok nižšie).



Obrázok 80 Pravidlá pohybu v štyroch smeroch

1. Keď vojdeš na nejaké políčko, kde si ešte nebol poznač si číslo kroku, s ktorým si sem vkročil.
2. Pokračuj ďalej:
    a. Skontroluj, či sa dá ísť smerom doprava, ak áno choď a pokračuj bodom 1.
    b. Skontroluj, či sa dá ísť smerom hore, ak áno choď a pokračuj bodom 1.
    c. Skontroluj, či sa dá ísť smerom doľava, ak áno choď a pokračuj bodom 1.
    d. Skontroluj, či sa dá ísť smerom dolu, ak áno choď a pokračuj bodom 1.
3. Ak sa ti nepodarilo ísť z daného miesta žiadnym smerom, zmaž stopu o tom, že si tu bol a vráť sa o krok späť.

Aké bude prvé riešenie, ktoré nájdete pre labyrint z úlohy 2?

Poznámka pre žiaka

Zjednodušene povedané, ideme stále vpravo. Keď sa nedá, snažíme sa spraviť krok hore a opäť sa snažíme ísť doprava. Ak sa nedá ísť ani hore, skúsime ísť smerom dolu a na novej bunke sa opäť snažíme posúvať doprava.

Programátorská stratégia, ktorá nám umožňuje v momente, keď sme sa "zasekli", vrátiť sa o krok späť a pokračovať iným smerom sa nazýva **backtracking**, často sa označuje aj ako prehľadávanie s návratom.

Výkladový text

Backtracking rieši problém tak, že hľadá riešenie prvého čiastkového problému, a keď ho nájde pokúša sa rekurzívne vyriešiť ďalšie čiastkové problémy. Postup si môžeme popísať v bodoch (86):

[1] Ak nás aktuálny smer riešenia problému nepriviedol k riešeniu a už nemáme možnosť pokračovať, tak v tomto kroku končíme.

[2] Zmažeme po sebe informácie, čo sme robili v danom kroku a vrátime sa o krok späť.

[3] Po návrate sa snažíme hľadať ďalšie možné riešenie „iným smerom".

[4] Proces hľadania riešenia daným smerom a návratu v prípade, ak sme boli neúspešný sa úplne skončí vtedy, keď sme sa po overení možností vrátili až k prvému čiastkovému

problému, pre ktorý už boli tiež overené všetky možnosti. Inak povedané, ak sme skúsili všetko a aj tak sme sa vrátili na začiatok.

Jednou z kľúčových vecí pri spätnom návrate je rekurzia. V prípade backtrackingu máme viacero nezávislých rekurzívnych vetiev, ktoré sa vykonávajú postupne. Začneme tým, že si vyberieme prvú z rekurzívnych možností, pričom k ďalším sa vrátime až vtedy, ak dospejeme k záveru, že aktuálne vykonávaná možnosť neposkytuje požadované riešenie. Tieto kroky budeme opakovať dovtedy, kým nebudeme mať požadované riešenie alebo neprejdeme každú z dostupných rekurzívnych možností (87).

Backtracking môžeme považovať za jednu z metód vyhľadávania typu rozdeľuj a panuj (anglicky divide and conquer, latinsky Divide et Impera), kde sa problém rieši tak, že sa rozdelí na menšie časti. Delenie prebieha dovtedy, až na niektorej úrovni dostaneme problém, ktorý je už triviálne riešiteľný.

Úloha 4

Naprogramujte príbeh z gréckej mytológie. Načítajte (vygenerujte) bludisko, umiestnite na konkrétne miesto Minotaura a navrhnite ľubovoľnú cestu pre Théseusa, ktorý bude vchádzať vždy z miesta so súradnicami `[0][0]`. Použime podklad `bludisko_ziak.py`.

Zamerajme sa najskôr na vytvorenie bludiska. Bludisko má dva rozmery, budeme ho uchovávať ako zoznam zoznamov. Potrebujeme kódovať štyri stavy (88):

1. Voľné miesto – označíme ho ako **'-'**,
2. Stenu – označíme ho ako **'X'**,
3. Minotaura – označíme ho ako **'M'**,
4. A miesto, kde sme už boli – označíme ho číslom `1 až N`, podľa toho koľkým krokom sme na políčko skočili.

```
def bludisko_definuj(self):
     bludisko = [['-','-','-','-','-','X'],
                 ['X','X','-','-','-','X'],
                 ['-','-','-','X','-','X'],
                 ['-','X','X','-','-','X'],
                 ['-','X','-','-','X','X'],
                 ['-','X','-','-','-','M']]
```

Bludisko môžeme aj náhodne vygenerovať. Vedy však nemáme pod dohľadom pozíciu stien.

```
def bludisko_generuj(rozmer, pocet_stien):
    bludisko = []                                    [1]
    for i in range(rozmer):
        bludisko.append(['-']*rozmer)

    for i in range(pocet_stien):                     [2]
        x = randint(0,rozmer-1)
        y = randint(0,rozmer-1)
        bludisko[x][y] = -1

    bludisko[rozmer-1][rozmer-1] = 'X'               [3]
    bludisko[0][0] = '-'                             [4]
```

[1] Najskôr vygenerujeme prázdne bludisko.
[2] Následne doň na náhodných pozíciách vygenerujeme steny, ktoré kódujeme znakom 'X'.
[3] Minotaura umiestnime do pravého dolného rohu bludiska.
[4] Vchod do bludiska musí ostať prázdny.

Posledná možnosť je načítať bludisko zo súboru.

```
def bludisko_zo_suboru():
    bludisko = []
    with open("bludisko.txt") as subor:                [5]

    for riadok in subor:
        bunky = riadok.split(",")                      [6]
        pom = []                                       [7]
        for i in bunky:
            pom = pom + [int(i)]
        bludisko.append(pom)
```

[5] V každom riadku súboru bude uložený jeden riadok bludiska. Každá bunka v riadku bude oddelená čiarkou.
[6] Načítaný riadok načítame do zoznamu. Metóda `split` vytvorí zoznam z aktuálneho riadka z hodnôt oddelených čiarkou.
[7] Po načítaní sú hodnoty sú uložené ako reťazce. Prekonvertujeme ich na čísla a uložíme do bludiska.

Aby sme videli, či sa nám bludisko naozaj vytvorilo, potrebujem si ho vypísať.

```
def zobraz():
    for riadok in bludisko:
        for bunka in riadok:
            print(f'{bunka:^3}', end="")
        print()
    print()
```

Výkladový text

Funkcia zobraz je klasickým príkladom výpisu hodnôt v tvare matice (tabuľky). Keďže bludisko obsahuje hodnoty rôznej dĺžky, je vhodné ich zarovnať smerom doprava. Na to sa používa formátovací reťazec, ktorý určuje nielen vypisovanú premennú ale aj to, koľko miest má vo výpise zaberať.

```
print(f'{b:3}', end="")
```

Zápis `{b:3}` znamená, že budeme vypisovať obsah premennej `b`, ktorá je znak a bude zaberať tri miesta. V prípade, ak reťazec má zabrať viac miest ako je jeho reálna dĺžka, tak sa výpis doplní prázdnymi medzerami.

```
>>> print(f'{"125":5}', end="")
  125
```

```
>>> print(f'{"125":10}', end="")
       125
print(f'{b:^3}', end="")
```

Zápis `{b:^3}` znamená, že budeme vypisovať obsah premennej `b`, tri miesta, vycentrovaná na stred.

Na záver nám ostáva vyriešiť srdce programu, vyhľadávací algoritmus. Algoritmus spustíme vyhľadávaním z bunky [0][0], pričom začíname prvým krokom: `hladaj_cestu(0,0,1)`.

Keďže ide o rekurzívny algoritmus potrebujeme zadefinovať triviále vetvy. Teda situácie, kedy už jednoznačne vieme určiť, či sme našli riešenie alebo že sme sa zasekli a vieme, že tadiaľ cesta nevedie. Uvažujeme o všetkých možnostiach, kedy nemá význam na aktuálnej úrovni pokračovať, a teda je možné ukončiť vykonávanie funkcie príkazom `return`.

- **Nájdenie Minotaura**. Zavedieme si premennú `ciel`, kde si poznačíme, že už máme riešenie a nemá už význam prehľadáva bludisko. Túto premennú môžme použiť ako podmienku pre jednu z triviálnych vetiev. Je potrebné nastaviť hodnotu tejto premennej na `False`. Nájdenú cestu je vhodné vypísať.

```
global ciel
if ciel == True:
    return
if bludisko[x][y] == 'M':
    print(f'[{x}][{y}] Minotaurus porazeny!')
    ciel = True
    zobraz()
    return
```

- Vkročenie do bunky, **kde je stena**.

```
elif bludisko[x][y] == 'X':
    print(f'[{x}][{y}] Stena')
    return
```

- Vkročenie do bunky, **kde sme už boli**. V bunke je poznačené číslo kroku.

```
elif bludisko[x][y] != '-':
    print(f'[{x}][{y}] Tu som uz bol')
    return
```

Ak sme prešli filtrom všetkých triviálnych podmienok je jasné, že sme sa ocitli v bunke, kde sme ešte neboli, nie je tam stena ale ešte sme nenašli Minotaura. Preto môžme rovno poznačiť, že sme na danú bunku vkročili.

```
print (f'[{x}][{y}] Krok {krok}')
bludisko[x][y] = krok
```

To znamená, že môžme pokračovať ďalším krokom. Ale v ktorom smere? Potrebujeme nadefinovať, kam a v akom poradí sa má Theseus v bludisku vybrať. Rekurzívna vetva, teda vetva, ktorá sa odkazuje na funkciu s rovnakým menom, v akom sa práve nachádzame, tiež

nebude len jedna. Budeme potrebovať pre každý z testovaných smerov jednu vetvu. Skôr ako zavoláme metódu `hladaj_cestu` skontrolujeme, či je to vzhľadom na hranice bludiska možné.

```
#doprava
if (x<len(bludisko)-1): hladaj_cestu(x+1, y, krok+1)

#hore
if (y>0): hladaj_cestu(x, y-1,krok+1)

#dolava
if (x>0): hladaj_cestu(x-1,y,krok+1)

#dolu
if (y<len(bludisko)-1): hladaj_cestu(x, y+1,krok+1)
```

V prípade, že sme sme sa pokúsili nájsť riešenie vo všetkých štyroch smeroch, ale zo všetkých sme sa už vrátili, tak opúšťame túto úroveň. Vraciame sa o krok späť a preto je vhodné po sebe zmazať stopy.

```
bludisko[x][y] = '-'
```

Úloha 5

Upravte úlohu tak, aby sa nevypísalo len jedno riešenie, ale všetky možné cesty od vstupu až k Minotaurovi.

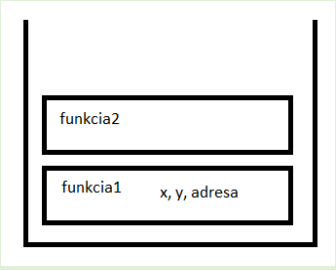
Výkladový text

Treba si len uvedomiť, že jedna z triviálnych vetiev obsahuje test, či sme už našli riešenie. Ak áno, tak v hľadaní nepokračuje. Stačí túto vetvu odstaviť a algoritmus, po nájdení prvého riešenia, vráti vykonávanie o krok späť a snaží sa preskúmať všetky ešte neprebádané smery.

Zopakujem si

Ako je to vôbec možné? Využíva sa tu princíp volania funkcií.

```
def funkcia1(x,y):
    print(x)
    funkcia2(x+y)
    print(y)
```



Keď sa zavolá `funkcia1`, tak sa v zásobníkovej pamäti, čo je dočasná pamäť slúžiaca na ukladanie hodnôt potrebných pre úspešné vykonanie danej metódy, uložia všetky lokálne premenné. V momente zavolania `funkcie2`, je potrebné aktuálnu funkciu opustiť a preto sa zapamätá miesto, odkiaľ sme odskočili do druhej funkcie. Pre `funkciu2` sa v zásobníku vytvorí nová úroveň s potrebnými hodnotami pre túto funkciu. Po skončení funkcie sa aktuálna úroveň zásobníku vymaže a pokračuje sa na zapamätanom mieste vo fukcii1.

Pri backtrackingu sa každé volanie uloží ako jedna úroveň zásobníka, a tak ak sme na danej nenašli riešenie, vieme sa vrátiť vždy o jednu úroveň nižšie.

Doplňujúce úlohy

1. Navrhnite riešenie, ktoré nájde najkratšiu cestu zo všetkých.
2. Navrhnite a zrealizujte grafickú vizualizáciu pre hľadanie cesty v bludisku.

Úloha 6

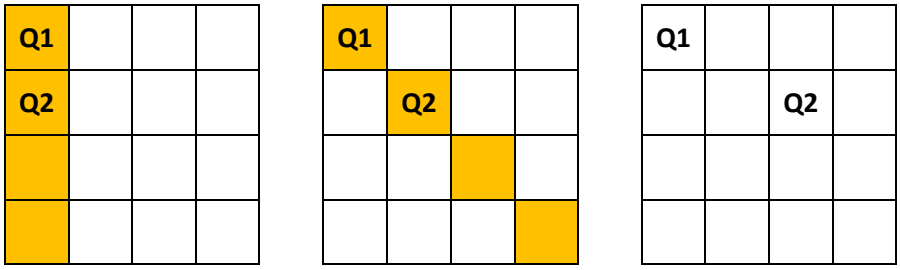
Riešme teraz úlohu na šachovnici. Ako môžeme na šachovnicu NxN uložiť N dám, pričom ani jedna z nich neohrozuje tie ostatné?

Vieme, že dáma je univerzálna figúrka, ktorá môže chodiť všetkými smermi. Môže teda útočiť horizontálne, vertikálne ako aj na diagonálach, a to o ľubovoľný počet políčok. Úlohu teda môžeme preformulovať aj tak, že hľadáme takú konfiguráciu N dám, v ktorej žiadne dve z nich sa navzájom neohrozujú a teda sa nenachádzajú v rovnakom stĺpci, riadku ale ani na diagonále (89). Vysvetlime si najskôr riešenie problému vizuálne na jednoduchej šachovnici 4x4.

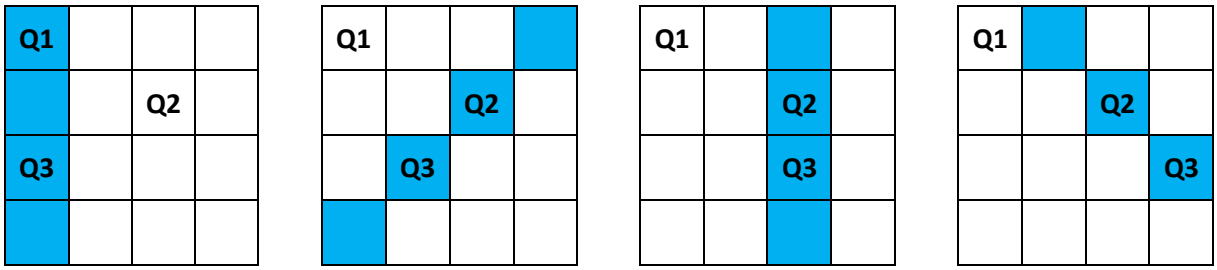
Q1			

Pre prvú dámu sme miesto našli rýchlo, umiestnili sme ju na prvé pole šachovnice.

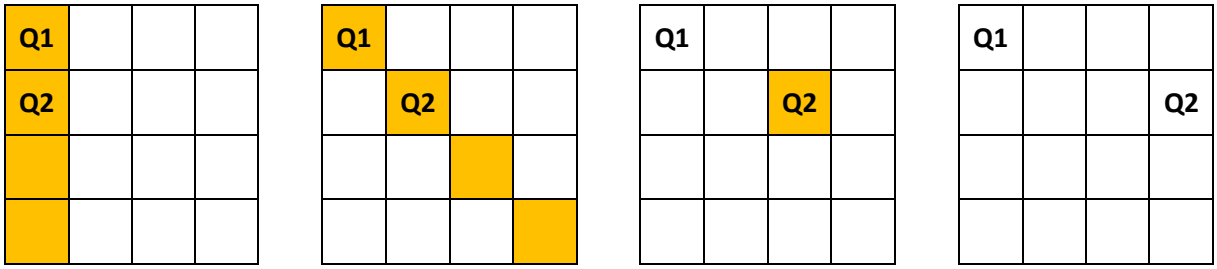
V druhom riadku sme postavili dámu Q2 na políčko v prvom stĺpci, tam ale bola ohrozená dámou Q1 vertikálne. Pretom sme Q2 posunuli o jedno políčko vpravo, ale tam nastalo ohrozenie od Q1 diagonálne. Na treťom ďalšom políčku je už dáma v bezpečí.



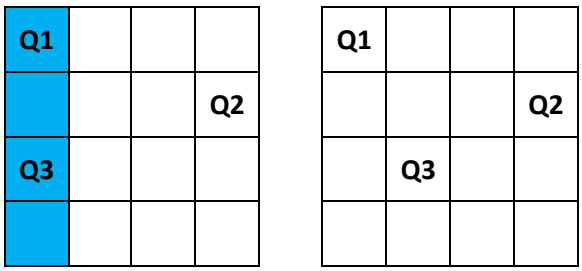
Podobne ako pri dáme dva by sme postupovali aj s treťou dámou. Môžeme si však všimnúť v aktuálnej konfigurácii nevieme pre tretiu dám nájsť miesto bez ohrozenia. Zasekli sme sa.



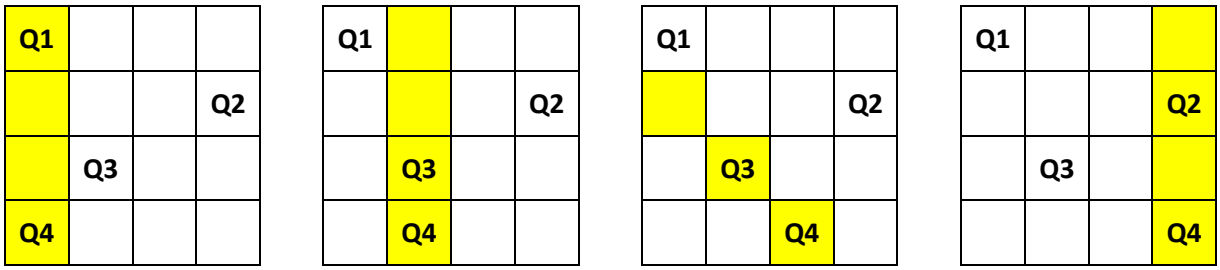
Preto sa musíme vrátiť o riadok vyššie a nájsť novú konfiguráciu pre druhú dámu.



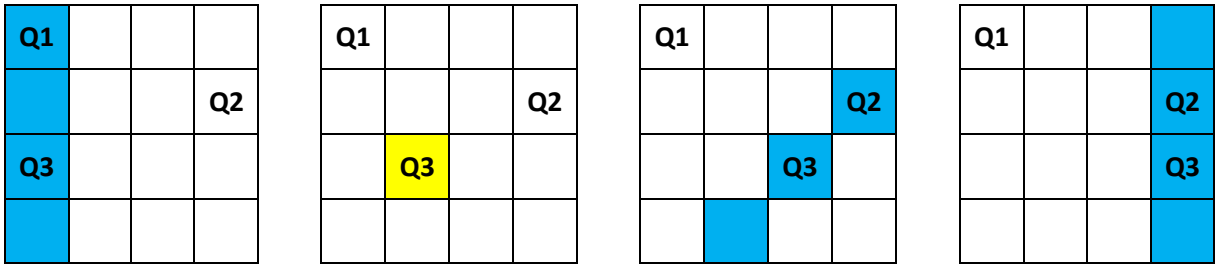
Vidíme, že v novom rozložení nájdeme miesto pre tretiu dámu už v druhom stĺpci.



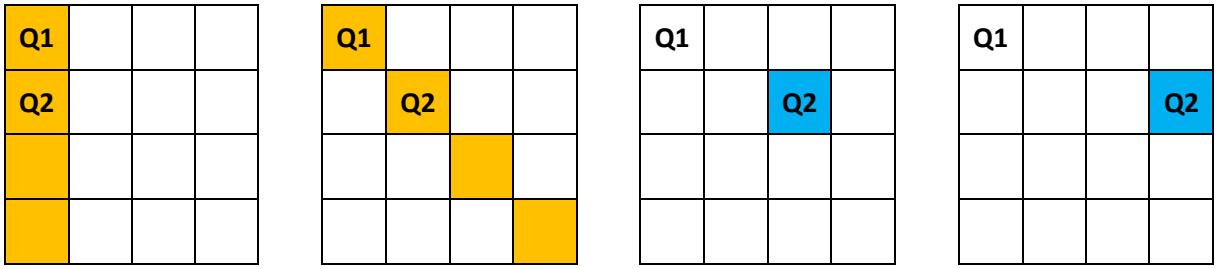
So štvrtou dámou sme sa dostali do rovnakej situácie ako pri dáme tri v prvej konfigurácii.



Preto sa musíme vrátiť o krok späť a posunúť dámu v riadku tri. Avšak pre dámu v treťom riadku, žiaľ už nie je iná možnosť a preto sa musíme vrátiť až do riadku dva.



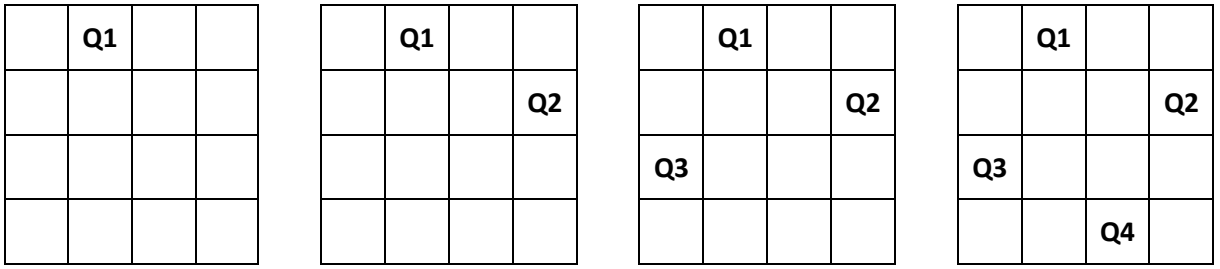
Takže v aktuálnej konfigurácii bola druhá kráľovná dvakrát ohrozená prvou dámou a zo zvyšných pozícií sme riešenie nenašli.



Otázka pre žiakov

Čo je potrebné urobiť teraz? Ktorú dámu kam posunieme?

Musíme sa vrátiť až do prvého riadka a posunúť prvú dámu o jeden stĺpec vpravo (a to nebude naposledy). Takže postupne by sme našli riešenie ktoré je uvedené nižšie.



Pozorný študent si mohol všimnúť, že sme postupným posúvaním (rekurzívna vetva) hľadali riešenie, ktoré sme v prípade, že sme sa zasekli mohli vrátiť späť (backtracking) a pokračovať

iným smerom. A to až dovtedy, kým sme nenašli konfiguráciu, kedy boli všetky dámy v bezpečí (triviálna vetva). Poďme teda programovať. Najskôr potrebujeme šachovnicu NxN. Kódovanie v našom prípade bude oveľa jednoduchšie, lebo si nepotrebujeme evidovať žiadne kroky, ako sme k riešeniu dospeli. Zaujímajú nás len výsledne pozície dám. Preto budeme len dva stavy **'0'** = dáma v políčku nie je, **'1'**= dáma v políčku je.

```
def sachovnica_generuj(rozmer):
    return [['0']*rozmer for i in range(rozmer)]
```

Zopakujme si

Na generovanie prvkov zoznamu sme použili *generátorovú notáciu zoznamu*. Štruktúra príkazu na generovanie zoznamu na základe pravidla, iterácie a filtra je nasledovná:

`list  = [pravidlo iterácia filter]`

sachovnica = [['0']*rozmer **for** i **in** range(rozmer)]

Každý prvok zoznamu sa vytvorí na základe pravidla `['0']*rozmer`. Počet prvkov určuje cyklus `for i in range(rozmer)`. Žiadny filter, ktorý by vyberal vhodnosť prvkov neuplatňujeme. Čo bude postupne obsahovať zoznam `l`?

```
str = "Ahoj 12345 Svet"
```

```
l = [x for x in string if x.isdigit()]
```

zoznam znakov, z reťazca `str`, ktoré reprezentujú čísla (digit)

```
l = [x**2 for x in range(10)]
```

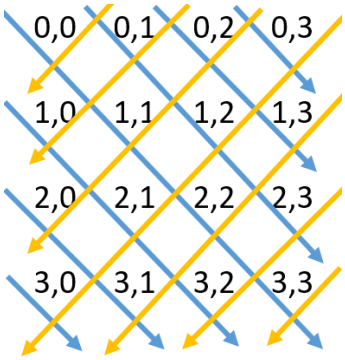
zoznam čísel, druhých mocnín z čísel od 0 po 9

```
l = [x for x in range(10) if x%2==0]
```

zoznam čísel, generovaný z čísel od 0 po 9, ale len ak sú párne

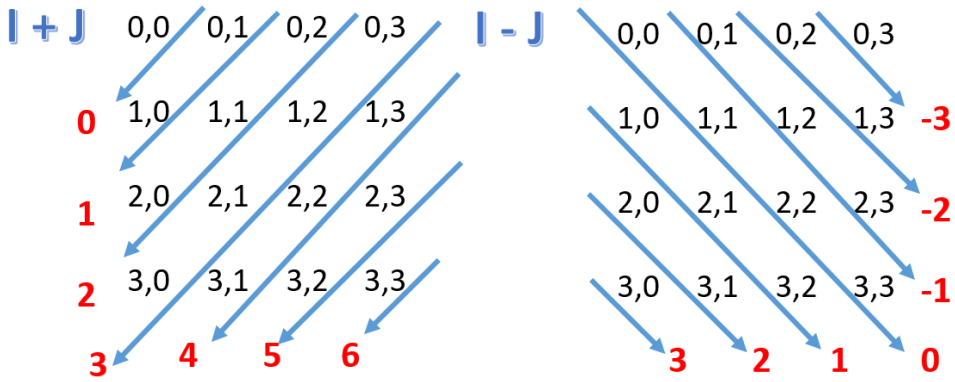
Ešte potrebujeme navrhnúť spôsob ako kódovať ohrozenia dám. Jeden z spôsobov je pre každé položenie dámy prechádzať všetky prvky v riadku, stĺpci a v diagonálach, čo nepovažujeme za efektívne riešenie. Nám totiž stačí evidovať čísla riadkov (stĺpcov), v ktorých sa už niečo nachádza. Podobne je to aj v prípade stĺpcov a diagonál.

```
riadky = []
stlpce = []
diagonala =[]
antidiagonala = []
```



Diagonál v šachovnici 4x4 je 7, v šachovnici 5x5 je 9, v šachovnici 6x6 je 11. Zovšeobecnene, v šachovnici NxN je 2*N-1 diagonál v každom smere.

Všetky prvky, ktoré sú na rovnakej vedľajšej diagonále (ideme sprava doľava) majú zaujímavú vlastnosť (pozri obrázok nižšie). Keď sčítame ich riadkový a stĺpcový index dostaneme pre všetky prvky rovnaké číslo (z intervalu 0 .. 6). Keď pre ľubovoľný prvok na hlavnej diagonále (ide zľava doprava) odčítame riadkový a stĺpcový index dostaneme tiež rovnaké číslo (z intervalu -3 .. 3).



Obrázok 81 Výpočet jednotného čísla pre prvky na diagonálach

To môžeme využiť pri zisťovaní či je dáma položená na políčko `(i,j)` v ohrození.

```
def je_bezpecne(i, j):
    if (i in riadky) or       i                    [1]
       (j in stlpce) or                            [2]
       ((i-j) in diagonala) or                     [3]
       ((i+j) in antidiagonala):                   [4]
                return False
            else:
                return True
```

[1] Je dáma na pozícii `(i,j)` ohrozená v riadku? Tzn. testujem, či sa v zozname nachádza rovnaké číslo riadka.

[2] Je dáma na pozícii `(i,j)` ohrozená v stĺpci? Tzn. testujem, či sa v zozname nachádza rovnaké číslo stĺpca.

[3] Je dáma na pozícii `(i,j)` ohrozená na hlavnej diagonále?

[4] Je dáma na pozícii `(i,j)` ohrozená na vedľajšej diagonále?

Rekurzívny algoritmus hľadania riešenia môžeme vytvoriť nasledovne.

```
def hladaj(q):                                     [5]
    global pocet_rieseni

    if q == n:                                     [6]
        pocet_rieseni += 1
        zobraz()
        return True
```

```
    for s in range(n):                                  [7]
        print(q,s) #priebežný výpis navštívených pozícií, možné vypnúť

        if (je_bezpecne(q,s)):                          [8]
            poloz(q,s)                                  [9]
            hladaj(q + 1)                               [10]
            zober(q,s)                                  [11]
```

[5] Začni kráľovnou na riadku q. Algoritmus začne volaním `hladaj(0)`

[6] Ak sú položené všetky kráľovné, vypíš riešenie, zarátaj do nájdených riešení a vráť sa hľadať ďalšie. Ak je N kráľovien, tak sa číslujú od `0` po `N-1`. Ak sa pokúšame hľadať riešenie pre kráľovnú s číslom `N`, je zrejmé, že všetky sú už položené.

[7] Vyskúšaj všetky stĺpce pre danú kráľovnú. Pre každé položenie testuj [8] až [11].

[8] Ak je kráľovná na políčku neohrozená …

[9] … polož ju tam …

[10] … a rekurzívne over, či táto pozícia neohrozuje nasledujúce dámy.

[11] Ak položenie kráľovnej neviedlo k riešeniu, zober ju, odznač aktuálne riešenie a vráť sa (backtrack) do bodu [7] a skús ďalší stĺpec.

Položenie alebo odobratie dámy spôsobí zmeny v našich testovacích zoznamoch. Vždy, keď položíme dámu, musíme si to poznačiť – pridáme číslo riadka, číslo stĺpca, číslo diagonály aj antidiagonály do príslušného zoznamu.

```
def poloz(q,s):                                         [12]
    riadky.append(q)
    stlpce.append(s)
    diagonala.append(q-s)
    antidiagonala.append(q+s)

def zober():                                            [13]
    stlpce.pop()
    riadky.pop()
    diagonala.pop()
    antidiagonala.pop()
```

[12] Aktualizuj zoznamy pridaním indexov aktuálnej pozície.

[13] Naposledy pridané pozície sú v zozname ako posledné. Pri odobratí dámy vyhodíme posledné pridané prvky pomocou metódy `pop`.

Posledné, čo potrebujeme naprogramovať je zobrazenie aktuálneho riešenia. Pozície dám si ukladáme v jednorozmernom zozname. Ak chceme vypísať rozloženie dám na šachovnici, musíme šachovnicu zrekonštruovať z aktuálnych dát.

```
def zobraz(self):
    global sachovnica
    sachovnica = sachovnica_generuj(n)                  [14]

    for r in range(n):
       sachovnica[r][stlpce[r]] = '1'                   [15]
       print(' '.join(sachovnica[r]))                   [16]
    print()
```

[14] Pre aktuálne riešenie vygenerujeme novú prázdnu šachovnicu.

[15] V každom riadku zapíšeme, na ktorej pozícii je umiestnená dáma.

[16] Všetky hodnoty v aktuálnom riadku šachovnice spojíme do jedného reťazca a vypíšeme.

Riešenie aktuálnej konfigurácie je uložené v zozname `stlpce`, ktorý nám hovorí na koľký stĺpec v danom riadku treba položiť dámu. Výpis riešenia by preto mohol byť aj jednoduchší.

```
def zobraz_stlpce(self):
    print(stlpce)
```

Spustíme našu aplikáciu.

```
def spusti(self):
            hladaj(0)

d = damy(4)
d.spusti()
```

Ďalšie úlohy na precvičenie a zamyslenie

1. Zahrajme si ešte raz šachy. Predstavte si, že máme šachovnicu ale k dispozícii len jednu figúrku, a to šachového koňa. Otázka znie: Dokážeme z pozície, na ktorú položíme figúrku koňa poskákať celú šachovnicu? Uvedomte si, že kôň skáče do L. Vypíšte šachovnicu s poradím skokov, od prvého skoku až po posledný. Poznámka: Využite základnú logiku z úlohy o bludisku, kde tiež ide o štvorcovú sieť a tiež je potrebné evidovať poradie krokov.

16. Rozdeľme sa spravodlivo

Kľúčové slová

pažravá stratégia, stratégia hrubej sila, generovanie všetkých možností, hodnotiaca funkcia, časová zložitosť

Čo sa naučíme a čo si precvičíme

- objektovo pristupovať k riešeniu problémov,
- implementovať rekurzívne algoritmy na generovanie postupností,
- aplikovať pažravé algoritmy pri riešení optimalizačných úloh,
- aplikovať algoritmy používajúce hrubú silu pri riešení optimalizačných úloh,
- uvažovať o algoritmoch, ktoré nie vždy vedú k najlepšiemu riešeniu,
- porovnávať algoritmy z hľadiska času potrebného na ich realizáciu,
- uvažovať o rôznych kritériách hodnotenia algoritmov.

Problémová situácia

Valachov Maťka a Kubka, známe večerníčkové postavičky, asi netreba nejako zvlášť predstavovať. Svojimi huncútstvami nás zabávajú už od roku 1973. V jednej časti tohto kresleného seriálu našli Maťko a Kubko poklad. Cestou domov im síce zlaté mince z truhlice povypadávali, ale ako to už v rozprávkach býva, všetko nakoniec skončilo dobre. Zo zlatých mincí valasi ukuli zlaté zvončeky pre svoje ovečky.

Poznámka na okraj

Rozprávky Maťkovi a Kubkovi nájdete aj na http://www.rozpravky.sk/seria/67/pasli-ovce-valasi-matko-a-kubko.htm [29. 5. 2019].

Predstavme si ale iný koniec tejto rozprávky. Maťko a Kubko by si chceli poklad spravodlivo rozdeliť. V truhlici mohli byť rôzne vzácne veci. Ako postupovať, aby si poklad spravodlivo rozdelili?

Rozdeľme sa dobre

Úloha 1

Dá sa poklad zo zlatých mincí spravodlivo rozdeliť medzi Maťka a Kubka? Ak nie, aké bude najspravodlivejšie rozdelenie?

Úloha 2

V poklade, ktorý Maťko a Kubko našli, neboli len mince, ale aj rôzne iné veci: zlaté prstene, strieborné príbory, diamantové náušnice a mnoho ďalšieho. Nech sa Maťko a Kubko delili akokoľvek, stále mal jeden z nich pocit, že má omnoho menej ako ten druhý. Navrhnite nejakú stratégiu, podľa ktorej by sa mal poklad medzi Maťka a Kubka rozdeliť. Overte, či vami navrhnutá stratégia nájde vždy najlepšie rozdelenie.

Výkladový text

Stratégie, pri ktorých sme predmety prideľovali podľa nejakého hodnotenia, majú niečo spoločné. Pri každej z nich postupujeme tak, že v danom okamžiku sa snažíme urobiť najlepšie rozhodnutie a dúfame, že séria týchto lokálne najlepších rozhodnutí nás privedie k celkovo najlepšiemu riešeniu.

Takéto stratégie sa označujú spoločným názvom – **pažravé stratégie** alebo pažravé algoritmy (greedy algorithm) (23).

Vo všeobecnosti pri pažravých stratégiách postupujeme nasledovne:

- rozhodnutie, ktoré v danom okamžiku spravíme, musí byť realizovateľné (napr. nemôžeme Maťkovi dať predmet z pokladu, ak sme ho už predtým dali Kubkovi),
- rozhodnutie, ktoré v danom okamžiku spravíme, musí byť zo všetkých najlepšie možné (napr. predmet pridelíme tomu, ktorý má momentálne menej),
- ak sme už nejaké lokálne rozhodnutie spravili, nezrušíme ho (napr. ak sme Maťkovi priradili diamantové náušnice, už mu ich nezoberieme).

Pažravá stratégia sa používa na riešenie optimalizačných úloh, pri ktorých z množstva riešení chceme nájsť to najlepšie.

Tak, ako sme sa presvedčili vyššie, táto stratégia nie vždy vedie k optimálnemu riešeniu. Napriek tomu tieto stratégie nezamietneme ako zlé. Neskôr sa k nim ešte vrátime.

Úloha 3

Z pažravých algoritmov, ktoré ste navrhli v úlohe 2 si jeden vyberte a implementujte ho. Definujte triedu `DobreRozdelenie()` Vytvorte metódu `__dobre_rozdel()`, ktorá pre ceny predmetov nájde zodpovedajúce rozdelenie predmetov do dvoch skupín. Definujte si vhodné `property`, aby objekt vedel nájdene rozdelenie vrátiť. Riešenie uložte do súboru `pazravy_algoritmus.py`.

Rozdeľme sa najlepšie

V prípadoch ako je tento, keď nevieme nájsť nejakú stratégiu, ktorá by nás vždy doviedla k optimálnemu (najlepšiemu) riešeniu, môžeme postupovať aj inak. Preskúmajme všetky možné rozdelenia predmetov a z nich vyberme to najlepšie. Táto stratégia nám zaručí, že sa dopracujeme k najlepšiemu riešeniu, lebo zo všetkých možných vyberieme to najlepšie. Alebo inak povedané, neexistuje lepšie riešenie ako to, ktoré týmto spôsobom nájdeme.

Výkladový text

Stratégia, pri ktorej systematicky prechádzame všetky možné riešenia a vyberieme z nich to najlepšie sa nazýva **stratégia riešenia hrubou silou** (Brute-force search). Táto stratégia nám vždy umožní nájsť najlepšie riešenie. (22)

Stratégia riešenia hrubou silou nájde uplatnenie tam, kde nie sme schopní nájsť nejakú inú, ľahko implementovateľnú stratégiu.

Ak sa rozhodneme pôvodný problém rozdelenia pokladu riešiť hrubou silou, musíme nájsť spôsob, ako vygenerovať všetky možné rozdelenia predmetov z pokladu. Zamyslime sa, ako by sme jednoducho mohli reprezentovať jednotlivé rozdelenia predmetov.

Pre jednoduchosť uvažujme, že skupiny do ktorých delíme predmety z pokladu označíme 0 a 1.

O každom predmete rozhodujeme, či ho dáme do skupiny 0 alebo do skupiny 1. Po každom rozdelení každý predmet patrí buď do skupiny 0 alebo do skupiny 1. Rozdelenie tovarov teda vieme reprezentovať ako postupnosť 0 a 1, ktorá má rovnakú dĺžku ako pôvodný zoznam cien tovarov.. Napríklad, ak predmety s cenami:

```
ceny = [5, 1, 3, 4, 2]
```

rozdelíme do skupín nasledovne:

`skupina0 = [5, 3]` `skupina1 = [1, 4, 2]`

vieme toto rozdelenie zapísať ako `[0, 1, 0, 1, 1]`.

A platí to aj naopak. Pre každú vhodne dlhú postupnosť 0 a 1 vieme vytvoriť rozdelenie tovarov do skupín.

Ak sa nám podarí vygenerovať všetky postupnosti 0 a 1 danej dĺžky, vieme z nich vytvoriť všetky rozdelenia predmetov a z nich vybrať to najlepšie.

Napr. pre tri predmety potrebujeme vygenerovať tieto postupnosti:

0, 0, 0
0, 0, 1
0, 1, 0
0, 1, 1
1, 0, 0
1, 0, 1
1, 1, 0
1, 1, 1

Úloha 4

Definujte triedu `NajlepsieRozdelenie()` a v nej metódu `__generuj_rozdelenia()`, ktorá bude generovať postupnosti z čísiel 0 a 1. Dĺžka generovaných postupností je rovnaká ako počet predmetov v poklade. Pre kontrolu správnosti môžete vygenerované postupnosti priebežne vypisovať. Riešenie uložte do súboru `hruba_sila.py`.

Pomôcka: Uvažujte nasledovne. Problém vygenerovať všetky postupnosti sa dá rozložiť na dva podproblémy:

- vygenerujme prvé číslo postupnosti,
  - toto je triviálna úloha, prvé číslo môže byť len 0 alebo 1
- vygenerujme zvyšné čísla postupnosti,

- toto je identický problém ako pôvodný problém, rozdiel je len v tom, že generujeme postupnosť o 1 kratšiu, postupovať teda môžeme rovnako ako pri riešení pôvodného problému.

Každá inštancia metódy `__generuj_rozdelenia()` by mala vedieť, ako aktuálna časť generovanej postupnosti vyzerá a či už nie je vygenerovaná celá postupnosť. Generovaný zoznam si môžu funkcie posielať ako parameter. Či už je vygenerovaná celá postupnosť vieme podľa toho, že počet jej prvkov je rovnaký ako počet predmetov v zozname predmetov.

Úloha 5

Vytvorte metódu `__cena_rozdelenia(self, rozdelenie)`, ktorá pre zadané rozdelenie vráti jeho cenu. Premyslite si, ako definujete hodnotiace kritérium pre cenu rozdelenia. Metódu zavolajte pre každé vygenerované rozdelenie. Cenu rozdelenia môžete pre každé vygenerované rozdelenie priebežne vypisovať.

Definovali sme metódu `__generuj_rozdelenia()`, ktorá generuje všetky rozdelenia a metódu `__cena_rozdelenia()`, ktorá vie dané rozdelenie ohodnotiť. Aby sme našli najlepšie rozdelenie, musíme vyhodnotiť každé z nich a zapamätať si to najlepšie.

Úloha 6

Zrušte kontrolné výpisy v metóde `__generuj_rozdelenia()` a metódu upravte tak, aby si v nejakej inštančnej premennej priebežne uchovávala to najlepšie rozdelenie, ktoré do danej chvíle vygenerovala.

Definujte metódu `__get_rozdelenie()`, ktorá po otestovaní všetkých rozdelení vráti najlepšie nájdené rozdelenie. Uvedomte si, ako reprezentujeme rozdelenie pre potreby triedy (postupnosť 0 a 1) a ako ho chceme vyjadriť pre používateľa (dvojica zoznamov cien predmetov z pokladu).

Definujte si vhodné `property`, aby objekt vedel nájdene rozdelenie vrátiť.

Dobre alebo najlepšie?

Implementovali sme dve rôzne stratégie rozdeľovania predmetov pokladu medzi Maťka a Kubka. Vzájomne ich porovnajme. Cieľom je zistiť, či pažravá stratégia je natoľko zlá, že v porovnaní so stratégiou riešenia hrubou silou o nej nemusíme uvažovať.

Úloha 7

Vytvorte hodnotiacu funkciu `cena_rozdelenia()`, ktorá ohodnotí zadané rozdelenie. Funkciu uložte do súboru `delenie_pokladu.py`.

Úloha 8

V súbore `delenie_pokladu.py` zrealizujte malý test. Vygenerujte náhodný zoznam cien predmetov. Porovnajte rozdelenie, ktoré získame z pažravého algoritmu s rozdelením, ktoré získame algoritmu využívajúceho hrubú silu. Začnite s krátkym zoznamom (napr. 5 prvkový) a postupne ho rozširujte. Skúmajte, vlastnosti oboch stratégií. Vysvetlite svoje zistenia.

Výkladový text

Porovnávať algoritmy môžeme podľa rôznych kritérií. Jedným z nich je časová zložitosť algoritmu (24). Túto hodnotu zrejme nemôžeme merať v časových jednotkách. Ak by sme algoritmus spustili na rôzne výkonných počítačoch, dostali by sme rôzne hodnoty. Porovnávať algoritmy na základe týchto výsledkov by nebolo správne. Navyše doba trvania našich algoritmov záležala aj od vstupu. Čím väčší vstup, tým dlhšie vykonávanie algoritmu trvalo.

