UNIVERZITA PAVLA JOZEFA ŠAFÁRIKA V KOŠICIACH
Prírodovedecká fakulta
ÚSTAV INFORMATIKY

Ján GUNIŠ, Ľubomír ŠNAJDER

PROGRAMOVANIE V PYTHONE 1

Košice 2021

Vysokoškolský učebný text bol vytvorený v rámci projektu KEGA 029UKF-4/2018 Inovatívne metódy vo výučbe programovania v príprave učiteľov a IT odborníkov.

Bibliografický odkaz:

GUNIŠ, Ján a Ľubomír ŠNAJDER. *Programovanie v Pythone 1*. Košice: Prírodovedecká fakulta UPJŠ, 2021. ISBN 978-80-8152-969-6. Dostupné tiež z: https://unibook.upjs.sk/img/cms/2021/pf/programovanie-v-pythone-1.pdf

Programovanie v Pythone 1

Vysokoškolský učebný text

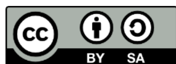
Autori: PaedDr. Ján Guniš, PhD., doc. RNDr. Ľubomír Šnajder, PhD.

Pracovisko: *Ústav informatiky, Prírodovedecká fakulta, UPJŠ v Košiciach*

Recenzenti:

doc. RNDr. Gabriela Andrejková, CSc.
Ústav informatiky, Prírodovedecká fakulta, UPJŠ v Košiciach

PaedDr. Patrik Voštinár, PhD.
Katedra informatiky, Fakulta prírodných vied, UMB v Banskej Bystrici



Obsah podlieha licencii Creative Commons CC BY SA 4.0. Dielo sa môže rozmnožovať, rozširovať a vystavovať. Licencia umožňuje ostatným upravovať, vylepšovať a vytvárať odvodené diela na základe pôvodného diela a to aj na komerčné účely, za predpokladu, že bude uvedený autor pôvodného diela a novovzniknuté diela budú k dispozícií za rovnakých licenčných podmienok.

Umiestnenie: https://unibook.upjs.sk/img/cms/2021/pf/programovanie-v-pythone-1.pdf

Dostupné od: 24.3.2021

ISBN 978-80-8152-969-6 (tlačená publikácia)

ISBN 978-80-574-0009-7 (e-publikácia)

Obsah

Úvod 4

Riešenie problémov 5

1 Jazyk Python 20

2 Korytnačia grafika, vlastné funkcie bez parametrov a návratovej hodnoty........ 28

3 Príkaz cyklu for a funkcia range() 43

4 Funkcie s parametrami a návratovou hodnotou 54

5 Podmienky a podmienený príkaz 67

6 Reťazce a metódy reťazcov 81

7 Algoritmy na reťazcoch 96

8 Chyby a spracovanie výnimiek 105

9 Generovanie výnimiek 119

10 Zoznamy a metódy zoznamov 130

11 Algoritmy na zoznamoch 142

12 Príkaz cyklu while 157

Informačné zdroje 168

Príloha 169

Úvod

Vážení študenti rozširujúceho štúdia informatiky,

vytvorili sme pre Vás skriptá, ktoré by Vám mali slúžiť ako podpora pri štúdiu programovania v jazyku Python. Obsahovo skriptá nadväzujú na výučbu programovania v jazyku Scratch a predstavujú úvod do textového programovania v jazyku Python.

Programovanie chápeme ako prostriedok rozvíjania schopností riešiť problémy, preto sme prvú kapitolu venovali problematike riešenia problémov. Skriptá ďalej obsahujú 12 kapitol, v ktorých by ste si mali osvojiť a precvičiť základné jazykové konštrukcie, dátové štruktúry a vybrané typy algoritmov.

Každá kapitola začína časťou „Jemný úvod do problematiky“, v ktorej sa môžete prvotne oboznámiť s témou kapitoly, a to čítaním stručných učebných textov a experimentovaním s programovými kódmi v interaktívnych Jupyter notebookoch. Táto časť je primárne určená na samoštúdium, ako príprava na cvičenia. Nasledujúca podkapitola „Vysvetlenie problematiky“ sumarizuje a kompletizuje poznatky a sú v nej uvedené aj ďalšie poznatky a prístupy súvisiace s témou kapitoly. V časti „Zbierka úloh“ sú uvedené problémy, riešením ktorých si môžete precvičiť a prehĺbiť vedomostí a zručností z danej témy. V záverečnej časti kapitoly, v podkapitole „Riešené úlohy“ sú uvedené vzorové riešenia vybraných úloh spolu s komentármi a zdôvodnením. Táto časť vám poslúži ako návod na riešenie úloh zo zbierky.

V kapitole „Informačné zdroje“ uvádzame odkazy na zdroje, ktoré sme využili alebo ktoré môžete využiť pri ďalšom štúdiu.

Takto koncipované skriptá sa dajú využiť aj pri tzv. obrátenej výučbe (angl. flipped learning), ktorá umožňuje, aby sa študenti pred samotnou kontaktnou výučbou individuálne v pokoji domova oboznámili so základmi danej témy a v následnej kontaktnej výučbe s učiteľom spoločne riešili vybrané úlohy a diskutovali tieto riešenia. Učiteľ v tejto kontaktnej výučbe poskytne študentom potrebné vysvetlenie a zdôvodnenie potrebné pre pochopenie danej problematiky. Pri ďalšej nekontaktnej výučbe môžu študenti samostatne precvičovať a prehlbovať svoje poznatky a prípadné riešenia svojich úloh konzultovať s učiteľom pomocou diskusného fóra.

Veríme, že tieto skriptá Vám umožnia pokročiť v programovaní v jazyku Python a tiež Vás presvedčiť, že programovanie môže byť zábavné a užitočné.

Ďakujeme recenzentom za cenné rady a pripomienky, ktoré prispeli ku kvalite tejto publikácie.

Autori

Riešenie problémov

Programovanie nie je o písaní programového kódu, ale o hľadaní a nachádzaní riešení problémov. Bolo by naivné predpokladať, že len z popisu problému budeme vedieť zostaviť jeho riešenie. Často musíme s problémom „pracovať", pozrieť sa naň z iného uhla pohľadu, uvedomiť si čo o probléme a jeho riešení vieme a čo nie. Možno objavíme podobnosť s nejakým iným problémov, ktorá nám pomôže vyriešiť aj náš problém. Možno budeme vedieť vyriešiť niektoré jeho časti, zatiaľ čo iné nie. Niekedy nám pomôže vizualizácia, náčrt alebo schéma. A niekedy budeme natoľko „zúfalí", že jediné čo nám zostáva je skúšať.

Počas toho, ako budeme prechádzať obsahom tejto publikácie a riešiť jednotlivé problémy si všimneme, že aj keď riešime rôzne úlohy z rôznych oblastí, v prístupoch ktoré používame sa objavujú určité vzory, prístupy. Všetky uvedené prístupy predstavujú nejaké stratégie riešenia problémov. Znalosť o ich existencii a schopnosť použiť ich nám pomôže aj v situáciách, keď riešime problém z oblasti, o ktorej toho veľa nevieme. V tejto kapitole sa pozrieme sa niektoré z týchto stratégií bližšie.

Pri riešení problémov sa môžeme pridržiavať všeobecne známej schémy (Polya, 1945):

1. Pochopenie problému.
2. Vytvorenie plánu riešenia.
3. Realizácia plánu.
4. Obzretie sa späť.

Etapy riešenia problému

Pochopenie problému

Pochopiť problém, ktorý chceme vyriešiť je základným predpokladom pre úspech. Napriek tomu v tejto oblasti robíme množstvo chýb.

Vidina rýchleho výsledku v podobe napísaného programového kódu nás často ženie nesprávnym smerom. Zadanie problému nečítame celé, pretože už z prvých viet nadobudneme presvedčenie, že vieme o čo ide. Problém často vnímame skreslene, lebo sa nám vybaví iný, na prvý pohľad podobný alebo dokonca rovnaký problém, ktorý sme už v minulosti riešili. Namiesto riešenia zadaného problému tak vytvoríme riešenie nejakého iného problému.

Skôr ako začneme problém riešiť, prečítajme si jeho zadanie pozorne a viackrát, aby sme sa uistili, že zadaniu rozumieme. Skúsme zadanie povedať vlastnými slovami. Prerozprávajme ho niekomu inému, aby sme a uistili, že problém chápeme správne. Klaďme si otázky typu:

- Čo sú vstupné hodnoty, ktoré je potrebné spracovať? V akom formáte sú vstupné hodnoty? Sú nejaké obmedzenia na vstupy?
- Aké výstupné hodnoty je potrebné zistiť? V akom formáte? Čo treba spraviť s výsledkom?
- Ako zo vstupov získame výstupy? Dá sa to vôbec? Máme dostatok informácií alebo potrebujeme ďalšie, doplňujúce informácie?
- Ktoré informácie sú podstatné a ktoré naopak nie?
- Aké sú podmienky riešenia?
- Rozumiem problému správne? Viem ho preformulovať vlastnými slovami? Viem identifikovať podstatu problému?

Klásť si vyššie uvedené otázky a hľadať odpovede na ne, sa môže zo začiatku zdať ako zbytočné zdržanie. Opak je však pravdou. Takáto analýza na začiatku nám môže ušetriť množstvo času venovaného riešeniu problému, prípadne nás ochrániť pred tým, že problém nevyriešime.

Predstavme si, že máme vyriešiť a implementovať riešenie problému: „Prepočítať čas na prípravu a množstvo surovín podľa receptu pre výsledný počet porcií.“ Na prvý pohľad jednoduchá úloha, prináša niekoľko otázok:

- Základný recept je normovaný na počet porcií alebo na celkové množstvo jedla (napr. kg)?
- Môžeme počítať čisto matematicky alebo musíme zohľadniť „nedeliteľnosť“ ingrediencií, napr. 3/7 vajíčka?
- Ak zohľadňujeme nedeliteľnosť ingrediencií, máme pracovať s hornou alebo dolnou časťou? Alebo s tou, ktorá je zaokrúhlením matematickej hodnoty. Upravíme na základe toho výsledné množstvo jedla?
- Ako sa prepočítavajú časové jednotky, Ak sa 1 kg mäsa pečie 1 hodinu, ako dlho je potrebné piecť 3 kg mäsa?
- V akom formáte sú vstupné hodnoty? Sú to čísla alebo reťazce v tvare čísiel? Ak sú to reťazce, musíme vyriešiť ich prevod na čísla.
- Čo má byť výstupom? Čísla, reťazce alebo znenie prepočítaného receptu?
- Sú nejaké limity, ktoré musíme akceptovať? Napr. kapacita kuchyne?

Kým nemáme uspokojivé odpovede na uvedené otázky nemá zmysel pokračovať. Rôzne odpovede v podstate definujú rôzne problémy.

Vytvorenie plánu riešenia

Mať plán a pridržiavať sa ho je dôležité. Vyhneme sa tak bezcieľnej činnosti alebo činnostiam, z ktorých každá nás vedie iným smerom, pričom v globále sa otáčame na mieste. Mať plán neznamená mať do detailu naplánovanú a premyslenú každú činnosť, každý technický alebo implementačný detail. Rovnako by sme mali byť pripravení v prípade potreby plán zmeniť alebo ho úplne opustiť. Začať písať

program bez toho, aby sme aspoň v hlave mali premyslený plán riešenia je nezmysel.

Súčasťou plánu môže byť aj návrh ďalších stratégií, ktoré sa nám javia ako vhodné pre riešenie problému.

Predpokladajme, že potrebujeme vyhodnotiť progres žiakov školy na základe výsledkov vstupného a výstupného testu. Výsledky každého z testov sú uložené v samostatnom súbore. Náš plán pre riešenie tohto problému by mohol vyzerať nasledovne:

- Analyzujme štruktúru súborov a prípadne ich upravme tak, aby sme vedeli výsledky žiaka v každom z nich identifikovať podľa rovnakého identifikátora. Preskúmajme, pre aký typ súborov sú dostupné vhodné knižnice, prípadne zmeňme formát súborov.
- Spárujme výsledky žiakov zo vstupného a výstupného testu. Skúsme použiť nejaký algoritmus alebo dátovú štruktúru tak, aby sme tento proces čo najviac urýchlili. Možno by sa dali použiť dátové štruktúry slovník alebo množiny, ktoré umožňujú rýchly prístup k prvkom. Hľadať pre každého žiaka z prvého súboru výsledky v druhom súbore môže byť pomalé. Pri samotnej implementácii si premyslime nejaký efektívny postup.
- Preskúmajme štatistické metódy a vyberme tú, ktorá je vhodná pre náš prípad. Preskúmajme dostupné knižnice, či táto metóda už nie je implementovaná.
- Spravme niekoľko cielených testov, aby sme si overili, že naša implementácia je správna.
- Vyhodnoťme výsledky žiakov a vytvorme záverečnú správu.

Realizácia plánu

Zatiaľ čo plán riešenia je istým nadhľadom, postupom v hrubých rysoch, v ktorom neriešime detaily, pri jeho realizácii sa tomu nevyhneme. V tejto časti budeme musieť navrhnúť konkrétne postupy ako plán realizovať a postupy implementovať (napr. napísaním programu).

Obzretie sa späť

Vyriešením problému, i keď to znie lákavo, naša práca nekončí. Neoddeliteľnou súčasťou riešenia by mala byť aj reflexia, obzretie sa späť. Je to možnosť ako lepšie pochopiť nájdené riešenie, potvrdiť si jeho správnosť alebo riešenie vylepšiť. V tejto fáze by sme mali hľadať odpovede na otázky typu:

- Rozumiem riešeniu a viem ho vysvetliť alebo zdôvodniť?

  Niekedy sme k riešeniu dospeli, ale nie celkom rozumieme niektorým jeho častiam. Toto je príležitosť vyjasniť si a pochopiť všetky jeho časti.

- Overil som správnosť riešenia? Nezabudol som na nejaké krajné, málo pravdepodobné prípady?

  Pri hľadaní riešenia sa často sústredíme na „typické“, očakávané prípady. Naše riešenie by malo zahŕňať všetky možné prípady. V tejto fáze je dôležite mať pripravené testovacie hodnoty, ktoré zahŕňajú všetky možné inštancie problému. Krajné hodnoty možno vyriešime ako samostatné prípady alebo upravíme nájdené riešenie tak, aby zahrnulo aj tieto krajné hodnoty.

- Dal by sa problém vyriešiť aj iným, efektívnejším spôsobom? Dá sa riešenie vyjadriť aj zrozumiteľnejšie?

  O refaktorizácii (vylepšení) riešenia sa často neuvažuje, pretože riešenie funguje správne a čo viac nám treba? Možno uvedené riešenie v budúcnosti opäť použijeme alebo ho bude chcieť použiť niekto iný. V každom prípade bude užitočné, ak riešenie je efektívne, prehľadné, ľahko čitateľné pretože sa ľahšie udržiava a rozširuje. Refaktorizácia je príležitosť aktualizovať a vylepšiť vnútornú štruktúru nájdeného riešenia (programového kódu) skôr, ako sa stane neprehľadnou alebo spôsobí problém v budúcnosti.

- Ako problém vyriešili iní?

  Porovnať vlastné riešenie s nejakým iným riešením je skvelou príležitosťou naučiť sa niečo nové. Iný pohľad na problém alebo iná implementácia riešenia obohatí naše poznanie. Aj opačná situácia, keď naše riešenie sa ukáže ako najlepšie, má pre nás prínos. Minimálne pozdvihne naše sebavedomie a dodá nám viac istoty pri riešení nasledujúcich problémov.

- Môžeme nájdené riešenie, alebo jeho časť, využiť aj pri riešení iných problémov?

  Jednou z vlastností algoritmov je ich znovu použiteľnosť. Možno si to hneď neuvedomíme, ale naše riešenie môže byť zároveň riešením alebo časťou riešenia nejakého budúceho problému. Alebo nám pomôže vyriešiť iný problém, ktorý sme už v minulosti začali riešiť, ale uviazli sme na mŕtvom bode. V týchto prípadoch nejde ani tak o konkrétne časti programového kódu, ktoré sme napísali. Podstatné sú skôr myšlienky alebo postupy, ktoré sme objavili a použili.

Stratégie riešenia problémov

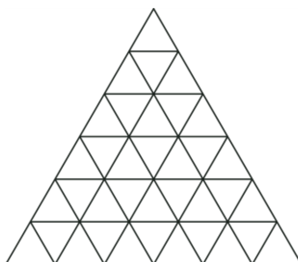
Pozrime sa bližšie na to, aké stratégie pri riešení problémov môžeme využiť. Pri navrhovaných stratégiách si môžeme všimnúť, že navzájom nie sú úplne disjunktné. Vo viacerých môžeme postrehnúť sekundárny presah časti inej stratégie. Niekedy nevieme vopred povedať, ktorá stratégia je tá najvhodnejšia a možno si

na začiatok nevyberieme tu správnu. Postupom času, tak ako získavame skúsenosti s riešením problémov, rastie aj náš cit pre to, kedy ktorú stratégiu použiť.

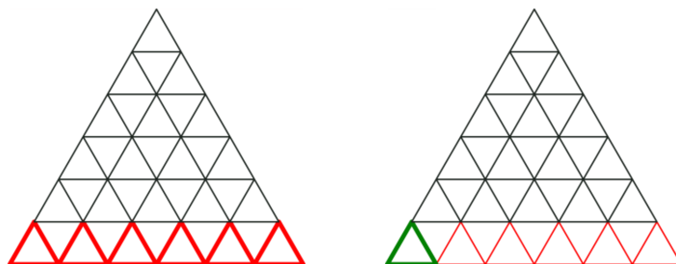
Dekompozícia problému na podproblémy

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 menšie 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.

Predstavme si, že našou úlohou je nakresliť nasledovnú pyramídu z trojuholníkov:



Na prvý pohľad vyzerá pyramída komplikovane. Pozrime sa bližšie z čoho je zložená. Na prvý pohľad vidíme trojuholníky. Trojuholník vieme nakresliť jednoducho. Táto úvaha nám ale veľmi nepomôže. Trojuholníkov je v pyramíde veľa, sú rôzne otočené a v rôznych počtoch. Nájdime ešte nejaký „medzi krok“. Pri bližšom skúmaní si všimneme, že pyramída pozostáva z riadkov, v ktorých sú rovnako orientované trojuholníky:

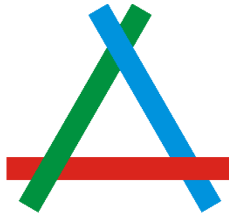


Riadky z trojuholníkov môžu byť medzi krokom, vďaka ktorému vieme z trojuholníkov poskladať celú pýramídu. Problém kreslenia pyramídy sme tak rozložili na menšie problémy (kreslenie riadku trojuholníkov) a tie na ešte menšie (kreslenie trojuholníka). Ak si všimneme počty trojuholníkov v riadkoch vidíme, že smerom hore má každý riadok o jeden trojuholník menej ako ten pod ním (samozrejme okrem toho najspodnejšieho) a zároveň je posunutý o stranu trojuholníka (smerom vpravo hore). Veľký problém sme nahradili niekoľkými menšími problémami a spojením ich riešení vieme vyriešiť aj pôvodný problém.

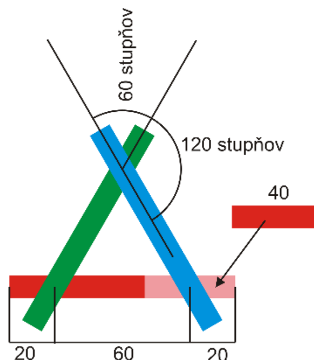
Nakresli si obrázok

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 ňom.

Predstavme si, že máme vykresliť nasledovný útvar:



Na obrázku vidíme tri obdĺžniky, ktoré sa vzájomne prekrývajú. Obdĺžniky sú vzájomne pootočené a pretínajú sa v rovnakých vzdialenostiach od koncov. Problém je aj v tom, že žiaden z obdĺžnikov nie je úplne navrchu. Zvýraznime si, aké vzťahy v obrázku platia.



Ak si obrázok nakreslíme a doplníme do neho vzdialenosti a uhly, pomôže nám to jednoduchšie zostaviť výsledný program. Vyriešili sme aj problém s tým, že žiaden z obdĺžnikov nie je navrchu a časť červeného obdĺžnika sme prekryli „záplatou“ umiestnenou na najvyššej úrovni. Opticky to bude vyzerať tak, že obdĺžniky sa vzájomne prekrývajú.

Vyrieš jednoduchší problém

Ak sme narazili na problém, ktorý nevieme vyriešiť, zamyslime sa, či by sme vedeli vyriešiť nejakú jeho zjednodušenú verziu. Ak sa nám podarí zjednodušenú verziu problému vyriešiť, môže nám to poskytnúť iný pohľad na pôvodný problém. Tento prístup nám pomôže identifikovať problematickú časť riešenia, ktorú môžeme dočasne ignorovať a neskôr jej riešenie zahrnúť do riešenia pôvodného problému.

Predstavme si, že máme zistiť, či zadaný text je palindróm. Pre palindromické texty platí, že sa rovnako čítajú spredu aj zozadu, napr.: `Maťo, ó tam!`. Všimnime si,

že ignorujeme interpunkciu, diakritiku, veľkosť písmen a aj samotné rozdelenie vety na slová.

Ak nevieme nájsť riešenie akceptujúce všetky podmienky, môžeme to byť z viacerých dôvodov:

- nevieme ako „otočiť“ reťazec,
- nevieme, ako zhodne porovnať znaky, ktoré sa líšia len veľkosťou,
- nevieme, ako zhodne porovnať znaky, ktoré sa líšia len diakritikou,
- nevieme, ako ignorovať interpunkciu a delenie slov v texte,
- ...

Podľa toho, ktorý z uvedených bodov je pre nás problematický, vyriešme jednoduchší problém, v ktorom problematické body ignorujeme. Predpokladajme, že problém nám spôsobujú dva posledné body – diakritika a interpunkcia. Vyriešme problém pre texty obsahujú len písmená malej a veľkej anglickej abecedy. Nebudeme porovnávať znaky, ale ich malé verzie. A nemusíme to robiť po jednom, ale zmeňme všetky veľké písmená v texte na malé a porovnajme takto upravené texty.

Pri tomto riešení si možno uvedomíme, že problematické znaky (veľké písmená) sme zmenili na malé. Rovnako môžeme zmeniť znaky s diakritikou na znaky bez diakritiky. Vytvorme si pomocnú funkciu, ktorá v zadanom texte nahradí znak1 znakom2.

`nahrad('ť', 't', 'Maťo, ó tam!')` → `'Mato, ó tam!'`

Potom ju skúsme zovšeobecniť a upravme ju tak, aby funkcia v zadanom texte nahradila znaky jedného reťazca zodpovedajúcimi znakmi iného reťazca.

`nahrad('ťó', 'to', 'Maťo, ó tam!')` → `'Mato, o tam!'`

Podobne vyriešime aj diakritické znamienka. Tie môžeme nahrádzať prázdnym znakom alebo ich jednoducho odstrániť. Riešenie zjednodušeného problému nám „ukázalo“ smer ako vyriešiť aj náročnejšiu verziu problému.

Prístup, keď sa snažíme najskôr vyriešiť jednoduchší problém, môže byť v niektorých prípadoch problematický. Zjednodušenie problému môže byť také výrazné a vzdialené od pôvodného problému, že nám neposkytne iný pohľad na pôvodný problém a riešenie pôvodného problému sa nám touto stratégiou nájsť nepodarí.

Nájdi vzor

V situáciách, v ktorých potrebujeme vysloviť predpovede alebo zovšeobecnenia na základe niekoľkých prípadov, môžeme použiť stratégiu riešenia problémov Nájdi vzor. Pri tejto stratégii analyzujeme objekty ako zložené obrázky, postupnosti hodnôt alebo krokov a snažíme si v nich identifikovať opakujúce sa vzory. Ak ich

identifikujeme, vieme vysloviť predpoveď, ako bude vývoj pokračovať ďalej alebo zovšeobecnenie na základe niekoľkých konkrétnych prípadov.

Hľadanie vzorov nám pomôže pri kreslení komplexnejších obrázkov Predpokladajme nasledovný problém. Chceme pomocou počítača vykresliť nasledovný obrázok:

V obrázku si môžeme všimnúť pravidelne sa opakujúci vzor (alebo aj nejaký iný):

V hľadaní vzorov môžeme pokračovať. Vo vzore ><<>>< identifikujeme ďalšie vzory: >< a <>. Tieto dva vzory sú vytvorené len pomocou > a <, čo je v podstate len jeden vzor pootočený o 180°. Na prvý pohľad komplikovaný obrázok vieme vykresliť len pomocou dvoch (resp. jedného) vzorov, ktoré sú súčasťou väčších a komplexnejších vzorov, v ktorých sa s istou pravidelnosťou opakujú.

Stratégiu „Nájdi vzor“ nemusíme používať len pre obrázky. Predpokladajme nasledovný problém. Chceme postaviť pyramídu z kociek, v ktorej pre susedné vrstvy pyramídy platí, že veľkosti ich strán sa líšia o 1. Koľko kociek budeme potrebovať, ak chceme postaviť pyramídu, ktorá je vysoká ako 10 kociek?

Aby sme odpoveď našli, potrebovali by sme vedieť, aká je závislosť medzi výškou pyramídy a počtom kociek v nej. Pokúsme sa ju objaviť. Začnime od najmenších pyramíd. Najmenšie pyramídy si vieme celkom dobre predstaviť, prípadne nie je problém si ich nakresliť alebo postaviť.

výška pyramídy – h	počet kociek v pyramíde – p(h)
1	1
2	5
3	14
4	30
5	??

...

Niekoľko prvých riadkov sme vyplnili pomerne ľahko. Čím ďalej, tým viac je to náročnejšie. Pozrime sa, čo sa deje s pyramídou, keď do nej pridáme ďalšie poschodie. Pod existujúcu pyramídu „podložíme" ďalšiu, väčšiu základňu. Počet kociek v novej pyramíde je súčtom počtu kociek v predchádzajúcej pyramíde a počtu kociek v novej základni. Doplňme do predchádzajúcej tabuľky aj údaje o novej základni.

výška pyramídy h	počet kociek strany základne	počet kociek v základni	počet kociek v pyramíde p(h)
1	1	1	1
2	2	4	5
3	3	9	14
4	4	16	30
5	5	25	+ 55
...			
`h`	`h`	h^2	h^2 + `p(h-1)`

Z tabuľky už ľahko odvodíme, že počet kociek `p(h)` v pyramíde s výškou `h` je:

$$p(h) = h^2 + p(h-1), \text{ pričom } p(1) = 1$$

Našli sme vzor, podľa ktorého sa zvyšuje počet kociek v pyramíde pridaním ďalšieho poschodia. Toto nám umožní postupne vypočítať počet kociek pre ľubovoľne vysokú pyramídu.

Vyrieš/preskúmaj konkrétny prípad

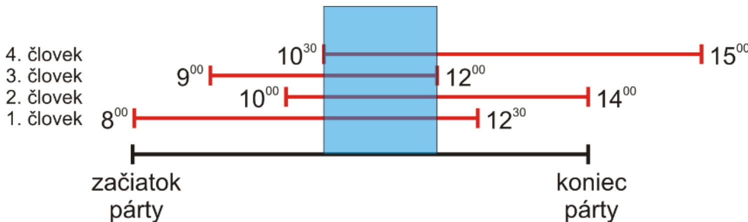
Niektoré problémy môžu byť zadané príliš abstraktne, hromadne. Napriek tomu, že abstrakcia je užitočná, môže byť problematické problém vyriešiť, pretože nemáme nič konkrétne, za čo by sme ho vedeli „uchopiť". Jednou zo stratégií, ktorá nám v tejto situácii môže pomôcť je vyriešiť alebo preskúmať konkrétne prípady. S konkrétnymi hodnotami sa ľahšie manipuluje a sú pre nás pochopiteľnejšie. Využitím poznatkov z riešenia konkrétnych prípadov, môžeme zostaviť všeobecné riešenie.

Predpokladajme, že našou úlohou je vytvoriť funkciu, ktorá pre zadané časy príchodu a odchodu ľudí na párty zistí, či sa všetci ľudia mohli na párty naraz stretnúť. Ľudia mohli prichádzať a odchádzať v rôznych časoch.

Vyriešme nejaký konkrétny prípad, napr.

človek	príchod	odchod
1.	8^{00}	12^{30}
2.	10^{00}	14^{00}
3.	9^{00}	12^{00}
4.	10^{30}	15^{00}

Pre lepšie pochopenie si môžeme časy znázorniť aj graficky:



Pre túto situáciu vidíme, že existuje spoločný čas pre všetkých: 10^{30} - 12^{00} (modrý obdĺžnik). Ak bližšie preskúmame hranice modrého obdĺžnika zistíme, že všetky príchody sú naľavo od neho a všetky odchody sú napravo od neho. Obdĺžnik teda začína tam, kde je posledný čas príchodu a končí tam, kde je prvý čas odchodu. Aby sme ho teda dokázali vytvoriť, posledný príchod by mal byť pred prvým odchodom. Matematicky povedané: `max(príchody) < min(odchody)`. Tým, že sme vyriešili jeden konkrétny prípad sme zároveň našli aj všeobecné riešenie.

Použi analógiu

V situácii, keď ostaneme stáť pri riešení nejakého problému a potrebujeme nejaký nápad ako pokračovať ďalej, môžeme využiť analógiu s nejakou inou oblasťou alebo riešením iného problému. Analógia je založená na spoločných vzťahoch, ktoré nachádzame v aktuálne riešenom probléme a rovnako aj v nejakom inom, už vyriešenom probléme. Ak sa rôzne oblasti zhodujú v určitých aspektoch, potom sa pravdepodobne zhodujú aj v iných aspektoch. Vďaka analógii môžeme spájať na prvý pohľad odlišné oblasti alebo problémy. Analógia, na rozdiel od podobnosti, prepája oblasti na vyššej úrovni na základe vzťahov medzi štruktúrami. Myslieť analogicky znamená pochopiť riešenie problému v jednej oblasti a vytvoriť analogické riešenie problému v inej oblasti.

Typickou programátorskou analógiou je prirovnanie programovania k vareniu. Na začiatku máme zoznam surovín (= vstupné hodnoty), výsledné jedlo (=výstupné hodnoty) a presný recept (=algoritmus, program) ako zo surovín uvariť jedlo (= transformovať vstupné hodnoty na výstupné).

Pri výučbe témy zoznamy v jazyku Python, ukazujeme žiakom podobnosť zoznamov s typom reťazec, ktorý už poznajú. V etape skúmania v jednej z úloh necháme žiakov, aby skúmali konkrétne príkazy a funkcie zoznamov a na základe analógie s reťazcom predpovedali výsledok.

Príkladom použitia analógie sú genetické algoritmy, ktoré sú inšpirované Darwinovou teóriou o vývoji druhov a používajú sa na riešenie optimalizačných problémov. Podobne ako v prírode, aj tu je cieľom nájsť najlepšieho jedinca (objekt, parametre funkcie, umiestnenie objektu ...) pričom sa využívajú postupy analogické s prírodnými: výber jedincov pre ďalšie množenie, kríženie jedincov a mutácie pri rozmnožovaní.

Priebežne testuj

Tak ako riešenie priebežne nachádzame a implementujeme, mali by sme ho aj priebežne testovať. Výber testovacích dát si vyžaduje dobrú analýzu problému. Cieľom testovania nie je „potvrdiť" si správnosť nájdeného riešenia, ale odhaliť jeho nedostatky. Testovacie dáta by sme mali vyberať cielene, aby sme postihli všetky možné inštancie riešeného problému. Odporúča sa, pripraviť si testovacie dáta vopred, keď ešte nie sme ovplyvnení nájdeným riešením. Rovnako by sme mali vopred vedieť, akú odpoveď na testovacie dáta by malo naše riešenie poskytnúť. Vyhneme sa tak situácii, keď sami seba presvedčíme o správnosti chybnej odpovede.

Predpokladajme, že chceme testovať riešenie a implementáciu riešenia problému: Nájsť najväčšieho spoločného deliteľa dvoch prirodzených čísiel. Ak sa zamyslíme nad tým, aké rôzne vzťahy vzhľadom na deliteľnosť môžu medzi dvoma prirodzenými číslami nastať, môžeme uvažovať nasledovné testovacie dáta:

- 1, 12 → 1; 1, 1 → 1

  jedno z čísiel je 1, čo je deliteľ každého prirodzeného čísla,
- 17, 17 → 17; 15, 15 → 15

  čísla sú rovnaké, sú prvočísla alebo zložené čísla,
- 17, 12 → 1

  jedno z čísiel je prvočíslo, druhé je zložené číslo, čísla sú nesúdeliteľné,
- 24, 72 → 24

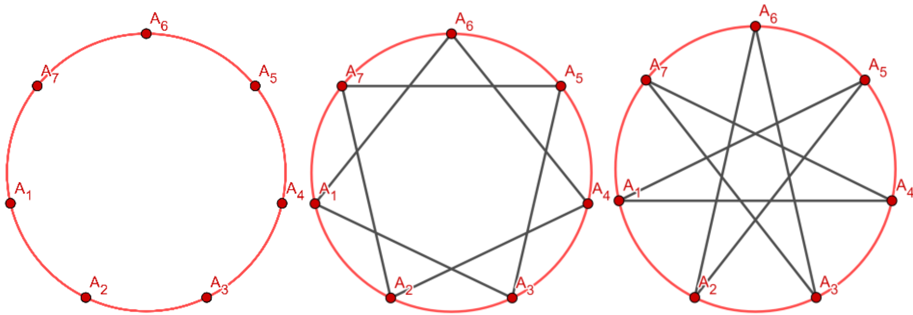
  čísla sú súdeliteľné, pričom jedno z čísiel je deliteľom druhého čísla,
- 72, 108 → 36

  čísla sú súdeliteľné, pričom v prvočíselnom rozklade výsledku (36) majú prvočísla nie jednotkové exponenty ($2^2 * 3^2$).

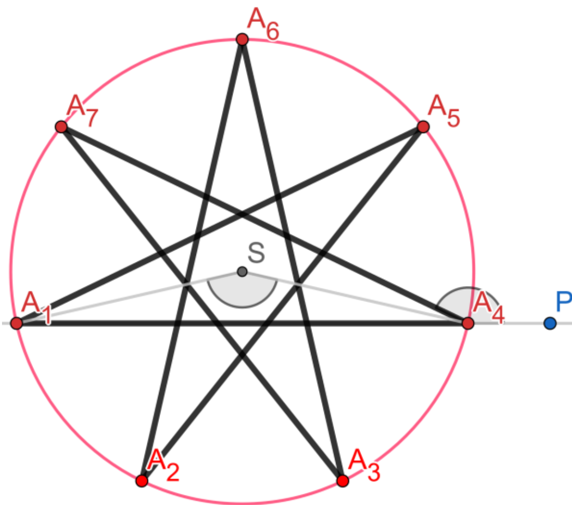
Všimnime si, že nie sme ovplyvnení budúcim algoritmom (prvočíselné rozklady čísiel, Euklidov algoritmus a pod.) Testujeme aj krajné prípady a nie len očakávaný, typický prípad, keď čísla sú súdeliteľné (72, 108). Pre každú testovaciu dvojicu čísiel vieme, aký je správny výsledok. Vyhneme sa tak situácii, že sami seba presvedčíme o správnosti chybného riešenia.

Zaveď pomocný prvok

Zavedením pomocného prvku môžeme objaviť určité súvislosti nápomocné k nájdeniu riešenia problému. Pomocným prvkom môže byť dodatočný grafický či iný typ objektu napr. priamka, kružnica, (štvorcová, trojuholníková, ...) mriežka, pomocná premenná či pomocná funkcia. Častým pomocným prvkom pri grafických problémoch je mriežka, do ktorej zakreslíme obrázok, v ktorom objavujeme a dopĺňame patričné číselné či symbolické hodnoty relevantné s nájdením riešenia problému. V nájdenom riešení problému sa tento pomocný prvok už nemusí vyskytovať.

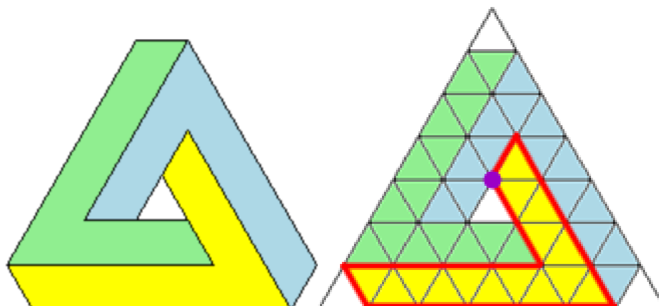


Pri probléme vykreslenia *n*-cípej hviezdy ako lomenej čiary z *n* úsečiek je pomocným prvkom kružnica prechádzajúca všetkými vrcholmi lomenej čiary. Pri vykresľovaní 7-cípej hviezdy môžeme spájať každý duhý, tretí ... bod pomocnej kružnice. Na základe tejto úvahy dospejeme k dvom riešeniam vykreslenia 7-cípej hviezdy, a to s krokom 2 a krokom 3.



Ak chceme nájsť veľkosť uhla natočenia grafického pera pri postupnom vykresľovaní úsečiek, použijeme pomocné prvky stred kružnice a bod P ležiaci

na priamke určenej jedným ramenom hviezdy. Pri vykresľovaní 7-cípej hviezdy s krokom 3 je veľkosť uhla natočenia pera $\angle PA_4A_7$ rovná veľkosti stredového uhla $\angle A_1SA_4$, ktorý je trojnásobkom stredového uhla $\angle A_1SA_2$ s veľkosťou 360 / *n*. Takto dostaneme veľkosť uhla natočenia pera rovnú 3 · (360 / *n*). V prípade vykreslenia *n*-cípej hviezdy s krokom *k* je uhol natočenia grafického pera rovný *k* · (360 / *n*).



Pri vykresľovaní Penroseovho trojuholníka je vhodné použiť trojuholníkovú mriežku, ktorá značne uľahčí určenie opakujúceho sa vzoru spolu s jeho dĺžkami a uhlami. Na obrázku vpravo je zvýraznený tvar vzoru (ohraničený červenou), ktorý sa v pôvodnom obrázku trikrát opakuje. Vzor pozostáva z nepravidelného 6-uholníka s dĺžkami strán [1, 4, 5, 1, 4, 2] a im odpovedajúcimi doplnkovými uhlami [-120, -120, -60, -120, 120, -60]. Počiatok a koniec vykresľovania vzoru je zobrazený kruhom v strede obrázku.

Pokus omyl

Pokus omyl je stratégiou riešenia problému, keď sa prostredníctvom množstva rôznorodých pokusov snažíme nájsť riešenie. Nejde však o úplne náhodné a nepremyslené ;y. Takémuto spôsobu sa radšej vyhnime. Naše pokusy by mali byť systematické a založené na predchádzajúcich skúsenostiach a znalostiach. Táto stratégia sa da použiť, ak máme dostatok času na experimentovanie a množstvo možností, ktoré chceme vyskúšať nie je príliš veľké. Cieľom je nájsť nejaké riešenie, nie nutne všetky riešenia alebo to najlepšie riešenie.

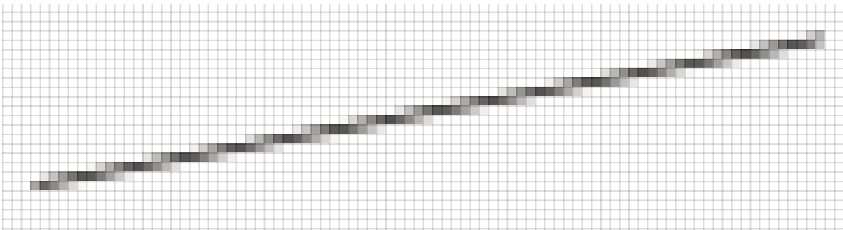
Predstavme si, že sme sa ako začínajúci programátori v Pythone dozvedeli, že program môžeme sprehľadniť tak, že pomocné funkcie umiestnime do samostatného súboru – modulu. Tieto funkcie potom pripojíme pomocou príkazu `import` k hlavnému programu. Máme skúsenosti s importom celých modulov. Nevieme však presne, ako sa pripojí len konkrétna funkcia. A tak začneme skúšať rôzne formáty `import`-u. Skúsime za slovom `import` napísať priamo názov funkcie, potom možno názov súboru a názov funkcie oddelený medzerou, čiarkou alebo bodkou. Možno použijeme aj `import` v kombinácii s `from`. Možností nie je až tak veľa. Nakoniec sa nám to možno podarí, nemusí to byť však jediný spôsob ako to spraviť a ani ten najlepší. Mohli by sme namiesto toho preštudovať

dokumentáciu, ale tá je v jazyku, ktorému nie celkom rozumieme, takže stratégia pokus omyl sa zdá byť najrýchlejšia. Ak by sme nemali žiadne znalosti o moduloch alebo programovaní ako takom, tento spôsob by pravdepodobne neviedol k úspechu.

Metóda čiernej skrinky

Čiernou skrinkou je ľubovoľný neznámy systém, ktorého správanie odhaľujeme a testujeme pomocou podnetov (vstupov) a sledovaných reakcií (výstupov) na tieto podnety.

Metódou čiernej skrinky môžeme odhaliť napr. princíp vyhladzovania hrán, a to vykresľovaním úsečiek rôznej hrúbky s rôznou smernicou a zaznamenaním farieb jednotlivých bodov tejto úsečky.



Použitím nástroja Pipeta v rastrovom grafickom editore zistíme, že na obrázku s úsečkou, ktorú je vykreslená čiernou farbou na bielom pozadí, sú okrem čiernej a bielej farby aj pixely s odtieňmi šedej farby [x, x, x], kde x je z {238, 218, 198, 179, 159, 139, 120, 100, 80, 60, 40, 20}. Ďalším experimentovaním aj s inými farbami pozadia dospejeme k záveru, že pri zobrazení úsečky s vyhladzovaním hrán s určitou farbou pera a farbou pozadia sa pixely úsečky vykresľujú farbou, ktorá je určitou kombináciou farby pera a farby pozadia.

