

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.

Poznámka pre učiteľa

V tejto kapitole si zopakujeme obsah základného kurzu oblasti Algoritmické riešenie problémov. Táto kapitola nie je náhradou za základný kurz a odporúčame, aby ho už žiaci mali absolvovaný. Pre absolvovanie tohto predmetu predpokladáme, že žiaci vedia programovať na úrovni definovanej v iŠVP.

Ak žiaci absolvovali základný kurz v jazyku Python, môžete túto kapitolu preskočiť alebo ju využiť na rýchle zopakovanie. Pre ďalšie kapitoly je potrebné, aby žiaci vedeli samostatne riešiť úlohy typovo podobné úlohám z tejto časti.

Ak žiaci absolvovali základný kurz v inom jazyku ako Python, využite túto kapitolu na to, aby ste žiakom ukázali aké príkazy, konštrukcie a prístupy pri riešení problémov používa jazyk Python. Aj v tomto prípade je potrebné, aby žiaci vedeli v jazyku Python samostatne riešiť úlohy typovo podobné úlohám z tejto časti.

V tejto kapitole uvádzame výkladový text a zadania úloh určené žiakovi. Pri úlohách uvádzame riešenie, príp. návod na riešenie. Odporúčame učiteľovi, aby riešenie sprostredkoval žiakovi pre kontrolu jeho riešenia, príp. ako návod na riešenie, ak žiak nie je schopný riešenie dosiahnuť samostatne. Podotýkame, že uvedené riešenia nie sú jediné správne. V prípade iného riešenia úlohy môžeme so žiakmi diskutovať o riešeníach, porovnávať ich z pohľadu efektívnosti alebo všeobecnosti.

Jazyk Python

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

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

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.

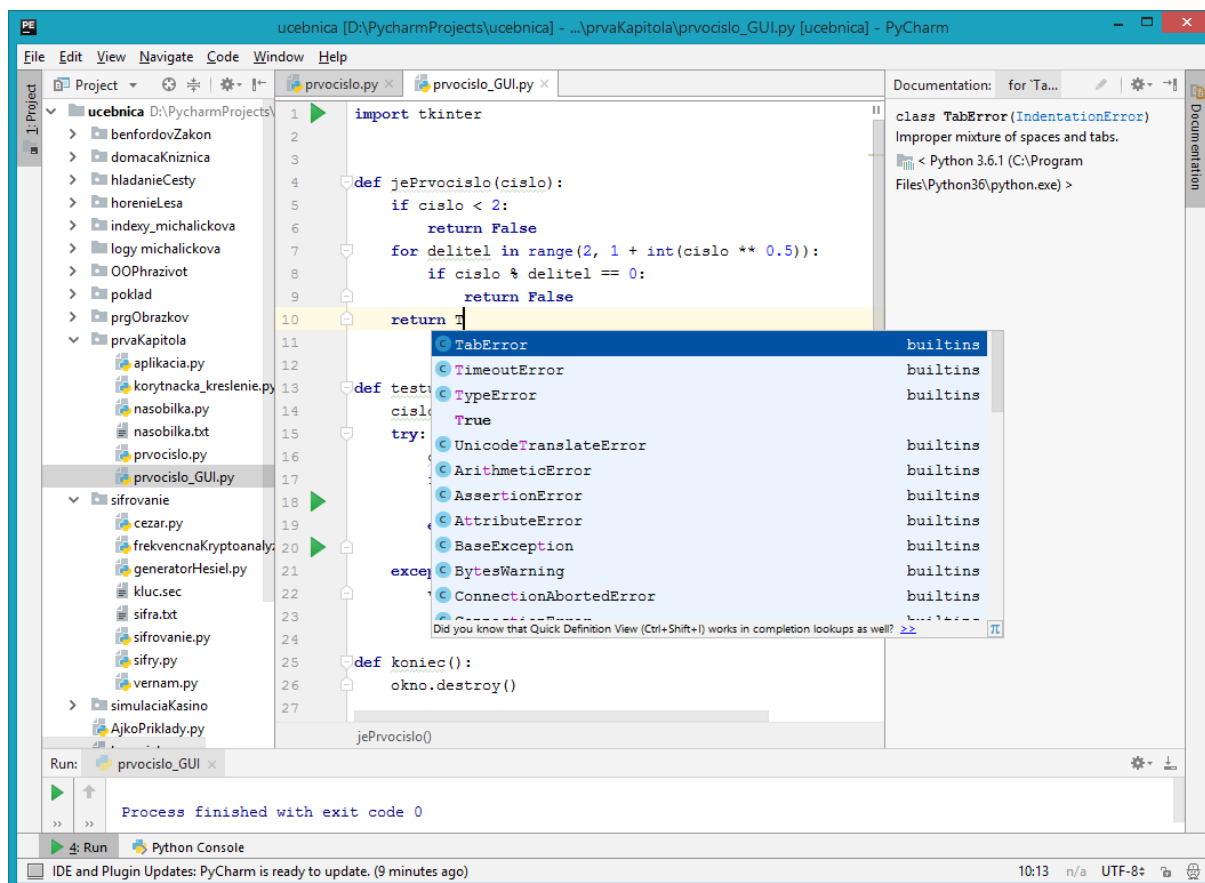
Interpreter jazyka Python nevytvára žiaden spustiteľný kód (exe alebo com súbor). Pre vykonávanie Python-ovských programov musí byť v počítači inštalovaný interpreter jazyka Python. Preto prívlastok interpretovaný. Jedným z dôsledkov je možnosť interaktívneho režimu, pri ktorom sú výrazy automaticky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

Python je distribuovaný pod otvorenou licenciou kompatibilnou s GPL (GNU General Public License). Samotný jazyk a jednoduché vývojové prostredie si môžeme stiahnuť zo stránky <https://www.python.org/downloads/>. 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, IBM 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