V informatike sa časová zložitosť vyjadruje ako funkcia veľkosti vstupu. Zjednodušene povedané, časová zložitosť predstavuje počet elementárnych operácii potrebných pre spracovanie vstupu danej veľkosti. Ak sa zložitosť algoritmov líši len o konštantu, budeme ich považovať za rovnako časovo zložité a túto konštantu zanedbáme.

Príklad:

Ak algoritmus1 vykoná pre vstup dĺžky n 2n krokov a algoritmus2 10n krokov, budeme ich považovať za rovnako časovo zložité. „Pomalosť“ algoritmu1 vieme kompenzovať 5 krát rýchlejším počítačom. Časovú zložitosť oboch algoritmov označíme ako *O*(n).

Pri časovej zložitosti nás zaujíma najmä zložitosť v najhoršom prípade.

Ak sa pozrieme na naše dva algoritmy, tak:

- Pažravý algoritmus postupne prechádzal cenami predmetov (predmetov je n) a pri každej cene rozhodol, či predmet zaradí do jednej alebo druhej skupiny. Jeho časovú zložitosť by sme vyjadrili ako *O*(n).
- Algoritmus využívajúci hrubú silu vygeneroval všetky možné postupnosti dĺžky n (je ich 2^n) a pre každú vypočítal jej ohodnotenie (na to potreboval n krokov). Celkovo by sme teda časovú zložitosť tohto algoritmu mohli označiť ako $O(2^n * n)$.

Poznámka na okraj

V informatike si pri výpočtoch zložitostí algoritmov môžeme dovoliť aj väčšie zanedbania ako len konštanty. Pozrime sa na priebeh funkcie $f(x) = 2^x * x$. Pre malé hodnoty x má hodnota x v celom súčine nezanedbateľný vplyv na hodnotu funkcie:

x	f(x)	vplyv x v súčine
x = 1	2	1/2
x = 2	8	1/4
x = 3	24	1/8
x = 4	64	1/16

Ak sa pozrieme na väčšie hodnoty x, zistíme že vplyv hodnoty x na celý súčin sa dramaticky znižuje.

x = 10	10240	1/1024
x = 20	20971520	1/1048576
x = 30	32212254720	1/1073741824

Vplyv hodnoty x na celý súčin je už taký malý, že ho môžeme zanedbať. Bez ujmy na hodnotení zložitosti algoritmu môžeme teda namiesto zložitosti $O(2^n * n)$ uvažovať $O(2^n)$.

Záver

V tejto kapitole sme hľadali riešenie problému, ako rozdeliť predmety do dvoch skupín tak, aby v oboch skupinách bol tovar s celkovo rovnakou hodnotou. Zistili sme, že úloha nemá vždy riešenie (neexistuje také rozdelenie). Navrhli sme dve rôzne stratégie, ako problém riešiť. Použitím hrubej sily nájdeme najlepšie rozdelenie, ale môže to trvať veľmi dlho. Použitím pažravej stratégie nájdeme riešenie veľmi rýchlo ale cenou za rýchlosť je fakt, že nie vždy nájdeme to najlepšie riešenie. Je preto na zvážení, či sa uspokojiť s nie celkom najlepším riešením ktoré vieme nájsť veľmi rýchlo alebo trvať na najlepším riešení aj za cenu veľkého časového zdržania.

Čo sme sa naučili

- využívať objektový prístup k riešeniu problémov,
- implementovať rekurzívne algoritmy na generovanie postupností,
- aplikovať rôzne stratégie pri riešení optimalizačných problémov: pažravú stratégiu a stratégiu využívajúcu hrubú silu,
- uvažovať o algoritmoch, ktoré nie vždy vedú k najlepšiemu riešeniu,
- porovnávať algoritmy z hľadiska času potrebného na ich realizáciu,
- uvažovať o rôznych kritériách hodnotenia algoritmov.

Ďalšie úlohy na precvičenie a zamyslenie

1. Požiadavka vygenerovať nejakú postupnosť z 0 a 1 nie je pri programovaní zriedkavá záležitosť. Modul `itertools` obsahuje nástroje pre generovanie rôznych postupností. Ak potrebujeme generovať postupnosti z 0 a 1 zadanej dĺžky, dosiahneme to nasledovne:

   ```
   import itertools

   dlzka = 4
   for postupnost in itertools.product((0, 1), repeat=dlzka):
       print(postupnost)
   ```

   Upravte triedu `NajlepsieRozdelenie()` tak, aby namiesto rekurzívneho generovania postupností využila funkciu modulu `itertools`.

2. Problém rozdelenia pokladu sme riešili pre dvoch valachov, pre Maťka a Kubka. Uvažujme, že poklad si chce rozdeliť viac valachov.
   - Ako by sa zmenilo hodnotiace kritérium pre ohodnotenie rozdelenia?
   - Ako by sa zmenili triedy `DobreRozdelenie()` a `NajlepsieRozdelenie()`?

   Upravte triedy `DobreRozdelenie()` a `NajlepsieRozdelenie()` tak, aby našli dobré, resp. najlepšie rozdelenie pre zadaný počet valachov.

3. Pri platbe v obchode by sme chceli platiť použitím čo najmenšieho počtu platidiel. Vytvorte funkciu, ktorá pre zadanú cenu nákupu vráti počty jednotlivých platidiel.

   Akú stratégiu riešenia použijete a prečo?

4. V aplikácii sa heslá neukladajú do súboru priamo, ale ukladá sa ich odtlačok. Odtlačok sa vypočíta nasledovne:

   ```
   import hashlib

   heslo = 'super tajne'
   odtlacok = hashlib.sha256(heslo.encode()).hexdigest()
   print(odtlacok)
   ```

   ```
   892e558675f7cf5a15a85ec9263a26d4382cbb832b4263f81a9068b8a0847d51
   ```

   Zabudli sme heslo, ale pamätáme si, že heslo malo 4 alebo 5 znakov a použili sme v ňom len písmená malej anglickej abecedy. Zo súborov aplikácie vieme, že odtlačok nášho hesla je:

   ```
   9aeab6643d73b7e2d210471a3e82112b074271226e5d1b17411f3413c925097b
   ```

   Vytvorte program, ktorý zistí naše zabudnuté heslo.

17. Vzťahy v sociálnej sieti

Kľúčové slová

graf ako abstraktná údajová štruktúra, implementácia grafu, prehľadávanie grafu do hĺbky, prehľadávanie grafu do šírky, komponenty súvislosti

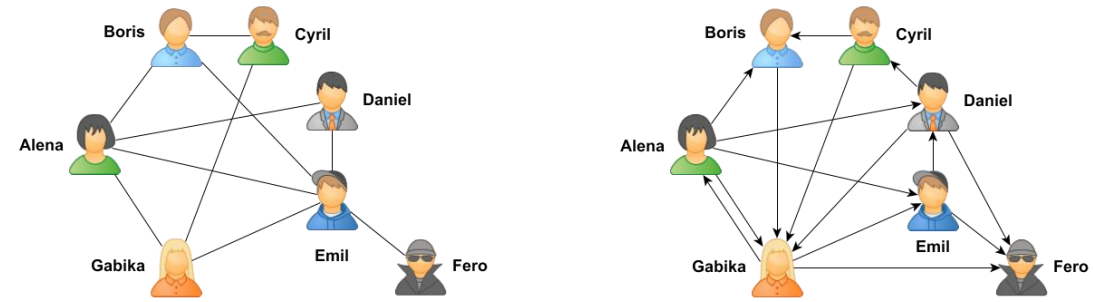
Čo sa naučíme a čo si precvičíme

- modelovať problémy z reálneho života pomocou grafu,
- uvažovať o rôznych implementáciách údajovej štruktúry graf,
- navrhnúť a implementovať triedu reprezentujúcu graf,
- skúmať vlastnosti grafu,
- prehľadávať graf do hĺbky a do šírky.

Problémová situácia

Väčšina aktívnych používateľov internetu má účet aj na niektorej zo sociálnej sietí (komunikujú s inými členmi globálnej komunity, zdieľajú informácie a zážitky, publikujú svoje názory a hodnotenia, vyhľadávajú rady a pracovné príležitosti, vytvárajú záujmové skupiny a pod.)

Pozrime sa na niekoľko vybraných používateľov Facebooku a Twittera (pozri obrázok nižšie):



Obrázok 82 Príklady sociálnych sietí

Facebook: Najviac priateľov má Emil, najmenej Fero. Alena a Gabika sú priateľky. Boris a Daniel nie sú priatelia.

Twitter: Najviac sledovateľov má Gabika, najmenej Cyril. O Fera sa zaujímajú až 3 používatelia. Fero však nesleduje nikoho. Alena a Gabika sa sledujú navzájom.

Poznámka na okraj

Facebook uvádza v štatistikách približne 2,7 miliardy aktívnych používateľov. Sociálna sieť Twitter „len“ 330 miliónov.
Zdroj: https://en.wikipedia.org/wiki/List_of_social_platforms_by_monthly_active_users [30. 6. 2020]

Otázka
Poznáte aj iné sociálne siete?

Výkladový text

Na znázornenie vzťahov medzi členmi sociálnej siete sme práve použili **graf**. Osoby predstavujú **vrcholy** grafu. Čiary, ktoré ich spájajú, sú **hrany**. Ak sú vrcholy spojené hranou, nazývajú sa **susedné**.

Facebookový vzťah „byť priateľom“ *(friend)* je obojstranný. Hrany v prvom grafe preto nemajú orientáciu. Takýto graf voláme **neorientovaný**.

Ak je používateľ Twitteru sledovateľom *(follower)* iného používateľa, ten ho sledovať nemusí. Hrany v druhom grafe sú preto orientované smerom k sledovanej osobe. Takýto graf voláme **orientovaný**.

Hrana, ktorá vychádza z a aj vchádza do toho istého vrcholu, sa nazýva **slučka**. Dva vrcholy by mohli byť spojené aj viacerými hranami. V našich úlohách však slučky ani **násobné hrany** uvažovať nebudeme, nepotrebujeme ich .

Poznámka na okraj

Skúmaním grafov sa zaoberá **teória grafov**. Patrí k najviac aplikovaným matematickým disciplínam. Poznatky z teórie grafov sú dôležité nielen pre informatiku. Uplatňujú sa aj v prírodných vedách, doprave, elektrotechnike, ekonómii, lingvistike, sociológii či pedagogike (91), (92), (93).

Graf je **abstraktný matematický model**, pomocou ktorého možno vyjadriť štruktúru a vlastnosti navonok úplne rozdielnych reálnych systémov. A práve v tom je jeho sila.

V praxi sa stretávame s mnohými inými situáciami, ktoré je výhodné modelovať pomocou grafu. Vždy v nich vystupuje množina objektov, pričom medzi niektorými dvojicami z tejto množiny existuje fyzický alebo logický vzťah, napr.:

- medzi mestami (križovatkami, zástavkami, stanicami, letiskami) vedú cesty, železničné trate, letecké spojenia,
- súčiastky v elektrospotrebiči spájajú vodiče,
- medzi atómami v molekule existujú chemické väzby,
- zariadenia v telekomunikačnej sieti sú prepojené káblami,
- neuróny v mozgu spájajú synapsie,
- tímy na športovom turnaji súperia vo vzájomných zápasoch,
- spolužiaci v triede zdieľajú spoločné záujmy,
- z jednej webovej stránky existuje odkaz na inú webovú stránku,
- pri plánovaní úloh je potrebné dodržať časové nadväznosti,
- medzi bankovými účtami sa realizujú prevody,

- jeden dokument cituje iný dokument,
- medzi živočíšnymi druhmi existuje vzťah predátor – korisť atď.

Výkladový text

O vrcholoch a hranách grafu môžu byť k dispozícii rôzne informácie. Napr. mestá (*vrcholy*) majú svoje názvy, cestné spojenia medzi mestami (*hrany*) majú rôzne dĺžky. Ak sú údaje o vrcholoch alebo hranách grafu podstatné pre algoritmus, ktorý graf spracúva, musíme pracovať **s ohodnoteným grafom** (orientovaným alebo neorientovaným). Čísla či reťazce priradené vrcholom alebo hranám potom nazývame **ohodnotenia**.

Poznámka na okraj

Ak nepovieme inak, máme pri pomenovaní ohodnotený graf na mysli ohodnotenia hrán. S ohodnoteným grafom budeme pracovať neskôr, v osobitnej kapitole 18 s názvom *Najkratšia cesta*.

V tejto kapitole budeme podrobnejšie skúmať iba **neorientovaný neohodnotený graf.** V našom prípade takýmto grafom modelujeme sociálnu sieť typu Facebook. Zistíme:

- či v nej existujú takí dvaja používatelia, ktorí o sebe určite nevedia (nie sú priateľmi a nemôžu sa o sebe dozvedieť ani prostredníctvom priateľov svojich priateľov, pretože sú členmi navzájom oddelených skupín),
- koľkí používatelia musia zdieľať príspevok, aby sa o ňom dozvedela iná konkrétna osoba (tá nemusí byť priateľom autora príspevku).

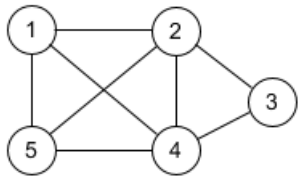
Hľadajme riešenie

Údajová štruktúra graf

Ak chceme programovať **grafové algoritmy**, musíme si najprv premyslieť, akým spôsobom bude vhodné graf reprezentovať v pamäti počítača. Údaje o vrcholoch a hranách musia byť uložené tak, aby bolo spracovanie grafu efektívne.

Abstraktnú údajovú štruktúru graf je možné implementovať rôznym spôsobom. Porovnajme dva najviac používané prístupy (94), (95):

Neorientovaný graf



Graf na obrázku **má 5 vrcholov**: 1, 2, 3, 4, 5 a **8 hrán**: (1,2), (1,4), (1,5), (2,3), (2,4), (2,5), (3, 4), (4,5).

Matica susednosti

	1	2	3	4	5
1	F	T	F	T	T
2	T	F	T	T	T
3	F	T	F	T	F
4	T	T	T	F	T
5	T	T	F	T	F

Matica má **5 riadkov a 5 stĺpcov**. Ak sú vrcholy *v* a *w* spojené hranou, prvok v riadku *v* a stĺpci *w* má hodnotu `True`, inak `False`.

Zoznamy susedov

1	2,4,5
2	1,3,4,5
3	2,4
4	1,2,3,5
5	1,2,4

Pre každý z 5 vrcholov si zapamätáme zoznam všetkých jeho susedných vrcholov.

Všimnime si: V grafe máme hranu (2, 4). V matici susednosti (označme ju `A`) majú preto prvky `A[2][4]` a `A[4][2]` hodnotu `True`. V zozname susedov vrcholu `2` vidíme vrchol `4`. Rovnako aj v zozname susedov vrcholu `4` vidíme vrchol `2`. Namiesto logických hodnôt sme v matici susednosti mohli ukladať celé čísla 0 a 1.

Úloha 1

V programoch musíme na grafe realizovať rôzne operácie (zisťovať jeho vlastnosti alebo meniť jeho štruktúru). V nasledujúcej tabuľke označte, ktorá z implementácii je pre príslušnú operáciu výhodnejšia:

Operácia na grafe	***Matica susednosti***	***Zoznamy susedov***
pridanie/odstránenie hrany		
overenie existencie hrany		
získanie zoznamu susedov konkrétneho vrcholu		
zistenie počtu susedov konkrétneho vrcholu		

Poznámka na okraj

Počet susedov vrcholu sa nazýva aj **stupeň vrcholu**.

Výkladový text

Výhodou matice susednosti je možnosť overiť existenciu hrany ihneď, keďže k príslušnému prvku máme priamy prístup pomocou indexov. V prípade implementácie so zoznamami susedov je nutné prehľadávať zoznam susedov jedného z vrcholov hrany.

Vďaka implementácii so zoznamami susedov máme zase pre každý vrchol ihneď k dispozícii zoznam jeho susedných vrcholov. V matici susednosti musíme kontrolovať, ktoré z prvkov v riadku pre príslušný vrchol majú hodnotu `True`.

Pridávanie alebo odstraňovanie hrán je rýchlejšie pri maticovej implementácii. Veľkou nevýhodou matice sú však vysoké nároky na pamäť. V praxi sa často stretávame s grafmi, ktoré sú **riedke** – majú veľký počet vrcholov, ktoré sú spojené relatívne malým počtom hrán. V prípade riedkeho grafu je tak väčšina prvkov matice nastavená na hodnotu `False`.

Úloha 2

Vytvorte a otestujte triedu `Graf` pre prácu s neorientovaným neohodnoteným grafom. Vrcholy v grafe označujte celými číslami 1, 2, ..., n. Triedu `Graf` implementuje **pomocou zoznamov susedov**.

Poznámka na okraj

Mená osôb nebudú pre nás podstatné, indexy vieme ľahko transformovať na reťazce s využitím pomocného *zoznamu* alebo *slovníka*.

Dohodneme sa, že pri vytváraní nového grafu určíme počet jeho vrcholov (ten sa už nebude neskôr meniť). Následne do grafu pridáme postupne všetky jeho hrany.

Graf s 5 vrcholmi a 8 hranami (zo strany 5) vytvoríme takto:

```
g = Graf(5)
g.pridaj_hranu(1, 2)
g.pridaj_hranu(1, 4)
g.pridaj_hranu(1, 5)
g.pridaj_hranu(2, 3)
g.pridaj_hranu(2, 4)
g.pridaj_hranu(2, 5)
g.pridaj_hranu(3, 4)
g.pridaj_hranu(4, 5)
print(g)                                                    #[1]
print(repr(g))                                              #[2]
print(g.existuje_hrana(1,2))
```

Výstup testovacieho skriptu:

```
pocet vrcholov: 5
pocet hran: 8
hrany: (1, 2) (1, 4) (1, 5) (2, 1) (2, 3) (2, 4) (2, 5) (3, 2) (3,
4) (4, 1) (4, 2) (4, 3) (4, 5) (5, 1) (5, 2) (5, 4)
Graf([[], [2, 4, 5], [1, 3, 4, 5], [2, 4], [1, 2, 3, 5], [1, 2, 4]])
True
```

Príkazom výpisu v riadku [1] vypíšeme stav objektu `g` na konzolu. Metóda `__str__` vracia počet vrcholov, počet hrán a zoznam všetkých hrán vo forme trojriadkového reťazca.

Príkazom výpisu v riadku [2] poskytneme alternatívny pohľad na graf v podobe zodpovedajúcej jeho implementácii. Naprogramovaním metódy `__repr__` zobrazíme zoznam zoznamov susedov jednotlivých vrcholov.

V Pythone má prvý prvok zoznamu index 0. Vrcholy grafu však číslujeme kladnými celými číslami. V záujme prehľadnosti zápisu môžeme preto nultý prvok hlavného zoznamu v našom riešení ignorovať.

Poznámka na okraj

V ukážke zdrojového kódu vyššie vidíme metódu na *pridávanie hrany* a *overenie existencie hrany*. Trieda reprezentujúca graf by mala poskytovať aj ďalšie metódy, napr. na získanie

- zoznamu susedov konkrétneho vrcholu,
- zoznamu všetkých hrán,
- počtu vrcholov a
- počtu hrán.

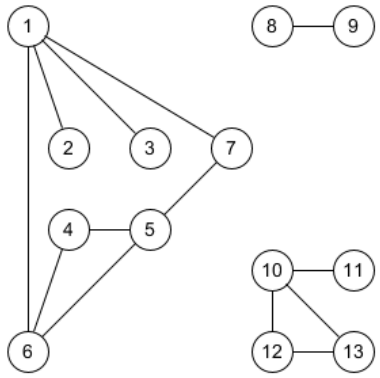
Úloha 3

Upravte riešenie predchádzajúcej úlohy tak, aby bolo možné graf načítavať z textového súboru.

Vstupný súbor opisujúci graf môže mať v prvých dvoch riadkoch uvedený počet vrcholov a počet hrán grafu. V ďalších riadkoch môžu byť potom systematicky vymenované jednotlivé hrany.

Prehľadávanie grafu do hĺbky

Uvažujme neorientovaný graf reprezentujúci sociálnu sieť s 13 používateľmi:



graf.txt:

```
13
13
1 2
1 3
1 6
1 7
4 5
4 6
5 6
5 7
8 9
10 11
10 12
10 13
12 13
```

Ak chceme zistiť, či v nej existujú také skupiny používateľov, ktoré o sebe navzájom nevedia, máme vlastne overiť, **či je graf reprezentujúci túto sociálnu sieť súvislý**. Čo to znamená?

Výkladový text

Cestou nazývame postupnosť vrcholov, ktoré sú spojené hranami. Postupnosť 1, 6, 4, 5 predstavuje jednu z ciest z vrcholu 1 do vrcholu 5. Jej dĺžka je 3 (počet hrán, po ktorých sme prešli).

Graf je **súvislý**, ak z každého vrcholu grafu existuje cesta do každého iného vrcholu.

Graf na obrázku zjavne súvislý nie je, prečo?

Zvolíme jeden z vrcholov (napr. 1), z ktorého začneme graf systematicky prehľadávať. Ak sa nám podarí navštíviť všetky vrcholy grafu, znamená to, že zo začiatočného vrcholu existujú cesty do všetkých ostatných vrcholov a graf je súvislý.

Poznámka na okraj

Ak existujú cesty z vrcholu 1 do ostatných vrcholov, tak zrejme existujú aj cesty z každého iného vrcholu do ostatných vrcholov (stačí sa vrátiť do vrcholu 1 a pokračovať v ceste k cieľovému vrcholu).

Prehľadávanie grafu chceme realizovať rozumne. Aby sme v grafe zbytočne neblúdili (nevracali sa opakovane do vrcholov, v ktorých sme už boli), **o každom vrchole si poznačíme,**

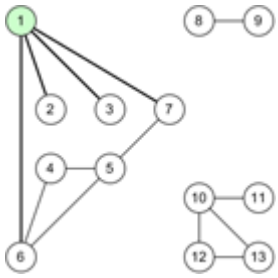
že sme ho práve navštívili. Z každého vrcholu, do ktorého prídeme, budeme pokračovať navštívením niektorého z jeho susedných vrcholov, v ktorom sme ešte neboli (ak taký existuje). Takýto algoritmus nazývame **prehľadávanie do hĺbky** (v angl. *Depth First Search*, *DFS*) a môžeme ho jednoducho implementovať s využitím rekurzie:

```
class DfsSuvislost:
    def __init__(self, g):
        self.__navstivene = [False]*(g.pocet_vrcholov()+1)
        self.__dfs(g, 1)

    def __dfs(self, g, v):
        self.__navstivene[v] = True                          #[1]
        for w in g.susedia(v):                               #[2]
            if not self.__navstivene[w]:                     #[3]
                self.__dfs(g, w)

    def je_suvisly(self):
         return not (False in self.__navstivene[1:])
```

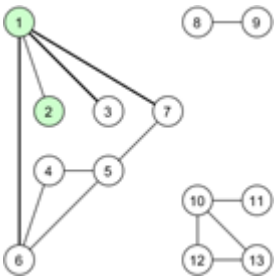
Pri inicializácii objektu typu `DfsSuvislost` vytvoríme pomocný zoznam s hodnotami `False`, ktorý budeme potrebovať na evidenciu navštívených vrcholov. Následne spustíme prehľadávanie z vrcholu 1. V metóde `__dfs`:

[1] si poznačíme, že sme práve navštívili vrchol *v*,
[2] v zozname všetkých susedov vrcholu *v* hľadáme taký susedný vrchol, v ktorom sme zatiaľ neboli,
[3] ak taký vrchol *w* existuje, spustíme rekurzívne prehľadávanie z tohto vrcholu (zavoláme metódu `__dfs` s parametrom *w*).

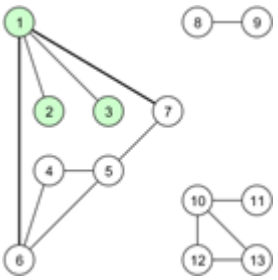
Jedinou metódou, ktorá tvorí verejné rozhranie triedy `DfsSuvislost`, je metóda `je_suvisly`, ktorú zavoláme, aby sme zistili, ako prehľadávanie do hĺbky dopadlo.



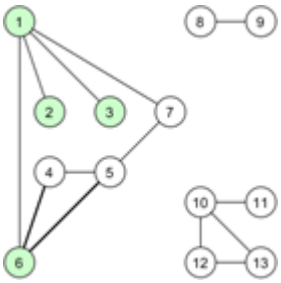
Vrchol 1 má 4 susedné, zatiaľ nenavštívené vrcholy.



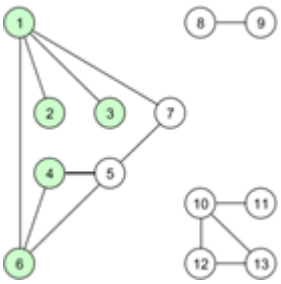
Vyberieme prvý z nich, a označíme ho ako navštívený. Vrchol 2 už nemá ďalších susedov. Pokračujeme preto navštívením vrcholu 3.



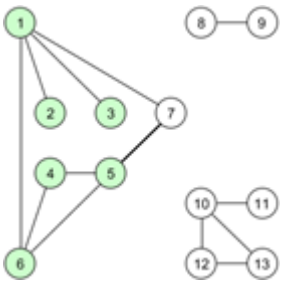
Vrchol 3 tiež nemá ďalších susedov. Pokračujeme preto ďalším nenavštíveným susedom vrcholu 1, ktorým je vrchol 6.



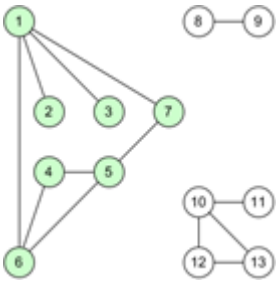
Vrchol 6 označíme ako navštívený. Má dva susedné vrcholy, v ktorých sme ešte neboli, vrcholy 4 a 5.



Pokračujeme do vrcholu 4. Ten má jeden susedný vrchol, ktorý zatiaľ nie je navštívený.



Navštívime vrchol 5. S týmto vrcholom susedia tri vrcholy, nenavštívili sme ale len jeden, vrchol 7.



Vrchol 7 už nemá žiadnych nenavštívených susedov. Rekurzívne volanie metódy `__dfs` spustené z vrcholu 7 preto skončí. Rovnako postupne skončia aj rekurzívne volania z predošlých krokov prehľadávania.

Do vrcholov 8, 9, 10, 11, 12 a 13 sme sa pri prehľadávaní nedostali (nie sú susedom žiadneho z navštívených vrcholov), zostali nenavštívené („biele“).

Poznámka na okraj

Vrcholy grafu sme *pri prehľadávaná do hĺbky* navštívili v tomto poradí:

1, 2, 3, 6, 4, 5, 7

V grafe sme sa dostávali stále ďalej od počiatočného vrcholu (išli sme „do hĺbky“ grafu). Keď sa už pokračovať nedalo, vrátili sme sa späť a navštívili ďalšieho nenavštíveného suseda. S rovnakou myšlienkou sme sa stretli už v kapitole 15 s názvom *Backtracking* o prehľadávaní s návratom.

Úloha 4

Na vlastných (súvislých aj nesúvislých) grafoch zrealizujte prehľadávanie do hĺbky. Vypíšte, či je graf súvislý.

Úloha 5

Keď graf nie je súvislý, musí v ňom byť aspoň jeden taký vrchol, do ktorého sme sa zo začiatočného vrcholu pri prehľadávaní nedostali (v pomocnom zozname `__navstivene` bude mať príslušný prvok hodnotu `False`). V grafe z podkapitoly o prehľadávaní do hĺbky vidíme takýchto vrcholov viac. Náš graf obsahuje 3 samostatné **komponenty súvislosti**:

K_1: vrcholy 1, 2, 3, 4, 5, 6, 7 a hrany medzi nimi,
K_2: vrcholy 8, 9 a hrana medzi nimi,
K_3: vrcholy 10, 11, 12, 12 a hrany medzi nimi.

Z jedného komponentu súvislosti sa nemáme ako dostať do iného. Inak povedané: *„osoby nachádzajúce sa v komponente K_i nevedia o osobách v komponente K_j" (pre $i \neq j$)*

Navrhnite, ako by ste zistili, **z koľkých komponentov súvislosti sa graf skladá**.

Otázka

Koľko komponentov súvislosti by tvorilo graf s 13 vrcholmi, v ktorom by neexistovali žiadne hrany?

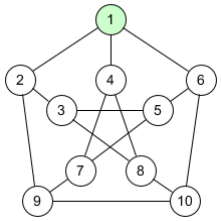
Prehľadávanie grafu do šírky

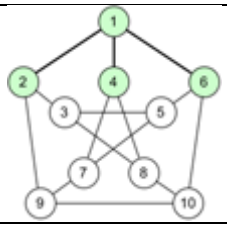
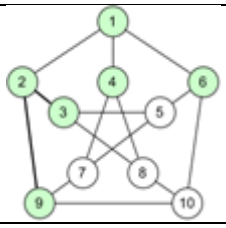
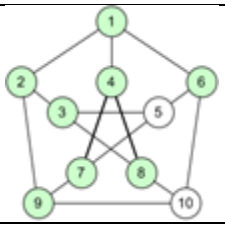
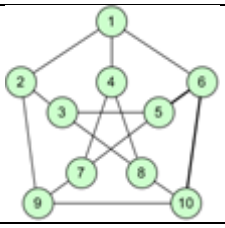
Zamyslime sa ešte raz nad tým, v akom poradí sme navštevovali vrcholy grafu pri prehľadávaní do hĺbky:

Začali sme vo vrchole 1. Potom sme navštívili jedného z jeho zatiaľ navštívených susedov (v našom prípade vrchol 2). Ostatné susedné vrcholy (3, 6 a 7) vrcholu 1 sa spracúvali až po skončení rekurzívneho volania metódy `__dfs` spusteného z vrcholu 2 (keďže vrchol 2 nemá ďalších susedov, toto volanie skončilo ihneď). Rovnako sme postupovali pre každý iný práve navštívený vrchol. Vďaka mechanizmu rekurzie sme si nemuseli osobitne zapamätať všetkých susedov aktuálneho vrcholu. Po návrate riadenia do pôvodného volania sa vo vykonávaní metódy pokračovalo automaticky ďalej (overovaním stavu ďalšieho susedného vrcholu).

Niekedy je však výhodnejšie preskúmať **najprv celé okolie prvého vrcholu** a **navštíviť všetkých jeho susedov**. Potom navštívime všetkých nenavštívených susedov týchto susedov atď. V grafe tak budeme od počiatočného vrcholu postupovať viac „do šírky" (vzďaľovať sa od neho rovnomerne všetkými smermi). Preto sa tento algoritmus nazýva **prehľadávanie do šírky** (v angl. *Breadth First Search*, *BFS*).

Algoritmus BFS aplikujeme na konkrétnom neorientovanom grafe:

<table>
<tr><td> 
Rad: 1</td><td colspan="3">Prehľadávanie začneme vo vrchole 1. Označíme ho ako navštívený a pridáme ho do zoznamu vrcholov, ktorých okolie je potrebné preskúmať.

Vrcholy uložené v pomocnom zozname budeme spracúvať v rovnakom poradí, v akom ich do zoznamu vkladáme. Takúto údajovú štruktúru voláme rad (v angl. queue). Nové vrcholy pridávame na koniec radu. Vrcholy na spracovanie vyberáme z jeho začiatku.</td></tr>
<tr><td>  </td><td>  </td><td>  </td><td>  </td></tr>
<tr><td>Z radu vyberieme vrchol 1 a preskúmame jeho okolie – navštívime všetkých jeho susedov. Vložíme ich tiež na koniec radu.</td><td>Z radu vyberieme vrchol 2 a preskúmame jeho okolie – neboli sme v dvoch jeho susedných vrcholoch (3, 9), navštívime ich a pridáme do radu.</td><td>Z radu vyberieme vrchol 4 a preskúmame jeho okolie – neboli sme v dvoch jeho susedných vrcholoch (7, 8), navštívime ich a pridáme do radu.</td><td>Z radu vyberieme vrchol 6 a preskúmame jeho okolie – neboli sme v dvoch jeho susedných vrcholoch (5, 10), navštívime ich a pridáme do radu.</td></tr>
<tr><td>Rad: 2 4 6</td><td>Rad: 4 6 3 9</td><td>Rad: 6 3 9 7 8</td><td>Rad: 3 9 7 8 5 10</td></tr>
</table>

Prehľadávanie pokračuje ďalej: Vrchol 3 vyberieme z radu, nemá nenavštívených susedov. Vrchol 9 vyberieme z radu, nemá nenavštívených susedov. Podobne vrchol 7, 8, 5. Ako posledný zostane v rade vrchol 10, nemá nenavštívených susedov. Rad zostane prázdny, prehľadávanie skončí.

Poznámka na okraj

Neorientovaný graf z posledného príkladu sa nazýva *Petersenov graf*. Má 10 vrcholov a 15 hrán. Každý vrchol má stupeň 3 (má práve tri susedné vrcholy).

Výkladový text

Rad sa často nazýva aj pamäť typu **FIFO** (z angl. *First In First Out*). Prvok, ktorý do radu pridáme ako prvý, spracujeme ako prvý. Ide o analógiu radu z bežného života. Keď stojíme napr. v rade na obed, dostaneme obed v tom poradí, v akom sme prišli (predbiehanie nepripúšťame).

Úloha 6

Prehľadajte (ručne na papieri):

a) graf z podkapitoly o prehľadávaní do šírky aplikovaním algoritmu DFS,
b) graf z podkapitoly o prehľadávaná do hĺbky aplikovaním algoritmu BFS.

Aj prehľadávanie do šírky by sme mohli využiť na overovanie súvislosti grafu. My sa však sústredíme na iný problém:

Graf, ktorý chceme spracúvať my, reprezentuje sociálnu sieť. Ak budeme vedieť, či existuje cesta od jednej osoby k inej a koľko hrán ju tvorí, získame odpoveď na druhú otázku z úvodu kapitoly: **Koľkí používatelia musia zdieľať príspevok zvolenej osoby, aby sa o ňom dozvedela iná osoba?** Ako prvý zdieľa so svojimi priateľmi (susednými vrcholmi) príspevok sám jeho autor.

Ak si pre každý vrchol zapamätáme, z ktorého vrcholu sme sa doň dostali, na záver ľahko nájdeme najkratšiu cestu z počiatočného vrcholu do ktoréhokoľvek iného vrcholu. Dokonca budeme vedieť aj jej dĺžku.

Poznámka na okraj

V prípade neorientovaného grafu rozumieme najkratšou cestou takú, ktorá obsahuje najmenší počet hrán.

Úloha 7

Preštudujte implementáciu triedy `BfsCesty` uvedenú nižšie.

a) Vyhľadajte v zdrojovom kóde riadok, v ktorom sa pre práve navštívený vrchol zaznamená jeho predchodca na najkratšej ceste.

b) Vysvetlite, prečo sa po zavolaní metódy `dlzka_cesty_do` s parametrom *v* (cieľový vrchol cesty) dozvieme správny výsledok.

c) Aký bude výstup testovacieho skriptu napr. pre Petersenov graf?

Algoritmus BFS použijeme s cieľom zistiť existenciu cesty a jej dĺžku. Jeho implementáciu opäť oddelíme od triedy reprezentujúcej graf:

```
class BfsCesty:
    def __init__(self, g, s):
        self.__start = s
        # inicializácia pomocných zoznamov
        self.__navstivene = [False]*(g.pocet_vrcholov()+1)
        self.__p = [-1]*(g.pocet_vrcholov() + 1)
        self.__d = [-1]*(g.pocet_vrcholov() + 1)
        self.__bfs(g, s)

    def __bfs(self, g, s):
        # vytvoríme prázdny rad
        rad = Rad()
        # počiatočný vrchol označíme ako navštívený
        self.__navstivene[s] = True
        self.__d[s] = 0
```

```
        # najprv budeme skúmať susedné vrcholy vrcholu s
        # počiatočný vrchol s preto pridáme do radu
        rad.pridaj_na_koniec(s)
        while not rad.je_prazdny():
            # z radu vyberieme ďalší vrchol
            v = rad.odstran_zo_zaciatku()
            # preskúmame okolie vrcholu v
            for w in g.susedia(v):
                if not self.__navstivene[w]:
                    self.__navstivene[w] = True
                    self.__p[w] = v
                    self.__d[w] = self.__d[v] + 1
                    rad.pridaj_na_koniec(w)

    def existuje_cesta_do(self, v):
        return self.__navstivene[v]

    def cesta_do(self, v):
        if not self.existuje_cesta_do(v):
            return None
        cesta = [v]
        x = v
        while x != self.__start:
            x = self.__p[x]
            cesta.append(x)
        cesta.reverse()
        return cesta

    def dlzka_cesty_do(self, v):
        return self.__d[v]

# testovací skript
g = Graf('graf.txt')
print(g)
bfs = BfsCesty(g,1)
print(bfs.existuje_cesta_do(10))
print(bfs.cesta_do(10))
print(bfs.dlzka_cesty_do(10))
```

Poznámka na okraj

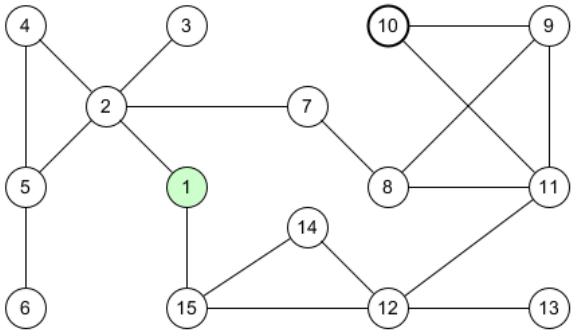
Rad môže byť implementovaný rôzne. V triede `Rad` v module *rad.py* sme ho naprogramovali s využitím zoznamu. Názvy metód korešpondujú s ich významom.

Pri rekonštruovaní cesty, po ktorej sme sa do cieľového vrcholu dostali, postupujeme od cieľového vrcholu `v` k počiatočnému vrcholu `__start`. Zoznam `cesta` preto po skončení cyklu obrátime.

Triedu `BfsCesty` z Úlohy 7 obsahuje súbor *prehladavanie_do_sirky.py*

Úloha 8

Pripravte vstupný textový súbor opisujúci graf na obrázku (pozri obrázok nižšie) a aplikujte naň algoritmus BFS. Graf reprezentuje vybranú časť sociálnej siete. Ak osoba 1 publikuje príspevok (zdieľa ho so svojimi priateľmi), koľkí ďalší používatelia ho nutne musia zdieľať, aby si ho mohla prečítať osoba 10?



Obrázok 83 Graf reprezentujúci časť sociálnej siete

Čo sme sa naučili

- Graf je abstraktný matematický model. V programe ho môžeme realizovať rôznym spôsobom. Naše rozhodnutie závisí aj od toho, s akými grafmi chceme pracovať a aký grafový algoritmus chceme naprogramovať. Najčastejšie sa používa implementácia grafu pomocou zoznamov susedov.
- Ak sa nám počas prehľadávania grafu podarí navštíviť všetky jeho vrcholy, graf je súvislý. Nesúvislý graf je tvorený viacerými komponentmi súvislosti.
- Prehľadávať graf do hĺbky znamená systematicky pokračovať z aktuálneho vrcholu do niektorého jeho nenavštíveného suseda a to až dovtedy, kým nenavštívime vrchol, v ktorého všetkých susedoch sme už boli. Nasleduje návrat späť a pokračovanie prehľadávania navštívením ďalšieho suseda predošlého vrcholu (ak taký v grafe existuje). Algoritmus DFS je možné ľahko implementovať ako rekurzívnu metódu.
- Pri prehľadávaní grafu do šírky (algoritmus BFS) navštívime vždy najprv všetkých susedov aktuálneho (práve navštíveného) vrcholu a až potom ich susedov. Na uloženie poradia, v akom máme vrcholy spracúvať, používame pri prehľadávaní do šírky údajovú štruktúru rad. Ak si pre každý navštívený vrchol poznačíme, z ktorého vrcholu sme doň prišli, ľahko získame cestu z počiatočného vrcholu do ktoréhokoľvek iného dostupného vrcholu. V neorientovanom grafe ide o cestu, ktorá má najmenší počet hrán a je preto najkratšia.

Ďalšie úlohy na precvičenie a zamyslenie

1. Doplňte do triedy reprezentujúcej graf metódu, ktorá vráti stupeň zadaného vrcholu. Ak metódu zavoláme bez parametra, vráti maximálny stupeň vrcholu v grafe. Ak budeme vedieť stupeň vrcholu, ľahko zistíme, ktorý z používateľov sociálnej siete má najviac alebo najmenej priateľov a pod.

2. V predošlých úlohách sme graf implementovali ako zoznam zoznamov susedov alebo ako maticu susednosti. Pre niektoré algoritmy môže byť výhodná implementácia grafu v podobe **zoznamu všetkých jeho hrán**. Upravte triedu `Graf` tak, aby bola implementovaná práve týmto spôsobom a otestujte ju.

3. Dal by sa pri implementácii triedy `Graf` využiť aj slovník (asociatívne pole)?

4. Prehľadávanie do hĺbky sme implementovali s využitím rekurzie. V niektorých prípadoch môže nastať situácia, že počet rekurzívnych volaní je väčší ako maximálna hĺbka rekurzie dovolená Pythonom (vyskúšajte v konzole príkaz `sys.getrecursionlimit()`). Vedeli by sme algoritmus DFS naprogramovať aj bez spoliehania sa na mechanizmus rekurzie?

18. Najkratšia cesta

Kľúčové slová

ohodnotený graf, najkratšia cesta, Dijkstrov algoritmus, dynamické programovanie

Čo sa naučíme a čo si precvičíme

- modelovať problémy z reálneho života pomocou grafu,
- implementovať údajovú štruktúru ohodnotený graf,
- aplikovať Dijkstrov algoritmus na hľadanie najkratšej cesty,
- porozumieť princípu dynamického programovania ako ďalšej užitočnej stratégii riešenia problémov v programovaní.

Problémová situácia

Problém najkratšej cesty patrí ku klasickým problémom teórie grafov. V praxi majú algoritmy, ktoré ho riešia, široké využitie. Typický príkladom sú optimalizačné problémy v rôznych dopravných a komunikačných sieťach, napr. hľadanie najkratšej alebo najlacnejšej cesty z mesta X do mesta Y, smerovanie paketov medzi uzlami siete Internet a pod.

Otázka

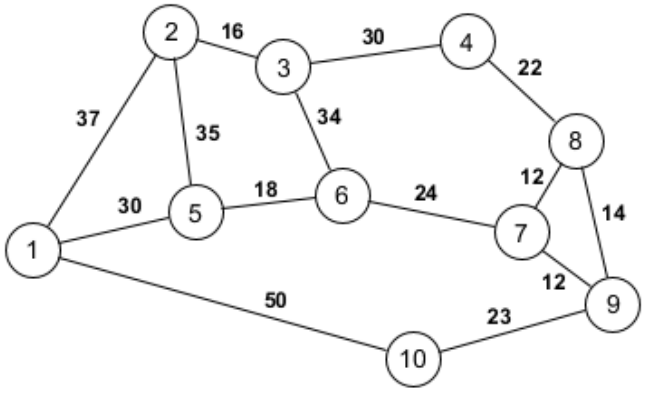
Vyhľadajte cestu medzi dvoma mestami pomocou navigačnej funkcie mapovej služby Google Mapy (napr. z Nitry do Bojníc).