Ďalším príkladom využitia tejto metódy je odhalenie princípu uloženia textových súborov vo formáte UTF-8, kde skúmaním reprezentácií rôznych textov vieme odhaliť „hlavičku" (BOM – angl. byte order mark) UTF-8 formátu a tiež poznatky, že anglické znaky sa kódujú jedným bajtom a znaky s našou diakritikou môžu byť kódované dvomi bajtami.

reťazec	šestnástková reprezentácia v UTF-8
„"	**EF BB BF**
„čaša"	**EF BB BF** C4 8D 61 C5 A1 61
„casa"	**EF BB BF** 63 61 73 61

Pri výučbe programovania môžeme objaviť význam neznámych príkazov resp. metód ich použitím pre rôzne vstupy a analyzovaním ich výstupov. Ďalšou situáciou, v ktorej využijeme metódu čiernej skrinky je rýchle overenie správnosti

výpočtu cudzieho programu bez čítania jeho programového kódu, a to jeho spúšťaním s rôznymi reprezentatívnymi vstupnými hodnotami a analyzovaním vypočítaných výsledkov. Samozrejme to nie je dôkaz jeho správnosti, len sa týmto postupom zvyšuje presvedčenie človeka, že daný program vypočítava správne výsledky.

Vyhnite sa rozptýleniu

Vyhnúť sa rozptýleniu nie je priamo stratégia, ktorej výsledkom by bolo riešenie nejakého problému. Výrazne však umožňuje využiť potenciál ostatných stratégií. Riešenie problémov a programovanie sú intelektuálne náročné činnosti. Často pracujeme na abstraktnej úrovni pričom registrujeme množstvo vzťahov a procesov medzi objektami, ktoré sú súčasťou problému. Pri každom vyrušení sa nám prácne vybudovaný mentálny model problému môže rozpadnúť ako domček z karát. Kým ho znova vytvoríme, zaberie nám to nezanedbateľné množstvo času a energie. Výhodné je preto už na začiatku vytvoriť pracovné prostredie, v ktorom budeme eliminovať rušivé vplyvy okolia.

1 Jazyk Python

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

Možno máte skúsenosť s udalosťami riadeným programovaním (napr. Scratch), so štruktúrovaným alebo objektovým programovaním. Jazyk Python nenúti programátora, aby používal konkrétny štýl programovania, ale dáva mu slobodu používať viaceré štýly. Preto prívlastok **multi-paradigmový**.

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[1] kompatibilnou s GPL (GNU General Public License). Samotný jazyk a jednoduché vývojové prostredie si môžeme stiahnuť zo stránky https://www.python.org/downloads/. V tomto učebnom skripte používame verziu 3.9 alebo novšiu.

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**.

Využívať funkcionalitu, písať a spúšťať programy jazyka Python môžeme v rôznych prostrediach.

Postredie IDLE

IDLE (Integrated Development and Learning Environment) je súčasťou samotnej inštalácie jazyka Python. Interaktívny režim aktivujeme spustením programu IDLE z ponuky Štart|Python. Editor programového kódu spustíme pomocou File|New File. Samotný program, ktorý v tomto prostredí napíšeme spustíme v menu Run|Run Module F5. Toto prostredie poskytuje programátorovi len minimálnu podporu a pomoc.

[1] Licencia jazyka Python https://docs.python.org/3/license.html

Komplexné vývojové prostredie

Pre plnohodnotnú prácu je výhodné využiť niektoré z riešení tretích strán. Jednou z možností je profesionálne prostredie PyCharm, resp. jeho Edu verzia[2]. Vývojové prostredie PyCharm Edu získame na adrese https://www.jetbrains.com/pycharm-edu/. Program spustíme kliknutím na zelenú šípku ▸ alebo klávesovou skratkou Ctrl + Shift + F10.

Interaktívny notebook

Ďalšou z možností je využiť interaktívne Jupyter notebooky. Jupyter notebook je webová stránka, ktorá kombinuje výkladový text s možnosťou priameho spúšťania pythonovských programových kódov. Elektronické verzie notebookov sú súčasťou elektronickej prílohy tohto učebného textu. Stačí ich nahrať na príslušný Jupyter server a môžete s nimi začať priamo pracovať. Jednoduchý návod ako s notebookom pracovať nájdete v súbore **Python 1.1 – Jupyter Notebook.ipynb**.

Jemný úvod do problematiky

Úvod do jazyka Python

Použite notebook **Python 1.2 - Úvod do jazyka Python.ipynb** alebo interaktívny režim jazyka Python.

Cieľ*: V tomto notebooku si ukážeme, ako používať aritmetické operácie v jazyku Python, ako Python reaguje na chyby a ako pomenovať hodnoty pre neskoršie použitie.*

Python je interpretovaný jazyk. Jednotlivé príkazy môžeme spúšťať aj samostatne (v lokálnej inštalácii na to slúži konzola jazyka). Túto vlastnosť budeme v tomto prostredí často využívať.

Python, aritmetické operácie

Výsledkom spustenia nasledujúceho programového kódu je hodnota výrazu: 2 * 12.

```
# spustite tento programvý kód
# poznámka: riadok začínajúci znakom # je komentár
2 * 12
```

Okrem násobenia Python pozná aj ďalšie aritmetické operácie:

- `+` sčítanie
- `-` odčítanie
- `*` násobenie alebo opakovanie reťazca
- `/` delenie

[2] JetBrains poskytuje voľnú licenciu pre študentov a učiteľov aj na ďalšie produkty.

% zvyšok po delení
// celočíselné delenie
** mocnina

Cvičenie 1

Napíšte programový kód, ktorý vyhodnotí výrazy:

- $(3^2 + 4^2)$
- $\sqrt{3^2 + 4^2}$
- zvyšok 123,45 po delení číslom 10,25

Chyby

Nie všetkému čo napíšeme, bude Python rozumieť.

Cvičenie 2

Postupne vyhodnoťte nasledovné výrazy:

- 1/0
- 10 +* 20
- ahoj

Premenná – pomenovanie hodnoty

Hodnoty, s ktorými pracujeme si môžeme pomenovať:

```
premenna = hodnota
```

Dôvodom pre pomenovanie hodnoty je, že cez meno hodnoty vieme k hodnote pristupovať aj neskôr. Navyše, skôr si zapamätáme dobre navrhnuté meno než hodnotu.

Cvičenie 3

Experimentujte s nasledovnými výrazmi. Meňte hodnoty premenných a výrazy opakovane vyhodnocujte.

```
cena = 123.85
zlava = 0.1
```

```
cena
```

```
zlava
```

```
cena_po_zlave = cena - cena * zlava
cena_po_zlave
```

Cvičenie 4

V stánku rýchleho občerstvenia predávajú: hotdog za 55 centov, hamburger za 1 € a 20 centov a hranolčeky za 70 centov. Predavač je síce dobrý kuchár, ale zlý počtár. Pomôžme mu, aby vedel cenu nákupu rýchlejšie vypočítať.

Koľko stojí nákup: 2 * hotdog, 5 * hamburger a 3 * hranolčeky?

Pomôcka: Ak si niektoré hodnoty pomenujete, pomôže vám to pri výpočtoch.

Cvičenie 5

Na čerpacej stanici predávajú benzín, naftu a LPG. Ceny pre dnešný deň boli stanovené nasledovne:

benzín: 1,234 €/liter, nafta: 1,109 €/liter, LPG: 0,520 €/liter.

Ak je kupujúci stálym zákazníkom, dostane zľavu 5 %.

- Koľko má zaplatiť nový zákazník, ktorý natankoval 48,9 litrov benzínu?
- Koľko má zaplatiť stály zákazník, ktorý natankoval 55 litrov nafty?
- Koľko má zaplatiť stály zákazník, ktorý natankoval 45,9 litrov benzínu a 41,2 litrov LPG?

Vysvetlenie problematiky

Jazyk Python je interpreter. Konzolu môžeme využiť na interaktívnu prácu. Každý výraz je po odoslaní vyhodnotený a výsledok vyhodnotenia sa nám v konzole zobrazí.

Hodnoty s ktorými pracujeme si môžeme **pomenovať** (napr. `cena_hranolky = 0.5`). Premennú chápeme ako pomenovanie hodnoty pre jej neskoršie použitie. Pomenovanie by malo byť dostatočne výstižné a popisné, aby sme na základe tohto pomenovania vedeli určiť, aký význam pomenovaná hodnota má. Pre názvy premenných používame malé písmená. Ak je názov zložený z viacerých slov, oddelíme ich znakom podčiarkovník.

S hodnotami vieme vykonávať rôzne operácie, buď priamo alebo cez ich pomenovanie.

Ak pracujeme s **číslami**, môžeme použiť aritmetické operácie:

`+`	sčítanie (`2 + 3.0` → `5.0`)
`-`	odčítanie (`8 - 10` → `-2`)
`*`	násobenie (`3 * 1.5` → `4.5`)
`/`	delenie (`10 / 2` → `5.0`)

`%`	zvyšok po delení (`10 % 2` → `0`, `11.5 // 2` → `1.5`)
`//`	celočíselné delenie (`10 // 2` → `5`, `11.2 // 5` → `2`)
`*`	mocnina (`2 ** 4` → `16`, `8 ** (1 / 3)` → `2.0`)

Ak pracujeme s **reťazcami**, reťazce uzatvoríme do apostrofov `''` a použiť môžeme operácie:

`+`	zreťazenie (`'Ahoj ' + 'kamoš'` → `'Ahoj kamoš'`)
`*`	duplikovanie reťazca (`'Ahoj' * 3` → `'AhojAhojAhoj'`)

Ak spravíme v zápise nejakú **chybu**, Python nás na to upozorní chybovou správou:

`2 / 0`	`ZeroDivisionError: division by zero`
`2 */ 3`	`SyntaxError: invalid syntax`

Vstup do programu realizujeme pomocou funkcie `input()`. Funkcia `input()` zobrazí zadanú výzvu a vráti reťazec reprezentujúci vstup používateľa. Ak očakávame vstup číslo, musíme hodnotu pretypovať (`float()`, `int()`)

```
vek = input('Zadaj svoj vek v rokoch: ')
vek = int(vek)
```

```
Zadaj svoj vek v rokoch: 25
```

Výstup programu realizujeme pomocou funkcie `print()`.

```
print('Ahoj')

pozdrav = 'Ahoj'
print(f'3 x pozdrav {3 * pozdrav}')
```

```
Ahoj
3 x pozdrav AhojAhojAhoj
```

Ak potrebujeme do výstupu vložiť statické texty a hodnoty premenných súčasne, použijeme `f-string`. Reťazec začína prefixom `f` a premenné alebo výrazy, ktoré majú byť nahradené skutočnou hodnotou uvedieme v zátvorkách `{}`.

Zbierka úloh

1. Vytvorte program, ktorý si od používateľa vypýta sumu v EUR a pre pevne zadaný kurz českej koruny (aktuálny výmenný kurz nájdeme na stránke: https://www.nbs.sk/sk/statisticke-udaje/kurzovy-listok/denny-kurzovy-listok-ecb) vypíše hodnotu v českých korunách.

2. Vytvorte program, ktorý si od používateľa vypýta sumu v CZK a pre pevne zadaný kurz českej koruny (aktuálny výmenný kurz nájdeme na stránke: https://www.nbs.sk/sk/statisticke-udaje/kurzovy-listok/denny-kurzovy-listok-ecb) vypíše hodnotu v EUR.

3. Vytvorte program, ktorý si od používateľa vypýta hodnoty koeficientov (a, b, c) kvadratickej rovnice a vypočíta a vypíše korene kvadratickej rovnice.

   Rovnica $ax^2 + bx + c = 0$ má korene:

   $$x_{1,2} = \frac{-b \pm \sqrt{D}}{2a}, \text{ pričom } D = b^2 - 4ac.$$

   Nájdite situáciu, v ktorej výpočet skončí s chybou (overte si to).

4. Vytvorte program, ktorý si od používateľa vypýta veľkosť uhla v stupňoch a vypíše veľkosť uhla v stupňoch, minútach a sekundách.

   napr.: `12.548°` → `12° 32′ 52.8″`

5. Vytvorte program, ktorý si od používateľa vypýta veľkosť uhla v stupňoch, minútach a sekundách a vypíše veľkosť uhla v stupňoch.

   napr.: `12° 32′ 52.8″` → `12.548°`

6. Zapíšte výraz obsahujúci premenné (`a`, `b`, `c`, `d`) tak, aby ho Python vyhodnotil nasledovne:

   $\frac{a+b}{c+d}$ $\quad a + \frac{b}{c} + d$ $\quad \frac{a+b}{c} + d$ $\quad \frac{a}{b+c+d}$

   $a + \frac{b}{c+d}$ $\quad a + \frac{b+c}{d}$ $\quad \frac{a+b+c}{d}$ $\quad \frac{\frac{a}{b}}{\frac{c}{d}}$

   Svoje riešenie si otestujte spustením programu a porovnaním výstupu s očakávanou hodnotou.

7. Vytvorte program, ktorý si od používateľa vypýta dvojciferné číslo `AB` a vypíše k nemu „zrkadlové“ číslo `BA`.

   Napr.: `37` → `73`

8. Vytvorte program, ktorý si od používateľa vypýta prejdenú vzdialenosť v km a vypíše, koľko je to km, m, cm a mm.

   Napr.: `13.4582843` → `13 km 458 m 28 cm 4.3 mm`

Riešené úlohy

1. Vytvorte program, ktorý si od používateľa vypýta sumu v EUR a pre pevne zadaný kurz českej koruny (aktuálny výmenný kurz nájdeme na stránke: https://www.nbs.sk/sk/statisticke-udaje/kurzovy-listok/denny-kurzovy-listok-ecb) vypíše hodnotu v českých korunách.

   Riešenie:

   Aktuálny kurz (13. 4. 2021) k euru je podľa ECB 26,031 CZK. Výhodné je, ak si túto hodnotu a sumu menených peňazí vhodne pomenujeme. Nezabudnime, že vstup je typu reťazec a je potrebné ho pretypovať. Samotný prepočet už nie je náročný.

```
suma_eur = input('Zadaj sumu v EUR: ')
suma_eur = float(suma_eur)
kurz_eur_czk = 26.031
suma_czk = suma_eur * kurz_eur_czk
print(f'{suma_eur} EUR = {suma_czk} CZK')
```

```
Zadaj sumu v EUR: 100
100.0 EUR = 2603.1 CZK
```

4. Vytvorte program, ktorý si od používateľa vypýta veľkosť uhla v stupňoch a vypíše veľkosť uhla v stupňoch, minútach a sekundách.

   napr.: `12.548°` → `12° 32′ 52.8″`

   Riešenie:

   Pri riešení môžeme postupovať tak, že zo zadaného uhla extrahujeme celé stupne, zo zvyšku celé minúty a posledný zvyšok prevedieme na sekundy. Pripomeňme si, že 1′ je rovná 1 / 60° a 1″ je rovná 1 / 3600°.

```
uhol = input('Zadaj veľkosť uhla: ')
uhol = float(uhol)

stupne = uhol // 1
stupne = int(stupne)

uhol = uhol - stupne

minuty = int(uhol * 60) // 1
minuty = int(minuty)

uhol = uhol - minuty / 60
sekundy = uhol * 3600

print(f'{stupne}° {minuty}′ {sekundy}″')
```

```
Zadaj veľkosť uhla: 12.548
```

```
12° 32′ 52.80000000000018″
```

Poznámka: Zdanlivá chyba v počte sekúnd je dôsledkom odlišnej reprezentácie čísiel v počítači. Tejto problematike sa budeme venovať neskôr.

2 Korytnačia grafika, vlastné funkcie bez parametrov a návratovej hodnoty

Jemný úvod do problematiky

Korytnačia grafika

Použite notebook **Python 2.1 - Korytnačia grafika.ipynb** alebo niektoré z lokálnych vývojových prostredí jazyka Python.

Poznámka: V prostredí Jupyter notebook používame modul `ipyturtle2`*. Overte si, či je tento modul na vašom serveri inštalovaný.*

Cieľ: *V tomto notebooku sa zoznámime s korytnačou grafikou a naučíme sa používať základné príkazy korytnačej grafiky.*

Korytnačia grafika je užitočný nástroj pre vizualizáciu priebehu výpočtu. Pracujeme s virtuálnym grafickým perom a výsledok sa postupne zobrazuje v grafickej ploche. Postupné zobrazovanie výsledku nám môže pomôcť odhaliť, v ktorej časti programového kódu je prípadná chyba.

Modul turtle

Modul `turtle` je pythonovský modul, ktorý umožňuje vizualizovať priebeh výpočtu.

Spustite nasledovný programový kód.

Poznámka: Tento kód je určený pre lokálne inštalované vývojové prostredie. Použitie korytnačej grafiky v prostredí Jupyter Notebook je vysvetlené v notebooku Python 2.1 - Korytnačia grafika.ipynb.

```
# importujeme si potrebný modul a vytvoríme virtuálne pero
import turtle
tabula = turtle.Screen()
pero = turtle.Turtle()
```

```
# pomocou virtuálneho pera nakreslíme jednoduchý obrázok (dve
na seba kolmé čiary)
pero.forward(100)
pero.right(90)
pero.pencolor('red')
pero.forward(50)
```

Predchádzajúci programový kód robí nasledovné:

- importujeme modul `turtle`,
- vytvoríme si grafické pero, pomenujeme ho názvom `pero`,

- presunuli sme pero dopredu o 100 bodov,
- otočili ho o uhol 90° vpravo,
- presunuli sme pero dopredu o 50 bodov.

Príkazy korytnačej grafiky

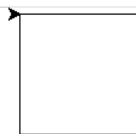
Grafické pero má definovaných niekoľko príkazov. Experimentujte s uvedenými príkazmi, aby ste zistili, aká je ich funkcionalita.

- `pero.forward()`
- `pero.back()`
- `pero.left()`
- `pero.right()`
- `pero.reset()`
- `pero.penup()`
- `pero.pendown()`
- `pero.pencolor()`

O ďalších možnostiach použitia korytnačej grafiky na lokálnom počítači sa dozviete v odkaze na konci tejto podkapitoly.

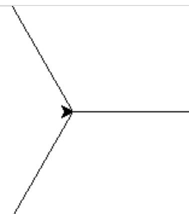
Cvičenie 1

Napíšte programový kód, ktorý vykreslí štvorec:



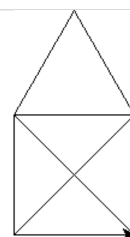
Cvičenie 2

Napíšte programový kód, ktorý vykreslí hviezdičku:



Cvičenie 3

Napíšte programový kód, ktorý vykreslí domček jedným ťahom:



Dokumentácia k modulu `turtle` je k dispozícii na https://docs.python.org/3.9/library/turtle.html.

Vlastné funkcie bez parametrov a návratovej hodnoty

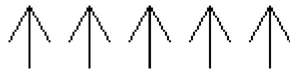
Použite notebook **Python 2.2 - Vlastné funkcie bez parametrov a návratovej hodnoty.ipynb** alebo niektoré z lokálnych vývojových prostredí jazyka Python.

.

***Cieľ:** V tomto notebooku si pripomenieme jednu zo stratégií riešenia problémov – dekompozíciu. Ukážeme si, ako definovať a používať vlastné funkcie, riešiace jednotlivé podproblémy.*

O jednej zo stratégií riešenia problémov – dekompozícii sme už hovorili v jednej z predchádzajúcich kapitol. Stratégia spočíva v tom, že veľký problém rozdelíme na menšie, tie vyriešime a z ich riešení poskladáme riešenie pôvodného veľkého problému.

Predstavme si, že chceme nakresliť takýto obrázok:



Z obrázka vidno, že potrebujeme nakresliť 5 šípok. Ak tento problém rozdelíme na menšie, prirodzene dospejeme k problému: vykreslenie jednej šípky. Definujme si postupnosť príkazov na kreslenie jednej šípky a pomenujme si ju `sipka`.

```
def sipka():
    pero.forward(30)
    pero.right(30)
    pero.back(20)
    pero.forward(20)
    pero.left(60)
    pero.back(20)
    pero.forward(20)
    pero.right(30)
    pero.back(30)
```

Definícia začína slovom `def`, za ktorým nasleduje názov funkcie, zátvorky a dvojbodka. Dvojbodka znamená, že nasleduje blok príkazov. Všimnime si, že blok príkazov je odsadený vpravo.

Poznámka: Pre odsadenie bloku príkazov sa štandardne používajú štyri medzery.

Samotné použitie funkcie je jednoduché:

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

sipka()

tabula.mainloop()
```

Cvičenie 1

Doplňte do predchádzajúceho programového kódu vykreslenie druhej šípky a programový kód spustite. Vykreslili sa dve šípky?

Na kreslenie ďalších šípok musíme kresliace pero posunúť. Keďže aj túto činnosť budeme vykonávať viackrát, definujme si funkciu `posun`.

```
def posun():
    pero.penup()
    pero.right(90)
    pero.forward(30)
    pero.left(90)
    pero.pendown()
```

Cvičenie 2

Otestujte kreslenie šípky s posunom pre vykreslenie druhej šípky.

Samotné kreslenie piatich šípok vyzerá potom nasledovne:

```
pero.reset()  # zmažeme predchádzajúci výstup
sipka()
posun()
sipka()
posun()
sipka()
posun()
sipka()
posun()
sipka()
posun()
```

Všimnime si, že dvojicu `sipka()` a `posun()` voláme vždy spolu. Je preto rozumné, definovať si funkciu `sipka_s_posunom()`.

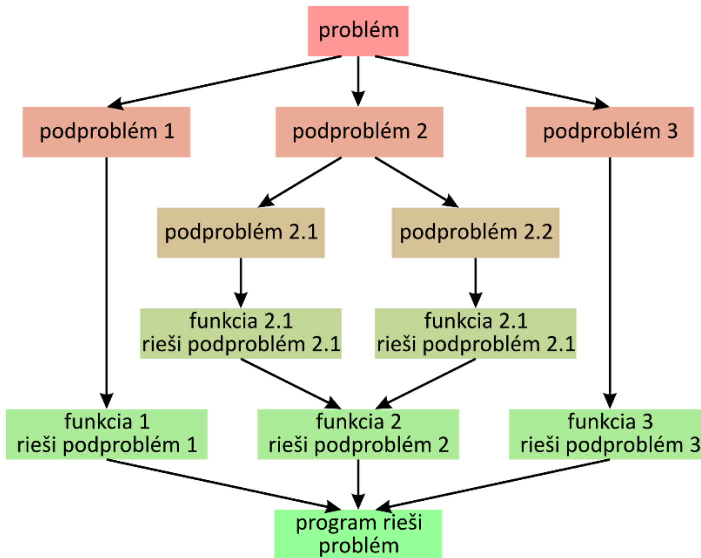
```
def sipka_s_posunom():
    sipka()
    posun()
```

Kreslenie piatich šípok bude potom vyzerať nasledovne:

```
pero.reset()  # zmažeme predchádzajúci výstup
sipka_s_posunom()
sipka_s_posunom()
sipka_s_posunom()
sipka_s_posunom()
sipka_s_posunom()
```

Ak vám napadlo, že takéto opakovanie by sa dalo vyriešiť pomocou nejakého príkazu opakovania, uvažujete správne. Príkaz opakovania si ukážeme v nasledujúcej časti.

Vo všeobecnosti môžeme tento postup rozdeľovania problému do podproblémov a ich riešenie pomocou funkcií znázorniť nasledovne:



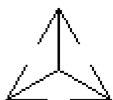
Predchádzajúci programový kód sme vytvárali po častiach a nikde ho zatiaľ nevidíme ako celok. Ak by sme ho napísali ako celok, vyzeral by nasledovne:

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

def sipka():
    pero.forward(30)
    pero.right(30)
    pero.back(20)
    pero.forward(20)
    pero.left(60)
    pero.back(20)
    pero.forward(20)
    pero.right(30)
    pero.back(30)

def posun():
    pero.penup()
    pero.right(90)
    pero.forward(30)
    pero.left(90)
    pero.pendown()
```

```
def sipka_s_posunom():
    sipka()
    posun()

sipka_s_posunom()
sipka_s_posunom()
sipka_s_posunom()
sipka_s_posunom()
sipka_s_posunom()

tabula.mainloop()
```

Cvičenie 3

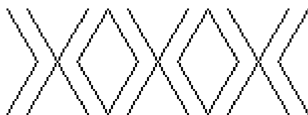
Napíšte programový kód pre vykreslenie nasledovného obrázka:



Môžete využiť, že niektoré funkcie sú už definované.

Cvičenie 4

Napíšte programový kód pre vykreslenie nasledovného obrázka:



Analyzujte z akých častí sa obrázok skladá a navrhnite vhodné funkcie pre ich vykreslenie.

Vysvetlenie problematiky

Korytnačia grafika

Korytnačia grafika je jednou z možností ako vizualizovať priebeh výpočtu. Nie je teda cieľom, ale len prostriedkom. Prostriedkom, ktorý nám pomôže odhaliť prípadné chyby v programoch. Keď vidíme, v ktorej časti kreslenia nastala chyba, vieme túto chybu ľahšie spojiť s konkrétnou časťou programu. Pre inicializáciu grafického pera budeme používať nasledovnú schému:

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

```
# príkazy pre kreslenie pomocou grafického pera

tabula.mainloop()  # [4]
```

[1] importujeme modul `turtle`,

[2] vytvoríme kresliacu plochu, do ktorej budeme perom kresliť, vytvorenú kresliacu plochu sme pomenovali identifikátorom `tabula`,

[3] vytvoríme grafické pero a pomenujeme ho identifikátorom `pero`, všetky požiadavky na kreslenie budú začínať prefixom `pero`,

[4] spustíme nekonečnú slučku udalostí `mainloop()`, dôsledkom je, že okno s grafickou plochou ostáva otvorené aj po skončení kreslenia, okno zatvoríme kliknutím na ikonku × v pravej hornej časti okna.

Grafické pero je v základnej pozícii umiestnené v strede grafickej plochy a natočené smerom na východ.

Samotné grafické pero má pomerne bohatú funkcionalitu. Pre naše potreby budeme používať metódy:

`pero.forward(vzdialenosť)` – Grafické pero sa presunie v aktuálnom smere o zadanú vzdialenosť. Je možné použiť aj zápornú hodnotu (pero sa bude pohybovať smerom vzad).
`pero.backward(vzdialenosť)` – Grafické pero sa presunie v opačnom smere o zadanú vzdialenosť. Môžeme povedať, že pero „cúva“. Je možné použiť aj zápornú hodnotu (pero sa bude pohybovať smerom vpred).
`pero.right(uhol)` – Grafické pero sa otočí o zadaný uhol (v stupňoch) doprava. Je možné použiť aj zápornú hodnotu (pero sa bude otáčať vľavo).
`pero.left(uhol)` – Grafické pero sa otočí o zadaný uhol (v stupňoch) doľava. Jje možné použiť aj zápornú hodnotu (pero sa bude otáčať vpravo).
`pero.circle(polomer)` – Grafické pero vykreslí kružnicu so zadaným polomerom. Kružnica sa kreslí z aktuálnej pozície grafického pera s postupným otáčaním grafického pera doľava. Je možné použiť aj zápornú hodnotu (pero sa bude otáčať doprava).
`pero.dot(priemer)` – Grafické pero vykreslí kruh so zadaným priemerom.
`pero.speed(rychlost)` – Nastaví rýchlosť vykresľovania, 0 – najrýchlejšie, možné sú aj ďalšie hodnoty od 1 (najpomalšie) až po 10 (rýchlo).
`pero.pendown()` – Grafické pero sa „priloží“ na tabuľu, pri pohybe zanecháva stopu.

`pero.penup()` – Grafické pero sa „zdvihne“, pri pohybe nezanecháva stopu.
`pero.width(hrubka)` – Nastaví hrúbku pera pri kreslení.
`pero.pencolor(farba)` – Nastaví farbu kreslenia, farbu zadáme názvom (napr. `'red'`) alebo ako trojicu definujúcu úroveň RGB (napr. `(1, 1, 0)`, pričom 0 je najnižšia a 1 najvyššia úroveň).
`pero.fillcolor(farba)` – Nastaví farbu výplne, farbu zadávame rovnako ako pri `pero.pencolor(farba)`.
`pero.beginfill()` – Metódu voláme pred začiatkom kreslenia krivky, ktorá má byť vyplnená.
`pero.endfill()` – Metódu zavoláme po ukončení kreslenia krivky, krivka sa uzatvorí a ohraničená plocha sa vyfarbí.
`pero.reset()` – Vymaže výsledok kreslenia a presunie kresliace pero do stredu kresliacej plochy.
`pero.clear()` – Vymaže výsledok kreslenia.
`pero.hideturtle()` – Skryje grafické pero, pričom následné kreslenie sa zrýchli.
`pero.showturtle()` – Zobrazí grafické pero.

Aj keď niektoré príkazy majú svoje skrátené formy (`pero.forward()` skrátene `pero.fd()`) v tomto učebnom texte používame neskrátené formy. Z neskráteného výrazu je ľahšie dedukovať význam daného príkazu.

Kresliacich pier si môžeme vytvoriť viac a každé z nich ovládať samostatne.

```
import turtle
tabula = turtle.Screen()
pero1 = turtle.Turtle()
pero2 = turtle.Turtle()

pero1.forward(100)
pero2.left(90)
pero2.forward(50)

tabula.mainloop()
```

Vlastné funkcie bez parametrov a návratovej hodnoty

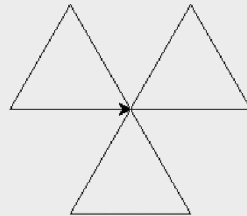
Ak máme nejakú postupnosť príkazov, ktorú potrebujeme vykonávať opakovane, nemusíme ju v programe opakovane „opisovať“. Uvedenú postupnosti môžeme definovať ako samostatnú funkciu. Funkcia pomenuje túto skupinu príkazov

a zavolaním tohto mena sa uvedená postupnosť vykoná. Pre názvy funkcií používame malé písmená. Ak je názov zložený z viacerých slov, oddelíme ich znakom podčiarkovník.

Rovnaký prístup môžeme využiť aj v prípade, ak riešime „veľký“ problém. Ten môžeme rozdeliť na menšie, jednoduchšie riešiteľné problémy. Pre riešenie každého z menších problémov definujeme vlastnú funkciu.

Kreslenie vrtuľky z troch rovnostranných trojuholníkov môžeme realizovať nasledovne:

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

def trojuholnik():
    pero.forward(100)
    pero.left(120)
    pero.forward(100)
    pero.left(120)
    pero.forward(100)
    pero.left(120)

def trojuholnik_s_otockou():
    trojuholnik()
    pero.left(120)

def vrtulka():
    trojuholnik_s_otockou()
    trojuholnik_s_otockou()
    trojuholnik_s_otockou()

vrtulka()
tabula.mainloop()
```

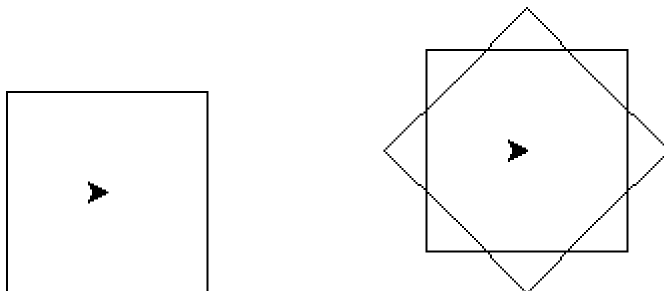


Všimnime si odsadenie bloku príkazov. Python nepoužíva žiadne obaľovacie znaky (napr. zátvorky {}) alebo kľúčové slová (napr. begin, end) na určenie bloku programu (telo cyklu, telo funkcie a pod). Blok je definovaný odsadením. Pre odsadenie (=určenie bloku) stačí 1 medzera. Nech sa rozhodneme akokoľvek, musíme to v celom programe jednotne dodržiavať. Pre odsadenie sa odporúčajú 4 medzery. Všimnime si, že riadok pred začiatkom bloku končí znakom dvojbodka.

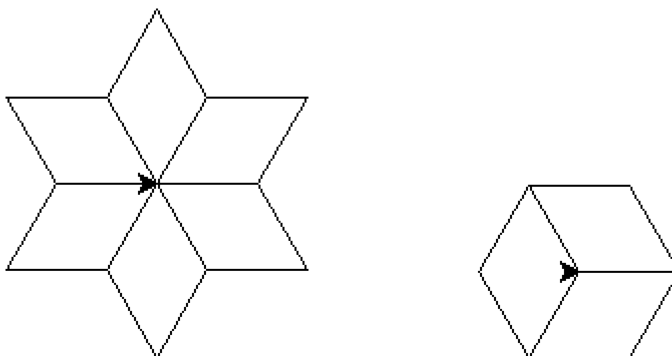
Zbierka úloh

Nasledujúce kresliace úlohy sú zamerané na dekompozíciu. Pokúste sa identifikovať vzory v obrázkoch resp. jednotlivé podproblémy a navrhnúť na ich vykreslenie, resp. riešenie vhodné funkcie. Pri niektorých obrázkoch sa predpokladá aj viacúrovňová dekompozícia.

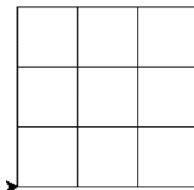
1. Vytvorte program, ktorý vykreslí štvorec/štvorce okolo korytnačky (začiatočná aj koncová pozícia korytnačky je v strede štvorca):



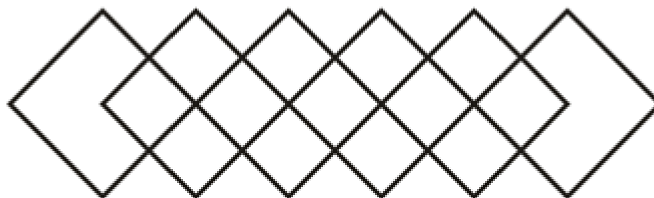
2. Vytvorte program, ktorý vykreslí nasledovné obrázky:



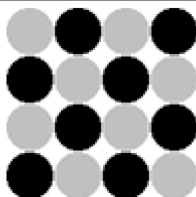
3. Vytvorte program, ktorý vykreslí nasledovnú mriežku:



4. Vytvorte program, ktorý vykreslí nasledovný obrázok:

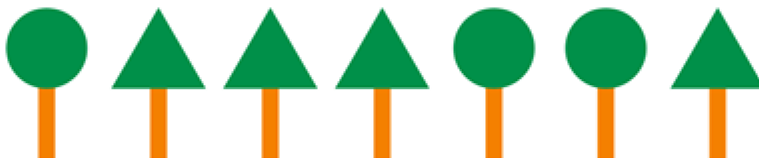


5. Vytvorte program, ktorý vykreslí nasledovnú bodkovanú šachovnicu:



Pomôcka: Čiernu bodku veľkosti 40 vykreslíte príkazom `pero.dot(40, 'black')`, *šedá farba má pomenovanie* `'silver'`.

6. Navrhnite funkcie pre kreslenie stromoradí typu:



Pomôcka: Vyplnený trojuholník nakreslíte nasledovne:

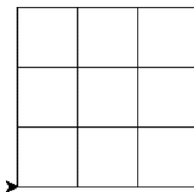
```
pero.fillcolor('green')
pero.begin_fill()
# samotné kreslenie trojuholníka
pero.end_fill()
```

7. Vytvorte program, ktorý vykreslí nasledovný obrázok dlaždice:

8. Vytvorte funkcie `rad()`, `dvoj_rad()` a `striedavy_dvoj_rad()`, ktoré vykreslia nasledovné obrázky:

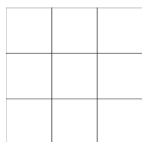
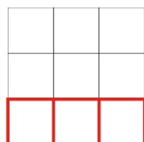
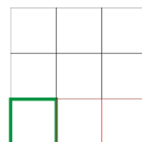
Riešené úlohy

3. Vytvorte program, ktorý vykreslí nasledovnú mriežku:



Riešenie:

Úloha je zameraná na dekompozíciu problému na podproblémy. V mriežke na prvý pohľad rozpoznáme niekoľko štvorcov. Mohli by sme teda problém „mriežka“ nahradiť niekoľkými menšími problémami typu „štvorec“. Ak by sme mali vykresliť deväť štvorcov usporiadaných do mriežky, trochu sa „potrápime“ s ich usporiadaním. Nájdime inú dekompozíciu, napr. mriežka – rad štvorcov – štvorec.

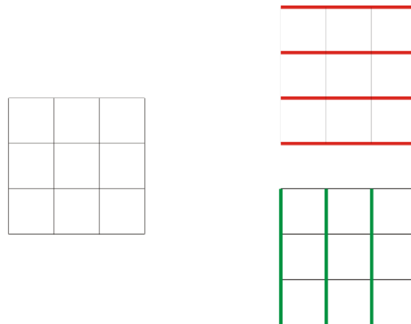
Na riešenie príslušných problémov definujme zodpovedajúce funkcie.

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

def stvorec():
    pero.forward(50)
    pero.left(90)
    pero.forward(50)
    pero.left(90)
    pero.forward(50)
    pero.left(90)
    pero.forward(50)
    pero.left(90)

def rad():
    stvorec()
    pero.forward(50)
    stvorec()
    pero.forward(50)
    stvorec()
    pero.backward(100)
```

```
def mriezka():
    rad()
    pero.left(90)
    pero.forward(50)
    pero.right(90)
    rad()
    pero.left(90)
    pero.forward(50)
    pero.right(90)
    rad()
    pero.left(90)
    pero.backward(100)
    pero.right(90)

pero.speed(0)
mriezka()
tabula.mainloop()
```

Všimnime si ešte jeden detail. Funkcie sme navrhli tak, že žiadna z nich nezmení stav kresliaceho pera. Aj keď funkcia pomocou pera vykreslí nejaký útvar, pero vždy vráti do východiskovej pozície. Toto pravidlo je vhodné dodržiavať. Funkcie sa ľahšie „skladajú" ak vieme, že po volaní funkcie sa stav pera nezmení.

Zamyslime sa ešte nad použitou dekompozíciou. Toto nie je jediná možnosť. V obrázku nemusíme vidieť len štvorčeky. Môžeme v ňom identifikovať štyri vodorovné a štyri zvislé čiary.

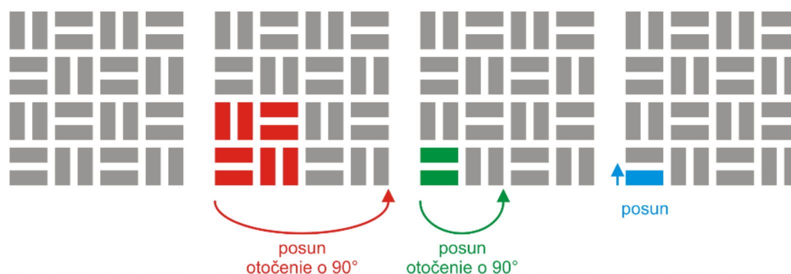


Na kreslenie každej štvorice môže využiť jednu a tú istú funkciu, stačí pred kreslením presunúť a pootočiť kresliace pero.

7. Vytvorte program, ktorý vykreslí nasledovný obrázok dlaždice:

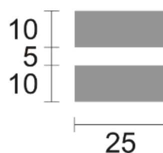
Riešenie:

Úloha je zameraná na dekompozíciu a následnú definíciu funkcií riešiacich identifikované podproblémy. V obrázku môže identifikovať malé šedé obdĺžniky. Dekomponovať celý obrázok priamo na šedé obdĺžniky nám ale veľmi nepomôže. Musíme nájsť plynulejšiu dekompozíciu. Pozrime sa, ktoré časti sa v obrázku opakujú. Postupujme od tých najväčších až po tie najmenšie:



V obrázku dlaždice sme identifikovali vzor štyroch dvojíc obdĺžnikov, v ňom vzor dvojice obdĺžnikov a v ňom vzor obdĺžnik. Niektoré vzory sú vykreslené rôzne otočené, ale to vyriešime natočením grafického pera.

Pred samotným definovaním funkcií sa zamyslime aj nad rozmermi jednotlivých častí. Tu je rozumné začať od obdĺžnikov, pretože dvojica obdĺžnikov je vpísaná do štvorca. Rozmery môžeme nastaviť nasledovne:




```
import turtle

tabula = turtle.Screen()
pero = turtle.Turtle()

def sedy_obdlznik():
    pero.begin_fill()
    pero.forward(25)
    pero.left(90)
    pero.forward(10)
    pero.left(90)
    pero.forward(25)
    pero.left(90)
    pero.forward(10)
    pero.left(90)
    pero.end_fill()
```

```
def dvojica_obdlznikov():
    sedy_obdlznik()
    pero.left(90)
    pero.forward(15)
    pero.right(90)
    sedy_obdlznik()
    pero.left(90)
    pero.backward(15)
    pero.right(90)

def styri_dvojice_obdlznikov():
    dvojica_obdlznikov()
    pero.forward(55)
    pero.left(90)
    dvojica_obdlznikov()
    pero.forward(55)
    pero.left(90)
    dvojica_obdlznikov()
    pero.forward(55)
    pero.left(90)
    dvojica_obdlznikov()
    pero.forward(55)
    pero.left(90)

def dlazdica():
    styri_dvojice_obdlznikov()
    pero.forward(115)
    pero.left(90)
    styri_dvojice_obdlznikov()
    pero.forward(115)
    pero.left(90)
    styri_dvojice_obdlznikov()
    pero.forward(115)
    pero.left(90)
    styri_dvojice_obdlznikov()
    pero.forward(115)
    pero.left(90)

pero.speed(0)
pero.penup()
pero.fillcolor('gray')
dlazdica()
tabula.mainloop()
```

Uvedená dekompozícia a spôsob vykresľovania nie sú jediné možné.