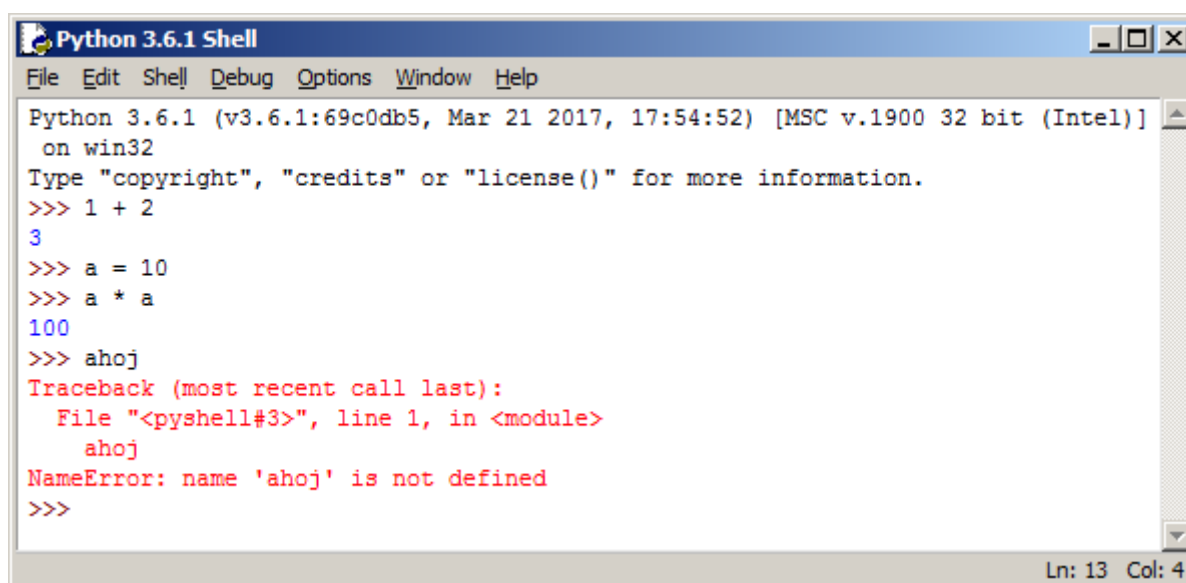
Pri písaní programov môžeme využiť jednoduché prostredie IDLE (Integrated DeveLopment Environment), ktoré je súčasťou základnej inštalácie jazyka. Pre vytváranie rozsiahlejších programov odporúčame využiť pokročilejšie vývojové prostredie (IDE - Integrated Development Environment) (napr. PyCharm Edu, <https://www.jetbrains.com/pycharm-edu/> Obrázok 1).



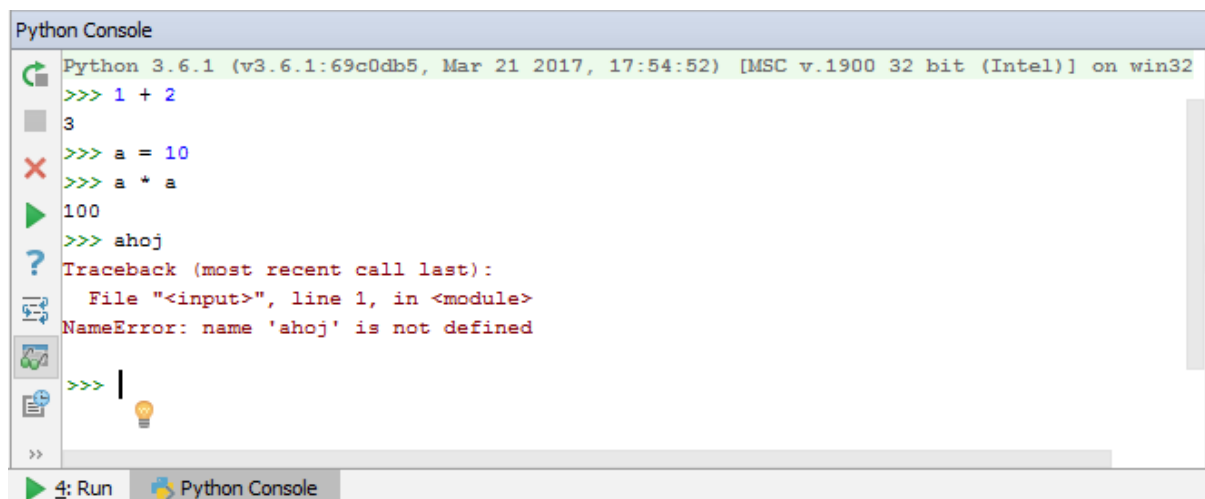
Obrázok 1 Vývojové prostredie PyCharm Edu

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 časti 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.

```
>>> cenaListka = 3.5
>>> pocetListkov = 17
>>> platba = cenaListka * pocetListkov
>>> 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
```

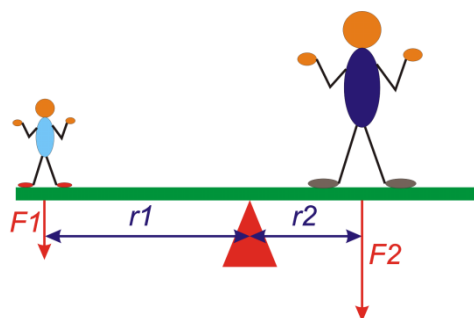
Poznámka pre učiteľa

V nasledujúcich úlohách nie je cieľom len nájsť postup ako vypočítať správny výsledok, ale aj prehľadný zápis riešenia. Odporúčame, aby si žiaci vhodne pomenovali jednotlivé hodnoty a ďalej pracovali s menami týchto hodnôt. Riešenie tak bude prehľadnejšie a ľahšie pochopiteľné.

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?

Riešenie:

matematické odvodenie:

$$r_1 + r_2 = 2 \Rightarrow r_1 = 2 - r_2$$

$$m_1 \cdot r_1 = m_2 \cdot r_2$$

...

$$r_2 = 2 * m_1 / (m_1 + m_2)$$

riešenie v konzole:

```
>>> m1 = 800
>>> m2 = 14000
>>> r2 = 2 * m1 / (m1 + m2)
>>> r2
>>> 0.10810810810810811
```

Odpoveď: Železnú tyč by sme mali podložiť najviac 11 cm od auta.

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.

Navrhnete 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?

Riešenie:

Ako jednotku môžeme uvažovať 1 útržok. Keďže cena útržku by bola príliš nízka, uvažujme cenu za 100 útržkov.

Ostáva nám zistiť, ako prepočítať metre na útržky. Jednoduchým premeraním zistíme, že 10 útržkov má dĺžku asi 1,17 metra.

```
>>> cena = 2.9
>>> pocetKotucov = 6
>>> dlzkaKotuca = 68
>>> pocetUtrzkov = dlzkaKotuca / 0.117
>>> cena100Utrzkov = 100 * cena / (pocetKotucov * pocetUtrzkov)
>>> cena100Utrzkov
0.08316176470588235
```

alebo

```
>>> cena = 2.33
>>> pocetKotucov = 10
>>> pocetUtrzkov = 200
>>> cena100Utrzkov = 100 * cena / (pocetKotucov * pocetUtrzkov)
>>> cena100Utrzkov
0.1165
```

Reťazec (string), dátová štruktúra zoznam (list)

Poznámka pre učiteľa

Nepredpokladáme, že žiaci poznajú všetky metódy reťazcov a zoznamov. Očakávame však, že žiaci vedia čítať dokumentáciu a na jej základe si vybrať a vhodne kombinovať potrebné metódy pre riešenie problému.