Koľko ciest vám služba odporučila? Optimálna by mala byť jedna z možných ciest alebo nie?

Aj my budeme cestovať z jedného mesta do druhého. Najprv však chceme vedieť, ktorá z možných ciest je najkratšia. Situáciu si priblížime nakreslením **ohodnoteného grafu** (pozri obrázok nižšie). Vrcholy sú mestá, do ktorých môžeme cestovať (očíslujeme ich). Ohodnotenia hrán predstavujú dĺžku cestných úsekov medzi mestami v kilometroch. Všetky cestné úseky sú obojsmerné (graf bude preto neorientovaný):

Poznámka na okraj

O ohodnotených grafoch sme už hovorili aj v predchádzajúcej kapitole venovanej prehľadávaniu grafov pomocou algoritmov DFS a BFS.



Obrázok 84 Ohodnotený graf reprezentujúci cestnú sieť medzi mestami

Výkladový text

V neohodnotenom grafe predstavuje najkratšiu cestu medzi dvoma vrcholmi **cesta s minimálnym počtom hrán**. V predchádzajúcej kapitole sme ju našli pomocou algoritmu prehľadávania do šírky (BFS).

V ohodnotených grafoch ale rozumieme najkratšou cestou takú cestu, na ktorej je **minimálny súčet ohodnotení hrán**. Tu nám už algoritmus BFS nepomôže. **V ohodnotenom grafe najkratšia cesta nemusí byť zároveň cestou s najmenším počtom hrán.**

Ako ohodnotenia hrán sa najčastejšie používajú **kladné čísla**. Niekedy má význam použiť aj záporné číslo (môžeme ním reprezentovať napr. stratu pri neúspešnej transakcii na burze). Ak však pripustíme aj záporné ohodnotenia, nesmie v grafe existovať **záporný cyklus** (súčet ohodnotení hrán takého cyklu je totiž záporný). V grafe so záporným cyklom nemá zmysel najkratšiu cestu vôbec hľadať. Ak je totiž možné dostať sa z počiatočného vrcholu do niektorého vrcholu záporného cyklu, potom ku ktorejkoľvek „najkratšej“ ceste vždy nájdeme cestu ešte kratšiu. Stačí prejsť po hranách záporného cyklu dookola toľkokrát, aby súčet ohodnotení dostatočne klesol.

Otázka

Náš graf obsahuje viac cyklov, napr.:

1, 2, 3, 6, 5, 1
7, 8, 9, 7
3, 4, 8, 7, 6, 3

Viete vymyslieť príklad grafu s jedným *záporným* cyklom?

Hľadajme riešenie

Skôr ako sa pustíme do riešenia problému najkratšej cesty, musíme si pripraviť **triedu pre prácu s ohodnoteným neorientovaným grafom**.

Implementácia ohodnoteného grafu

V tejto kapitole budeme ohodnotený graf implementovať ako **maticu vzdialeností** medzi dvojicami miest. Pre každú dvojicu miest si v matici poznačíme, či medzi nimi existuje cesta (zapamätáme si jej dĺžku v km). Ak dve mestá nie sú spojené priamou cestou, na príslušnej pozícii bude hodnota -1. Keďže sme vrcholy očíslovali prirodzenými číslami, nultý riadok a nultý stĺpec matice nebudeme používať. Zoznam zoznamov (t. j. maticu vzdialeností) vytvoríme s jedným riadkom a stĺpcom navyše, aby sme jej prvky indexovali v súlade s grafickým znázornením.

Poznámka na okraj

Ohodnotený graf sme mohli implementovať aj inak. V prípade zoznamov susedov by sme si okrem susedného vrcholu (koncového vrcholu hrany) museli pamätať aj jej ohodnotenie. Prvkami zoznamu susedov by teda boli dvojice.

Úloha 1

Znázornite ohodnotený graf z úvodu kapitoly ako maticu vzdialeností uloženú v pamäti počítača.

Náš vzorový graf má 10 vrcholov. Medzi niektorými dvojicami priame cestné úseky neexistujú, medzi niektorými áno. Poznáme ich dĺžky (ohodnotenia hrán). V riadku s indexom 0 a v stĺpci s indexom 0, s ktorými pracovať nechceme, budú natrvalo zapísané hodnoty -1:

Matica vzdialeností `D`:

	0	1	2	3	4	5	6	7	8	9	10
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	-1	-1	**37**	-1	-1	**30**	-1	-1	-1	-1	**50**
2	-1	**37**	-1	**16**	-1	**35**	-1	-1	-1	-1	-1
3	-1	-1	**16**	-1	**30**	-1	**34**	-1	-1	-1	-1
4	-1	-1	-1	**30**	-1	-1	-1	-1	**22**	-1	-1
5	-1	**30**	**35**	-1	-1	-1	**18**	-1	-1	-1	-1
6	-1	-1	-1	**34**	-1	**18**	-1	**24**	-1	-1	-1
7	-1	-1	-1	-1	-1	-1	**24**	-1	**12**	**12**	-1
8	-1	-1	-1	-1	**22**	-1	-1	**12**	-1	**14**	-1
9	-1	-1	-1	-1	-1	-1	-1	**12**	**14**	-1	**23**
10	-1	**50**	-1	-1	-1	-1	-1	-1	-1	**23**	-1

Úloha 2

Dokončite implementáciu triedy reprezentujúcej ohodnotený neorientovaný graf a otestujte ju vytvorením vzorového grafu:

```
class OhodnotenyGraf:
    '''Trieda reprezentujúca neorientovaný ohodnotený graf.'''
    def __init__(self, n):
        '''Inicializuje atribúty grafu.'''
        # počet vrcholov
        self.__n = n
        # počet hrán
        self.__m = 0
        # matica vzdialeností (zatiaľ neexistuje žiadna hrana)
        self.__D = [[-1] * (self.__n + 1)
                    for i in range(self.__n + 1)]           #[1]

    def pridaj_hranu(self, u, v, h):
        '''Pridá hranu uv s ohodnotením h.'''
        pass

    def susedia(self, v):                                   #[2]
        '''Vráti zoznam susedov vrcholu v.'''
        return [w for w in range(self.__n+1)
                if self.__D[v][w] != -1]

    def pocet_vrcholov(self):
        '''Vráti počet vrcholov grafu.'''
        return self.__n

    def existuje_vrchol(self, v):
        '''Overí existenciu vrcholu.'''
        return v <= self.__n

    def vrcholy(self):
        '''Vráti zoznam všetkých vrcholov grafu.'''
        return [v+1 for v in range(self.__n)]

    def pocet_hran(self):
        '''Vráti počet hrán grafu.'''
        return self.__m

    def existuje_hrana(self, u, v):
        '''Overí existenciu hrany.'''
        return self.__D[u][v] != -1

    def hodnota_hrany(self, u, v):
        '''Vráti hodnotu hrany uv.'''
        pass


    def hrany(self):
        '''Vráti zoznam všetkých hrán grafu.'''
        return [(u, v, self.hodnota_hrany(u,v))             #[3]
                for u in range(self.__n+1)
```

```
                for v in range(u+1, self.__n+1)
                if self.__D[u][v] != -1]

    def __str__(self):
        '''Vráti reťazec so stavom objektu.'''
        pass

    def __repr__(self):
        return f'Graf({self.__D})'
```

[1] Matica vzdialeností je štvorcová. Má rovnaký počet riadkov ako stĺpcov. Pre každý prvok s indexom 0, 1, 2, ..., n vytvoríme riadok matice s prvkami inicializovanými na hodnotu -1. Nultý riadok a stĺpec sú navyše, nebudeme ich používať.

[2] Keď máme vrátiť zoznam všetkých susedov vrcholu *v*, zaujímajú nás v príslušnom riadku matice len tie prvky, ktorých hodnota je rôzna od -1 (teda tie mestá *v*, do ktorých vedie priama cesta z vrcholu *u*).

[3] Pri vytváraní zoznamu všetkých hrán pre jednotlivé dvojice vrcholov zisťujeme, či hrana existuje (ak áno, jej hodnota je rôzna od -1). Do výsledného zoznamu zapisujeme pre každú hranu trojicu čísel *(u, v, h)*: koncové vrcholy hrany a jej ohodnotenie. Systematicky generujeme a overujeme tieto dvojice vrcholov:
1,2 1,3 ... 1,n
2,3 2,4 ... 2,n
...
n-1,n

Na testovacie účely nám postačí implementácia ohodnoteného grafu bez načítavania grafu zo súboru.

Dijkstrov algoritmus

Najkratšiu cestu medzi dvoma vrcholmi grafu s hranami ohodnotenými nezápornými číslami nájdeme pomocou algoritmu, ktorého autorom je **E. W. Dijkstra** (92), (93), (96).

Poznámka na okraj

Na hľadanie najkratšej cesty medzi dvoma vrcholmi alebo všetkými dvojicami vrcholov v ohodnotenom grafe existujú rôzne algoritmy. Zväčša fungujú rovnako pre orientované aj neorientované grafy.

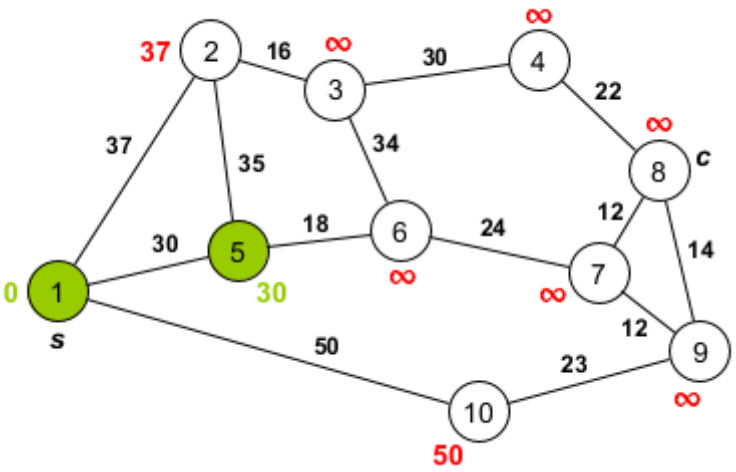
Ktorý z algoritmov sa pri riešení konkrétneho problému použije závisí aj od toho, čo o danom grafe a ohodnoteniach jeho hrán vieme.

Výkladový text

Dijkstrov algoritmus postupne nájde najkratšie cesty **z počiatočného vrcholu *s* do všetkých ostatných vrcholov**, teda aj do cieľového vrcholu *c*. Je založený na jednoduchých myšlienkach:

Dĺžka najkratšej cesty z vrcholu *s* do toho istého vrcholu *s* sa zrejme rovná 0. Ak sú vrcholy *s* a *x* spojené hranou a táto hrana má spomedzi všetkých hrán vychádzajúcich z vrcholu *s* najmenšie ohodnotenie, tak práve táto hrana nutne predstavuje najkratšiu cestu z vrcholu *s* do vrcholu *x*. Ohodnotenia hrán sú totiž nezáporné čísla. Každá iná cesta z *s* do *x* vedúca cez nejaké ďalšie vrcholy by mohla byť len dlhšia (nanajvýš rovná, v prípade nulového ohodnotenia hrany).

Pozrime sa na konkrétny príklad:



Obrázok 85 Hľadanie najkratšej cesty v ohodnotenom grafe

Niektorým vrcholom grafu sme priradili číslo – ide o dĺžku najkratšej cesty z počiatočného vrcholu *s* do príslušného vrcholu, ktorú sme doteraz našli. Hodnota ∞ znamená, že o dĺžke cesty do tohto vrcholu zatiaľ nevieme nič.

Poznámka na okraj

V programe v jazyku Python symbol ∞ nahradíme dostatočne veľkou konštantou. Môžeme využiť napr. hodnotu `sys.maxsize` definovanú v module `sys`.

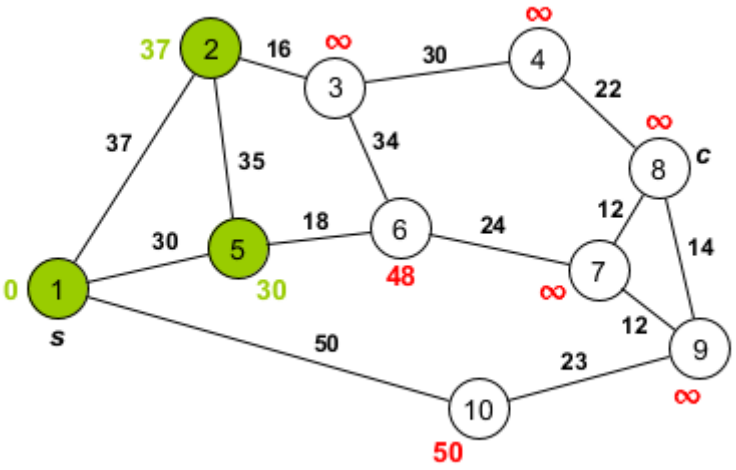
Cesta z vrcholu *s* do vrcholu *s* má dĺžku 0. Ohodnotenie vrcholu *s* je už trvalé, preto sme vrchol vyfarbili zelenou farbou.

Vrchol *s* má 3 susedné vrcholy, do ktorých vedú hrany s dĺžkami 37, 30 a 50. Hrana s dĺžkou 30 je najkratšia, preto je to určite aj dĺžka najkratšej cesty z vrcholu s do vrcholu 5. Ohodnotenie vrcholu 5 sa už meniť nemôže, je trvalé. Cesty do vrcholov 2 a 10 by sa mohli dať v priebehu vykonávania algoritmu ešte skrátiť (používame preto červenú farbu).

Výkladový text

Ak *s, ..., x, ..., c* je najkratšia cesta z vrcholu *s* do vrcholu *c*, tak zrejme aj *s, ..., x* musí byť najkratšia cesta z vrcholu *s* do vrcholu *x*. Inak by bolo možné cestu *s, ..., x, ..., c* ešte skrátiť.

Pokračujme v našom príklade ďalej:



Obrázok 86 Hľadanie najkratšej cesty v ohodnotenom grafe – pokračovanie

V predošlom kroku sme objavili najkratšiu cestu do vrcholu 5. Jej dĺžka je 30. Skontrolujeme všetky susedné vrcholy vrcholu 5, ktoré ešte nemajú trvalé ohodnotenie a overíme, či sa cesta do nich nedá skrátiť práve prechodom cez vrchol 5.

Výpočet pre susedný vrchol 2: **30** + 35 = 75, čo je viac ako 37. Ohodnotenie vrcholu 2 preto meniť nebudeme. Cez vrchol 5 sa cesta do vrcholu 2 skrátiť nedá.

Výpočet pre susedný vrchol 6: **30** + 18 = **48**, čo je menej ako „nekonečno“. Ohodnotenie vrcholu sa preto zmení na 48. Cez vrchol 5 sa do vrcholu 6 vieme z počiatočného vrcholu *s* dostať po ceste s dĺžkou 48.

Ďalším vrcholom s trvalým ohodnotením sa stal vrchol 2. Spomedzi všetkých vrcholov grafu bez trvalého ohodnotenia (vrcholy 2, 3, 4, 6, 7, 8, 9, 10) má práve vrchol 2 najmenšie ohodnotenie (**37**). Najkratšia cesta z vrcholu *s* do vrcholu 2 má určite dĺžku 37.

Overili sme, či sa cesty do jednotlivých vrcholov nedajú skrátiť prechodom cez vrchol 5. Rovnaký postup teraz zopakujeme pre vrchol 2, keďže sme ho práve pridali medzi vrcholy, do ktorých najkratšiu cestu už poznáme.

Úloha 3

Pokračujte vo vykonávaní Dijkstrovho algoritmu zopakovaním vyššie opísaného postupu pre vrchol *u=2* a následne pre ďalšie vrcholy. V cykle opakujte 3 činnosti:

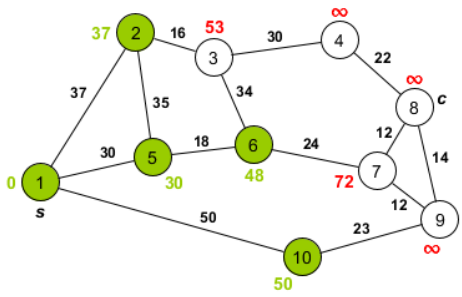
1. Vezmite vrchol *u*, ktorému ste práve priradili trvalé ohodnotenie.
2. Pre všetky vrcholy, do ktorých vedie z vrcholu *u* hrana overte, či nemožno cestu do nich skrátiť cez vrchol *u*.
3. Nájdite ďalší vrchol *u* – teda ten, ktorý má aktuálne najmenšiu dočasnú hodnotu a vyhláste ju za trvalú.

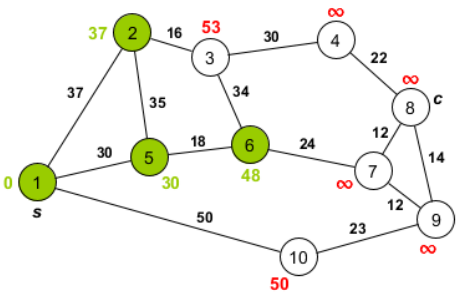
Cyklus opakujeme dovtedy, kým nebudú mať všetky vrcholy trvalé ohodnotenie (budeme poznať najkratšie cesty do každého z nich).

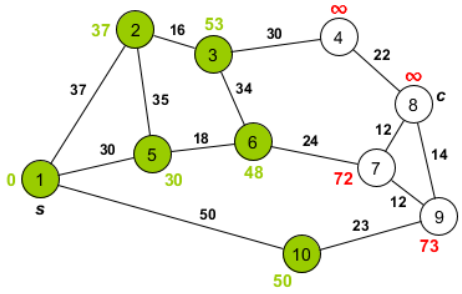
Poznámka na okraj

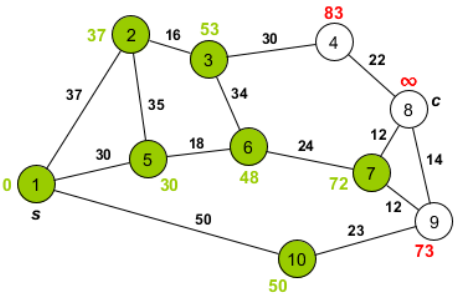
Riešenie nájdite samostatne, postupnosť obrázkov využite na kontrolu správnosti svojho riešenia.

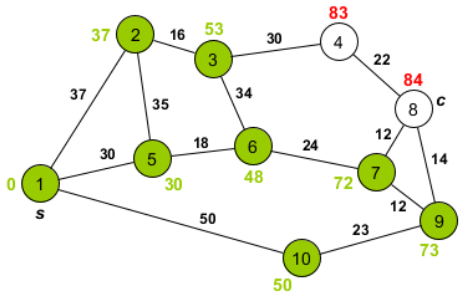
Vizualizácia riešenia:

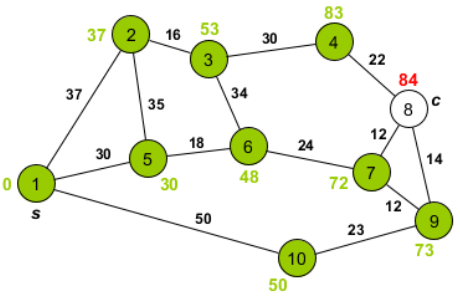


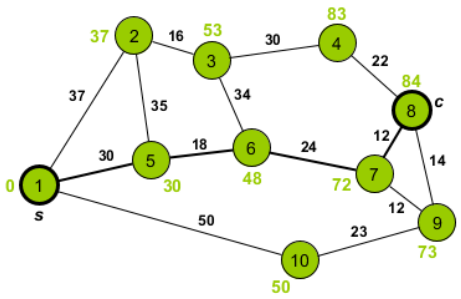












Po skončení algoritmu poznáme dĺžky najkratších ciest z vrcholu *s* do všetkých ostatných vrcholov (ich dĺžky aj postupnosť vrcholov, ktoré najkratšiu cestu tvoria). Najkratšiu cestu z mesta 1 do mesta 8 sme v obrázku vyznačili hrubšou čiarou. Ide o cestu 1, 5, 6, 7, 8. Jej dĺžka je 84 km.

Úloha 4

Naprogramujte triedu `Dijkstra`, pomocou ktorej budete môcť na ohodnotenom grafe aplikovať Dijkstrov algoritmus a zistiť najkratšiu cestu z počiatočného vrcholu *s* do cieľového vrcholu *c*.

V programe budeme potrebovať viacero premenných: Na uloženie informácie o najkratších vzdialenostiach z počiatočného vrcholu `s` využijeme pomocný zoznam `d`. V zozname `p` si musíme pre každý vrchol zapamätať jeho predchodcu na najkratšej ceste. Inak by sme najkratšiu cestu nevedeli na záver zrekonštruovať. Vrcholy, ktoré už majú trvalé ohodnotenie (najkratšia cesta do nich je už známa) budeme priebežne pridávať do množiny `t`.

V triede najprv zabezpečíme správnu inicializáciu premenných a spustenie algoritmu. Poskytneme tiež metódy, pomocou ktorých budeme môcť získať výsledok výpočtu:

```
from ohodnoteny_graf import OhodnotenyGraf
import sys                                                    #[1]

class Dijkstra:
    '''Algoritmus na hľadanie najkratšej cesty v ohodnotenom
       grafe.'''
    def __init__(self, g, s):
        '''Inicializuje atribúty objektu.'''
        # počiatočný vrchol
        self.__start = s
        # zoznam s ohodnoteniami vrcholov
        self.__d = [sys.maxsize] * (g.pocet_vrcholov()+1)
        # zoznam predchodcov na najkratšej ceste
        self.__p = [-1] * (g.pocet_vrcholov()+1)              #[2]
        self.__najkratsia_cesta(g, s)

    def __najkratsia_cesta(self, g, s):
        '''Implementácia Dijkstrovho algoritmu'''
        # dĺžka cesty z vrcholu s do s
        self.__d[s] = 0
        # množina vrcholov s trvalým ohodnotením
        t = set()
        # hľadanie najkratšej cesty začína vo vrchole s
        u = s
        while not u in t:
```

```
            # vrchol u pridáme do množiny t vrcholov s trvalým
            # ohodnotením
            t.add(u)
            # prezrieme všetky susedné vrcholy vrcholu u
            for v in g.susedia(u):
                h = g.hodnota_hrany(u, v)
                # overíme, či sa dá cesta do v skrátiť
                if self.__d[u] + h < self.__d[v]:
                    # vrchol v má nové ohodnotenie
                    self.__d[v] = self.__d[u] + h
                    # aj nového predchodcu na najkratšej ceste
                    self.__p[v] = u
            # nájdeme ďalší vrchol s trvalým ohodnotením
            u = s
            mind = sys.maxsize                                    #[3]
            for v in g.vrcholy():
                if (not v in t) and (self.__d[v] < mind):
                    mind = self.__d[v]
                    u = v

    def dlzka_najkratsej_cesty(self, c):
        '''Vráti dĺžku najkratšej cesty do vrcholu c.'''
        return self.__d[c]

    def najkratsia_cesta(self, c):
        '''Vráti najkratšiu cestu do vrcholu c (ako zoznam).'''
        cesta = [c]
        x = c                                                     #[4]
        while x != self.__start:
            x = self.__p[x]
            cesta.append(x)

        cesta.reverse()
        return cesta
```

[1] Modul `sys` importujeme preto, lebo chceme použiť konštantu `sys.maxsize`. Ide o konštantu, ktorá je dostatočne veľká na to, aby bola určite väčšia od všetkých dĺžok najkratších ciest z vrcholu *s* do ostatných vrcholov. Vypíšte si jej hodnotu na konzolu.

[2] Prvky zoznamu predchodcov môžeme pred začiatkom výpočtu nastaviť napr. na hodnotu -1 (zatiaľ nie sú známe).

[3] V zozname vrcholov grafu hľadáme *minimum* spomedzi aktuálnych ohodnotení tých vrcholov, ktoré zatiaľ nie sú v množine `t`.

[4] Najkratšiu cestu sme spätne zrekonštruovali rovnakým spôsobom aj v implementácii prehľadávanie do šírky (algoritmu BFS) v predchádzajúcej kapitole.

Úloha 5

Triedu `Dijkstra` otestujte na vlastnom ohodnotenom grafe reprezentujúcom cestnú dopravnú sieť.

Výkladový text

Dijkstrov algoritmus je príkladom algoritmu založeného na tzv. **dynamickom programovaní**. Ide o jednu zo stratégií navrhovania efektívnych algoritmov, ktorá môže byť užitočná aj pri riešení mnohých iných problémov. **Princíp dynamického programovania** priblížime v nasledujúcich bodoch:

1. zmenši rozmer úlohy až na ľahko riešiteľný triviálny prípad
2. výsledok riešenia triviálneho prípadu si zapamätaj
3. zväčši rozmer úlohy a využi pri jeho riešení výsledok predchádzajúceho výpočtu
4. pokračuj v postupnom výpočte riešení úloh väčších rozmerov s využitím už známych výsledkov
5. oznám výsledok riešenia pôvodnej úlohy

Úloha 6

Pre každý z bodov v predchádzajúcom všeobecnom vysvetlení metódy dynamického programovania sformulujte, ako sa príslušná myšlienka uplatňuje v Dijkstrovom algoritme.

Riešenie:

1.	Spomedzi susedných vrcholov počiatočného vrcholu *s* ľahko nájdeme ten, do ktorého vedie hrana s najmenším ohodnotením (a preto je nutne najkratšou cestou do tohto vrcholu). Nazvime ho *u*.
2.	Dĺžku najkratšej cesty do vrcholu *u* uložíme do zoznamu `d`, predchodcu na najkratšej ceste (vrchol *s*) si zapamätáme v zozname `p` (v oboch prípadoch ide o nastavenie prvku s indexom *u*).
3.	Poznáme najkratšiu cestu do vrcholu *u*. Teraz nás budú zaujímať dĺžky ciest do tých vrcholov, do ktorých sa dá dostať cez vrchol *u*. Využijeme hodnoty dočasných ohodnotení susedov vrcholu *u* v zozname `d` a porovnáme ich s dĺžkami ciest cez vrchol *u*. Ak je cesta cez vrchol *u* kratšia, aktualizujeme príslušné hodnoty v zoznamoch `d` a `p`. Nájdeme ďalší vrchol s trvalým ohodnotením.
4.	Pre každý ďalší vrchol, do ktorého najkratšiu cestu už poznáme, overíme, či nie je možné cezeň skrátiť cestu do niektorého z jeho susedných vrcholov. Používame pritom údaje o dočasných ohodnoteniach vrcholov v zozname `d` získané v predošlých krokoch.
5.	Po skončení algoritmu obsahuje zoznam `d` dĺžky najkratších ciest z počiatočného vrcholu *s* do všetkých vrcholov grafu. Vypíšeme dĺžku najkratšej cesty pre zvolený vrchol (vrcholy). Z údajov v poli `p` o predchodcoch vrcholov na najkratšej ceste ľahko zrekonštruujeme poradie vrcholov, ktoré najkratšiu cestu predstavuje.

Čo sme sa naučili

- Dopravnú sieť (napr. mestá prepojené úsekmi ciest) reprezentujeme ohodnoteným grafom. Ohodnotený graf možno implementovať rôznym spôsobom, napr. ako maticu vzdialeností.
- Jedným zo základných problémov v teórii grafov je problém hľadania najkratšej cesty v ohodnotenom grafe. Dijkstrovým algoritmom vypočítame dĺžky najkratších ciest z počiatočného vrcholu do všetkých ostatných vrcholov grafu. Výsledkom výpočtu je tiež informácia o poradí vrcholov na týchto najkratších cestách.
- Dijkstrov algoritmus je príkladom uplatnenia stratégie riešenia problémov známej pod názvom dynamické programovanie.

Ďalšie úlohy na precvičenie a zamyslenie

1. Zamyslite sa nad časovou zložitosťou Dijkstrovho algoritmu na hľadanie najkratšej cesty. Ak má graf *n* vrcholov, koľko porovnaní sa vykoná?

2. Doplňte do triedy `Dijkstra` metódu na výpis všetkých vypočítaných výsledkov. Na výstupe chceme *pre každý vrchol* vidieť dĺžku najkratšej cesty doplnenú zodpovedajúcim poradím vrcholov.

3. Vráťme sa k mape regiónu znázornenej ohodnoteným grafom z úvodu kapitoly. Predstavme si, že chceme otvoriť nové obchodné centrum a máme sa rozhodnúť, v ktorom meste ho postavíme. Najlepšie by bolo vybrať také mesto, aby nikto z obyvateľov ostatných okolitých miest nemal do nového obchodu príliš ďaleko. Sformulujeme teraz tento problém v jazyku teórie grafov: V grafe musíme vyhľadať *centrálny vrchol* t. j. taký vrchol, ktorý má spomedzi všetkých vrcholov *najmenšiu excentricitu* (excentricitou vrcholu *x* rozumieme najdlhšiu spomedzi najkratších ciest z vrcholu *x* do všetkých ostatných vrcholov grafu).

   Pomôcka: Aby sme vrchol s touto vlastnosťou našli, potrebujeme zrejme poznať najkratšie cesty medzi všetkými dvojicami vrcholov daného grafu.

19. Ako odhaliť podvod pomocou štatistiky

Kľúčové slová

Benfordov zákon, manipulácia s dátami, podvod, otvorené dáta, frekvenčná analýza, csv formát, xml formát, json formát, serializácia, deserializácia

Čo sa naučíme a čo si precvičíme

- niektoré prirodzene vzniknuté množiny čísiel majú zaujímavé vlastnosti týkajúce sa prvej číslice čísiel,
- využívať otvorené dáta na analýzu,
- analyzovať dáta a odhaľovať možné manipulácie s dátami,
- rozumieť rôznym formátom dát,
- čítať a ukladať dáta vo formáte CSV, XML a JSON,
- používať moduly pre prácu s rôznymi formátmi dát.

Problémová situácia

Zahrajte sa vo dvojiciach nasledovnú hru:

1. každý z vás si vymyslí nejaké prirodzené číslo,
2. umocnite tieto čísla medzi sebou ($číslo1^{číslo2}$),
3. ak je prvá cifra mocniny 1, 2 alebo 3 vyhráva starší z vás,

   ak je prvá cifra mocniny 4, 5, 6, 7, 8 alebo 9 vyhráva mladší z vás.

Je táto hra spravodlivá? Ak nie, kto je znevýhodnený a prečo?

Úloha 1

Navrhnite jednoduchý model uvedenej hry a simulujte niekoľko hier. V koľkých prípadoch vyhral starší a v koľkých mladší?

Zrejme vás výsledok simulácie prekvapil. Ako je to možné? Pozrime sa na to, akými ciframi mocniny myslených čísiel začínajú.

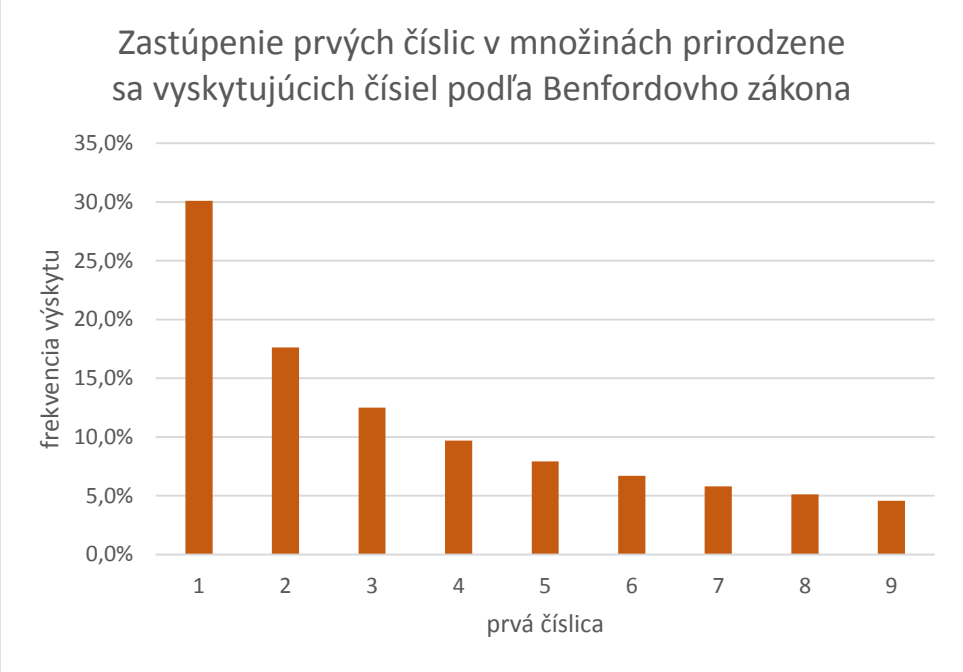
Úloha 2

Upravte váš model z úlohy 1 tak, aby ste zistili, s akou frekvenciou sa vyskytujú jednotlivé cifry na prvom mieste mocnín.

Ak váš model funguje správne, zistili ste že jednotlivé cifry nie sú na prvom mieste zastúpené rovnako. Podobnú disproporciu si všimol aj v roku 1881 kanadský astronóm Simon Newcomb (97).

Zaujali ho logaritmické tabuľky, na ktorých boli začiatočné strany omnoho viac ošúchané ako tie koncové. Usúdil, že ľudia omnoho častejšie hľadajú logaritmy čísiel, ktoré začínajú nižšou číslicou než vysokou číslicou. Aj keď Newcomb odvodil pravdepodobnostné výskyty jednotlivých číslic na prvých miestach a svoje zistenie publikoval, jeho kolegov to príliš nezaujalo. Záujem vedeckej obce vzbudil až v roku 1938 Frank Benford (98), ktorý znovuobjavil

túto anomáliu. Podľa neho je zákon popisujúci zastúpenie číslic na prvých miestach v niektorých množinách čísiel pomenovaný ako Benfordov zákon (pozri obrázok nižšie).



Obrázok 87 Benfordov zákon

Svoje zistenia podložil rozsiahlym výskumom na niekoľkých datasetoch (pozri obrázok nižšie).

PERCENTAGE OF TIMES THE NATURAL NUMBERS 1 TO 9 ARE USED AS FIRST DIGITS IN NUMBERS, AS DETERMINED BY 20,229 OBSERVATIONS

Group	Title	First Digit									Count
		1	2	3	4	5	6	7	8	9	
A	Rivers, Area	31.0	16.4	10.7	11.3	7.2	8.6	5.5	4.2	5.1	335
B	Population	33.9	20.4	14.2	8.1	7.2	6.2	4.1	3.7	2.2	3259
C	Constants	41.3	14.4	4.8	8.6	10.6	5.8	1.0	2.9	10.6	104
D	Newspapers	30.0	18.0	12.0	10.0	8.0	6.0	6.0	5.0	5.0	100
E	Spec. Heat	24.0	18.4	16.2	14.6	10.6	4.1	3.2	4.8	4.1	1389
F	Pressure	29.6	18.3	12.8	9.8	8.3	6.4	5.7	4.4	4.7	703
G	H.P. Lost	30.0	18.4	11.9	10.8	8.1	7.0	5.1	5.1	3.6	690
H	Mol. Wgt.	26.7	25.2	15.4	10.8	6.7	5.1	4.1	2.8	3.2	1800
I	Drainage	27.1	23.9	13.8	12.6	8.2	5.0	5.0	2.5	1.9	159
J	Atomic Wgt.	47.2	18.7	5.5	4.4	6.6	4.4	3.3	4.4	5.5	91
K	$n^{-1}, \sqrt{n}, \cdots$	25.7	20.3	9.7	6.8	6.6	6.8	7.2	8.0	8.9	5000
L	Design	26.8	14.8	14.3	7.5	8.3	8.4	7.0	7.3	5.6	560
M	*Digest*	33.4	18.5	12.4	7.5	7.1	6.5	5.5	4.9	4.2	308
N	Cost Data	32.4	18.8	10.1	10.1	9.8	5.5	4.7	5.5	3.1	741
O	X-Ray Volts	27.9	17.5	14.4	9.0	8.1	7.4	5.1	5.8	4.8	707
P	Am. League	32.7	17.6	12.6	9.8	7.4	6.4	4.9	5.6	3.0	1458
Q	Black Body	31.0	17.3	14.1	8.7	6.6	7.0	5.2	4.7	5.4	1165
R	Addresses	28.9	19.2	12.6	8.8	8.5	6.4	5.6	5.0	5.0	342
S	$n^1, n^2 \cdots n!$	25.3	16.0	12.0	10.0	8.5	8.8	6.8	7.1	5.5	900
T	Death Rate	27.0	18.6	15.7	9.4	6.7	6.5	7.2	4.8	4.1	418
Average		30.6	18.5	12.4	9.4	8.0	6.4	5.1	4.9	4.7	1011
Probable Error		±0.8	±0.4	±0.4	±0.3	±0.2	±0.2	±0.2	±0.2	±0.3	—

Obrázok 88 Relatívne frekvencie výskytu jednotlivých číslic na prvom mieste pre rôzne datasety. Prevzaté z (98)

Napriek tomu, že mnohé množiny hodnôt vykazujú tendenciu správať sa podľa Benfordovho zákona, uspokojivý dôkaz prečo tomu tak je, zatiaľ nemáme. Vo všeobecnosti tento zákon platí:

- v množinách prirodzene sa vyskytujúcich čísiel, napr. výsledky volieb, zostatky na účtoch, účtovné výkazy,
- v množinách čísiel, ktoré vznikli matematickými operáciami z čísiel z rôznych množín čísiel,
- v množinách s veľkou variabilitou čísiel,
- v množinách, kde nie je definované minimum (s výnimkou nulového minima) a maximum.

Na druhej strane by sme mali byť opatrní a nepredpokladať, že Benfordov zákon platí v prípadoch:

- cien tovarov v obchodoch, pretože obchodníci majú tendenciu používať takzvané baťovské ceny,
- výšky odmien za prácu, lebo tu platí presne opačný princíp, zamestnávatelia majú tendenciu posunúť výšku odmeny za nejakú okrúhlu hranicu než ostať tesne pod ňou,
- obmedzeného rozsahu hodnôt, napr. výšky ľudí, hmotnosti ľudí a pod.

Poznámka na okraj

Ďalšie konkrétne množiny dát, pre ktoré platí Benfordov zákon môžete nájsť na (99).

Ak predpokladáme, že v danej množine dát by mal Benfordov zákon platiť, je možné tento zákon využiť na detekciu falšovania týchto dát. Pre volebné výsledky, finančné, účtovné a poistné údaje by mal Benfordov zákon platiť. Viac o Benfordovom zákone nájdete v (100).

V súbore `benford.py` je pripravená trieda `BenfordovZakon()`, ktorá sa dá použiť pre analýzu čísiel poľa Benfordovho zákona.

Trieda obsahuje nasledovné verejné metódy a atribúty:

- `bz = BenfordovZakon()`
  vytvoríme objekt pre analýzu čísiel podľa Benfordovho zákona,
- `bz.registruj(self, udaj)`
  zaregistruje číselný údaj pre neskoršiu analýzu frekvencie výskytu číslic na prvých miestach, nulové a prázdne hodnoty sa ignorujú, reťazce sa konvertujú na čísla,
- `bz.reset(self)`
  zmaže všetky registrované hodnoty,
- `bz.pocet`
  počet platných registrovaných hodnôt,
- `bz.frekvencie`
  slovník absolútnych početností číslic na prvých miestach zadaných údajov,
- `bz.frekvencie_percenta`
  slovník relatívnych početností číslic na prvých miestach zadaných údajov,

- `bz.test`
  výsledok testu, či pre zadané číselné údaje platí Benfordov zákon.

Poznámka na okraj

Na overenie, či početnosti výskytu číslic na prvých miestach čísiel zodpovedajú Benfordovmu zákonu, sme v triede `BenfordovZakon()` použili test dobrej zhody CHÍ-KVADRÁT. (101)

Pravdepodobnosť, že množina dát podlieha Benfordovmu zákonu a výsledok testu povie, že dáta tomuto zákonu nepodliehajú je 0,05.

Predpokladá sa, že počet registrovaných hodnôt je aspoň 109 (dôsledok Cochranovho pravidla).

Úloha 3

Otestujte, či mocniny náhodne vygenerovaných dvoch čísiel zodpovedajú Benfordovmu zákonu. Overte rôzne počty mocnín.

Otvorené dáta

Podľa (102), „Otvorené dáta sú informácie alebo údaje voľne a bezplatne dostupné pre každého za rovnakých podmienok, ktoré je možné použiť na akýkoľvek účel komerčného či nekomerčného charakteru. Sú sprístupnené na internete v štruktúrovanej forme, ktorá umožňuje ich hromadné strojové spracovanie.“

Otvorené dáta predstavujú výbornú príležitosť na kontrolu ako funguje a ako hospodári štát, mesto alebo samospráva. Myšlienka spočíva v tom, že tieto dáta môžeme spracovávať, analyzovať a získavať z nich ďalšie užitočné informácie. Keďže sú zväčša zverejňované vo formátoch, ktoré umožňujú strojové spracovanie, môžeme ich analyzovať pomocou počítačov.

Poznámka na okraj

Realita týkajúca sa zverejňovania otvorených dát ale nie je taká ružová. Aj keď sú dáta zverejnené v elektronickej podobe, problémy spôsobujú rôzne formáty dokumentov, nekonzistencia alebo duplicita dát, nejednotnosť v pomenovaní rovnakých atribútov, vzájomná neprepojenosť súborov dát a pod. Presvedčíme sa o tom pri ich spracovaní v nasledujúcej časti kapitoly.

Otvorené dáta by mali byť poskytované v otvorených formátoch. Najčastejšie tieto dáta nájdeme vo formátoch csv, xml alebo json. Pozrime sa postupne na to, ako s dátami v týchto formátoch pracovať v jazyku Python.

Formát CSV

Poznámka na okraj

Presná špecifikácia formátu CSV neexistuje. Popis tohto formátu nájdeme v RFC 4180 (103).

Formát CSV (Comma-separated values) predstavuje jeden z najjednoduchších formátov pre zverejňovanie dát. Podľa názvu by mali byť hodnoty oddelené čiarkami. V skutočnosti sa stretneme s rôznymi oddeľovačmi: bodkočiarky, tabulátory, dvojbodky a pod.

Jednoduchý CSV súbor, v ktorom sú uložené výšky odmien zamestnancov (`odmeny.csv`) môže vyzerať napr. takto:

```
meno;priezvisko;odmena
Andrej;Pracovitý;540
Jakub;Pohodový;20
Tereza;"Rozum-Šikovná";360
Roman;Ničnerobý;
```

Výkladový text

Pre prácu s dátami vo formáte CSV môžeme v Pythone použiť modul `csv`. Modul poskytuje široké spektrum nástrojov.

Na čítanie dát z CSV súboru môžeme použiť objekt triedy `DictReader()`. Pomocou tejto triedy vieme čítať riadky CSV súboru. K jednotlivým položkám riadku (záznamu) pristupujeme podobne ako k prvkom slovníka.

```
# spracuj_csv.py
import csv

#citanie dat z CSV suboru
with open('odmeny.csv', newline='', encoding='utf-8') as subor:  # [1]
    reader = csv.DictReader(subor, delimiter=';')                # [2]
    print(f'zahlavie: {reader.fieldnames}')                      # [3]
    for riadok in reader:                                        # [4]
        meno = riadok['meno']                                    # [5]
        priezvisko = riadok['priezvisko']
        odmena = riadok['odmena']
        print(f'meno: {meno}, priezvisko: {priezvisko}, odmena: {odmena}')
```

[1] Súbor otvoríme na čítanie. Je vhodné uviesť použité kódovanie znakov v súbore.