3 Príkaz cyklu for a funkcia range()

Jemný úvod do problematiky

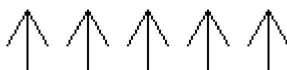
Cyklus for a funkcia range()

Použite notebook **Python 3 - Príkaz cyklu for a funkcia range().ipynb** alebo niektoré z lokálnych vývojových prostredí jazyka Python.

Cieľ: *V tomto notebooku si pripomenieme ďalšiu zo stratégií riešenia problémov – hľadaj vzor. Ukážeme si, ako opakovať postupnosť príkazov pomocou príkazu cyklu* `for` *a ako pri opakovaní využiť funkciu* `range()`.

Príkaz cyklu for

Po dekompozícii niektorých problémov na podproblémy zistíme, že rovnaké podproblémy sa opakujú niekoľkokrát za sebou. Vráťme sa k jednej z úloh z predchádzajúcej časti. Mali sme vytvoriť program pre vykreslenie obrázka:



Problém sme vyriešili nasledovne:

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

def sipka():
    pero.forward(30)
    pero.right(30)
    pero.back(20)
    pero.forward(20)
    pero.left(60)
    pero.back(20)
    pero.forward(20)
    pero.right(30)
    pero.back(30)

def posun():
    pero.penup()
    pero.right(90)
    pero.forward(30)
    pero.left(90)
    pero.pendown()

def sipka_s_posunom():
    sipka()
    posun()
```

```
sipka_s_posunom()
sipka_s_posunom()
sipka_s_posunom()
sipka_s_posunom()
sipka_s_posunom()

tabula.mainloop()
```

Samotnú funkciu `sipka_s_posunom()` sme opakovane volali 5-krát za sebou. Túto časť vieme šikovnejšie, pomocou príkazu cyklu, zapísať takto:

```
pero.reset()
for pocitadlo in range(5):
    sipka_s_posunom()
```

Všimnime si, že príkazy, ktoré sa majú v cykle opakovať sú odsadené vpravo. S hodnotou v zátvorke funkcie `range()` môžeme experimentovať.

V tomto prípade je pre nás dôležité, že príkaz `sipka_s_posunom()` sa vykonal 5-krát. V istých situáciách bude pre nás dôležitý nie len celkový počet opakovaní, ale aj samotný priebeh opakovania. Na premennú `pocitadlo` sa pozrieme v nasledujúcej podkapitole.

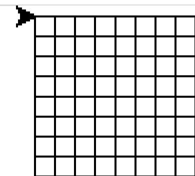
Cvičenie 1

Napíšte programový kód pre vykreslenie nasledovného obrázka:

Pomôcka: Premyslite si, pre ktoré časti obrázka je vhodné definovať si samostatné funkcie.

Cvičenie 2

Napíšte programový kód pre vykreslenie mriežky:



Funkcia range()

Premenná `pocitadlo` sa správa podobne, ako sa správalo počítadlo, ktoré sme používali v jazyku Scratch. Na rozdiel od jazyka Scratch, sa zmena hodnoty počítadla pri každom opakovaní cyklu deje automaticky. Pozrime sa na to, akú hodnotu premenná `pocitadlo` nadobúda:

```
for pocitadlo in range(5):
    print(pocitadlo)   # príkaz print slúži na výpis hodnoty
```

Časť `range()` v cykle `for` vieme použiť rôznymi spôsobmi a ovplyvniť tak priebeh zmeny počítadla. Spustite nasledovné bloky príkazov a experimentuje s hodnotami v časti `range()`.

```
# určíme hodnoty počítadla OD a DO
for pocitadlo in range(5, 15):
    print(pocitadlo)
```

```
# určíme hodnoty počítadla OD, DO a KROK
for pocitadlo in range(5, 15, 3):
    print(pocitadlo)
```

```
# určíme hodnoty počítadla OD a DO, KROK môže byť aj záporný
for pocitadlo in range(50, 0, -5):
    print(pocitadlo)
```

```
# hodnoty počítadla môžeme v cykle použiť,
# spočítajme párne čísla do 100
sucet = 0
for pocitadlo in range(0, 101, 2):
    sucet = sucet + pocitadlo
print(sucet)
```

Poznámka: Tento príklad sme uviedli len ako demonštráciu pre využitie počítadla v cykle. Ak by sme v skutočnosti potrebovali spočítať párne čísla do 100, spravíme to jednoduchšie takto:

```
sucet = sum(range(0, 101, 2))
print(sucet)
```

alebo použijeme vzorec pre výpočet súčtu prvých n členov aritmetickej postupnosti: $S_n = \frac{(a_1+a_n)\ n}{2}$.

Cvičenie 3

Napíšte programový kód, ktorý vypíše druhé mocniny prirodzených čísiel od 1 do 10.

Cvičenie 4

V zábavnom podniku RockMath vyberajú vstupné zaujímavým spôsobom. Prvý zákazník zaplatí 1 €. Druhý neplatí nič. Tretí zaplatí 3 €, Štvrtý opäť nič. Piaty 5 € atď. Koľko vyzbierajú na vstupnom, ak do podniku prišlo *n* zákazníkov?

Napíšte programový kód, ktorý vypočíta celkové vstupné.

Vysvetlenie problematiky

Príkaz cyklu for

V situáciách, v ktorých potrebuje opakovane vykonávať rovnakú postupnosť príkazov využijeme riadiaci príkaz – príkaz cyklu `for`. Iterácie (opakovania) cyklu `for` sa vždy realizujú nad nejakou sekvenciou (postupnosťou hodnôt).

V niektorých prípadoch je pre nás dôležitý len počet prvkov danej sekvencie:

```
# vypíšeme 3 x Sláva
for i in range(3):
    print('Sláva')
```

```
Sláva
Sláva
Sláva
```

Vo vyššie uvedenom príklade sme využili fakt, že funkcia `range(3)` vygeneruje 3 hodnoty. V princípe nám nezáleží na tom, aké to sú hodnoty.

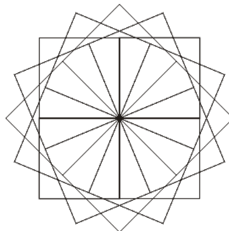
V iných prípadoch budú pre nás dôležité aj konkrétne hodnoty postupnosti:

```
# vypíšeme 3 x Sláva aj s poradím
for pocitadlo in range(1, 4):
    print(f'Sláva {pocitadlo}. krát')
```

```
Sláva 1. krát
Sláva 2. krát
Sláva 3. krát
```

Všimnime si, že premenná (`pocitadlo`) uvedená v cykle za slovom `for`, nadobúda v jednotlivých iteráciách cyklu postupne hodnoty prvkov danej sekvencie (`range(1, 4)`).

Cykly môžeme do seba vzájomne vnárať. Túto možnosť však využívajme opatrne, pretože vnáranie cyklov znižuje prehľadnosť a zrozumiteľnosť programu. Ak je to možné, vnútorný cyklus umiestnime do funkcie. Porovnajme dve rôzne riešenia toho istého problému, vykreslenie nasledovnej vrtuľky:



vnorené cykly

```
def vrtulka():
    for i in range(16):
        for j in range(4):
            pero.forward(100)
            pero.left(90)
        pero.left(360 / 16)

vrtulka()
```

funkcia v cykle

```
def stvorec():
    for j in range(4):
        pero.forward(100)
        pero.left(90)

def vrtulka():
    for i in range(16):
        stvorec()
        pero.left(360 / 16)

vrtulka()
```

Funkcia range()

Funkcia `range()` slúži na generovanie hodnôt aritmetickej postupnosti. Najčastejšie ju zrejme využime v príkaze cyklu `for`. Funkciu `range()` môžeme použiť rôznymi spôsobmi:

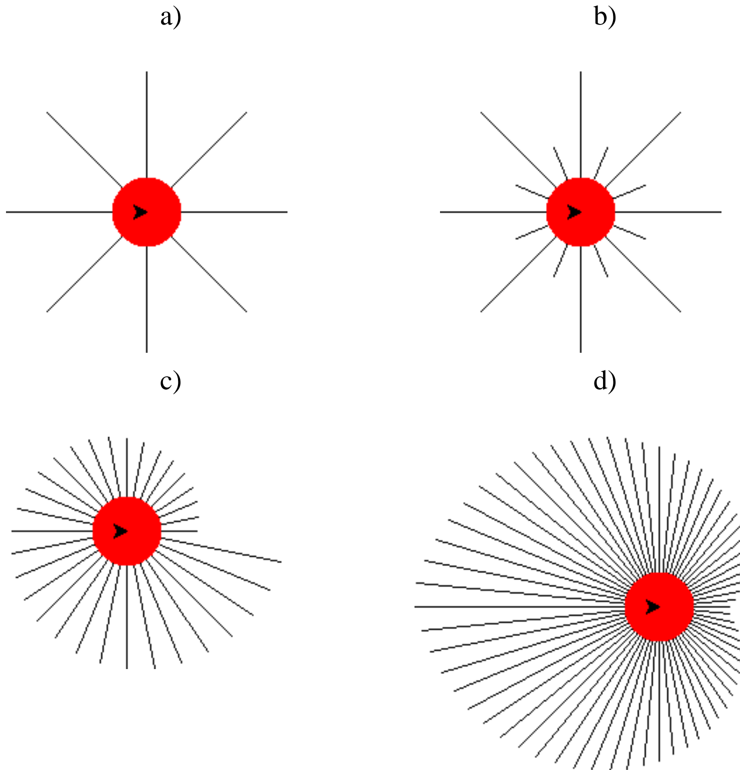
`range(stop)` hodnota `start` je nastavená na 0 hodnota `krok` je nastavená na 1	napr: `range(5)` → 0, 1, 2, 3, 4
`range(start, stop)` hodnota `krok` je nastavená na 1	napr: `range(5, 10)` → 5, 6, 7, 8, 9
`range(start, stop, krok)`	napr: `range(9, 1, -2)` → 9, 7, 5, 3

Všimnime si jeden dôležitý detail. Hodnota `stop` sa vo výslednej sekvencii nikdy nenachádza. Hodnota `krok` môže byť aj záporná. Výsledkom je klesajúca aritmetická postupnosť hodnôt. Každá zo zadaných hodnôt musí byť celočíselná a navyše krok nesmie mať hodnotu 0.

Zbierka úloh

Nasledujúce úlohy sú zamerané na dekompozíciu a hľadanie vzorov. Pokúste sa identifikovať vzory v obrázkoch alebo vo výpočtoch a navrhnúť vhodné funkcie. Pri niektorých úlohách sa predpokladá aj viacúrovňová dekompozícia.

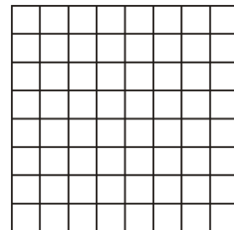
1. Vytvorte funkcie `slniecko()`/`kvietok()`/`ulita()`/`listok()`, ktoré vykreslia obrázky nasledovných typov. Uvažujte aj rôzne počty a dĺžky „lúčov“.



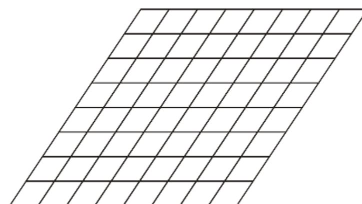
2. Vytvorte program, ktorý vypočíta a vráti faktoriál zadaného prirodzeného čísla.

   Napr.: faktoriál čísla 5 = 5! = 5 * 4 * 3 * 2 * 1

3. Vytvorte funkciu `mriezka()` pre vykreslenie mriežky 8x8:



4. Ak zafúka vietor, mriežka sa mierne ohne. Vytvorte program pre kreslenie takejto mriežky:

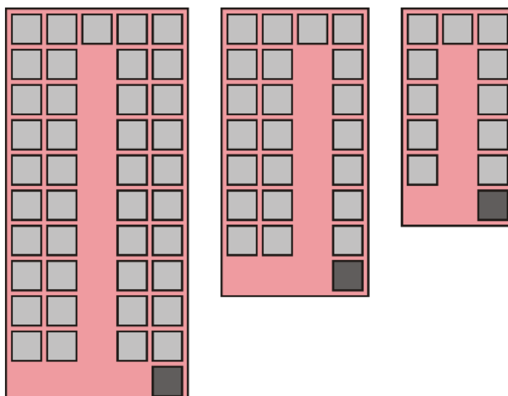


5. Rosnička Martina si vymyslela zaujímavý spôsob skákania po číselnej osi Na začiatku sa postavila na číslo 0. O hodinu skočila jeden skok vpravo a ocitl

sa na čísle 1. O ďalšiu hodinu skočila dva skoky vľavo a pristála na čísle -1 O ďalšiu hodinu skočila tri skoky vpravo a ostala stáť na čísle 2. Týmto spôsobom pokračovala ďalej.

Vytvorte program, ktorý vypočíta a vypíše, na akej pozícii bude rosnička na konci zadanej hodiny.

6. V softvérovej firme, ktorá vyvíja rezervačný systém pre autobusy, potrebujú program pre zobrazenie rozmiestnenia sedadiel v autobuse. Vytvorte funkcie ktoré budú vykresľovať rozmiestnenie sedadiel podobne, ako je na obrázku:



Pomôcka: Všimnite si opakujúce sa časti pre jednotlivé autobusy.

7. Vytvorte program pre vypísanie malej násobilky. Napr. pre hodnotu 5 program vypíše:

```
1 x 1 = 1     1 x 2 = 2     1 x 3 = 3     1 x 4 = 4     1 x 5 = 5
2 x 1 = 2     2 x 2 = 4     2 x 3 = 6     2 x 4 = 8     2 x 5 = 10
3 x 1 = 3     3 x 2 = 6     3 x 3 = 9     3 x 4 = 12    3 x 5 = 15
4 x 1 = 4     4 x 2 = 8     4 x 3 = 12    4 x 4 = 16    4 x 5 = 20
5 x 1 = 5     5 x 2 = 10    5 x 3 = 15    5 x 4 = 20    5 x 5 = 25
```

Pomôcka. Ak potrebujete formátovať výstup, môžete použiť nasledovné:

`print('nieco1\tnieco2')`	*oddelí text tabulátorom:* `nieco1        nieco2`
`print('nieco1', end='')` `print('nieco2')`	*zabezpečí, že nasledujúci výpis bude pokračovať v tom istom riadku:* `nieco1nieco2`

8. Vytvorte program, ktorý vypíše 100-krát text:

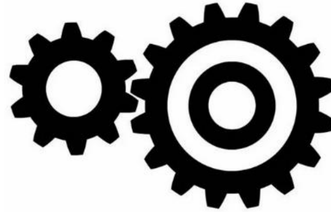
1. Programovanie v Python-e je super.

2. Programovanie v Python-e je super.

...

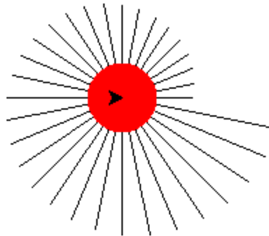
100. Programovanie v Python-e je super.

9. Máme dve ozubené kolieska s počtom zubov `pocet1` a `pocet2`, ktoré pri otáčaní do seba zapadajú. Vytvorte funkciu `pis_dotyky()`, ktorá vypíše, ktorý zub prvého kolieska do ktorej medzierky druhého kolieska bude zapadať, ak kolieskami budeme otáčať.



Riešené úlohy

1.c Vytvorte funkciu `ulita()`, ktorá vykreslí nasledovný typ obrázka. Uvažujte aj rôzne počty a dĺžky „lúčov“.



Riešenie:

Pri riešení môžeme použiť stratégiu *Vyrieš jednoduchší problém* a vykresliť rovnakú ulitu ako na obrázku. Ulita má 32 lúčov a ich dĺžka sa postupne zväčšuje. Ak si určíme, že najkratší lúč má dĺžku napr. 100 a rozdiel dvoch susedných je 3, najdlhší lúč bude mať 100 + 31 * 3. Červený kruh vo vnútri má polomer rovný polovici najmenšieho lúča. Funkciu `ulita()` definujeme nasledovne:

```
def ulita():
    for dlzka in range(100, 100 + 32 * 3, 3):
        pero.forward(dlzka)
        pero.backward(dlzka)
        pero.left(360 / 32)
    pero.color('red')
    pero.dot(100)
```

Vo funkcii `range()` sme ponechali výraz (`100 + 32 * 3`). Konkrétna hodnota by bola po čase už nejasná. Navyše, zápis výrazu nám môže pomôcť pri neskoršom zovšeobecnení úlohy. Ak nás prekvapí hodnota `32` v danom výraze, pretože v analýze úlohy sme v dĺžke posledného lúča použili hodnotu 31, stačí si uvedomiť, že hodnota stop sa do výslednej sekvencie už nedostane.

Pozrime sa teraz na všeobecnejšie riešenie, v ktorom si určíme nielen počet lúčov, ale aj ich dĺžku a rozdiel.

Uvažujme nasledovné parametre ulity: dĺžka najkratšieho lúča, rozdiel dĺžok susedných lúčov a počet lúčov. Všeobecnejšie riešenie tejto úlohy vyzerá nasledovne:

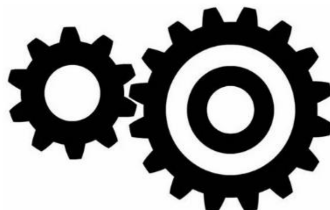
```
def ulita():
    dlzka_min = 100
    pocet = 32
    rozdiel = 3
    dlzka_max = dlzka_min + pocet * rozdiel
    for dlzka in range(dlzka_min, dlzka_max, rozdiel):
        pero.forward(dlzka)
        pero.backward(dlzka)
        pero.left(360 / pocet)
    pero.color('red')
    pero.dot(dlzka_min)
```

Pri overovaní správnosti riešenia je rozumné na začiatok použiť rovnaké hodnoty ako v predchádzajúcej verzii. Nová verzia by mala vykresliť rovnaký obrázok. Až potom môžeme experimentovať aj s inými hodnotami.

V riešení uvažujeme len celočíselné hodnoty dĺžok a rozdielov dĺžok lúčov. Vo všeobecnosti to môžu byť aj nie celé hodnoty. Problémom je, že funkcia `range()` akceptuje len celočíselné hodnoty. Výpočet hodnôt dĺžok lúčov preto musíme „presunúť" mimo funkciu `range()`. V tomto prípade riešenie môže vyzerať nasledovne:

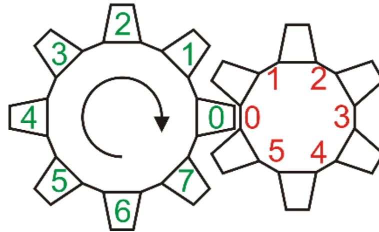
```
def ulita():
    dlzka_min = 100
    pocet = 32
    rozdiel = 3
    for pocitadlo in range(pocet):
        dlzka = dlzka_min + pocitadlo * rozdiel
        pero.forward(dlzka)
        pero.backward(dlzka)
        pero.left(360 / pocet)
    pero.color('red')
    pero.dot(dlzka_min)
```

9. Máme dve ozubené kolieska s počtom zubov `pocet1` a `pocet2`, ktoré pri otáčaní do seba zapadajú. Vytvorte funkciu `pis_dotyky()`, ktorá vypíše, ktorý zub prvého kolieska do ktorej medzierky druhého kolieska bude zapadať, ak kolieskami budeme otáčať.



Riešenie:

Pri riešení úlohy môžeme postupovať tak, že **zuby** aj **medzierky** si vzájomne očíslujeme a počiatočnú pozíciu nastavíme tak, že **zub 0** zapadá do **medzierky 0**.



Spravme si zoznam dvojíc **zub** – **medzera**, ktoré sa pri otáčaní budú dotýkať.

dotyk	**zub**	**medzera**
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	0
7	7	1
8	0	2
9	1	3
10	2	4
11	3	5
...	...	...

Vidíme, že čísla **zubov** a **medzier** sa cyklicky opakujú. Pri bližšom pohľade zistíme, že:

```
zub = dotyk % počet1
```

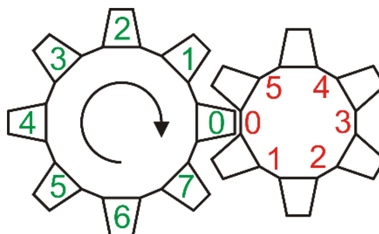
```
medzera = dotyk % počet2
```

Teraz už nie je problém program vytvoriť.

```
def pis_zub_medzera():
    pocet1 = 8
    pocet2 = 6
    pocet_zapadnuti = 12
    for dotyk in range(pocet_zapadnuti):
        zub = dotyk % pocet1
        medzera = dotyk % pocet2
        print(f'zub: {zub} - medzera: {medzera}')
```

Isto ste si všimli, že riešenie sme si uľahčili tým, že kolieska sme očíslovali vo vzájomne opačnom poradí. Pri vzájomnom otáčaní sa totiž otáčajú

v opačnom smere. Pozrime sa na riešenie, ak by sme číslovanie robili na každom koliesku v rovnakom smere.



V tomto prípade budú dotyky nasledovné:

dotyk	**zub**	**medzera**
0	0	0
1	1	5
2	2	4
3	3	3
4	4	2
5	5	1
6	6	0
7	7	5
8	0	4
9	1	3
10	2	2
11	3	1
...	...	...

Čísla **zubov** sa nezmenili. Zmenili sa čísla **medzier**, ktoré sa opakujú v opačnom poradí. Na prvý pohľad komplikovaný vzťah medzi číslom **dotyku** a číslom **medzery** vieme popísať jednoducho tak, že **dotyky** budeme uvádzať záporné.

```
def pis_zub_medzera():
    pocet1 = 8
    pocet2 = 6
    pocet_zapadnuti = 12
    for i in range(pocet_zapadnuti):
        zub = i % pocet1
        medzera = -i % pocet2
        print(f'zub: {zub} - medzera: {medzera}')
```

4 Funkcie s parametrami a návratovou hodnotou

Jemný úvod do problematiky

Použite notebook **Python 4 - Funkcie s parametrami a návratovou hodnotou.ipynb** alebo niektoré z lokálnych vývojových prostredí jazyka Python.

Cieľ: *V tomto notebooku si ukážeme ako definovať a používať funkcie všeobecnejšie, pomocou parametrov. Naučíme sa vytvárať funkcie pre výpočet, použitím návratovej hodnoty. Budeme rozvíjať stratégie riešenia problémov: dekompozícia a nájdi vzor.*

Funkcie s parametrami

V predchádzajúcich častiach sme použili funkcie v zmysle pomenovania skupiny príkazov. Vždy, keď sme takto definovanú funkciu zavolali, vykonala sa tá istá postupnosť príkazov. Vráťme sa opäť k príkladu s 5 šípkami:

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

def sipka():
    pero.forward(30)
    pero.right(30)
    pero.back(20)
    pero.forward(20)
    pero.left(60)
    pero.back(20)
    pero.forward(20)
    pero.right(30)
    pero.back(30)

def posun():
    pero.penup()
    pero.right(90)
    pero.forward(30)
    pero.left(90)
    pero.pendown()

def sipka_s_posunom():
    sipka()
    posun()

for pocitadlo in range(5):
    sipka_s_posunom()

tabula.mainloop()
```

Upravme programový kód tak, aby sme mohli kresliť ľubovoľne veľké šípky. Konkrétne hodnoty vo funkcii `sipka()`, ktoré predstavovali dĺžky musíme nahradiť premennými. Zároveň musíme funkciu upraviť tak, aby očakávala, že hodnotu veľkosti šípky jej pri volaní pošleme:

```
def sipka(velkost):
    pero.forward(velkost)
    pero.right(30)
    pero.back(2 * velkost / 3)
    pero.forward(2 * velkost / 3)
    pero.left(60)
    pero.back(2 * velkost / 3)
    pero.forward(2 * velkost / 3)
    pero.right(30)
    pero.back(velkost)
```

Funkciu `sipka()` sme upravili tak, že vieme pomocou nej kresliť ľubovoľne veľké šípky. Ak ale zmeníme veľkosť šípky, mali by sme zmeniť aj vzdialenosť medzi nimi. Upravme preto aj funkciu `posun()`:

```
def posun(vdialenost):
    pero.penup()
    pero.right(90)
    pero.forward(vdialenost)
    pero.left(90)
    pero.pendown()
```

Funkcie `sipka()` a `posun()` voláme z funkcie `sipka_s_posunom()`. Aj túto funkciu musíme upraviť.

```
def sipka_s_posunom(velkost):
    sipka(velkost)
    posun(velkost)
```

Nakresliť 5 šípok teda môžeme nasledovne:

```
for pocitadlo in range(5):
    sipka_s_posunom(30)
```

V pôvodnom zadaní sme chceli nakresliť 5 šípok. Zovšeobecnime toto zadanie. Kreslime zadaný počet šípok zadanej veľkosti. Vytvorme si funkciu `kresli_sipky()`, ktorá sa postará o vykreslenie zadaného počtu šípok zadanej veľkosti:

```
def kresli_sipky(pocet, velkost):
    for pocitadlo in range(pocet):
        sipka_s_posunom(velkost)
```

Ukážme si, ako celý programový kód vyzerá:

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

def sipka(velkost):
    pero.forward(velkost)
    pero.right(30)
    pero.back(2 * velkost / 3)
    pero.forward(2 * velkost / 3)
    pero.left(60)
    pero.back(2 * velkost / 3)
    pero.forward(2 * velkost / 3)
    pero.right(30)
    pero.back(velkost)

def posun(vdialenost):
    pero.penup()
    pero.right(90)
    pero.forward(vdialenost)
    pero.left(90)
    pero.pendown()

def sipka_s_posunom(velkost):
    sipka(velkost)
    posun(velkost)

def kresli_sipky(pocet, velkost):
    for pocitadlo in range(pocet):
        sipka_s_posunom(velkost)

kresli_sipky(6, 20)
```

Všimnime si dve dôležité veci:

- použitím parametrov sa naše riešenie zovšeobecnilo, vieme kresliť ľubovoľný počet šípok ľubovoľnej veľkosti,
- vďaka dekompozícii je výsledný program prehľadný, čitateľný a ľahko testovateľný.

Funkcie s návratovou hodnotou

Niekedy je výhodné mať funkciu, ktorá niečo vypočíta. V tomto prípade musíme vyriešiť, čo má funkcia s výsledkom spraviť. Nemusí ho vypísať alebo nakresliť. Môže ho vrátiť. Napr.:

```
def pocitaj_bmi(hmotnost, vyska):
    vysledok = hmotnost / vyska ** 2
    return vysledok
vyska = 1.9
```

```
hmotnost = 90
bmi = pocitaj_bmi(hmotnost, vyska)
print(bmi)
```

Výraz `return hodnota` vo funkcii `pocitaj_bmi()` spôsobí zastavenie vykonávania funkcie a vrátenie hodnoty `vysledok` na miesto, odkiaľ bola funkcia volaná. S touto hodnotou môžeme ďalej v programe pracovať. Vo vyššie uvedenom príklade sme ju pomenovali názvom `bmi` a následne vypísali.

Všimnime si našu cestu svetom funkcií:

- Začali sme funkciami na kreslenie, ktoré vždy kreslili rovnaký obrázok. Výsledok funkcie sme už ale nevedeli ďalej v programe použiť (toto je v poriadku, ak cieľom funkcie je, aby niečo nakreslila).
- Funkciám sme pridali parametre. Vďaka tomu sme vedeli ovplyvniť to, čo funkcia robí. Ani tu sme ale nevedeli výsledok použiť v programe ďalej (aj tu je to v poriadku, ak cieľom funkcie je niečo vykresliť).
- Funkciám sme pridali návratovú hodnotu. Vedeli sme ovplyvniť čo funkcia robí a výslednú hodnotu sme vedeli v programe ďalej použiť. Toto riešenie je zo všetkých najuniverzálnejšie a v prípade funkcií na výpočet ho budeme preferovať.

Cvičenie 1

Definujte funkciu `obsah()`, ktorá pre zadanú dĺžku strany štvorca vráti jeho obsah.

Cvičenie 2

Definujte funkciu `objem_bazena()`, ktorá pre zadané rozmery bazéna vráti jeho objem.

Cvičenie 3

Definujte funkciu `doba_plnenia_bazena()`, ktorá pre zadané rozmery bazéna a výkon čerpadla (objem prečerpanej vody za jednotku času) vráti, za akú dobu sa bazén naplní.

Pomôcka: Využite funkciu `objem_bazena()` z predchádzajúceho cvičenia.

Vysvetlenie problematiky

Funkcie s parametrami

Funkcia je vo všeobecnosti samostatný blok programového kódu, ktorý rieši konkrétny problém alebo niekoľko súvisiacich problémov. Ak tento blok programového kódu spracováva nejaké hodnoty, ktoré môžu byť pre rôzne inštancie problému iné, odovzdáme funkcii konkrétne hodnoty v parametroch. Parametre uvedieme v hlavičke funkcie pri jej definícii: `def funkcia(parameter1, parameter2, ...)`. Konkrétne hodnoty pošleme funkcii pri jej volaní pomocou argumentov: `funkcia(hodnota1, hodnota2, ...)`.

Funkcia `pozdrav()` vypíše pozdrav pre konkrétneho človeka:

```
def pozdrav(meno_cloveka):
    print(f'Ahoj {meno_cloveka}, prajem Ti pekný deň')

pozdrav('Kamil')
```

```
Ahoj Kamil, prajem Ti pekný deň
```

Počet a význam argumentov pri volaní funkcie musí zodpovedať počtu a významu parametrov v definícii funkcie. Funkcia `pis_bmi()` vypočíta a vypíše hodnotu BMI. Pozrime sa, aké dôsledky môže mať porušenie týchto pravidiel.

```
# BMI = hmotnost (kg) / vyska (m) na druhú
def pis_bmi(hmotnost, vyska):  # [1]
    bmi = hmotnost / vyska ** 2
    print(bmi)

hmotnost_cloveka = 80
vyska_cloveka = 1.75

pis_bmi(hmotnost_cloveka, vyska_cloveka)   # [2]
pis_bmi(vyska_cloveka, hmotnost_cloveka)   # [3]
pis_bmi(hmotnost_cloveka)   # [4]
```

```
26.122448979591837
0.0002734375

Traceback (most recent call last):
  File "...", line 11, in <module>
    pis_bmi(hmotnost_cloveka)
TypeError: pis_bmi() missing 1 required positional argument:
'vyska'
```

[1] V definícii funkcie sme uviedli dva parametre `hmotnost`, `vyska` v tomto význame a v tomto poradí.

[2] Pri volaní funkcie sme použili dva argumenty `hmotnost_cloveka` a `vyska_cloveka`, ktoré v tomto počte a poradí **zodpovedajú** tomu, ako je definovaná funkcia. Vypísaná hodnota **zodpovedá** hodnote BMI daného človeka.

[3] Pri volaní funkcie sme použili dva argumenty `vyska_cloveka` a `hmotnost_cloveka`, ktoré v tomto poradí **nezodpovedajú** tomu, ako je definovaná funkcia. Vypísaná hodnota **nezodpovedá** hodnote BMI daného človeka a predstavuje nezmyselnú hodnotu.

[4] Pri volaní funkcie sme použili len **jeden argument** `vyska_cloveka`. Volanie funkcie v tomto prípade skončí s **chybou** a program predčasne skončí.

Funkcie s návratovou hodnotou

Vyššie definovaná funkcia `pis_bmi()` vypočíta a vypíše hodnotu BMI. Problém je, že k tejto hodnote už ďalej v programe nemáme prístup. Výhodnejšie by bolo, keby funkcia túto hodnotu nie len vypočítala, ale aj vrátila pre ďalšie použitie.

```
def pocitaj_bmi(hmotnost, vyska):
    bmi = hmotnost / vyska ** 2
    return bmi   # [1]

hmotnost_cloveka = 80
vyska_cloveka = 1.75

bmi = pocitaj_bmi(hmotnost_cloveka, vyska_cloveka)   # [2]
print(f'Hodnota BMI je: {bmi}')   # [3]
```

```
Hodnota BMI je: 26.122448979591837
```

[1] Namiesto výpisu hodnoty (`print`) funkcia výsledok výpočtu vráti (`return`). Príkaz `return` ukončí vykonávanie funkcie a uvedenú hodnotu vráti ako výsledok funkcie. Príkazov `return` môže byť vo funkcii viac. Prvý z nich ktorý sa vykoná, zastaví vykonávanie funkcie a funkcia vráti hodnotu uvedenú v príkaze `return`.

[2] Vrátenú hodnotu sme si v hlavnom programe pomenovali menom `bmi`.

[3] Hodnotu BMI vieme použiť, napr. vo výpise pre používateľa.

Funkcie predstavujú silný nástroj, pomocou ktorého vieme implementovať riešenie jednotlivých podproblémov a celkovo tak sprehľadniť riešenie celého problému. Vďaka parametrom môžu funkcie pracovať s rôznymi hodnotami. Ak funkcia výsledok svojho výpočtu vráti, vieme túto hodnotu použiť aj v ďalšej časti programového kódu.

Pri riešení väčšieho problému, ktorý dekomponujeme na niekoľko menších funkcií, môže riešenie pozostávať z viacerých funkcií, ktoré sa vzájomne volajú a odovzdávajú si hodnoty.

```
import math

def plocha_obdlznika(strana1, strana2):
    plocha = strana1 * strana2
    return plocha

def plocha_stien_plafon(sirka, dlzka, vyska):
    strop = plocha_obdlznika(sirka, dlzka)
    stena1 = plocha_obdlznika(sirka, vyska)
    stena2 = plocha_obdlznika(dlzka, vyska)
    plocha = strop + 2 * stena1 + 2 * stena2
    return plocha

def pocet_plechoviek(sirka, dlzka, vyska, vydatnost):
    plocha = plocha_stien_plafon(sirka, dlzka, vyska)
    mnozstvo = plocha / vydatnost
    mnozstvo = math.ceil(mnozstvo)
    return mnozstvo

sirka = input('Zadaj šírku miestnosti v metroch: ')
sirka = float(sirka)
dlzka = input('Zadaj dĺžku miestnosti v metroch: ')
dlzka = float(dlzka)
vyska = input('Zadaj výšku miestnosti v metroch: ')
vyska = float(vyska)
vydatnost = input('Zadaj výdatnosť plechovky v m2: ')
vydatnost = float(vydatnost)

pocet = pocet_plechoviek(sirka, dlzka, vyska, vydatnost)
print(f'Počet potrebných plechoviek: {pocet}')
```

```
Zadaj šírku miestnosti v metroch: 5
Zadaj dĺžku miestnosti v metroch: 8
Zadaj výšku miestnosti v metroch: 2.5
Zadaj výdatnosť plechovky v m2: 4.8
Počet potrebných plechoviek: 22
```

Problém „Koľko plechoviek farby potrebujeme na vymaľovanie miestnosti“ sme dekomponovali na menšie podproblémy: výpočet plochy stien a plafóna a výpočet plochy obdĺžnika. Riešenie každého z podproblémov sme implementovali do samostatných funkcií. Každá z funkcií pre zadané hodnoty parametrov vypočíta požadovanú hodnotu, ktorú vráti ako výsledok.

Všimnime si pomenovanie hodnôt na rôznych miestach programu. Aj keď na viacerých miestach programu používame tú istú hodnotu (napr. šírku miestnosti), pristupujeme k tejto hodnote cez rôzne mená. V hlavnom programe sme túto hodnotu pomenovali globálnym názvom `sirka`. Vo funkciách `pocet_plechoviek()` a `plocha_stien_plafon()` sme túto hodnotu

pomenovali rovnakým, ale lokálnym menom `sirka`. Vo funkcii `plocha_obdlznika()` sme túto hodnotu pomenovali `strana1`. Každá z funkcií si teda vytvára svoje vlastné pomenovanie hodnôt s ktorými pracuje a ktoré dáva zmysel v kontexte danej funkcie.

Ukážme si ešte jedno, nesprávne použitie funkcií. Demonštrujeme ho na príklade funkcie, ktorá prepočíta dĺžku v palcoch na centimetre:

funkcia s parametrom

```
def palce_na_cm(palce):
    cm = palce * 2.54
    return cm

palce = 21
cm = palce_na_cm(palce)
print(cm)
```

```
53.34
```

funkcia bez parametra

```
def palce_na_cm():
    cm = palce * 2.54
    return cm

palce = 21
cm = palce_na_cm()
print(cm)
```

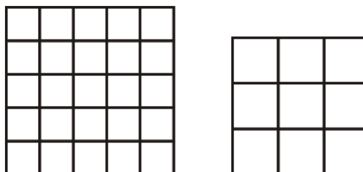
```
53.34
```

Na počudovanie, obidve verzie fungujú a zobrazujú správne výsledky. Problémom druhého riešenia je, že funkcia sa spolieha na kontext, v ktorom sa nachádza – presnejšie na existenciu číselnej premennej `palce` v hlavnom programe. Ak by sme funkciu preniesli na iné miesto (napr. do iného programu), už nemusí fungovať správne. Museli by sme zároveň preniesť aj globálnu premennú `palce`. Funkcie budeme definovať tak, aby všetky premenlivé hodnoty, s ktorými potrebujú pracovať dostali v parametroch.

Zbierka úloh

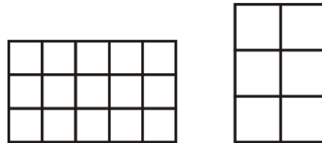
Nasledujúce úlohy sú zamerané na dekompozíciu a hľadanie vzorov. Dekompozícia by sa mala premietnuť do návrhu funkcií s vhodnými parametrami, ktoré v prípade potreby vracajú požadované výstupné hodnoty. Pri niektorých problémoch sa predpokladá aj viacúrovňová dekompozícia. Okrem požadovanej funkcie je vhodné si vytvoriť aj ďalšie funkcie, riešiace menšie podproblémy.

1. Vytvorte funkciu `mriezka1()`, pomocou ktorej budeme vedieť kresliť štvorcovú mriežku so zadaným **počtom políčok** v riadku a so zadanou **veľkosťou políčka**, napr.:

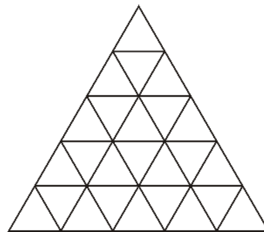


2. Vytvorte funkciu `mriežka2()`, pomocou ktorej budeme vedieť kresliť štvorcovú mriežku so zadaným **počtom políčok** v riadku a celkovou **veľkosťou mriežky**.

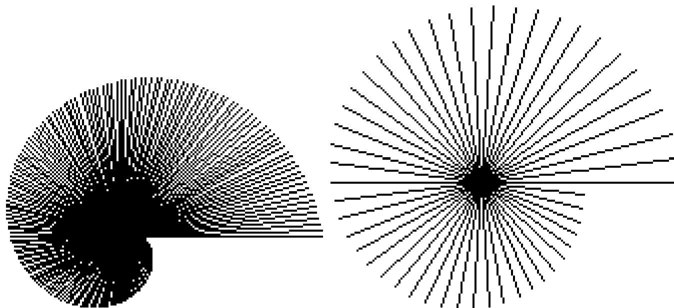
3. Vytvorte funkciu `mriežka3()`, pomocou ktorej budeme vedieť kresliť mriežku so zadaným **počtom políčok v riadku**, so zadaným **počtom políčok v stĺpci** a so zadanou **veľkosťou políčka**, napr.:



4. Vytvorte funkciu `pyramida()`, ktorá pre zadanú dĺžku strany malého trojuholníka a počet poschodí vykreslí pyramídu typu:



5. Vytvorte funkciu `ulita()` s vhodnými parametrami, pomocou ktorej vykreslíme obrázky nasledovného typu:



Pomôcka: Analyzujte obrázok. Pomocou ktorých parametrov by ste ho vedeli popísať?

6. Vytvorte funkciu `priemerna_rychlost()`, ktorá pre zadanú dĺžku trasy a čas jej prejdenia vypočíta a vráti priemernú rýchlosť.

7. Vytvorte funkciu `cas_plnenia_suda()`, ktorá pre vhodné zadané hodnoty vypočíta a vráti, ako rýchlo sa sud naplní vodou. Uvažujte sud tvaru valca.

   Pomôcka: Uvažujte, aké hodnoty je vhodné funkcii posielať.

8. Vytvorte funkciu `cena_po_zlave()`, ktorá vráti výslednú cenu výrobku po uplatnení zadanej percentuálnej zľavy.

9. Vytvorte funkciu `cena_za_taxi()`, ktorá vypočíta a vráti cenu za odvoz taxíkom.

   Z cenníka taxislužby:

   - štartovné – 1,00 €,
   - cena za km – 0,90 €,
   - minimálne jazdné – 3,00 €.

   Pomôcka: Na zistenie najmenšej, resp. najväčšej hodnoty má Python funkcie `min()`*, resp.* `max()`*.*

   Viac informácií nájdete na: https://docs.python.org/3/library/functions.html.

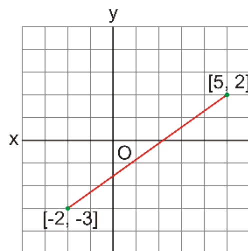
10. Vytvorte funkciu `obsah_medzikruzia()`, ktorá vypočíta a vráti obsah medzikružia. Uvažujte vhodné parametre pre zadanie medzikružia.

11. Vytvorte funkciu `hmotnost_snehuliaka()`, ktorá pre zadanú výšku snehuliaka a hustotu snehu vráti hmotnosť snehuliaka.

    Telo snehuliaka staviame z troch gúľ tak, že priemer vrchnej z dvoch susedných gúľ je 2/3 priemeru spodnej gule. Ruky snehuliaka majú polovičnú veľkosť ako hlava.

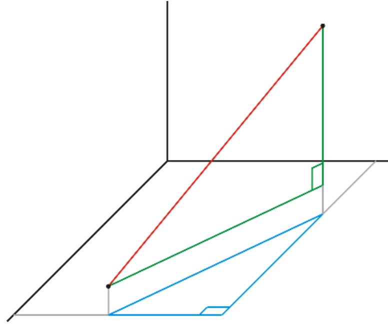
12. Budeme skladať hárok papiera. Zložíme ho na polovicu. Potom ešte raz na polovicu atď. Ak by sme uvažovali čisto teoreticky, aká bude výška (súčet hrúbok listov papiera v skladačke) papierovej skladačky po n zloženiach? Vytvorte funkciu `vyska_skladacky()`, ktorá túto hodnotu vypočíta a vráti.

13. Vytvorte funkciu `vzdialenost_2d()`, ktorá vypočíta a vráti vzdialenosť dvoch bodov v rovine.

    Pomôcka: Pytagorova veta.

14. Vytvorte `funkciu vzdialenost_3d()`, ktorá vypočíta a vráti vzdialenosť dvoch bodov v priestore.



Pomôcka: Pytagorova veta a riešenie predchádzajúcej úlohy.

Riešené úlohy

2. Vytvorte funkciu `mriezka2()`, pomocou ktorej budeme vedieť kresliť štvorcovú mriežku so zadaným **počtom políčok** v riadku a celkovou **veľkosťou mriežky**.

   Riešenie:

   Kreslenie mriežky sme čiastočne riešili v kapitole 2. Teraz ale poznáme väčšie množstvo nástrojov, vďaka čomu môže byť riešenie jednoduchšie a univerzálnejšie zároveň.

   Už samotné zadanie úlohy napovedá, v čom by sa mali jednotlivé mriežky líšiť:

   - počtom políčok,
   - veľkosťou mriežky.

   Ak chceme, aby funkcia `mriezka2()` vedela kresliť mriežky líšiace sa v týchto dvoch parametroch, musíme jej konkrétne hodnoty poslať. Využijeme na to parametre funkcie.

   Druhým krokom je samotná dekompozícia problému kreslenia mriežky. Jedna z možných dekompozícií je: mriežka → rad štvorcov → štvorec.

   Hodnotami počet políčok a veľkosť mriežky budú ovplyvnené nie len mriežka, ale aj riadok mriežky a štvorček v riadku:

```
mriezka2(pocet_stvorcov, strana_mriezky)
↓
rad(pocet_stvorcov, strana_stvorca)
↓
stvorec(strana)
```

   Funkcie na vyšších úrovniach preto musia poslať relevantné hodnoty funkciám na nižších úrovniach.

Dekompozíciu sme spravili zhora od najväčšieho problému smerom nadol k najmenšiemu problému. Samotnú implementáciu je výhodné robiť opačne, zdola smerom nahor. Dôvod je jednoduchý. Kreslenie štvorca vieme otestovať aj bez toho, aby existovala funkcia `mriezka2()`. Naopak to možné nie je.