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 = '' #prazdny retazec
>>> retazec1 = 'Ahoj' #retazec 'Ahoj'
>>> retazec2 = 'kamarat' #retazec 'kamarat'
>>> pozdrav = retazec1 + ' ' + retazec2 #zretazenie 'Ahoj kamarat'
>>> pozdrav
'Ahoj kamarat'
>>> pozdrav[5]
'k'
```

Viac o reťazcoch: <https://docs.python.org/3.6/library/stdtypes.html#text-sequence-type-str>

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 = [] #prazdny zoznam
>>> dobreZnamky = [1, 2] #zoznam [1, 2]
>>> dobreZnamky.append(3) #vlozenie na koniec [1, 2, 3]
>>> zleZnamky = [4, 5] #
>>> znamky = dobreZnamky + zleZnamky #spojenie [1, 2, 3, 4, 5]
>>> znamky[2]
3
```

Viac o zoznamoch: <https://docs.python.org/3.6/tutorial/datastructures.html#more-on-lists>

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

Reťazec/zoznam

0	1	2	3	4	5
P	y	t	h	o	n

indexovanie sprava

-6	-5	-4	-3	-2	-1
----	----	----	----	----	----

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 pretypovanie 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: ')
celeCislo = int(vstup)
realneCislo = 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}')
```

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.

Riešenie:

Aj pri riešení tejto úlohy dbajme na vhodné názvy premenných a zrozumiteľnosť výsledku.

```
dlzkaTyce = input('Zadaj dlzku tyce v metroch: ')
dlzkaTyce = float(dlzkaTyce)
hmotnostClovek = input('Zadaj hmotnost cloveka v kilogramoch: ')
hmotnostClovek = float(hmotnostClovek)
hmotnostAuto = input('Zadaj hmotnost auta v kilogramoch: ')
hmotnostAuto = float(hmotnostAuto)
vzdialenostOdAuta = dlzkaTyce * hmotnostClovek / (hmotnostClovek +
hmotnostAuto)
print('Tyc je potrebne podlozit ' + str(vzdialenostOdAuta) + ' m od auta.')
#alebo
print(f'Tyc je potrebne podlozit {vzdialenostOdAuta} m od auta.')
```

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.

Riešenie:

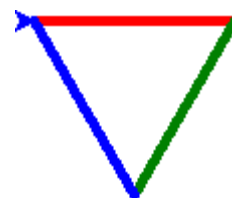
Aj pri riešení tejto úlohy dbajme na vhodné názvy premenných a zrozumiteľnosť výsledku.

```
cena = input('Zadaj cenu balenia EUR: ')
cena = float(cena)
pocetKotucov = input('Zadaj pocet kotucov v baleni: ')
pocetKotucov = int(pocetKotucov)
pocetUtrzkov = input('Zadaj pocet utrzkov v rolke: ')
pocetUtrzkov = int(pocetUtrzkov)

cena100Utrzkov = 100 * cena / (pocetKotucov * pocetUtrzkov)
print('Cena za 100 kusov utrzkov: ' + str(cena100Utrzkov) + ' EUR.')
#alebo
print(f'Cena za 100 kusov utrzkov: {cena100Utrzkov} EUR.')
```

Korytnačia grafika

Jeden z možných výstupov našich programov môže byť obrázok. Na kreslenie môžeme využiť modul turtle.



```
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: `from 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/3.6/library/turtle.html>.

Úloha 5

Využitím modulu turtle vykreslite nasledovný obrázok:



Riešenie:

Stačí si uvedomiť, že niektorú z čiar (miesto, kde sa pretína s inou čiarou) budeme musieť prekresliť, aby sme dosiahli efekt, že sa posunula vyššie.

```
import turtle
platno = turtle.Screen()
pero = turtle.Turtle()

pero.pensize(20)
```

```
pero.color('red')
pero.forward(100)
pero.forward(-20)
pero.right(120)

pero.color('green')
pero.forward(-20)
pero.forward(100)
pero.forward(-20)
pero.right(120)

pero.color('blue')
pero.forward(-20)
pero.forward(100)
pero.forward(-20)
pero.right(120)

pero.color('red')
pero.forward(-20)
pero.forward(40)

platno.mainloop()
```

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 4 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 i in range(100):
    print(f'{i}. 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 100 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 `i` 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 i in range(1, 100):  
    print(f'{i}. 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('Nasobky 7:')  
for i in range(7, 71, 7):  
    print(i)
```

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

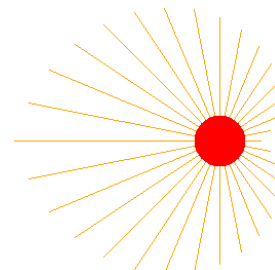
Namiesto funkcie `range()` môžeme použiť aj nejakú iterovateľnú štruktúru (štruktúra, cez prvky ktorej možno prechádzať).

```
>>> 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.



Riešenie:

Všimnime si, že lúče sa postupne predlžujú až po najdlhší a potom sa postupne skracujú. Stačí teda v dvoch cykloch kresliť lúče, ktorých dĺžka je odvodená od premennej cyklu. Premennú jedného cyklu bude postupne narastať, premenná druhého cyklu bude postupne klesať.

```
import turtle
tabula = turtle.Screen()
pero = turtle.Turtle()
pero.color('orange')

for i in range(40, 200, 10):
    pero.forward(i)
    pero.forward(-i)
    pero.left(360 / 32)

for i in range(200, 40, -10):
    pero.forward(i)
    pero.forward(-i)
    pero.left(360 / 32)

pero.color('red')
pero.dot(50)
tabula.mainloop()
```

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 = input('Zadaj heslo: ')
while heslo != 'abracadabra':
    heslo = input('Zadaj heslo: ')

print('Heslo je spravne')
```

Úloha 7

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

$$\text{NSD}(a, a) = a$$
$$\text{NSD}(a, b) = \text{NSD}(b, a)$$
$$\text{NSD}(a, b) = \text{NSD}(a, b - a) \Leftrightarrow b > 0$$

Na základe tohto poznatku vytvoril nasledovný program:

```
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 zvykom po delení. Naprogramujte jeho vylepšenú verziu.

Riešenie

Rozdiel okrem zrýchlenia je aj v tom, že ak nastane situácia, že menšie číslo je deliteľom väčšieho, väčšie číslo sa nahradí 0. Preto je podmienka cyklu iná, žiadne z čísiel nie je 0 (stačí testovať súčin na nulu). Keďže končíme, ak jedno z čísiel je nulové a druhé je rovné NSD, stačí ich súčet vyhlásiť za NSD.

```
while a * b != 0:
    if a > b:
        a = a % b
    else:
        b = b % a
nsd = a + b
```

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

Funkcia môže, okrem samotného vykonania kódu, vrátiť nejakú výslednú hodnotu. S touto hodnotou môžeme ďalej, mimo funkcie, pracovať.