[2] Vytvoríme objekt triedy `DictReader()`. Tu je vhodné špecifikovať znak, ktorý je oddeľovačom hodnôt v riadku.

[3] Prvý riadok v súbore sa automaticky použije ako hlavička, v ktorej sú definované názvy jednotlivých polí. Hlavička je prístupná v atribúte `fieldnames`.

[4] Objekt triedy `DictReader()` je iterovateľný. Jeho iterovaním postupne prechádzame riadkami súboru.

[5] Jednotlivé polia v riadku sú prístupné cez index definovaný v hlavičke.

Na zápis dát do CSV súboru využijeme objekt triedy `DictWriter()`.

```
# spracuj_csv.py
import csv

#zapis dat do CSV suboru
with open('platy.csv', 'w', newline='', encoding='utf-8', ) as subor:          # [6]
    zahlavia = ['meno', 'priezvisko', 'odmena']                                # [7]
    writer = csv.DictWriter(subor, delimiter=';', fieldnames=zahlavia)         # [8]
    writer.writeheader()                                                       # [9]
    writer.writerow({'meno': 'Jakub', 'priezvisko': 'Pohodový', 'odmena': 860})#[10]
    writer.writerow({'meno': 'Tereza', 'priezvisko': '"Rozum-Šikovná"', 'odmena':
930})
```

[6] Súbor otvoríme na zápis. Je vhodné uviesť použité kódovanie znakov v súbore.

[7] Záhlavia definujeme ako zoznam reťazcov.

[8] Vytvoríme objekt triedy `DictWriter()`. Tu je vhodné špecifikovať znak, ktorý je oddeľovačom hodnôt v riadku.

[9] Zapíšeme hlavičku do súboru.

[10] Postupne zapisujeme jednotlivé záznamy. Všimnime si, že každý záznam (riadok) definujeme ako slovník. Kľúče v slovníku zodpovedajú názvom, ktoré sme definovali v hlavičke.

Podrobný popis modulu `csv` nájdete na https://docs.python.org/3/library/csv.html [30. 5. 2019]

Na webovej stránke Národnej agentúry pre sieťové a elektronické služby (https://data.gov.sk/) sú prístupné rôzne datasety. Zverejnené datasety sa dajú filtrovať podľa rôznych kritérií.

Úloha 4

Na stránke https://data.gov.sk/ v časti „Datasety“ nájdite dokument `Štatistika kriminality v Slovenskej republike za rok 2017 / v SR - podľa §`. Stiahnite tento dataset vo formáte CSV a overte, či počty zistených a počty objasnených trestných činov podliehajú Benfordovmu zákonu.

Pomôcka: Preskúmajte súbor a prípadne ho upravte tak, aby zodpovedal popisu CSV formátu. Overte si, aké kódovanie bolo použité pri vytváraní dokumentu.

Formát XML

Poznámka na okraj

Presnú špecifikáciu formátu XML nájdeme (104).

Formát XML (eXtensible Markup Language), na rozdiel od CSV, je definovaný štandardom. XML je značkovací jazyk primárne určený na zverejňovanie dát a výmenu dát medzi aplikáciami. Jednotlivé značky nie sú definované a je len na používateľovi, aké značky si zvolí. Výsledný dokument sa skladá zo značiek a znakov. Značky definujú štruktúru a znaky samotný obsah dokumentu.

Jednoduchý XML dokument, v ktorom sú uložené výšky odmien zamestnancov (`odmeny.xml`) môže vyzerať napr. takto:

```
<?xml version="1.0" encoding="UTF-8"?>
<zamestnanci>
    <zamestnanec kategoria="dohoda o vykonani prace">
        <meno>Andrej</meno>
        <priezvisko>Pracovitý</priezvisko>
        <odmena>540</odmena>
    </zamestnanec>
    <zamestnanec kategoria="trvaly pracovny pomer">
        <meno>Jakub</meno>
        <priezvisko>Pohodový</priezvisko>
        <odmena>20</odmena>
    </zamestnanec>
    <zamestnanec kategoria="dohoda o vykonani prace">
        <meno>Tereza</meno>
        <priezvisko>Rozum-Šikovná</priezvisko>
        <odmena>360</odmena>
    </zamestnanec>
    <zamestnanec>
        <meno>Roman</meno>
        <priezvisko>Ničnerobý</priezvisko>
        <odmena></odmena>
    </zamestnanec>
</zamestnanci>
```

Poznámka na okraj

Ak vám XML dokument pripomína zdrojový kód webovej stránky, nie je to náhoda. Jazyk XHTML je založený na jazyku XML.

HTML alebo HTML5 vyzerajú podobne. Nie sú však aplikáciou XML. Používajú menej prísnu syntax.

Prvý riadok obsahuje XML deklaráciu, ktorá špecifikuje verziu XML, ktorú dokument používa a kódovanie dokumentu. Tento riadok nie je povinný (prednastavené kódovanie je UTF-8).

Všimnime si, že XML dokument má stromovú (rekurzívnu) štruktúru. Dokument musí obsahovať práve jeden koreňový element (<**zamestnanci**>). Do neho môžu byť postupne

vnorené ďalšie elementy (<zamestnanec>) a v nich ďalšie (<meno>, <priezvisko>, <odmena>). V elementoch môžu byť definované atribúty s priradenými hodnotami (kategoria="dohoda o vykonani prace").

Výkladový text

Pre prácu s dátami vo formáte XML môžeme v Pythone použiť modul `xml.etree.ElementTree`. Modul poskytuje jednoduché rozhranie pre parsovanie a vytváranie XML dokumentov.

Prečítať obsah XML dokumentu môžeme nasledovne:

```
# spracuj_xml.py
import xml.etree.ElementTree as xml

# citanie dat z XML suboru
strom = xml.parse('odmeny.xml')                                    # [1]
zamestnanci = strom.getroot()                                      # [2]
for zamestnanec in zamestnanci:                                    # [3]
    print(zamestnanec.attrib['kategoria'])                         # [4]
    print(f'meno: {zamestnanec[0].text}, '                         # [5]
          f'priezvisko: {zamestnanec[1].text}, '
          f'odmena: {zamestnanec[2].text}')
```

[1] Analyzujeme XML dokument. Výsledkom je referencia na stromovú štruktúru tohto dokumentu.

[2] Získame referenciu na koreňový element v danej stromovej štruktúre.

[3] Keďže elementy môžu mať v sebe vnorené ďalšie elementy, môžeme nimi prechádzať v cykle. Pri jednotlivých iteráciách cyklu je `zamestnanec` referenciou na jednotlivé elementy (<zamestnanec> ... </zamestnanec>) vnorené v koreňovom elemente <zamestnanci>.

[4] Ak element má definovaný nejaký atribút, metóda `attrib()` nám vráti jeho hodnotu.

[5] Ku vnoreným elementom sa vieme dostať aj cez index. Obsah elementu je prístupný pomocou atribútu `text`. Keďže element `zamestnanec` má v sebe vnorené ďalšie elementy, aj k týmto sa vieme dostať v cykle. Riadok [5] by sme mohli nahradiť nasledovne:

```
for info in zamestnanec:
    print(f'{info.tag}: {info.text}')
```

Vytvoriť nový XML dokument môžeme nasledovne:

```
# spracuj_xml.py
import xml.etree.ElementTree as xml

# zapis dat do XML suboru
zamestnanci = xml.Element('zamestnanci')                           #[6]

jakub = xml.SubElement(zamestnanci, 'zamestnanec', kategoria='brigada')#[7]
```

```
xml.SubElement(jakub, 'meno').text = 'Jakub'                                    #[8]
xml.SubElement(jakub, 'priezvisko').text = 'Pohodový'
xml.SubElement(jakub, 'plat').text = '860'

tereza = xml.SubElement(zamestnanci, 'zamestnanec', kategoria='dohoda')
xml.SubElement(tereza, 'meno').text = 'Tereza'
xml.SubElement(tereza, 'priezvisko').text = 'Rozum-Šikovná'
xml.SubElement(tereza, 'plat').text = '930'

xml.ElementTree(zamestnanci).write('platy.xml', encoding='utf-8',
xml_declaration='<?xml version="1.0">')                                         #[9]
```

[6] Vytvoríme koreňový element a referenciu naň uložíme do premennej `zamestnanci`.

[7] Vytvoríme vnorený element. Prvým atribútom funkcie `SubElement()` je nadradený element (`zamestnanci`), druhým názov nového vnoreného elementu (**`zamestnanec`**). Špecifikovať môžeme aj atribúty a ich hodnoty. Funkcia vracia referenciu na novovytvorený element.

[8] Postupne vytvárame ďalšie vnorené elementy. Ich obsah priradíme atribútu `text`.

[9] Vytvorený XML obsah zapíšeme do súboru. Využijeme triedu `ElementTree()`.

Podrobný popis modulu `xml.etree.ElementTree` nájdeme na https://docs.python.org/3/library/xml.etree.elementtree.html [30. 5. 2019].

Mesto Prešov na svojej webovej stránke (http://egov.presov.sk/) zverejňuje množstvo dát a štatistických materiálov.

Úloha 5

Na stránke http://egov.presov.sk/ v časti „Katalóg OPEN DATA“ je zverejnený dokument „3. Zoznam faktúr“. Stiahnite tento dokument vo formáte XML a overte, či ceny, ktoré boli fakturované podliehajú Benfordovmu zákonu.

V čom je rozdiel (okrem formátu) pri spracovaní CSV a XML dokumentu?

Pomôcka: Preskúmajte súbor a zistite, v ktorom elemente a kde je uvedená fakturovaná suma.

Zaujímavým zistením je, že množina fakturovaných cien nepodlieha Benfordovmu zákonu. Vyhlásiť dáta za zmanipulované by bolo príliš zjednodušené a unáhlené. Dáta je potrebné podrobiť ďalšej analýze.

- Faktúry pre jednotlivé roky môžeme skúmať ako samostatné množiny a každú vyhodnotiť zvlášť.
- Zvlášť môžeme vyhodnotiť množinu cien pre každého dodávateľa.

Úloha 6

Analyzujte množiny cien pre každý rok samostatne.

Výsledok tejto analýzy opäť ukáže, že dáta pre žiadny rok nepodliehajú Benfordovmu zákonu. Výnimky tvoria roky, v ktorých je príliš málo záznamov pre takúto analýzu.

Úloha 7

Analyzujte množiny cien pre každého dodávateľa samostatne.

Formát JSON

Poznámka na okraj

Presnú špecifikáciu formátu JSON nájdeme v (105).

Formát JSON (JavaScript Object Notation) je textový formát určený pre výmenu dát. Formát JSON používa syntax jazyka JavaScript, ale napriek tomu je jazykovo nezávislý. Dáta sú uložené ako dvojice: `názov : hodnota` a vzájomne oddelené čiarkou. Dáta uzatvorené v hranatých zátvorkách sú interpretované ako prvky polí (v Pythone zoznam) a dáta uzatvorené v kučeravých zátvorkách sú interpretované ako atribúty objektu (v Pythone slovník). JSON dokumenty používajú kódovanie utf-8.

Jednoduchý JSON súbor, v ktorom sú uložené výšky odmien zamestnancov (`odmeny.json`) môže vyzerať napr. takto:

```
[
  {
    "meno": "Andrej",
    "priezvisko": "Pracovitý",
    "odmena": 540
  },
  {
    "meno": "Jakub",
    "priezvisko": "Pohodový",
    "odmena": 20
  },
  {
    "meno": "Tereza",
    "priezvisko": "Rozum-Šikovná",
    "odmena": 360
  },
  {
    "meno": "Roman",
    "priezvisko": "Ničnerobý",
    "odmena": ""
  }
]
```

Výkladový text

Pre prácu s dátami vo formáte JSON môžeme v Pythone použiť modul `json`. Modul poskytuje rozhranie pre prácu s dátami vo formáte JSON.

Prečítať obsah JSON dokumentu môžeme nasledovne:

```
# spracuj_json.py
import json
```

```
# citanie dat z JSON suboru
with open('odmeny.json', encoding='utf-8') as f:
    zamestnanci = json.load(f)                                  # [1]
    print(type(zamestnanci))
    for zamestnanec in zamestnanci:                             # [2]
        for kluc, hodnota in zamestnanec.items():               # [3]
            print(f'{kluc}: {hodnota}')                        # [4]
```

[1] Deserializujeme dáta uložené v súbore. Výsledkom funkcie `load()` je objekt, reprezentujúci dáta zo súboru. V tomto prípade je to zoznam slovníkov.

[2] Prechádzame prvkami zoznamu. Každý prvok je slovník.

[3] Prechádzame prvkami slovníka.

[4] Vypíšeme dvojicu kľúč a hodnota.

Vytvoriť nový JSON dokument môžeme nasledovne:

```
# spracuj_json.py
import json

# zapis dat do JSON suboru
jakub = {'meno': 'Jakub', 'priezvisko': 'Pohodový', 'plat': 860}
tereza = {'meno': 'Tereza', 'priezvisko': 'Rozum-Šikovná', 'plat': 930}
data = (jakub, tereza)                                          # [5]
with open('platy.json', 'w', encoding='utf-8') as f:
    json.dump(data, f, ensure_ascii=False, indent=2)            # [6]
```

[5] Vytvoríme zoznam obsahujúci dva slovníky. Každý je záznamom jedného zamestnanca.

[6] Funkcia `dump()` serializuje objekt do formátu JSON a výsledok uloží do súboru.

Podrobný popis modulu `json` nájdeme na https://docs.python.org/3/library/json.html [30. 5. 2019]

Úloha 8

Na stránke http://egov.presov.sk/ v časti „Katalóg OPEN DATA“ je zverejnený dokument „16. Počet občanov podľa ulíc“. Stiahnite tento dokument vo formáte JSON a overte, či počty obyvateľov ulíc podliehajú Benfordovmu zákonu.

Úloha 9

V roku 2019 sa uskutočnili voľby prezidenta Slovenskej republiky. V súbore „`PRE_2019_KOLO1.json`“ sú počty hlasov, ktoré získali jednotliví kandidáti v obciach v prvom kole.

Údaje sme získali zo stránky Slovenského štatistického úradu (http://volby.statistics.sk/prez/prez2019/sk/download.html [31. 5. 2019]) a skonvertovali do formátu JSON.

Preskúmajte uvedené dáta a overte, či počty hlasov jednotlivých kandidátov podliehajú Benfordovmu zákonu.

Pomôcka: Navrhnite spôsob, ako na jeden prechod súborom analyzovať počty hlasov pre každého kandidáta.

Čo sme sa naučili

- niektoré prirodzene vzniknuté množiny čísiel podliehajú Benfordovmu zákonu prvej číslice,
- využívať otvorené dáta na analýzu a následnú kontrolu manipulácie s nimi,
- rozumieť rôznym formátom dát,
- čítať a ukladať dáta vo formáte SCV, XML a JSON,
- používať moduly pre prácu s rôznymi formátmi dát,
- CSV formát je vhodný pre tabuľkové dáta,
- JSON a XML sú vhodné aj pre dáta, pre ktoré tabuľková štruktúra nie je postačujúca,
- dáta vo formáte XML zaberajú približne 3 krát viac miesta ako dáta vo formáte CSV,
- dáta vo formáte JSON zaberajú približne 2 krát viac miesta ako dáta vo formáte CSV.

Ďalšie úlohy na precvičenie a zamyslenie

1. Pre spracovanie dát štandardne používame tabuľkový kalkulátor. Väčšina z nich používa na výmenu dát formát CSV. Vytvorte program `json2csv.py`, ktorý skonvertujte volebné výsledky (`PRE_2019_KOLO1.json`) z formátu JSON do formátu CSV. Pre kontrolu správnosti výsledný súbor importujte do hárku tabuľkového kalkulátora.

2. Je možné skonvertovať každý JSON alebo XML súbor do formátu CSV?

3. Mohli by sme znalosť o Benfordovom zákone využiť na zvýšenie šance v hrách typu Športka alebo Loto? Svoju odpoveď overte analýzou histórie vyžrebovaných čísiel. Archív vyžrebovaných čísiel nájdete tu: http://www.tipos.sk/Default.aspx?CatID=115.
4. Pri niektorých dátach vyžadujeme aktuálnosť (napr. predpoveď počasia). Uložiť a analyzovať takéto dáta offline preto nie je prijateľné. Vytvorte program, ktorý vám zobrazí aktuálnu predpoveď počasia na najbližších 24 hodín.

   Pomôcka:

   - XML súbor s aktuálnou predpoveďou počasia môžete získať vďaka Nórskemu meteorologickému inštitútu. Na stránke https://www.yr.no/ vyhľadajte predpoveď počasia pre vybranú destináciu. Do url riadku zobrazenej predpovede doplňte názov súboru forecast.xml (napr.: https://www.yr.no/place/Slovakia/Prešov/Prešov/forecast.xml). Preskúmajte uvedený XML dokument.
   - Obsah XML dokumentu z webu (pracujeme s aktuálnou verziou dokumentu, ktorá je uložená na webe) získame pomocou modulu `urllib.request` nasledovne (adresu skopírujte priamo z prehliadača):

     ```
     f = urllib.request.urlopen('https://www.yr.no.../forecast.xml')
     data = f.read()
     ```
   - Referenciu na koreňový element získame priamo z textu:

     ```
     weatherdata = xml.fromstring(data)
     ```

   Upozornenie: Ak plánujete vašu aplikáciu poskytnúť aj ostatným, prečítajte si podmienky pre použitie dát z yr.no na stránke https://om.yr.no/info/verdata/vilkar/.

20. Programovanie obrázkov

Kľúčové slová

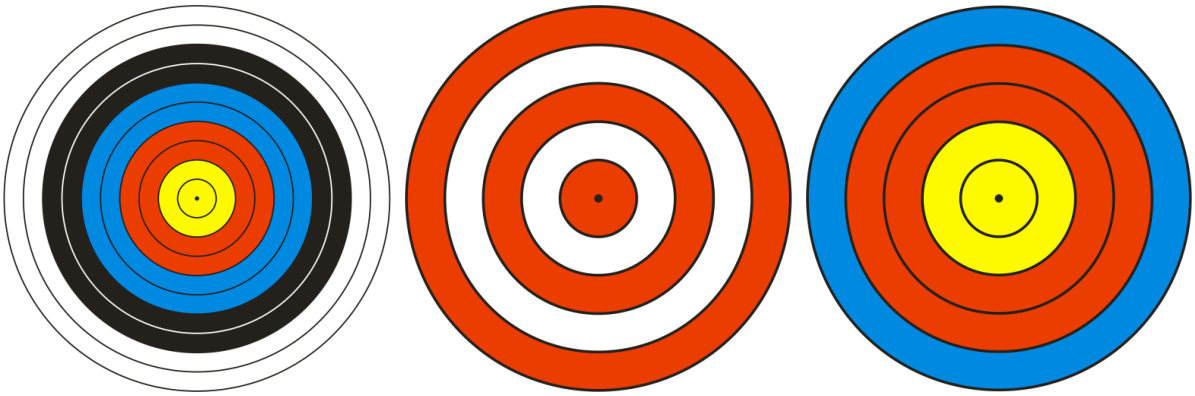
programovanie obrázkov, formát SVG, formát XML, inštalácia modulu, modul svgwrite, vlastná výnimka, dedičnosť, otvorený formát,

Čo sa naučíme a čo si precvičíme

- vytvárať obrázkové súbory (obrázky) pomocou programovania,
- pracovať s formátom SVG,
- definovať triedy a používať objekty,
- definovať vlastné výnimky,
- využívať vlastnosť objektového programovania – dedičnosť.

Problémová situácia

Priatelia z lukostreleckého krúžku nás požiadali a malú pomoc. Na tréningoch streľby zničia veľké množstvo terčov a tak stále potrebujú nové. Požiadali nás, či by sme im vedeli pomôcť s kreslením terčov. Lukostrelecký terč nie je náročný na kreslenie a vyzerá približne takto (pozri obrázok nižšie):



Obrázok 89 Lukostrelecké terče

Prípustné sú aj iné farebné kombinácie, rôzne veľkosti, rôzne počty medzikruží alebo rôzne šírky medzikruží v jednom terči.

Na prvý pohľad jednoduchá úloha sa ukázala, kvôli rôznej variabilite terčov, ako pomerne prácna. Navyše lukostrelci si stále vymýšľajú nové typy terčov.

Hľadajme riešenie

Kreslenie terčov v nejakom nástroji pre kreslenie obrázkov (grafickom editore) neuvažujme. Nie je to síce intelektuálne náročné, ale je to prácne. Pre každý nový typ terča je potrebné terč v editore nakresliť. Naprogramujme si vlastný nástroj na kreslenie terčov. Kým sa do toho pustíme, uvažujme o rôznych grafických formátoch, v ktorých môžeme obrázky vytvárať.

Vo všeobecnosti môžeme uvažovať o rastrovom alebo vektorovom formáte. Vlastnosti oboch poznáme z hodín informatiky. Pre naše potreby je výhodnejší vektorový formát.

Úloha 1

Preskúmajte rôzne vektorové formáty a vyberte ten, s ktorým sa dá v jazyku Python pracovať. Pre naše potreby sú vhodné otvorené formáty, ktoré sú podporované nástrojmi pre spracovanie obrázkov – grafickými editormi, resp. prehliadačmi.

Výkladový text

SVG (Scalable Vector Graphics) je značkovací jazyk založený na jazyku XML. Je primárne určený na opis dvojrozmernej, statickej alebo animovanej vektorovej grafiky. SVG je otvorený formát definovaný W3C (World Wide Web Consortium). Jeho špecifikáciu nájdeme na (106).

Nasledovný obrázok:

je popísaný v súbore `ukazka.svg` nasledovne:

```
<?xml version="1.0" encoding="utf-8" ?>
<svg baseProfile="full"                                          [1]
     height="10cm" width="10cm"                                  [2]
     viewBox="0 0 10 10"                                         [3]
     version="1.1"
     xmlns="http://www.w3.org/2000/svg"
xmlns:ev="http://www.w3.org/2001/xml-events"
xmlns:xlink="http://www.w3.org/1999/xlink">
  <defs/>
  <circle cx="5" cy="5" fill="white" r="4.5" stroke="black" stroke-
width="0.05"/>                                                   [4]
  <circle cx="5" cy="5" fill="red" r="4"/>                       [5]
  <rect fill="white" height="1.5" width="7" x="1.5" y="4.25"/>   [6]
</svg>
```

Vidíme, že súbor `ukazka.svg` je XML dokument (s XML formátom sme sa oboznámili v kapitole 19. Ako odhaliť podvod pomocou štatistiky). Pozrime sa na časti dokumentu, ktoré sú pre nás podstatné.

[1] Koreňovým elementom je element `svg`. Atribúty koreňového elementu definujú vlastnosti dokumentu.

[2] Rozmery dokumentu sú 10cm × 10cm (`height="10cm"` a `width="10cm"`).

[3] Definícia súradnicového systému vo vnútri SVG časti. Vľavo hore je súradnica [0, 0] a rozsah súradníc je [10, 10]. Súradnice objektov vo vnútri `svg` sa takto stanú nezávislé od

absolútnej veľkosti `svg` elementu. Rozmery a umiestnenie elementov preto môžeme uvádzať bez jednotiek. Ak zmeníme rozmery SVG dokumentu, vnútorný súradnicový systém sa „prispôsobí“ novým rozmerom.

[4] Kružnica so stredom v bode [5, 5] (`cx="5" cy="5"`) s polomerom 4,5 (`r="4.5"`) je vyplnená bielou (`fill="white"`) a ohraničená čiernou farbou (`stroke="black"`). Hrúbku čiary sme definovali atribútom `stroke-width`. Všimnime si, že pri súradniciach a rozmeroch už jednotky nepoužívame.

[5] Kružnica so stredom v bode [5, 5] s polomerom 4 je vyplnená červenou farbou.

[6] Biely obdĺžnik, široký 7 (`width="7"`) a vysoký 1,5 (`height="1.5"`). Umiestnenie je definované umiestnením jeho ľavého horného rohu (`x="1.5"` a `y="4.25"`).

Poznámka na okraj

Obrázok, ktorého popis je uložený v SVG súbore si môžeme **zobraziť** napr. pomocou webového prehliadača.

SVG súbor môžeme **editovať** v textovom editore, ktorý podporuje kódovanie znakov utf-8.

Obrázok uložený v SVG formáte môžeme **editovať** vo vektorových grafických editoroch. Najlepšie si s SVG formátom rozumie grafický editor Inkscape (https://inkscape.org/ [4. 6. 2019]) ale obrázky v tomto formáte vedia importovať aj iné grafické programy.

Ďalšie námety pre kreslenie vo formáte SVG môžeme nájsť v rôznych návodoch a tutoriáloch, napr. v (107).

Poznámka na okraj

Otvorený formát (alebo aj otvorený štandard) je zverejnená špecifikácia na uchovávanie údajov. Otvorený formát je možné implementovať do otvorených ale aj do proprietárnych softvérov. Výhodou otvorených formátov je dlhodobá záruka pre používateľov ohľadom ich používania a prístupu k špecifikácii a podpora rôznych konkurenčných implementácii.

Poznámka na okraj

Farby v SVG formáte je možné definovať rôznymi spôsobmi.

V ukážke sme použili názvy farieb. Ich zoznam nájdete na adrese: https://www.w3.org/TR/SVG11/types.html#ColorKeywords [4. 6. 2019].

Okrem toho je možné farby definovať aj pomocou modelu RGB. Viac na https://www.w3.org/TR/SVGColor12/#Color_syntax [4. 6. 2019].

Úloha 2

Upravte zdrojový kód súboru `ukazka.svg` tak, aby ste vytvorili dopravnú značku „Zákaz státia“.

Pomôcka: Šikmé prečiarknutie je možné vytvoriť červeným otočeným obdĺžnikom. Ak potrebujeme nejaký objekt v SVG dokumente otočiť, definujeme to atribútom `transform` *s hodnotou* `rotate(uhol, stredX, stredY)`. `Uhol` *je uhol natočenia v stupňoch a súradnice* `stredX` *a* `stredY` *definujú bod, okolo ktorého sa má objekt otočiť.*

Programujeme SVG obrázky

Návrh nášho riešenia zatiaľ nevyzerá ako efektívny spôsob na kreslenie terčov. Obrázky SVG sme vytvárali ručne a aj to nie veľmi pohodlne. Ukážme si, ako vytvárať SVG obrázky pomocou programovania.

S XML dokumentom sme sa už stretli. Čítať a vytvárať XML dokumenty sme sa naučili v kapitole „19. Ako odhaliť podvod pomocou štatistiky“. SVG dokumenty by sme teda vedeli vytvárať aj pomocou modulu `xml.etree.ElementTree`. My však použijeme modul `svgwrite` určený špeciálne pre vytváranie SVG dokumentov.

Výkladový text

Modul `svgwrite` nie je súčasťou štandardnej inštalácie jazyka Python. Je potrebné ho inštalovať dodatočne.

Ukážme si postupne, ako nájsť a inštalovať nejaký potrebný modul jazyka Python. Zoznam modulov sa udržiava na stránke Python Package Index s adresou https://pypi.org/ [4. 6. 2019].

Ak potrebujeme nejaký modul nájsť, využijeme vyhľadávanie na uvedenej stránke. S vyhľadávaním sa musíme trochu pohrať, lebo množstvo nájdených modulov je príliš veľké a nie každý zodpovedá tomu, čo potrebujeme. Zadajme napr. dopyt „create svg“.

Po kliknutí na názov modulu `svgwrite` dostaneme základné informácie o module aj s návodom na inštaláciu: `pip install svgwrite`.

pip je program na inštaláciu modulov a balíčkov jazyka Python. Nájdeme ho v priečinku `Scripts` priečinku, kde sme inštalovali jazyk Python. Inštalácia prebehne automaticky a **pip** si potrebné dáta stiahne z internetu.

Dokumentáciu k modulu `svgwrite` nájdeme v (108).

Poznámka na okraj

Názov **pip** je slovná hračka – rekurzívny akronym. Pip je skratkou slov „Pip Installs Packages“ alebo „"Pip Installs Python“.

Výkladový text

Dopravnú značku Zákaz vjazdu všetkých vozidiel sme pomocou modulu `svgwrite` vytvorili nasledovne:

```
# ukazka.py
import svgwrite                                                    # [1]

svg_dokument = svgwrite.Drawing(filename='ukazka.svg',             # [2]
                                size=('10cm', '10cm'),             # [3]
                                viewBox=('0 0 10 10'))             # [4]

svg_dokument.add(svg_dokument.circle(center=(5, 5),                # [5]
                                     r=4.5,
                                     fill='white',
                                     stroke='black',
                                     stroke_width='0.05'))
svg_dokument.add(svg_dokument.circle(center=(5, 5),
                                     r=4,
                                     fill='red'))

svg_dokument.add(svg_dokument.rect(insert=(1.5, 4.25),             # [6]
                                   size=(7, 1.5),
                                   fill='white'))

svg_dokument.save(pretty='True')                                   # [7]
```

[1] Import modulu `svgwrite`.

[2] Vytvoríme nový objekt triedy `Drawing`. Objekt tejto triedy slúži ako kontajner pre všetky SVG elementy, ktoré budeme neskôr do dokumentu vkladať. Okrem toho má prístupné metódy pre uloženie do súboru. Názov výsledného SVG obrázka definujeme parametrom `filename`.

[3] Definujeme veľkosť obrázka.

[4] Definujeme vnútorný súradnicový systém.

[5] Metóda `add` objektu triedy `Drawing` slúži na vkladanie SVG elementov do výsledného obrázka. V našom príklade takto do obrázka vkladáme jednoduché geometrické útvary: kružnice, obdĺžniky a pod. SVG element kružnicu vytvoríme ako objekt triedy `circle`. Atribúty kružnice definujeme pomocou argumentov triedy. Definovať by sme mali minimálne súradnice stredu a polomer. Zoznam ďalších SVG elementov nájdeme v dokumentácii na stránke https://svgwrite.readthedocs.io/en/master/classes/shapes.html [14. 9. 2020]

[6] Do SVG dokumentu vložíme obdĺžnik.

[7] Výsledný obrázok zapíšeme do súboru. Parametrom `pretty` s hodnotou `True` dosiahneme, že výsledný SVG dokument bude pekne formátovaný a bude sa nám ľahšie čítať. Pre strojové spracovanie je to ale nepodstatné.

Úloha 3

Vytvorte skript `ukazka_zakaz_statia.py`, ktorý vygeneruje SVG dokument popisujúci dopravnú značku Zákaz státia.

Úloha 4

Vytvorte funkciu `vytvor_terc()` pomocou ktorej budete môcť vytvárať rôzne lukostrelecké terče vo formáte SVG.

Prediskutujte, aké parametre definujú terč a ako hodnoty týchto parametrov vhodne reprezentovať.

Úloha 5

Funkciu `vytvor_terc()` z predchádzajúcej úlohy budú používať prevažne lukostrelci. Ak pri zadávaní údajov spravia chybu, program zrejme skončí s nejakou chybovou hláškou (neošetrenou výnimkou). Upravte funkciu `vytvor_terc()` tak, aby skontrolovala, či zadané údaje sú korektné a prípadne vyhodila nejakú zmysluplnú výnimku. Argumentom výnimky by mal byť text, na základe ktorého aj neskúsený používateľ zistí, kde a akú chybu spravil.

Uvažujte akých chýb sa môže zadávateľ dopustiť. Ošetrite tieto chyby.

Výkladový text

Aby sme nemuseli odchytávať rôzne typy výnimiek, definujme si vlastnú výnimku. Výhodou tohto postupu je aj to, že namiesto všeobecných výnimiek budeme pracovať s konkrétnymi výnimkami.

Výnimka v Pythone nie je nič iné ako objekt nejakej triedy výnimiek. Všetky typy výnimiek sú usporiadané v hierarchickej štruktúre od najvšeobecnejších po najkonkrétnejšie. Hierarchiu výnimiek nájdeme na https://docs.python.org/3/library/exceptions.html#exception-hierarchy [6. 6. 2019].

Výnimky sa síce volajú rôzne, resp. sú rôznych typov, ale pracuje sa s nimi rovnako. Na jednej strane chceme, aby sme mali rôzne výnimky, pretože vieme takto lepšie rozlíšiť o aký problém sa jedná. Na druhej strane chceme, aby sa s výnimkami dalo pracovať jednotne. Bolo by nepraktické, aby sme každú triedu výnimiek museli definovať úplne od začiatku. Pri objektovo orientovanom programovaní vieme definovať nové triedy tak, že využijeme už existujúce triedy. Tejto vlastnosti hovoríme dedičnosť. Nová trieda výnimiek sa síce bude volať inak, ale zdedí všetko to, čo má pôvodná, rodičovská trieda. Prípadne vieme jej možnosti rozšíriť definovaním ďalších metód alebo atribútov.

Výnimky, ktoré použijeme pri chybných stavoch počas generovania terča, môžeme definovať nasledovne:

```
class TercChyba(Exception):
    pass
```

V prípade nejakého problému výnimku vyhodíme nasledovne:

```
raise TercChyba('Bližší popis chyby')
```

Takto vyhodenú výnimku, vieme odchytiť nasledovne:

```
try:
    # nejake volanie funkcii na vytvaranie tercov
except TercChyba as chyba:
    print(chyba)
```

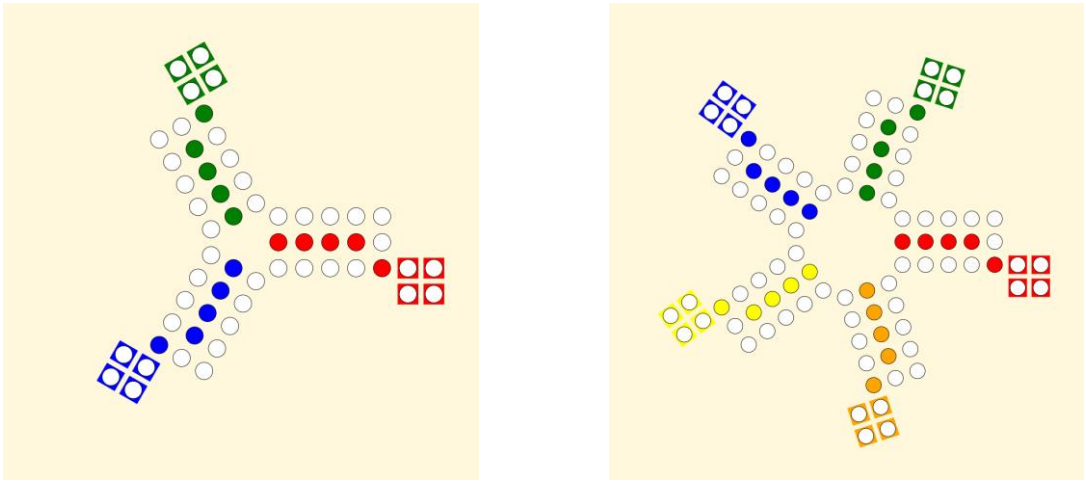
Vidíme, že aj s našou vlastnou výnimkou pracujeme rovnako ako s hocijakou inou výnimkou.

Ak si ešte raz pozrieme hierarchiu výnimiek vidíme, že najvšeobecnejšou výnimkou je `BaseException`. Z nej sú odvodené ďalšie konkrétnejšie a z nich ďalšie, ešte konkrétnejšie výnimky.

Pre definíciu vlastnej výnimky odporúčame použiť ako rodičovskú tú výnimku, ktorá pokrýva všetky typy chýb, ktoré chceme ošetriť. Bolo by zrejme nezmyselné, aby sme v našom prípade použili ako rodičovskú výnimku napr. `NameError`. Mohli by sme použiť aj `ValueError`, ale v prípade chyby pri zápise do súboru by to nedávalo veľký zmysel. V prípade, že typy chýb ktoré chceme ošetriť sú príliš rôznorodé, môžeme definovať viac typov vlastných výnimiek. Každú s iným rodičom.

Úloha 6

Spoločenskú hru „človeče, nehnevaj sa!“ pozná zrejme každý. Plánik hry je najčastejšie vytvorený pre 4 alebo 6 hráčov. Samozrejme, hrať môže aj menší počet, ale v tom prípade nemusia byť podmienky pre hráčov rovnocenné. Vytvorte program, pomocou ktorého je možné vytvoriť plánik hry vo formáte SVG pre zadaný počet hráčov (pozri obrázok nižšie).



Obrázok 90 Plánik hry Človeče, nehnevaj sa! pre troch a pre päť hráčov

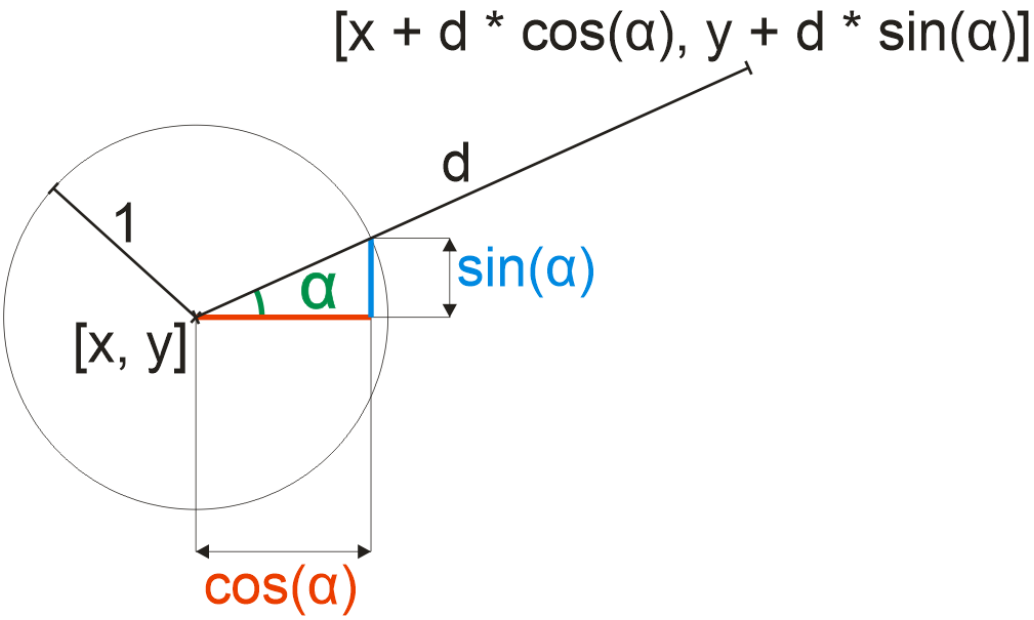
Pomôcka1: Podobné obrázky ste možno kreslili využitím korytnačej, relatívnej geometrie (modul `turtle`). Ak poznáme vzdialenosti jednotlivých krúžkov a uhly o ktoré sa pri kreslení otáčame, vykresliť obrázok pomocou korytnačej grafiky nie je komplikované.

Pri umiestňovaní objektov do SVG dokumentu však pracujeme s absolútnymi súradnicami. Vytvorte si pomocnú triedu `VirtualnePero()`*, pomocou ktorej sa budete virtuálne pohybovať po kresliacej ploche. Objekt tejto triedy bude udržiavať aktuálne virtuálne súradnice a natočenie pera počas jeho pohybu. Vždy keď sa počas kreslenia potrebujeme niekam posunúť alebo otočiť, posuňme/otočme pero a zistime si jeho súradnice/smer natočenia.*

Pri vytvorení objektu triedy `VirtualnePero()` *umiestnite pero na konkrétnu súradnicu a natočte pero konkrétnym smerom. Trieda by mala umožniť nasledovné:*

- *vrátiť aktuálne súradnice a smer natočenia pera,*
- *posunúť pero o zadanú dĺžku vpred a vzad v smere jeho natočenia,*
- *otočiť pero o zadaný uhol vľavo a vpravo.*

Pri prepočte súradníc po pohybe pera môžete využiť poznatky z matematiky (pozri obrázok nižšie).



Obrázok 91 Zmena súradníc bodu pri jeho posunutí v rovine o danú dĺžku a daným smerom

Pomôcka2: Zamyslime sa nad tým, ako zvoliť veľkosť políčka tak, aby sa celý plánik vošiel do kresliacej plochy. Spravme si približný výpočet polomeru hracieho políčka. Pomôžme si jednoduchou matematikou (pozri obrázok nižšie).

Predpokladajme, že každé hracie políčko má polomer `r_policko` *a susedné políčka sú od seba vzdialené o polomer políčka.*

Zamerajme sa na čierny kruh, ktorý spája políčka najbližšie k stredu. Polomer tohto vnútorného kruhu by sme vedeli približne vyjadriť ako (zo vzorca pre obvod kruhu `o = 2πr`*):*

```
r_vnutorny = (pocet_hracov * 3 * 3 * r_policko) / (2 * π)
```

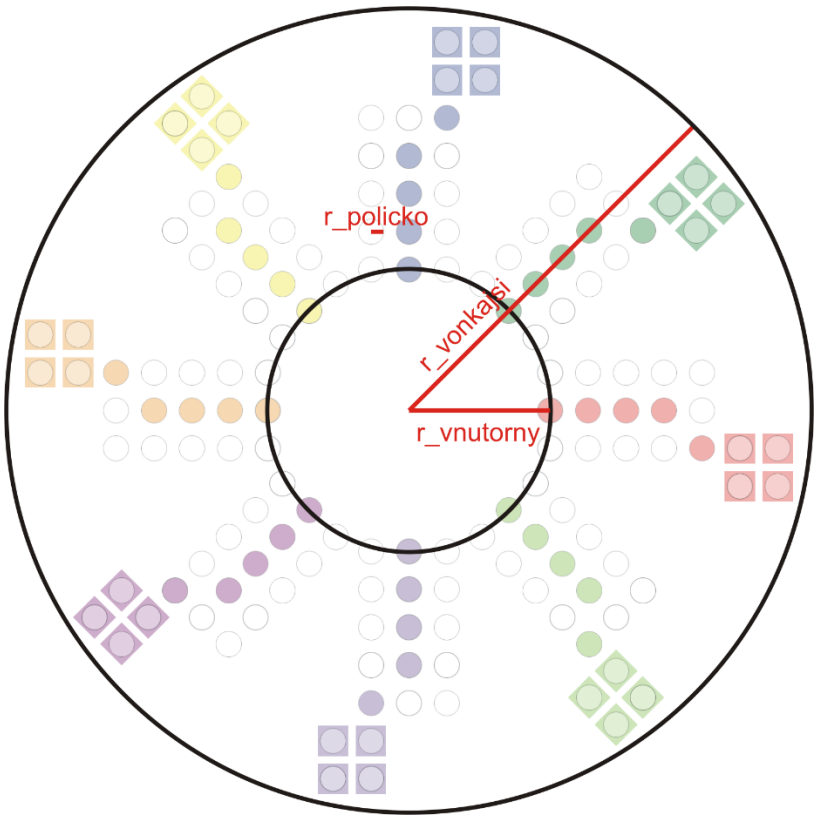
Ak chceme celý hrací plán umiestniť do oblasti 100 x 100 (takto si môžeme definovať viewbox v svg), tak polomer vonkajšieho kruhu by mohol byť napr. 40 (a z každej strany ostáva rezerva 10). Polomer vonkajšieho kruhu získame aj tak, že k polomeru vnútorného kruhu pripočítame:

`6 * 3 * r_policko` *(~ 6 hracie políčko aj s medzerou), teda:*

`40 = r_vnutorny + 6 * 3 * r_policko`.