```
def stvorec(strana):
    for i in range(4):
        pero.forward(strana)
        pero.left(90)

def rad(pocet_stvorcov, strana_stvorca):
    for i in range(pocet_stvorcov):
        stvorec(strana_stvorca)
        pero.forward(strana_stvorca)
    pero.backward(pocet_stvorcov * strana_stvorca)

def mriezka2(pocet_stvorcov, strana_mriezky):
    strana_stvorca = strana_mriezky / pocet_stvorcov
    for i in range(pocet_stvorcov):
        rad(pocet_stvorcov, strana_stvorca)
        pero.left(90)
        pero.forward(strana_stvorca)
        pero.right(90)
    pero.left(90)
    pero.backward(strana_sachovnice)
    pero.right(90)
```

Ak porovnáme uvedené riešenie s riešením úlohy 3 v kapitole 2, zistíme, že:

- využitie cyklu `for` umožnilo riešenie skrátiť,
- použitie parametrov umožnilo riešenie zovšeobecniť.

11. Vytvorte funkciu `hmotnost_snehuliaka()`, ktorá pre zadanú výšku snehuliaka a hustotu snehu vráti hmotnosť snehuliaka.

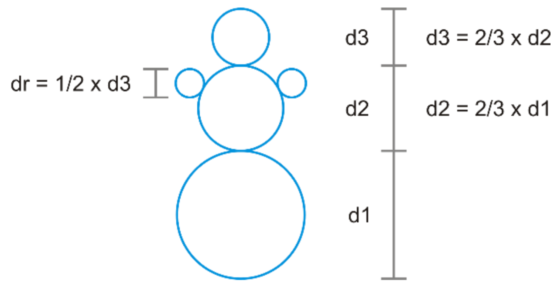
Telo snehuliaka staviame z troch gúľ tak, že priemer vrchnej z dvoch susedných gúľ je 2/3 priemeru spodnej gule. Ruky snehuliaka majú polovičnú veľkosť ako hlava.

Riešenie:

Samotné riešenie rozdeľme na dve časti:

1. zistime rozmery jednotlivých gúľ snehuliaka,
2. dekomponujme výpočet hmotnosti.

(1) Analyzujme rozmery snehuliaka. Výsledkom môže byť takýto náčrt:



Podľa zadania poznáme len výšku snehuliaka, označme ju h. Pre výpočet ostatných rozmerov stačí zistiť priemer spodnej gule.

$h = d1 + d2 + d3 = d1 + \frac{2}{3}d1 + \frac{4}{9}d1$, po úprave: $d1 = \frac{9h}{19}$

(2) Pre výpočet hmotnosti snehuliaka potrebujeme poznať jeho celkový objem. Celkový objem je súčtom objemov jednotlivých gúľ, z ktorých je snehuliak postavený. Ak poznáme celkový objem snehuliaka, vypočítať jeho hmotnosť už nebude problém.

```
def objem_gule(priemer):
    polomer = priemer / 2
    objem = 4 / 3 * math.pi * polomer ** 3
    return objem

def hmotnost_snehuliaka(vyska, hustota_snehu):
    priemer_spodna = 9 * vyska / 19
    priemer_stredna = 2 / 3 * priemer_spodna
    priemer_vrchna = 2 / 3 * priemer_stredna
    priemer_ruka = 1 / 2 * priemer_vrchna

    objem_snehuliaka = objem_gule(priemer_spodna)
    objem_snehuliaka += objem_gule(priemer_stredna)
    objem_snehuliaka += objem_gule(priemer_vrchna)
    objem_snehuliaka += 2 * objem_gule(priemer_ruka)

    hmotnost = hustota_snehu * objem_snehuliaka
    return hmotnost
```

Keďže objem gule počítame viackrát, je výhodné si pre tento výpočet vytvoriť samostatnú funkciu `objem_gule()`. Výpočet jednotlivých rozmerov snehuliaka sme realizovali vo funkcii `hmotnost_snehuliaka()` na základe výsledkov analýzy z časti (1).

Všimnime si skrátený zápis pre „pripočítanie“ hodnoty k objemu snehuliaka `+=`. Zápis `a += b` je pri číselných hodnotách ekvivalentom dlhšieho zápisu `a = a + b`.

5 Podmienky a podmienený príkaz

Jemný úvod do problematiky

Použite notebook **Python 5 - Podmienky a podmienený príkaz.ipynb** alebo niektoré z lokálnych vývojových prostredí jazyka Python.

Cieľ: *V tomto notebooku si ukážeme ako pracovať s logickým výrazom – podmienkou a ako vetviť vykonávanie programu pomocou príkazu vetvenia* `if`.

Logické výrazy

Vyhodnoťte postupne nasledovné výrazy:

```
1 < 2
1 + 2 == 3
1 > 1
1 >= 1
4 <= 5 < 10
1 != 2
```

Vo výrazoch vyššie sme použili operátory porovnania: `<`, `<=`, `>`, `>=`, `==`, `!=`. Výsledkom vyhodnotenia týchto výrazov boli len logické hodnoty `True` – pravda alebo `False` – nepravda. Výraz, ktorého výsledkom je logická hodnota, nazývame logický výraz.

Jednoduché logické výrazy vieme skladať do zložitejších. Využívame na to logické spojky. Vyhodnoťte nasledovné výrazy:

```
1 < 2 and 2 < 3
1 < 2 or 2 < 1
not (1 < 2 or 2 < 1)
```

V logických výrazoch, tak ako aj v iných výrazoch, môžeme používať premenné, napr.:

```
x = 10
0 < x < 20
```

Cvičenie 1

Priraďte do premennej `x` takú hodnotu, aby bol výraz `x <= 2 * x` pravdivý.

Priraďte do premennej `x` takú hodnotu, aby bol výraz `x <= 2 * x` nepravdivý.

Svoje riešenie otestujte.

Podmienený príkaz if

Priebeh vykonávania programu nemusí byť vždy rovnaký. Na základe pravdivosti, resp. nepravdivosti nejakého logického výrazu vieme vykonávanie programu vetviť. Nasledujúci programový kód zisťuje, či zadané číslo je väčšie ako 0 (spustite nasledujúci programový kód).

```
cislo = input('Zadaj číslo')
cislo = float(cislo)

if cislo > 0:
    print('Zadal si číslo väčšie ako 0')

print('Koniec programu')
```

Pomocou príkazu `if` vykonávanie programu môže prebiehať dvoma spôsobmi. Program vypíše informáciu, že číslo je väčšie ako nula alebo nevypíše nič. Všimnime si dvojbodku za logickým výrazom a následné odsadenie programového kódu.

Príkaz `if` má viacero tvarov (spustite nasledujúci programový kód a zadávajte rôzne vstupy):

```
cislo = input('Zadaj číslo')
cislo = float(cislo)

if cislo > 0:
    print('Zadal si číslo väčšie ako 0')
else:
    print('Zadal si číslo menšie alebo rovné 0')

print('Koniec programu')
```

alebo

```
cislo = input('Zadaj číslo')
cislo = float(cislo)

if cislo > 0:
    print('Zadal si číslo väčšie ako 0')
elif cislo < 0:
    print('Zadal si číslo menšie ako 0')
else:
    print('Zadal si číslo rovné 0')

print('Koniec programu')
```

Príkaz `if` môže mať niekoľko častí `elif` (alebo aj žiadnu). Ak je splnená podmienka vo viacerých vetvách `elif`, vykoná sa tá časť, ktorá je prvá za splnenou

podmienkou. Príkaz `if` môže, ale nemusí mať aj časť `else`, ktorá sa vykoná, ak žiaden z predchádzajúcich logických výrazov nebol pravdivý.

Zložené podmienky môžeme vytvárať nielen z operátorov porovnania a logických spojok. Súčasťou podmienok môžu byť aj funkcie, ktorých návratová hodnota je logického typu.

Nasledujúci program testuje, či zadané celé číslo je párne a nezáporné zároveň.

```
def je_parne(cislo):
    return cislo % 2 == 0

def je_zaporne(cislo):
    return cislo < 0

cislo = input('Zadaj celé číslo: ')
cislo = int(cislo)

if je_parne(cislo) and not je_zaporne(cislo):
    print(f'Zadané číslo párne a nezáporné.')
else:
    print(f'Zadané číslo je nepárne alebo záporné.')
```

Cvičenie 2

Napíšte programový kód, ktorý pre zadané celé číslo zistí, či na mieste jednotiek je 0.

Cvičenie 3

Cukríky môžeme baliť dvoma spôsobmi: 20 cukríkov do vrecúška a 12 vrecúšok do krabice alebo 25 cukríkov do vrecúška a 10 vrecúšok do krabice. Napíšte programový kód, ktorý si od používateľa vypýta počet cukríkov a vypíše, ktorý spôsob balenia je výhodnejší (cukríky, ktoré nezaplnia celú krabicu sa vyhodia do odpadu).

Upravte programový kód tak, aby v prípade, že obidva spôsoby sú rovnako výhodné, túto informáciu vypísal.

Cvičenie 4

Prirodzené číslo nazveme dokonalým, ak sa rovná súčtu svojich vlastných deliteľov okrem seba samého. Dokonalým číslom je napr. 6, lebo 6 = 1 + 2 + 3.

Vytvorte program ktorý zistí, či zadané prirodzené číslo je dokonalé alebo nie.

Vysvetlenie problematiky

Logické výrazy

Logická hodnota je jedna z dvoch hodnôt: `True` (pravda) a `False` (nepravda). Obidve hodnoty sú typu `bool` (z angl. boolean).

Logický výraz je výraz, ktorého hodnota je `True` alebo `False`. Jednoduché logické výrazy často obsahujú operátory porovnania. Porovnávané hodnoty nemusia byť (a zväčša ani nie sú) typu `bool`.

`x == y`	výsledkom je `True`, ak sa hodnoty `x` a `y` rovnajú, inak `False`
`x != y`	výsledkom je `True`, ak sa hodnoty `x` a `y` nerovnajú, inak `False`
`x > y`	výsledkom je `True`, ak hodnota `x` je väčšia ako `y`, inak `False`
`x >= y`	výsledkom je `True`, ak hodnota `x` je väčšia alebo rovná `y`, inak `False`
`x < y`	výsledkom je `True`, ak hodnota `x` je menšia ako `y`, inak `False`
`x <= y`	výsledkom je `True`, ak hodnota `x` je väčšia alebo rovná `y`, inak `False`

Všimnime si, že operátory porovnania `==` a `!=`, `>` a `<=`, `<` a `>=` sú vzájomné logické opaky.

Jednoduché logické výrazy môžeme v prípade potreby skladať do zložených logických výrazov. Využívame k tomu logické spojky:

`x and y`	Vo význame **a zároveň**. Výsledkom je `True`, ak obidve hodnoty `x` a `y` majú hodnotu `True`, inak `False`.
`x or y`	Vo význame **a alebo**. Výsledkom je `True`, ak aspoň jedna z hodnôt `x` a `y` má hodnotu `True`, inak `False`.
`not x`	Vo význame **nie**. Výsledkom je `True`, ak hodnota `x` je `False`, inak `True`.

Pozrime sa na konkrétne logické výrazy a spôsob ich vyhodnocovania:

```
cislo1 = 1
cislo2 = 2
pravda = True
nepravda = False

print(cislo1 < cislo2)   # True
print(cislo2 > cislo1 >= cislo1)   # True
print(cislo1 == cislo2)   # False
```

```
print(cislo1 != cislo2)  # True

print(pravda and nepravda)  # False
print(pravda or nepravda)  # True
print(not pravda)  # False
print(cislo1 < cislo2 and cislo2 < cislo1 + cislo2)  # True

print(nepravda or pravda and nepravda)  # False
print(nepravda or pravda and not cislo1 < cislo2)  # False
```

Pri vyhodnocovaní zložených logických výrazov majú logické spojky a operátory porovnania vopred určenú prioritu – poradie v akom sa vyhodnocujú. Ich priorita, od najnižšej po najvyššiu je nasledujúca:

- `or`
- `and`
- `not`
- `<`, `<=`, `>`, `>=`, `!=`, `==`

Rozoberme si podrobnejšie vyhodnotenie posledného zo zložených logických výrazov:

`nepravda or pravda and not cislo1 < cislo2`.

Výraz sa vyhodnocuje sprava doľava, pretože v tomto smere klesá priorita operátorov. Ak si nie sme istí, ako sa bude výraz vyhodnocovať alebo chceme uprednostniť vyhodnotenie nejakej jeho časti, môžeme prioritu určiť pomocou zátvoriek. Výraz v zátvorke sa vyhodnotí prednostne. Ak v uvedenom výraze „potvrdíme" prioritu vyhodnocovania zátvorkami, výsledkom je výraz:

`nepravda or (pravda and (not (cislo1 < cislo2)))`.

Podmienený príkaz if

Sila logických výrazov spočíva najmä v tom, že na základe ich hodnoty môžeme vetviť vykonávanie programu – rozhodnúť, ktorý z niekoľkých blokov príkazov sa vykoná. Na samotné vetvenie programu slúži príkaz `if`. Podľa problému, ktorý riešime, môžeme využiť niektorý z možných tvarov príkazu `if`. Všimnime si odsadenie blokov príkazov v jednotlivých častiach príkazu `if`.

`if`

```
if logický výraz:
    blok príkazov
```

Ak je `logický výraz` pravdivý, vykoná sa `blok príkazov`. V opačnom prípade sa `blok príkazov` nevykoná.

`if-else`

```
if logický výraz:
    blok príkazov 1
```

```
else:
    blok príkazov 2
```

Ak je `logický výraz` pravdivý, vykoná sa `blok príkazov 1`. V opačnom prípade sa vykoná `blok príkazov 2`.

`if-elif-else`

```
if logický výraz 1:
    blok príkazov 1
elif logický výraz 2:
    blok príkazov 2
else:
    blok príkazov 3
```

Ak je `logický výraz 1` pravdivý, vykoná sa `blok príkazov 1`. V opačnom prípade sa overuje pravdivostná hodnota výrazu `logický výraz 2`. Ak je `logický výraz 2` pravdivý, vykoná sa `blok príkazov 2`. V opačnom prípade sa vykoná `blok príkazov 3`.

Častí `elif` môže byť uvedených viac. Prvý z logických výrazov ktorý sa vyhodnotí ako pravdivý určí, ktorý blok príkazov sa vykoná. Zvyšné logické výrazy sa už nevyhodnocujú a celá konštrukcia `if - elif - else` sa ukončí.

Ak žiaden z logických výrazov v častiach `if` a `elif` nie je pravdivý, vykoná sa blok príkazov v časti `else`. Časť `else` nie je povinná a ak nepotrebujeme vykonávať `blok príkazov 3`, môžeme ju vynechať.

Konkrétne použitie príkazu `if` môže vyzerať nasledovne. V ukážkach sme pred aj za konštrukciu `if` vložili výpis, aby sme demonštrovali, že tieto časti nie sú príkazom `if` ovplyvnené.

```
print('Začiatok programu')
a = 15
b = 10
if a > b:
    print(f'{a} je väčšie ako {b}')
print('Koniec programu')
```

```
Začiatok programu
15 je väčšie ako
10
Koniec programu
```

```
print('Začiatok programu')
a = 5
b = 10
if a > b:
    print(f'{a} je väčšie ako {b}')
print('Koniec programu')
```

```
Začiatok programu
Koniec program
```

```
print('Začiatok programu')
a = 5
```

```
Začiatok programu
```

```
b = 10
if a > b:
    print(f'{a} je väčšie ako {b}')
else:
    print(f'{a} nie je väčšie ako {b}')
print('Koniec programu')
```

```
5 nie je väčšie
ako 10
Koniec programu
```

```
print('Začiatok programu')
a = 5
b = 5
if a > b:
    print(f'{a} je väčšie ako {b}')
elif a < b:
    print(f'{a} je menšie ako {b}')
else:
    print(f'{a} sa rovná {b}')
print('Koniec programu')
```

```
Začiatok programu
5 sa rovná 5
Koniec programu
```

V logických výrazoch môžeme kombinovať logické operátory a operátory porovnania a vytvárať tak pomerne komplikované logické výrazy. Vždy by sme sa mali zamyslieť, či výsledné zostavenie podmienok je to najefektívnejšie. Pozrime sa na problém, ako osloviť súťažiaceho vo vedomostnej hre podľa pohlavia a veku. Pre oslovenie sme si definovali nasledovné pravidlo:

pohlavie / vek	<19	>=19
muž	žiak	študent
žena	žiačka	študentka

Prvoplánové riešenie, v ktorom zostavíme podmienku pre každé oslovenie zvlášť, by mohlo vyzerať nasledovne:

```
if pohlavie == 'muž' and vek < 19:
    print('žiak')
elif pohlavie == 'muž' and vek >= 19:
    print('študent')
elif pohlavie == 'žena' and vek < 19:
    print('ziačka')
else:
    print('študentka')
```

Porovnajme ho s nasledujúcim riešením:

```
if pohlavie == 'muž':
    if vek < 19:
        print('žiak')
    else:
        print('študent')
else:
```

```
    if vek < 19:
        print('žiačka')
    else:
        print('študentka')
```

Zamyslime sa, koľko výrazov sa musí vyhodnotiť, aby sme dostali oslovenie `žiak`?

V prvom prípade tri: test pohlavia, test veku a spojenie oboch testov spojkou `and`.

V druhom prípade sa vyhodnocujú len dva výrazy: test pohlavia a test veku. Pre ostatné oslovenia je neefektívnosť prvého riešenia ešte viac výraznejšia.

Táto neefektívnosť je dôsledkom toho, že zistenia z vyhodnotenia predchádzajúcich logických výrazov nijako nevyužívame pri ďalších logických výrazoch. Ak by sme v prvej podmienke zistili, že pohlavie účastníka je `muž` a jeho vek nie je `<19`, aj tak v druhej podmienke opätovne testujeme hodnotu pohlavia.

V druhom prípade každú z hodnôt testujeme len raz. Nenechajme sa „pomýliť“ tým, že test veku je v programe napísaný 2-krát. Z týchto testov sa vždy realizuje len jeden, prvý pre pohlavie `muž` a druhý pre pohlavie `žena`.

Funkcie s logickou návratovou hodnotou

Ak odpoveď na logickú otázku nie je jednoduché získať alebo ak potrebujeme logický výraz vyhodnocovať opakovane, je výhodné na vyhodnotenie hodnoty definovať funkciu. Takáto funkcia, podobne ako logický výraz, vracia jednu z hodnôt, `True` alebo `False`.

Ukážme si funkciu, ktorá odpovie, či zadané číslo je zložené. Podľa definície: „Zložené číslo je každé také prirodzené číslo n, ktoré sa dá vyjadriť v tvare $n = a \cdot b$ kde a, b sú prirodzené čísla rôzne od 1“. Pre zložené čísla teda musí platiť viac podmienok:

- číslo musí byť celé,
- číslo musí byť väčšie ako 1,
- číslo musí mať deliteľa rôzneho od 1 a samého seba.

Ak zistíme, že niektorá z podmienok neplatí, ostatné už nie je potrebné vyhodnocovať.

```
def je_zlozene(n):
    if n % 1 != 0:
        return False
    if n < 2:
        return False
    odmocnina = int(n ** 0.5)
    for delitel in range(2, odmocnina + 1):
        if n % delitel == 0:
            return True
    return False
```

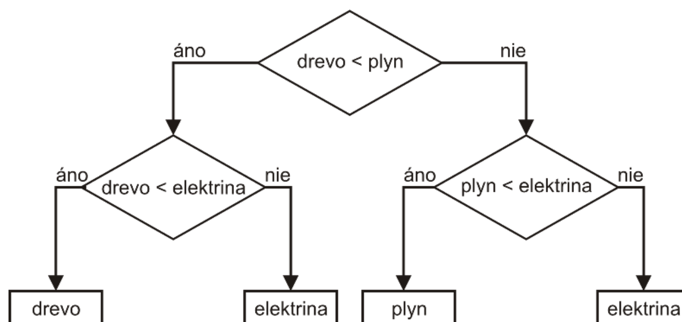
```
cislo = input('Zadaj číslo: ')

if je_zlozene(float(cislo)):
    print(f'Číslo {cislo} je zložené.')
else:
    print(f'Číslo {cislo} nie je zložené.')
```

Všimnime si niekoľko podstatných faktov:

- samotný názov funkcie `je_zlozene()` napovedá, že funkcia „odpovie“ `True` alebo `False`,
- hlavný program je prehľadný a ľahko pochopiteľný, ak by sme celý test realizovali v hlavnom programe, program by bol menej zrozumiteľný,
- v samotnej funkcii testujeme podmienky samostatne, ak sa prvý alebo druhý výraz vyhodnotí ako `True`, číslo nie je zložené, funkciu ukončíme a vrátime odpoveď `return False`,
- ak nájdeme netriviálneho deliteľa, číslo je zložené (`return True`),
- ak sme nenašli netriviálneho deliteľa, číslo nie je zložené (`return False`).

Zbierka úloh

1. V súťaži PALMA junior[3] bolo úlohou žiakov rozhodnúť, ktorý spôsob vykurovania je najvýhodnejší. Najskôr si žiaci vypočítali náklady na kúrenie pre jednotlivé typy paliva. Potom na základe rozhodovacieho stromu:



rozhodli, ktoré palivo je najvýhodnejšie.

Vytvorte rozhodovací strom podľa ktorého zistíme, či zadaný rok je priestupný.

[3] https://di.ics.upjs.sk/palmaj/zadania/2012_2013/1/6.doc

2. Na základe vytvoreného rozhodovacieho stromu z úlohy 1 formálne vyjadrite logický výraz pre priestupný rok.

3. Na základe výsledkov úlohy 1 a úlohy 2 vytvorte program, ktorý pre zadaný rok vypíše informáciu, či je priestupný lebo nie.

   Čo je výhodnejšie? Testovať celú podmienku naraz alebo po častiach?

4. Vytvorte program, ktorý pre zadané číslo mesiaca (od 1 do 12) vypíše, do ktorého ročného obdobia mesiac patrí.

5. Vytvorte program, ktorý pre zadané prirodzené číslo vypíše všetky jeho delitele.

6. Vytvorte funkciu `je_prvocislo()`, ktorá pre zadané prirodzené číslo zistí a vráti informáciu, či je prvočíslo alebo nie.

7. Vytvorte funkciu `bmi()`, ktorá pre zadanú výšku (v metroch) a hmotnosť (v kilogramoch) človeka, vypočíta a vráti jeho BMI.

   Vytvorte funkciu `kategoria()`, ktorá pre zadanú výšku a hmotnosť človeka, vráti kategóriu do ktorej patrí.

   Pre kategorizáciu využite napr.:
   https://sk.wikipedia.org/wiki/Index_telesnej_hmotnosti.

8. Vytvorte funkciu `korene_rovnice()`, ktorá pre zadané koeficienty (a, b, c) vypočíta a vráti korene rovnice $ax^2 + bx + c = 0$.

   - Uvažujte, že koeficient a je rôzny od 0.
   - Uvažujte, že koeficient a môže byť rovný 0.

   Pomôcka: Vytvorte si rozhodovací strom a rozdiskutujte prípady, keď sú jednotlivé koeficienty rovné 0.

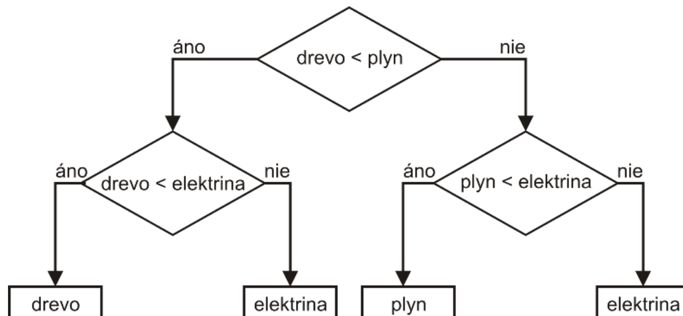
9. Vytvorte funkciu `kvadrant()`, ktorá pre zadané súradnice [x, y] bodu roviny vráti, v ktorom kvadrante bod leží alebo na ktorej osi leží alebo či je počiatkom súradnicovej sústavy.

10. Vytvorte funkciu `existuje_trojuholnik()`, ktorá pre tri zadané dĺžky strán zistí a vráti, či existuje trojuholník so zadanými dĺžkami strán.

11. Hru „Prehadzovaná“ hrá niekoľko tímov, ktoré si na začiatku rozdelia niekoľko lôpt. Počet hráčov tímov sa volí tak, aby sa všetky deti rozdelili do rovnako početných tímov. Lopty sa rozdelia tak, aby každý tím dostal rovnaký počet lôpt. Vytvorte funkciu `da_sa_rozdelit()`, ktorá pre zadaný počet detí a počet lôpt zistí, či zadaný počet lôpt je vhodný pre všetky prípustné rozdelenia detí do tímov.

Riešené úlohy

1. V súťaži PALMA junior[4] bolo úlohou žiakov rozhodnúť, ktorý spôsob vykurovania je najvýhodnejší. Najskôr si žiaci vypočítali náklady na kúrenie pre jednotlivé typy paliva. Potom na základe rozhodovacieho stromu:

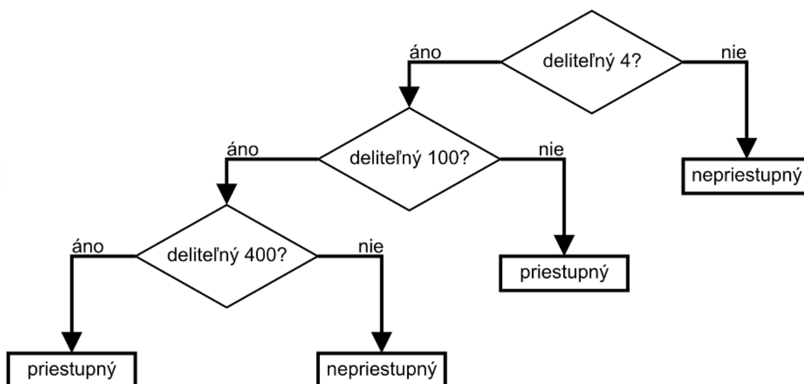


   rozhodli, ktoré palivo je najvýhodnejšie.

   Vytvorte rozhodovací strom podľa ktorého zistíme, či zadaný rok je priestupný.

   Riešenie:

   Podľa pravidiel pre priestupné roky platí, že rok je priestupný, ak je deliteľný číslom 4. Ak by bol zároveň deliteľný číslom 100, tak priestupný nie je. Ak by bol zároveň deliteľný aj číslom 400, tak priestupný je. Tieto pravidlá si môžeme vyjadriť vo forme rozhodovacieho stromu nasledovne:



2. Na základe vytvoreného rozhodovacieho stromu z úlohy 1 formálne vyjadrite logický výraz pre priestupný rok.

   Riešenie:

[4] https://di.ics.upjs.sk/palmaj/zadania/2012_2013/1/6.doc

Všimnime si (v rozhodovacom strome), že odpoveď „priestupný“ sa nachádza na konci dvoch vetiev stromu. Ak budú splnené podmienky v jednej alebo druhej vetve, rok bude priestupný.

Vo vetve končiacej vľavo dole platia nasledovné podmienky:

```
rok % 4 == 0 and rok % 100 == 0 and rok % 400 == 0
```

Vo vetve končiacej v strednej časti stromu platia nasledovné podmienky:

```
rok % 4 ==0 and rok % 100 != 0
```

Pre priestupný rok by mala platiť jedna alebo druhá podmienka. Zapíšme ich spoločne do jedného logického výrazu:

```
(rok % 4 == 0 and rok % 100 == 0 and rok % 400 == 0
     or rok % 4 == 0 and rok % 100 != 0)
```

Poznámka: Vonkajšie zátvorky sme použili len preto, aby sme mohli zalomiť príliš dlhý riadok. Ak v programe zapíšeme celý výraz do jedného riadku, vonkajšie zátvorky môžeme odstrániť.

3. Na základe výsledkov úlohy 1 a úlohy 2 vytvorte program, ktorý pre zadaný rok vypíše informáciu, či je priestupný lebo nie.

   Čo je výhodnejšie? Testovať celú podmienku naraz alebo po častiach?

   Riešenie:

   Ak máme definovaný logický výraz pre priestupný rok, môžeme ho priamo použiť v podmienenom príkaze `if`:

```
if (rok % 4 == 0 and rok % 100 == 0 and rok % 400 == 0
        or rok % 4 == 0 and rok % 100 != 0):
    print('priestupný')
else:
    print('nepriestupný')
```

   Tento zápis je logicky správny, ale takto zložitá podmienka je dosť neprehľadná. Pozrime si ešte raz rozhodovací strom. Z neho je logika rozhodovania ľahko čitateľná. Pokúsme sa rozhodovací strom priamo prepísať do textovej podoby.

```
if rok % 4 == 0:
    if rok % 100 == 0:
        if rok % 400 == 0:
            print('priestupný')
        else:
            print('nepriestupný')
    else:
        print('priestupný')
else:
    print('nepriestupný')
```

Všimnime si, že v tomto zápise aj opticky vidno stromovú štruktúru pôvodného rozhodovacieho stromu. Každá podmienka rozhodovacieho stromu je reprezentovaná logickým výrazom za slovom `if`. Navyše, ak si pozrieme vyššie definovaný rozsiahly logický výraz, všimneme si, že niektoré v ňom uvedené elementárne logické výrazy sa testujú opakovane. V druhej verzii programu sa každý z elementárnych logických výrazov testuje najviac raz.

4. Vytvorte program, ktorý pre zadané číslo mesiaca (od 1 do 12) vypíše, do ktorého ročného obdobia mesiac patrí.

Riešenie:

Podľa zadania môžeme predpokladať, že číslo mesiaca bude niektoré z čísiel od 1 do 12. Pre jednotlivé mesiace môžeme uvažovať nasledovné zaradenie do ročných období:

Jar: 3, 4, 5
Leto: 6, 4, 8
Jeseň: 9, 10, 11
Zima: 1, 2, 12

Pokúsme sa definovať, aké logické výrazy platia pre mesiace v jednotlivých ročných obdobiach:

Zima:	1, 2, 12	`mesiac < 3 or mesiac == 12`
Jar:	3, 4, 5	`2 < mesiac < 6`
Leto:	6, 7, 8	`5 < mesiac < 9`
Jeseň:	9, 10, 11	`8 < mesiac < 12`

Prvá verzia programu, v ktorej priamo využijeme predchádzajúce logické výrazy, by mohla vyzerať nasledovne:

```
if mesiac < 3 or mesiac == 12:
    print('zima')
elif 2 < mesiac < 6:
    print('jar')
elif 5 < mesiac < 9:
    print('leto')
elif 8 < mesiac < 12:
    print('jesen')
```

Toto riešenie je síce správne, ale nie veľmi efektívne. V čom je jeho neefektívnosť? Ak sa vykonávanie programu dostane k niektorému z logických výrazov (okrem prvého), nijako nevyužívame skutočnosť, že predchádzajúce logické výrazy neplatili. Napr. v druhom logickom výraze

nemá zmysel testovať, či `2 < mesiac`. To už vyplýva z toho, že prvý logický výraz neplatil.

Upravme program tak, aby sme priebežne využívali výsledky testov predchádzajúcich logických výrazov.

```
if mesiac < 3:
    print('zima')
elif mesiac < 6:
    print('jar')
elif mesiac < 9:
    print('leto')
elif mesiac < 12:
    print('jeseň')
else:
    print('zima')
```

Všimnime si aj podmienku pre zimu. Podmienku `mesiac == 12` vôbec netestujeme. Ani nemusíme. Ak vylúčime každú z možnosti od 1 do 11, ostáva jediná možnosť, mesiac 12, ktorý patrí zime.

6 Reťazce a metódy reťazcov

Jemný úvod do problematiky

Použite notebook **Python 6 - Reťazce a reťazcové metódy.ipynb** alebo niektoré z lokálnych vývojových prostredí jazyka Python.

Cieľ: *V tomto notebooku si ukážeme ako pracovať s reťazcami a ako využívať ich metódy.*

Reťazce

Reťazec je postupnosť znakov uzatvorená v `'` (apostrof) alebo `"` (úvodzovka). Aj keď sú tieto zápisy ekvivalentné, budeme používať apostrof.

`'Ahoj'`

`retazec = 'Dobrý deň'`

Ak už raz **reťazec** vytvoríme, je **nemenný**. Vieme však z neho vytvárať ďalšie reťazce. Vyhodnoťte nasledovné výrazy a pokúste sa vysvetliť, ako sa interpretuje číslo v zátvorke:

`retazec[0]`

`retazec[3]`

`retazec[-1]`

`retazec[-3]`

```
retazec = 'Dobrý deň'

print(retazec[0])
# alebo
novy = retazec[0]
print(novy)

# otestujte ďalšie výrazy, experimentujte s inými hodnotami
```

Číslo v zátvorke sa interpretuje ako index (pozícia) znaku v reťazci. Ak je kladné, číslujeme zľava (od 0). Ak je záporné, číslujeme sprava (od -1):

indexy zľava:	0	1	2	3	4	5	6	7	8	9	10
reťazec:	**i**	**n**	**f**	**o**	**r**	**m**	**a**	**t**	**i**	**k**	**a**
indexy sprava:	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Cvičenie 1

Vypíšte druhý znak z reťazca, ktorého hodnotou je vaše meno.

Vypíšte dvadsiaty znak z reťazca, ktorého hodnotou je vaše meno.

Prechod znakmi reťazca

Reťazec je iterovateľná štruktúra – to znamená, že môžeme pomocou cyklu prechádzať cez jednotlivé znaky reťazca. Pri prechode uvažujeme dve situácie. Buď potrebujeme pracovať s indexom (pozíciou) daného znaku alebo nie. Vyskúšajte nasledovné prístupy:

```
retazec = 'Dobrý deň'
for znak in retazec:
    print(znak)
```

alebo

```
retazec = 'Dobrý deň'
for index in range(len(retazec)):
    print(index, retazec[index])
```

Zamyslime sa! Čo robí funkcia `len()`?

Cvičenie 2

Vypíšte znaky vášho mena aj s ich pozíciou.

Cvičenie 3

Vypíšte každý druhý znak vášho mena aj s jeho pozíciou.

Pomôcka: Spomeňte si na rôzne variácie funkcie `range()`.

Operátor príslušnosti

Reťazec je štruktúra, t.j. je zložená z menších častí: znaky reťazca alebo jeho podreťazce. Užitočným operátorom je operátor príslušnosti `in`, pomocou ktorého vieme otestovať prítomnosť podreťazca v reťazci. Otestujte rôzne situácie:

```
'myš' in 'myšlienka'
'lienka' in 'myšlienka'
```

Výrezy

Z reťazcov môžeme vyberať aj väčšie časti (výrezy) ako len jeden znak. Vyhodnoťte nasledovné výrazy a pokúste sa vysvetliť, ako sa interpretujú čísla v zátvorke? Svoje predpoklady si overte zmenou hodnôt v zátvorkách.

`retazec = 'informatika'`

`retazec[1:7]`

`retazec[1:]`

`retazec[:7]`

`retazec[1:5:2]`

```
retazec = 'Dobrý deň'

print(retazec[1:7])
# alebo
novy = retazec[1:7]
print(novy)

# otestujte aj ďalšie výrezy
```

Výrez z reťazca môžeme definovať pomocou troch čísiel oddelených znakom dvojbodka. Ak niektoré z nich neuvedieme, nahradí sa prednastavenou hodnotou.

`retazec[start:]`	znaky reťazca od pozície `start` až do konca s krokom 1
`retazec[start:stop]`	znaky reťazca od pozície `start` do pozície `stop` s krokom 1
`retazec[:stop]`	znaky reťazca od pozície 0 do pozície `stop` s krokom 1
`retazec[start:stop:krok]`	znaky reťazca od pozície `start` do pozície `stop` s krokom `krok`

Mimochodom, aký podreťazec dostaneme pomocou výrezu: `retazec[::-1]`?

Cvičenie 4

Vypíšte každý druhý znak vášho mena. Nepoužite funkciu `range()`.

Metódy reťazcov

Reťazce nám ponúkajú pomerne veľa funkcií (metód), pomocou ktorých s nimi vieme pracovať. Metódy sú špeciálne funkcie reťazcov, ktoré voláme tak, že za reťazcom alebo reťazcovou premennou uvedieme oddeľovač `.` a názov metódy.

Čo robia nasledovné metódy? Svoje predpoklady si overte zmenou ich parametrov.

`retazec = '   Dobrý deň   '`

`retazec.index('ý')`

`retazec.replace('D', 'd')`

`retazec.strip()`

`retazec.count(' ')`

`retazec.upper()`

Viac o reťazcových metódach nájdeme na:

- https://docs.python.org/3.8/library/string.html#module-string
- https://docs.python.org/3/library/stdtypes.html?#string-methods

Cvičenie 5

Len pomocou metód reťazcov upravte reťazec

`'   moje      visvedcenie   '`

do tvaru

`'Moje vysvedčenie'`.

Formátovanie reťazcov

Pri práci s reťazcami budeme často vytvárať nové reťazce kombináciou iných reťazcov, premenných alebo hodnôt. Od verzie Python 3.6 je k dispozícii jednoduchý a pritom efektívny nástroj ako reťazce formátovať: f-string.

Vyskúšajme nasledovný programový kód:

```
meno = 'Karol'
vek = 25
print(f'Volám sa {meno} a mám {vek} rokov.')
```

alebo

```
meno = 'Karol'
vek = 25
print(f'Volám sa {meno} a mám {vek * 12} mesiacov.')
```

Cvičenie 6

Napíšte programový kód, ktorý bude produkovať pre zadané číslo (napr. 10) nasledovný výpis:

```
Druhá mocnina čísla 1 je 1.
Druhá mocnina čísla 2 je 4.
Druhá mocnina čísla 3 je 9.
Druhá mocnina čísla 4 je 16.
Druhá mocnina čísla 5 je 25.
Druhá mocnina čísla 6 je 36.
Druhá mocnina čísla 7 je 49.
Druhá mocnina čísla 8 je 64.
Druhá mocnina čísla 9 je 81.
Druhá mocnina čísla 10 je 100.
```

Cvičenie 7

Napíšte programový kód, ktorý bude produkovať pre zadané číslo (napr. 10) a zadaný reťazec (napr.: `'Toto je nejaký reťazec'`) nasledovný výpis:

```
Reťazec dĺžky 1: 'T'
Reťazec dĺžky 2: 'To'
Reťazec dĺžky 3: 'Tot'
Reťazec dĺžky 4: 'Toto'
Reťazec dĺžky 5: 'Toto '
Reťazec dĺžky 6: 'Toto j'
Reťazec dĺžky 7: 'Toto je'
Reťazec dĺžky 8: 'Toto je '
Reťazec dĺžky 9: 'Toto je n'
Reťazec dĺžky 10: 'Toto je ne'
```

Vysvetlenie problematiky

Reťazce

Textové hodnoty sú v Pythone reprezentované pomocou reťazcov. Reťazec je postupnosť znakov uzatvorená v apostrofoch `'` alebo úvodzovkách `"`.