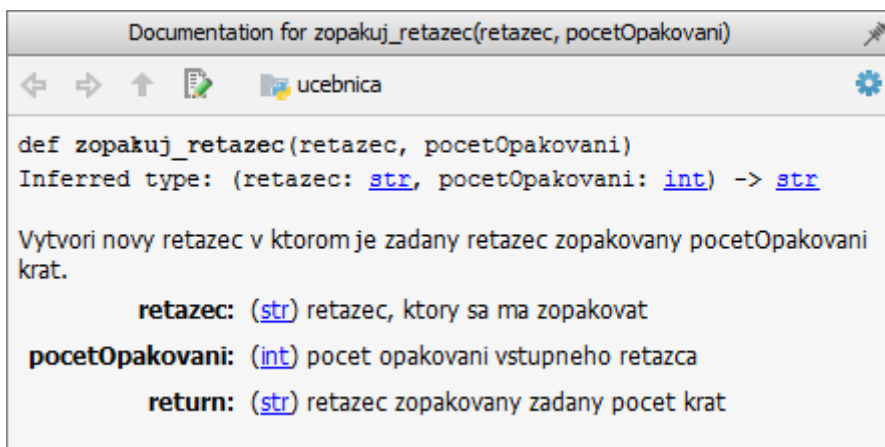
```
def zopakuj_retazec(retazec, pocetOpakovani):  
    return retazec * pocetOpakovani  
  
print(zopakuj_retazec('ahoj', 5)) #ahojahojahojahojahoj
```

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.

```
def zopakuj_retazec(retazec, pocetOpakovani):  
    ''' Vytvori novy retazec v ktorom je zadany retazec  
        zopakovany pocetOpakovani krat.  
  
        :param retazec: retazec, ktory sa ma zopakovat  
        :type retazec: str  
        :param pocetOpakovani: pocet opakovani vstupneho retazca  
        :type pocetOpakovani: int  
        :return: retazec zopakovany zadany pocet krat  
        :rtype: str  
        '''  
    return retazec * pocetOpakovani
```

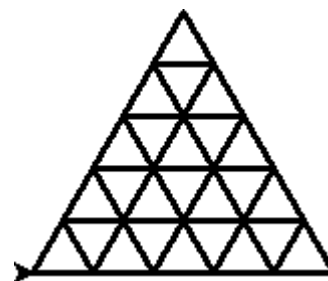
Moderné vývojové prostredia uľahčujú programátorom prácu a na požiadanie zobrazia dokumentáciu k vybranej funkcii. Aj bez toho, aby sme skúmali kód funkcie, vieme presne čo funkcia robí, aké požaduje vstupy a aké sú jej výstupy.



Obrázok 4 Zobrazenie dokumentácie funkcie

Úloha 8

Vytvorte program, ktorý vykreslí nasledovnú pyramídu.



Riešenie:

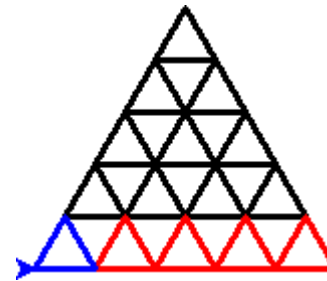
Toto je typická úloha zameraná na dekompozíciu. Ak si pyramídu prezrieme pozornejšie, zistíme, že pyramída sa skladá z **malých trojuholníkov**, ktoré vytvárajú **rad**y. Jednotlivé rady naskladané nad sebou tvoria pyramídu. V zmysle takejto dekompozície definujeme aj jednotlivé funkcie.

Pri kresliacich úlohách je výhodné, ak po kreslení ostane pero v rovnakej pozícii ako bolo pred kreslením. Toto pravidlo sme aplikovali na všetky funkcie.

```
import turtle
pero = turtle.Turtle()
pero.width(3)
tabula = turtle.Screen()

def trojuholnik(dlзка):
    for i in range(3):
        pero.forward(dlзка)
        pero.left(120)

def rad(dlзка, pocet):
    for i in range(pocet):
        trojuholnik(dlзка)
        pero.forward(dlзка)
    pero.forward(-dlзка * pocet)
```



```
def pyramida(dlзка, pocet):
    for i in range(pocet, 0, -1):
        rad(dlзка, i)
        pero.left(60)
        pero.forward(dlзка)
        pero.right(60)
    pero.left(60)
    pero.forward(-dlзка * pocet)
    pero.right(60)

pyramida(30, 5)
tabula.mainloop()
```

Úloha 9

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

$$V_k(n) = \frac{n!}{(n-k)!}$$

$$V'_k(n) = n^k$$

$$P(n) = n!$$

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

$$C_k(n) = \frac{n!}{k!(n-k)!}$$

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

Riešenie

Úloha je zameraná na dekompozíciu. Všimnime si, že faktoriál je pomerne častou operáciou. Výhodné je preto definovať si funkciu na výpočet faktoriálu. Výsledkom je vždy celé číslo. Pri delení je teda výhodné použiť celočíselné delenie.

Poznámka: Funkcia pre výpočet faktoriálu je umiestnená aj v module *math*. Riešenie si jej využitím môžeme zjednodušiť.

```
def faktorial(n):
    vysledok = 1
    for i in range(1, n + 1):
        vysledok *= i
    return vysledok

def variace(k, n):
    return faktorial(n) // faktorial(n - k)

def variacie_s_opakovanim(k, n):
    return n ** k

def permutacie(n):
    return faktorial(n)

def permutacie_s_opakovanim(pocetnosti, n):
    menovatel = 1
    for pocetnost in pocetnosti:
        menovatel *= faktorial(pocetnost)
    return faktorial(n) // menovatel
```

```
def kombinacie(k, n):  
    return faktorial(n) // (faktorial(k) * faktorial(n - k))  
  
def kombinacie_s_opakovanim(k, n):  
    return faktorial(n + k - 1) // (faktorial(n - 1) * faktorial(k))  
  
print(permutacieSOpakovanim([20, 30, 30], 80))
```

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.

funkcie.py:

```
'''Modul pre zdravenie  
'''  
def pozdrav():  
    ''' Pozdravi pouzivателя vypisom 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 `statistika` a vložte doň funkcie pre riešenie kombinatorických úloh z predchádzajúcej úlohy. Doplníte k funkciám a k modulu dokumentačné reťazce. V hlavnom programe otestujte ich použitie.

Podmienený príkaz IF

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 v kg, výška v m vráti, do ktorej kategórie človek patrí.

Využiť môžete nasledujúcu klasifikáciu platnú pre európsku populáciu:

- $bmi < 18,5$ podvýživa,
- $18,5 \leq bmi < 25$ ideálna a zdravá váha,
- $25 \leq bmi < 30$ mierna nadváha,
- $30 \leq bmi < 40$ obezita,
- $40 \leq bmi$ ťažká obezita.

Riešenie:

Pre efektívne riešenie tejto úlohy je potrebné si uvedomiť, že niektoré z podmienok (podľa zápisu vyššie) nie je potrebné opakovane testovať. Napr. pre kategóriu „obezita“ nemusíme testovať, či $bmi \geq 30$. Ak by nebol, odchytila by túto možnosť niektorá z predchádzajúcich vetiev príkazu `if` a `bmi` by napatril do tejto kategórie.