Po úprave dostaneme hodnotu pre polomer hracieho políčka:

`r_policko = 40 / ((pocet_hracov * 3 * 3)/(2 * π) + 3 * 6)`. Výpočet je len približný, ale pre naše potreby dostačujúci.



Obrázok 92 Propozície plánu hry Človeče, nehnevaj sa.

Čo sme sa naučili

- programovať obrázky vo formáte SVG,
- inštalovať moduly jazyka Python,
- pracovať s modulom svgwrite,
- definovať vlastné výnimky,
- využívať vlastnosť objektového programovania – dedičnosť.

Ďalšie úlohy na precvičenie a zamyslenie

1. Aké typy obrázkov je výhodnejšie kresliť „ručne“ v grafickom editore a aké je výhodnejšie programovať?
2. Vytvorte hrací plánik pre hru Piškvorky. Existuje niekoľko verzií tejto hry. Vytvorte program/funkciu na kreslenie plánikov pre rôzne typy hry:
   - Hracia štvorcová sieť pokrýva celý hrací papier.
   - Hrá sa vo štvorcovej sieti 15×15. Takýchto sietí je na jednom papieri niekoľko.
   - Hrá sa vo štvorcovej sieti 5×5. Takýchto sietí je na jednom papieri niekoľko.
   - Hrá sa vo štvorcovej sieti 3×3. Takýchto sietí je na jednom papieri niekoľko.
3. Turnajový pavúk je schéma, do ktorej sa zaznamenáva priebeh turnaja. Z každej dvojice postupuje do ďalšieho kola víťaz. Vytvorte program/funkciu na kreslenie turnajových pavúkov pre rôzne počty hráčov.
4. Úlohu 3 vyriešte aj pre počet hráčov, ktorý nie je mocninou čísla 2. Premyslite si, ako budete kombinovať víťazov predchádzajúcich kôl s hráčom, ktorý v predchádzajúcom kole nehral.

21. Analýza logovacieho súboru

Kľúčové slová

logovací súbor, webový server, web log mining, virtuálne vzdelávacie prostredie, množina

Čo sa naučíme a čo si precvičíme

- vysvetliť význam logovacieho súboru webového servera pre tvorcov a administrátorov webu,
- analyzovať formát a obsah logovacieho súboru webovej aplikácie (virtuálneho vzdelávacieho prostredia),
- vymenovať vlastnosti údajovej štruktúry množina,
- používať množinu a operácie s množinami pri spracovaní údajov.

Problémová situácia

V školách (možno aj v tej vašej) používame často na podporu vyučovania **virtuálne vzdelávacie prostredie**. Obvykle ide o webovú aplikáciu, v ktorej má registrovaný používateľ prístup k **e-learningovým kurzom.** Autor kurzu alebo administrátor systému majú k dispozícii logovací súbor so záznamom činností používateľov, ktorý je možné analyzovať. **Čo všetko sa môžeme o používateľoch kurzu z neho dozvedieť?**

Poznámka na okraj

Príkladom veľmi populárneho virtuálneho vzdelávacieho prostredia, ktorého logovací súbor budeme v tejto kapitole analyzovať, je **Moodle**.

Kurzom rozumieme väčšinou webovú stránku určenú pre konkrétny predmet. Obsahuje dôležité informácie, študijné materiály, zadania úloh, testy, diskusné fóra a pod.

Webový server okrem skutočných používateľov navštevujú aj roboty (= programy), ktoré sa o web zaujímajú z rôznych dôvodov. *Googlebot* navštevuje stránky webu s cieľom zbierať aktuálne informácie pre vyhľadávač Google. Iné roboty môžu zbierať zo stránok e-mailové adresy, ktoré sa zneužijú na odosielanie spamu.

Hľadajme riešenie

Virtuálne vzdelávacie prostredie je príkladom webovej aplikácie, ktorej používatelia nie sú anonymní. Tiež je však umiestnená **na webovom serveri**, podobne ako stránky webov, po ktorých denne surfujeme bez registrácie. Webový server automaticky vytvára svoj **logovací súbor**. O každej požiadavke prijatej od klienta (najčastejšie máme na mysli prehliadač, ktorý používateľ použil) sa do logovacieho súboru zapíše riadok s dôležitými údajmi (*IP adresa klienta*, *dátum a čas prístupu*, *URL požadovanej stránky, resp. súboru, návratový kód, počet prenesených bajtov, odkazujúca stránka, verzia prehliadača*). Logovací súbor môže byť veľmi veľký (uvidíme to aj v Úlohe 1). Uvedomme si, čo všetko si od webového servera „vypýtame“ jedným kliknutím na hypertextový odkaz (nielen text stránky, ale tiež súbory s obrázkami, štýlmi, skriptami a pod., ktoré príslušná stránka obsahuje). Virtuálne vzdelávacie prostredie

ukladá do databázy niektoré údaje, ktoré zistí od webového servera (napr. IP adresu), ale aj ďalšie užitočné údaje o prihlásených používateľoch a ich aktivite v kurze.

Výkladový text

Webové stránky sú zdrojom informácií pre používateľov. Aj tvorca či administrátor webového sídla (resp. webovej aplikácie) sa môžu dozvedieť veľa o používateľoch. Majú k dispozícii **veľký objem dát** zaznamenaný v logovacom súbore webového servera (príp. aj v databáze webovej aplikácie). Obrovské množstvo zozbieraných dát má však **malú výpovednú hodnotu**.

Dolovanie znalostí z dát (*data mining*) chápeme ako proces, ktorý zahŕňa výber dát, transformáciu dát, analýzu dát a interpretáciu výsledkov. **Web log mining** sa zameriava na analýzu správania sa používateľov pri prechádzaní webu. Okrem štatistík o návštevnosti stránok môžeme identifikovať typické správanie sa používateľov a **objaviť pravidlá užitočné** pre ďalšie rozhodovanie. Na základe týchto znalostí môžeme napr. opraviť navigačné chyby, odstrániť z webu nepotrebný obsah, prispôsobiť jeho obsah a štruktúru potrebám a záujmom špecifických skupín používateľov, naplánovať vhodný čas údržby, zlepšiť bezpečnosť a výkonnosť systému (109).

Úloha 1

Prezrite si textový súbor *access.log*, ktorý pochádza z webového servera univerzity. Odpovedzte na uvedené otázky:

- Z akého časového obdobia pochádzajú záznamy o prístupoch používateľov na stránky?
- Koľko riadkov obsahuje?
- Aké údaje webový server do súboru zaznamenáva?
- Vyhľadajte prvý prístup používateľa s IP adresou 85.216.143.202. Prezrite si aj nasledujúce riadky s rovnakou IP adresou. Čo obsahujú?
- Odhadnite, aký veľký by bol logovací súbor s údajmi za 1 mesiac.

Úloha 2

Logovací súbor *logs_pus2017.csv* obsahuje záznamy súvisiace s činnosťou študentov a učiteľov v e-learningovom kurze zameranom na programovanie. Vytvorte program, v ktorom zobrazíte riadky logovacieho súboru na obrazovke. Potom preskúmajte formát a obsah logovacieho súboru detailnejšie.

S textovým súborom sme už pracovali, obsah logovacieho súboru prečítame a zobrazíme celý na obrazovke. Nezabudnime upresniť použité kódovanie:

```
with open('logs_pus2017.csv', 'r', encoding='UTF-8') as subor:
    print(subor.read())
```

Pri analýze formátu sa sústredíme iba na jeden riadok súboru:

```
with open('logs_pus2017.csv', 'r', encoding='UTF-8') as subor:
    riadok = subor.readline()                                    #[1]
    riadok = subor.readline()
    print(riadok)                                                #[2]
    print(repr(riadok))                                          #[3]
```

[1] Prvý riadok súboru obsahuje názvy polí tvoriacich záznam logovacieho súboru, môžeme ho preskočiť.
[2] Vo výpise na obrazovku nemusí byť jednoznačne vidieť, ktorý oddeľovač sa v súbore používa.
[3] Funkcia `repr()` nám pomôže uvidieť vo výpise skutočný obsah reťazca vrátane tabulátorov, znakov konca riadku či iných špeciálnych znakov.

Otázka

Náš logovací súbor obsahuje záznamy o udalostiach v kurze zozbierané v priebehu štyroch mesiacov. Koľko riadkov má?

Výkladový text

Pri parsovaní (analyzovaní) reťazca použijeme jeho metódu `split()`.

Metóda `split()` vracia zoznam častí, ktoré sú v parsovanom reťazci oddelené znakom `str`. Ak vo volaní metódy hodnotu parametra `str` neurčíme, považuje sa za oddeľovač ľubovoľný biely znak alebo ich postupnosť.

```
retazec1 = 'Abeceda zjedla deda,'
retazec2 = 'povedala\tna\tmedveďa.'
retazec3 = 'Medveď;na;líšku,;líška;na;pipíšku.'

zoznam1 = retazec1.split()                                  #[1]
zoznam2 = retazec2.split()
zoznam3 = retazec3.split(';')


def vypis(z):
  for prvok in z:
      print(prvok)

vypis(zoznam1)
vypis(zoznam2)
vypis(zoznam3)
print(len(zoznam3))                                         #[2]
```

[1] V druhom reťazci je oddeľovačom tabulátor (znak `'\t'`). Môžeme, ale nemusíme ho uviesť ako parameter.

[2] Čo sa vypíše na obrazovku na tomto mieste?

Výstup programu z vyššie uvedeného výkladového textu:

```
Abeceda
zjedla
```

```
deda,
povedala
na
medveďa.
Medveď
na
líšku,
líška
na
pipíšku.
6
```

Vráťme sa znovu k riešeniu **Úlohy 2**. Dokončíme parsovanie reťazca predstavujúceho jeden záznam v logovacom súbore:

```
# riadok prečítaný zo súboru rozdelíme na časti, oddeľovačom je ';'
zoznam = riadok.split(';')
print(len(zoznam))
#vypíšeme všetky prvky zoznamu
for i in range(len(zoznam)):
    print(f'prvok s indexom {i}: {zoznam[i]}')
```

Úloha 3

Už sme zistili, že v logovacom súbore *logs_pus2017.csv* sú jednotlivé časti každého záznamu oddelené bodkočiarkami, napr.:

```
02.10.2017 17:12;Súbor: 01x06 Cyklus;Zdroj;Zobrazené moduly
kurzu;217.144.26.164
```

Vybraný záznam môžeme interpretovať takto: 2. 10. 2017 o 17:12 používateľ z IP adresy 217.144.26.164 klikol na odkaz na súbor so študijným materiálom týkajúcim sa témy Cyklus, čím tento zdroj zobrazil.

Poznámka na okraj

Z pôvodného logovacieho súboru exportovaného zo systému Moodle sme odstránili niektoré stĺpce prezrádzajúce identitu skutočných používateľov.

Používatelia s rolou *Učiteľ* môžu v kurze vykonávať aj iné aktivity ako ich študenti (upravovať obsah kurzu, hodnotiť odovzdané zadania a pod.).

Z piatich získaných údajov nás bude zaujímať IP adresa používateľa. V zozname vrátenom metódou `split()` pôjde zrejme o prvok **s indexom 4**. **Z koľkých rôznych IP adries sa do kurzu pristupovalo? Ktoré to boli?**

V súbore sa určite veľmi často opakujú riadky s rovnakou IP adresou, keďže používatelia po vstupe do kurzu obvykle klikajú na viac zdrojov a aktivít (hľadajú informácie, študujú, riešia

úlohy a pod.). Opakované výskyty tej istej IP adresy však nie sú pre nás v tejto úlohe podstatné, nepotrebujeme si ich pamätať. Tiež sa nepýtame na chronologické poradie prístupov. Ak by sme objavené IP adresy priebežne ukladali do zoznamu, museli by sme pred ich pridaním kontrolovať, či príslušnú IP adresu už v zozname nemáme. Aby sme sa tomu vyhli, použijeme pre náš problém vhodnejšiu údajovú štruktúru – množinu:

Výkladový text

Jazyk Python obsahuje štandardný údajový typ **set** reprezentujúci množinu. **Prvky sa v množine neopakujú, na ich poradí nezáleží**. Hovoríme, že množina je neusporiadaná kolekcia jedinečných prvkov.

Prvky množiny musia byť **nemenné typy** (*immutable*), napr. reťazce, čísla, n-tice. Nemenným typom nie je napr. zoznam. Nemôžeme preto pracovať s množinou zoznamov.

Pri práci s množinami používame rôzne metódy a množinové operácie, ukážeme si tie najdôležitejšie:

```
m = set()                                                    #[1]
m.add('Adam')                                                #[2]
m.add('Boris')
m.add('Cyril')
m.add('Boris')                                               #[3]
print(m)                                                     #[4]
print(len(m))
m.discard('Boris')                                           #[5]
print('Boris' in m)                                          #[6]
```

[1] Vytvorili sme prázdnu množinu.

[2] Do množiny sme postupne pridali 3 prvky.

[3] Reťazec `'Boris'` sa už v množine `m` nachádza, volanie metódy `add()` je preto bez účinku.

[4] Vypísali sme celú množinu a počet jej prvkov.

[5] Prvok `'Boris'` sme z množiny odstránili.

[6] Prvok sa v množine nenachádza, vypíše sa `False`.

S množinami môžeme pracovať aj pomocou množinových operátorov:

```
m1 = {1, 2, 3, 4}                                            #[7]
m2 = {4, 5}
zjednotenie = m1 | m2                                        #[8]
prienik = m1 & m2
rozdiel = m1 - m2
print(zjednotenie, prienik, rozdiel)
m3 = {5, 1, 3, 4, 2}
print(zjednotenie == m3)                                     #[9]
print(m2 < m3)                                               #[10]
```

[7] Vytvorili sme dve množiny vymenovaním ich prvkov.

[8] Vytvorili sme množiny, ktoré sú zjednotením, prienikom a rozdielom množín `m1` a `m2`.

[9] Množiny `m3` a `zjednotenie` obsahujú rovnaké prvky, vypíše sa `True`.

[10] Množina `m2` je podmnožinou množiny `m3`, vypíše sa `True`.

Poznámka na okraj

Pozor, príkazom

```
m = {}
```

nevytvoríme prázdnu množinu, ale prázdny slovník, čo ľahko overíme vypísaním typu premennej:

```
print(type(m))
```

V skripte *mnoziny.py* nájdete príklady použitia ďalších metód a operátorov užitočných pri práci s množinami.

S využitím množiny už ľahko zistíme odpovede na obe otázky z **Úlohy 3**. Pri parsovaní riadkov budeme IP adresy ukladať do množiny. Na záver vypíšeme jej veľkosť aj obsah.

Úloha 4

Existujú IP adresy, z ktorých sa do kurzu používatelia prihlásili iba jediný raz? Ak áno, zapíšte všetky unikátne IP adresy do súboru *unikaty.txt*.

Pomôcka: Pri riešení tejto úlohy vám môžu pomôcť **množinové operácie**.

Čo sme sa naučili

- Web log mining sa zaoberá aktuálnym problémom – spracovaním veľkého objemu automaticky generovaných dát s cieľom objaviť užitočné znalosti.
- Logovací súbor webového servera má textový formát, jeden riadok súboru obsahuje údaje o jednej požiadavke klienta. Aj logovací súbor virtuálneho vzdelávacieho prostredia má textový formát, obsahuje údaje získané z databázy. Mnohé zo zapísaných údajov pochádzajú práve od webového servera.
- Na jednoduché parsovanie reťazca môžeme v jazyku Python použiť metódu `split()`.
- Pri riešení problémov súvisiacich s testovaním príslušnosti prvku do množiny, filtrovaním duplicít či realizovaním množinových operácií môžeme v jazyku Python použiť štandardný údajový typ `set`.

Ďalšie úlohy na precvičenie a zamyslenie

1. V októbri a v novembri sa študenti zúčastnili vo viacerých termínoch (v rôznych dňoch a v rôznych hodinách) praktického testu v počítačovej učebni. Z logovacieho súboru zistite, koľko študentov bolo pravdepodobne v jednotlivé dni prítomných v škole. Pokúsil sa niekto v čase testu o prístup z iného ako školského počítača? (napr. s cieľom podvádzať).

   Pomôcka: Budú nás zaujímať záznamy s kontextom udalosti `'Test: Praktický test 1'` a názvom udalosti `'Zahájený pokus'`. IP adresy školských počítačov začínajú prefixom `'10.160.3'`

2. Preskúmajte implementáciu triedy v súbore *vlastna_mnozina.py* a otestujte ju. Dala by sa použiť aj v riešení predchádzajúcich úloh? Čo má spoločné a čím sa odlišuje od štandardného údajového typu `set`?

22. Index textového dokumentu

Kľúčové slová

textový dokument, slovo, slovník, hešovacia tabuľka, vyhľadávanie slov v textovom dokumente

Čo sa naučíme a čo si precvičíme

- zopakujeme si prácu s údajovou štruktúrou slovník,
- preskúmame podrobnejšie implementáciu údajovej štruktúry slovník,
- z textového dokumentu získame všetky slová, ktoré obsahuje,
- vytvoríme index so všetkými slovami a s vybranou podmnožinou slov,
- naprogramujeme aplikáciu na efektívne vyhľadávanie v lokálnom úložisku textových dokumentov,
- v používateľskom rozhraní aplikácie použijeme komponent `Listbox` z modulu `tkinter`.

Problémová situácia 1

V textových dokumentoch (najmä v odborných publikáciách) nám pri vyhľadávaní pomáha index (register) pojmov. Ide o usporiadaný zoznam slov vybraných autormi publikácie (významných v kontexte spracovanej problematiky) s uvedením čísel strán, na ktorých sa vyskytujú. **Napíšeme program na automatické vytvorenie indexu pre zvolený textový dokument**.

Hľadajme riešenie

Textovým dokumentom budeme v celej kapitole rozumieť obyčajný textový súbor s kódovaním UTF-8. Budeme pracovať s textami v slovenskom jazyku. Nebudeme sa zaoberať extrahovaním čistého textu z dokumentov, ktoré sú v inom formáte (napr. *html*, *doc*, *docx*, *rtf*, *pdf*).

V postupnosti znakov zapísaných v súbore musíme najprv identifikovať slová a zapamätať si, kde sa nachádzajú. Pri spracúvaní čistého textu bez formátovania nemáme v súbore k dispozícii informáciu o čísle strany. Pre slová si preto najprv zapamätáme iba poradové čísla riadkov, v ktorých sa vyskytujú. Na uloženie tejto informácie môžeme využiť **údajovú štruktúru slovník**. Pamätáte si, ako sa so slovníkom pracuje v Pythone? Pripomenieme si to v nasledujúcej úlohe:

Úloha 1

V nižšie uvedenom programe vypočítame molovú hmotnosť chemických zlúčenín zadaných vzorcom. Údaje o hmotnostiach chemických prvkov pochádzajú z textového súboru *prvky.csv*:

```
Hydrogen,1,H,1.01
Helium,2,He,4.00
Lithium,3,Li,6.94
Beryllium,4,Be,9.01
Boron,5,B,10.81
```

```
Carbon,6,C,12.01
...
```

Na uloženie hmotností chemických prvkov používame slovník. **Kľúčom k hodnote hmotnosti** prvku je jeho chemická značka.

Vo vzorci zlúčeniny používame ako oddeľovač prvkov bodku, napr.: H_2SO_4 napíšeme na vstupe ako `H2.S.O4`. Pomôže nám to pri parsovaní vzorca.

Do triedy `HmotnostiPrvkov` doplňte na vynechané miesta správne príkazy:

```
class HmotnostiPrvkov:
    '''Tabuľka s hmotnosťami chemických prvkov.'''
    def __init__(self, subor):
        '''Premennú typu dict naplní údajmi zo vstupného súboru.'''
        self.__slovnik = {}
        with open(subor, 'r', encoding='UTF-8') as f:
            riadky = f.readlines()
            for riadok in riadky:
                zoznam = riadok.split(',')[2:]
                prvok = zoznam[0]
                hmotnost = float(zoznam[1])
                [                                    ]

    def __str__(self):
        '''Vráti tabuľku ako reťazec.'''
        return self.__slovnik.__str__()

    def obsahuje_prvok(self, prvok):
        '''Otestuje, či je daný prvok v tabuľke.'''
        return [                                    ]

    def get_hmotnost(self, prvok):
        '''Vráti hmotnosť prvku alebo 0.0 (ak prvok neexistuje).'''
        try:
            return [                                    ]
        except KeyError:
            return 0.0

class Kalkulacka:
    '''Kalkulačka na výpočet molovej hmotnosti zlúčeniny.'''
    def __init__(self):
        '''Vytvorí objekt triedy HmotnostiPrvkov.'''
        self.__tabulka = HmotnostiPrvkov('prvky.csv')

    def pocitaj(self, vzorec):
        '''Parsuje vstupný vzorec a vypočíta molovú hmotnosť.'''
        vysledok = 0.0
        prvky = vzorec.split('.')
        cislice = '012345678'
```

```
        for prvok in prvky:
            i = len(prvok)-1
            while i>=0 and prvok[i] in cislice:
                i -= 1
            hmotnost = self.__tabulka.getHmotnost(prvok[:i+1])
            if prvok[i+1:] != '':
                hmotnost *= int(prvok[i+1:])
            vysledok += hmotnost
        return round(vysledok, 2)

# hlavný program (predpokladáme korektný vstup)
kalkulacka = Kalkulacka()
print(kaklulacka.pocitaj('H2.O'))     # voda
print(kalkulacka.pocitaj('H.Cl'))     # kyselina chlorovodíková
print(kalkulacka.pocitaj('H2.S.O4')) # kyselina sírová
print(kalkulacka.pocitaj('C.O2'))     # oxid uhličitý
```

Otázky

Ktorá z molekúl je najľahšia?

Vysvetlite, ako kalkulačka spracúva vstupný vzorec.

Možnosť indexovať prvky inak ako číslom (v tomto prípade textovým reťazcom) nám pomáha písať prehľadnejšie programy. Hlavnou výhodou slovníka je však **efektívnosť vyhľadávania**:

Predstavme si, že by sme údaje z predchádzajúcej úlohy uložili **do zoznamu** a potrebovali by sme zistiť napr. hmotnosť atómu zlata. Zoznam by sme začali prehľadávať **sekvenčne**, t. j. prvok po prvku od jeho začiatku, v najhoršom prípade až po jeho koniec. Teraz našťastie skončíme s prehľadávaním skôr. Dvojicu `('Au',196.97)` nájdeme na pozícii s indexom 78. Program by teda pri vyhľadávaní vykonal 79 porovnaní.

Poznámka na okraj

V Mendelejevovej periodickej sústave prvkov má zlato (*Aurum*) značku *Au* a jeho atómové číslo je 79. V zozname má prvý prvok index 0.

V slovníku sa k hľadanej hodnote dostaneme oveľa rýchlejšie a to **priamym skokom** na to miesto tabuľky, kde sa príslušná dvojica nachádza. Jediné porovnanie, ktoré je potrebné na vyhľadanie hodnoty v slovníku vykonať, overuje existenciu kľúča. Umožňuje nám to dômyselná implementácia slovníkovej údajovej štruktúry založená na hešovaní.

Výkladový text

Hešovacia funkcia priraďuje každému kľúču celé číslo, tzv. **heš** (angl. *hash*). Toto číslo sa použije na výpočet indexu, na ktorom máme prvok v tabuľke hľadať. V prípade, že je na vypočítanej pozícii prázdne miesto, hľadaný prvok v tabuľke nie je.

Zostrojiť dobrú hešovaciu funkciu je náročné (my ju máme k dispozícii v knižnici Pythonu). **Hešovacia funkcia musí byť jednoduchá na výpočet a vymyslená tak, aby pre rovnaké kľúče vracala vždy rovnaký heš. Pre rôzne kľúče musí naopak vrátiť rôzne heše** (príp. len v zriedkavých prípadoch rovnaké). Tiež by mala garantovať, že tabuľka, do ktorej sa údaje na

vypočítané indexy ukladajú, bude pri pridávaní nových prvkov obsadzovaná rovnomerne a náhodne. Určite ste si všimli, že pri výpise prvkov uložených v slovníku nemôžeme očakávať žiadne konkrétne poradie.

V prípade, že hešovacia funkcia vráti pre niektoré dva nerovnaké kľúče rovnaký heš, vznikne tzv. kolízia (t. j. situácia, keď chceme dva prvky ukladať v tabuľka na rovnakú pozíciu, čo sa nedá). Na riešenie kolízií existuje viacero šikovných algoritmov. V Pythonovskej hešovacej tabuľke sa hľadá iné voľné miesto tabuľky (95).

Na obrázku vpravo sme znázornili stav údajovej štruktúry slovník (jeho hešovaciu tabuľku) z triedy `HmotnostiPrvkov` po vložení prvých šiestich prvkov.

Overme v konzole Pythonu:

```
>>> hash('H') & 7
2
>>> hash('He') & 7
7
>>> hash('Li') & 7
4
>>> hash('Be') & 7
3
>>> hash('B') & 7
1
>>> hash('C') & 7
6
```

index	prvok
0	
1	('B', 10.81)
2	('H', 1.01)
3	('Be', 9.01)
4	('Li', 6.94)
5	
6	('C', 12.01)
7	('He', 4.00)

Pythonovská funkcia `hash()` vracia celé čísla (vyskúšajte aké). Pri výpočte indexu v tabuľke sa okrem hešu používa aj hodnota vyjadrujúca jej aktuálnu veľkosť zmenšená o 1 (na začiatku 7). Operátor `&` je bitový súčin. Pri vkladaní uhlíka sa pre kľúč `'C'` vypočítal index 6, preto bude uložený na pozíciu 6. Pri jeho vyhľadávaní sa pre kľúč `'C'` opäť vypočíta rovnaký index. Skontroluje sa, či prvok v tabuľke je, ak áno, vráti sa ako nájdený.

Hešovacia tabuľka použitá pri implementácii slovníka v Pythone má na začiatku veľkosť 8. Zväčšuje sa podľa potreby a to v prípade, že je obsadených viac ako 2/3 pozícií. V našom príklade sú riadky tabuľky s indexom 0 a 5 zatiaľ prázdne.

Poznámka na okraj

Bitový súčin je jedna z bitových operácií, ktorú môžeme realizovať s celými číslami, napr.:

```
>>> 155 & 114
18
```

Zapíšme zvolené čísla binárne:

$(155)_{10} = (1001\ 1011)_2$
$(114)_{10} = (0111\ 0010)_2$

Sledujme bity na rovnakých pozíciách a zapíšme binárnu reprezentáciu výsledku operácie &. Ak sú na rovnakej pozícii v oboch číslach jednotky, aj vo výsledku bude príslušný bit nastavený na hodnotu 1. Inak bude nulový:

  100**1** 10**1**1
& 0**1**1**1** 00**1**0
  000**1** 00**1**0

Je zrejmé, že sme dostali rovnaký výsledok ako v konzole: $(0001\ 0010)_2 = (18)_{10}$

Úloha 2

Vrátime sa teraz k pôvodnému problému. Z textového súboru chceme čítať postupne slová. Slová v riadkoch nebývajú oddelené iba medzerami, tabulátormi či koncami riadkov (t. j. bielymi znakmi). Texty bežne obsahujú aj interpunkciu (čiarky, bodky, otázniky, výkričníky, zátvorky, úvodzovky a iné). A tie nie sú súčasťou slov. Napíšte skript, v ktorom otvoríte vstupný textový súbor na čítanie a zobrazíte všetky jeho slová. Slová vypisujte malými písmenami.

Na otestovanie môžeme použiť súbor *vstup.txt* s jediným riadkom:

```
'Som "testovaci text", v ktorom je vela slov, vratane interpunkcie. Kludne ma zmente!\n'
```

Uvedené riešenie nie je správne. Prečo?

```
with open('vstup.txt', 'r', encoding='UTF-8') as f:
    riadok = f.readline()
    while riadok:
        slova = riadok.split()
        for slovo in slova:
            print(slovo.lower())
        riadok = f.readline()
```

Niektoré reťazce, ktoré sa objavia vo výpise, nie sú slovami. K postupnosti znakov tvoriacich slovo z textu sa totiž pridali aj znaky interpunkcie:

```
som
"testovaci
text",
v
ktorom
je
vela
slov,
vratane
```

```
interpukncie.
kludne
ma
zmente!
```

Výkladový text

Volaním `riadok.split()` získame zoznam častí, ktoré sú v parsovanom reťazci oddelené bielymi znakmi. V parametri metódy `split()` môžeme špecifikovať ako oddeľovač aj nejaký iný znak, ale nie viacej znakov.

Znaky, ktoré považujeme za interpunkciu, môžeme v programe vymenovať sami. V module `string` sa však nachádza užitočná konštanta:

```
punctuation = r"""!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~"""
```

Vo svojom skripte musíme modul `string` najprv importovať:

```
import string
```

Potom môžeme konštantu použiť, napr. vypísať:

```
print(string.punctuation)
```

Nesprávne riešenie sa dá ľahko opraviť. Ak riadok (zbavený prebytočných bielych znakov na začiatku a na konci) obsahuje nejaké znaky interpunkcie, nahradíme ich ešte pred volaním metódy `split()` medzerami.

Na tento účel bude slúžiť funkcia `bez_interpunkcie()`:

```
def bez_interpunkcie(retazec):
    ''' Funkcia vráti nový reťazec, v ktorom sú znaky interpunkcie
        nahradené medzerami.'''
    pom = ''
    interpunkcia = string.punctuation + '„"'
    for znak in retazec:
        if not znak in interpunkcia:
            pom += znak
        else:
            pom += ' '
    return pom
```

Poznámka na okraj

Slovenské texty často obsahujú úvodzovky. Ide o iné znaky ako tie, ktoré sa používajú v angličtine, preto sme reťazec `string.punctuation` doplnili.

Úloha 3

Vytvorte vlastnú triedu `IndexDokumentu`, pomocou ktorej budete schopní pre textový dokument a ľubovoľné jeho slovo zistiť, v ktorých riadkoch dokumentu sa vyskytuje.

Uvažujme vstupný textový súbor s 9 riadkami uvedený nižšie. Vymenujeme všetky slová, ktoré tento text obsahuje. Vedľa zapíšeme čísla riadkov, v ktorých sa jednotlivé slová vyskytujú.

Vstupný text:

Poslali ma naši k vašim,
aby prišli vaši k našim.
ak neprídu vaši k našim,
tak neprídu naši k vašim.

Poslali ma, poslali,
k vašim, k vašim,
neprišli ste, neprišli,
k našim, k našim.

slovo	čísla riadkov
poslali	1, 6, 6
ma	1, 6
naši	1, 4
k	1, 2, 3, 4, 7, 7, 9, 9
vašim	1, 4, 7, 7
aby	2
prišli	2
vaši	2, 3
našim	2, 3
ak	3
neprídu	3, 4
tak	4
neprišli	8, 8
ste	8

V triede `IndexDokumentu` budeme v slovníku mapovať slová z textu na čísla riadkov, v ktorých sa nachádzajú. Slovník bude obsahovať len také slová, ktoré sa v texte vyskytujú aspoň raz, do slovníka ich budeme pridávať priebežne pri čítaní textu zo vstupného súboru. **Akú údajovú štruktúru použijeme na uloženie čísel riadkov?**

Uvádzame dva nápady na riešenie. V prvom prípade použijeme zoznam, v druhom prípade množinu (pre množinu platia príkazy uvedené v komentároch). Je jedno, ktorú z možnosti si vyberieme?

```
class IndexDokumentu:
    '''Index slov v textovom súbore.'''
    def __init__(self, subor):
        '''Naplnenie slovníka slovami zo vstupného súboru.'''
        self.__slovnik = {}
        with open(subor, 'r', encoding='UTF-8') as f:
            riadok = f.readline()
            i = 0  # číslo spracúvaného riadku
            while riadok:
                i += 1
                riadok = self.bez_interpunkcie(riadok.strip())
                slova = riadok.split() # slová spracúvaného riadku
                for slovo in slova:
                    kluc = slovo.lower()
                    hodnota = self.__slovnik.get(kluc, set())    #[1]
                    hodnota.add(i)
```

```
                    #hodnota = self.__slovnik.get(kluc, [])      #[2]
                    #hodnota.append(i)
                    self.__slovnik[kluc] = hodnota               #[3]
                riadok = f.readline()
```

[1] V prvom prípade získame zo slovníka množinu čísel riadkov asociovanú s kľúčom alebo prázdnu množinu, ak také slovo v slovníku zatiaľ nemáme. Do množiny pridáme aktuálne číslo riadku metódou `add()`.

[2] Druhou možnosťou je použiť na uloženie čísel riadkov výskytov slova zoznam. Pri prvom objavení sa slova vracia metóda `get()` prázdny zoznam. Do zoznamu asociovaného so slovom pridáme číslo aktuálneho riadku metódou `append()`.

[3] Aktualizujeme prvok slovníka.

Poznámka na okraj

Nie je jedno, či budeme slová mapovať na zoznam čísel riadkov alebo na množinu čísel riadkov. Ak nás nezaujímajú prípadné opakované výskyty slova na jednom riadku, postačí nám množina. Pri práci s množinou však nesmieme zabudnúť, že neobsahuje žiadnu informáciu o poradí svojich prvkov. Pri vytváraní indexu zaujímavých slov budeme preto musieť postupnosť čísel riadkov pred jej spracovaním najprv usporiadať.

Doprogramujte aj ostatné metódy triedy `IndexDokumentu` v súbore *uloha3.py* a triedu otestujte:

```
class IndexDokumentu:
    '''Index slov v textovom súbore.'''
    def __init__(self, subor):
        '''Naplnenie slovníka slovami zo vstupného súboru.'''
        self.__slovnik = {}
        pass

    def bez_interpunkcie(self, retazec):
        ''' Vráti nový reťazec, v ktorom sú znaky interpunkcie
            nahradené medzerami.'''
        pass

    def __str__(self):
        '''Vráti obsah slovníka ako reťazec.'''
        return self.__slovnik.__str__()



    def obsahuje_slovo(self, slovo):
        '''Overí, či sa slovo nachádza v slovníku.'''
        return slovo in self.__slovnik.keys()

    def get_vyskyty(self, slovo):
        '''Pre zadané slovo vracia zoznam s číslami riadkov, na
           ktorých sa vyskytuje.'''
        try:
            return self.__slovnik[slovo]
        except KeyError:
```

```
            return set()

    def get_pocet_vyskytov(self, slovo):
        '''Pre zadané slovo vráti počet výskytov.'''
        try:
            return len(self.__slovnik[slovo])
        except KeyError:
            return 0

    def get_velkost_slovnika(self):
        '''Vráti počet prvkov slovníka.'''
        pass
```

Úloha 4

Na vyriešenie pôvodného problému nám ešte chýba vytvorenie indexu pre zvolenú množinu slov v podobe, v akej sa očakáva v tlačenej publikácii.

Na to, aby sme vedeli transformovať čísla riadkov na čísla strán, potrebujeme doplňujúcu informáciu: Pre jednoduchosť môžeme uvažovať, že na každej strane textového súboru je rovnaký počet riadkov, napr. 5. Číslo strany potom ľahko vypočítame delením.

Do triedy `IndexDokumentu` môžeme pridať metódu `vytlac_index()`, ktorej parametrom bude množina zaujímavých slov. Naprogramujte túto metódu a otestujte ju napísaním skriptu:

```
# testovací skript
index_dokumentu = IndexDokumentu('ranena_breza.txt')
print('INDEX DOKUMENTU')
zaujimave_slova = {'breza', 'deti', 'krv', 'mesiačik'}
index_dokumentu.vytlac_index(zaujimave_slova)
```

Výstup (s odsadením čísel strán vpravo):

```
INDEX DOKUMENTU
breza
     1,2,6
deti
     4
krv
     5
mesiačik
    1,3,5
```

Problémová situácia 2

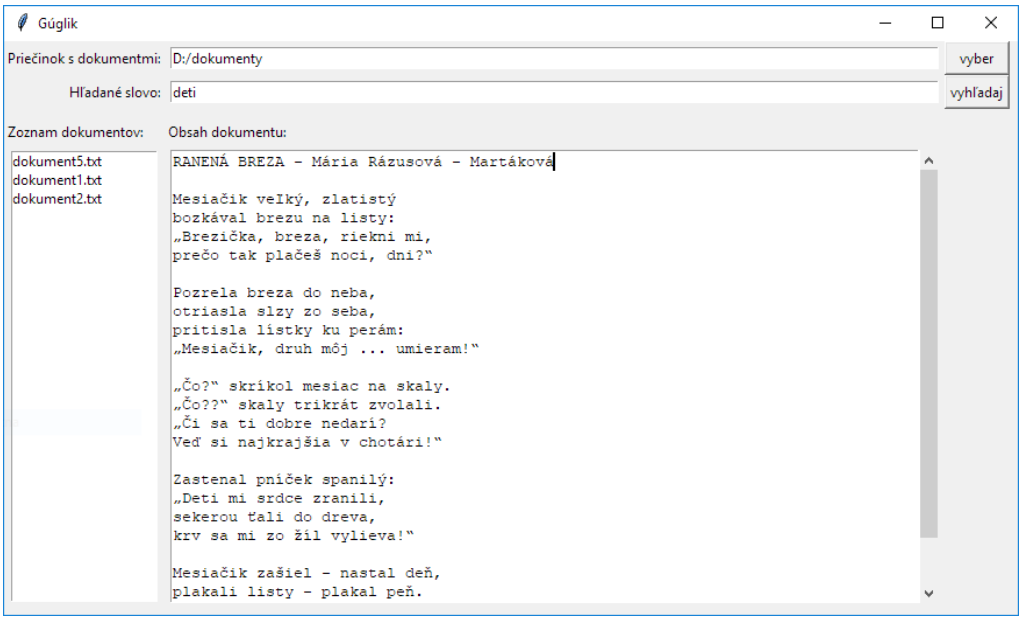
Pri vyhľadávaní informácií v elektronických zdrojoch, napr. na webe, očakávame po zadaní kľúčového slova zobrazenie zoznamu tých dokumentov, ktoré príslušné slovo obsahujú. Naprogramujeme **vlastný vyhľadávač**, ktorý nám pomôže pri opakovanom vyhľadávaní slov vo väčšej kolekcii textových dokumentov uložených na pevnom disku počítača.

Poznámka na okraj

Kritérium vyhľadávania by mohlo byť zložitejšie. Mohli by nás zaujímať len dokumenty v konkrétnom jazyku, z istého časového obdobia, príp. súčasne obsahujúce/neobsahujúce viaceré slová a slovné spojenia.

Pri vyhľadávaní na webe (napr. pomocou vyhľadávača Google) sa výsledok získava z údajov o indexovaných dokumentoch uložených v databázach.

Aplikácia môže mať okno rozdelené na dve časti. V ľavej časti budeme vidieť zoznam dokumentov (`Listbox`), ktoré obsahujú slovo zadané vo vyhľadávacom poli (`Entry`). V pravej časti textová plocha (`ScrolledText`) zobrazujúca obsah zvoleného dokumentu. Aplikácia nám umožní zvoliť priečinok, v ktorom sú uložené dokumenty, zadať slovo alebo slovné spojenie na vyhľadávanie (pozri obrázok nižšie).

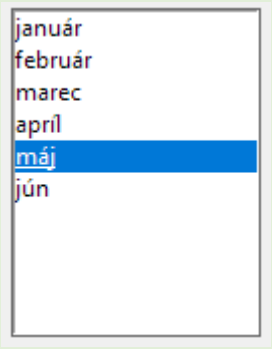


Obrázok 93 Grafické používateľské rozhranie aplikácie Gúglik

Na testovanie môžete použiť ľubovoľný textový súbor. Napr. digitalizované diela vybraných slovenských spisovateľov z projektu https://zlatyfond.sme.sk/ [30. 6. 2020].

Výkladový text

Listbox je vizuálny komponent z modulu `tkinter`. Zobrazuje zoznam hodnôt (textových reťazcov), z ktorých si vyberáme jednu alebo viac možností.



Prvá položka má index 0.

`Listbox` na obrázku vľavo obsahuje `6` hodnôt.

Hodnota s indexom `4` je označená. Obsahuje reťazec `'máj'`.

V nasledujúcich ukážkach práce s listboxom budeme predpokladať import celého modulu `tkinter`. Vyhneme sa tak opakovanému písaniu `tkinter` pri vytváraní komponentu a používaní konštánt:

V nasledujúcich ukážkach práce s listboxom budeme predpokladať import celého modulu `tkinter`. Vyhneme sa tak opakovanému písaniu `tkinter` pri vytváraní komponentu a používaní konštánt:

```
from tkinter import *
```

Pri vytváraní listboxu uvádzame rodičovské okno, v ktorom sa nachádza, a zoznam ďalších nastavení.

```
listbox = Listbox(rodičovské okno, nastavenia)
```

Do listboxu môžeme vkladať hodnoty na koniec alebo pred pozíciu určenú indexom:

```
mesiace = ['február', 'marec', 'apríl', 'jún']

for m in mesiace:
    listbox.insert(END, m)

listbox.insert(0,'január')

listbox.insert(5,'máj')
```

Výsledok po vkladaní názvov mesiacov ukazuje obrázok vľavo hore. Konštanta `END` má hodnotu `'end'`, mohli by sme používať aj tento reťazec.

Môžeme tiež zistiť aktuálny počet hodnôt:

```
pocet = listbox.size()
```

Hodnotu na konkrétnom indexe získame takto:

```
hodnota = listbox.get(index)
```

Hodnoty môžeme z listboxu odstraňovať. Môžeme zadať jeden index, ale i hranice úseku s hodnotami, ktoré treba odstrániť (vrátane indexov hraníc):

```
listbox.delete(index)
listbox.delete(index1, index2)
```

Pri vytváraní listboxu sa dá nastaviť vlastnosť `selectmode`. Ide o spôsob označovania hodnôt pomocou myši. Máme štyri možnosti – konštanty definované v module `tkinter`:

`BROWSE` pri ťahaní myšou sa označuje 1 aktuálny riadok (východiskové nastavenie)

`SINGLE` vyberáme tiež len 1 riadok, ale klikaním

`MULTIPLE` klikaním môžeme vybrať/zrušiť výber viacerých možností

`EXTENDED` označovanie súvislej postupnosti riadkov ťahaním

Ak chceme reagovať na udalosť „označenie hodnoty“ , zabezpečíme to takto:

```
listbox.bind('<<ListboxSelect>>', obsluzna_metoda)
```

Obslužná metóda má dva parametre, druhý reprezentuje vzniknutú udalosť:

```
def obsluzna_metoda(self, evt)
   w = evt.widget  # zdroj udalosti (komponent listbox)
   pass
```

Volaním metódy `oznacene = listbox.curselection()`
získame n-ticu s indexami označených hodnôt:

Uvažujme listbox so 6 riadkami, označujme ich pomocou myši. Ak označíme 1 riadok, metóda `curselection()` vráti napr. `(5,)`. Ak označíme 3 riadky, vráti napr. `(1,2,5)`. Ak nie je označené nič, vráti prázdnu n-ticu, t. j. : `()`.