```
"Toto je textový reťazec"
'Aj toto je textový reťazec'
```

Obidva zápisy sú ekvivalentné, ale v učebných textoch používame znak apostrof `'`. Použitie apostrofu sa javí výhodnejšie z dvoch dôvodov:

- Pri samotnom programovaní máme väčšinou aktivovanú anglickú klávesnicu. Znak `'` vieme napísať priamo stlačením jedného klávesu. Znak `"` píšeme pomocou kombinácie s klávesom Shift.
- V slovenčine sa skôr stretneme so znakom `"`. Reťazec obsahujúci `"` uzatvorený `'` je korektný zápis. Ak by bol takýto reťazec uzatvorený v úvodzovkách, zápis by bol nekorektný.

```
'Spýtal sa: "Si pripravený?"'  # korektný zápis
"Spýtal sa: "Si pripravený?""  # nekorektný zápis
```

V špeciálnych prípadoch, ak potrebujeme vytvoriť viacriadkový reťazec, obalíme reťazec do trojitých apostrofov `'''`:

```
basen = '''A ty mor ho! - hoj mor ho! detvo môjho rodu,
kto kradmou rukou siahne na tvoju slobodu;
a čo i tam dušu dáš v tom boji divokom:
Mor ty len, a voľ nebyť, ako byť otrokom.'''

print(basen)
```

```
A ty mor ho! - hoj mor ho! detvo môjho rodu,
kto kradmou rukou siahne na tvoju slobodu;
a čo i tam dušu dáš v tom boji divokom:
Mor ty len, a voľ nebyť, ako byť otrokom.
```

Znak konca riadku (~Enter) je korektným znakom, a môže byť súčasťou pythonovských reťazcov. Ak potrebujeme do reťazca vložiť znak, ktorý by mal byť interpretovaný iným spôsobom, použijeme znak spätnej lomky `\`, napr.:

```
print('Text s \'')
print('Text \t s tabulátorom')
print('Text s \\')
print('Text \nv riadkoch')
```

```
Text s '
Text 	 s tabulátorom
Text s \
Text
v riadkoch
```

V niektorých prípadoch by sme potrebovali pracovať s konkrétnym znakom reťazca Jednotlivé znaky reťazca sú prístupné cez ich pozície – indexy. Python umožňuje indexovať reťazec zľava (od hodnoty 0) aj sprava (od hodnoty -1):

indexy zľava:	0	1	2	3	4	5	6	7	8	9	10
reťazec:	**i**	**n**	**f**	**o**	**r**	**m**	**a**	**t**	**i**	**k**	**a**
indexy sprava:	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
retazec = 'informatika'
print(retazec[1])
print(retazec[-2])
print(retazec[15])
```

```
n
k

Traceback (most recent call last):
  File "...", line 4, in <module>
    print(retazec[15])  # IndexError ...
IndexError: string index out of range
```

Všimnime si, že ak sa pokúsime pristúpiť k indexu, ktorý v reťazci neexistuje, spôsobí to chybu a predčasné ukončenie programu.

Reťazce nemusíme vytvárať len ich priamym zápisom. Nové reťaze vieme vytvoriť aj spájaním (reťazením) iných reťazcov pomocou operandu `+`:

```
zviera1 = 'myš'
zviera2 = 'lienka'
spojenie = zviera1 + zviera2
print(spojenie)  # myšlienka
```

```
myšlienka
```

Prechod znakmi reťazca

V situáciách, ak potrebujeme systematicky pracovať s každým znakom reťazca využijeme skutočnosť, že reťazec je iterovateľná štruktúra. Táto vlastnosť nám umožní pomocou cyklu prechádzať cez jednotlivé znaky reťazca.

```
retazec = 'Ahoj'
for znak in retazec:
    print(znak)
```

```
A
h
o
j
```

Premenná `znak` postupne nadobúda hodnoty znakov zadaného reťazca. Všimnime si, že k znakom reťazca pristupujeme aj bez toho, aby sme poznali ich index.

Niekedy potrebujeme pracovať nie len s hodnotami znakov, ale aj s ich indexami. V tomto prípade použijeme cyklus cez indexy reťazca. Samotné hodnoty indexov budeme generovať pomocou funkcie `range()`.

```
retazec = 'Ahoj'
for index in range(len(retazec)):
    print(f'index {index}, znak {retazec[index]}')
```

```
index 0, znak A
index 1, znak h
index 2, znak o
index 3, znak j
```

Všimnime si funkciu `len()`, ktorá vracia dĺžku (počet znakov) reťazca.

Rozhodnutie, ktorý z dvoch predchádzajúcich prístupov použijeme záleží od konkrétnej situácie. Ak potrebujeme pracovať len s hodnotami znakov, použijeme cyklus cez znaky reťazca. Ak potrebujeme pracovať aj s indexami znakov, použijeme cyklus cez indexy reťazca. Hodnoty indexov generujeme pomocou funkcie `range()`. Stop hodnotu definuje dĺžka reťazca, ktorú zistíme funkciou `len()`. Ak poznáme index znaku, vieme zistiť aj jeho hodnotu. Naopak to neplatí.

Operátor príslušnosti

Operátor príslušnosti `in` použijeme v situácii, ak potrebujeme zistiť, či nejaký znak alebo reťazec sa nachádza v inom reťazci.

```
print('myš' in 'myšlienka')
print('myš' in 'myš')
print('' in 'myš')
print('lienka' in 'myšlienka')
print('myšlienka' in 'lienka')
```

```
True
True
True
True
False
```

Pomocou operátora `in` vytvárame logické výrazy. Ich hodnotou je jedna z hodnôt `True` alebo `False`. Všimnime si, že prázdny reťazec `''` je podreťazcom každého reťazca.

Operátor `in` sa dá použiť všeobecne. Neskôr si ukážeme jeho použitie aj pri iných štruktúrach než len reťazce.

Výrezy

Nové reťazce vieme zatiaľ vytvoriť:

- priamym zápisom reťazca,
- výberom znaky reťazca pomocou indexu,
- zreťazením iných reťazcov.

Výrezy (angl. slices) sú ďalším spôsobom ako vytvárať nové reťazce z už existujúcich reťazcov. Výrez definujeme štartovacím indexom, koncovým indexom a krokom. Tieto hodnoty vzájomne oddelíme znakom `:`.

```
retazec = 'informatika'
print(retazec[:5])   # infor
print(retazec[5:])   # matika
print(retazec[2:7])   # forma
print(retazec[2:7:2])   # fra
```

```
infor
matika
forma
fra
```

Ak niektoré z hodnôt výrezu neuvedieme, nahradia sa prednastavenou hodnotou. Vo všeobecnosti definujeme výrezy nasledovne.

`retazec[start:]`	znaky reťazca od pozície `start` až do konca s krokom 1
`retazec[start:stop]`	znaky reťazca od pozície `start` do pozície `stop` s krokom 1
`retazec[:stop]`	znaky reťazca od pozície 0 do pozície `stop` s krokom 1
`retazec[start:stop:krok]`	znaky reťazca od pozície `start` do pozície `stop` s krokom `krok`

Výrezy umožňujú aj komplikované činnosti realizovať pomerne jednoducho. Napr. otočenie reťazca, čo by v iných programovacích jazykoch znamenalo „rozoberanie" reťazca po znakoch a ich spájanie v opačnom poradí, sa v Python-e zrealizuje jednoducho: `retazec[::-1]`. Pripomeňme si, že reťazce sú nemenné. Výrez z reťazca teda nemení pôvodný reťazec a jeho výsledkom je nový reťazec.

Metódy reťazcov

Reťazce sú jedným najfrekventovanejších typov, s ktorými sa pri programovaní v Pythone stretneme. Samotné reťazce ponúkajú množstvo funkcií (metód), pomocou ktorých s nimi vieme pracovať. Metódy sú špeciálne funkcie reťazcov, ktoré voláme tak, že za reťazcom alebo reťazcovou premennou uvedieme oddeľovač `.` a názov metódy.

`capitalize()`	vráti kópiu reťazca, kde prvý znak je veľký a ostatné malé `'skratka USB'.capitalize()` → `'Skratka usb'`
`count(podreťazec)`	vráti počet výskytov podreťazca v reťazci `'blablabla'.count('blabla')` → `1`
`find(podreťazec)`	vráti najmenší index, na ktorom sa podreťazec v reťazci nachádza, ak sa podreťazec v reťazci nenachádza, vracia `-1` `'12 13 13 12'.find('13')` → `1`
`index()`	podobne ako `find()`, ale ak sa podreťazec v reťazci nenachádza, výsledkom je chyba
`isalnum()`	vráti `True`, ak reťazec obsahuje len alfanumerické znaky a reťazec má dĺžku aspoň 1, inak `False`

	`'3 2 1 štart'.isalnum()` → `False` `'321štart'.isalnum()` → `True`

`isalpha()`	vráti `True`, ak reťazec obsahuje len alfa znaky a reťazec má dĺžku aspoň 1, inak `False`
	`'321štart'.isalnum()` → `False`
	`'štart'.isalnum()` → `True`

`islower()`	vráti `True`, ak reťazec obsahuje len malé znaky a reťazec má dĺžku aspoň 1, inak `False` `'321štart'.islower()` → `True`
`isupper()`	vráti `True`, ak reťazec obsahuje len veľké znaky a reťazec má dĺžku aspoň 1, inak `False` `'321štart'.isupper()` → `False`
`lower()`	vráti kópiu reťazec, kde všetky znaky sú malé `'3 2 1 ŠTART'.lower()` → `'3 2 1 štart'`
`replace(stare, nove)`	vráti kópiu reťazec, kde podreťazce `stare` sú nahradene reťazcom `nove` `'blablabla'.replace('bla', 'BLA ')` → `'BLA BLA BLA '`
`strip()`	vráti kópiu reťazca, v ktorom sú vynechané biele znaky na jeho začiatku a konci `'  bla    '.strip()` → `'bla'`
`swapcase()`	vráti kópiu reťazca, v ktorom sú malé znaky skonvertované na veľké a naopak `'HOREdole'.swapcase()` → `'horeDOLE'`
`upper()`	vráti kópiu reťazca, kde všetky znaky sú veľké `'3 2 1 štart'.upper()` → `'3 2 1 ŠTART'`

Uvedené metódy sú len výberom z existujúcich metód. Navyše, správanie sa niektorých metód môžeme ovplyvniť nepovinnými parametrami. Viac o reťazcových metódach a reťazcoch nájdeme na:

- https://docs.python.org/3/library/stdtypes.html?#string-methods
- https://docs.python.org/3.9/library/string.html

Formátovanie reťazcov

S formátovaním reťazcom pomocou `f-string` (Python 3.6 a novšie) sme sa čiastočne oboznámili v 1. kapitole a priebežne sme ho používali aj neskôr. `F-string` má však aj iné využitie než len vkladanie hodnôt premenných do reťazcov. Vkladané hodnoty je možné formátovať.

Zaokrúhlenie čísla:

```
pi = 3.141592653589793
print(f'pi na tri desatinné miesta: {pi:.3f}')
```

```
pi na tri desatinné miesta: 3.142
```

Odsadzovanie, resp. zarovnanie na určitý počet miest:

```
meno1 = 'Ivo'
plat1 = 1258
meno2 = 'Kamila'
plat2 = 450
print(f'{meno1:8} -> {plat1:5d}')
print(f'{meno2:8}->{plat2:5d}')
```

```
Ivo      -> 1258
Kamila   ->  450
```

Zápis čísla v inej sústave (binárnej, osmičkovej, desiatkovej, šestnástkovej):

```
for i in range(256):
    print(f'{i:8b}{i:5o}{i:5d}{i:4X}')
```

```
       0    0    0   0
       1    1    1   1
      10    2    2   2
...
11111110  376  254  FE
11111111  377  255  FF
```

Viac informácií nájdeme v dokumentácii jazyka Python:

- https://docs.python.org/3/tutorial/inputoutput.html#formatted-string-literals
- https://docs.python.org/3/library/string.html#formatspec

Zbierka úloh

Nasledujúce úlohy sú zamerané na manipuláciu s reťazcami a používanie metód reťazcov. Zvážte, v ktorých prípadoch je výhodné definovať ďalšie funkcie (okrem tých, ktoré sú požadované v zadaní) s vhodnými návratovými hodnotami.

1. Palindróm je text, ktorý pri čítaní spredu aj zozadu znie rovnako, napr. `'anna'`. Vytvorte funkciu `je_palindrom()`, ktorá pre zadaný text zistí a vráti, či je palindróm alebo nie. Úlohu môžete riešiť vo viacerých úrovniach:
   - Uvažujte texty obsahujúce len písmená malej abecedy (napr.: `'anna'`).
   - Uvažujte texty obsahujúce navyše aj veľké písmená (napr.: `'Anna'`).
   - Uvažujte texty obsahujúce navyše aj medzery (napr.: `'Elze je zle'`).
2. Pri registrácii do rôznych systémov je potrebné vytvoriť si prihlasovacie meno a heslo. Meno by nemalo byť súčasťou hesla a naopak (napr.: 'lienka' a 'myšlienka'). Vytvorte funkciu `je_sucastou()`, ktorá pre zadané meno a heslo zistí, či jedno je súčasťou druhého alebo naopak.
3. Vytvorte funkciu `vytvor_diktat()`, ktorá zadaný text upraví tak, že všetky výskyty znakov `'i'` a `'y'` nahradí znakom podčiarkovník a výsledok vráti. Úlohu môžete riešiť vo viacerých úrovniach:
   - Uvažujte texty obsahujúce len písmená malej anglickej abecedy `'yi'`.
   - Uvažujte texty obsahujúce aj veľké písmená anglickej abecedy `'iyIY'`.
   - Uvažujte texty obsahujúce aj znaky s diakritikou `'iyIYíýÍÝ'`.
4. Vytvorte funkciu `pocet_slov()`, ktorá pre zadaný text zistí, koľko je v ňom slov. Úlohu môžete riešiť vo viacerých úrovniach.
   - Predpokladajte, že medzera je použitá iba na oddelenie dvoch slov.
   - Predpokladajte, že niekde je aj viac medzier medzi slovami.
   - Predpokladajte, že medzery môžu byť aj na začiatku aj na konci textu.
5. PigLatin je slovná hračka (skôr pre anglické texty), ktorá každé slovo v texte upraví nasledovne:
   - Ak slovo začína samohláskou, pridá sa na jeho koniec koncovka `'yay'`, napr: `'omelet'` → `'omeletyay'`.
   - Ak slovo začína na spoluhlásku, presunie sa táto na jeho koniec a pridá sa koncovka `'ay'`,
     napr. `'happy'` → `'appyhay'`.

   Vytvorte funkciu `slovo_do_piglatin()`, ktorá vráti preklad zadaného slova do jazyka PigLatin.
6. Pre prihlasovacie meno požadujeme, aby:
   - malo aspoň 8 znakov,
   - obsahovalo len znaky abecedy,
   - prvé písmeno bolo veľké,
   - všetky písmená okrem prvého boli malé.

Vytvorte funkciu `je_meno_dobre()`, ktorá otestuje zadané prihlasovacie meno podľa vyššie uvedených pravidiel a vráti výsledok testu.

7. Jednoduchý šifrovací algoritmus spočíva vtom, že z textu vyberieme znaky na párnych pozíciách a znaky na nepárnych pozíciách. Každú z nových postupností otočíme a vzájomne ich spojíme do jednej postupnosti.

   Napr.: `'informatika'` → `'aiarfiktmon'`.

   Vytvorte funkciu `sifruj_par_nepar()`, ktorá zadaný text zašifruje podľa uvedeného pravidla a výsledok vráti.

8. Metóda reťazca `count()` vráti počet výskytov reťazca v podreťazci.

   Napr.: `'mama ma emu ema ma mamu'.count('ma')` → `6`

   Niekedy však funguje inak, než by sme očakávali.

   Napr.: `'abababa'.count('aba')` → `2`. V tomto prípade by sme očakávali odpoveď 3.

   Vytvorte funkciu `pocet_vsetkych_vyskytov()`, ktorá pre zadanú vzorku a text vráti počet všetkých výskytov vzorky v texte.

   Napr.: `pocet_vsetkych_vyskytov('aba', 'abababa')` → `3`

9. Jednoduchá kalkulačka

   Vytvorte funkciu `vyhodnot_vyraz()`, ktorej zadáme reťazec typu: `'cislo1 operand cislo2'`, funkcia výraz vyhodnotí a výsledok vráti.

   Napr.: `vyhodnot_vyraz('13 / 5')` → `2.6`

Riešené úlohy

1. Palindróm je text, ktorý pri čítaní spredu aj zozadu znie rovnako, napr. `'anna'`. Vytvorte funkciu `je_palindrom()`, ktorá pre zadaný text zistí a vráti, či je palindróm alebo nie. Úlohu môžete riešiť vo viacerých úrovniach:

   - Uvažujte texty obsahujúce len písmená malej abecedy (napr.: `'anna'`).
   - Uvažujte texty obsahujúce navyše aj veľké písmená (napr.: `'Anna'`).
   - Uvažujte texty obsahujúce navyše aj medzery (napr.: `'Elze je zle'`).

   Riešenie:

   Ak uvažujeme len písmená malej abecedy, stačí priamo porovnať reťazec s jeho otočenou verziou. Otočiť reťazec môžeme pomocou výrezov. Výsledná funkcia môže vyzerať nasledovne:

```
def je_palindrom(text):
    return text == text[::-1]
```

Ak by sme uvažovali, že v texte môžu byť písmená rôznej veľkosti, pričom malú a veľkú verziu toho istého písmena budeme považovať za zhodné, musíme text najskôr upraviť. Stačí zjednotiť veľkosť písmen.

```
def je_palindrom(text):
    text = text.lower()
    return text == text[::-1]
```

Ak uvažujeme aj znak medzera, ktorý pri čítaní „zanedbáme“, stačí z textu na začiatku medzery odstrániť.

```
def je_palindrom(text):
    text = text.replace(' ', '')
    text = text.lower()
    return text == text[::-1]
```

3. Vytvorte funkciu `vytvor_diktat()`, ktorá zadaný text upraví tak, že všetky výskyty znakov `'i'` a `'y'` nahradí znakom podčiarkovník a výsledok vráti. Úlohu môžete riešiť vo viacerých úrovniach:

- Uvažujte texty obsahujúce len písmená malej anglickej abecedy `'yi'`.
- Uvažujte texty obsahujúce aj veľké písmená anglickej abecedy `'iyIY'`.
- Uvažujte texty obsahujúce aj znaky s diakritikou `'iyIYíýÍÝ'`.

Riešenie:

Samotná úprava textu pozostáva z nahradenia znakov `'yi'` podčiarkovníkom. Tu môžeme priamo využiť metódu reťazcov `replace()`. Riešenie vyzerá nasledovne:

```
def vytvor_diktat(text):
    text = text.replace('i', '_')
    text = text.replace('y', '_')
    return text
```

Ak budeme uvažovať aj ostatné verzie znakov `'yi'`, stačí zopakovať nahradzovanie aj pre ostatné verzie:

```
def vytvor_diktat(text):
    text = text.replace('i', '_')
    text = text.replace('y', '_')
    text = text.replace('I', '_')
    text = text.replace('Y', '_')
    text = text.replace('í', '_')
    text = text.replace('ý', '_')
    text = text.replace('Í', '_')
    text = text.replace('Í', '_')
    return text
```

Toto riešenie je síce správne, ale kopírovanie programového kódu nie je najlepšou vizitkou programátora. Ak by v kopírovanom programovom kóde

bola chyba, kopírovaním ju násobíme a pri jej oprave musíme opraviť všetky kópie pôvodného programového kódu. Uvedomme si pôvodný zámer „stačí zopakovať nahradzovanie“, ktorý nás priamo nabáda na použitie cyklu. Cyklus by mal prechádzať cez všetky verzie znakov `'yi'` a realizovať ich nahradzovanie znakom podčiarkovník.

```
def vytvor_diktat(text):
    icka = 'iyIYíýÍÝ'
    for icko in icka:
        text = text.replace(icko, '_')
    return text
```

7 Algoritmy na reťazcoch

Jemný úvod do problematiky

Použite notebook **Python 7 - Algoritmy na reťazcoch.ipynb** alebo niektoré z lokálnych vývojových prostredí jazyka Python.

Cieľ: *V tomto notebooku si ukážeme ako využívať reťazce pri riešení náročnejších algoritmických úloh.*

Manipulácia s reťazcami

Pri manipulácii s reťazcami budeme využívať nasledovné znalosti a prístupy:

- reťazce sú nemenné,
- prístup k znakom reťazcov cez index,
- extrahovanie podreťazcov z reťazcov pomocou výrezov,
- prechod cez znaky reťazca, či už priamo alebo pomocou indexov,
- metódy reťazcov https://docs.python.org/3/library/stdtypes.html?#string-methods,
- formátovanie reťazcov.

Väčšina úloh sa dá riešiť prvoplánovo (~ riešenie, ktoré nás napadne ako prvé, pričom nemusí byť veľmi šikovné). Toto však nebude našim cieľom. Budeme sa snažiť v čo najväčšej miere využiť možnosti, ktoré nám jazyk Python ponúka a vytvárať efektívne a zrozumiteľné programy. Nesnažme sa však už od začiatku vytvárať „dokonalé“ riešenie. Vytvorme nejaké fungujúce riešenie a postupne ho vylepšujme.

Problém: Vymysleli sme si jednoduchý algoritmus na šifrovanie textov. V každej susednej dvojici znakov vzájomne otočíme poradie znakov. Napr. text `'Ahoj'` zašifrujeme na `'hAjo'`. Tento algoritmus sme implementovali nasledovne:

```
def sifruj(text):
    sifrovany_text = ''
    for i in range(0, len(text), 2):
        dvojica = text[i:i+2]
        sifrovany_text = sifrovany_text + dvojica[::-1]
    return sifrovany_text

print(sifruj('Ahoj'))
```

```
hAjo
```

Cvičenie 1

Upravte funkciu `sifruj()`, aby sme text šifrovali tak, že otáčanie znakov budeme realizovať podľa zadaného `n` pre každú n-ticu.

Cvičenie 2

Vytvorte funkciu na dešifrovanie takto zašifrovaných textov.

Cvičenie 3

V súťaži v písaní na stroji sa porovnáva znak po znaku originálny text s textom, ktorý prepísal súťažiaci.

- Za každú zhodnú dvojicu znakov získa súťažiaci bod.
- Ak súťažiaci vzájomne prehodil dva po sebe idúce znaky, jeden bod sa mu odpočíta.
- Ak súťažiaci napísal iný znak ako sa očakával, odpočítajú sa mu dva body.

Vytvorte funkciu `vyhodnot()`, ktorá porovná originálny text s prepísaným textom a vráti bodový zisk súťažiaceho.

Napr. pre originálny text `'súťažiaci'` a prepis `'síťažaici'` je bodový zisk 3 body.

Poznámka: Môžete predpokladať, že porovnávané texty sú rovnako dlhé.

Cvičenie 4

Pri písaní na počítači sme si nevšimli, že klávesy Z a Y sú vzájomne vymenené. Vytvorte funkciu `vymen_z_y()` ktorá opraví takto chybne napísaný text.

- Uvažujte len malé písmená `z` a `y`.
- Uvažujte aj veľké písmená `ZY`.
- Uvažujte aj písmená s dĺžňami `źýŹÝ`.
- Uvažujte aj písmená, pri ktorých sa napísal mäkčeň `ž` ↔ `ˇy` a `Ž` ↔ `ˇY`.

Vysvetlenie problematiky

Kódovanie znakov

Programy, ktoré pracujú so znakmi musia byť schopné pracovať so širokou množinou znakov. Nestačí uvažovať len znaky príslušnej národnej abecedy rozšírené o niekoľko špeciálnych znakov (napr. interpunkčné znamienka). S programom môže pracovať aj inojazyčný cudzinec alebo odborník, ktorý vo svojom odbore používa špeciálne znaky (napr. matematické ∃∞ ±). Python preto používa štandardné kódovanie UNICODE (https://www.unicode.org/). Vďaka tomuto kódovaniu zvládne Python pracovať so znakmi rôznych abecied

či rôznych špeciálnych množín znakov. Každému znaku je pridelený jednoznačný číselný kód, ktorý reprezentuje poradie znaku v postupnosti všetkých znakov v UNICODE.

Napríklad znaku `'a'` je priradený kód 97. Znak `'A'` predstavuje rozdielny znak a je mu pridelený kód 65. Celkom môžeme v Pythone pracovať so 1114112 znakmi, ktoré majú pridelené kódy od 0 do 1114111.

Pre prácu zo znakmi a ich kódmi využijeme dvojicu funkcií:

- `ord()` – vráti kód zadané znaku, jeho poradie v tabuľke UNICODE
  `ord('a')` → `97`,
- `chr()` – vráti znak so zadaným poradím v tabuľke UNICODE
  `chr(97)` → `'a'`.

Problém s porovnaním reťazcov

Kódovanie UNICODE môže v istých špecifických prípadoch spôsobiť problémy pri porovnávaní reťazcov. Tento problém je spôsobený faktom, že niektoré znaky môžu mať priradených viacero kódov. Napr. písmeno `'ô'` (malé písmeno o s vokáňom) môže byť reprezentované jedným kódom 244 alebo dvojicou kódov 111 a 770.

- `chr(244)` → `ô`
  písmeno ô,
- `chr(111)` + `chr(770)` → `o` + `̂`
  písmeno o nasledované znakom vokáň.

Obidve reprezentácie pri výstupe (napr. pomocou funkcie `print()`) zobrazia ten istý symbol, písmeno `ô`.

```
o_vokan1 = f'{chr(244)}'
o_vokan2 = f'{chr(111)}{chr(770)}'
print(o_vokan1)
print(o_vokan2)
```

```
ô
ô
```

Pri porovnaní reťazcov, ktoré obsahujú rôzne kódy písmena `ô`, bude výsledok `False`.

```
print(o_vokan1 == o_vokan2)
```

```
False
```

Jednou z možností, ako vyriešiť tento problém, je normalizovať reťazce pomocou funkcie `normalize()` z modulu `unicodedata`. Funkcia skonvertuje reťazec do niektorej z normalizovaných foriem, kde sú kombinované kódy nahradené jedným kódom.

```
import unicodedata
o_vokan1 = unicodedata.normalize('NFD', o_vokan1)
o_vokan2 = unicodedata.normalize('NFD', o_vokan2)

print(o_vokan1 == o_vokan2)
```

```
True
```

Vyššie uvedený problém s porovnávaním nastane najčastejšie vtedy, ak porovnávame reťazce pochádzajúce z rôznych systémov, napr. súbor a vstup z klávesnice.

Zbierka úloh

Nasledujúce úlohy sú zamerané na algoritmy s reťazcami. V úlohách budeme využívať výrezy, indexy a metódy reťazcov a ich vzájomné kombinácie.

1. Robot pri svojom pohybe zaznamenáva všetky vykonané kroky:

   `s` = sever,

   `j` = juh,

   `v` = východ,

   `z` = západ.

   Výsledkom je postupnosť znakov (napr.: `'sssvvszzzj'`). Vytvorte funkciu `navrat()`, ktorá pre zadanú postupnosť vykonaných krokov robota vráti najkratšiu postupnosť krokov pre jeho návrat do východiskovej pozície.

2. Vytvorte funkciu `porovnaj()` na opravu diktátu, ktorá porovná zadaný reťazec žiaka a reťazec učiteľa. Na výstupe vráti reťazec, ktorý na líšiacich sa pozíciách oboch reťazcov bude obsahovať podčiarkovník. Úlohu môžete riešiť vo viacerých úrovniach:
   - Uvažujte reťazce s rovnakou dĺžkou.
     `porovnaj('bycigel','bicykel')` → `'b_c__el'`.
   - Uvažujte reťazce s rôznou dĺžkou.
     `porovnaj('bycigelik','bicykel')` → `'b_c__el__'`.

3. Vytvorte funkciu `je_rodne_cislo_ok()`, ktorá pre zadaný reťazec vypíše či môže predstavovať zápis rodného čísla, t. j. či obsahuje zápis 10 ciferného čísla (s prípadnou oddeľovacou lomkou medzi šesticou a štvoricou číslic), ktoré je deliteľné číslom 11. Úlohu môžete riešiť vo viacerých úrovniach:
   - Kontrolujte len deliteľnosť číslom 11.
   - Kontrolujte aj existenciu dátumu. Pre kontrolu korektnosti dátumu môžete využiť funkciu `je_datum_ok()`:

```
def je_datum_ok(den, mesiac, rok):
    ''' Vráti, či zadaný deň, mesiac a rok
        predstavuje korektný dátum

    :param den: číslo dňa v mesiaci
    :type den: int
    :param mesiac: číslo mesiaca v roku
    :type mesiac:  int
    :param rok: číslo roku
    :type rok: int
    :return: True, ak je dátum korektný, inak False
    :rtype: bool
    '''
    import datetime
    try:
        den = int(den)
        mesiac = int(mesiac)
        rok = int(rok)
        datetime.date(rok, mesiac, den)
    except ValueError:
        return False
    return True
```

Napr. funkcia `overenie('965524/4379')` vráti výsledok `True`.

4. Vytvorte funkciu `preklop()`, ktorá v zadanom texte ponechá samohlásky a medzery na pôvodnom mieste a spoluhlásky sa otočia odzadu.

   Napr. `preklop('ahojte kamarati')` → `'atorme katajahi'`. Pre jednoduchosť uvažujme len o textoch, ktoré používajú malé písmena anglickej abecedy a medzeru.

5. Cézarova šifra pracuje na princípe posúvania znakov v abecede. Každé písmeno textu sa nahradí písmenom, ktoré je v abecede o pevne určený počet (=kľúč) miest ďalej. Ak by sme sa pri posune dostali za posledný znak abecedy, pokračujeme od začiatku. Ak sa znak v abecede nenachádza, ponechá sa bez zmeny.

   Ak uvažujeme len písmená malej anglickej abecedy, šifra slova `informatika` s posunom `10` je `sxpybwkdsuk`.

   Vytvorte funkciu `sifruj_s_posunom()`, ktorá bude text šifrovať posúvaním jeho znakov v abecede a výslednú šifru vráti. Úlohu môžete riešiť vo viacerých úrovniach:

   - Uvažujte šifrovaciu abecedu, v ktorej sú len znaky malej anglickej abecedy.
   - Uvažujte šifrovaciu abecedu, v ktorej sú aj znaky veľkej anglickej abecedy.

- Uvažujte šifrovaciu abecedu, v ktorej sú aj znaky s diakritikou.
- Uvažujte šifrovaciu abecedu, v ktorej sú aj interpunkčné znamienka a medzera.
- Pre rôzne pozície znakov v texte uvažujte rôzne posuny. Kľúč určujúci jednotlivé posuny zadáme ako reťazec znakov, pričom ordinálne hodnoty znakov určujú jednotlivé posuny. Napr. pre kľúč `key` sa znaky textu rozdelia do trojíc, pričom znaky v trojiciach sa budú postupne posúvať o:
    - 107 miest = `ord('k')`,
    - 101 miest = `ord('e')`,
    - 121 miest = `ord('y')`.

Pre výslednú funkciu definujte aj funkciu `desifruj_s_posunom()`.

Poradie znakov v abecede si zvoľte podľa vlastného uváženia.

6. Jednoduchý šifrovací algoritmus spočíva v tom, že z textu vyberieme znaky na párnych pozíciách a znaky na nepárnych pozíciách. Každú z nových postupností otočíme a vzájomne ich spojíme do jednej postupnosti.

   Napr.: `'informatika'` → `'aiarfiktmon'`.

   Túto úlohu sme riešili v predchádzajúcej kapitole.

   Vytvorte funkciu `desifruj_par_nepar()`, ktorá zadaný zašifrovaný text dešifruje podľa uvedeného pravidla a výsledok vráti. Napr.:

   `desifruj_par_nepar('aiarfiktmon')` → `'informatika'`

7. Jednoduchý šifrovací algoritmus spočíva v tom, že slovo sa rozdelí na štvrtiny a tie sa v slove preusporiadajú tak, že žiadna štvrtina neostane na svojom mieste a že žiadne dve štvrtiny, ktoré boli v pôvodnom slove vedľa seba už nebudú vedľa seba (preusporiadanie štvrtín si môžete zvoliť napevno).
   - Vytvorte funkciu `sifruj_stvrtiny()`, ktorá zadané slovo zašifruje podľa uvedených pravidiel a výsledok šifrovania vráti.
   - Vytvorte inverznú funkciu `desifruj_stvrtiny()`, ktorá dešifruje zadané zašifrované slovo a výsledok dešifrovania vráti.

8. Náročnosť slova sa vypočíta ako pomer počtu spoluhlások k počtu samohlások. Vytvorte funkciu `narocnost_slova()`, ktorá pre zadané slovo vráti jeho náročnosť.

9. Pomocou operátora príslušnosti `in` vieme zistiť, či text obsahuje zadanú vzorku.

   Napr.: `'ľad' in 'priehľadný'` → `True`

   Hľadaná vzorka však musí byť v prehľadávanom texte súvislá.

Navrhnite funkciu `je_v_texte(vzorka, text)` ktorá zistí, či sa vzorka nachádza v text, pričom jej výskyt nemusí byť súvislý.

napr.: `je_v_texte('nota', 'informatika')` → `True`

`je_v_texte('mafin', 'informatika')` → `False`

Riešené úlohy

1. Robot pri svojom pohybe zaznamenáva všetky vykonané kroky:

    `s` = sever,

    `j` = juh,

    `v` = východ,

    `z` = západ.

    Výsledkom je postupnosť znakov (napr.: `'sssvvszzzj'`). Vytvorte funkciu `navrat()`, ktorá pre zadanú postupnosť vykonaných krokov robota vráti najkratšiu postupnosť krokov pre jeho návrat do východiskovej pozície.

    Riešenie:

    Riešenie tejto úlohy môžeme rozdeliť ma dve časti. O koľko a ktorým smerom sa robot posunul zo štartovacej pozície v smere sever-juh a koľko a ktorým smerom sa robot posunul zo štartovacej pozície v smere východ-západ.

    Rozoberme si situáciu v smere sever-juh. Krok na sever a krok na juh sa vzájomne eliminujú a to bez ohľadu na to, v akom poradí boli realizované.

    Napr.: `'ssj'` = `'sjs'` = `'jss'` = `'s'`

    Ak zistíme rozdiel v počtoch krokov na sever a na juh, vieme koľko ktorých krokov má robot spraviť v smere sever-juh. Podobne môžeme uvažovať aj v smere východ-západ. Funkcia `navrat()` môže vyzerať nasledovne:

```
def navrat(cesta:str):
    pocet_j = cesta.count('j')
    pocet_s = cesta.count('s')
    pocet_z = cesta.count('z')
    pocet_v = cesta.count('v')
    cesta_spat = ''
    if pocet_v > pocet_z:
        cesta_spat += (pocet_v - pocet_z) * 'z'
    else:
        cesta_spat += (pocet_z - pocet_v) * 'v'
    if pocet_j > pocet_s:
        cesta_spat += (pocet_j - pocet_s) * 's'
    else:
        cesta_spat += (pocet_s - pocet_j) * 'j'
    return cesta_spat
```

Všimnime si, že správnych odpovedí môže byť viac. Budú sa líšiť len v inom usporiadaní krokov. Je teda jedno v akom poradí krokov „vyskladáme“ cestu späť. Počet krokov v ceste späť je jednoznačne určený rozdielom vzájomne opačných krokov.

5. Cézarova šifra pracuje na princípe posúvania znakov v abecede. Každé písmeno textu sa nahradí písmenom, ktoré je v abecede o pevne určený počet (=kľuč) miest ďalej. Ak by sme sa pri posune dostali za posledný znak abecedy, pokračujeme od začiatku. Ak sa znak v abecede nenachádza, ponechá sa bez zmeny.

   Ak uvažujeme len písmená malej anglickej abecedy, šifra slova `informatika` s posunom `10` je `sxpybwkdsuk`.

   Vytvorte funkciu `sifruj_s_posunom()`, ktorá bude text šifrovať posúvaním jeho znakov v abecede a výslednú šifru vráti.

   - Uvažujte šifrovaciu abecedu, v ktorej sú znaky malej a veľkej anglickej abecedy.
   - Pre rôzne pozície znakov v texte uvažujte rôzne posuny. Kľúč určujúci jednotlivé posuny zadáme ako reťazec znakov, pričom ordinálne hodnoty znakov určujú jednotlivé posuny. Napr. pre kľúč `key` sa znaky textu rozdelia do trojíc, pričom znaky v trojiciach sa budú postupne posúvať o:
     - 107 miest = `ord('k')`,
     - 101 miest = `ord('e')`,
     - 121 miest = `ord('y')`.

   Pre výslednú funkciu definujte aj funkciu `desifruj_s_posunom()`.

   Poradie znakov v abecede si zvoľte podľa vlastného uváženia.

   Riešenie:

   Vyriešme najskôr vytvorenie šifrovacej abecedy: malé a veľké písmená anglickej abecedy. Využijeme fakt, že znaky abecedy sú v UNICODE tabuľke zoradené za sebou. Vieme ich teda „pozbierať“ v cykle:

```
abeceda = ''
for i in range(ord('a'), ord('z') + 1):
    abeceda += chr(i)
```

   K malým písmená pridáme aj veľké:

```
abeceda += abeceda.upper()
```

   Šifru textu budeme realizovať po znakoch. Ak znak textu je v šifrovanej abecede, tak do výslednej šifry textu pridáme jeho šifru. V opačnom prípade znak nešifrujeme a do šifry pridáme pôvodný znak.

Keďže znaky kľúča, ktoré určujú posuny znakov v texte, budeme používať cyklicky, budeme si udržiavať aj informáciu o aktuálnej pozícii v kľúči `idx_kluc`. Pozíciu po každom zašifrovanom znaku zväčšíme o 1. Ak by nová pozícia v kľúči bola mimo kľúč, posunieme ju na začiatok. Výsledná funkcia môže vyzerať nasledovne:

```
def sifruj_s_posunom(text, kluc):
    abeceda = ''
    for i in range(ord('a'), ord('z') + 1):
        abeceda += chr(i)
    abeceda += abeceda.upper()
    sifra = ''
    idx_kluc = 0
    for znak in text:
        if znak in abeceda:
            posun = ord(kluc[idx_kluc])
            poz_znak = abeceda.index(znak)
            poz_sifra_znak = (poz_znak + posun)
                             % len(abeceda)
            sifra_znak = abeceda[poz_sifra_znak]
            sifra += sifra_znak
            idx_kluc = (idx_kluc + 1) % len(kluc)
        else:
            sifra += znak
    return sifra
```

8 Chyby a spracovanie výnimiek

Jemný úvod do problematiky

Použite notebook **Python 8 - Chyby a spracovanie výnimiek.ipynb** alebo niektoré z lokálnych vývojových prostredí jazyka Python.

Cieľ: *V tomto notebooku si ukážeme, s akými typmi chýb sa pri programovaní stretávame. Zameriame sa najmä na chyby, ktoré vznikajú počas behu programu a na to, ako ich v programe spracovať.*

Typy chýb

Porovnajte nasledovné tri bloky programových kódov a zamyslite sa nad tým, v čom (typovo) sa líšia chyby, ktoré sme v nich urobili.

Chyba 1

```
# chyba 1
delenec = 15
delitel = 3
pritn(f'{delenec} / {delitel} = {delenec / delitel}')
print('.. ďalší výpočet pokračuje')
```

```
Traceback (most recent call last):
  File "...", line 4, in <module>
    pritn(f'{delenec} / {delitel} = {delenec / delitel}')
NameError: name 'pritn' is not defined

Process finished with exit code 1
```

Chyba 2

```
# chyba 2
delenec = 15
delitel = 3
print(f'{delenec} / {delitel} = {delitel / delenec}')
print('.. ďalší výpočet pokračuje')
```

```
15 / 3 = 0.2
.. ďalší výpočet pokračuje
```

Chyba 3

```
# chyba 3
delenec = input('Zadaj delenca: ')
delenec = float(delenec)
delitel = input('Zadaj deliteľa: ')
delitel = float(delitel)
```

```
print(f'Výsledok delenia: {delenec} / {delitel} = {delenec / 
delitel}')
print('.. ďalší výpočet pokračuje')
```

```
Zadaj delenca: 15
Zadaj deliteľa: 0

Traceback (most recent call last):
  File "...", line 20, in <module>
    print(f'Výsledok delenia: {delenec} / {delitel} = 
{delenec / delitel}')
ZeroDivisionError: float division by zero
Process finished with exit code 1
```

Zhrnutie

Ak ste si pozorne prezreli vyššie vzniknuté chyby zistili ste, že:

- chyba 1 je **syntaktická**, spravili sme preklep, takúto chybu vieme odhaliť ešte pred samotným spustením programu (napr. editor výraz označí červenou farbou),
- chyba 2 je **logická**, v podiele sme vymenili čitateľa s menovateľom, takúto chybu vieme odhaliť dôsledným testovaním programu, skúšame či pre vopred premyslené vstupné hodnoty dostávame očakávané výstupné hodnoty,
- chyba 3 je **behová**, používateľ zadal deliteľa rovného 0, táto chyba sa teda prejavila až počas behu programu a je spôsobená chybným použitím nášho programu.

Ak program **predčasne skončil s nejakou chybou** hovoríme, že **vygeneroval (vyhodil) výnimku**. V nasledujúcej časti sa bližšie pozrieme na to, ako pracovať s behovými chybami.

Odchytávanie výnimiek

Syntaktické a logické chyby vieme vopred eliminovať. Behová chyba predstavuje akýsi „výnimočný“ prípad behu programu, keď vykonávanie programu nemôže štandardným spôsobom pokračovať ďalej. Dôsledkom takejto chyby je, že program nepracuje správne alebo skončí predčasne. K behovým chybám pristupujeme inak ako k syntaktickým a logickým. Snažíme sa ich predvídať a vopred sa na ne pripraviť. Vyššie uvedený programový kód (chyba 3) upravíme nasledovne. Overte si, ako prebehne výpočet pre 0 v menovateli.

```
# chyba 3
delenec = input('Zadaj delenca: ')
delenec = float(delenec)
delitel = input('Zadaj deliteľa: ')
delitel = float(delitel)
```

```
try:
    print(f'Výsledok delenia: {delenec} / {delitel} =
{delenec / delitel}')
except ZeroDivisionError:
    print('Zadal si delitela rovného 0. Nulou sa deliť
nedá!')
print('.. ďalší výpočet pokračuje')
```

Blok `try` obsahuje programový kód, ktorého vykonávanie môže byť problematické. V bloku `try` teda očakávame, že nejaká chyba nastane (samozrejme, nie vždy). Za blokom `try` nasleduje blok `except` s typom chyby (`ZeroDivisionError`), ktorú očakávame. V tomto bloku je časť programu, ktorá sa vykoná ak chyba `ZeroDivisionError` nastane. Ak táto chyba nenastane, blok sa preskočí a vykonávanie programu pokračuje za ním.