```
def bmi(hmotnost, vyska):  
    hodnotaBMI = hmotnost / vyska ** 2  
    if hodnotaBMI < 18.5:  
        return 'podvýživa'  
    elif hodnotaBMI < 25:  
        return 'ideálna a zdravá váha'  
    elif hodnotaBMI < 30:  
        return 'mierna nadváha'  
    elif hodnotaBMI < 40:  
        return 'obezita'  
    else:  
        return 'ťažká obezita'
```

Úloha 12

Pig Latin 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 slová a písmená malej anglickej abecedy.

Riešenie:

V riešení musíme nájsť spôsob, ako efektívne otestovať, či slovo začína samohláskou alebo spoluhláskou, a následne vytvoriť zo zadaného slova jeho preklad do Pig Latin.

```
def slovo_do_pig_latin(slovo):
    samohlasky = 'aeiouy'
    if slovo[0] in samohlasky:
        return slovo + 'way'
    else:
        return slovo[1:] + slovo[0] + 'ay'
```

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.

Riešenie:

Riešenie trochu komplikuje fakt, že musíme odlíšiť, či je zadávané množstvo v metroch alebo v útržkoch. Keď si pozrieme parametre niekoľkých balení (napr. na heureka.sk) zistíme, že ak výrobca udáva množstvo v metroch, tak viac ako 120 m v kotúči nenájde (bežne +- 20 m, ekonomické balenie 68 m, maxi kotúč asi 117 m). Môžeme preto spraviť rozhodnutie, že ak zadané množstvo je < 120, budeme ho interpretovať ako počet metrov. Inak ho budeme interpretovať ako počet útržkov.

```
def jednotkova_cena_toaletného_papiera(cena, pocetKotucov,
mnozstvoVKotuci):
    ''' Vráti cenu toaletneho papiera prepocitanu na 1000 utrzkov.

    Ak mnozstvoVKotuci < 120, tak mnozstvoVKotuci ~ pocet metrov v kotuci

    Ak mnozstvoVKotuci >= 120, tak mnozstvoVKotuci ~ pocet utrzkov v kotuci

    :param cena: cena za balenie v EUR
    :type cena: float
    :param pocetKotucov: pocet kotucov v baleni
    :type pocetKotucov: int
    :param mnozstvoVKotuci: pocet utrzkov alebo metrov v kotuci
    :type mnozstvoVKotuci: float
    :return: cena za 1000 utrzkov v EUR zaokruhlene na centy
    :rtype: float
    '''
    if mnozstvoVKotuci < 120:
```

```
pocetUtrzkov = mnozstvoVKotuci / 0.117
else:
    pocetUtrzkov = mnozstvoVKotuci

return round(1000 * cena / (pocetKotucov * pocetUtrzkov), 2)
```

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,
- **behové chyby** – objavia 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.

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

#tu bude tajny vypocet
```

Riešenie:

```
heslo = input('Zadaj heslo: ')
if heslo != 'sezam otvor sa':
    print('Heslo je nesprávne. Program skončil.')
    exit()

#tu bude tajny vypocet
```

Logické chyby

Odhaliť logickú chybu môžeme niekoľkými spôsobmi:

- Pripravme si testovacie dáta a skontrolujme, či pre ne dostaneme správne výsledky.

- Analyzujeme program a skontrolujeme, či každá časť robí to, čo má robiť a tak, ako to má robiť.
- Spustíme program po častiach a skontrolujeme každú časť, uistíme sa, či premenné nadobúdajú požadované hodnoty, či cykly bežia požadovaný počet krát, či funkcie vracajú požadované hodnoty a pod. Jednou z možností, ako kontrolu realizovať, sú kontrolné výpisy. Druhá možnosť je spustiť ladiaci (debugg) režim a kontrolovať krok za krokom, či sa program správa tak, ako to očakávame.

Úloha 15

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

```
def je_prvocislo(cislo):
    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.

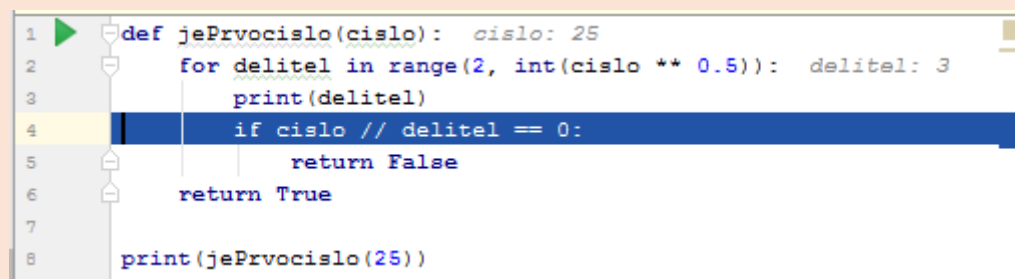
Riešenie

Testovacie dáta by mali obsahovať štandardné hodnoty (napr. 13, 18) ale aj menej časté, okrajové prípady (napr. 0, 1, 2), alebo špeciálne prípady vzhľadom na riešenie (napr. druhé mocniny prvočísel).

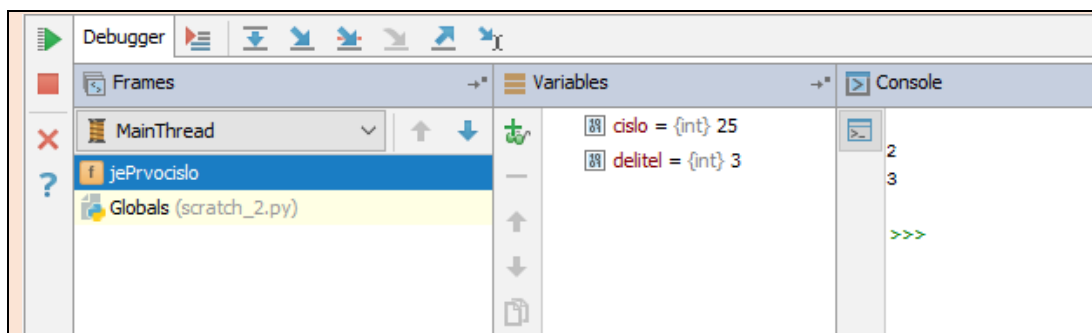
Kontrolné výpisy (napr. hodnota riadiacej premennej cyklu) nám môže ukázať, či počet iterácií cyklu alebo hodnoty riadiacej premennej sú správne (očakávané).