Ak chceme údaje na zobrazenie uchovávať v premennej, musíme vlastnosť `listvariable` asociovať s premennou typu `StringVar`:

```
m = StringVar()
m.set('január február marec apríl máj jún')

listbox = Listbox(rodicovské okno, listvariable = m)

m.set('september október')
```

Pozor, metóda `m.get()` vracia reťazec v tvare:

```
"('september', 'október', 'november')"
```

Vlastnosti listboxu môžeme zmeniť aj dodatočne, po vytvorení (napr. farbu pozadia):

```
listbox.config(bg = 'yellow')
```

Úloha 5

V súbore *uloha5.py* nájdete čiastkové riešenie obsahujúce triedu `Okno` s inicializáciou grafického používateľského rozhrania. Doplňte do riešenia triedu, ktorá zabezpečí indexovanie slov pre dokumenty zo zvoleného priečinka. Doprogramujte tiež chýbajúce metódy v triede `Okno` (reakcie na stláčanie tlačidiel či označenie položky v zozname nájdených dokumentov).

Čo sme sa naučili

- Slovník je údajová štruktúra umožňujúca efektívne vyhľadávanie hodnôt asociovaných so zadaným kľúčom. V Pythone je slovník implementovaný ako hešovacia tabuľka.
- Ak má byť kľúč v slovníku asociovaný s viacerými hodnotami, na ich uloženie môžeme použiť zoznam alebo množinu.
- Komponent `ListBox` z modulu `tkinter` slúži na zobrazenie viacerých hodnôt (reťazcov), z ktorých si môže používateľ vybrať označením jednej alebo viacerých možností.

Ďalšie úlohy na precvičenie a zamyslenie

1. Upravte aplikáciu na vyhľadávanie v textových dokumentoch tak, aby sa v okne s obsahom zvoleného dokumentu zvýraznil prvý alebo všetky výskyty hľadaného slova.

   Pomôcka: V komponente `Text` alebo `ScrolledText` vieme nastavovať vlastnosti častí textu. Ak poznáme indexy začiatku a konca reťazca, môžeme ho označiť definovaním tagu pomocou metódy `tag_add()` a potom použiť metódu `tag_config()`. V jej parametroch definujeme farbu pozadia a popredia.

2. Ako by sme vyriešili požiadavku na zadávanie zložených kritérií vyhľadávania pomocou logických spojok (`AND`, `OR`, `NOT`)?

3. Pythonovský slovník sme už použili rôznym spôsobom. Mapovali sme reťazce na počty ich výskytov (t. j. na celé čísla), ale aj na zoznamy či množiny čísel riadkov, na ktorých sa v texte vyskytovali. Vymyslite problém, v riešení ktorého by sme na uloženie údajov mohli využiť zoznam slovníkov.

23. Myslíš toto?

Kľúčové slová

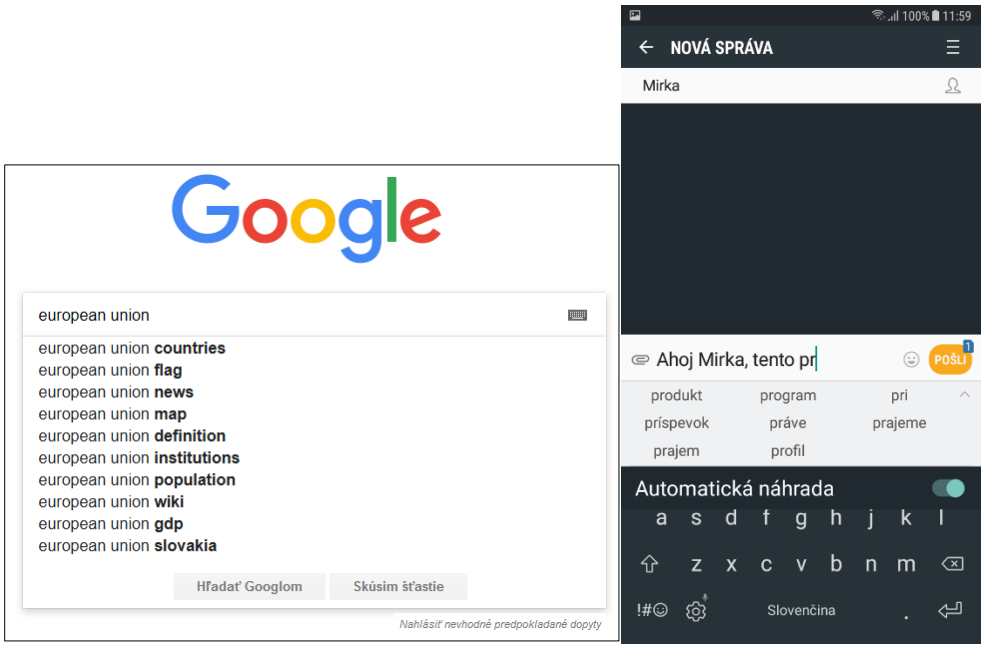
automatické dokončovanie textu, algoritmy usporadúvania, metóda sort(), funkcia sorted(), binárne vyhľadávanie

Čo sa naučíme a čo si precvičíme

- budeme riešiť zaujímavý problém, s ktorým sa stretávame denne, napr. pri vyhľadávaní na internete,
- porovnáme rôzne spôsoby usporadúvania údajov,
- zamyslíme sa nad efektívnosťou vyhľadávania hodnôt v neusporiadanom a usporiadanom zozname,
- naučíme sa používať metódu `sort()` a funkciu `sorted()`,
- vytvoríme aplikáciu s grafickým používateľským rozhraním, ktorá bude schopná realizovať automatické dokončovanie textu.

Problémová situácia

Automatické dokončovanie je dôležitou vlastnosťou používateľského rozhrania mnohých aplikácií a služieb. Na prvom obrázku (pozri obrázok nižšie) vidíme, ako vyhľadávač Google v reálnom čase zobrazuje návrhy na dokončenie dopytu (kľúčového slova, slovného spojenia), ktorý používateľ píše do vstupného textového poľa (pozri obrázok nižšie). Druhý obrázok ukazuje, ako mobilná aplikácia urýchľuje písanie textovej správy na nepohodlnej softvérovej klávesnici smartfónu. V tomto prípade však ide skôr o automatické nahradzovanie textu ako o dokončovanie.



Obrázok 94 Pohľad na rozhranie vyhľadávača a aplikácie na písanie textových správ

V oboch prípadoch program predvída, čo chce používateľ napísať, a ponúka zoznam návrhov na dokončenie usporiadaný zostupne podľa váh jednotlivých možností (od

najpravdepodobnejšej, najvýznamnejšej po tú najmenej pravdepodobnú, najmenej významnú). Váhy návrhov sa v praxi určujú rôznym spôsobom, napr. na základe frekvencie rovnakého dopytu od ostatných používateľov vyhľadávača Google alebo zo štatistických údajov o písaní znakov na klávesnici mobilného telefónu v minulosti.

Poznámka na okraj

So zobrazovaním návrhov na automatické dokončenie sa stretávame často. Otestujte vyhľadávanie na stránke svojho obľúbeného e-shopu, vo filmovej databáze alebo na sociálnej sieti.

Aj v integrovaných vývojových prostrediach (napr. IDE PyCharm) môžu programátori využívať výhody automatického dokončovania. Zobrazovanie návrhov (napr. názvov dostupných vlastností a metód) urýchľuje editovanie zdrojového kódu.

Aby mala funkcia automatického dokončovania zmysel (bola prínosom pre používateľa), musí byť implementovaná efektívne (používateľ je spravidla netrpezlivý). Navyše mnohé aplikácie poskytujú vyhľadávacie služby veľkému počtu používateľov súčasne. Predstavme si, koľko ľudí na Slovensku (v Európe, na celom svete) práve v tejto chvíli píše na klávesnici dopyt vo vyhľadávači Google. Po každom stlačení ďalšieho klávesu je potrebné každému z nich zobraziť aktualizovaný zoznam návrhov na dokončenie jeho dopytu. Výpočet sa deje na serveroch Googlu v reálnom čase s výsledkom oneskoreným nanajvýš o niekoľko desiatok milisekúnd.

V tejto kapitole budeme riešiť problém automatického dokončovania pre zoznam potenciálnych návrhov uložených v textovom súbore s pevne priradenými váhami (110). **Vytvoríme aplikáciu schopnú dokončovať názvy miest sveta**.

Zoznam návrhov ani ich váhy sa po spustení aplikácie už nebudú meniť. **Sústredíme sa teda len na správne a rýchle zobrazovanie návrhov na dokončenie dopytu**, historické dáta o spracovaných dopytoch nás nebudú zaujímať.

Hľadajme riešenie

Vstupný textový súbor *mesta_sveta.txt* obsahuje informácie o mestách sveta a ich populácii. V každom riadku sa nachádzajú vždy dve hodnoty oddelené tabulátorom: **váha** (počet obyvateľov) a **návrh** (názov mesta). Riadky sú vo vstupnom súbore usporiadané zostupne podľa počtu obyvateľov.

Keď používateľ napíše do vstupného poľa niekoľko znakov, musíme vedieť rýchlo **vyhľadať** tie mestá, názvy ktorých začínajú príslušným prefixom. Návrhy na automatické dokončenie budeme chcieť zobrazovať **usporiadané** od možnosti s najväčšou váhou po možnosť s najmenšou váhou.

Aby sme naprogramovali vlastnú aplikáciu podporujúcu automatické dokončovanie textu, musíme vedieť údaje efektívne usporadúvať a vyhľadávať v nich. Predtým, ako budeme spracúvať dáta z nášho vstupného súboru, **usporiadame** postupnosť celočíselných hodnôt a zamyslíme sa nad tým, ako možno v takejto postupnosti hodnoty **vyhľadávať**.

Poznámka na okraj

V slovenských zdrojoch o algoritmoch sa často stretávame s termínom *triediaci algoritmus*. My však dáta netriedime do skupín, ale usporadúvame vzostupne alebo zostupne podľa požadovaného kritéria. Namiesto triediaci algoritmus budeme preto hovoriť *algoritmus usporadúvania*.

Úloha 1

Uvažujme postupnosť celočíselných hodnôt uloženú v zozname:

```
data = [5, 3, 8, 6, 1, 2]
```

Chceme ju usporiadať vzostupne (od najmenšej hodnoty po najväčšiu). Simulujme výpočet nasledujúceho algoritmu usporadúvania:

```
def bubble_sort(a):
    '''Usporiada prvky zoznamu a.'''
    for i in range(len(a)-1,0,-1):
        for j in range(i):
            if a[j] > a[j+1]:
                a[j], a[j+1] = a[j+1], a[j]
```

Zavolajme vyššie uvedenú funkciu s parametrom `data`:

```
data = [5, 3, 8, 6, 1, 2]
bubble_sort(data)
```

Náš zoznam `data` má 6 prvkov. Hodnota výrazu `len(a)-1` bude preto 5. Vonkajší cyklus `for` sa vykoná 5 krát, pretože premenná `i` postupne nadobudne hodnotu 5, 4, 3, 2 a 1. Počet opakovaní vnútorného cyklu `for` závisí od aktuálnej hodnoty premennej `i`:

`i`	vnútorný cyklus `for` sa vykoná
5	5 krát, pre `j` = 0, 1, 2, 3, 4
4	4 krát, pre `j` = 0, 1, 2, 3
3	3 krát, pre `j` = 0, 1, 2
2	2 krát, pre `j` = 0, 1
1	1 krát, pre `j` = 0

Vnútorným cyklom `for` realizujeme prechod po zozname. Úsek, ktorý spracúvame, sa postupne skracuje. V každom prechode opakujeme pre všetky dvojice susedných prvkov rovnaké príkazy: porovnáme ich a ak sú v nesprávnom poradí, vymeníme ich. Sledujme, čo sa deje pri prvom prechode zoznamom pre `i=5`:

	0	1	2	3	4	5	
data	5	3	8	6	1	2	stav pred prechodom
j=0	5	3	8	6	1	2	data[0]>data[1], vymeníme ich
j=1	3	5	8	6	1	2	data[1]<data[2], nevymieňame ich
j=2	3	5	8	6	1	2	data[2]>data[3], vymeníme ich
j=3	3	5	6	8	1	2	data[3]>data[4], vymeníme ich
J=4	3	5	6	1	8	2	data[4]>data[5], vymeníme ich
	3	5	6	1	2	**8**	stav po prechode

Po skončení prvého prechodu sa na posledné miesto v zozname dostal najväčší prvok. Sledujme, čo sa udeje pri druhom prechode zoznamom pre `i=4`:

	0	1	2	3	4	5	
data	3	5	6	1	2	**8**	stav pred prechodom
j=0	3	5	6	1	2	**8**	data[0]<data[1], nevymieňame ich
j=1	3	5	6	1	2	**8**	data[1]<data[2], nevymieňame ich
j=2	3	5	6	1	2	**8**	data[2]>data[3], vymeníme ich
j=3	3	5	1	6	2	**8**	data[3]>data[4], vymeníme ich
	3	5	1	2	**6**	**8**	stav po prechode

Po druhom prechode sa na svoje miesto (predposledné miesto v zozname) dostal druhý najväčší prvok. V ďalšom prechode nás už budú zaujímať len prvé štyri prvky. Pokračujte v simulácii ďalej. Po poslednom prechode bude zoznam určite usporiadaný.

Poznámka na okraj

Koľko porovnaní prvkov sa vykoná pre 6 prvkový zoznam? Spočítajme porovnania realizované v jednotlivých prechodoch:

$5 + 4 + 3 + 2 + 1 = 15.$

Všeobecne pre n-prvkový zoznam:

$(n\text{-}1) + \ldots + 2 + 1 = \frac{n.(n\text{-}1)}{2} = \frac{n^2\text{-}n}{2}.$

Použili sme vzorec na súčet prvých *n* členov aritmetickej postupnosti.

Ak chceme skontrolovať, či sme pri simulovaní usporadúvania postupovali správne (t. j. vidieť stav zoznamu po skončení každého prechodu), môžeme si jeho priebeh overiť spustením skriptu, v ktorom funkciu s algoritmom zavoláme. Vo funkcii na vhodné miesta doplníme príkaz výpisu:

```
def bubble_sort(a):
    '''Usporiada prvky zoznamu a.'''
    for i in range(len(a)-1,0,-1):
        print('prechod c. '+str(len(a)-i)+': ',end='')
        for j in range(i):
            if a[j] > a[j+1]:
                a[j], a[j+1] = a[j+1], a[j]
        print(a)
```

Poznámka na okraj

Algoritmus `bubble_sort`, ktorý práve skúmame, sa nazýva *Bublinové usporadúvanie* alebo *Usporadúvanie výmenami*. Ak sa na hodnoty uložené v zozname dívame ako na „bubliny“, potom môžeme povedať, že po každom prechode zoznamom „vyletí“ najväčšia bublina práve spracúvaného úseku na svoje miesto. Ilustruje to aj video dostupné na https://goo.gl/ayJbUo [30. 6. 2020]

Algoritmus z **Úlohy 1** je príkladom jednoduchého, ale pomalého usporadúvania. Vo funkcii `bubble_sort` musíme pre *n* prvkový zoznam vykonať až $\frac{n.(n-1)}{2} = \frac{n^2-n}{2}$ porovnaní. Ak nepracujeme s veľkým počtom prvkov, môžeme sa uspokojiť aj s niektorým základným algoritmom usporadúvania a naprogramovať si v aplikácii vlastnú funkciu. V praxi sa však stretneme s efektívnejšími riešeniami. Programátori na usporiadanie prvkov bežne využívajú hotové funkcie z knižníc, ktoré implementujú niektorý z rýchlych algoritmov usporadúvania (napr. QuickSort – *Rýchle triedenie*, MergeSort – *Triedenie zlučovaním*).

Výkladový text

V Pythone máme pri usporadúvaní zoznamu na výber metódu `sort()` a funkciu `sorted()`. Usporiadajme čísla uložené v zozname `data1`:

```
>>> data1 = [5, 3, 8, 6, 1, 2]
>>> data1.sort()
>>> print(data1)
[1, 2, 3, 5, 6, 8]
>>> data1 = [5, 3, 8, 6, 1, 2]
>>> print(sorted(data))
[1, 2, 3, 5, 6, 8]
>>> print(data1)
[5, 3, 8, 6, 1, 2]
```

Metóda `sort()` realizuje usporadúvanie „na mieste“, t. j. **modifikuje zoznam, pre ktorý je volaná**. Funkcia `sorted()` vytvorí **nový zoznam**, v ktorom budú prvky usporiadané vzostupne a vráti ho ako výsledok. V oboch prípadoch je možné špecifikovať ďalšie dva parametre.

V parametri *reverse* môžeme požadovať *zostupné usporiadanie* prvkov:

```
>>> data1 = [5, 3, 8, 6, 1, 2]
>>> data1.sort(reverse=True)
```

```
>>> print(data1)
[8, 6, 5, 3, 2, 1]
```

Vieme tiež ovplyvniť kritérium, podľa ktorého sa má usporadúvať. Parameter *key* je **referencia na funkciu** s jedným parametrom, ktorá pre hodnotu každého prvku vypočíta jeho *kľúč*. Prvky sa budú usporadúvať podľa hodnôt kľúčov, napr.:

```
data2 = ['Matus', 'Peto', 'Katka', 'Hanka', 'Jan', 'Martina']
data2.sort()
print(data2)                                                    #[1]


def fun(x):
    return x[-1]

data2 = ['Matus', 'Peto', 'Katka', 'Hanka', 'Jan', 'Martina']
data2.sort(key=fun)
print(data2)                                                    #[2]
```

[1] Vypíše sa zoznam mien usporiadaný podľa abecedy (t. j. lexikograficky):
`['Hanka', 'Jan', 'Katka', 'Martina', 'Matus', 'Peto']`

[2] Vypíše sa zoznam mien usporiadaný podľa ich posledných písmen, pretože funkcia `fun` vracia pre každý prvok jeho posledný znak. A práve tie sa pri porovnávaní berú do úvahy:
`['Katka', 'Hanka', 'Martina', 'Jan', 'Peto', 'Matus']`

Keď je funkcia na výpočet hodnoty kľúča jednoduchá, jednoriadková (ide o vyhodnotenie jedného výrazu, tak ako v našom prípade), je praktické definovať ju ihneď pri nastavovaní parametra *key* s využitím konštrukcie **lambda**, napr.:

```
data2 = ['Matus', 'Peto', 'Katka', 'Hanka', 'Jan', 'Martina']
data2.sort(key=lambda x: x[-1])                                 #[3]
print(data2)                                                    #[4]
```

[3] Zápisom `lambda x: x[-1]` sme definovali **anonymnú funkciu** (nepotrebuje mať meno, lebo sa používa len na tomto mieste programu). Pre hodnotu parametra `x` uvedeného pred dvojbodkou, vráti táto funkcia výsledok vyhodnotenia výrazu `x[-1]` uvedeného za dvojbodkou.

[4] Aj v tomto prípade sa vypíše zoznam mien usporiadaný podľa ich posledných písmen, podobne ako v riadku [2], keď sme v riešení použili funkciu `fun`:
`['Katka', 'Hanka', 'Martina', 'Jan', 'Peto', 'Matus']`

Všimnite si pozície prvkov končiace na rovnaké písmeno v pôvodnom zozname. Metóda `sort()` zachovala ich poradie.

Úloha 2

Čo sa vypíše na obrazovku v nasledujúcich skriptoch?

a)

```
data = [1980, 1969, 1971, 1978, 2015, 1945, 2012, 1984]
data.sort()
print(data)
data.sort(key = lambda x: x%100)
print(data)
```

b)

```
data = [(5,'Nitra'), (4,'Kosice'), (3,'Banska Bystrica'),
        (2, 'Bratislava'), (1,'Zilina')]
print(sorted(data, key=lambda x:x[1]))
print(sorted(data, key=lambda x:x[0]))
```

c)

```
retazec = 'Python'
print(sorted(retazec, reverse=True))
print(''.join(sorted(retazec, key=str.lower)))
```

Úloha 3

Vygenerujte zoznam celých čísel a usporiadajte ho s využitím vlastného algoritmu usporadúvania (napr. *BubbleSort*) a zoznamovej metódy `sort()`. Porovnajte, koľko sekúnd bude trvať výpočet.

Poznámka na okraj

Funkciu `sorted()` môžeme aplikovať na zoznamy, ale aj na množinu, znakový reťazec alebo n-ticu. Vždy však vráti nový zoznam prvkov pôvodnej postupnosti usporiadaný vzostupne alebo zostupne, podľa hodnoty parametra `reverse`.

V Pythone je možné uchovávať v zozname hodnoty rôzneho typu. Pri usporadúvaní je potrebné, aby sa hodnoty prvkov dali vzájomne porovnávať. Nie je možné porovnávať napr. celé čísla a reťazce.

Ak by sme usporadúvali zoznam s objektami, napr. typu `Ziak`, kľúčom by mohli byť hodnoty získané alebo vypočítané na základe rôznych atribútov, napr.: meno, vek, aritmetický priemer vypočítaný zo známok a pod.

Údaje uložené v zozname už vieme usporiadať. Porozmýšľajme teraz nad efektívnym spôsobom vyhľadávania.

Úloha 4

Vygenerujte zoznam celých čísel a usporiadajte ho. Potom napíšte funkciu, pomocou ktorej budete môcť v zozname vyhľadávať. Ak sa hľadaná hodnota v zozname *nachádza*, vráťte jej pozíciu. Ak v zozname nie je, vyvolajte výnimku.

Náš skript bude jednoduchý, hodnotu budeme v zozname vyhľadávať tak, že postupne prezrieme celý zoznam prvok po prvku:

```
import random

def sekvencne_vyhladavanie(zoznam, hodnota):
    '''Vráti pozíciu hľadanej hodnoty alebo ValueError, ak sa hodnota
     v zozname nenachádza.'''
    for i in range(len(zoznam)):
        if zoznam[i]==hodnota:
            return i
    raise ValueError


# generovanie a usporiadanie zoznamu
data = [random.randint(0,100) for i in range(10)]
data.sort()
print(data)

try:
    index = sekvencne_vyhladavanie(data, 7)
    print('Hladana hodnota je na pozicii:',index)
except ValueError:
    print('Hladana hodnota sa v zozname nenachadza!')
```

Koľko porovnaní budeme musieť vykonať v najhoršom prípade? Koľko porovnaní postačí v najlepšom prípade?

Hľadaná hodnota by mohla byť na prvej pozícii, ale aj na poslednej pozícii v zozname. A možno v zozname nie je vôbec. Pri prehľadávaní n-prvkového zoznamu od začiatku teda v najhoršom prípade vykonáme práve *n* porovnaní (skontrolujeme všetky prvky). V najlepšom prípade bude hľadaná hodnota na prvej pozícii a postačí nám jediné porovnanie. Náš zoznam mal len 10 prvkov, mohol by ich mať aj oveľa viac (1000, 100 000 či 1 000 000). Pokúsme sa preto vyhľadávať v zozname efektívnejšie.

Otázka

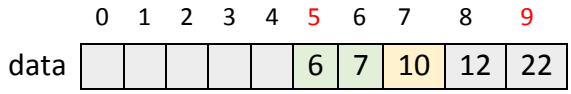
Ako by sme postupovali pri hľadaní prekladu anglického slova v anglicko-slovenskom slovníku?

Uvažujme zoznam `data` s 10 prvkami a hľadajme v ňom prvok s hodnotou 7. Na obrázku vidíme, že sa nachádza na pozícii s indexom 6. Na jeho vyhľadanie by sme teda pri sekvenčnom prístupe potrebovali 7 porovnaní. Vieme to však urobiť oveľa rýchlejšie, keď si uvedomíme, že **náš zoznam je usporiadaný**!

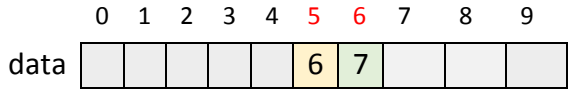
	0	1	2	3	4	5	6	7	8	9
data	1	2	2	3	5	6	7	10	12	22

Skúsme najprv pozrieť, či sa hľadaná hodnota nenachádza v strede zoznamu, teda na pozícii s indexom 4. Prvok `data[4]` má hodnotu 5. Hodnota, ktorú hľadáme, je väčšia ako 5. V zozname sa preto môže nachádzať len vpravo od nej (zelená časť zoznamu). Na prvky zoznamu s indexami 0 až 3 už môžeme zabudnúť, nemá zmysel ich prezerať. Tento postup (delenie postupnosti na polovice) opakujeme dovtedy, kým prvok nenájdeme, alebo kým nezistíme, že v zozname hľadaný prvok určite nie je.

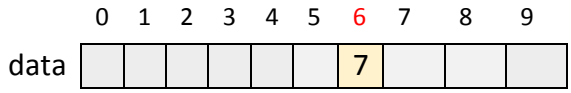
Pokračujme teda ďalej:



V strede úseku s hranicami 5 a 9 sa nachádza prvok `data[7]` s hodnotou 10. Hľadaná hodnota je menšia ako 10. Ak v zozname je, nachádza sa vľavo, kde pripadá do úvahy už len dvojprvkový úsek:



Vypočítajme index „prostredného z dvoch prvkov“: `(5+6)//2 == 5`. Prvok `data[5]` má hodnotu 6. Hľadaná hodnota je väčšia, budeme preto pokračovať v hľadaní vpravo od nej. Zostal nám už len jednoprvkový úsek s prvkom na pozícii 6:



Hodnota prvku `data[6]` je 7, čo je hľadaná hodnota. Na jej vyhľadanie sme potrebovali len 4 porovnania.

Ak by sme hľadali napr. hodnotu 8, mala by sa nachádzať vpravo od hodnoty 7. Ďalšie prvky, ktoré by prichádzali do úvahy, už ale vpravo nemáme. Vyhľadávanie by teda v tomto prípade skončilo neúspechom.

Poznámka na okraj

Napr. pre 1024 prvkov by sme na vyhľadanie ľubovoľnej hodnoty potrebovali najviac 10 porovnaní. Počet porovnaní závisí od počtu prvkov logaritmicky: $\log_2 1024 = 10$. Inak povedané: 1024 prvkov musíme 10 krát rozdeliť na polovicu, aby sme získali úsek s jediným prvkom.

Algoritmus, ktorý sme práve demonštrovali, sa nazýva **binárne vyhľadávanie** (v angl. *binary search*)**.** Môžeme ho implementovať takto:

```
def binarne_vyhladavanie(zoznam, hodnota):
    '''Vráti pozíciu hľadanej hodnoty alebo ValueError, ak sa
       hodnota v zozname nenachádza.'''

    lavy = 0
    pravy = len(zoznam) - 1
    while lavy <= pravy:
        stred = (lavy + pravy) // 2
        prvok = zoznam[stred]
        if hodnota < prvok:
            pravy = stred - 1
        elif hodnota > prvok:
            lavy = stred + 1
        else:
```

```
            return stred
    raise ValueError
```

Funkcia vracia index nájdeného prvku. Ak sa hľadaná hodnota v zozname nenachádza, funkcia to oznámi výnimkou typu `ValueError`.

Premenné `lavy, pravy` obsahujú indexy hraníc úseku zoznamu, v ktorom vyhľadávame.

Do premennej `stred` ukladáme index prostredného prvku aktuálneho úseku.

Už vieme, ako sa dá v Pythone usporiadať zoznam prvkov a ako v ňom možno vyhľadávať. Vrátime sa preto k pôvodnému problému:

Úloha 5

Údaje zo vstupného súboru *mesta_sveta.txt* uložte do zoznamu dvojíc (prvým prvkom dvojice nech je *počet obyvateľov*, druhým *názov mesta)*. Tento zoznam usporiadajte **vzostupne podľa názvov miest**, aby ste v ňom mohli neskôr rýchlejšie vyhľadávať aplikovaním algoritmu binárneho vyhľadávania. Riešenie implementujte v samostatnej triede s názvom `MestaSveta`.

Úloha 6

Vždy keď používateľ napíše na klávesnici nejaký reťazec znakov, budeme chcieť získať zoznam všetkých miest (návrhov na dokončenie), ktoré týmto reťazcom (prefixom) začínajú. Výsledný zoznam návrhov je potrebné ešte usporiadať *zostupne* podľa váhy, ktorou je v tomto prípade veľkosť populácie.

Rozšírime riešenie **Úlohy 5** doplnením ďalších metód a riešenie otestujeme, napr. takto:

```
mesta = MestaSveta('mesta_sveta.txt')
vysledok = mesta.vrat_navrhy('Nit')
for navrh in vysledok:
    print(navrh)
```

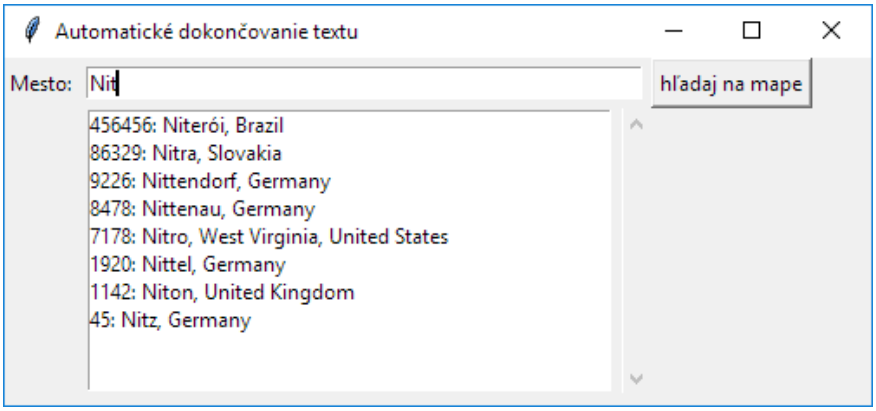
Výstup testovacieho skriptu:

```
(456456, 'Niterói, Brazil')
(86329, 'Nitra, Slovakia')
(9226, 'Nittendorf, Germany')
(8478, 'Nittenau, Germany')
(7178, 'Nitro, West Virginia, United States')
(1920, 'Nittel, Germany')
(1142, 'Niton, United Kingdom')
(45, 'Nitz, Germany')
```

Trieda `MestaSveta` bude zapuzdrovať hlavnú funkcionalitu našej aplikácie, t. j. podporu pre automatické dokončovanie textu. Vstup pre metódu `vrat_navrhy()` bude neskôr zadávať používateľ tým, že napíše do vstupného textového poľa v okne aplikácie konkrétnu postupnosť znakov.

Úloha 7

Otestovanú triedu `MestaSveta` zužitkujte v aplikácii s grafickým používateľským rozhraním (pozri obrázok nižšie). V aplikácii môžete využiť na zobrazovanie aktuálnych návrhov na dokončenie komponent `Listbox`. Pre zaujímavosť (a kontrolu správnosti) zobrazujte aj váhy jednotlivých možností. Zabezpečte, aby sa zo zoznamu návrhov dala vybrať tá možnosť, ktorá sa má použiť na dokončenie vstupu.



Obrázok 95 Aplikácia na testovanie automatického dokončovania textu

Poznámka na okraj

Aplikácia má zoznam návrhov na dokončenie aktualizovať **po každom stlačení klávesu** na klávesnici.

Tlačidlom „hľadaj na mape“ budeme predstierať, že sme vstup od používateľa na niečo zmysluplné použili (napr. pri zobrazovaní polohy mesta na digitálnej mape). V reakcii na jeho stlačenie môžeme vymazať obsah `Listboxu` alebo zobraziť správu.

Čo sme sa naučili

- Existujú rôzne algoritmy riešiace problém usporadúvania. Bublinové usporadúvanie je príkladom jednoduchého, ale pomalého algoritmu, s ktorým sa môžeme uspokojiť len pri spracúvaní malého počtu prvkov.
- V knižnici Pythonu máme k dispozícii efektívnu implementáciu algoritmu usporadúvania v metóde `sort()` alebo vo funkcii `sorted()`. Každá z nich môže mať dva voliteľné parametre (`key`, `reverse`), v ktorých špecifikujeme funkciu na výpočet kľúča reprezentujúceho prvok pri porovnávaní a vyberáme medzi vzostupným a zostupným usporiadaním. Hodnotu parametra `key` často definujeme ako `lambda` funkciu.
- Pri sekvenčnom vyhľadávaní v n-prvkovom zozname potrebujeme v najhoršom prípade vykonať až *n* porovnaní. Binárne vyhľadávanie je efektívnejšie, počet porovnaní závisí od veľkosti vstupného zoznamu logaritmicky. Môžeme ho však použiť len pre usporiadané postupnosti.

Ďalšie úlohy na precvičenie a zamyslenie

1. Pre rovnaké vstupné údaje simulujte výpočet iných dvoch algoritmov usporadúvania. Vysvetlite slovne, ako usporadúvanie prebieha.

```
def min_sort(a):
    for i in range(0, len(a)-1):
        pozMin = i
        for j in range(i+1, len(a)):
            if a[j] < a[pozMin]:
                pozMin = j
        a[i], a[pozMin]  = a[pozMin], a[i]


def insert_sort(a):
    for i in range(1, len(a)):
        x = a[i]
        poz = i
        while poz > 0 and a[poz - 1] > x:
            a[poz] = a[poz - 1]
            poz = poz - 1
        a[poz] = x
```

2. V knižnici Pythonu sa nachádza modul s názvom `bisect`. Prezrite si v dokumentácii, aké funkcie obsahuje. Obsahuje aj podporu pre binárne vyhľadávanie?

24. Morfing

Kľúčové slová

`prelínanie`, `morfing`, `alfa kanál`, `interpolácia`, `animácia.`

Čo sa naučíme a čo si precvičíme

- vysvetlíme si ako vytvoriť animovaný obrázok,
- vysvetlíme si ako vytvoriť efekt plynulého prechodu od jedného obrázka k druhému,
- čo je to morfing,
- naučím sa ako morfovať písmená a číslice,
- precvičíme si prácu v korytnačej geometrii.

Problémová situácia

Pravdepodobne ste sa už niekde v médiách stretli s animáciami, kde obrazy plynulo prechádzali od jedného do druhého. Animačnou sekvenciou môžeme aj v Pythone dosiahnuť prechod jedného obrazu do druhého postupným prelínaním, kedy sa stráca obsah prvého obrazu a postupne sa zobrazuje obsah toho druhého. Postupným prelínaním a miešaním svojich obsahov dosiahneme tzv. „blend effect“ (111). Vo svete filmových špeciálnych efektov je taktiež možné vidieť animačné sekvencie s plynulou transformáciu jedného objektu na druhý, tzv. „morphing“ (112). V tejto kapitole si ukážeme ako spraviť prelínanie v rastrovej grafike a ako urobiť morfing písmen zapísaných pomocou vektorov bodov.

Úloha 1

Vytvorte animáciu pohybujúcej sa loptičky na bielom pozadí a uložte ju ako animovaný obrázok vo formáte gif.

V tejto úlohe budeme pracovať s knižnicou **Python Imaging Library** (`from PIL import Image, ImageDraw`) (111). Pomocou príkazu `Image.new` vytvoríme v pamäti nový obrázok s konkrétnou veľkosťou a bielym pozadím [1]. Príkazom `ImageDraw.ellipse` vykreslíme do obrázka modrý kruh [2].

```
def animovana_lopta_img(sirka, vyska, lopta_x, lopta_y,
lopta_veľkosť):
    img = Image.new('RGB', (vyska, sirka), (255, 255, 255))
    draw = ImageDraw.Draw(img)                          [1]
    draw.ellipse((lopta_x,
                  lopta_y,
                  lopta_x + lopta_size,
                  lopta_y + lopta_size),
                  fill='blue')                          [2]
    return img
```

Ďalej potrebujeme vytvoriť sériu obrázkov s pohybujúcou sa loptou. Vytvorme funkciu `animacia_lopty(sirka, vyska, priemer)`.

```
snimky = []                                                    [3]
x, y = 0, 0
for i in range(10):
    novy_zaber = lopta_img(sirka, vyska , x, y, 40)            [4]
    snimky.append(novy_zaber)
    x += 40                                                    [5]
    y += 40
return snimky
#-----------------------------------
snimky = animacia_lopty(400,400, 40)
```

[3] Inicializujeme prázdny zoznam snímkov animácie.

[4] V cykle vytvoríme sériu záberov s loptou o veľkosti 400x400 na zadaných súradniciach a pridáme novovytvorený záber animácie do zoznamu.

[5] Posunieme súradnice lopty a vrátime zoznam s obrázkami.

Teraz nasleduje aktuálne pre nás najdôležitejšia časť, potrebujeme spojiť všetky snímky do jedného animovaného obrázka. Na to slúži metóda `save`.

```
snimky[0].save('animovana_lopta.gif',                          [6]
               format='GIF',
               append_images=snimky[1:],                       [7]
               save_all=True,
               duration=100,                                   [8]
               loop=0)                                         [9]
print('Hotovo. Animacia je vytvorena.')
```

[6] Nastavíme meno výsledného GIF obrázka.

[7] Sekvencia začne obrázkom `snimky[0]`. V parametri `append_images` sa uvedie zoznam všetkých obrázkov, ktoré majú byť pripojené. V našom prípade sú to všetky okrem prvého.

[8] Nastavíme čas, na ako dlho sa má každý snímok zo sekvencie zobraziť.

[9] Parameter určí správanie sa sekvencie obrázka. Ak nastavíme `loop = 0`, animácia sa opakuje do nekonečna. Ak uvedieme konkrétne číslo, tak sa animácia zopakuje len zvolený počet krát.

Výkladový text

Výsledný súbor nebude uložený v tom istom priečinku ako je uložený zdrojový kód. Súbor sa uloží do tzv. **current working directory**. Ak chceme mať pokope kód aj vytvorenú animáciu, musíme uviesť buď konkrétnu cestu alebo zistíme relatívnu cestu, teda cestu k zdrojovému kódu.

```
import os
def relativna_cesta_file(img):
    cesta = os.path.dirname(os.path.realpath(__file__))
    return cesta + '\\' + img
```

Úloha 2

Pri ukladaní obrázka experimentujte s parametrami `duration` a `loop`. Vyskúšajte si ukladať rôzne verzie obrázka a porovnajte rozdiely v animácii.

Poznámka na okraj

Čo je to **GIF**? Graphics Interchange Format je bitmapový formát vyvinutý v roku 1987, ktorý dokáže okrem statického obrázka uchovávať aj sériu obrazov (113). Napriek tomu, že gif je už staromódny formát s 8bitovým kódovaním farieb, ktorý ukladá dáta bez zvuku a kompresie, má stále svoje výhody. Formát je podporovaný všetkými prehliadačmi a hlavne je jednoduché ho vytvoriť. Dnes sa často používa v komunikácii miesto emotikonov. Rozmach tohto starého formátu podporuje aj existencia stránok Giphy alebo Gfycat. (114)

Úloha 3

Vytvorte efekt plynulého prechodu medzi dvoma obrazmi. Výstup uložte ako animovaný obrázok s príponou gif.

Obrázok 96 Chceme vytvoriť efekt prechodu medzi obrázkom s hviezdami a obrázkom s hviezdami a vesmírnou flotilou

Vytvoríme si triedu `BlendMorph`, ktorá bude mať funkcie na vytvorenie efektu prechodu medzi dvoma rastrovými obrázkami (Obrázok 96). Umožní prechod uložiť na disk ako sekvenciu obrázkov alebo ako jeden animovaný obrázok formátu gif.

Aby sme mohli vytvoriť efekt, je potrebné mať obrázky rovnakých rozmerov. Vytvorme funkciu, ktorá rozmer skontroluje. Ak nechceme písať celú adresu, opäť využijeme funkciu, ktorá sama upraví adresu podľa aktuálneho umiestnenia zdrojového kódu. Obrázky sa tak budú hľadať v tom istom priečinku, kde je uložený aj súbor `.py`.

```
def skontroluj_rozmer(self,
                      img1,
                      img2):

    im1 = Image.open(self.relativna_cesta_file(img1))
    im2 = Image.open(self.relativna_cesta_file(img2))
    if im1.size != im2.size:
        return False
    else:
        return True
```

Ak obrázky nemajú rovnaký rozmer, alebo ich chceme pred tvorbou animácie jednoducho upraviť, tak využijeme metódu `Image.Resize`. Potrebujeme už len vhodne dopočítať zmeny vo veľkosti a to tak, aby vznikla, čo najmenšia deformácia pôvodného obrázku.

```
def uprav_rozmer(self,
                 maxRozmer,
                 image):

    img = Image.open(self.relativna_cesta_file(image))      [10]

    sirkaRatio  = maxRozmer[0]/img.size[0]                  [11]
    vyskaRatio  = maxRozmer[1]/img.size[1]

    novaSirka    = int(sirkaRatio*img.size[0])              [12]
    novaVyska    = int(vyskaRatio*img.size[1])

    newImage    = img.resize((novaSirka, novaVyska))        [13]
    return newImage

#----------------------------------

im1 = m.uprav_rozmer((800,500),'hviezdy.jpg')
im2 = m.uprav_rozmer((800,500),'hviezdyLOD.jpg')
```

[10] Otvoríme obrázok, načítame ho do pamäti. Ak nechceme písať celú adresu, opäť využijeme funkciu, ktorá sama upraví adresu podľa aktuálneho umiestnenia zdrojového kódu. Obrázky sa tak budú hľadať v tom istom priečinku, kde je uložený aj súbor `.py`.

[11] Zistíme koeficient zmeny medzi žiadanou a aktuálnou šírkou (výškou) obrázka. Ak je žiadaný rozmer väčší ako rozmer originálneho obrázka, bude koeficient väčší ako jedna, ak sú obrázky rovnaké, bude koeficient rovný jednej a ak je žiadaný rozmer menší, bude koeficient menší ako jedna.

[12] Nová šírka (výška) sa vypočíta na základe koeficientu.

[13] Pomocou metódy `resize` prepočítame (deformujeme) obrázok na nový rozmer.

Obrázky by mali mať nielen rovnaký rozmer ale aj rovnaký farebný model, napríklad RGBA. Je lepšie, ak pre istotu upravíme funkciou `kovertuj_farebny_model` ich modely na rovnakú hodnotu.

```
def konvertuj_farebny_model(self, image, model):
    return image.convert(model)

#----------------------------------

im1 = m.konvertuj_farebny_model(im1, 'RGBA')
im2 = m.konvertuj_farebny_model(im2, 'RGBA')
```

Výkladový text

Python podporuje rôzne farebné modely. Vyberáme z nich aspoň tie pre nás najzaujímavejšie:

- L (8-bit, čiernobiely obrázok)
- RGB (3x8-bit, true color)
- RGBA (4x8-bit, true color s trasparentnou maskou)
- CMYK (4x8-bit)
- HSV (3x8-bit, Hue, Saturation, Value)

Ak chceme vytvoriť animovaný obrázok, budeme postupovať rovnako ako v prvej úlohe. Potrebujeme vygenerovať sekvenciu obrázkov, ktoré spojíme do jedného GIF obrázka. Tu však nemôžeme objekty len tak prekresliť, keďže my potrebujeme vytvoriť plynulý prechod. Potrebujeme prepočítať, na koľko percent máme zobraziť začiatočný obrázok a na koľko percent finálny obrázok. Na to nám bude slúžiť tzv. alfa faktor.

```
def listBlended(self, image1, image2, pocet):
    list = []
    koeficient = 1/pocet                                          [14]
    for i in range(pocet):                                        [15]
        list = list +
               Image.blend(image1, image2, alpha=koeficient*i)]   [16]
    return list

#-----------------------------------

fazy = m.listBlended(im1, im2, 20)
```

[14] Koeficient, alebo veľkosti kroku, o koľko sa mám v každom behu programu posunúť bližšie k riešeniu, vypočítame na základe počtu žiadaných prechodov.

[15] Vytvoríme toľko prelínaní, koľko je žiadaných prechodov.