Ak očakávame, že v bloku `try` môžu nastať rôzne typy chýb, blokov `except` môže byť za sebou uvedených viac. Každý s iným typom chyby.

Cvičenie 1

Vytvorili sme jednoduchý program pre výpočet druhej odmocniny čísla. Ak však zadáme zápornú hodnotu, program predčasne skončí s chybou. Upravte program tak, aby v prípade zadaného záporného čísla program výnimku odchytil, vypísal upozornenie a pokračoval ďalej.

```
import math

cislo = input('Zadaj číslo, ktorého odmocninu chceš vedieť:
')
cislo = float(cislo)
print(f'Odmocnina z čísla {cislo} je {math.sqrt(cislo)}.')
print('.. ďalší výpočet pokračuje')
```

Cvičenie 2

Z hodín fyziky si pamätáme, že prejdenú dráhu pri rovnomerne zrýchlenom pohybe vypočítame podľa vzťahu: $s = \frac{1}{2}at^2$, kde *s* je dráha, *a* je zrýchlenie a *t* je čas.

Zo vzťahu sme vyjadrili čas $t = \sqrt{\frac{2s}{a}}$ a definovali sme funkciu `cas_zrychleneho_pohybu()`. Pri niektorých vstupoch však program skončí s chybou.

Ošetrite časť hlavného programu tak, aby program neskončil s chybou, ale so zrozumiteľnou informáciou pre používateľa.

```
import math

def cas_zrychleneho_pohybu(draha, zrychlenie):
    draha = float(draha)
    zrychlenie = float(zrychlenie)
    cas = math.sqrt(2 * draha / zrychlenie)
    return cas


# upravte nasledujúci programový kód
draha = input('Zadaj dráhu: ')
zrychlenie = input('Zadaj zrýchlenie: ')

print(f'Dráhu dĺžky {draha} pri zrýchlení {zrychlenie}
prejdeme za čas {cas_zrychleneho_pohybu(draha,
zrychlenie)}.')
```

Vysvetlenie problematiky

Typy chýb

Pri programovaní robíme množstvo chýb. Niektoré sú dôsledkom našej nepozornosti, napr. preklep. Iné sú spôsobné chybnou analýzou problému, napr. nesprávny výpočet hodnoty. No a nakoniec sú tu chyby, ktoré vzniknú, ak sa náš program dostane do situácie o ktorej sme nepredpokladali, že nastane.

Vo všeobecnosti môžeme chyby pri programovaní rozdeliť do troch kategórií:

- syntaktické chyby,
- logické chyby,
- behové chyby.

Pozrime sa na jednotlivé typy chýb a na možnosti, ako ich eliminovať.

Syntaktické chyby

Syntaktické chyby sú spôsobené preklepom v programe, nesprávnym zostavením príkazov programu a pod. Na tieto chyby nás môže upozorniť samotné vývojové prostredie (nie každé prostredie má túto funkcionalitu integrovanú) alebo samotný syntaktický analyzátor (parser) programového kódu jazyka po spustení programu. Samotný chybový výpis je dostatočne zrozumiteľný na to, aby sme tieto chyby vedeli odstrániť. Napr.:

```
pritn('ahoj')
```

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

```
print('ahoj)
```

```
  File "...", line 1
    print('ahoj)
                ^
SyntaxError: EOL while scanning string literal...
```

```
if 1 < 2:
print('ahoj')
```

```
File "...", line 2
    print('ahoj')
    ^
IndentationError: expected an indented block
```

Syntaktický analyzátor identifikuje miesto v programe kde nastala chyba (číslo riadku, názov chybného príkazu a pod) a odstránenie týchto chýb je v princípe jednoduché.

Logické chyby

Na logické chyby nás neupozorní žiadna implicitná kontrola jazyka Python. Tieto chyby musí cielene odhaliť samotný autor programu. Aj tu existujú postupy, ktoré nám pomôžu.

Pozrime si nasledujúci program, ktorý rieši rovnicu tvaru $ax^2 + bx + c = 0$, kde a, b, c sú reálne koeficienty:

```
# načítanie koeficientov a, b, c

if a != 0:
    d = b ** 2 - 4 * a * c
    if d > 0:
        x1 = (-b + d ** 0.5) / (2 * a)
        x2 = (-b + d ** 0.5) / (2 * a)
        print(x1, x2)
    elif d == 0:
        x = b / (2 * a)
        print(x)
    else:
        print(None)
elif b != 0:
    x = -a / b
    print(x)
elif c == 0:
    print('R')
else:
    print(None)
```

Na jednej strane môžeme oceniť snahu autora odlíšiť situácie, v ktorých sa niektoré koeficienty rovnajú 0. Na druhej strane výsledný program je neprehľadný a ťažko kontrolovateľný. Pravdepodobne sú v ňom aj nejaké chyby, ale ťažko ich nejako systematicky hľadať.

Pozrime sa na iné riešenie toho istého problému:

```
def ries_kvad_rovnicu(a, b, c):
    d = b ** 2 - 4 * a * c
    if d > 0:
        x1 = (-b + d ** 0.5) / (2 * a)
        x2 = (-b - d ** 0.5) / (2 * a)
        return x1, x2
    elif d == 0:
        x = -b / (2 * a)
        return x

def ries_lin_rovnicu(a, b):
    x = -b / a
    return x

def ries_konst_rovnicu(a):
    if a == 0:
        return 'R'

def ries_rovnicu(a, b, c):
    if a != 0:
        return ries_kvad_rovnicu(a, b, c)
    if b != 0:
        return ries_lin_rovnicu(b, c)
    return ries_konst_rovnicu(a)
```

V tomto prípade bola dekompozícia dôsledná a riešenia jednotlivých prípadov sú implementované v samostatných funkciách. Každú z funkcií vieme otestovať samostatne. Najskôr nezávislé funkcie (`ries_kvad_rovnicu()`, `ries_lin_rovnicu()` a `ries_konst_rovnicu()`) a potom výslednú, na nich závislú funkciu (`ries_rovnicu()`). Vďaka dôslednej dekompozícii vieme samostatne testovať čiastkové funkcie a pomocou ich overených správnych definícií zostaviť a otestovať aj riešenie výslednej funkcie. Týmto prístupom riešenie budujeme a testujeme postupne.

Dáta pre testovanie by sme mali vyberať cielene. Vyberajme dáta:

- pre ktoré vieme predikovať výsledok,
- ktoré predstavujú aj krajné, málo pravdepodobné inštancie problému,
- ktoré pokrývajú rôzne typy inštancií problému.

Aj napriek takémuto prístupu sa môže stať, že riešenie niektorej z funkcií je chybné. Chyba môže byť taká „nenápadná“, že ju pohľadom nedokážeme odhaliť. Vieme

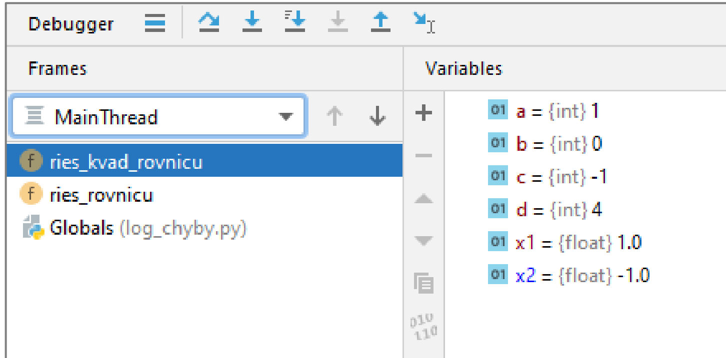
o nej len na základe chybných výstupov. Štandardnou funkciou vývojových prostredí býva režim ladenia. V tomto režime ručne spúšťame program, riadok po riadku. Pri vykonaní príkazu v riadku overujeme, či sledovaná premenná nadobudla očakávanú hodnotu alebo nie. Takto dokážeme lokalizovať riadok (príkaz), v ktorom chyba nastala a opraviť ju.

Nasledujúce obrázky demonštrujú ladenie funkcie `ries_kvad_rovnicu()` v prostredí PyCharm Edu.

Na prvom obrázku vidíme hodnoty premenných v aktuálnom stave vykonávania programu a zvýraznený riadok, ktorý sa v nasledujúcom kroku vykoná.

```
def ries_kvad_rovnicu(a, b, c):   a: 1    b: 0    c: -1
    d = b ** 2 - 4 * a * c   d: 4
    if d > 0:
        x1 = (-b + d ** 0.5) / (2 * a)   x1: 1.0
        x2 = (-b - d ** 0.5) / (2 * a)   x2: -1.0
        return x1, x2
    elif d == 0:
        x = -b / (2 * a)
        return x
```

Na druhom obrázku sú zobrazené ovládacie prvky ladiaceho režimu aj s hodnotami všetkých premenných.



Viac o použití ladiaceho režimu v konkrétnom vývojovom prostredí sa dozviete na cvičeniach.

Behové chyby

Behová chyba nastane v prípade, keď počítač nie je schopný pokračovať v realizácii programu. Krok, ktorý nie je možné vykonať, spôsobí behovú chybu. Ak program **predčasne skončil s nejakou chybou** hovoríme, že **vygeneroval (vyhodil) výnimku**. Takáto chyba nastane, ak sa napr. pokúsime deliť 0, otvoriť neexistujúci súbor alebo zistiť hodnotu znaku reťazca na pozícii, ktorá v reťazci neexistuje.

Problémom pri odstraňovaní behových chýb je skutočnosť, že niekedy nastanú a niekedy nie. Aby sme identifikovali takéto problematické situácie a lokalizovali miesto v programe, kde k nim dochádza, musíme dobre analyzovať riešený problém a implementáciu navrhnutého riešenia. Ak lokalizujeme problematickú časť programového kódu, budeme ju spúšťať v akomsi „pokusnom" režime. Pythonu oznámime, že na tomto mieste očakávame nejaký problém a definujeme čo spraviť, ak očakávaný problém nastane.

Pozrime sa bližšie na program, ktorý vypočíta podiel dvoch čísiel:

```
delenec = input('Zadaj delenca: ')
delitel = input('Zadaj deliteľa: ')

delenec = float(delenec)
delitel = float(delitel)

podiel = delenec / delitel

print(podiel)
```

Jeden z možných problémov, ktorý môže nastať je, ak používateľ zadá deliteľa rovného 0. V tomto prípade program skončí nasledovne:

```
Zadaj delenca: 5
Zadaj deliteľa: 0
Traceback (most recent call last):
  File "...", line 7, in <module>
    podiel = delenec / delitel
ZeroDivisionError: float division by zero
```

Z výpisu vidno, že v riadku 7 nastala behová chyba typu `ZeroDivisionError` (delenie nulou). Príkaz v tomto riadku ošetríme nasledovne:

```
try:
    podiel = delenec / delitel
except ZeroDivisionError:
    print('Nulou sa deliť nedá')
else:
    print(podiel)
```

Blok `try` obsahuje problematickú časť programového kódu, v ktorej očakávame možný problém, delenie 0. V časti `except ZeroDivisionError` definujeme, čo sa má stať, ak tento problém nastane. V tomto prípade vypíšeme chybovú správu. V časti `else` definujeme, čo sa má stať, ak očakávaná chyba nenastala – vypíšeme výsledok delenia. Táto verzia program s rovnakým vstupom ako predchádzajúca skončí s nasledovným výstupom:

```
Zadaj delenca: 5
```

```
Zadaj deliteľa: 0
Nulou sa deliť nedá
```

Riadiaci príkaz `try – except` má vo všeobecnosti tvar:

```
try:
    # problematický programový kód
except chyba1:
    # programový kód, ktorý sa vykoná, ak nastane chyba 1
except chyba2:
    # programový kód, ktorý sa vykoná, ak nastane chyba 2
...
except:
    # programový kód, ktorý sa vykoná,
    # ak nastane chyba rôzna od predchádzajúcich
else:
    # programový kód ktorý sa vykoná, ak nenastane žiadna
chyba
finally:
    # programový kód ktorý sa vykoná vždy
```

Pozrime sa pozornejšie na náš program. Môže nastať aj nejaká iná behová chyba? Rýchlo prídeme na to, že problém môže nastať aj pri pretypovaní vstupov. Ak na vstupe nezadáme číslo, program skončí s chybou:

```
Zadaj delenca: 5
Zadaj deliteľa: x
Traceback (most recent call last):
  File "...", line 5, in <module>
    delitel = float(delitel)
ValueError: could not convert string to float: 'x'
```

Ošetrime aj túto chybu:

```
try:
    delenec = float(delenec)
    delitel = float(delitel)
    podiel = delenec / delitel
except ValueError:
    print('Zadal si nečíselnú hodnotu')
except ZeroDivisionError:
    print('Nulou sa deliť nedá')
else:
    print(podiel)
```

Program v prípade nečíselnej hodnoty skončí nasledovne:

```
Zadaj delenca: 5
Zadaj deliteľa: x
Zadal si nečíselnú hodnotu
```

Programový kód Python-u môže generovať množstvo rôznych typov výnimiek. Výnimky sú usporiadané v hierarchickej štruktúre od všeobecných po konkrétne, viď časť hierarchie výnimiek:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
```

Prevzaté z https://docs.python.org/3/library/exceptions.html#exception-hierarchy

Ak očakávame, že jeden blok programového kódu môže generovať rôzne typy výnimiek, odchytávajme ich v poradí od konkrétnejších (umiestnené vpravo) po všeobecnejšie (umiestnené vľavo). V opačnom prípade môže náš programový kód zareagovať na všeobecnú výnimku namiesto konkrétnej.

Zbierka úloh

Nasledujúce úlohy sú zamerané na chyby (syntaktické, logické a behové), na ich identifikáciu a ošetrenie chybových stavov.

1. Máme k dispozícii program na nájdenie koreňov rovnice $ax^2 + bx + c = 0$, kde a, b, c sú reálne čísla. Navrhnite vhodné testovacie dáta, pomocou ktorých overíme správnosť riešenia. Svoj návrh zdôvodnite.

2. Vytvorili sme program na výpočet koreňa lineárnej rovnice $ax + b = 0$:

```
def koren_linearnej_rovnice(a, b):
    a = float(a)
    b = flaot(b)
    koren = b / a
    returne koren

a = input('Zadaj lineárny koeficient: ')
b = input('Zadaj absolútny koeficient: ')
print(f'Koreň rovnice {a}x + {b} = 0 je:
{koren_linearnej_rovnice(b, a)}')
```

Spravili sme v ňom niekoľko chýb. Nájdite a opravte ich. Výsledok otestujte s vhodnými testovacími dátami.

3. Nasledujúci program počíta počet fliaš potrebných na uskladnenie kečupu:

```
objem_kecupu = input('Objem kečupu (v litroch): ')
objem_kecupu = float(objem_kecupu)
objem_flase = input('Objem fľaše (v litroch): ')
objem_flase = float(objem_flase)

# zaujimaju nas len doplna naplnene flase
print(f'Počet fliaš: {objem_kecupu // objem_flase}')
```

Spustite program pre rôzne vstupy. Pokúste sa identifikovať také vstupné hodnoty, ktoré spôsobia zastavenie behu programu.

Upravte program tak, aby pre identifikované chybné vstupy program neskončil s chybou, ale so zrozumiteľnou informáciou pre používateľa.

4. Úlohu „Jednoduchá kalkulačka“ a funkciu `vyhodnot_vyraz()` sme riešili v zbierke 6. Máme takéto riešenie:

```
def vyhodnot_vyraz(vyraz):
    return eval(vyraz)

print(vyhodnot_vyraz('15 / 3'))
```

Pre správne zadané výrazy program funguje. Ak je zadaný výraz nekorektný alebo sa nedá vyhodnotiť, program skončí s chybou. Upravte hlavný program tak, aby v prípade „zlého“ vstupu program neskončil s chybou, ale s výpisom informácie pre používateľa.

5. Funkcia `duplikuj_znak()` v zadanom texte duplikuje znak na zadanej pozícii.

Napr.: `duplikuj_znak('ahoj', 2)` → `'ahooj'`.

```
def duplikuj_znak(text, idx):
    text = text[:idx] + text[idx] + text[idx:]
    return text
```

Pokiaľ zadaný index v texte existuje, funkcia funguje správne. Ak neexistuje, funkcia vygeneruje výnimku. Upravte funkciu tak, aby v prípade neexistujúceho indexu funkcia vrátila pôvodný reťazec.

6. V záhradkárskej osade „Štvorčekovo“ si záujemca môže prenajať len pozemok v tvare štvorca. Stačí, ak oznámi, na akej ploche chce záhradkárčiť a geodet mu v osade určí štvorcový pozemok s požadovanou plochou. K tomu však geodet potrebuje poznať rozmery tohto pozemku. Pokúsil sa vytvoriť program, ktorý mu vypočíta rozmery pozemku s požadovanou plochou:

```
import math

plocha = input('Zadajte plochu pozemku v m2: ')
plocha = float(plocha)
strana = math.sqrt(plocha)
print(f'Pozemok má rozmery {strana} m x {strana} m.')
```

Nedokázal ho však dokončiť tak, aby v prípade nekorektného vstupu program zrozumiteľne oznámil túto chybu. Upravte program tak, aby pre korektný vstup vypísal rozmery pozemku a v opačnom prípade vypísal správu, že vstupný údaj nebol správny a prečo.

7. V programe na výpočet obvodu trojuholníka sa nám pomiešali riadky:

```
strana_a = input('a = ')
strana_b = input('b = ')
strana_c = input('c = ')
try:
    print('Program počíta obvod trojuholníka.')
    print('Zadaj dĺžky strán trojuholníka (a, b, c)')
    strana_a = float(strana_a)
    strana_b = float(strana_b)
    strana_c = float(strana_c)
    print(f'Obvod trojuholníka je {obvod(strana_a,
strana_b, strana_c)}.')
except ValueError:
    print('Na vstupe sa vyskytli nenumerické hodnoty.')
except ZeroDivisionError:
    print('Počas výpočtu došlo k deleniu nulou.')
else:
```

Ktoré riadky môžeme presunúť pred konštrukciu `try - except`?

Ktoré riadky presunieme do časti `else` konštrukcie `try - except`?

Sú všetky ošetrenia chýb (v časti `except`) potrebné? Ak nie, ktoré riadky by sme mohli vypustiť?

Svoje odpovede zrealizujte a ich správnosť si overte spustením programu pre rôzne vstupné hodnoty.

8. Študenti predmetu „Python pre pokročilých“ dostali za úlohu vytvoriť program, ktorý:

- overí, či zadaný reťazec je platným rodným číslom,
- pre zadané rodné číslo vytvorí nasledujúce rodné číslo (s rovnakým dátumom narodenia a pohlavím).

Navrhnite testovacie dáta pre overenie správnosti študentských programov.

Pre zjednodušenie uvažujte len desaťmiestne rodné čísla a roky od 1953.

Riešené úlohy

1. Máme k dispozícii program na nájdenie koreňov rovnice $ax^2 + bx + c = 0$, kde *a, b, c* sú reálne čísla. Navrhnite vhodné testovacie dáta, pomocou ktorých overíme správnosť riešenia. Svoj návrh zdôvodnite.

   Riešenie:

   V tejto úlohe nie je potrebné nič programovať. Úlohou je navrhnúť testovacie dáta pre overenie správnosti programu. Testovacie dáta by sme mali vybrať tak, aby sme vedeli predikovať výsledok, obsiahli všetky typy inštancií problému a zahrnuli aj krajné prípady.

   Podľa hodnôt koeficientov, môže ísť o kvadratickú, lineárnu alebo konštantnú rovnicu. Navrhnúť môžeme nasledovné testovacie dáta:

   1, 2, -3 → kvadratická rovnica, dve riešenia: 1, -3,
   1, -6, 9 → kvadratická rovnica, jedno riešenie: -3,
   2, 0, 1 → kvadratická rovnica, žiadne riešenie,
   0, 2, -1 → lineárna rovnica, jedno riešenie 0,5,
   0, 0, 0 → konštantná rovnica, nekonečne veľa riešení,
   0, 0, 1 → konštantná rovnica, žiadne riešenie.

   Všimnime si fakt, že hodnoty koeficientov sú zvolené tak, aby koeficienty nemali rovnaké hodnoty. Týmto vylúčime ich zámenu v programe.

   Podľa zadania sú koeficienty reálne čísla. Ak by sme poznali presné zadania úlohy, mohli by sme otestovať aj to, či program zareaguje správne aj na nečíselné vstupy.

5. Funkcia `duplikuj_znak()` v zadanom texte duplikuje znak na zadanej pozícii.

   Napr.: `duplikuj_znak('ahoj', 2)` → `'ahooj'`.

```
def duplikuj_znak(text, idx):
    text = text[:idx] + text[idx] + text[idx:]
    return text
```

   Pokiaľ zadaný index v texte existuje, funkcia funguje správne. Ak neexistuje, funkcia vygeneruje výnimku. Upravte funkciu tak, aby v prípade neexistujúceho indexu funkcia vrátila pôvodný reťazec.

   Riešenie:

   Jedno z riešení ktoré nám napadne, môže byť otestovať existenciu indexu v reťazci a podľa výsledku testu, text upraviť:

```
def duplikuj_znak(text, idx):
    if idx < len(text):
        text = text[:idx] + text[idx] + text[idx:]
    return text
```

Po lepšej analýze si uvedomíme, že príliš malé číslo by testom prešlo a Python podporuje aj záporné indexy. Upravíme preto test aj na ohraničenie zdola.

```
def duplikuj_znak(text, idx):
    if -len(text) <= idx < len(text):
        text = text[:idx] + text[idx] + text[idx:]
    return text
```

Pri ďalšej analýze si uvedomíme, že testom prejdú aj neceločíselne indexy. Mali by sme preto doplniť test na celočíselnosť indexu. Kým implementujeme ďalší z testov, zamyslime sa. Rovnaké testy predsa musí realizovať aj Python pri prístupe k znakom reťazca. Robíme teda duplicitnú prácu. Pozrime sa na to inak. Nechajme Python, aby zdvojenie zrealizoval a „spýtajme“ sa, či sa mu to podarilo. Ak áno, vrátime upravený reťazec. Ak nie, vrátime pôvodný reťazec.

```
def duplikuj_znak(text, idx):
    try:
        text = text[:idx] + text[idx] + text[idx:]
        return text
    except (IndexError, TypeError):
        return text
```

Tento prístup, ktorý by sme mohli nazvať „Ľahšie je žiadať o odpustenie, ako o dovolenie“ má aj inú výhodu. Medzi vykonaním testu a realizáciou akcie sa situácie môže zmeniť. Napr. v čase testu súbor existuje, ale keď z neho chceme čítať už nie, lebo ho medzitým operačný systém vymazal.

9 Generovanie výnimiek

Jemný úvod do problematiky

Použite notebook **Python 9 - Generovanie výnimiek.ipynb** alebo niektoré z lokálnych vývojových prostredí jazyka Python.

Cieľ: *V tomto notebooku si ukážeme, ako zabezpečiť, aby náš program v prípade problematického výpočtu vhodne zareagoval a upozornil, že výpočet nemôže pokračovať.*

Generovanie výnimiek

V predchádzajúcej kapitole sme si ukázali, že ak sa vykonávanie programu dostane do situácie, v ktorej nemôže štandardne pokračovať ďalej, Python vygeneruje výnimku. V takejto situácii sa môže ocitnúť aj náš programový kód. Pozrime si nasledovný programový kód, v ktorom sme vytvorili funkciu pre výpočet obsahu štvorca. Počíta funkcia `obsah_stvorca()` vždy správne? Nájdite prípad, kedy funkcia vráti nie správny výsledok.

```
def obsah_stvorca(strana):
    obsah = strana ** 2
    return obsah

dlzka_strany = input('Zadaj dĺžku strany štvorca: ')
dlzka_strany = float(dlzka_strany)

print(f'Obsah štvorca so stranou {dlzka_strany} je
{obsah_stvorca(dlzka_strany)}.')
```

Asi nebolo ťažké prísť na to, že problémom budú záporné čísla. Napriek tomu, že dĺžka strany nemôže byť záporná, naša funkcia obsah takéhoto štvorca vypočíta. Keby nás niekto požiadal, aby sme vypočítali obsah štvorca so stranou napr. -5, budeme protestovať, že to sa nedá. Podobne by mala zareagovať aj naša funkcia. Upravme ju nasledovne. Programový kód otestujte.

```
def obsah_stvorca(strana):
    if strana < 0:
        raise ValueError('Strana štvorca je záporná. Obsah sa
nedá vypočítať.')
    obsah = strana ** 2
    return obsah

dlzka_strany = input('Zadaj dĺžku strany štvorca: ')
dlzka_strany = float(dlzka_strany)

print(f'Obsah štvorca so stranou {dlzka_strany} je
{obsah_stvorca(dlzka_strany)}.')
```

Ak funkcia dostane záporné číslo (`if strana < 0:`), tak vygeneruje výnimku (`raise`). Výnimka zastaví vykonávanie programu. Súčasťou vygenerovanej výnimky je okrem jej typu (`ValueErrorr`) aj informácia o tom, prečo sa program zastavil (`Strana štvorca je záporná. Obsah sa nedá vypočítať.`)

Generovanie a odchytávanie výnimiek

Bežnou programátorskou praxou je, že niektoré časti programov výnimky generujú, zatiaľ čo iné časti programov výnimky odchytávajú. Program s funkciou `obsah_stvorca()` predčasne skončí (s chybou) pre záporné čísla. Upravme ho tak, aby v prípade záporného čísla program upozornil používateľa a pokračoval ďalej.

```
def obsah_stvorca(strana):
    if strana < 0:
        raise ValueError('Strana štvorca je záporná. Obsah sa nedá vypočítať.')
    obsah = strana ** 2
    return obsah

dlzka_strany = input('Zadaj dĺžku strany štvorca: ')
dlzka_strany = float(dlzka_strany)

try:
    print(f'Obsah štvorca so stranou {dlzka_strany} je {obsah_stvorca(dlzka_strany)}.')
except ValueError as chyba:
    print(chyba)
```

V predchádzajúcom programovom kóde sme v podstate spojili dve z predchádzajúcich častí dokopy: odchytávanie výnimiek a generovanie výnimiek. Je v podstate jedno, či výnimku cielene generujeme my alebo ju vygeneruje Python. Odchytávanie je v oboch prípadoch rovnaké.

Cvičenie 1

Vytvorili sme funkciu `obsah_obdlznika()` pre výpočet obsahu obdĺžnika. Upravte funkciu tak, aby v prípade nesprávnych vstupov funkcia vygenerovala výnimku.

```
def obsah_obdlznika(strana_a, strana_b):
    obsah = strana_a * strana_b
    return obsah

dlzka_strany_a = input('Zadaj dĺžku strany a: ')
dlzka_strany_a = float(dlzka_strany_a)
dlzka_strany_b = input('Zadaj dĺžku strany b: ')
dlzka_strany_b = float(dlzka_strany_b)
```

```
print(f'Obsah obdĺžnika so stranami {dlzka_strany_a} a
{dlzka_strany_b} je {obsah_obdlznika(dlzka_strany_a,
dlzka_strany_b)}.')
```

Cvičenie 2

Upravte predchádzajúci programový kód tak, aby v prípade nesprávnych vstupov program upozornil používateľa a pokračoval ďalej.

Cvičenie 3

Preskúmajte výsledný programový kód z predchádzajúceho cvičenia, nájdite a ošetrite ďalšie problematické situácie.

Viac o problematike výnimiek nájdete na:

- https://docs.python.org/3/tutorial/errors.html.

Viac o rôznych typoch výnimiek nájdete na:

- https://docs.python.org/3/library/exceptions.html?highlight=exception.

Vysvetlenie problematiky

Generovanie výnimiek

Výnimky štandardne generuje interpreter jazyka Python, ak sa vykonávanie programu dostane do situácie, v ktorej nie je možné pokračovať ďalej. Niektoré kroky, ktoré sú technicky realizovateľné, logicky nedávajú zmysel.

Napr. vypočítať hodnotu výrazu `5 - 7` nie je problém. Začleňme tento výraz do kontextu cestujúcich v autobuse. V autobuse cestuje 5 ľudí, pričom na zastávke z neho vystúpi 7 ľudí. Výraz `5 - 7` zaradený do tohto kontextu má úplne iný význam. Aj keď ho je možné matematicky vyhodnotiť, logicky je to nezmysel. Každý z nás by v tejto situácii zareagoval spôsobom: toto sa nedá, je to chyba, je to nezmysel a pod. Podobne by mal zareagovať aj program. Problém je, že interpreter jazyka nevie, že čísla 5 a 7 predstavujú počty ľudí.

Takúto reakciu musí zabezpečiť autor programu, napr. takto:

```
pocet_cestujucich = 5
pocet_vystupujucich = 7
if pocet_vystupujucich > pocet_cestujucich:
    raise ValueError('Nemôže vystúpiť viac ľudí ako je
v autobuse')
```

```
Traceback (most recent call last):
  File "...", line 4, in <module>
```

```
    raise ValueError('Nemôže vystúpiť viac ľudí ako je
v autobuse')
ValueError: Nemôže vystúpiť viac ľudí ako je v autobuse
```

Príkaz `raise` slúži na vygenerovanie výnimky. Vygenerovanie výnimky má aj tu vlastnosť, že táto chyba nemôže prejsť bez povšimnutia, pretože vykonávanie program sa ukončí predčasne. Okrem samotného vygenerovania výnimky môžeme spresniť o aký typ chyby ide (`ValueError`) a aj ju bližšie popísať. Takto náš programový kód poskytuje pomerne presnú informáciu, kde a aký problém nastal, a prečo nie je možné pokračovať ďalej. Podľa situácie môže programový kód generovať rôzne typy výnimiek (pozri. https://docs.python.org/3/library/exceptions.html#exception-hierarchy). Vyberme tú, ktorá je najbližšie k typu problému, pre ktorý výnimku vyhadzujeme.

Generovanie a odchytávanie výnimiek

Upozorniť na problematickú situáciu vygenerovaním výnimky je jedna vec a predčasné zastavenie programu druhá. Aj keď nejaká chyba nastane, vždy je možné sa z nej „zotaviť“ a aspoň v nejakom núdzovom režime pokračovať vo vykonávaní programu ďalej alebo program aspoň riadne ukončiť. Tak ako vieme odchytávať pythonovské výnimky, vieme odchytávať aj tie naše.

Nasledujúci program pre zadaný polomer gule vypíše jej objem. Funkcia na výpočet objemu kontroluje, či zadaná hodnota môže byť polomerom. Ak nie, vygeneruje výnimku aj s popisom problému. V hlavnom programe túto výnimku očakávame a zobrazíme upozornenie pre používateľa.

```
import math

def objem_gule(r):
    if r < 0:
        raise ValueError('Polomer nemôže byť záporný')
    objem = 4 / 3 * math.pi * r ** 3
    return objem

polomer = input('Zadaj polomer gule: ')
polomer = int(polomer)
try:
    objem = objem_gule(polomer)
except ValueError as chyba:
    print(chyba)
else:
    print(objem_gule(polomer))
```

```
Zadaj polomer gule: -5
Polomer nemôže byť záporný
```

Všimnime si, že vďaka dobrému popisu problému kvôli ktorému funkcia výnimku vygenerovala, môžeme tento popis priamo zobraziť používateľovi. Odchytenú výnimku (`ValueError`) si pomenujeme (`as chyba`) a priamo používateľovi vypíšeme (`print(chyba)`).

Prebublávanie výnimiek

Aj keď vieme, že nejaká časť programu môže generovať výnimky, nemusíme ju v zapätí odchytávať. Môžeme ju nechať „prebublať“ na vyššiu úroveň. Mali by sme si byť však istí tým, že na vyššej úrovni si s tým programový kód poradí.

Pozrime sa, čo sa stane, ak si s tým programový kód neporadí:

```
def funkcia_a():
    raise ValueError('Chyba v A')

def funkcia_b():
    funkcia_a()

funkcia_b()
```

```
Traceback (most recent call last):
  File "...", line 7, in <module>
    funkcia_b()
  File "...", line 5, in funkcia_b
    funkcia_a()
  File "...", line 2, in funkcia_a
    raise ValueError('Chyba v A')
ValueError: Chyba v A
```

Výnimku vygenerovala `funkcia_a()`. `funkcia_b()` na ňu nijako nereagovala a tak výnimka prebublala až do hlavného programu. Keďže sme ju neodchytili ani tam, neostalo nič iné, len zastaviť program a zobraziť chybovú správu. Ak by sme ju v hlavnom programe odchytili, výsledok by bol iný:

```
def funkcia_a():
    raise ValueError('Chyba v A')

def funkcia_b():
    funkcia_a()

try:
    funkcia_b()
except ValueError as chyba:
    print(chyba)
```

```
Chyba v A
```

Reťazenie výnimiek

Nasledujúci program pre zadané dĺžky strán trojuholníka vypočíta jeho obsah. Na výpočet obsahu sme použili Herónov vzorec. Riešenie vyzerá nasledovne:

```
import math

def obsah_trojuholnika(a, b, c):
    s = (a + b + c) / 2
    obsah = math.sqrt(s * (s - a) * (s - b) * (s - c))
    return obsah

dlzka1 = input('Zadaj dĺžku 1. strany trojuholníka: ')
dlzka1 = float(dlzka1)
dlzka2 = input('Zadaj dĺžku 2. strany trojuholníka: ')
dlzka2 = float(dlzka2)
dlzka3 = input('Zadaj dĺžku 3. strany trojuholníka: ')
dlzka3 = float(dlzka3)

obsah = obsah_trojuholnika(dlzka1, dlzka2, dlzka3)
print(f'Obsah trojuholníka: {obsah}')
```

Otestovali sme ho a program pracuje správne. Zaujímavá situácia nastane pre dĺžky strán napr. 1, 1 a 3.

```
Zadaj dĺžku 1. strany trojuholníka: 1
Zadaj dĺžku 2. strany trojuholníka: 1
Zadaj dĺžku 3. strany trojuholníka: 3

Traceback (most recent call last):
  File "...", line 15, in <module>
    obsah = obsah_trojuholnika(dlzka1, dlzka2, dlzka3)
  File "...", line 5, in obsah_trojuholnika
    obsah = math.sqrt(s * (s - a) * (s - b) * (s - c))
ValueError: math domain error
```

Počas výpočtu obsahu trojuholníka nastal nejaký problém, ale z chybového výpisu nie je celkom jasné, čo sa stalo a prečo sa to stalo. Používateľ programu alebo samotnej funkcie `obsah_trojuholnika()` teda nevie, kde a akú chybu spravil alebo či spravil chybu. K tejto situácii dochádza, ak nejaká nízkoúrovňová výnimka prebuble na vyššiu úroveň. Kým na nízkej úrovni výnimka dávala zmysel, na vyššej úrovni je už nezrozumiteľná. Pri lepšej analýze zistíme, že problém nastal pri odmocňovaní, kde sa pod odmocninou ocitlo záporné číslo. Spôsobené je to tým, že trojica čísiel 1, 1 a 3 nemôže predstavovať strany trojuholníka (neplatí trojuholníková nerovnosť). Funkcia `obsah_trojuholnika()` by mala na túto situáciu zareagovať a na vyššiu úroveň poslať zrozumiteľnú výnimku. Upravme ju nasledovne:

```
def obsah_trojuholnika(a, b, c):
    s = (a + b + c) / 2
    try:
        obsah = math.sqrt(s * (s - a) * (s - b) * (s - c))
    except ValueError:
        raise ValueError('Zadané dĺžky nemôžu byť dĺžkami
strán trojuholníka')
    return obsah
```

Po tejto úprave je správa pre používateľa už zrozumiteľná:

```
Traceback (most recent call last):
  File "...", line 6, in obsah_trojuholnika
    obsah = math.sqrt(s * (s - a) * (s - b) * (s - c))
ValueError: math domain error

During handling of the above exception, another exception
occurred:

Traceback (most recent call last):
  File " ...", line 18, in <module>
    obsah = obsah_trojuholnika(dlzka1, dlzka2, dlzka3)
  File "...", line 8, in obsah_trojuholnika
    raise ValueError('Zadane dlzky nemozu byt dlzkami stran
trojuholnika')
ValueError: Zadané dĺžky nemôžu byť dĺžkami strán
trojuholníka
```

Všimnime si, že vo výpise chyby sa okrem našej výnimky objavila aj pôvodná výnimka. Tomuto mechanizmu spájania výnimiek hovoríme reťazenie výnimiek.

Zbierka úloh

Nasledujúce úlohy sú zamerané na analýzu problémov, identifikáciu problematických situácií v navrhovaných riešeniach a ošetrenie chybových stavov.

1. Definujte funkciu `znamka_na_text()`, ktorá pre zadanú známku (1 až 5) vráti jej slovný ekvivalent (výborný až nedostatočný). Ak vstupná hodnota nie je korektná, nech funkcia generuje výnimku.

   Doplňte do funkcie dokumentačný reťazec (popis, význam a typ vstupných parametrov, výstup a typ výstupu).

2. Vytvorte program, ktorý si po spustení vypýta známku (1 až 5). Využitím funkcie `znamka_na_text()` prevedie a vypíšte slovný ekvivalent známky. Prípadné výnimky generované funkciou spracujte v hlavnom programe a zobrazte správu pre používateľa.

3. Definujte funkciu `pocet_autobusov()`, ktorá pre zadaný počet detí a kapacitu autobusu vráti počet autobusov potrebný pre odvezenie detí na výlet. V prípade nekorektného vstupu nech funkcia generuje výnimku.

   Mohla by funkcia akceptovať aj nečíselné vstupy?

4. Vytvorte program, ktorý si po spustení vypýta počet detí a kapacitu autobusu. Využitím funkcie `pocet_autobusov()` vypočítajte a vypíšte počet potrebných autobusov. Prípadné výnimky spracujte v hlavnom programe a zobrazte správu pre používateľa.

5. Máme nasledovnú špecifikáciu pre funkciu:

```
def obsah_kruhu(polomer):
    '''Pre zadaný polomer vráti obsah kruhu

    :param polomer: polomer kruhu, číslo alebo číselný
reťazec
    :type polomer: int | float | str
    :return: polomer kruhu
    :rtype: float
    :raises TypeError: ak polomer nie je konvertovateľný
na číslo
    :raises ValuError: ak je polomer menší ako 0
    '''
    pass
```

   Doplňte telo funkcie tak, aby vyhovovalo špecifikácii.

   Vytvorte program pre výpočet obsahu kruhu. Využite definovanú funkciu `obsah_kruhu()` a ošetrite (v hlavnom programe) prípadné generované výnimky.

6. Vytvorili sme funkciu pre výpočet vzdialenosti dvoch bodov v rovine. Implementovali a použili sme ju nasledovne:

```
def vzdialenost_bodov(x1, y1, x2, y2):
    '''Vypočíta vzdialenosť dvoch bodov v rovine

    :param x1: súradnica X prvého bodu, číslo alebo
číselný reťazec
    :type x1: int | float | str
    :param y1: súradnica Y prvého bodu, číslo alebo
číselný reťazec
    :type y1: int | float | str
    :param x2: súradnica X druhého bodu, číslo alebo
číselný reťazec
    :type x2: int | float | str
    :param y2: súradnica Y druhého bodu, číslo alebo
číselný reťazec
```

```
    :type y2: int | float | str
    :return: vzdialenosť bodov v rovine
    :rtype: float
    :raises TypeError: ak súradnice nie su korektné
    '''
    vzdialenost = ((x1 - x2) ** 2 + (y1 - y2) ** 2) **
0.5
    return vzdialenost

x1 = input('Súradnica x bodu 1: ')
y1 = input('Súradnica y bodu 1: ')
x2 = input('Súradnica x bodu 2: ')
y2 = input('Súradnica y bodu 2: ')

print(vzdialenost_bodov(vzdialenost_bodov(x1, y1, x2,
y2)))
```

Doplňte funkciu tak, aby akceptovala v dokumentačnom reťazci uvedené typy vstupov. Pre ostatné nech vygeneruje relevantnú výnimku.

Doplňte hlavný program tak, aby v prípade výnimky generovanej funkciou program neskončil s chybou, ale so zrozumiteľnou informáciou pre používateľa.

7. Vytvorte funkciu `rok_dochadzky()`, ktorá pre zadaný názov triedy gymnázia vráti, ktorý rok školskej dochádzky žiaka predstavuje. Názov triedy môže byť zadaný rôznymi spôsobmi, napr.: „1. A“, „2.B“, „prima A“, „oktáva D“, „oktáva A“. Ak je názov nekorektný, funkcia by mala vygenerovať relevantnú výnimku.

   Vytvorte hlavný program, ktorý si vypýta od používateľa názov triedy a vypíše rok školskej dochádzky. Prípadné výnimky ošetrite.

Riešené úlohy

1. Definujte funkciu `znamka_na_text()`, ktorá pre zadanú známku (1 až 5) vráti jej slovný ekvivalent (výborný až nedostatočný). Ak vstupná hodnota nie je korektná, nech funkcia generuje výnimku.

   Doplňte do funkcie dokumentačný reťazec (popis, význam a typ vstupných parametrov, výstup a typ výstupu).

   Riešenie:

   Uvažujme, že korektná známka môže byť zadaná ako číslo (napr. `2`) alebo ako reťazec (napr. `'4'`). Ak bude parameter iného typu, budeme to považovať za chybu a funkcia vygeneruje výnimku. Ak známka nebude niektorá z očakávaných, budeme o tiež považovať za chybu. Aby sme nemuseli

porovnávať hodnotu známky s číslom a reťazcom zároveň, pretypujeme ju na reťazec. Výsledná verzia funkcie môže vyzerať nasledovne:

```
def znamka_na_text(znamka):
    ''' Prevedie znamku (1 až 5) na slovný ekvivalent

    :param znamka: známka, 1 až 5
    :type znamka: int | str
    :return: slovný ekvivalent známky
    :rtype: str
    :raises TypeError: Ak typ znamka nie je int alebo str
    :raises ValueError: Ak vstupná hodnota je rôzna od 1,
2, 3, 4, 5
    '''

    if not isinstance(znamka, (int, str)):
        raise TypeError('Nekorektný typ pre známku')
    znamka = str(znamka)
    if znamka == '1':
        return 'výborný'
    elif znamka == '2':
        return 'chválitebný'
    elif znamka == '3':
        return 'dobrý'
    elif znamka == '4':
        return 'dostatočný'
    elif znamka == '5':
        return 'nedostatočný'
    else:
        raise ValueError('Nekorektná známka')
```

Všimnime si dokumentačný reťazec funkcie. Je uvedený ako viacriadkový text ihneď za hlavičkou funkcie. Sú v ňom uvedené všetky potrebné informácie pre použitie tejto funkcie. Ak si zobrazíme nápoveď pre túto funkciu, vývojové prostredie dokáže z dokumentačného reťazca vygenerovať (napr. v PyCharm-e klávesovou skratkou Ctrl + Q) nasledovný výstup:

```
def znamka_na_text(znamka: Union[int, str]) -> str

Prevedie znamku (1 .. 5) na slovny ekvivalent

Params:  znamka – znamka 1 .. 5

Returns: slovny ekvivalent znamky

Raises:  TypeError – Ak typ znamka nie je int alebo str
         ValueError – Ak vstupna hodnota je rozna od 1, 2, 3, 4, 5
```

2. Vytvorte program, ktorý si po spustení vypýta známku (1 až 5). Využitím funkcie `znamka_na_text()` prevedie a vypíšte slovný ekvivalent známky. Prípadné výnimky generované funkciou spracujte v hlavnom programe a zobrazte správu pre používateľa.

   Riešenie:

   Pri riešení väčších problémov je vhodné výsledný programový kód logicky rozdeliť do samostatných súborov – modulov. Predpokladajme, že pre manipuláciu so známkami sme si vytvorili viacero užitočných funkcií a tie sme spolu s funkciou `znamka_na_text()` umiestnili do súboru `klasifikacia.py`. Riešenie tejto úlohy realizujme v samostatnom súbore.