```
def jePrvocislo(cislo):
    for delitel in range(2, int(cislo ** 0.5)):
        print(delitel)
        if cislo // delitel == 0:
            return False
    return True
```

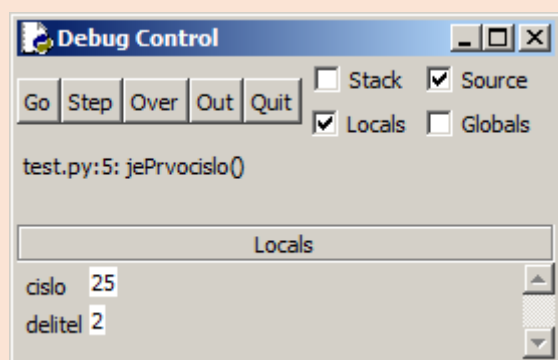
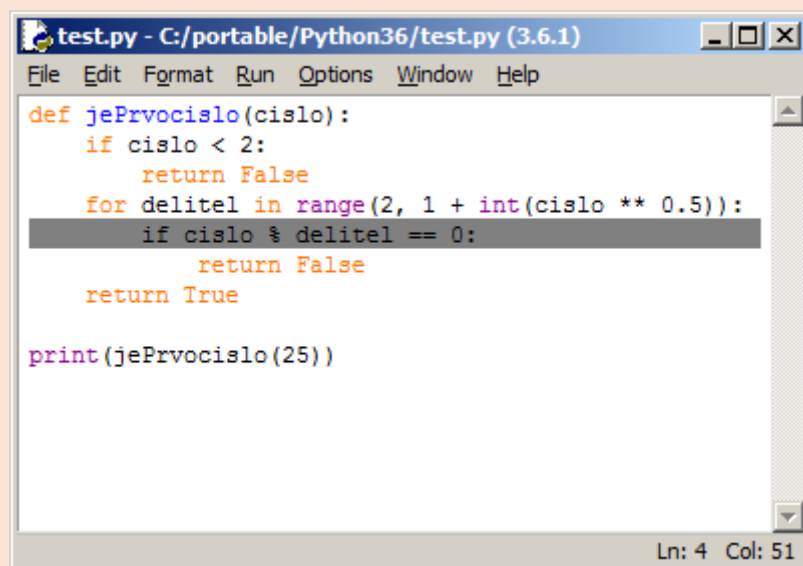
Ladiaci režim v prostredí PyCharm Edu ()



```
1 def jePrvocislo(cislo):
2     for delitel in range(2, int(cislo ** 0.5)):
3         print(delitel)
4         if cislo // delitel == 0:
5             return False
6     return True
7
8 print(jePrvocislo(25))
```



Ladiaci režim v prostredí IDLE



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.

```
citatel = float(input('Zadaj citatela: '))
menovatel = float(input('Zadaj menovателя: '))
print(citatel / menovatel)
```

```
Zadaj citatela: 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 citatela: '))
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 citatela: 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.

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

```
try:
    citatel = float(input('Zadaj citatela: '))
    menovatel = float(input('Zadaj menovateľa: '))
    print(citatel / menovatel)
except ValueError:
    print('Nezadal si číslo.')
except ZeroDivisionError:
    print('Nulou sa deliť neda.')
except:
    print('Neocakavana chyba')
```

```
Zadaj citatela: 1
Zadaj menovateľa: 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 odchyťáva. 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 odchyťávať v poradí od najkonkrétnejších po najvšeobecnejšie. Hierarchiu výnimiek nájdeme na <https://docs.python.org/3.6/library/exceptions.html#exception-hierarchy>. 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:

```
while True:
    try:
        citatel = float(input('Zadaj citatela: '))
        menovatel = float(input('Zadaj menovателя: '))
        print(citatel / menovatel)
        break
    except ValueError:
        print('Nezadal si cisla.')
    except ZeroDivisionError:
        print('Nulou sa delit neda.')
    except:
        print('Neocakavana chyba')

print('Podiel sme uspesne vypocitali.')
```

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. Ošetrte situácie, v ktorých dôjde počas výpočtu k chybe.

Riešenie:

Predpokladajme, že zoznam obsahuje len korektné známky. Problém nastane, ak je zoznam známok prázdny. Toto je korektná situácia, lebo žiak nemusí mať žiadnu známku.

```
znamky = [1, 2]  
#alebo  
znamky = []  
try:  
    priemer = sum(znamky) / len(znamky)  
    print(f'Priemer známok: {priemer}')
```

```
except ZeroDivisionError:  
    print('Priemer nie je možné vypočítať. Nie sú zadane žiadne známky.')
```

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. Filozofiu 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 vyhodíť výnimku.

```
import math
def obsah_kruhu(polomer):
    if polomer < 0:
        raise ValueError('Zadany zaporny 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: Zadany zaporny polomer kruhu.

Process finished with exit code 1
```

Pri vyhadzovaní výnimiek zvolíme taký typ, ktorý zodpovedá chybe, kvôli ktorej výnimku vyhadzujeme. Zoznam výnimiek aj so stručným popisom nájdeme na <https://docs.python.org/3.6/library/exceptions.html#concrete-exceptions>.

Pri rozsiahlejších programoch je bežnou praxou to, že jedna časť nášho programu výnimky vyhadzuje (lebo nie je schopná požadovanú akciu vykonať) a druhá ich odchyťava 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.

```
def vyslednaZnamka (znamky) :
    try:
        return round (sum (znamky) / len (znamky))
    except ZeroDivisionError:
        raise ValueError ('Nie su zadane ziadne znamky.')
```

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

Riešenie:

Funkcia `vyslednaZnamka()` pre zadaný zoznam známok vráti výslednú známku, resp. vyhodí výnimku, ak je zoznam známok prázdny. To môžeme využiť vo funkcii `vysvedcenie()` a v prípade výnimky žiakovi pridáme hodnotenie „neklasifikovany“.