[16] Pomocou metódy `blend` spojíme dva obrázky. Ak je `alfa` 0,0, vráti sa kópia prvého obrázka. Ak je `alpha` 1,0, vráti sa kópia druhého obrázka. Ak je `alpha` 0,5 vznikne 50 percentné prekrytie z každého obrázka.

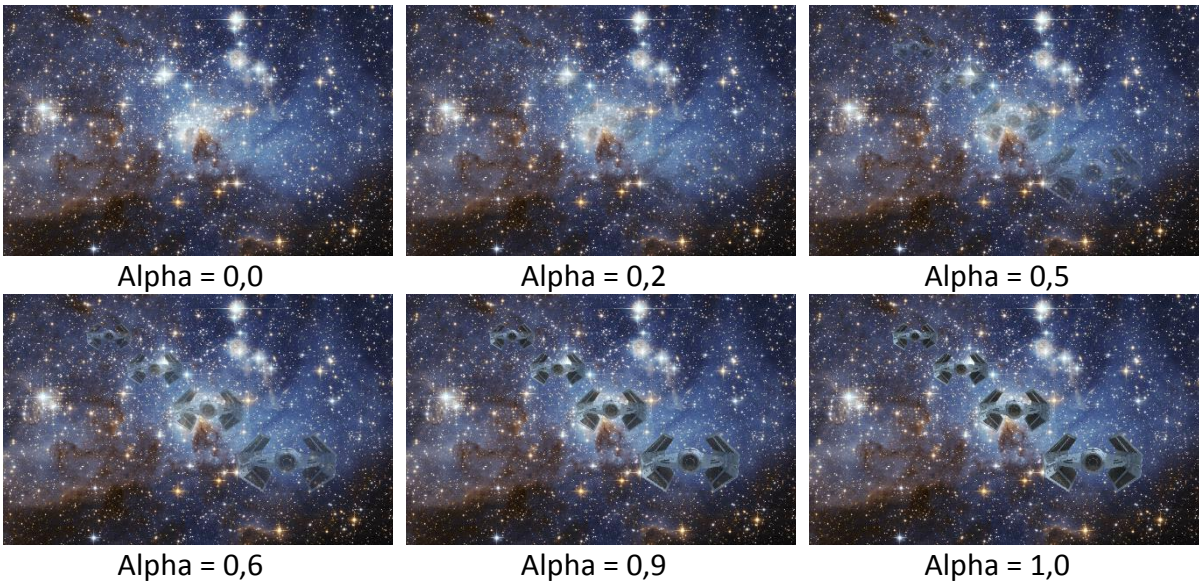
Posledným krokom je spojenie obrázkov. To sme už robili. Funkcia `morfuj` spojí zoznam v parametri `list`. Pokiaľ nezadáme inak, tak medzi všetkými spojenými obrázkami bude pauza 100 milisekúnd. Animácia má prebehnúť len raz, preto je parameter `loop` nastavený na hodnotu 1.

```
def morfuj(self, list, dur = 100):
    list[0].save(self.relativna_cesta() + "\\anitest.gif",
             save_all=True,
             append_images=list[1:],
             duration=dur,
             loop=1)

#-----------------------------------------

m.morfuj(fazy, 200)
```

Úloha 4

Uchovajte sekvenciu obrázkov, z ktorých sa vyskladala animácia. Umožnite zadať priečinok, do ktorého sa obrázky majú uložiť.

V našom riešení je potrebné uložiť zoznam `fazy` na disk (pozri obrázok nižšie). Počet uložených obrázkov bude závisieť od počtu obrázkov vygenerovaných vo funkcii `listBlended(self, image1, image2, pocet)`. Napríklad prechody pre 10 krokov, by fázy mohli vyzerať nasledovne.



Obrázok 97 Ukážka obrázkov vytvorených funkciou blend

```
def uloz(self, image, priecinok, meno):
    cesta = self.relativna_cesta()
    cesta = cesta + '\\' + priecinok
    if not(os.path.isdir(cesta)):                                    [17]
        os.mkdir(cesta)
    image.save(cesta + '\\' + meno)

def ulozList(self, list):
        for i in range(len(list)):
        self.uloz(list[i], priecinok, str(i)+".png")                 [18]
```

[17] Funkcia `uloz` vyskladá cestu, kde sa má súbor uložiť. Ak priečinok neexistuje `os.path.isdir(cesta)` vytvorí sa pomocou príkazu `os.mkdir(cesta)`.

[18] Každý obrázok zoznamu sa uloží do zadaného priečinka. Názov sa vytvorí automaticky na základe poradia obrázka v zozname.

Algoritmus morfingu

Efekty *Tweening* a *Morphing* sa často používajú v počítačových animáciách na zmenu tvaru objektu morfovaním objektu z jedného tvaru do druhého. V metóde **tweeting** sa medzi kľúčové snímky (prvý a posledný) vypočítavajú tzv. „medzisnímky“, aby sa vytvorila plynulá animácia (115). V tejto časti implementujeme morfovací algoritmus, kde pomocou lineárnej interpolácie zmeníme tvar písmena abecedy (napr. „A“ na iné písmeno „Z“). Písmená budeme

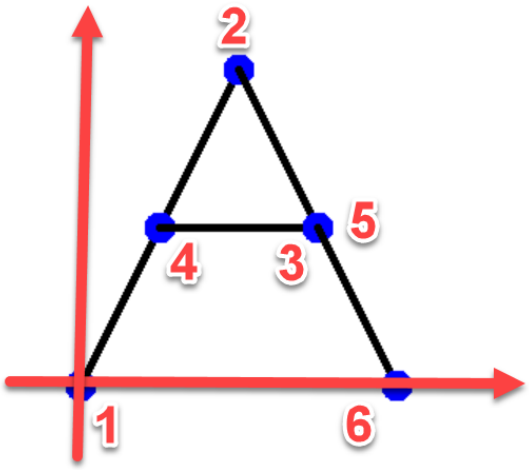
považovať za zoznam spojených uzlov (bodiek), kde každá bodka má svoju vlastnú súradnicu v priestore (x, y). Na vykreslenie využite modul `turtle`.

Úloha 5

Zakódujte písmeno A ako zoznam uzlov, kde každý uzol má súradnice x a y, udávajúcu jeho polohu v dvojrozmernom priestore.

Písmeno A si musíme najskôr zakódovať pomocou súradníc, najlepšie nezávisle od veľkosti daného písmena (pozri obrázok nižšie). Začnime vykresľovať v začiatku súradnicovej sústavy a navrhnime si takéto kódovanie. Ak sa posunieme o celú dĺžku, je jedno akú veľkú, poznačíme si hodnotu 1. Ak sa posunieme len o polovicu dĺžky poznačíme si 0,5. Takýmto prístupom by sme si písmeno A zakódovali ako zoznam šiestich bodov, ktoré musí korytnačka pri kreslení navštíviť.

```
[(0,0),(0.5,1),(0.75,0.5),(0.25,0.5),(0.75,0.5),(1,0)]
```



Obrázok 98 Písmeno A zapísané ako zoznam bodov, ktoré treba pri kreslení navštíviť

Keď máme takto zakódované písmeno, môžeme ho nakresliť v ľubovoľnej veľkosti. Rovnako flexibilný môžeme byť aj pri určení začiatku „súradnicovej sústavy“.

```
def vykresli_tvar(znak, font, x, y):
    penup()
    tracer(0)
    for bod in znak:                                          [19]
        goto(x + bod[0]*font, y + bod[1]*font)                [20]
        penup()
        dot(20,'blue')                                        [21]
        pendown()
    penup()
    goto(znak[0][0], znak[0][1])                              [22]
    update()

#--------------------

A = [(0,0),(0.5,1),(0.75,0.5),(0.25,0.5),(0.75,0.5),(1,0)]
vykresli_tvar(A, 300, -150, -150)
```

[19] Funkcia `vykresli_tvar` prejde zoznam všetkých bodov v zadanom písmene.
[20] Príkaz `goto` presunie korytnačku z bodu do bodu, pričom za sebou bude kresliť čiaru.
[21] V každom uzle vykreslí bod. Vykresľovanie bodiek v uzloch vo finálnej verzii bude nežiadúce. Aktuálne je veľmi prospešné.
[22] Presun korytnačky na pôvodné miesto nie je potrebný, tento riadok nijako neovplyvní fungovanie programu.

Poznámka

Kombinácia príkazov `tracer` a `update` nám urýchli vykresľovanie. Obraz sa zobrazí naraz. V prípade, že chceme vidieť postupné vykresľovanie musíme tieto dva príkazy zakomentovať.

Za účelom zefektívnenia ďalšej práce sme pripravili slovník obsahujúci definície všetkých písmen abecedy pomocou zoznamu ich uzlov. Nájdete ich v súbore `pismena.py (115)`. Ako súbor používať?

```
from znaky import abeceda
tvar = abeceda['Q']
vykresli_tvar(tvar, 300, -150, -150, 0.01)
```

`Abeceda` je slovník (`dict`), ktorý ako kľúče obsahuje preddefinované zoznamy bodov interpretujúce písmena abecedy a číslice. Experimentujte s vykresľovaním rôznych znakov.

Úloha 6

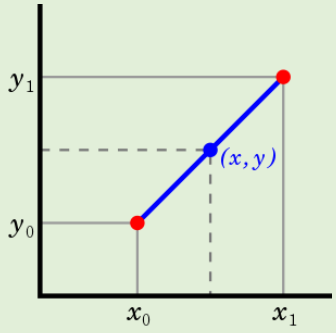
Vytvorte animáciu, kde bude jedno písmeno abecedy plynulo prechádzať do druhého písmena.

Musíme vlastne vytvoriť morfovaciu sekvenciu medzi dvoma písmenami abecedy, tzn. že musíme vypočítať lineárnu interpoláciu každého bodu zoznamu.

Výkladový text

Lineárna interpolácia je v matematike metóda na popis kriviek. Na základe interpolácie vieme na základe známych bodov dopočítať ďalšie body na krivke.

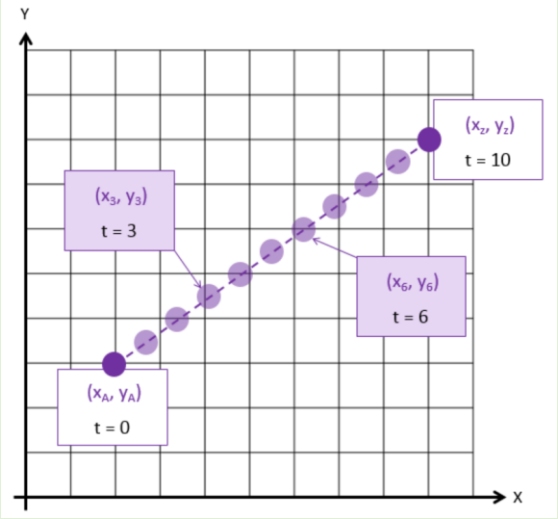
Predstavme si zjednodušene, že máme začiatočný bod `(x0, y0)` a koncový bod `(x1,y1)`. Ako vypočítame súradnice bodu `(x,y)`? Ak sú nám známe súradnice začiatočného a koncového bodu, vieme jeho pozície vypočítať. (116)



$$x = x_0 + (x_1 - x_0) * 0,5$$

$$y = y_0 + (y_1 - y_0) * 0,5$$

Interpoláciu môžeme využiť aj na výpočet sekvencie bodov medzi koncovou a začiatočnou hodnotou. Ak by sme interpoláciu rozdelili na 10 krokov výsledok by bol podobný tomu, čo vidíme na obrázku nižšie.



(115)

Každý bod v čase t by sme vypočítali na základe vzťahu:

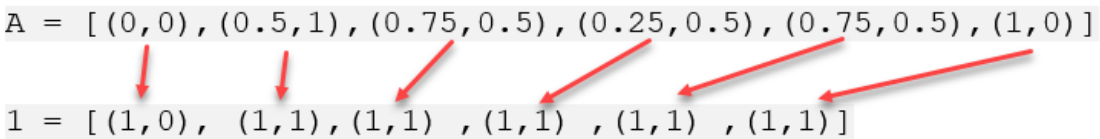
$$x(t) = x_A + (x_Z - x_A) * t / 10$$

$$y(t) = y_A + (y_Z - y_A) * t / 10$$

Zoberme si písmeno A a číslicu 1 zo slovníka abeceda. Aby bola interpolácia plynulá musia zoznamy obsahovať rovnaký počet uzlov.

```
A = [(0,0),(0.5,1),(0.75,0.5),(0.25,0.5),(0.75,0.5),(1,0)]
1 = [(1,0),(1,1)]
```

Musíme nájsť spôsob ako zoznam vhodne doplniť. Najlepšie riešenie je, aby body, ktoré nemajú „dvojičku“ v druhom zozname interpolovali do rovnakého bodu, a to do posledného bodu zoznamu, kratší zoznam teda opakovane doplníme hodnotou posledného bodu (pozri obrázok nižšie).



Obrázok 99 Ukážka, ako budú interpolovať uzly z písmena A do číslice 1

```
def dopln_body(T1, T2):
    pocetBodovT1 = len(T1)                                  [23]
    pocetBodovT2 = len(T2)
    posledyBodT2 = T2[pocetBodovT2-1]                       [24]
    pocetChybajucichBodov = pocetBodovT1 - pocetBodovT2     [25]
    for i in range(0, pocetChybajucichBodov):               [26]
        T2.append(posledyBodT2)
    return T2
```

[23] Funkcia `dopln_body(T1, T2)` je navrhnutá tak, že zoznam T1 je vždy ten dlhší. Kontrolu treba však vykonať mimo metódy doplň body. Najskôr zistíme koľko uzlov obsahuje každý zoznam.
[24] Zistíme hodnotu posledného prvku kratšieho zoznamu.
[25] Zistíme rozdiel v počte bodov.
[26] Do kratšieho zoznamu opakovane pridáme jeho posledný bod. Po skončení cyklu bude mať T2 a T1 rovnaký počet uzlov.

Teraz už môžeme vykonať interpoláciu. Pre každý bod postupne vypočítame jeho postupné približovanie sa k jeho koncovému bodu. Uplatníme tu vzorce pre lineárnu interpoláciu.

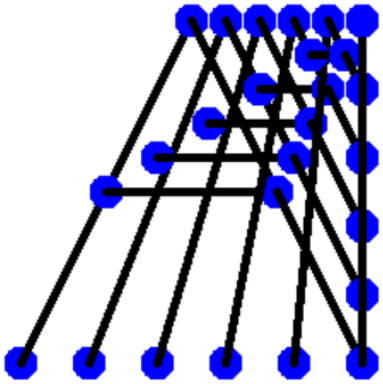
```
def interpolacia(T1, T2, t, n):                             [27]
    interpolacia_t= []
    for i in range(0,len(T1)):
        surx=T1[i][0] + (T2[i][0]-T1[i][0])*t/n             [28]
        sury=T1[i][1] + (T2[i][1]-T1[i][1])*t/n
        interpolacia_t.append([surx,sury])                  [29]
    return interpolacia_t
```

[27] Funkcia `interpolacia` vypočíta priblíženie sa v čase `t`, pričom vieme, že všetkých priblížení je `n`.
[28] Vypočítame x-ovú súradnicu v čase t.
[29] Vypočítame y-ovú súradnicu v čase t.
[30] Pridáme súradnice do zoznamu aktuálnej interpolácie.

Ako bude vyzerať päť blížení pre interpoláciu znaku A do čísla 1? Každý bod sa bude počítať samostatne, každý krok bude mať vlastnú veľkosť (pozri obrázok nižšie).

```
[[0.0, 0.0], [0.5, 1.0], [0.75, 0.5], [0.25, 0.5], [0.75, 0.5], [1.0, 0.0]]
[[0.2, 0.0], [0.6, 1.0], [0.8, 0.6], [0.4, 0.6], [0.8, 0.6], [1.0, 0.2]]
[[0.4, 0.0], [0.7, 1.0], [0.85, 0.7], [0.55, 0.7], [0.85, 0.7], [1.0, 0.4]]
[[0.6, 0.0], [0.8, 1.0], [0.9, 0.8], [0.7, 0.8], [0.9, 0.8], [1.0, 0.6]]
[[0.8, 0.0], [0.9, 1.0], [0.95, 0.9], [0.85, 0.9], [0.95, 0.9], [1.0, 0.8]]
[[1.0, 0.0], [1.0, 1.0], [1.0, 1.0], [1.0, 1.0], [1.0, 1.0], [1.0, 1.0]]
```

Obrázok 100 Päť kroková interpolácia bodov zoznamu



Obrázok 101 Ukážka, ako budú vyzerať uzly pre päť krokov interpolácie písmena A do číslice 1

Aby sme morfovací algoritmus dokázali aj vizualizovať, stačí, keď budeme každý prechod priebežne vykresľovať. Vytvoríme tak vizuálnu transformáciu jedného objektu do druhého (Obrázok 101).

```
def morfuj(T1, T2, font, zacx, zacy, pocetKrokov, cakaj):
    for krok in range(0,pocetKrokov+1):                          [31]
        clear()                                                  [32]
        morf_tvar = interpolacia(T1, T2, krok, pocetKrokov)      [33]
        vykresli_tvar(morf_tvar, font, zacx, zacy, cakaj)        [34]
```

[31] Funkcia `morfuj` bude postupne počítať prechody. Počet krokov určuje parameter.

[32] Ak nepoužijeme funkciu `clear()`, budeme vidieť priebeh všetkých stavov. Vyčistenie kresliacej plochy nám však umožní docieliť dojem skutočného pohyb bodov priestore.

[33] Vypočítame interpoláciu pre body T1 a T2 v určenom kroku.

[34] Vykreslíme aktuálny „medzitvar“. Parameter čakaj určuje rýchlosť animácie.

Poznámka

Vo funkcii `vykresli_tvar` sme pridali parameter `cakaj`, ktorý určuje rýchlosť prechodov pri vykresľovaní interpolácií. Ak má hodnotu 1, znamená to, že čaká 1 sekundu.

```
import time
def vykresli_tvar(znak, font, x, y, cakaj):
    ...
    time.sleep(cakaj)
    update()
```

Už nám zostáva len celý algoritmus vhodne pustiť. Najskôr určíme, do ktorého zoznamu sa majú pridať body a následne zavoláme morfovaciu sekvenciu.

```
def spusti(T1, T2, font, zacx, zacy, pocetKrokov, cakaj):

    if len(T1)>len(T2):
        T2 = dopln_body(T1, T2)
    elif len(T1)<len(T2):
        T1 = dopln_body(T2, T1)

    morfuj(T1, T2, font, zacx, zacy, pocetKrokov, cakaj)
#---------------------------------
pensize(5)
font = 200
x=0
y=0
cakanie = 0.1
pocetKrokov = 5
tvar1 = abeceda['A']
tvar2 = abeceda['1']
spusti(tvar1, tvar2, font, -font//2, -font//2, pocetKrokov, cakanie)
```

Úloha 7

Vytvorte animáciu, kde bude jedno písmeno abecedy plynulo prechádzať do druhého písmena a späť.

Na vyriešenie úlohy už máme všetko pripravené, stačí len ako posledný príkaz spustiť morfovací algoritmus pre tvary v opačnom poradí.

```
pensize(5)
font = 200
x=0
y=0
cakanie = 0.1
pocetKrokov = 5
tvar1 = abeceda['A']
tvar2 = abeceda['1']
spusti(tvar1, tvar2, font, -font//2, -font//2, pocetKrokov, cakanie)
spusti(tvar2, tvar1, font, -font//2, -font//2, pocetKrokov, cakanie)
```

Ďalšie úlohy na precvičenie a zamyslenie

1. Vytvorte morfovaciu sekvenciu, ktorá prejde všetky písmená abecedy.
2. Vytvorte morfovaciu sekvenciu, ktorá bude prechádzať z jedného náhodného písmena do iného náhodného písmena.
3. Vytvorte morfovaciu sekvenciu ktorá prejde všetkými písmenami zadaného ľubovoľného slova.

25. Domáci knižničný systém

Kľúčové slová

grafická aplikácia, správa objektov, dedičnosť, polymorfizmus, rodičovská trieda, odvodená trieda, prekrývanie,

Čo sa naučíme a čo si precvičíme

- Efektívne navrhovať triedy pre reprezentáciu podobných objektov,
- Navrhovať triedy pre správu iných objektov,
- Navrhovať grafické rozhrania pre manipuláciu s objektmi,
- K riešenie problémov pristupovať objektovo,
- Definovať rodičovské a odvodené triedy,
- Prekrývať metódy v odvodených triedach,
- Využívať polymorfizmus,

Problémová situácia

Kde som si tú knihu odložil? Myslel som že sem na tretiu poličku, ale tu nie je!

Kam som si nahral e-knihu o programovaní v Pythone? Myslel som že do čítačky, ale nie je tu!

Áno, toho autora poznám, mám od neho pár kníh! Ale teraz si nespomeniem aké a kde ich mám odložené!

Rovnaké alebo podobné situácie sme zažili už neraz. Množstvo vecí, ktoré máme k dispozícii, s ktorými manipulujeme a meníme ich umiestnenie už nie sme schopní registrovať len v hlave. Potrebovali by sme nejaké riešenie odolné voči zabúdaniu a schopné pamätať si veľa informácií a veľa veciach.

Môžeme si písať poznámky na lepiace papieriky alebo všetko zaznamenávať v tabuľkovom kalkulátore. Možno by to fungovalo, ale nie je to ono. Potrebovali by sme systém ušitý na mieru.

Hľadajme riešenie

Naprogramujme si vlastný systém pre správu kníh. Kým sa do toho pustíme, pozrime sa poriadne na to, čo chceme spraviť. Analyzujme:

- s akými dátami/informáciami budeme pracovať,
- ako tieto dáta/informácie reprezentovať,
- aké činnosti s dátami realizovať

Pokúsme sa nájsť nejakú hierarchiu v našom systéme. Uvažovať môžeme napríklad takto.

Primárne pracujeme s knihami. Bolo by teda vhodné nejako reprezentovať knihu.

Tých kníh je pomerne veľa. Bolo by vhodné mať nejaký systém, ktorý by vedel v knihách udržiavať poriadok.

Ak už nejaký systém naprogramujeme, mohol by mať príjemné ovládanie. Môžeme pre tento systém vytvoriť nejaké grafické rozhranie.

Pozrime sa na tieto tri časti podrobnejšie:

kniha:

- kniha môže byť tlačená alebo elektronická,
- každá kniha má autora, názov, ISBN a je niekde umiestnená,
    - tlačené knihy sú umiestnené na nejakej poličke v nejakom regály, predpokladajme, že regály a poličky číslujeme celými číslami väčšími ako 0,
    - elektronické knihy sú umiestnené v nejakom zariadení (počítač, tablet, čítačka),
- umiestnenie knihy, môžeme zmeniť,
- autora a názov knihy môžeme zmeniť (napr. v prípade preklepu),
- knihu budeme identifikovať podľa ISBN,
- ISBN knihy sa zmeniť nedá,
- nemal by sa dať vytvoriť nekorektný záznam o knihe,

knižničný systém:

- systém spravuje náš zoznam kníh,
- systém by mal umožniť:
    - pridať knihu do zoznamu kníh (ak kniha nie je v zozname),
    - vyradiť knihu zo zoznamu kníh (ak kniha je v zozname),
    - vyhľadávať knihy podľa autora, názvu,
    - usporiadať zoznam kníh podľa autora a podľa názvu,
    - uložiť zoznam kníh do súboru a načítať zoznam kníh so súboru,

grafické rozhranie:

- sprostredkováva komunikáciu medzi používateľom a knižničným systémom,
- pri spustení nahrá všetky záznamy o knihách,
- pri ukončení uloží všetky záznamy o knihách.

Implementovať uvedený systém nie je triviálna záležitosť. Už v predchádzajúcich kapitolách sme sa presvedčili, že použitie objektovo orientovaného programovania značne uľahčuje implementáciu takýchto systémov. Implementovať náš systém pre správu kníh bez OOP by bolo neprehľadné náročné. Použijeme ho preto aj v tomto prípade.

Poznámka na okraj

V tejto kapitole využijeme množstvo poznatkov, ktoré sme získali v predchádzajúch častiach. Pre rýchle pripomenutie môžeme využiť online dokumentáciu jazyka Python (117).

Keď veci sú aj nie sú rovnaké

V malej virtuálnej ZOO má majiteľ niekoľko zvierať a rád ich predvádza publiku. Každé zviera má svoje jedinečné meno, vie sa predstaviť a vydať nejaký zvuk. Ak by sme chceli navrhnúť triedu pre takéto virtuálne zvieratá, narazíme na pár problémov:

- každé zviera má meno, vie sa predstaviť a vydať nejaký zvuk,
  - môžeme definovať **jednu triedu zviera** a z nej vytvárať objekty pre konkrétne zvieratá,
- pes sa zrejme predstaví ako pes a zašteká, mačka sa predstaví ako mačka a zamňauká,
  - keďže sa každé zviera predstavuje inak, bolo by výhodnejšie definovať ***pre každý typ zvieraťa samostatnú triedu.***

Dospeli sme k dvom, na prvý pohľad rozporuplným záverom. Pri objektovom návrhu to nie je nič výnimočné a vieme to vyriešiť.

Výkladový text

Nové triedy nemusíme definovať od nuly. Novú triedu môžeme definovať tak, že použijeme ako základ nejakú inú triedu. Takáto trieda sa nazýva **rodičovská trieda** alebo **základná trieda**.

Novú triedu, ktorú vytvoríme z rodičovskej triedy, nazývame **odvodenou triedou** alebo **potomkom rodičovskej triedy**.

Odvodená trieda automaticky zdedí všetky metódy a atribúty rodičovskej triedy. Tomuto mechanizmu hovoríme **dedičnosť** a je to jedna zo základných vlastností objektovo orientovaného programovania.

```
class RodicovskaTrieda():
    #metódy a atribúty
    pass

class OdvodenaTrieda(RodicovskaTrieda):
    #metódy a atribúty zdedené z rodičovskej triedy
    #metódy a atribúty definované v odvodenej triede
    pass

#vytvorený objekt má všetko čo definovala RodicovskáTrieda +
#všetko, čo definovala OdvodenaTrieda
objekt = OdvodenaTrieda()
```

Ak novú triedu vytvárame z rodičovskej triedy, môžeme sa rozhodnúť, ktoré metódy z rodičovskej triedy preberieme do novej triedy bez zmeny a ktoré si prispôsobíme pre potreby novej triedy.

V odvodenej triede môžeme definovať metódu s rovnakým menom, ako mala metóda v rodičovskej triede. Ak to spravíme, tak v odvodenej triede nebudeme používať metódu z rodičovskej triedy, ale metódu, ktorú sme definovali v odvodenej triede. Pôvodná metóda z rodičovskej triedy sa síce nezmaže, ale už sa nedá v odvodenej triede tak jednoducho použiť. Tomuto mechanizmu hovoríme **prekrývanie**.

Poznámka na okraj

Všetky triedy, ktoré v Pythone vytvárame sú v skutočnosti odvodené triedy. Ak nepovieme inak, tak pre každú novú triedu sa použije rodičovská trieda s názvom `object`.

S dedičnosťou sme sa už stretli. Využili sme ju v kapitole „20. Programovanie obrázkov“ kde sme definovali vlastnú výnimku `TercChyba` odvodenú od triedy `Exception`.

Definícia rodičovskej triedy `Zviera` pre virtuálnu ZOO by mohla vyzerať nasledovne:

```
# zoo1.py

class Zviera():
    '''Trieda reprezentujuca zviera'''

    def __init__(self, meno):
        self.meno = meno

    def __set_meno(self, meno):
        self.__meno = meno

    def __get_meno(self):
        return self.__meno

    meno = property(__get_meno, __set_meno)

    def predstav_sa(self):
        '''Zviera sa predstavi.
        Tuto metodu je potrebne v odvodenej triede definovat.'''
        pass

    def ozvi_sa(self):
        '''Zviera sa ozve svojim zvukom.
        Tuto metodu je potrebne v odvodenej triede definovat.'''
        pass
```

Každé zviera má svoje meno. Preto sme v rodičovskej triede definovali všetko potrebné pre prácu s menom. Metódy `predstav_sa()` a `ozvi_sa()` zatiaľ nedokážeme zmysluplne definovať. Ich definícia záleží od konkrétneho zvieraťa.

Definujme triedy pre konkrétne druhy zvierat. Pre psa a mačku. Inicializáciu a metódy pre prácu s menom využijeme z rodičovskej triedy. Zvyšné metódy musíme definovať pre každý typ zvieraťa inak.

```
# zoo2.py
...

class Macka(Zviera):
    '''Trieda reprezentujuca macku.'''
    def __init__(self, meno):
        '''Inicializacia nastaveni vlastnosti macky.

        :param meno: meno macky
        :type meno: str
        '''
        super().__init__(meno)                    # [1]

    def predstav_sa(self):
```

```
        '''Macka sa predstavi.'''
        print(f'Ahoj, som mačka {self.meno}.')     # [2]

    def ozvi_sa(self):
        '''Macka sa ozve.'''
        print('mňau')

class Pes(Zviera):
    '''Trieda reprezentujuca psa.'''
    def __init__(self, meno):
        '''Inicializacia nastaveni vlastnosti psa.

        :param meno: meno psa
        :type meno: str
        '''
        super().__init__(meno)

    def predstav_sa(self):
        '''Pes sa predstavi.'''
        print(f'Ahoj, som pes {self.meno}.')

    def ozvi_sa(self):
        '''Pes sa ozve.'''
        print('hav hav')


mačka = Macka('Maggie')                            # [3]
mačka.predstav_sa()                                # Ahoj, som mačka Maggie.
mačka.ozvi_sa()                                    # mňau

pes = Pes('Rex')
pes.predstav_sa()                                  # Ahoj, som pes Rex.
pes.ozvi_sa()                                      # hav hav
```

[1] Definíciou metódy `__init__()` v triede `Macka` sme prekryli pôvodnú metódu `__init__()` z rodičovskej triedy `Zviera`. Ak potrebujeme v odvodenej triede zavolať metódu z rodičovskej triedy, využijeme na to funkciu `super()`. `super().__init()__` teda zavolá metódu `__init__()` rodičovskej triedy. Všimnime si, že sme jej odovzdali parameter meno. Rodičovská metóda `__init__()` sa postará o korektné spracovanie mena a nastavenie hodnoty atribútu `self.meno`.

[2] Atribút `self.meno` trieda `Macka` zdedila z triedy `Zviera`. Bez problémov ho teda môžeme použiť.

[3] Objekty vytvárame z tried `Macka` alebo `Pes`. Každý z nich sa vie predstaviť a vydať pre seba typický zvuk.

Úloha 1

Čo by sa stalo, keby sme v triede `Zviera` nedefinovali metódy `predstav_sa()` a `ozvi_sa()`? Svoju odpoveď si overte zmenou triedy `Zviera`.

Čo by sa stalo, keby sme v triedach `Macka` a `Pes` nedefinovali metódy `predstav_sa()` a `ozvi_sa()`? Svoju odpoveď si overte zmenou tried `Macka` a `Pes`.

Ak chceme programátorom, ktorí budú vytvárať potomkov triedy `Zviera`, jasne povedať, že ich odvodené triedy musia mať metódy `predstav_sa()` a `ozvi_sa()` a že ich musia v odvodených triedach prekryť, spravíme to nasledovne:

```
# zoo3.py

class Zviera():

    def __init__(self, meno):
        self.meno = meno

    def __set_meno(self, meno):
        self.__meno = meno

    def __get_meno(self):
        return self.__meno

    meno = property(__get_meno, __set_meno)

    def predstav_sa(self):                                  # [1]
        '''Zviera sa predstavi.
        Tuto metodu je potrebne v odvodenej triede prekryt.

        :raise NotImplementedError: Ak metoda nie je implementovana
        '''
        raise NotImplementedError('Metodu predstav_sa() musite v odvodenych
triedach prekryt.')

    def ozvi_sa(self):
        '''Zviera sa ozve svojim zvukom.
        Tuto metodu je potrebne v odvodenej triede prekryt.

        :raise NotImplementedError: Ak metoda nie je implementovana
        '''
        raise NotImplementedError('Metodu ozvi_sa() musite v odvodenych
triedach prekryt.')


class Macka(Zviera):
    def __init__(self, meno):
        super().__init__(meno)


mačka = Macka('Maggie')                                     # [2]
mačka.predstav_sa()

zver = Zviera('Neznamy druh')                               # [3]
zver.ozvi_sa()
```

Poznámka na okraj

Niektoré vývojové prostredia dokážu v odvodených triedach doplniť metódy, ktoré je potrebné prekryť. Napr. v prostredí Pycharm Edu nájdeme túto funkcionalitu v menu Code|Implement Methods...

[1] Ak sa niekto pokúsi vykonať metódu `predstav_sa()` z triedy `Zviera` neprejde to v tichosti. Metóda generuje výnimku s textom, že túto metódu musíme v odvodených triedach prekryť. Podobne to platí aj pre metódu `ozvi_sa()`.

[2] Ak v triede Macka neprekryjeme metódu `predstav_sa()`, takéto použitie skončí s chybou:
NotImplementedError: Metodu predstav_sa() musite v odvodenych triedach prekryt.

[3] Podobne skončí aj volanie metódy `ozvi_sa()` ak sme objekt vytvorili z triedy `Zviera`:
NotImplementedError: Metodu ozvi_sa() musite v odvodenych triedach prekryt.

Trieda Kniha

Úloha 2

Premyslite si, aké triedy je potrebné definovať pre záznam knihy v našom knižničnom systéme. Čo je výhodné definovať v rodičovskej triede (kniha) a čo v odvodených triedach (tlačená kniha, elektronická kniha)?

Definujte vhodné triedy pre záznam knihy. Definujte metódy tak, aby nebolo možné vytvoriť objekt s chybnými hodnotami. Riešenie uložte do súboru `kniha.py`.

Poznámka na okraj

Medzinárodné štandardné číslo knihy ISBN (International Standard Book Number) musí spĺňať isté pravidlá. Pri kontrole správnosti ISBN môžeme využiť niektorý z množstva modulov pre prácu s ISBN. Ich zoznam nájdeme na stránke https://pypi.org/.

Ak použijeme modul `isbnid`, kontrolu správnosti ISBN zrealizujeme nasledovne:

```
import isbn
zadane_isbn = '9788025131527'

try:
    isbn_id = isbn.ISBN(zadane_isbn)
except (isbn.ISBNError, isbn.ISBNRangeError):
    raise ValueError('Zadané ISBN je chybné.')
```

ISBN vieme zobraziť vo formáte, v ktorom sú jednotlivé časti oddelené znakom „-“.

```
print(isbn_id.hyphen())        # 978-80-251-3152-7
```

Majiteľ virtuálnej ZOO si na každé predstavenie vyberá do zoznamu účinkujúcich konkrétne zvieratá. Aby ich po predstavení odmenil, tak si chcel vypísať zoznam účinkujúcich.

```
...
mačka = Macka('Maggie')
pes = Pes('Rex')
šteniatko = Pes('Rexo')

zoznam_ucinkujucich = [mačka, pes, šteniatko]
```

```
for ucinkujuci in zoznam_ucinkujucich:
    print(ucinkujuci)
```

Výsledok ktorý dostal, ho prekvapil:

```
<__main__.Macka object at 0x000000793FAE5470>
<__main__.Pes object at 0x000000793FAE54A8>
<__main__.Pes object at 0x000000793FAE54E0>
```

Výkladový text

Ak potrebujeme definovať, ako má vyzerať textová reprezentácia objektu, spravíme to pomocou metódy `__str__()`. Túto metódu Python zavolá automaticky vždy, keď sa vyskytne požiadavka na textovú reprezentáciu objektu (napr. pri výpise).

Jednoduchou úpravou triedy `Zviera` docielime, že pri pokuse o výpis zoznamu účinkujúcich, sa nám zobrazia mená zvierat.

```
# zoo4.py

class Zviera():

    def __init__(self, meno):
        self.meno = meno

    def __set_meno(self, meno):
        self.__meno = meno

    def __get_meno(self):
        return self.__meno

    meno = property(__get_meno, __set_meno)

    def predstav_sa(self):  # [1]
        raise NotImplementedError('Metodu predstav_sa() musite v
odvodenych triedach prekryt.')

    def ozvi_sa(self):
        raise NotImplementedError('Metodu ozvi_sa() musite v
odvodenych triedach prekryt.')

    def __str__(self):
        '''Vrati textovu reprezentaciu zvierata

        :return: textova reprezentacia zvierata
        :rtype: str
        '''
        return self.meno
```

Následný výpis už zodpovedá očakávanému výstupu:

```
mačka = Macka('Maggie')
pes = Pes('Rex')
šteniatko = Pes('Rexo')

zoznam_ucinkujucich = [mačka, pes, šteniatko]

for ucinkujuci in zoznam_ucinkujucich:
    print(ucinkujuci)
```

```
Maggie
Rex
Rexo
```

Výkladový text

Všimnime si ešte jednu zaujímavú vec. V zozname zvierat máme rôzne druhy zvierat. Ak by sme chceli, aby sa každé z nich ozvalo svojim typickým zvukom, môžeme to spraviť veľmi jednoducho:

```
mačka = Macka('Maggie')
pes = Pes('Rex')
šteniatko = Pes('Rexo')

zoznam_ucinkujucich = [mačka, pes, šteniatko]

for ucinkujuci in zoznam_ucinkujucich:
    ucinkujuci.ozvi_sa()
```

Nemusíme rozlišovať, či zviera je pes alebo mačka. Jednoducho zavoláme jeho metódu `ozvi_sa`. Tejto vlastnosti hovoríme **polymorfizmus** a je to jedna zo základných vlastností objektovo orientovaného programovania. Vďaka polymorfizmu môžeme pre rôzne typy objektov (napr. `Pes`, `Macka`) využívať rovnaké rozhranie (metóda `ozvi_sa`). Napriek tomu, že rozhranie je rovnaké, každý objekt vykoná inú činnosť.

Len pre úplnosť dodáme, že polymorfizmus by fungoval rovnako dobre aj v prípade, kedy by triedy `Pes` a `Macka` nemali rovnakú rodičovskú triedu.

Úloha 3

V triedach pre reprezentáciu kníh definujte metódu `__str__()` tak, aby sme pri výpise knihy dostali všetky informácie o knihe. Zamyslite, kde všade je potrebné metódu `__str__()` definovať?

Pomôcka: Niekedy je potrebné z takejto textovej podoby spätne extrahovať jednotlivé údaje objektu. Navrhnite takú formu textovej reprezentácie, aby sa jednotlivé údaje dali z reťazca ľahko extrahovať.

Zvieratkám vo virtuálnej ZOO sa zapáčilo vystupovanie a hlavne odmena za vystúpenie. Niektorým zvieratkám sa podarilo v jednom predstavení vystúpiť viac krát. To sa majiteľovi nepáčilo, lebo chce, aby si zarobilo každé zvieratko. Majiteľ virtuálnej ZOO začal preto

kontrolovať, či zvieratko, ktoré chce v predstavení vystúpiť už nie je v zozname účinkujúcich. Ale jeho riešenie neprinieslo požadovaný efekt.

```
# zoo5.py
...

mačka = Macka('Maggie')
pes = Pes('Rex')
iny_pes = Pes('Rex')

zoznam_ucinkujucich = []
if mačka not in zoznam_ucinkujucich:
    zoznam ucinkujucich.append(mačka)
if pes not in zoznam_ucinkujucich:
    zoznam_ucinkujucich.append(pes)
if iny_pes not in zoznam_ucinkujucich:
    zoznam_ucinkujucich.append(iny_pes)

for ucinkujuci in zoznam_ucinkujucich:
    print(ucinkujuci)
```

```
Maggie
Rex
Rex
```

Výkladový text

Ak potrebujeme definovať, podľa akých kritérií sa majú objekty porovnávať (operátor `==`), spravíme to pomocou metódy `__eq__()`.

Pri vyhodnotení situácie `objekt1 == objekt2`, Python zavolá metódu `__eq__()` objektu `objekt1` a ako parameter jej predá `objekt2`

Jednoduchou úpravou triedy `Zviera` definujeme, ako sa majú zvieratá porovnávať a zamedzíme tomu, aby nejaké zviera vystúpilo v jednom predstavení viac krát.

```
#zoo6.py
class Zviera():
    def __init__(self, meno):
        self.meno = meno

    def __set_meno(self, meno):
        self.__meno = meno

    def __get_meno(self):
        return self.__meno

    meno = property(__get_meno, __set_meno)

    def predstav_sa(self):
        pass

    def ozvi_sa(self):
        pass

    def __str__(self):
        return self.meno
```

```
    def __eq__(self, other):
        return self.meno == other.meno
```

```
mačka = Macka('Maggie')
pes = Pes('Rex')
iny_pes = Pes('Rex')

zoznam_ucinkujucich = []
if mačka not in zoznam_ucinkujucich:
    zoznam_ucinkujucich.append(mačka)
if pes not in zoznam_ucinkujucich:
    zoznam_ucinkujucich.append(pes)
if iny_pes not in zoznam_ucinkujucich:
    zoznam_ucinkujucich.append(iny_pes)

for ucinkujuci in zoznam_ucinkujucich:
    print(ucinkujuci)
```

Ak zviera už je v zozname účinkujúcich, tak ho už do zoznamu nepridáme.

```
Maggie
Rex
```

Úloha 4

Na začiatku sme si definovali, že jednoznačným identifikátorom knihy bude jej ISBN. Upravte triedy pre reprezentáciu kníh tak, aby knihy, ktoré majú rovnaké ISBN, boli vyhodnotené ako rovnaké.

Pomôcka: Pri definovaní metódy `__eq__()` je vhodné myslieť aj na prípad, že objekt porovnávame s objektom iného typu. V takomto prípade by test rovnosti mal skončiť s hodnotou `False`. Overiť, či objekt je inštanciou danej triedy (presnejšie typu), môžeme pomocou funkcie `isinstance()`.

Poznámka na okraj

Objekty môžeme porovnávať nie len na rovnosť. Aj pre ostatné operátory porovnania môžeme definovať príslušné metódy. Viac na https://docs.python.org/3.6/reference/datamodel.html#richcmpfuncs [21. 6. 2019].

To, či je potrebné implementovať aj ostatné metódy porovnania, záleží na konkrétnom prípade.

Trieda Kniznica

V tejto chvíli máme všetko potrebné pre reprezentáciu knihy pripravené. Začneme budovať systém pre správu kníh. Pripomeňme si, čo by mal systém umožňovať:

- pridať knihu do zoznamu kníh (ak kniha nie je v zozname),
- vyradiť knihu zo zoznamu kníh (ak kniha je v zozname),
- vyhľadávať knihy podľa autora, názvu,
- usporiadať zoznam kníh podľa autora a podľa názvu,

- uložiť zoznam kníh do súboru a načítať zoznam kníh so súboru,

Úloha 5

Vytvorte triedu `Kniznica` a implementujte metódy na pridanie knihy do zoznamu kníh a na vyradenie knihy zo zoznamu kníh.

Premyslite si, v akej dátovej štruktúre budete udržiavať zoznam kníh.

Využite metódy, ktoré sme definovali v triedach pre knihy.

Riešenie uložte do súboru `kniznica.py`.

Pomôcka: V triede `Kniznica` spravujeme len knihy (tlačené a elektronické). Trieda by nemala umožniť do zoznamu kníh pridať iné typy objektov.