```
import klasifikacia

znamka = input('Zadaj známku (1 až 5): ')
try:
    slov_znamka = klasifikacia.znamka_na_text(znamka)
except (ValueError, TypeError) as chyba:
    print(chyba)
else:
    print(slov_znamka)
```

   Všimnime si niekoľko benefitov celého riešenia:

   - Presunom programového kódu do pomocného modulu, je hlavný program krátky a prehľadný.
   - Vďaka dokumentačnému reťazcu funkcie `znamka_na_text()` vieme, čo funkcia robí a ako sa správa aj bez toho, aby sme ju fyzicky skúmali. Dokumentáciu funkcie si vieme zobraziť aj mimo súboru `klasifikacia.py`.
   - Funkcia `znamka_na_text()` generuje zrozumiteľné správy pri vyhadzovaní výnimiek. Tie môžeme, v prípade problémov, priamo zobraziť používateľovi.
   - Kontrolu vstupov prevádzame len vo funkcii. V hlavnom programe to vôbec neriešime. Výhodou tohto prístupu je, že funkcia funguje „autonómne“ nech ju prenesiete kamkoľvek. Nespolieha sa na to, že nejakú kontrolu už niekto pred jej volaním vykonal.

10 Zoznamy a metódy zoznamov

Jemný úvod do problematiky

Použite notebook **Python 10 - Zoznamy a metódy zoznamov.ipynb** alebo niektoré z lokálnych vývojových prostredí jazyka Python.

Cieľ: *V tomto notebooku si predstavíme dátovú štruktúru zoznam a metódy pre prácu so zoznamom.*

Dátová štruktúra zoznam

Doteraz sme používali jednoduché dátové typy: `int`, `float`, `boolean` a `string` (i keď `string` môžeme chápať aj ako dátovú štruktúru – postupnosť znakov). Existuje však množstvo situácií, kedy si s nimi nevystačíme.

Ako by sme reprezentovali známky žiaka: 1, 3, 2? Napríklad takto?:

```
# známky žiaka
znamka1 = 1
znamka2 = 3
znamka3 = 2

print(znamka1)
print(znamka2)
print(znamka3)
```

Toto riešenie je však problematické. Ak by žiak dostal aj štvrtú známku, musíme zaviesť štvrtú premennú. Samotné dáta teda vplývajú na programový kód programu, čo je nežiadúce. Prakticky je toto riešenie nepoužiteľné. Existuje aj šikovnejšie riešenie. Vytvorme si zoznam známok:

```
# známky žiaka
znamky = [1, 3, 2]
print(znamky)

# alebo

for znamka in znamky:
    print(znamka)
```

Ak potrebujeme žiakovi pridať ďalšiu známku (napr. 4), upravíme zoznam známok. Novú známku môžeme vložiť na koniec zoznamu.

```
# známky žiaka
znamky.append(4)
print(znamky)

# alebo
```

```
for znamka in znamky:
    print(znamka)
```

Zoznam je kontajner pre dáta, v ktorej majú dáta svoje poradie a jeho obsah je možné modifikovať. Zoznam vytvoríme jednoducho tak, že hodnoty alebo premenné uzatvoríme do hranatých zátvoriek. Dáta v zozname nie sú obmedzené žiadnym typom. V jednom zozname môžeme uchovávať údaje rôznych typov.

Určite ste si všimli istú podobnosť s reťazcami pri výpise obsahu zoznamu pomocou cyklu. Podobností tu je však viac:

```
# porovnanie reťazcov a zoznamov
retazec = 'znaky'
zoznam = ['z', 'n', 'a', 'k', 'y']

# indexovanie
print(retazec[2])
print(zoznam[2])

# výrezy
print(retazec[1:4])
print(zoznam[1:4])
```

```
# porovnanie reťazcov a zoznamov
retazec = 'znaky'
zoznam = ['z', 'n', 'a', 'k', 'y']

# prechod cez
for znak in retazec:
    print(znak)
for prvok in zoznam:
    print(prvok)
```

```
# porovnanie reťazcov a zoznamov
retazec = 'znaky'
zoznam = ['z', 'n', 'a', 'k', 'y']

# test na prítomnosť hodnoty
print('a' in retazec)
print('a' in zoznam)

# spájanie
print('zna' + 'ky')
print(['z', 'n', 'a'] + ['k', 'y'])
```

Napriek množstvu podobností existujú medzi reťazcami a zoznamami podstatné rozdiely:

- reťazec môže obsahovať len znaky, zoznam môže obsahovať hodnoty rôznych typov,
- reťazec je nemenný, zoznam je možné modifikovať (viď. nasledujúca časť).

Cvičenie 1

Napíšte programový kód, ktorý vypíše prvky zoznamu odzadu.

Metódy zoznamov

Zoznamy nám ponúkajú pomerne veľa funkcií (metód), pomocou ktorých s nimi vieme pracovať. Čo robia nasledovné metódy? Svoje predpoklady si overte zmenou parametrov metód.

```
zoznam = [1, 2, 3, 2, 1]
zoznam.append(0)
zoznam.insert(2, 10)
zoznam.sort()
zoznam.pop(2)
zoznam.remove(2)
zoznam.clear()
zoznam.index(2)
zoznam.count(1)
```

Viac informácií o metódach zoznamov nájdeme na https://docs.python.org/3/tutorial/datastructures.html#more-on-lists.

Cvičenie 2

Zoznam známok žiaka obsahuje (v rôznom počte) hodnoty 1, 2, 3, 4, 5 a 'n'. Vytvorte funkciu `uprav_znamky()`, ktorá zoznam známok upraví tak, že hodnotu 'n' nahradí známkou 5.

Napríklad:

`uprav_znamky([1, 1, 'n', 3, 'n'])` → `[1, 1, 5, 3, 5]`

Cvičenie 3

Vytvorte funkciu `vysledna_znamka()`, ktorá pre zadaný zoznam známok vráti výslednú známku žiaka. Výslednú známku získame zaokrúhlením priemeru známok.

Pomôcka: Možno vám pomôžu funkcie `sum(zoznam)` a `len(zoznam)`.

Vysvetlenie problematiky

Dátová štruktúra zoznam

Zoznam je dátová štruktúra a definujeme ju ako postupnosť hodnôt uzatvorených v hranatých zátvorkách a oddelených čiarkou, napr.:

```
[11, 21, 22, 12]
['tri', 'dva', 'jeden', 'štart']
```

Zoznamy môžeme vytvárať aj z iných iterovateľných štruktúr, napr. z reťazcov:

```
zoznam_pismen = list('ahoj')
print(zoznam_pismen)

zoznam_cisiel = list(range(10))
print(zoznam_cisiel)
```

```
['a', 'h', 'o', 'j']
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Ak použijeme funkciu `list()` a nezadáme žiaden argument, výsledkom je prázdny zoznam. Vytvoriť prázdny zoznam však vieme aj rýchlejšie: `[]`.

Zoznamy sa sémanticky používajú primárne ako homogénne dátové štruktúry, to znamená, že obsahujú hodnoty rovnakého typu. Technicky zoznam môže obsahovať hodnoty rôznych typov, ale v tomto prípade je vhodnejšie tieto hodnoty umiestniť do inej dátovej štruktúry (napr. n-tica). Zoznamy využívame v situáciách, keď potrebujeme pracovať s nejakým, často vopred neznámym počtom hodnôt rovnakého typu.

Zoznamy vieme vytvárať aj spájaním iných zoznamov pomocou operácie `+`:

```
zoznam1 = ['a', 'h']
zoznam2 = ['o', 'j']
zoznam = zoznam1 + zoznam2
print(zoznam)
```

```
['a', 'h', 'o', 'j']
```

Indexy v zoznamoch

K jednotlivým prvkom zoznamov pristupujeme cez ich indexy, podobne ako tomu bolo pri reťazcoch:

indexy zľava:		0	1	2	3	4	5	6	
zoznam:	[	'p'	'o'	'č'	'í'	't'	'a'	'č'	]
indexy sprava:		-7	-6	-5	-4	-3	-2	-1	

```
zoznam = ['p', 'o', 'č', 'í', 't', 'a', 'č']
print(zoznam[1])
print(zoznam[-2])
print(zoznam[15])
```

```
o
a

Traceback (most recent call last):
  File "...", line 4, in <module>
    print(zoznam[15])
IndexError: list index out of range
```

Všimnime si, že aj v tomto prípade pokus o prístup k prvku na neexistujúcom indexe vedie ku chybe.

Prechod cez prvky zoznamov

Zoznam je iterovateľná štruktúra. Vďaka tomu môžeme prechádzať cez prvky zoznamu priamo v cykle:

```
zoznam = ['a', 'h', 'o', 'j']
for prvok in zoznam:
    print(prvok)
```

```
a
h
o
j
```

K prvkom zoznamu môžeme pristupovať aj cez ich indexy. Ak potrebujeme z nejakého dôvodu pracovať nielen s prvkami zoznamu, ale aj s ich indexami, môžeme prechod realizovať po mocou indexov:

```
zoznam = ['a', 'h', 'o', 'j']
for idx in range(len(zoznam)):
    print(f'index: {idx}, hodnota prvku: {zoznam[idx]}')
```

```
index: 0, hodnota prvku: a
index: 1, hodnota prvku: h
index: 2, hodnota prvku: o
index: 3, hodnota prvku: j
```

V ukážke sme použili funkciu `len()`, ktorá v tomto prípade vracia dĺžku (počet prvkov) zoznamu.

Operátor príslušnosti

Ak potrebujeme otestovať prítomnosť nejakého prvku v zozname, využijeme operátor príslušnosti `in`:

```
print('h' in ['a', 'h', 'o', 'j'])
print('ahoj' in ['a', 'h', 'o', 'j'])
```

```
True
False
```

Výrezy

Aj pri zoznamoch vieme použiť výrezy. Výrezy zoznamov fungujú rovnako ako výrezy reťazcov. Ich výsledkom je však zoznam:

```
zoznam = ['p', 'o', 'č', 'í', 't', 'a', 'č']
print(zoznam[:3])  # ['p', 'o', 'č']
print(zoznam[3:])  # ['í', 't', 'a', 'č']
print(zoznam[1:4])  # ['o', 'č', 'í']
print(zoznam[2:7:2])  # ['č', 't', 'č']
```

```
['p', 'o', 'č']
['í', 't', 'a', 'č']
['o', 'č', 'í']
['č', 't', 'č']
```

Metódy zoznamov

Podobne ako pri reťazcoch, tak aj pri zoznamoch môžeme využiť ich metódy. Keďže zoznamy sú na rozdiel od reťazcom meniteľné, je tu podstatný rozdiel. Ak metóda modifikuje zoznam, zmena sa prejaví na zozname, na ktorom sme metódu zavolali. Tieto metódy teda nevytvárajú nový, zmenený zoznam.

<table>
<tr><td><code>append(hodnota)</code></td><td>vloží hodnotu na koniec zoznamu<pre>z = [1, 2, 3, 2, 1]
z.append(0)
print(z)</pre><pre>[1, 2, 3, 2, 1, 0]</pre></td></tr>
<tr><td><code>insert(index, hodnota)</code></td><td>vloží hodnotu pred zadaný index<pre>z = [1, 2, 3, 2, 1]
z.insert(2, 10)
print(z)</pre><pre>[1, 2, 10, 3, 2, 1]</pre></td></tr>
</table>

<table>
<tr><td><code>sort()</code></td><td>usporiada prvky zoznamu vzostupne<pre>z = [1, 2, 3, 2, 1]
z.sort()
print(z)</pre><pre>[1, 1, 2, 2, 3]</pre></td></tr>
<tr><td><code>pop(index)</code></td><td>vyberie a vráti hodnotu na zadanom indexe, ak index neexistuje, generuje výnimku<pre>z = [1, 2, 3, 2, 1]
print(z.pop(2))
print(z)</pre><pre>3
[1, 2, 2, 1]</pre></td></tr>
<tr><td><code>remove(hodnota)</code></td><td>odstráni prvý výskyt hodnoty v zozname, ak sa hodnota v zozname nenachádza, generuje výnimku<pre>z = [1, 2, 3, 2, 1]
z.remove(2)
print(z)</pre><pre>[1, 3, 2, 1]</pre></td></tr>
<tr><td><code>clear()</code></td><td>odstráni všetky prvky zo zoznamu<pre>z = [1, 2, 3, 2, 1]
z.clear()
print(z)</pre><pre>[]</pre></td></tr>
<tr><td><code>extend(struktura)</code></td><td>pridá do zoznamu hodnoty z danej štruktúry (napr. reťazec, zoznam, range)<pre>z = ['a', 'h']
z.extend('oj')
print(z)</pre><pre>['a', 'h', 'o', 'j']</pre></td></tr>
<tr><td><code>reverse()</code></td><td>zmení poradie prvkov zoznamu<pre>z = [1, 2, 3]
z.reverse()
print(z)</pre><pre>[3, 2, 1]</pre></td></tr>
</table>

<table>
<tr><td><code>index(hodnota)</code></td><td>vráti prvý index zoznamu na ktorom sa nachádza zadaná hodnota, ak hodnota v zozname neexistuje, generuje výnimku<pre>z = [1, 2, 3, 2, 1]
print(z.index(2))
print(z)</pre><pre>1
[1, 2, 3, 2, 1]</pre></td></tr>
<tr><td><code>count(hodnota)</code></td><td>vráti počet výskytov hodnoty v zozname<pre>z = [1, 2, 3, 2, 1]
print(z.count(1))
print(z)</pre><pre>2
[1, 2, 3, 2, 1]</pre></td></tr>
</table>

Okrem samotných metód zoznamov môžeme využívať aj štandardné funkcie jazyka Python. Tieto funkcie zoznam nemodifikujú, ale vracajú nejakú informáciu o zozname.

`len()`	vráti počet prvkov zoznamu `len([2, 4, 6])` → `3`
`min()`	vráti najmenšiu hodnotu zo zoznamu, ak je zoznam prázdny, generuje výnimku `min([2, 4, 6])` → `2`
`max()`	vráti najväčšiu hodnotu zo zoznamu, ak je zoznam prázdny, generuje výnimku `max([2, 4, 6])` → `6`
`sum()`	vráti súčet prvkov v zozname, prvky musia byť čísla `sum([2, 4, 6])` → `12`

Zbierka úloh

Nasledujúce úlohy sú zamerané na dátovú štruktúru zoznam a manipuláciu so zoznamami a s prvkami zoznamov.

1. Definujte funkciu `pis_znamky()`, ktorá vypíše známky zo zadaného zoznamu známok.
   - Uvažujte výpis, kde každá známka bude v samostatnom riadku.
   - Uvažujte výpis, kde všetky známky budú v jednom riadku.

2. Upravte predchádzajúci funkciu `pis_znamky()` tak, aby funkcia vypísala len korektné známky (1 až 5). Za korektnú známku považujte aj známku v tvare reťazca (napr. `'2'`).

3. Teplota ovzdušia sa meria každý deň 3-krát. O 7^{00}, o 14^{00} a o 21^{00}. Priemerná denná teplota sa vypočíta podľa vzťahu: $t_{denná} = (t_7 + t_{14} + 2 * t_{21}) / 4$. Definujte funkciu `priemerna_teplota()`, ktorá pre zoznam teplôt z jedného dňa vráti dennú priemernú teplotu.

4. V zozname `obraty` sú zaznamenané pohyby na účte v banke počas mesiaca. Prvé číslo v zozname predstavuje zostatok z predchádzajúceho mesiaca. Ďalšie čísla predstavujú vklady (kladné hodnoty) a výbery (záporné hodnoty) počas mesiaca. Vytvorte funkcie:
   - `zostatok()` – funkcia vráti hodnotu nového zostatku na konci mesiaca,
   - `celkovy_vklad()` – funkcia vráti celkový vklad na účet počas mesiaca,
   - `pocet_vkladov()` – funkcia vráti celkový počet vkladov na účet,
   - `celkový_vyber()` – funkcia vráti celkový výber počas mesiaca,
   - `pocet_vyberov()` – funkcia vráti celkový počet výberov z účtu,
   - `bolo_precerpanie()` – funkcia vráti, či nastal okamih, kedy sme prečerpali účet (išli sme do mínusu),
   - `vypis_transakcie()` – funkcia vypíše všetky pohyby na účte aj s ich poradím,
   - `je_korektny()` – funkcia vráti, či zoznam obratov je korektný (prediskutujte, čo musí pre prvky zoznamu platiť).

   Upravte niektorú z vytvorených funkcií tak, aby v prípade nekorektného zoznamu obratov vygenerovala relevantnú výnimku. Upravte hlavný program tak, aby túto výnimku odchytil a vypísal korektné upozornenie.

5. Definujte funkciu `vysledna_znamka()`, ktorá pre zoznam známok z nejakého predmetu vráti výslednú známku. Predpokladajte, že všetky známky sú korektné.

   Výsledná známka sa vypočíta ako zaokrúhlený priemer všetkých známok, pričom hodnota x.5 sa zaokrúhľuje smerom dole.

   `vysledna_znamka([2, 1, 2, 4, 3, 2, 5, 1])` → `2`

6. Vytvorte funkciu `maturitna_znamka()`, ktorá dostane zoznam zoznamov známok z hlavného a z príbuzných predmetov a vráti maturitnú známku. Výsledná maturitná známka sa vypočíta ako priemer výsledných známok z jednotlivých predmetov.

Napr.:

známky z informatiky: 2, 1, 2,

známky z cvičení z informatiky: 4, 3,

známky zo seminára z informatiky: 2, 5, 1,

`maturitna_znamka([[2, 1, 2], [4, 3], [2, 5, 1]])` → `3`

7. Upravte predchádzajúce funkcie tak, aby v prípade nekorektnej známky funkcie vygenerovali výnimku (korektné známky sú 1 až 5).

8. Žiaci triedy cestujú na športové hry.

   V zozname `ziaci` je zoznam všetkých žiakov triedy.

   V zozname `sportovci` je zoznam tých žiakov triedy, ktorí trénujú a chceli by súťažiť.

   V zozname `neprospievajuci` je zoznam tých žiakov triedy, ktorí neprospievajú z nejakého predmetu.

   Definujte nasledovné funkcie (premyslite si vhodné parametre):

   - `nesportovci()` – vráti zoznam žiakov triedy, ktorí nie sú športovci,
   - `neprospievajuci_sportovci()` – vráti zoznam športovcov, ktorí neprospievajú,
   - `divaci()` – vráti zoznam divákov, súťažia len prospievajúci športovci, ostatní sú diváci.

Riešené úlohy

5. Definujte funkciu `vysledna_znamka()`, ktorá pre zoznam známok z nejakého predmetu vráti výslednú známku. Predpokladajte, že všetky známky sú korektné.

   Výsledná známka sa vypočíta ako zaokrúhlený priemer všetkých známok, pričom hodnota x.5 sa zaokrúhľuje smerom dole.

   `vysledna_znamka([2, 1, 2, 4, 3, 2, 5, 1])` → `2`

   Riešenie:

   Algoritmicky nie je riešenie úlohy náročné. Priemer známok vypočítame ako ich súčet predelený ich počtom. Využiť môžeme štandardné funkcie jazyka Python `sum()` a `len()`. Situáciu trochu komplikuje zaokrúhľovanie hodnôt, ktorých desatinná časť je rovná 0,5. Na zistenie toho, či tento stav nastal a prípadné zaokrúhlenie využijeme už známe aritmetické operácie zvyšok po delení `%` a celočíselné delenie `//`.

```
def vysledna_znamka(znamky):
    priemer = sum(znamky) / len(znamky)
    if priemer % 1 == 0.5:
        znamka = priemer // 1
    else:
        znamka = round(priemer)
    return znamka
```

6. Vytvorte funkciu `maturitna_znamka()`, ktorá dostane zoznam zoznamov známok z hlavného a z príbuzných predmetov a vráti maturitnú známku. Výsledná maturitná známka sa vypočíta ako priemer výsledných známok z jednotlivých predmetov.

   Napr.:

   známky z informatiky: 2, 1, 2,

   známky z cvičení z informatiky: 4, 3,

   známky zo seminára z informatiky: 2, 5, 1.

   `maturitna_znamka([[2, 1, 2], [4, 3], [2, 5, 1]])` → `3`

   Riešenie:

   Najskôr potrebujeme vypočítať výsledné známky z jednotlivých predmetov. Na to môžeme využiť už definovanú funkciu z predchádzajúcej úlohy `vysledna_znamka()`. Výsledkom tohto kroku bude zoznam výsledných známok z jednotlivých predmetov. Ak funkciu `vysledna_znamka()` použijeme na tento zoznam, výsledkom bude celková výsledná maturitná známka.

```
def maturitna_znamka(zoznamy_znamok):
    znamky_predmety = []
    for predmet in zoznamy_znamok:
        znamka_predmet = vysledna_znamka(predmet)
        znamky_predmety.append(znamka_predmet)
    return vysledna_znamka(znamky_predmety)
```

   Všimnime si, že v tomto prípade pracujeme so zoznamom zoznamov (`zoznamy_znamok`). V cykle prechádzame cez prvky tohto zloženého zoznamu. Každý prvok predstavuje jednoduchý zoznam známok z konkrétneho predmetu.

7. Upravte predchádzajúce funkcie tak, aby v prípade nekorektnej známky funkcie vygenerovali výnimku (korektné známky sú 1 až 5).

   Riešenie:

   Zamyslime sa, do ktorej funkcie kontrolu pridať. Funkcia `maturitna_znamka()` zo známkami priamo nepracuje. Manipuluje len so zoznamami známok.

Kontrolu korektnosti známok by sme mohli implementovať do funkcie `vysledna_znamka()`. Presnejšie povedané, na kontrolu korektnosti známok si vytvoríme pomocnú funkciu, ktorú budeme volať z funkcie `vysledna_znamka()` pred samotným spracovaním známok.

Podľa zadania by sme mali kontrolovať korektnosť známok ako takých. Ak si všimneme výpočet výslednej známky, spoliehame sa na to, že zoznam známok obsahuje aspoň jednu známku. Inak by nastalo delenie 0. Doplňme preto aj kontrolu na prázdny zoznam. Nemusíme priamo kontrolovať počet prvkov, stačí otestovať či výpočet priemeru zbehol bez problémov.

```
def su_korektne(znamky):
    for znamka in znamky:
        if znamka not in [1, 2, 3, 4, 5]:
            return False
    return True


def vysledna_znamka(znamky):
    if not su_korektne(znamky):
        raise ValueError('Nekorektná hodnota známky')
    try:
        priemer = sum(znamky) / len(znamky)
    except ZeroDivisionError:
        raise ValueError('Prázdny zoznam známok')
    if priemer % 1 == 0.5:
        znamka = priemer // 1
    else:
        znamka = round(priemer)
    return znamka
```

Všimnime si, že funkcia `su_korektne()` žiadnu výnimku nevyhadzuje. Ako názov funkcie napovedá, jej odpoveďou je logická hodnota `True` alebo `False`.

11 Algoritmy na zoznamoch

Jemný úvod do problematiky

Použite notebook **Python 11 - Algoritmy na zoznamoch.ipynb** alebo niektoré z lokálnych vývojových prostredí jazyka Python.

Cieľ: *V tomto notebooku si ukážeme ako efektívnejšie vytvárať zoznamy a ako využívať zoznamy pri riešení náročnejších algoritmických úloh.*

Generátorová notácia zoznamu

Požiadavka na vytvorenie zoznamu alebo na výber prvkov z nejakého iného zoznamu podľa daných kritérií je pomerne častá. Ukážme si, ako tieto činnosti realizovať šikovnejšie.

Problém: V zozname `pohyby` je zoznam pohybov na účte tak, ako ich zaznamenal bankový systém. Kladné čísla predstavujú prichádzajúce a záporné čísla odchádzajúce platby. Niektoré čísla však obsahujú príliš veľa desatinných miest. Vytvorme nový zoznam, kde budú čísla zaokrúhlené na dve desatinné miesta.

Riešenie, ktoré nás napadne ako prvé, môže vyzerať nasledovne:

```
pohyby = [129.9999, -24.99999, 1280.00013, -88.000001]

upravene_pohyby = []
for pohyb in pohyby:
    upravene_pohyby.append(round(pohyb, 2))

print(upravene_pohyby)
```

```
[130, -25.0, 1280, -88.0]
```

Existuje aj šikovnejší spôsob ako dosiahnuť to isté:

```
pohyby = [129.9999, -24.99999, 1280.00013, -88.000001]

upravene_pohyby = [round(pohyb, 2)
                   for pohyb in pohyby]

print(upravene_pohyby)
```

```
[130, -25.0, 1280, -88.0]
```

Takémuto „skrátenému“ zápisu pre vytvorenie zoznamu hovoríme generátorová notácia zoznamu (angl. list comprehension). Vo všeobecnosti má tento zápis tvar:

```
[výraz
 for položka in iterovateľná_štruktúra]
```

Problém: V zozname `pohyby` je zoznam pohybov na účte tak, ako ich zaznamenal bankový systém. Kladné čísla predstavujú prichádzajúce a záporné čísla odchádzajúce platby. Niektoré čísla však obsahujú príliš veľa desatinných miest. Vytvorme nový zoznam, kde budú len čísla reprezentujúce príjmy a navyše zaokrúhlené na dve desatinné miesta.

Riešenie, ktoré nás napadne ako prvé, môže vyzerať nasledovne:

```
pohyby = [129.9999, -24.99999, 1280.00013, -88.000001]

upravene_prijmy = []
for pohyb in pohyby:
    if pohyb > 0:
        upravene_prijmy.append(round(pohyb, 2))

print(upravene_prijmy)
```

Ukážme si riešenie využitím generátorovej notácie zoznamu:

```
pohyby = [129.9999, -24.99999, 1280.00013, -88.000001]

upravene_prijmy = [round(pohyb, 2)
                   for pohyb in pohyby
                   if pohyb > 0]

print(upravene_prijmy)
```

Vo všeobecnosti má tento zápis s podmienkou tvar:

```
[výraz
 for položka in iterovateľná_štruktúra
 if logický_výraz].
```

Aj keď generátorová notácia skracuje zápis, nie vždy je vhodné ju použiť. Ak by tento skrátený zápis bol na úkor zrozumiteľnosti programu, zvážme jeho použitie.

Cvičenie 1

V zozname `cisla` je zoznam čísiel. Vytvorte zoznam `mocniny`, v ktorom budú len tie čísla zo zoznamu `cisla`, ktoré sú druhou mocninou nejakého prirodzeného čísla.

```
cisla = [1, 2, 25, 25.0]
# dokončite program podľa zadania

mocniny =
```

Manipulácia so zoznamami

Pri manipulácii so zoznamami budeme využívať nasledovné:

- zoznamy sú meniteľné štruktúry,
- prístup k prvkom zoznamov cez index,
- extrahovanie podzoznamov zo zoznamov pomocou výrezov,
- prechod prvkami zoznamu, či už priamo alebo pomocou indexov,
- metódy zoznamov.

Zoznamy sú pravdepodobne najpoužívanejšou a najuniverzálnejšou dátovou štruktúrou v jazyku Python. Vďaka svojim vlastnostiam a metódam ich môžeme využiť v množstve prípadov.

Treba ale povedať aj to, že samotná skutočnosť, že zoznam sa dá pri riešení problému použiť, ešte nezaručuje, že neexistuje aj lepšie riešenie.

Problém: Denná priemerná teplota sa vypočíta nasledovne:

- Teplota v daný deň sa odmeria 3-krát, o 7^{00}, o 14^{00} a o 21^{00}.
- Priemerná denná teplota sa vypočíta podľa vzťahu $(t_7 + t_{14} + 2t_{21}) / 4$.

V zozname teploty je záznam meraní za niekoľko dní. Vytvorme funkciu `vypocitaj_priemerne_teploty()`, ktorá pre takýto zoznam vráti zoznam priemerných denných teplôt.

Napr. `[[15, 25, 14], [17, 28, 20]]` → `[17, 21.25]`

```
def priemerne_denne_teploty(teploty):
    priemerne_teploty = []
    for den in teploty:
        priemerna_teplota = (sum(den) + den[-1]) / 4
        priemerne_teploty.append(priemerna_teplota)
    return priemerne_teploty

teploty = [[15, 25, 14], [17, 28, 20]]
print(priemerne_denne_teploty (teploty))
```

```
[17.0, 21.25]
```

Cvičenie 2

Upravte predchádzajúce riešenie využitím generátorovej notácie zoznamu.

Cvičenie 3

V predchádzajúcich riešeniach predpokladáme, že máme kompletné záznamy z každého dňa. Záznam z prvého a z posledného dňa však nemusí byť úplný. Upravte funkciu `priemerne_denne_teploty` tak, aby v prípade nekompletného záznamu dňa priemer teplôt pre tento deň nepočítala.

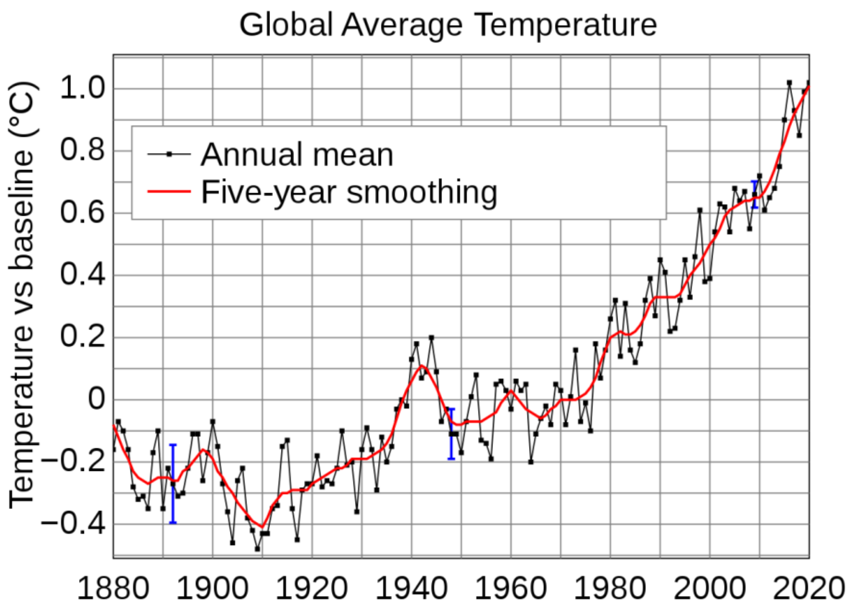
Cvičenie 4

Predchádzajúca funkcia nám vrátila priemerné teploty pre zadané dni merania. Takto upravené hodnoty sú však problematické pri analýze trendov (ako sa teplota bude vyvíjať), pretože lokálne extrémy tieto trendy skrývajú. Preto sa takéto dáta vyhladzujú. Možností je niekoľko. My použijeme postup, keď pre každú susednú trojicu hodnôt vypočítame jej priemer (trojdňový kĺzavý priemer). Napríklad:

`[17, 16, 25, 18, 18, 10, 17]` → `[19.3, 19.7, 20.3, 15.3, 15.0]`

Vytvorte funkciu `vyhlad_postupnost()`, ktorá zadanú postupnosť vyhladí podľa vyššie uvedeného postupu.

Pre lepšiu predstavu významu vyhladzovania uvádzame nasledovný obrázok. Obrázok zachytáva globálnu priemernú ročnú teplotu (čierne bodky) vyhladenú na 5-ročné intervaly (červená čiara). Všimnime si, že lokálne extrémy sa po vyhladení stratia, pričom začína byť lepšie viditeľný trend.



Zdroj: https://commons.wikimedia.org/w/index.php?title=File:Global_Temperature_Anomaly.svg&oldid=567175032

Cvičenie 5

Upravte predchádzajúcu funkciu tak, aby jej parametrom bola aj dĺžka intervalu pre vyhladzovanie (v predchádzajúcej funkcii bola konštantne 3).

Vysvetlenie problematiky

Viac mien pre jeden zoznam

Z úvodnej časti tohto kurzu vieme, že premenná je pomenovanie hodnoty. Pridajme k tomu informáciu, že zoznam je meniteľný typ a pozrime sa na to, aké to môže mať dôsledky.

```
zoznam1 = [1, 2, 3]
zoznam2 = zoznam1
zoznam2.append(4)
print(zoznam1)
```

```
[1, 2, 3, 4]
```

V tomto prípade cez dve rôzne mená (`zoznam1` a `zoznam2`) pristupujeme k jednej a tej istej hodnote `[1, 2, 3]`. Zmena cez meno `zoznam2` sa prejaví aj keď k hodnote pristupujeme cez meno `zoznam1`. Sú situácie, keď takýto dvojitý prístup nie je takto priamo viditeľný a môže nás zaskočiť.

Vytvorili sme si funkciu, ktorá vráti 3 najväčšie hodnoty v zadanom zozname.

```
def tri_maxima(zoznam):
    maxima = []
    for i in range(3):
        maximum = max(zoznam)
        maxima.append(maximum)
        zoznam.remove(maximum)
    return maxima

z = [5, 1, 4, 2, 3]
print(tri_maxima(z))
print(z)
```

```
[5, 4, 3]
[1, 2]
```

Vo funkcii sme si situáciu uľahčili tým, že nájdené maximá zo zoznamu odstraňujeme, aby sme ich nenašli viackrát. Všimnime si, že zoznam `z`, ktorý sme si v hlavnom programe vytvorili sa po volaní funkcie zmenil. Toto nie je žiaduce a môže to viesť k ťažko odhaliteľným chybám. Nečakáme, že funkcia `tri_maxima()` zmení zadaný zoznam. Čakáme, že vráti tri najväčšie čísla nič viac.

Funkcie, pokiaľ to od nich vyslovene neočakávame, by nemali meniť hodnoty parametrov meniteľných typov. Ak potrebujeme pre výpočet nejakú zmenu hodnoty parametra zrealizovať, vykonajme ju na kópii.

```
def tri_maxima(zoznam):
    zoznam_tmp = zoznam.copy()
    maxima = []
    for i in range(3):
        maximum = max(zoznam_tmp)
        maxima.append(maximum)
        zoznam_tmp.remove(maximum)
    return maxima

z = [5, 1, 4, 2, 3]
print(tri_maxima(z))
print(z)
```

```
[5, 4, 3]
[5, 1, 4, 2, 3]
```

Generátorová notácia zoznamu

Generátorová notácia predstavuje jednoduchý a ľahko čitateľný spôsob ako vytvoriť nový zoznam a vložiť doň hodnoty využitím prvkov iných dátových štruktúr.

Najjednoduchší spôsob použitia generátorovej notácie zoznamu môže vyzerať nasledovne:

```
zoznam2 = [cislo for cislo in range(0, 11, 2)]
print(zoznam2)
```

```
[0, 2, 4, 6, 8, 10]
```

V tomto prípade sme využili hodnoty generované funkciou `range()` a vytvorili sme z nich zoznam. Podobne by sme mohli namiesto funkcie `range()` použiť hocijakú inú iterovateľnú dátovú štruktúru, napr. reťazec:

```
pismena = [pismeno for pismeno in 'abeceda']
print(pismena)
```

```
['a', 'b', 'e', 'c', 'e', 'd', 'a']
```

Do zoznamu nemusíme vkladať len hodnoty danej štruktúry, ale aj hodnoty ľubovoľného výrazu:

```
zoznam = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
dvojnasobky = [2 * cislo for cislo in zoznam]
print(dvojnasobky)
```

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Použiť môžeme len niektoré z hodnôt, napr. len nepárne čísla:

```
zoznam = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
dvojnasobky_neparnych = [2 * cislo
```

```
                          for cislo in zoznam
                          if cislo % 2 == 1]
print(dvojnasobky_neparnych)
```

```
[2, 6, 10, 14, 18]
```

Pre vybrané hodnoty môžeme do výsledného zoznamu vkladať hodnoty rôznych výrazov:

```
zoznam = [1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5]
parnost_celych = [f'{cislo}: párne' if cislo % 2 == 0
                  else f'{cislo}: nepárne'
                  for cislo in zoznam
                  if cislo % 1 == 0]
print(parnost_celych)
```

```
['1: nepárne', '2: párne', '3: nepárne', '4: párne']
```

Vo všeobecnosti môže mať teda generátorová notácia zoznamu tvar:

```
[výraz1 if podmienka1 else
 vyraz2 if podmienka2 else
 ...
 else výrazn
 for hodnota in iterovateľná_štruktúra
 if podmienka]
```

Vidíme, že tento zápis môže byť pomerne rozsiahly. To by malo byť aj kritériom použiteľnosti generátorovej notácie. Ak je zápis príliš komplikovaný, rozsiahly alebo ťažko pochopiteľný, použime radšej kombináciu vnorených cyklov a podmienených príkazov. Každú generátorovú notáciu vieme prepísať do cyklov a podmienených výrazov. Naopak to neplatí.

Vnorené dátové štruktúry

Doteraz predstavené dátové štruktúry sme využívali ako lineárne – jednorozmerné. Index prvku bol prirodzené číslo. Zoznamy nemusia obsahovať len jednoduché dátové typy. Ak prvkami zoznamov budú iné dátové štruktúry, môžeme sa na celok pozerať ako na viacrozmernú dátovú štruktúru.

Napr. zoznam:

```
zoznam_2d = [[1, 2, 3], [4, 5], [6]]
```

si môžeme predstaviť aj nasledovne:

```
zoznam_2d = [      0   1   2
             0  [  1 , 2 , 3 ],
             1  [  4 , 5     ],
             2  [  6         ]
            ]
```

K prvkom takejto štruktúry môžeme pristupovať priamo cez dvojicu indexov, napr.:

```
zoznam_2d = [[1, 2, 3], [4, 5], [6]]
print(zoznam_2d[0][1])
zoznam_2d[2][0] = 66
print(zoznam_2d)
```

```
2
[[1, 2, 3], [4, 5], [66]]
```

Rovnaký prístup môžeme použiť, ak prvkami zoznamu sú reťazce :

```
zoznam_retazcov = ['zoznam', 'slov', 'jazyka']
print(zoznam_retazcov[1][3])
```

```
v
```

Pri prístupe k prvkom takejto štruktúry buďme opatrní. Ako vidno z ukážok vyššie, štruktúra nemusí byť pravidelná. Každý „riadok“ má iný počet stĺpcov. Ak nepotrebujeme pracovať s indexom, využívajme radšej iterovanie priamo cez prvky štruktúry. Porovnajme obidva prístupy, iteráciu cez prvky a iteráciu cez indexy:

```
zoznam = [[1, 2, 3],
          [4, 5],
          [6]]
for riadok in zoznam:
    for bunka in riadok:
        print(bunka)
```

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

```
zoznam = [[1, 2, 3],
          [4, 5],
          [6]]
for idx_r in range(len(zoznam)):
    for idx_s in range(len(zoznam[idx_r])):
        print(zoznam[idx_r][idx_s])
```

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

Zbierka úloh

Nasledujúce úlohy sú zamerané na dátovú štruktúru zoznam a algoritmy na zoznamoch.