```
def vysvedcenie(ziaci):
    vypis = ''
```

```
for ziac in ziaci:
    try:
        vypis += f'{ziak[0]}: {vyslednaZnamka(ziak[1:])}\n'
    except ValueError:
        vypis += f'{ziak[0]}: neklasifikovany\n'
return vypis

ziaci = [['Karol'], ['Imrich', 2, 3], ['Katka', 1, 1]]

print(vysvedcenie(ziaci))

Karol: neklasifikovany
Imrich: 2
Katka: 1
```

*Poznámka: Všimnime si Imrichovu výslednú známku. Je v poriadku? Spravili sme niekde chybu? Viac sa o tejto „chybe“ dozvieme v kapitole **Aritmetika počítača (spresniť názov)***

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, k hodnotám sme stratili prístup. 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:

```
with open('subor.dat', 'w', encoding='utf-8') as f:           [1]
    f.write('Data v subore')                                  [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 „Data v subore“. 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 vymýšľaní príkladov 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)) #nahodne cele cislo z intervalu 1..10
```

Riešenie:

```
import random
with open('nasobilka.txt', 'w', encoding='utf-8') as f:
    for i in range(10):
        cinitel1 = random.randint(1, 10)
        cinitel2 = random.randint(1, 10)
        f.write(f'{cinitel1} · {cinitel2} = \n')
```

Obsah súboru nasobilka.txt:

```
4 · 7 =
3 · 7 =
10 · 3 =
...
```

Čítanie dát zo súboru

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

```
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.

Riešenie:

Najskôr musíme prečítať jednotlivé riadky súboru a potom z nich extrahovať čísla – činitele súčinu. Z reťazca riadka odstránime nepotrebné znaky a potom reťazec rozdelíme pomocou deliaceho znaku „.“.

```
with open('nasobilka.txt', 'r', encoding='utf-8') as f:
    for riadok in f.readlines():
        riadok = riadok.strip()
        riadok = riadok.replace(' ', '')
        riadok = riadok.replace('=', '')
        cinitele = riadok.split('.')
        cinitel1 = int(cinitele[0])
        cinitel2 = int(cinitele[1])
        print(f'{cinitel1} · {cinitel2} = {cinitel1 * cinitel2}')
```

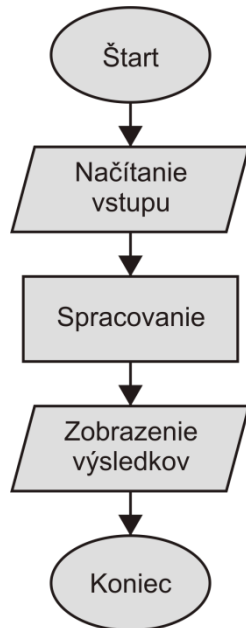
Grafické rozhranie

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 stal modul tkinter.

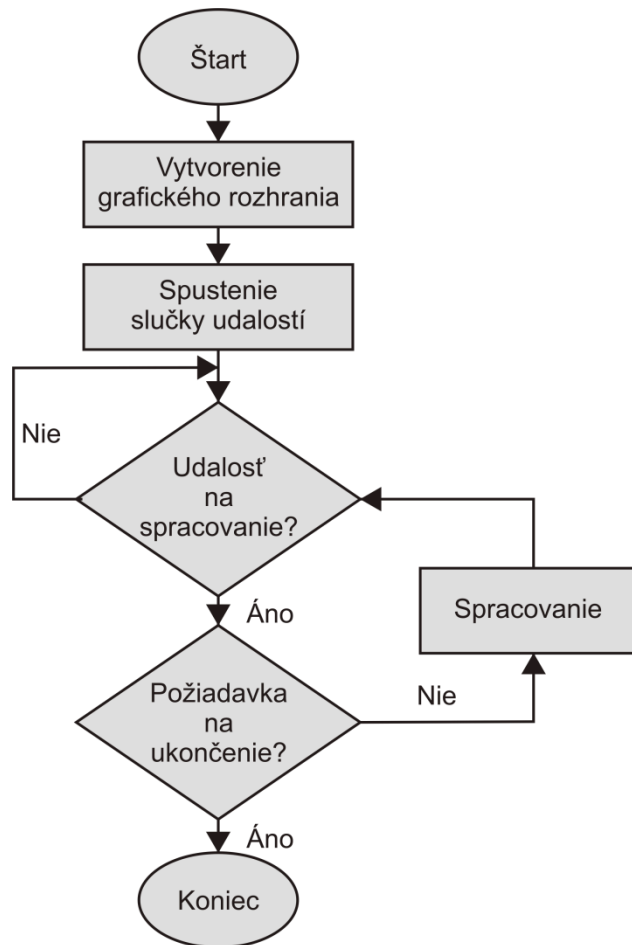
Pomocou modulu tkinter vieme vytvoriť grafické okno, do okna umiestniť prvky potrebné pre prácu s programom (vstupné textové polia, tlačidlá, grafickú plochu atď.) a prepojiť tieto grafické prvky s jednotlivými časťami nášho programu.

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 (Obrázok 5). Grafické programy pracujú odlišne (Obrázok 6). 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

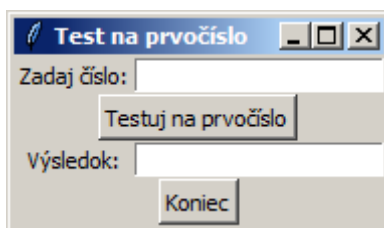
Ukážeme si to na príklade programu, ktorý testuje, či zadané číslo je prvočíslo. Samotný program vyzerá nasledovne:

```

def jePrvocislo(cislo):
    if cislo < 2:
        return False
    for delitel in range(2, 1 + int(cislo ** 0.5)):
        if cislo % delitel == 0:
            return False
    return True

cislo = input('Zadaj číslo: ')
try:
    cislo = int(cislo)
    if jePrvocislo(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:



```

import tkinter [1]

def jePrvocislo(cislo): [2]
    if cislo < 2:
        return False
    for delitel in range(2, 1 + int(cislo ** 0.5)):
        if cislo % delitel == 0:
            return False
    return True

def testuj(): [3]
    cislo = vstup.get() [4]
    try:
        cislo = int(cislo)
        if jePrvocislo(cislo):
            vystup.set(f'Číslo {cislo} je prvočíslo.') [5]
        else:
            vystup.set(f'Číslo {cislo} je 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 `jePrvocislo()` 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, 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 – zaškrávacie 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.

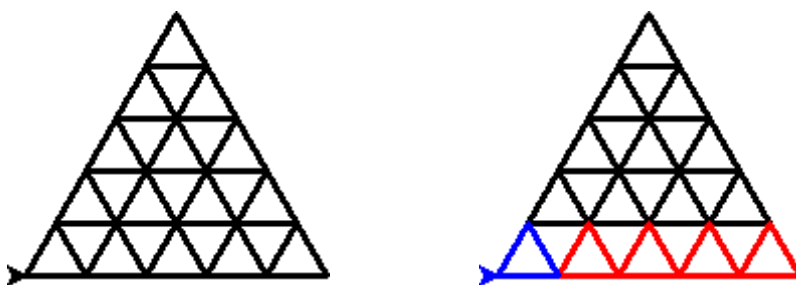
Stratégie riešenia problémov

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, ich zložením 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.