Výkladový text

V zozname kníh chceme registrovať len knihy. Ak by niekto vytvoril objekt triedy `Kniha`, takýto objekt by sme nemali do nášho zoznamu vložiť. Takýto objekt nepredstavuje reálnu knihu. Povolené sú len objekty tých tried, ktoré sú odvodené od triedy `Kniha`. Momentálne sú to len tlačená a elektronická kniha. Ak by sme testovali len tieto dva typy kníh, prídeme o budúcu možnosť registrovať aj iné typy kníh. Povolenou hodnotou by teda mali byť len objekty tried, ktoré sú odvodené od triedy `Kniha`. Test, či objekt `kniha` je objektom triedy odvodenej z triedy `Kniha`, zrealizujme nasledovne:

```
if not issubclass(type(kniha), (Kniha)):
    raise ValueError('Pokúšate sa vložiť objekt, ktorý nie je kniha.')
```

Majiteľa virtuálnej ZOO trochu zdržiava fakt, že vo výpise účinkujúcich zvierat nie sú tieto usporiadané. Sú vypísané v takom poradí v akom ich do zoznamu pridal. Rozhodol sa preto, že zoznam zvierat si usporiada podľa mena.

Výkladový text

Objekty triedy zoznam majú metódu `sort`. Táto metóda preusporiada prvky zoznamu tak, aby boli prvky zoznamu usporiadané, napr.:

```
zoznam = ['b', 'abc', 'cc']
zoznam.sort()
print(zoznam)     # ['abc', 'b', 'cc']
```

V metóde `sort` je implementovaný algoritmus usporadúvania. Vzájomným porovnávaným prvkov zoznamu tieto prvky usporiada. V tomto prípade lexikograficky.

Aby metóda `sort` vedela prvky zoznamu usporiadať, prvky zoznamu musia byť vzájomne porovnateľné. Stačí, ak tieto objekty majú implementovanú metódu `__lt__` pre operátor porovnania `<`.

Čo ale v prípade, ak potrebujeme prvky usporiadať podľa nejakého iného kľúča? Napr. podľa dĺžky reťazca.

Metóda `sort` má aj ďalší parameter `key`. Jeho hodnotou je názov funkcie, ktorá má len jeden parameter a vracia hodnotu kľúča pre zadaný prvok. Ak by sme chceli prvky zoznamu usporiadať podľa dĺžky (funkcia `len()`), spravíme to nasledovne:

```
zoznam.sort(key=len)
print(zoznam)      # ['b', 'cc', 'abc']
```

Samozrejme, môžeme si definovať aj vlastné funkcie pre iné kľúče. Napr. chceme prvky usporiadať podľa toho, či majú párny alebo nepárny počet znakov. Na začiatku budú reťazce párnej dĺžky a potom tie s nepárnou dĺžkou. Funkcia, ktorá vyhodnotí párnosť dĺžky reťazca môže vyzerať nasledovne:

```
def parnost(retazec):
    return len(retazec) % 2
```

Zoznam reťazcov podľa párnosti ich dĺžok usporiadame nasledovne:

```
zoznam.sort(key=parnost)
print(zoznam) # ['cc', 'abc', 'b']
```

Definovať funkciu na jedno použitie, len pre potreby zistenia hodnoty kľúča pri usporadúvaní, uberá z prehľadnosti kódu. Najmä v prípade, ak táto funkcia má len jeden riadok s jednoduchým výrazom. Pre takéto prípady Python ponúka možnosť použiť výraz `lambda`. `Lambda` výraz vygeneruje bezmennú funkciu, ktorú priamo na mieste môžeme použiť.

Usporiadanie reťazcov v zozname podľa párnosti ich dĺžok využitím `lambda` výrazu spravíme nasledovne:

```
zoznam.sort(key=lambda retazec: len(retazec) % 2)
print(zoznam) # ['cc', 'abc', 'b']
```

V `lambda` výraze pred dvojbodkou uvádzame zoznam parametrov. Za dvojbodkou výraz, ktorého hodnota sa použije ako návratová hodnota funkcia. `Lambda` výraz v Pythone nájde uplatnenie pomerne často. Použiť ho môžeme všade tam, kde Python očakáva len meno funkcie ale my potrebujeme túto funkciu zavolať aj s nejakými parametrami.

Pre ktorú možnosť usporadúvania prvkov sa nakoniec rozhodneme, záleží od konkrétnej situácie. Nikdy by sme ale nemali zabudnúť na to, že čitateľnosť kódu má prednosť pred jeho skracovaním.

Ak sa pokúsime zoznam zvierat, bez predchádzajúcej úpravy triedy `Zviera` usporiadať, skončí to s chybou.

```
zoznam_ucinkujucich.sort()
for ucinkujuci in zoznam_ucinkujucich:
    print(ucinkujuci)
```

TypeError: '<' not supported between instances of 'Pes' and 'Macka'

Python nevie, ako má vyhodnotiť porovnanie zvierat operátorom `<`. Máme dve možnosti. Definujeme metódu `__lt__` pre porovnanie alebo definujeme funkciu ktorú predáme ako hodnotu parametra `key` metóde `sort`.

```
#zoo7.py
class Zviera():
    def __init__(self, meno):
        self.meno = meno

    def __set_meno(self, meno):
        self.__meno = meno

    def __get_meno(self):
        return self.__meno

    meno = property(__get_meno, __set_meno)

    def predstav_sa(self):
        pass

    def ozvi_sa(self):
        pass

    def __str__(self):
        return self.meno

    def __eq__(self, other):
        return self.meno == other.meno

    def __lt__(self, other):                                    # [1]
        return self.meno < other.meno

pes = Pes('Rex')
šteniatko = Pes('Rexo')
mačka = Macka('Maggie')

zoznam_ucinkujucich = [pes, šteniatko, mačka]
for ucinkujuci in zoznam_ucinkujucich:
    print(ucinkujuci)      #Rex Rexo Maggie

zoznam_ucinkujucich.sort()                                      # [2]
for ucinkujuci in zoznam_ucinkujucich:
    print(ucinkujuci)      #Maggie Rex Rexo

zoznam_ucinkujucich.sort(key=lambda zviera: zviera.meno)     # [3]
for ucinkujuci in zoznam_ucinkujucich:
    print(ucinkujuci)      #Maggie Rex Rexo
```

[1] Definíciou metódy `__lt__()` určíme, ako sa má vyhodnotiť porovnanie operátorom `<`. Ak porovnáme dve zvieratá, Python porovná ich mená.
`zviera1 < zviera2` ⇔ `zviera1.meno < zviera2.meno`.

[2] Metóda `sort` používa pri usporadúvaní prvkov na ich vzájomné porovnanie operátorom `<`. Využije sa metóda `__lt__()`.

[3] Rovnaký výsledok dosiahneme aj použitím `lambda` výrazu. Pri porovnaní dvoch zvierat pri usporadúvaní, porovná Python výsledky `lambda` funkcie, t.j. mená zvierat.

Úloha 6

Knihy v našej knižnici chceme knihy usporadúvať podľa autora aj podľa názvu. Definujte metódu `usporiadaj`, ktorá podľa zadaného kľúča zabezpečí usporiadanie kníh podľa autora alebo podľa názvu.

Úloha 7

Definujte metódu `nájdi_knihy`, ktorá prehľadá zoznam kníh a vráti zoznam tých kníh, ktoré vyhovujú zadaným kritériám. Vyhľadávanie chceme realizovať podľa autora alebo podľa názvu knihy.

Pomôcka: Premyslite si, či pri vyhľadávaní akceptujete len presnú zhodu alebo stačí časť zhody.

Aj keď sme pôvodne nechceli v knihách vyhľadávať podľa ISBN, neskôr sa tomu nevyhneme. ISBN je totiž jednoznačný identifikátor knihy. Vhodné je preto implementovať aj vyhľadávanie podľa ISBN.

Úloha 8

Definujte metódy `uloz` a `nahraj`. Metóda `uloz` uloží zoznam kníh do súboru. Metóda `nahraj`, nahrá zoznam kníh zo súboru.

Pomôcka: Poznáte rôzne typy súborov, resp. formátov. Premyslite si, v akom formáte zoznam kníh uložíte.

Výkladový text

Prípady, keď pracujeme s pomerne komplikovanými objektami (napr. objekty tried `TlacenaKniha` a `ElektronickaKniha`) nie sú zriedkavé. Situácia sa ešte skomplikuje, ak takýto objekt chceme do súboru uložiť alebo zo súboru prečítať. V týchto prípadoch je výhodné objekty transformovať do formy, ktorá nám zjednoduší manipuláciu s nimi a umožní ich uloženie. Tento proces sa nazýva serializácia.

Pri binárnej serializácii sa objekt prevedie do tvaru postupnosti bajtov tak, ako je reprezentovaný v pamäti počítača. Takúto reprezentáciu vieme priamo do súboru zapísať alebo zo súboru prečítať. Serializované dáta vieme deserializovať a vytvoriť z nich pôvodný objekt. Na serializáciu dát použijeme modul `pickle`, ktorý implementuje binárny protokol pre serializáciu a deserializáciu. Viac o module `pickle` nájdeme na: https://docs.python.org/3/library/pickle.html.

```
import pickle                                                    [1]
data = [1, 2, [3, 4], (5, 6), 'text', 3.14]
with open('binarny_subor.bin', 'wb') as f:                       [2]
    pickle.dump(data, f)                                         [3]

with open('binarny_subor.bin', 'rb') as f:                       [4]
    kopia = pickle.load(f)                                       [5]


print(data)      #[1, 2, [3, 4], (5, 6), 'text', 3.14]           [6]
print(kopia)     #[1, 2, [3, 4], (5, 6), 'text', 3.14]
```

```
print(data == kopia) #True                                    [7]
print(data is kopia) #False                                   [8]
```

[1] Importujeme modul `pickle`. Modul `pickle` je súčasťou štandardnej inštalácie jazyka Python. Pozor, okrem Pythonu tento protokol iné aplikácie/programovacie jazyky nepoznajú. Navyše existuje niekoľko verzií tohto protokolu. Ak nepovieme inak, použije sa pri serializácii vždy najnovšia verzia a pri deserializácii tá, ktorou boli dáta serializované (ak je k dispozícii).

[2] Súbor otvoríme pre zápis „`w`“ v binárnom režime „`b`“.

[3] `pickle.dump()` dáta serializuje a zapíše do súboru.

[4] Súbor otvoríme pre čítanie „`r`“ v binárnom režime „`b`“.

[5] `pickle.load()` prečíta dáta zo súboru, deserializuje ich a vráti ich pôvodnú reprezentáciu. Deserializované dáta sme priradili do premennej `kopia`.

Nasledujúce kroky nie je potrebné robiť. Spravili sme ich len pre kontrolu a demonštráciu serializácie.

[6] Pre kontrolu vypíšeme pôvodný objekt „`data`“ a jeho kópiu „`kopia`“, ktorú sme získali deserializáciou. Vidíme, že hodnoty premenných sú rovnaké.

[7] Ak porovnáme obidva objekty, zistíme, že sú zhodné.

[8] Ak otestujeme, či „`data`“ a „`kopia`“ sú jeden a ten istý objekt, zistíme, že nie. Získali sme teda dva nezávisle objekty s rovnakým obsahom.

Na začiatku sme pre grafické rozhranie definovali požiadavky:

- *pri spustení nahrá všetky záznamy o knihách,*
- *pri ukončení uloží všetky záznamy o knihách.*

Ak budeme prostredie spúšťať prvý krát, automatické načítanie kníh zo súboru by spôsobilo chybu. Dátový súbor ešte neexistuje. V tomto prípade môžeme dátový súbor vytvoriť (i keď zrejme nebude obsahovať žiadne záznamy kníh).

Trieda pre grafické rozhranie

V tejto chvíli máme všetko potrebné pre manipuláciu s knihami v knižnici pripravené. Môžeme vytvoriť grafické rozhranie, ktoré bude slúžiť ako prostredník medzi používateľom a knižnicou. V tejto kapitole nepôjdeme spoločne krok po kroku. Akú funkcionalitu by malo grafické rozhranie ponúkať vyplýva z úvodnej analýzy a z funkcionality triedy `Kniznica`. V nasledujúcom texte uvedieme niekoľko informácií, ktoré nám môžu pomôcť pri vytváraní grafického rozhrania pre knižničný systém. Pre ďalšie informácie odporúčame (118).

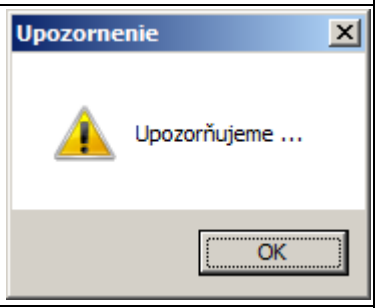
Okno pre chyby

Triedy `Kniha` a `Kniznica` sme navrhli tak, aby v prípade nejakého problému vyhodili výnimku aj s popisom problematickej situácie. Keď bude naše grafické rozhranie komunikovať

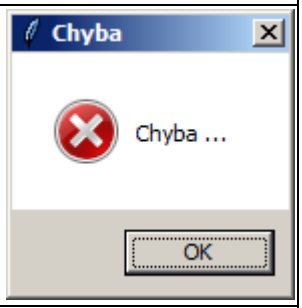
s uvedenými triedami, mali by sme odchytávať všetky vyhodené výnimky. V prípade zachytenia výnimky, by sme mali používateľa na problém upozorniť. Jednou z možností je zobraziť ďalšie okno s popisom chyby. Využiť môžeme modul `tkinter.messagebox` a niektorú z jeho funkcií:

```
import tkinter.messagebox
```

```
tkinter.messagebox.showwarning('Upozornenie',
'Upozorňujeme ...')
```




```
tkinter.messagebox.showerror('Chyba', 'Chyba ...')
```



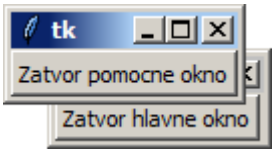

```
tkinter.messagebox.showinfo('Info', 'Info ...')
```

Viac informácii v (118).

Pomocné okno

V triede `Kniznica` sme definovali pomerne veľa metód pomocou ktorých vieme so zoznamom kníh pracovať. Každá z metód vyžaduje iné vstupy. Bolo by nepraktické, mať pre všetky tieto vstupy neustále zobrazené ovládacie prvky. V okne aplikácie by sme mali priveľa prvkov, ktoré by boli väčšinu času nevyužité. Situáciu môžeme vyriešiť tak, že v prípade potreby otvoríme pomocné okno. Využijeme na to prvok `tkinter.Toplevel`.

```
import tkinter

hlavne_okno = tkinter.Tk()
tkinter.Button(hlavne_okno,
               text='Zatvor hlavne okno',
               command=hlavne_okno.destroy).grid()

pomocne_okno = tkinter.Toplevel(hlavne_okno)
pomocne_okno.attributes('-topmost', 'true')
tkinter.Button(pomocne_okno,
               text='Zatvor pomocne okno',
               command=pomocne_okno.destroy).grid()
```



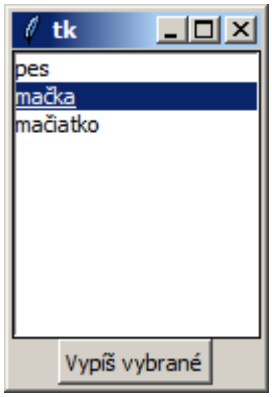
```
pomocne_okno.mainloop()
hlavne_okno.mainloop()
```

Niekedy je problém s poradím okien a okná sa nevhodne prekrývajú. Nastavením atribútu `-topmost` na hodnotu `True` docielime, že pomocné okno bude stále navrchu.

Zobrazenie prvkov zoznamu

Knihy zo zoznamu kníh by sme mali zobraziť ako zoznam. Každú knihu do samostatného riadku. Keď bude používateľ chcieť vykonať nejakú činnosť s vybranou knihou, musíme identifikovať, ktorú knihu v zobrazenom zozname označil. Na tento účel môžeme použiť prvok `tkinter.Listbox`.

```
import tkinter

def vypis_vybrane():
    try:
        poradie_vybrateho = zoznam_box.curselection()[0]
        vybrate = zoznam_box.get(poradie_vybrateho)
        print(vybrate)
    except:
        print('Nic vybraté')

hlavne_okno = tkinter.Tk()

zoznam_box = tkinter.Listbox()
zoznam_box.grid()

zoznam = ['pes', 'mačka', 'mačiatko']
zoznam_box.delete(0, 'end')
for prvok in zoznam:
    zoznam_box.insert('end', prvok)

tkinter.Button(hlavne_okno,
               text='Vypíš vybrané',
               command=vypis_vybrane).grid()

hlavne_okno.mainloop()
```



Viac informácií v (118).

Úloha 9

Navrhnite a implementujte grafické rozhranie pre knižničný systém. Riešenie uložte do súboru `gui_kniznica.py`.

Čo sme sa naučili

- navrhovať triedy pre reprezentáciu podobných objektov,
- navrhovať triedy pre správu objektov,
- využívať dedičnosť a polymorfizmus,
- využívať objektový prístup pri riešení problémov,
- vytvárať grafické rozhranie pre komunikáciu medzi používateľom a objektmi.

Ďalšie úlohy na precvičenie a zamyslenie

1. V návrhu knižničného systému sme uvažovali len dva typy kníh: tlačené a elektronické. Upravte aplikáciu tak, aby bolo možné registrovať aj audio knihy. Audio kniha má rovnaké vlastnosti ako elektronická kniha a navyše pri nej uvažujeme aj o dĺžke v minútach.
2. V návrhu knižničného systému sme neuvažovali o tom, že knihu môžeme niekomu zapožičať. Doplňte aplikáciu tak, aby sme mohli registrovať aj skutočnosti, že kniha je požičaná a komu je požičaná.

   Poznámka: O požičiavaní kníh zväčša uvažujeme len v prípade tlačených kníh. Elektronické a audio knihy môžu mať zabezpečenie, ktoré nedovoľuje ich prenos na iné zariadenia. Je na vašom zvážení, či možnosť zapožičania implementujete aj pre elektronické a audio knihy.

3. Predstavte si, že časť kníh niekomu darujeme.
   - Implementujte možnosť, aby sme vo výpise kníh mohli označiť viac kníh a ich zoznam exportovať do samostatného súboru.
   - Z pôvodného zoznamu sa tieto knihy po exporte odstránia.
   - Implementujte možnosť importovať knihy (z exportovaného zoznamu) do zoznamu kníh.

   Pomôcka: Prvok Listbox štandardne umožňuje vybrať len jednu položku. Ak nastavíme hodnotu parametra `selectmode=tkinter.MULTIPLE`*, môžeme vybrať viac položiek. Metóda* `curselection()` *vráti zoznam indexov vybraných položiek.*

Bibliografia

1. **Python Software Foundation.** Welcome to Python.org. *Python.* [Online] https://www.python.org/.

2. —. History and License — Python 3.8.6rc1 documentation. *Welcome to Python.org.* [Online] 10. 09 2020. https://docs.python.org/3/license.html.

3. **JetBrains s.r.o.** PyCharm Edu. *etBrains: Developer Tools for Professionals and Teams.* [Online] JetBrains s.r.o., 2020. https://www.jetbrains.com/pycharm-edu/.

4. **Wikipedia contributors.** Turtle graphics. *Wikipedia, The Free Encyclopedia.* [Online] https://en.wikipedia.org/w/index.php?title=Turtle_graphics&oldid=972979529.

5. **WHO.** Body mass index - BMI. *WHO/Europe.* [Online] 2020. https://www.euro.who.int/en/health-topics/disease-prevention/nutrition/a-healthy-lifestyle/body-mass-index-bmi.

6. **Wikipedia contributors.** Pig Latin. *Wikipedia, The Free Encyclopedia.* [Online] 09. 09 2020. https://en.wikipedia.org/w/index.php?title=Pig_Latin&oldid=977611177.

7. **Amos, David.** Python GUI Programming With Tkinter. *Real Python Tutorials.* [Online] Real Python, 2020. https://realpython.com/python-gui-tkinter/.

8. *Simulácia ako vedecká metóda.* **Paholok, Igor.** XV., Praha : Department of Philosophy of University of Economics, 2008, E-LOGOS. ISSN 1211-0442.

9. **Budíková, Marie.** *Zákon velkých čísel.* Brno, Česká republika : Masarykova univerzita, Brno, 2016. Statistika a pravděpodobnost. ISBN 978-80-210-8206-9.

10. **Přispěvatelé Wikipedie.** Monty Hallův problém. *Wikipedie: Otevřená encyklopedie.* [Online] 26. 05 2020. https://cs.wikipedia.org/w/index.php?title=Monty_Hall%C5%AFv_probl%C3%A9m&oldid=18555588.

11. *O metóde Monte Carlo a možnostiach jej aplikácií.* **Knežo, Dušan.** 22, Košice : Ústav technológií a manažmentu Strojníckej fakulty Technickej univerzity v Košiciach, 2012, Transfer inovácií. ISSN 1337-7094.

12. **Wikipedia contributors.** Projectile motion. [Online] 6. 5 2018. https://en.wikipedia.org/wiki/Projectile_motion#Trajectory_of_a_projectile_with_air_resistance.

13. **Rouinfar, Amy.** *Projectile Motion - Kinematics | Air Resistance | Parabolic Curve - PhET Interactive Simulations.* [Online] 2018. https://phet.colorado.edu/en/simulation/projectile-motion.

14. **Khan Academy.** *Discovery of the electron and nucleus.* [Online] 2018. https://www.khanacademy.org/science/chemistry/electronic-structure-of-atoms/history-of-atomic-structure/a/discovery-of-the-electron-and-nucleus.

15. **Wikipedia.** *Epicycloid.* [Online] 31. 8 2018. https://en.wikipedia.org/wiki/Epicycloid.

16. —. *Hypocycloid.* [Online] 24. 4 2018. https://en.wikipedia.org/wiki/Hypocycloid.

17. —. *Hypotrochoid.* [Online] 31. 8 2018. https://en.wikipedia.org/wiki/Hypotrochoid.

18. —. *Cycloid.* [Online] 26. 6 2018. https://en.wikipedia.org/wiki/Cycloid.

19. —. *Cyclogone.* [Online] 4. 2 2017. https://en.wikipedia.org/wiki/Cyclogon.

20. **Šnábl, Ivo.** *Matematická biologie učebnice: Psí křivka.* [Online] 2018. http://portal.matematickabiologie.cz/index.php?pg=analyza-a-modelovani-dynamickych-biologickych-dat--spojite-deterministicke-modely-i--nektere-klasicke-ulohy--psi-krivka.

21. **Cápay, Martin.** Od bunkových automatov k autonómnym agentom. [aut. knihy] Martin Drlík a Peter Švec. *ASIS 2008 – Adaptívne siete v informačných systémoch : Štúdie vybraných tém adaptívnych hypermediálnych systémov a umelej inteligencie.* Nitra : UKF, 2008.

22. **Wikipedia, contributors.** Brute-force search. *Wikipedia, The Free Encyclopedia.* [Online] 01. 04 2019. [Dátum: 29. 05 2019.] https://en.wikipedia.org/w/index.php?title=Brute-force_search&oldid=890487309.

23. **Forišek, Michal a Šišková, Juliana.** 2.Pažravé algoritmy. *Kapitoly z informatiky 1.* Bratislava : Štátny pedagogický ústav, Pluhová 8, 830 00 Bratislava, 2010, s. 14-19.

24. **Andrejková, Gabriela, Forišek, Michal a Šišková, Juliana.** Časová zložitosť. *Kapitoly z informatiky.* Bratislava : Štátny pedagogický ústav, Pluhová 8, 830 00 Bratislava, 2010, s. 6-12.

25. **Marek Bundzel.** Prírodou inšpirované algoritmy. *Čo je to Celulárny Automat.* [Online] [Dátum: 23. 8 2020.] https://dendrit.tuke.sk/~newalife/kapitola/500/.

26. **Wikipedia contributors.** Conway's Game of Life. *Wikipedia, The Free Encyclopedia.* [Online] 01. 09 2020. https://en.wikipedia.org/w/index.php?title=Conway%27s_Game_of_Life&oldid=976208010.

27. **Klein, Bernd.** Properties vs. Getters and Setters. *Python3 Tutorial.* [Online] 2018. https://www.python-course.eu/python3_properties.php.

28. **Python Software Foundation.** 2. Built-in Functions — Python 3.6.12 documentation. *Python.* [Online] 19. 08 2020. https://docs.python.org/3.6/library/functions.html#property.

29. —. PEP 20 -- The Zen of Python. *Welcome to Python.org.* [Online] 22. 08 2004. https://www.python.org/dev/peps/pep-0020/.

30. **Naraghi, Aryan.** An implementation of the forest-fire model. *Aryan Naraghi.* [Online] [Dátum: 24. 05 2019.] http://aryannayra.com/randomstuff/forestfire.html.

31. **Shodor.** Interactivate: Fire!! *Shodor: A National Resource for Computational Science Education.* [Online] [Dátum: 21. 05 2019.] http://www.shodor.org/interactivate/activities/Fire/.

32. **Wikipedia contributors.** Forest-fire model. *Wikipedia, The Free Encyclopedia.* [Online] 24. 12 2016. https://en.wikipedia.org/w/index.php?title=Forest-fire_model&oldid=756417243.

33. **Bezák, Pavol a Iringová, Miriam.** OPTIMALIZÁCIA POMOCOU GENETICKÝCH ALGORITMOV. [Online] [Dátum: 23. 8 2020.] https://www.mtf.stuba.sk/buxus/docs//internetovy_casopis/2008/8/bezak-iringova.pdf.

34. **Kvasnička, Vladimír, Pospíchal, Jiří a Tiňo, Peter.** *Evolučné algoritmy.* Bratislava : STU, 2000.

35. **Skalka, Ján a Cápay, Martin.** Evolúcia ako prirodzený spôsob hľadania riešení. *ASIS 2008 – Adaptívne siete v informačných systémoch : Štúdie vybraných tém adaptívnych hypermediálnych systémov a umelej inteligencie.* Nitra : UKF, 2008.

36. **Marczyk, Adam.** Genetic Algorithms and Evolutionary Computation. [Online] 23. 4 2004. [Dátum: 23. 8 2020.] http://www.talkorigins.org/faqs/genalg/genalg.html.

37. **Ficek, Tomáš.** Bakalárska práca. *Genetické algoritmy a návrh modelu živého organizmu so sociálnym a teritoriálnym správaním.* Nitra : UKF, 2009.

38. **Chuang, John.** Mozart's Musikalisches Würfelspiel. *http://sunsite.univie.ac.at/Mozart/dice/.* [Online] 1995. [Dátum: 21. 6 2019.]

39. **Barnig, Marco.** Musical Dice Games. *https://www.web3.lu/musical-dice-games/.* [Online] 2013. [Dátum: 21. 6 2019.]

40. **Homola, Martin a kol.** *Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika : Web, Multimédiá.* Bratislava : ŠPÚ, 2010. s. 68. ISBN 978-80-8118-051-4.

41. **Cornell University.** Mozart's Dice Game. *http://www.cs.cornell.edu/courses/cs1110/2008fa/assignments/a6/a6.html.* [Online] 2008. [Dátum: 21. 6 2019.]

42. **Tiwari, Shantnu.** Audio and Digital Signal Processing (DSP) in Python. *Python for Scientist and Engineers.* [Online] [Dátum: 21. 6 2019.] https://www.pythonforengineers.com/audio-and-digital-signal-processingdsp-in-python/.

43. **Chris Corzine.** Digital Audio - Creating a WAV (RIFF) file. [Online] [Dátum: 21. 6 2019.] http://www.topherlee.com/software/pcm-tut-wavformat.html.

44. **Python Audio.** Audio Programming In Python. [Online] 2014. [Dátum: 21. 6 2019.] http://pythonaudio.blogspot.com/2014/04/3-reading-wave-file.html.

45. **Arnold, Douglas N.** *The Patriot Missile Failure.* [Online] 2000. http://www-users.math.umn.edu/~arnold/disasters/patriot.html.

46. **Arnold, Douglas N.** *The Explosion of the Ariane 5.* [Online] 2000. http://www-users.math.umn.edu/~arnold/disasters/ariane.html.

47. **Neumann, Peter G.** *The RIsks Digest.* [Online] 2019. http://catless.ncl.ac.uk/Risks/.

48. **Python Software Foundation.** *Welcome to Python.org.* [Online] 2019. https://www.python.org/shell/.

49. **Lefèvre, Vincent.** *Double-precision floating-point format.* [Online] 2019. https://en.wikipedia.org/wiki/Double-precision_floating-point_format.

50. **wikiHow.** *How to Convert a Number from Decimal to IEEE 754 Floating Point Representation.* [Online] 2019. https://www.wikihow.com/Convert-a-Number-from-Decimal-to-IEEE-754-Floating-Point-Representation.

51. **Grondin, Francois.** *Double (IEEE754 Double precision 64-bit) Converter.* [Online] 2014. http://www.binaryconvert.com/convert_double.html .

52. **Regan, Rick.** *Decimal to Floating-Point Converter - Exploring Binary.* [Online] 2019. https://www.exploringbinary.com/floating-point-converter/ .

53. **Schmidt, H.** *IEEE-754 Floating Point Converter.* [Online] 2017. https://www.h-schmidt.net/FloatConverter/IEEE754.html.

54. **Wen, Quanfei a Brewer, Kevin J.** *IEEE-754 Floating-Point Conversion from Floating-Point to Hexadecimal.* [Online] 2008. https://babbage.cs.qc.cuny.edu/ieee-754.old/decimal.html.

55. **SymPy Development Team.** *SymPy Tutorial.* [Online] 2018. https://docs.sympy.org/latest/tutorial/index.html.

56. **Huraj, Ladislav.** *Nebojme sa šifrovania.* [online] Bratislava : Metodicko-pedagogické centrum v Bratislave, 2002. ISBN 80-8052-160-3.

57. **Prispievatelia Wikipédie.** Cézarova šifra. *Wikipédia, Slobodná encyklopédia.* [Online] 26. 07 2019. https://sk.wikipedia.org/w/index.php?title=C%C3%A9zarova_%C5%A1ifra&oldid=6872977.

58. **Přispěvatelé Wikipedie.** UTF-8. *Wikipedie: Otevřená encyklopedie.* [Online] 29. 08 2020. https://cs.wikipedia.org/w/index.php?title=UTF-8&oldid=18956529.

59. —. Vigenèrova šifra. *Wikipedie: Otevřená encyklopedie.* [Online] 26. 05 2018. https://cs.wikipedia.org/w/index.php?title=Vigen%C3%A8rova_%C5%A1ifra&oldid=16114438.

60. *Téměř dokonalá šifra.* **Voborník, Petr.** 1, Olomouc : PROMETHEUS, spol. s r. o., 2014, MATEMATIKA-FYZIKA-INFORMATIKA. Dostupné na: http://mfi.upol.cz/index.php/mfi/issue/archive. 1805-7705.

61. **Ovseník, Ľuboš.** Prenosové média. *Základné pojmy informatiky.* [Online] 2018. [Dátum: 23. 8 2020.] https://data.kemt.fei.tuke.sk/PM_PS_Prenosove_media/Prednasky/Pr01/Doplnkova%20literatura/.

62. **Stanek, Martin.** Kryptológia – úvod. *Úvod do informačnej bezpečnosti.* [Online] 2012. [Dátum: 23. 8 2020.] http://new.dcs.fmph.uniba.sk/files/uib/intro_crypto_2012-u.pdf.

63. **Cápay, Martin.** Úvod do kryptografie. *Kryptografia.* [Online] 2016. [Dátum: 23. 8 2020.] https://edu.ukf.sk/pluginfile.php/103117/mod_resource/content/1/KRY%20PR1.pdf.

64. **Hriňák, Martin.** Staršie kódovacie systémy. *Mladý vedec.* [Online] [Dátum: 23. 8 2020.] http://stary.mladyvedec.sk/archiv/archiv-druheho-cisla/46-starie-kodovacie-systemy.html.

65. **Neznámy.** Transpozičné šifry. *Kryptografia.* [Online] [Dátum: 23. 8 2020.] http://server.gphmi.sk/pages/sifry/trans.html.

66. **Kajínek, Milan.** Tajemství šifer - Po stopách kryptografie a steganografie IV. *The Epoch Times.* [Online] 11. 7 2008. [Dátum: 23. 8 2020.] https://www.epochtimes.cz/200807115516/Tajemstvi-sifer-Po-stopach-kryptografie-a-steganografie-IV.html.

67. **Halva, Martin.** Encryptor. *Baconova Šifra.* [Online] 2015. [Dátum: 23. 8 2020.] http://encryptor.wz.sk/index.php?page=bacon.

68. **Jozef, Tomáš.** Posterus. *Moderné metódy obrazovej steganografie.* [Online] 30. 6 2014. [Dátum: 23. 8 2020.] http://www.posterus.sk/?p=17075.

69. **Kolektív.** Steganografia. *Wikipedia.* [Online] 2020. [Dátum: 23. 8 2020.] https://cs.wikipedia.org/wiki/Steganografie.

70. **Cápay, Martin.** Úvod do kryptografie. *Kryptografia.* [Online] 2016. [Dátum: 23. 8 2020.] https://edu.ukf.sk/pluginfile.php/103117/mod_resource/content/1/KRY%20PR1.pdf.

71. **Egly, Nolan.** Cryptography 101. [Online] 17. 8 2013. [Dátum: 23. 8 2020.] https://www.slideshare.net/NolanEgly/cryptography-101-25330917.

72. **Perera, Hareendra Lakshan.** History of Steganography. *Hareenlaks.* [Online] 4. 2 2011. [Dátum: 23. 8 2020.] http://hareenlaks.blogspot.com/2011/04/history-of-steganography.html.

73. **Madoš, Branislav.** PC Revue. *Tajomstvá steganografie.* [Online] [Dátum: 23. 8 2020.] http://www.gljs.sk/~sjiricek/inf/pcrevue/steganografia.pdf.

74. **Li, Sophia.** Image Steganography in Python. [Online] 5. 3 2017. [Dátum: 23. 8 2020.] http://blog.justsophie.com/image-steganography-in-python/.

75. **Neznámy.** Pixabay. [Online] [Dátum: 23. 8 2020.] https://pixabay.com/en/friends-cat-and-dog-cats-and-dogs-1149841/.

76. **Skalka, Ján, a iní.** *Programátorské techniky.* Nitra : UKF, 2007.

77. **fyzmatik.** Droste efekt. *Fyzmatik.píše.* [Online] 20. 4 2010. [Dátum: 23. 8 2020.] https://fyzmatik.pise.cz/144-droste-efekt.html.

78. **Wikimedia Commons contributors.** Infinity Mirror. [Online] 3. 8 2007. [Dátum: 23. 8 2020.] https://commons.wikimedia.org/wiki/File:Infinity_Mirror.png.

79. **Piroško, Jozef.** Rekurzia. [Online] [Dátum: 23. 8 2020.] http://www.gymparnr.edu.sk/obsah/predmety/subory/informatika/rekurzia.pdf.

80. **Přispěvatelé Wikipedi.** Zásobník (datová struktura). [Online] [Dátum: 23. 8 2020.] https://cs.wikipedia.org/wiki/Z%C3%A1sobn%C3%ADk_(datov%C3%A1_struktura).

81. **Blaho, Andrej.** Rekurzia. [Online] 2016. http://input.sk/python2016/12.html.

82. **Vinař, Tomáš.** Rozdeľuj a panuj. *Efektívne algoritmy a zložitosť.* [Online] [Dátum: 23. 8 2020.] http://compbio.fmph.uniba.sk/vyuka/vkti1/old/2014/handouts/p06.pdf.

83. **Grňa, Ana Bažová.** Prečo fraktály? *Fraktálna kresba.* [Online] [Dátum: 23. 8 2020.] https://www.fraktalnakresba.sk/preco-fraktaly/.

84. **Revision, Jeff.** The Koch Snowflake. *Python-with-science.* [Online] 2019. [Dátum: 23. 8 2020.] https://python-with-science.readthedocs.io/en/latest/koch_fractal/koch_fractal.html.

85. **greekmyths-greekmythology.com.** The Myth Of Theseus And The Minotaur. *Greek Myths & Greek Mythology.* [Online] [Dátum: 23. 8 2020.] https://www.greekmyths-greekmythology.com/myth-of-theseus-and-minotaur/.

86. **Skalka, Ján, a iní.** *Programátorské techniky.* Nitra : UKF, 2007.

87. **GeeksforGeek.** Backtracking Algorithms. *GreeksforGeeks.* [Online] [Dátum: 23. 8 2020.] https://www.geeksforgeeks.org/backtracking-algorithms/.

88. **Miller, Brad, Ranum, David a College, Luther.** Exploring a Maze¶. *Problem Solving with Algorithms and Data Structures using Python.* [Online] [Dátum: 23. 8 2020.] https://runestone.academy/runestone/books/published/pythonds/Recursion/ExploringaMaze.html.

89. **Abiy, Thaddeus, Oli, Bipin a Lazar, Andrei.** Recursive Backtracking. *Brilliant.* [Online] [Dátum: 23. 8 2020.] https://brilliant.org/wiki/recursive-backtracking/.

90. **Wikipedia contributors.** Partition problem. *Wikipedia, The Free Encyclopedia.* [Online] 10. 08 2020. https://en.wikipedia.org/w/index.php?title=Partition_problem&oldid=972193498.

91. **Bosák, Ján.** *Grafy a ich aplikácie.* Bratislava : Alfa, 1980. s. 176.

92. **Kučera, Jozef.** *Kombinatorické algoritmy.* Praha : SNTL, 1983. s. 288.

93. **Plesník, Ján.** *Grafové algoritmy.* Bratislava : VEDA, 1983. s. 344.

94. **Sedgewick, Robert.** *Algoritmy v C.* Praha : SoftPress, 2003. ISBN 80-86497-56-9.

95. **Wróblewski, Piotr.** *Algoritmy: Datové struktury a programovací techniky.* Praha : Computer Press, 2014. ISBN 8025103439.

96. **Töpfer, Pavel.** *Algoritmy a programovací techniky.* Praha : Prometheus, 1995. s. 299. ISBN 80-85849-83-6.

97. *Note on the Frequency of Use of the Different Digits in Natural Numbers.* **Newcomb, Simon.** 1, s.l. : Johns Hopkins University, 1881, American Journal of Mathematics, Zv. 4.

98. *Proceedings of the American Philosophical Society.* **Benford, Frank.** s.l. : American Philosophical Society, 1938. The law of anomalous numbers.

99. **Long, Jasonl; Thornton, Bryce.** An experiment to test Benford's Law against large, publicly available datasets. *Testing Benford's Law.* [Online] https://testingbenfordslaw.com/.

100. **Jamain, Adrien.** Benford's Law. *Classification Research Group.* [Online] 2001. http://wwwf.imperial.ac.uk/~nadams/classificationgroup/Benfords-Law.pdf.

101. **Chráska, Miroslav.** *Metody pedagogického výzkumu.* Havlíčkův Brod : Grada Publishing, a.s., 2007. ISBN 978-80-247-1369-4.

102. **Redakcia ÚPVS, eb.** Open Data – otvorené dáta. *Ústredný portál verejnej správy.* [Online] 9. 1 2018. https://www.slovensko.sk/sk/zivotne-situacie/zivotna-situacia/_open-data-otvorene-data/.

103. **Y. Shafranovich.** Common Format and MIME Type for Comma-Separated Values (CSV) Files. *IETF Tools.* [Online] 10 2005. https://tools.ietf.org/html/rfc4180.

104. **Bray, Tim, a iní.** Extensible Markup Language (XML) 1.0 (Fifth Edition). *World Wide Web Consortium (W3C).* [Online] 26. 11 2008. https://www.w3.org/TR/xml/.

105. **Ecma International.** The JSON Data Interchange Syntax. *Welcome to Ecma International.* [Online] 12 2017. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

106. **W3C.** Scalable Vector Graphics (SVG) 2. *World Wide Web Consortium (W3C).* [Online] 04. 10 2018. https://www.w3.org/TR/SVG/.

107. **Data, Refsnes.** SVG Tutorial. *W3Schools Online Web Tutorials.* [Online] Refsnes Data. https://www.w3schools.com/graphics/svg_intro.asp.

108. **Moitzi, Manfred.** svgwrite 1.3.2 documentation. [Online] 06. 11 2019. [Dátum: 14. 09 2020.] http://readthedocs.org/docs/svgwrite/.

109. **Kapusta, Jozef a Munk, Michal.** *Web usage mining: Príprava a modelovanie dát.* Prírodovedec č. 592. Nitra : Univerzita Konštantína Filozofa v Nitre, 2014. s. 136. ISBN 978-80-558-0692-1.

110. **Sedgewick, Robert a Wayne, Kevin.** *Algorithms (4th Edition).* s.l. : Addison-Wesley, 2011. s. 976. ISBN: 978-0321573513.

111. **Wikipedia contributors.** Blend modes. *Wikipedia, The Free Encyclopedia.* [Online] [Dátum: 23. 8 2020.] https://en.wikipedia.org/wiki/Blend_modes.

112. —. Morphing. *Wikipedia.* [Online] Wikipedia, The Free Encyclopedia. [Dátum: 23. 8 2020.] https://en.wikipedia.org/wiki/Morphing.

113. **Haytam, Zanid.** Creating Gifs in Python Using Pillow. *Zanid Haytam's personal blog.* [Online] 21. 8 2018. [Dátum: 23. 8 2020.] https://blog.zhaytam.com/2018/08/21/creating-gifs-using-python-pillow/.

114. **Heinzman, Andrew.** What Is a GIF, and How Do You Use Them? *How-to Geek.* [Online] 25. 9 2019. [Dátum: 23. 8 2020.] https://www.howtogeek.com/441185/what-is-a-gif-and-how-do-you-use-them/.

115. **101 Computing.** Python Turtle – Morphing Algorithm. *101 Computing - Boost your programming skills!* [Online] 9. 2 2018. [Dátum: 23. 8 2020.] https://www.101computing.net/python-turtle-morphing-algorithm/.

116. **Přispěvatelé Wikipedie.** Lineární interpolace. *Wikipedie: Otevřená encyklopedie.* [Online] [Dátum: 23. 8 2020.] https://cs.wikipedia.org/wiki/Line%C3%A1rn%C3%AD_interpolace.

117. **Foundation, Python Software.** The Python Tutorial. *Welcome to Python.org.* [Online] 14. 09 2020. https://docs.python.org/3.6/tutorial/index.html.

118. **Shipman, John W.** Tkinter 8.5 reference: a GUI for Python. *NSF REU: Interdisciplinary Research Experience in Computational Sciences.* [Online] 23. 06 2013. [Dátum: 14. 09 2020.] Pôvodné umiestnenie dokumentu: http://infohost.nmt.edu/tcc/help/pubs/tkinter/tkinter.pdf. http://reu.cct.lsu.edu/documents/Python_Course/tkinter.pdf.

119. **Wirth, Niklaus.** *Algoritmy a štruktúry údajov.* Bratislava : Alfa, 1989. s. 488. ISBN 80-05-00153-3.

120. **UPJŠ v Košiciach.** PALMA junior, Programovanie, ALgoritmy, MAtematika, súťaž pre mladých programátorov v Pythone. *PALMA junior.* [Online] 2020. https://di.ics.upjs.sk/palmaj/.

121. **Štátny pedagogický ústav.** Informatika. *Inovovaný ŠVP pre gymnáziá so štvorročným a päťročným vzdelávacím programom.* [Online] 26. 3 2015. https://www.statpedu.sk/files/articles/dokumenty/inovovany-statny-vzdelavaci-program/informatika_g_4_5_r.pdf.

Register pojmov

Zoznam príloh

Príloha 1: priečinky 01 až 25 obsahujúce pracovné súbory.