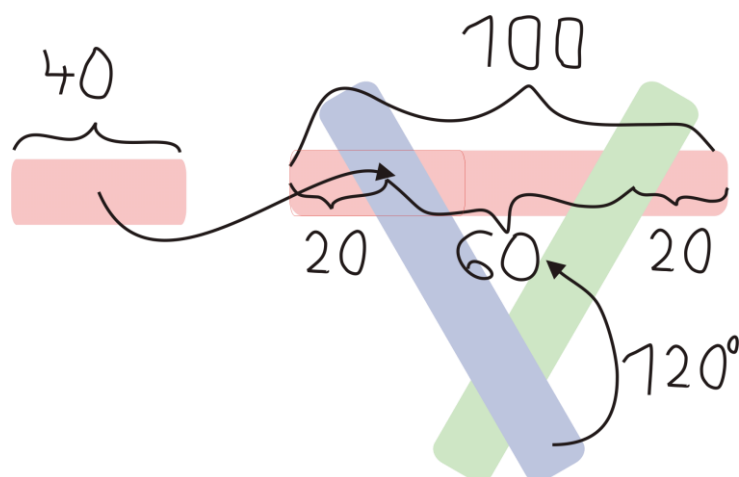
Nakresli si obrázok

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:

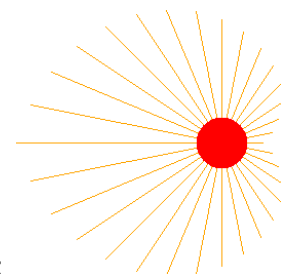
Ak si obrázok nakreslíme a doplníme do neho vzdialenosti a uhly, pomôže nám to jednoduchšie zostaviť výsledný program.





Sprav si zoznam

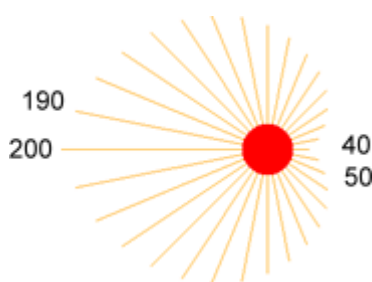
Táto stratégia nám pomôže, ak si potrebujeme nájsť nejaký systém, urobiť vo veciach poriadok.



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 pokračujeme 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 nájsť nájdené riešenie už nie je komplikované.

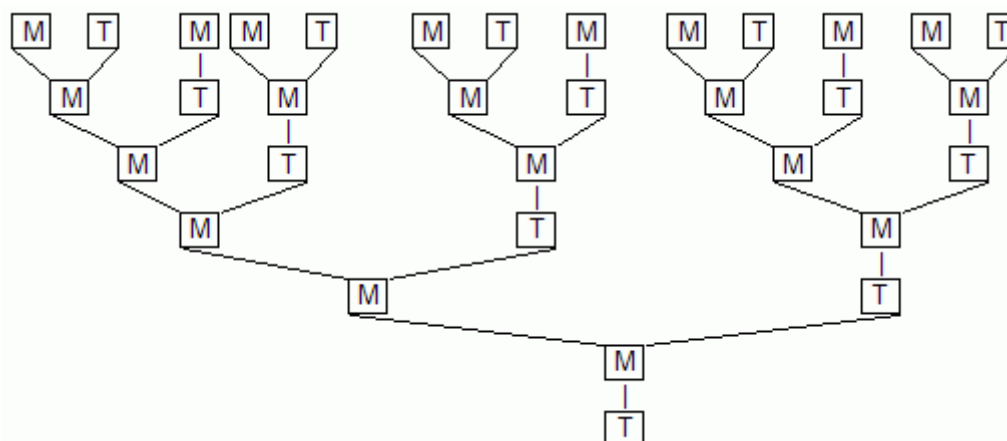
Nájdí vzor

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 problému¹.

V spoločenstve včiel sa vyskytuje včelia matka, robotnice a trúdy. Kým včelia matka a robotnice majú dvoch rodičov (včeliu matku a trúdu), 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 praprapraprarodičov)	8
5 (počet prapraprapraprarodičov)	13
6 (počet praprapraprapraprarodičov)	?

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.

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

Register pojmov

F

funkcia

- dokumentácia, 16
- s návratovou hodnotou, 16
- s parametrom, 16
- vlastná, 15

G

generátor

- pseudonahodných čísel, 30

grafika

- korytnačia, 10

C

chyba

- behová, 22
- logická, 22
- ladenie, 22
- syntaktická, 22

M

modul

- import, 10
- turtle, 10
- vlastný, 19

O

operácia

- aritmetická, 4
- logická, 4

P

pretypovanie, 8

R

range, 13

reťazec

- formátovanie, 9

return. *Pozri* funkcia:s návratovou hodnotou

riadiaca štruktúra

- podmienený príkaz
 - if, 19
- príkaz cyklu
 - for, 12

while, 14

rozhranie

- grafické, 31
- tkinter, 32

rozklad na podproblémy. *Pozri* stratégia riešenia problémov:dekompozícia

S

slučka udalostí, 11, 34

stratégia riešenia problémov, 35

- dekompozícia, 35
- nájdí vzor, 36
- nakresli si obrázok, 35
- sprav si zoznam, 36

súbor, 29

Š

štruktúra

- dátová
 - zoznam, 7
- reťazec, 7

T

tkinter. *Pozri* rozhranie:grafické

V

vstup

- grafické rozhranie, 34
- konzola, 9
- súbor, 30

výnimka, 24

- odchytávanie, 25
- vyhadzovanie, 27

výraz

- aritmetický, 4
- logický, 4

výrez, 7

výstup

- grafické rozhranie, 34
- konzola, 9
- súbor, 29

Z

zdrojový kód

- odsadenie, 12

Obsah

Programovanie v jazyku Python	1
Kľúčové slová	1
Čo sa naučíme a čo si precvičíme	1
Jazyk Python	1
Vývojové prostredie	2
Interaktívny režim	3
Aritmetické výrazy	4
Logické výrazy	4
Reťazec (string), dátová štruktúra zoznam (list)	7
Reťazec	7
Zoznam	7
Výrezy	7
Pretypovanie	8
Programy v súboroch	9
Korytnačia grafika	10
Formátovanie kódu	12
Príkazy cyklu	12
Príkaz cyklu for	12
Príkaz cyklu while	14
Vlastné funkcie	15
Vlastné moduly	19
Podmienený príkaz IF	19
Podmienený príkaz IF	19
Chyby v programoch	22
Syntaktické chyby	22
Logické chyby	22
Behové chyby	24
Práca so súbormi	29
Zápis dát do súboru	29
Čítanie dát zo súboru	30
Grafické rozhranie	31
Stratégie riešenia problémov	35

Dekompozícia	35
Nakresli si obrázok	35
Sprav si zoznam	36
Nájdí vzor.....	36
Register pojmov	38
Obsah.....	39