1. Vytvorte funkciu `generuj_vyhovorku()` ktorá vygeneruje a vráti výhovorku typu:

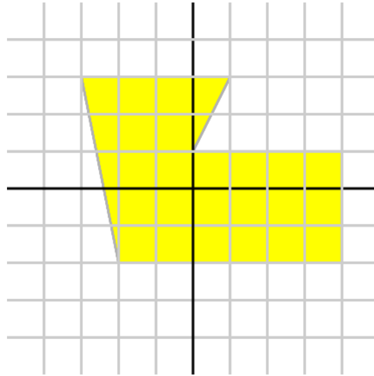
   Pani učiteľka prepáčte, nemám <čo>, pretože <niekto> <urobil><niečo>.

Namiesto slov v zátvorkách vyberte náhodné slová z pripravených zoznamov, napr.: *Pani učiteľka prepáčte, nemám domácu úlohu, pretože krtko uvaril melón.*

Pomôcka: Náhodný prvok zo zoznamu získame takto:

```
import random
zoznam = [1, 2, 3]
nahodny_prvok = random.choice(zoznam)
```

2. V štvorcovej sieti sme ohraničili plochu lomenou čiarou a zaznamenali sme si súradnice bodov lomenej čiary. Napr. plochu:



reprezentuje lomená čiara vyjadrená nasledovne:

[[4, -2], [4, 1], [0, 1], [1, 3], [-3, 3], [-2, -2]]

- Vytvorte funkciu `dlzka_ciary()`, ktorá vráti dĺžku celej lomenej čiary (nezabudnite, že čiara je uzatvorená, t.j. posledný bod je spojený s prvým).
- Vytvorte funkciu `bod_pre_max_skratenie()`, ktorá vráti index bodu lomenej čiary, vynechaním ktorého sa čiara maximálne skráti.
- Vytvorte funkciu `obsah()`, ktorá vráti obsah plochy ohraničenej lomenou čiarou. Pre výpočet obsahu môžete využiť postup na Area of a polygon (https://www.mathopenref.com/coordpolygonarea.html).

Pomôcka: Predpokladajte, že zoznam je korektný, lomená čiara ohraničuje nejakú plochu a spojnice bodov lomenej čiary sa vzájomne nepretínajú.

3. Po kúpe nového auta sme sa rozhodli zaznamenávať jeho spotrebu. Na začiatku sme natankovali plnú nádrž. Pri každom ďalšom tankovaní tankujeme až doplna a značíme si, koľko km sme prešli od posledného tankovania a koľko litrov paliva sme natankovali. Naše záznamy môžu vyzerať nasledovne:

[[300, 25], [540, 35], [380, 30], [400, 20], [350, 20], [200, 13], [400, 35], [520, 40], [430, 30], [500, 28], [500, 32], [480, 30], [300, 20], [500, 25]]

Tieto dáta by sme potrebovali analyzovať a priebežne upravovať. Vytvorte funkcie:

- `celkova_spotreba()` – vypočíta celkové množstvo paliva, ktoré sme použili,
- `prejdena_vzdialenost()` – vypočíta celkovú prejdenú vzdialenosť,
- `priemerna_spotreba()` – vypočíta doterajšiu priemernú spotrebu (l/100 km),
- `najvyššia_spotreba()` – vráti index toho tankovania, v ktorom sme zaznamenali najvyššiu spotrebu,
- `najnižšia_spotreba()` – vráti index toho tankovania, v ktorom sme zaznamenali najnižšiu spotrebu,
- `pridaj_zaznam_na_koniec()` – pridá záznam tankovania na koniec,
- `pridaj_zaznam_na_index()` – pridá záznam tankovania na určený index,
- `zmaz_zaznamy_nad_spotrebou()` – vymaže tie záznamy, v ktorých je spotreba vyššia ako zadaná spotreba (l/100 km).

4. Pascalov trojuholník je usporiadanie kombinačných čísiel v tvare trojuholníka (viď. Pascalov trojuholník). Vytvorte funkciu `pascalov_riadok()` ktorá vráti zoznam čísiel v zadanom riadku Pascalovho trojuholníka.
   - Prístup 1: hodnoty postupne vypočítavame z predchádzajúceho riadka.
   - Prístup 2: hodnoty vypočítame priamo C(k, n) = n! / ((n-k)! * k!).

5. Počas cyklistického tréningu cyklopočítač cyklistu v pravidelných minútových intervaloch meria nadmorskú výšku (v metroch) a vzdialenosť od posledného merania (v metroch). Záznam môže vyzerať nasledovne: [[200, 0], [210, 200], [212, 500], [240, 650], [200, 890], [180, 1000], ...] Po tréningu sa namerané údaje vyhodnotia, aby cyklista vedel, v akej je kondícii. Tréner údaje vyhodnocuje ručne, pomocou kalkulačky. Tento postup je však zdĺhavý a určite by sa dal pomocou počítača urýchliť. Vytvorte funkcie:
   - `vzdialenost()` – vráti vzdialenosť v km, ktorú cyklista prešiel,
   - `cas()` – vráti čas trvania tréningu v minútach,
   - `rychlost()` – vráti priemernú rýchlosť cyklistu v km/h,
   - `najdlhsie_stupanie()` – funkcia vráti dĺžku v km najdlhšieho úseku počas ktorého cyklista stúpal a počet metrov o ktoré vystúpil,

- `najdlhsie_klesanie()` – funkcia vráti dĺžku v km najdlhšieho úseku počas ktorého cyklista klesal a počet metrov o ktoré klesol,
- `stupanie()` – funkcia vráti prevýšenie v m, počas ktorého cyklista stúpal,
- `klesanie()` – funkcia vráti prevýšenie v m, počas ktorého cyklista klesal,
- `prevýšenie()` – funkcia vráti celkové prevýšenie trate v m,
- `najvyssia_rychlost()` – funkcia vráti rýchlosť v km/h, ktorú cyklista dosiahol v najrýchlejšom úseku,
- `najnizsia_rychlost()` – funkcia vráti rýchlosť v km/h, ktorú cyklista dosiahol v najpomalšom úseku,
- `najprudsie_stupanie()` – funkcia vráti úsek, v ktorom bolo najprudšie stúpanie,
- `najprudsie_klesanie()` – funkcia vráti úsek, v ktorom bolo najprudšie klesanie.

6. Počítač v aute zaznamenáva pri zmene rýchlosti aktuálnu rýchlosť auta. Pomocou senzorov počítač zaznamenáva aj dopravné značky (začiatok a koniec obce oz a ok, začiatok a koniec diaľnice dz a dk). Pre maximálne povolené rýchlosti platí:

   - obec – 50 km/h,
   - mimo obce a diaľnica v obci – 90 km/h,
   - diaľnica – 130 km/h.

   Vytvorte funkciu `prekrocena_rychlost()`, ktorá vyhodnotí záznam z počítača auta a vráti odpoveď, či vodič niekde prekročil maximálnu povolenú rýchlosť.

   Príklad: pre záznam počítača:

   ```
   ['zo', 20, 30, 50, 'zd', 80, 90, 'ko', 100, 120, 130, 'kd', 100, 90]
   ```

   je odpoveď `True`, pretože vodič išiel rýchlosťou 100 km/h mimo obce. Môžete predpokladať, že každý záznam začína 'zo'.

   Pomôcka: Spravte si vopred analýzu úlohy, aby ste vedeli dobre implementovať, na akom type cesty sa auto nachádzalo, keď bola meraná rýchlosť.

7. Ľudský mozog dokáže zaujímavé veci. Napr. porozumieť takémuto textu:

   tento txet dštojmruene zaumjvíaú ssnpcohoť ľsuhékdo mgozu

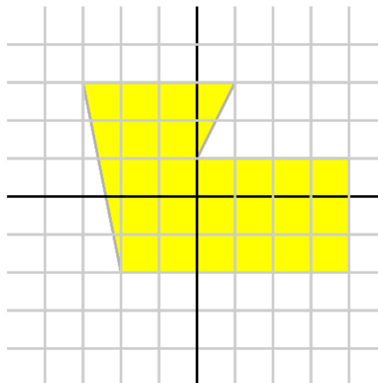
- Definujte funkciu `uprav_slovo()`, ktorá upraví slovo tak, ako v texte vyššie a upravené slovo vráti. Využiť môžete funkciu `zamiesaj_pismena()`.
- Definujte funkciu `uprav_vetu()`, ktorá upraví vetu tak, ako v texte vyššie a upravenú vetu vráti.

```
def zamiesaj_pismena(text):
    ''' Vráti reťazec, v ktorom su znaky
        zadaného reťazca náhodne preusporiadané

    :param text: vstupný reťazec
    :type text: str
    :return: reťazec s náhodne preusporiadanými znakmi
    :rtype: str
    '''
    import random
    pismena = list(text)
    random.shuffle(pismena)
    novy_text = ''.join(pismena)
    return novy_text
```

8. V časti Jemný úvod do problematiky sme vytvorili funkciu, ktorá číselné dáta v zozname vyhladila pomocou kĺzavého priemeru. Vytvorte funkciu `vyhlad_medianom()`, ktorá hodnoty vyhladí použitím kĺzavého mediánu pre zadanú dĺžku intervalu.

Riešené úlohy

2. V štvorcovej sieti sme ohraničili plochu lomenou čiarou a zaznamenali sme si súradnice bodov lomenej čiary. Napr. plochu:



reprezentuje lomená čiara vyjadrená nasledovne:

[[4, -2], [4, 1], [0, 1], [1, 3], [-3, 3], [-2, -2]]

- Vytvorte funkciu `dlzka_ciary()`, ktorá vráti dĺžku celej lomenej čiary (nezabudnite, že čiara je uzatvorená, t.j. posledný bod je spojený s prvým).
- Vytvorte funkciu `bod_pre_max_skratenie()`, ktorá vráti index bodu lomenej čiary, vynechaním ktorého sa čiara maximálne skráti.
- Vytvorte funkciu `obsah()`, ktorá vráti obsah plochy ohraničenej lomenou čiarou. Pre výpočet obsahu môžete využiť postup na Area of a polygon (https://www.mathopenref.com/coordpolygonarea.html).

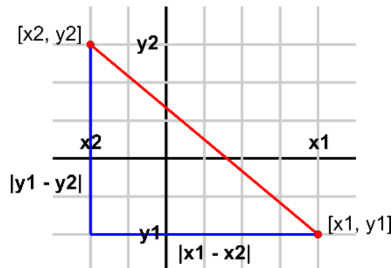
Pomôcka: Predpokladajte, že zoznam je korektný, lomená čiara ohraničuje nejakú plochu a spojnice bodov lomenej čiary sa vzájomne nepretínajú.

Riešenie:

Dĺžka čiary

Dĺžku lomenej čiary budeme počítať postupne. Postupne sčítame vzdialenosti všetkých susedných bodov čiary. Keďže výpočet vzdialenosti bodov budeme realizovať opakovanie, bude výhodné si pre tento účel vytvoriť pomocnú funkciu `vzdialenost()`.

Každý bod reprezentujeme ako zoznam jeho dvoch súradníc. Pre výpočet vzdialenosti využijeme Pytagorovu vetu:




```
def vzdialenost(bod1, bod2):
    dx = bod1[0] - bod2[0]
    dy = bod1[1] - bod2[1]
    vysledok = (dx ** 2 + dy ** 2) ** 0.5
    return vysledok
```

Pre samotný výpočet dĺžky lomenej čiary stačí spočítať vzdialenosti všetkých susedných dvojíc bodov čiary. Keďže budeme pracovať vždy s dvojicou bodov, využijeme prechod cez indexy.

Nezabudnime na spojenie prvého a posledného bodu lomenej čiary. Tento prípad môžeme vyriešiť zvlášť alebo využijeme možnosť záporných indexov (viď. nižšie).

```
def dlzka_ciary(ciara):
    dlzka = 0
    for idx in range(len(ciara)):
        bod1 = ciara[idx - 1]
        bod2 = ciara[idx]
        dlzka += vzdialenost(bod1, bod2)
    return dlzka
```

Bod pre maximálne skrátenie

Riešenie ktoré nás možno napadne ako prvé spočíva v tom, že postupne budeme z pôvodnej čiary vynechávať jednotlivé body. Index bodu, vynechaním ktorého dostaneme najmenšiu dĺžku lomenej čiary, bude výsledkom.

Zamyslime sa, či neexistuje šikovnejší spôsob. Ak vynecháme niektorý z bodov, zmena sa udeje len v jeho okolí. Dve čiary nahradíme jednou. Nemusíme teda počítať dĺžku celej čiary. Stačí spočítať, aký je rozdiel dĺžky pôvodnej „dvojčiary" a novej čiary. Bod, pre ktorý bude tento rozdiel najväčší, je náš hľadaný bod.

```
def bod_pre_max_skratenie(ciara):
    naj_skratenie = 0
    naj_idx = 0
    for idx in range(len(ciara)):
        bod_pred = ciara[idx - 1]
        bod = ciara[idx]
        bod_za = ciara[(idx + 1) % len(ciara)]
        skratenie = vzdialenost(bod_pred, bod) + \
                    vzdialenost(bod, bod_za) - \
                    vzdialenost(bod_pred, bod_za)
        if skratenie > naj_skratenie:
            naj_skratenie = skratenie
            naj_idx = idx
    return naj_idx
```

Všimnime si, aké referenčné hodnoty sme nastavili na začiatku. Najmenšie teoretické skrátenie môže byť 0. Žiadne zo skutočných skrátení teda nebude horšie. Index bodu pre maximálne skrátenie sme určili na 0. Ak nájdeme nejaký lepší bod, použijeme jeho index. Ak nie, výsledkom bude bod na indexe 0.

Pre výpočet úspory, ktorú získame vynechaním bodu, pracujeme aj s jeho susedmi. Zatiaľ čo index ľavého suseda (`idx - 1`) nemusíme kontrolovať (python akceptuje záporné indexy), index pravého suseda (`idx + 1`) môže byť mimo rozsah. Preto sme ho ošetrili (`(idx + 1) % len(ciara)`) tak, aby bol v prípade mimo rozsahu nahradený indexom zo začiatku.

Obsah ohraničenej plochy

Výpočet plochy zrealizujeme podľa postupu na https://www.mathopenref.com/coordpolygonarea.html. Aj keď nie je náročné tento postup zdôvodniť, tu ho berieme ako fakt.

$$S = \left| \frac{(x_0 y_1 - y_0 x_1) + (x_1 y_2 - y_1 x_2) + \cdots + (x_{n-1} y_0 - y_{n-1} x_0)}{2} \right|$$

Samotná implementácia už nie je náročná:

```
def obsah(ciara):
    citatel = 0
    for idx in range(len(ciara)):
        x1 = ciara[idx - 1][0]
        y1 = ciara[idx - 1][1]
        x2 = ciara[idx][0]
        y2 = ciara[idx][1]
        citatel += (x1 * y2 - y1 * x2)
    vysledok = abs(citatel / 2)
    return vysledok
```

Všimnime si, že na extrahovanie súradnice bodu v lomenej čiare sme využili dvojicu indexov. V tomto prípade nám to umožní zápis programu skrátiť a sprehľadniť.

12 Príkaz cyklu while

Jemný úvod do problematiky

Použite notebook **Python 12 - Príkaz cyklu while.ipynb** alebo niektoré z lokálnych vývojových prostredí jazyka Python.

Cieľ: *V tomto notebooku si ukážeme, ako opakovať skupinu príkazov, ak vopred nepoznáme počet opakovaní. Naučíme sa ako podmieniť opakovanie platnosťou logického výrazu.*

Cyklus s podmienkou

Po dekompozícii niektorých problémov na podproblémy zistíme, že rovnaké podproblémy sa opakujú niekoľkokrát za sebou. S takouto situáciou sme sa stretli už nespočetne veľakrát. Problém nastane, ak počet opakovaní nevieme vopred predikovať.

Problém: Vytvorili sme program, ale na jeho spustenie by mal používateľ poznať tajné heslo. Na začiatok programu sme umiestnili programový kód, ktorý si opakovane pýta heslo. Nedokončené riešenie vyzerá nasledovne. Vyskúšajte ho!

```
tajne_heslo = 'tajné'

for i in range(10):
    heslo = input('Zadaj heslo: ')
    if heslo ==  tajne_heslo:
        print('Uhádol si')

print('Nasleduje tajný program')
```

Toto riešenie je problematické z viacerých dôvodov:

- Maximálny počet pokusov je 10. Ak v tomto limite heslo neuhádneme, počítač nás pustí ďalej aj bez znalosti hesla.
- Ak heslo uhádneme, počítač nám to síce oznámi, ale vo výzvach na zadanie hesla pokračuje ďalej.

Uvedené problémy by sme mohli vyriešiť tak, že počet opakovaní nastavíme na nejaké veľmi veľké číslo a ak používateľ heslo uhádne, vykonávanie cyklu prerušíme. Otestujte toto riešenie.

```
tajne_heslo = 'tajné'

for i in range(1000000):
    heslo = input('Zadaj heslo: ')
    if heslo ==  tajne_heslo:
        print('Uhádol si')
        break
```

```
print('Nasleduje tajný program')
```

Toto riešenie problému v podstate nie je riešením.

Po prvé, príkaz `break` znížil prehľadnosť riešenia – cyklus sa neopakuje 1000000-krát, ale niekedy skončí skôr. Riadenie cyklu je ovplyvnené na dvoch miestach programu, čo nám môže skomplikovať prípadné ladenie.

Po druhé, ak by bol používateľ trpezlivý, tak aj bez znalosti hesla sa dostane do tajnej časti programu.

Pozrime sa na iný prístup. Overte nasledovné riešenie.

```
tajne_heslo = 'tajné'

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

print('Nasleduje tajný program')
```

Namiesto príkazu cyklu `for`, ktorý používame v prípade, ak počet opakovaní je vopred známy, sme použili príkaz cyklu `while`.

```
while logicky_vyraz:
    telo cyklu
```

Príkaz cyklu `while` opakuje vykonávanie príkazov v jeho tele na základe hodnoty logického výrazu. Pokiaľ je hodnota logického výrazu `True`, príkazy v jeho tele sa opakované vykonávajú.

Cvičenie 1

Hádaj číslo je jednoduchá hra pre dvoch hráčov. Prvý hráč si vymyslí celé číslo z vopred dohodnutého intervalu a druhý hráč sa pokúša toto číslo uhádnuť. Druhý hráč tipuje konkrétne čísla a prvý odpovedá jednou z nasledujúcich možností: viac, menej, uhádol si. Vytvorte program, ktorý bude simulovať prvého hráča.

Časť programu, vymyslenie čísla, je už vytvorená:

```
import random

min_cislo = 1
max_cislo = 100

vymyslene_cislo = random.randrange(min_cislo, max_cislo)
print(vymyslene_cislo)  # kontrolny vypis
```

Cvičenie 2

Euklidov algoritmus je algoritmus na nájdenie najväčšieho spoločného deliteľa dvoch prirodzených čísiel x a y. Pri výpočte NSD(x, y) postupujeme nasledovne:

NSD(x, y) = NSD(x - y, y); pre x > y,

NSD(x, y) = NSD(x, y - x); pre x < y,

NSD(x, y) = x; pre x = y.

Overte si daný postup na konkrétnych hodnotách x a y.

Definujte funkciu `nsd()`, ktorá pre dve zadané prirodzené čísla vypočíta a vráti ich najväčšieho spoločného deliteľa využitím Euklidovho algoritmu.

Vysvetlenie problematiky

Cyklus s podmienkou

Príkaz cyklu `for` sme používali v situáciách, v ktorých sme vedeli vopred predikovať počet opakovaní. Či už priamo alebo na základe prvkov nejakej štruktúry. V situáciách, kde počet opakovaní vopred nevieme, je použitie príkazu cyklu `for` problematické.

Riadenie cyklu môžeme upraviť tak, že počet opakovaní nastavíme na nejaké extrémne veľké číslo a v prípade potreby vykonávane prerušíme:

```
for i in range(1000000):
    nejaké príkazy 1
    if opakovanie ukončiť:
        break
    nejaké príkazy 2
```

Tento prístup je problematický z dvoch dôvodov:

1. Príkaz `break` znížil prehľadnosť riešenia - cyklus sa neopakuje 1000000x, ale niekedy skončí skôr. Riadenie cyklu je ovplyvnené na dvoch miestach programu, čo nám môže skomplikovať prípadné ladenie. Navyše v cykle `for` takéto predčasné ukončenie neočakávame.
2. Niekedy aj 1000000 opakovaní môže byť málo. Počet opakovaní môžeme samozrejme ešte zväčšiť, ale nikdy nebudeme mať istotu, že to bude stačiť.

V situáciách, keď potrebujeme opakovane vykonávať nejakú skupinu príkazov, ale vopred nevieme predikovať počet opakovaní, môžeme použiť cyklus opakovania `while`:

```
while logický_výraz:
    telo cyklu
```

Opakovanie sa riadi hodnotou logického výrazu za slovom `while`. Logický výraz sa vyhodnocuje pred každým opakovaním. Ak je jeho hodnota `False`, cyklus

skončí. V opačnom prípade sa vykonajú príkazy v tele cyklu. Všimnime si, že ak hodnota logického výrazu je na začiatku `False`, príkazy v tele cyklu sa nevykonajú ani raz.

V niektorých situáciách je žiadúce, aby sa telo cyklu (alebo aspoň nejaká jeho časť) vykonalo aspoň 1-krát. V tomto prípade môžeme časť príkazov vykonať už pred cyklom:

postupnosť príkazov sa nemusí vykonať **ani raz**

```
while logický_výraz:
    telo cyklu
```

postupnosť príkazov sa musí vykonať **aspoň raz**

```
postupnosť_príkazov
while logický_výraz:
    telo cyklu
```

Nekonečný cyklus

Podmienku pre opakovanie cyklu (logický výraz) zostavuje autor programu. Primárne uvažujeme tak, že príkazy v tele cyklu by mali mať vplyv na hodnotu logického výrazu. V opačnom prípade riskujeme vytvorenie nekonečného cyklu.

```
# odpočet
n = 5
while n > 0:
    print(n)
```

```
5
5
5
...
```

Aj keď zabezpečíme, že príkazy v tele cyklu ovplyvňujú platnosť podmienky, môžeme podmienku zostaviť chybne a výsledkom bude opäť nekonečný cyklus:

```
# odpočet
n = 5
while n != 0:
    print(n)
    n = n - 2
```

```
5
3
1
-1
-3
...
```

Riešením ako predchádzať nechcenému vytvoreniu nekonečného cyklu je dôsledná analýza problému a následné testovanie riešenia.

V niektorých prípadoch môžeme nekonečný cyklus vytvoriť zámerne. Uvažujme príkazy v tele cyklu, ktoré by sa mali vykonať aspoň raz. Jedno z riešení sme si uviedli vyššie:

```
príkazy_z_tela_cyklu
while logický_výraz:
    telo cyklu
```

Ak by počet príkazov, ktoré sa majú vykonať pred cyklom a zároveň aj v cykle, bol veľký, program sa zbytočne natiahne. Iné riešenie, v ktorom príkazy nekopírujeme aj pred cyklus je nasledovné:

```
postupnosť_príkazov
while logický_výraz:
    telo cyklu
```

→

```
while True:
    telo cyklu
    if not logický_výraz:
        break
```

Zámerne sme vytvorili nekonečný cyklus (`while True`). Na konci tela cyklu otestujeme hodnotu logického výrazu (`if not logický_výraz`), ktorý sme v pôvodnej verzii testovali na začiatku. Ak je jeho hodnota `False`, cyklus ukončíme (`break`).

Nekonečný cyklus sa oplatí použiť aj v situácii, ak testovaný logický výraz je rozsiahly a v hlavičke by zaberal veľa miesta alebo ak vieme, že podmienka pre opakovanie sa môže zmeniť na rôznych miestach tela cyklu.

Použitie konštrukcie `while True` nie je nič nezvyčajné. Aj tu by sme mali byť opatrní, aby sa nestalo, že výsledný cyklus nikdy neskončí.

Priraďovací výraz

Priraďovací výraz (od verzie Python 3.8), tak ako názov napovedá, v sebe skrýva dve veci:

- priradenie – výsledkom je priradenie hodnoty k premennej,
- výraz – výsledkom je hodnota.

Pozrime si nasledujúci programový kód:

```
1. print(premenna := 10)
2. print(premenna)
```

```
10
10
```

Výsledkom priraďovacieho výrazu je priraďovaná hodnota [1] a zároveň priradenie hodnoty k premennej [2]. Priraďovací výraz obsahuje operátor `:=`. Na ľavej strane je meno premennej a na pravej hodnota alebo výraz, ktorého hodnotu priraďujeme.

Využitím priraďovacieho výrazu môžeme odstrániť duplicitný programový kód, ktorý žiadal zadanie správneho hesla z úvodnej časti tejto kapitoly:

```
tajne_heslo = 'tajné'

heslo = input('Zadaj heslo: ')
while heslo != tajne_heslo:
    print(f'Heslo: "{heslo}" nie je správne!')
    heslo = input('Zadaj heslo: ')

print('Nasleduje tajný program')
```

```
tajne_heslo = 'tajné'

while (heslo := input('Zadaj heslo: ')) != tajne_heslo:
    print(f'Heslo: "{heslo}" nie je správne!')
    heslo = input('Zadaj heslo: ')

print('Nasleduje tajný program')
```

Všimnime si, že samotný priraďovací výraz sme uzavreli do zátvoriek. Ak by sme tak nespravili, do premennej `heslo` by sa priradila hodnota celého výrazu `input('Zadaj heslo: ') != tajne_heslo`.

Priraďovací výraz má pomerne obmedzené použitie. Je určený na vytvorenie čistejšieho programového kódu. Využiť ho môžeme v prípade, ak hodnotu výrazu použijeme v podmienke (`if`, `while`) a zároveň sa vyžaduje, aby sa hodnota čoskoro použila znova. Vždy však existuje spôsob, ako sa priraďovaciemu výrazu „vyhnúť“. Ak vám jeho použitie nezlepší prehľadnosť programového kódu, nepoužite ho.

Zbierka úloh

Nasledujúce úlohy sú zamerané na využitie cyklu s podmienkou.

1. Hra vojna je jednoduchá kartová hra. Hrajú ju dvaja hráči, ktorí si rozdelia hracie karty na polovicu. Každý hráč vyloží kartu, ktorú má na vrchu svojej kopy. Hráč s väčšou hodnotou berie obidve karty a vloží si ich na spodok svojej kopy. Ak by bola hodnota oboch kariet rovnaká, každý z hráčov vyloží tri karty, pričom hodnota poslednej karty rozhoduje. Prehráva ten hráč, ktorí príde o všetky karty alebo už nie je schopný vyložiť požadované množstvo kariet.

   Vytvorte funkciu `vojna()` ktorá, pre dva zadané zoznamy kariet bude simulovať priebeh hry vojna a vráti ktorý hráč vyhral.

   Kvôli jednoduchosti predpokladajte karty s hodnotami 2 až 14 (keďže karty majú rôzne farby, hodnoty kariet sa budú opakovať).

Pomôcka: Zamiešať prvky zoznamu je možné realizovať nasledovne:

```
import random

z = [1, 2, 3]
random.shuffle(z)
```

2. Na riešenie niektorých typov rovníc existujú zaužívané postupy (napr. riešenie kvadratickej rovnice). Ak takýto postup neexistuje, môžeme použiť niektorú z numerických (približných) metód.

   Implementujte riešenie rovnice f(x) = 0 metódou bisekcie – delenie intervalu na polovicu.

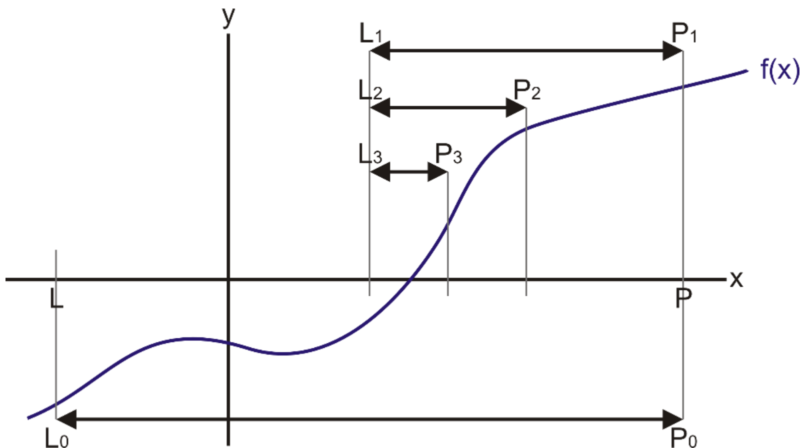
   Vytvorte funkciu `ries_fx_bisekciou()`, ktorá pre zadanú funkciu *f(x)*, ľavý a pravý okraj intervalu a požadovanú presnosť nájde a vráti koreň zadanej rovnice. Napr.:

```
print(ries_fx_bisekciou('(x - 5) * (x + 10)', -2, 8,
0.00001))
```

```
4.999998092651367
```

   Pomôcka:

   - *Na vyhodnotenie hodnoty zadanej funkcie v bode x použite funkciu `eval()`,*
   - *pri porovnanie znamienok využitie funkciu `math.copysign()`,*
   - *predpokladajte, že zadaná funkcia f(x) je na danom intervale spojitá,*
   - *ak majú funkčné hodnoty zadanej funkcie v krajných bodoch intervalu rovnaké znamienko, je to chyba.*



Metóda bisekcie

3. Euklidov algoritmus ktorý bol uvedený v prípravnom notebooku je pri niektorých hodnotách x, y príliš pomalý. Identifikujte pre aké hodnoty x, y

je výpočet pomalý a navrhnite lepšiu verziu algoritmu. Vylepšenú verziu implementujte.

4. Jožko sa na matematike učil sčitovať čísla do 20, ale veľmi mu to nejde. Potreboval by pravidelne trénovať. Vytvorte program, ktorý bude Jožkovi generovať príklady na sčítanie a načítavať jeho odpovede. Ak počet správnych odpovedí bude aspoň o 5 viac ako počet nesprávnych, program skončí.

   Pomôcka: Náhodne celé číslo z intervalu <a, b) vygenerujete nasledovne:

```
import random
cislo = random.randrange(a, b)
```

5. Definujte funkciu `zoznam_prvocisiel()`, ktorá pre zadané n vráti zoznam prvých n prvočísiel. Na test, či číslo je prvočíslo môžete využiť funkciu `je_prvocislo()` zo zbierky úloh v kapitole 5.

6. Jedna z úloh v prípravnom notebooku bola simulovať prvého hráča (hráč, ktorý si číslo na začiatku vymyslí) v hre Hádaj číslo. Implementujte simuláciu druhého hráča (hráč, ktorý háda číslo).

7. Automatický žeriav je schopný prekladať tovary z nakladacej rampy do prepravného vozíka, ktorý má určenú nosnosť. Obslužný systém žeriava má k dispozícii informácie o hmotnostiach tovaroch na rampe. Inžinieri navrhli niekoľko stratégií, ako vozíky plniť:

   a) Tovary sa budú nakladať v takom poradí v akom sú umiestnené na rampe. Ak by sa naložením tovaru, ktorý je v poradí, prekročila nosnosť vozíka, nakládka vozíka končí.
   b) Tovary sa budú nakladať v takom poradí v akom sú umiestnené na rampe. Ak by sa naložením tovaru, ktorý je v poradí prekročila nosnosť vozíka, žeriav tento tovar vynechá a pokračuje nasledujúcim tovarom na rampe.
   c) Tovary sa budú nakladať na striedačku. Najskôr sa naloží najťažší tovar, potom najľahší a toto sa opakuje až dovtedy, keď už nie je žiaden tovar, ktorý by sa dal naložiť.

   Implementujte jednotlivé stratégie. Funkcia, ktorá implementuje niektorú zo stratégií, dostane zoznam hmotností tovarov (v poradí v akom sú tovary uložené na rampe) a nosnosť vozíka a vráti zoznam zoznamov hmotností tovarov, ktoré žeriav naloží na jednotlivé vozíky.

Riešené úlohy

1. Hra vojna je jednoduchá kartová hra. Hrajú ju dvaja hráči, ktorí si rozdelia hracie karty na polovicu. Každý hráč vyloží kartu, ktorú má na vrchu svojej kopy. Hráč s väčšou hodnotou berie obidve karty a vloží si ich na spodok

svojej kopy. Ak by bola hodnota oboch kariet rovnaká, každý z hráčov vyloží tri karty, pričom hodnota poslednej karty rozhoduje. Prehráva ten hráč, ktorí príde o všetky karty alebo už nie je schopný vyložiť požadované množstvo kariet.

Vytvorte funkciu `vojna()` ktorá, pre dva zadané zoznamy kariet bude simulovať priebeh hry vojna a vráti ktorý hráč vyhral.

Kvôli jednoduchosti predpokladajte karty s hodnotami od 2 do 14 (keďže karty majú rôzne farby, hodnoty kariet sa budú opakovať).

Pomôcka: Zamiešať prvky zoznamu je možné realizovať nasledovne:

```
import random

z = [1, 2, 3]
random.shuffle(z)
```

Riešenie:

Rozdeľme si riešenie na časti:

1. Implementujeme hru Vojna,
2. Náhodne vygenerujeme karty pre hráčov a spustíme simuláciu

Hra vojna

Samotná fyzická hra Vojna prebieha tak, že hráči berú karty zo svojej kopy a vyhadzujú ich na spoločné hracie pole, aby sa dali hodnoty kariet vzájomne porovnať. Pri simulácii nemusíme karty vyhadzovať. Stačí, ak si budeme pamätať indexy kariet, ktorých hodnoty porovnávame. Ak jeden z hráčov vyloží vyššiu kartu, stačí presunúť postupnosti kariet z jednotlivých zoznamov na koniec zoznamu toho hráča, ktorú mal vyššiu kartu. Toto budeme opakovať dovtedy, kým hráči majú dostatok kariet na to, aby mohli svoje karty vykladať.

```
def vojna(hrac1:list, hrac2:list):
    idx = 0
    while idx < len(hrac1) and idx < len(hrac2):
        karta1 = hrac1[idx]
        karta2 = hrac2[idx]
        if karta1 > karta2:
            hrac1.extend(hrac1[:idx + 1])
            hrac1.extend(hrac2[:idx + 1])
            del(hrac1[:idx + 1])
            del(hrac2[:idx + 1])
            idx = 0
        elif karta1 < karta2:
            hrac2.extend(hrac2[:idx + 1])
            hrac2.extend(hrac1[:idx + 1])
            del(hrac1[:idx + 1])
            del(hrac2[:idx + 1])
            idx = 0
```

```
        else:
            idx = idx + 3

    if idx < len(hrac1):
        return 'hrac1'
    elif idx < len(hrac2):
        return 'hrac2'
    else:
        return 'remiza'
```

V premennej `idx` si priebežne pamätáme index karty, ktorú hráči porovnávajú. Na začiatku je to karta na vrchu – index 0. Keďže hráči vyhadzujú vždy rovnaké počty kariet, tento index je rovnaký pre oboch hráčov.

Pokiaľ obaja hráči majú kartu na danom indexe (to znamená, že ju môžu vyložiť a vzájomne porovnať), budeme zodpovedajúce karty hráčov porovnávať.

Ak niektorý z hráčov mal väčšiu kartu, presunieme karty zo začiatku až po porovnávanú kartu na koniec jeho kopy a zároveň pridáme aj zodpovedajúce karty súpera. Index porovnávanej karty opäť nastavíme na 0.

Ak majú vyložené karty rovnakú hodnotu, posunieme index porovnávaných kariet o 3.

Cyklus skončí, ak niektorý z hráčov už nemá dostatok kariet na to, aby mohol kartu vyložiť. Tu by sme si mali uvedomiť, že môže nastať situácia, keď žiaden z hráčov už nemôže kartu vyložiť. Tento stav vyhlásime za remízu.

Výslednú funkciu by sme mali otestovať. Testovanie využitím všetkých kariet náročné, lebo by sme museli vopred predikovať, ako hra skončí. Môžeme ju preto testovať na menších kôpkach, napr.:

```
print(vojna([1, 2, 3, 4], [1, 2, 3, 4]))   # remíza
print(vojna([2, 2], [1, 1]))   # hráč1
print(vojna([2, 2], [3, 3]))   # hráč2
print(vojna([1, 2, 3, 4, 5], [2, 2, 3]))   # hráč1
print(vojna([2, 2, 3], [1, 2, 3, 4, 5]))   # hráč2
```

Generujeme karty v kôpkach

Podľa zadania máme k dispozícii karty s hodnotami 2 až 14 a každú hodnotu v štyroch farbách. Keďže farba v hre vojna nerozhoduje, môžeme ju ignorovať a každú hodnotu budeme mať 4-krát. Samotné vygenerovanie, zamiešanie a rozdanie kariet môžeme simulovať nasledovne:

```
import random

karty = list(range(2, 15)) * 4
random.shuffle(karty)
```

```
hrac1 = karty[:len(karty)//2]
hrac2 = karty[len(karty)//2:]
```

3. Euklidov algoritmus ktorý bol uvedený v prípravnom notebooku je pri niektorých hodnotách x, y príliš pomalý. Identifikujte pre aké hodnoty x, y je výpočet pomalý a navrhnite lepšiu verziu algoritmu. Vylepšenú verziu implementujte.

   Riešenie:

   Všimnime si, že algoritmus sa „dopracuje“ k výsledku postupným odpočítavaním menšieho čísla od väčšieho.

   NSD(x, y) = NSD(x - y, y); pre x > y,

   NSD(x, y) = NSD(x, y - x); pre x < y,

   NSD(x, y) = x; pre x = y.

   Ak bude rozdiel čísiel príliš veľký, odpočítavanie bude zdĺhavé. Tento postup vieme skrátiť. Ak budeme od väčšieho čísla odpočítavať menšie, dopracujeme sa k zvyšku po delení. Napr.

   `15 - 4 - 4 - 4 = 3 = 15 % 4`

   Nahraďme teda postupné odpočítavanie zvyškom po delení.

   Pôvodná verzia algoritmu končí, ak sa hodnoty x a y rovnajú. Pozrime sa, čo sa stane, ak väčšie číslo je násobkom menšieho: 16 % 4 = 0. Vylepšená verzia algoritmu teda skončí, ak jedno z čísiel bude 0 a výsledkom bude druhé, nenulové číslo. Napr.:

   `NSD(12, 18) = NSD(12, 6) = NSD(0, 6) = 6`

   Výsledná funkcia môže vyzerať nasledovne:

```
def nsd1(x, y):
    while x != 0 and y != 0:
        if x > y:
            x = x % y
        else:
            y = y % x
    return x + y
```

   Vylepšenú verziu Euklidovho algoritmu sme vytvárali z dôvodu urýchlenia výpočtu. Na záver môžeme otestovať, aké skrátenie času výpočtu nová verzia algoritmu prinesie. Jednoduchý test času výpočtu môžeme realizovať nasledovne:

```
cas = time.monotonic()
print(nsd1(100000000, 1))
print(time.monotonic() - cas)  # rozdiel v sekundách
```

Informačné zdroje

- BEECHER, Karl. *Computational thinking: A beginner's guide to problem-solving and programming*. UK: BCS Learning & Development, 2017. ISBN 978-1-78017-36-58.
- GRALLA, Linn, Thora TENBRINK, Michael SIEBERS a Ute SCHMID. Analogical Problem Solving: Insights from Verbal Reports. In: *Proceedings of the Annual Meeting of the Cognitive Science Society*. 2012, s. 396-401. ISSN 1069-7977. Dostupné tiež z: https://escholarship.org/uc/item/55j4q6k3
- LANE, Matt. Problem Solving Strategies. *Rithm School* [online]. San Francisco: Rithm, 2016 [cit. 2021-6-1]. Dostupné z: https://www.rithmschool.com/blog/problem-solving-strategies-01
- *PALMA junior* [online]. Košice: Univerzita Pavla Jozefa Šafárika v Košiciach, Slovensko, Prírodovedecká fakulta, Ústav informatiky, 2021 [cit. 2021-5-31]. Dostupné z: https://di.ics.upjs.sk/palmaj/
- POLYA, G. *How to Solve It: A New Aspect of Mathematical Method*. Princeton; Oxford: Princeton University Press, 1945. ISBN 978-1-4008-2867-8.
- *PyCharm Edu: A Professional Tool to Learn and Teach Programming with Python* [online]. JetBrains, 2021 [cit. 2021-5-31]. Dostupné z: https://www.jetbrains.com/pycharm-edu/
- *Python 3.9.5 documentation* [online]. Python Software Foundation, 2021 [cit. 2021-5-31]. Dostupné z: https://docs.python.org/3/
- *Python Tutorial* [online]. Refsnes Data, 2021 [cit. 2021-5-31]. Dostupné z: https://www.w3schools.com/python/default.asp
- QUANTIFIEDCODE. *Python Anti-Patterns: The Little Book of Python Anti-Patterns and Worst Practice*. Berlin: QuantifiedCode, 2018. Dostupné tiež z: https://docs.quantifiedcode.com/python-anti-patterns/#
- SPRAUL, V. Anton. *Think like a programmer: An Introduction to Creative Problem Solving*. San Francisco: William Pollock, 2012. ISBN 978-1-59327-424-5.
- The do's and dont's of teaching problem solving in math. *Https://www.homeschoolmath.net/* [online]. HomeschoolMath.net [cit. 2021-5-31]. Dostupné z: https://www.homeschoolmath.net/teaching/problem_solving.php

Príloha

Interaktívne Jupyter notebooky:

- Python 1.1 - Jupyter Notebook.ipynb
- Python 1.2 - Úvod do jazyka Python.ipynb
- Python 2.1 - Korytnačia grafika.ipynb
- Python 2.2 - Vlastné funkcie bez parametrov a návratovej hodnoty.ipynb
- Python 3 - Príkaz cyklu for a funkcia range().ipynb
- Python 4 - Funkcie s parametrami, funkcie s návratovou hodnotou.ipynb
- Python 5 - Podmienky a podmienený príkaz.ipynb
- Python 6 - Reťazce a reťazcové metódy.ipynb
- Python 7 - Algoritmy na reťazcoch.ipynb
- Python 8 - Chyby a spracovanie výnimiek.ipynb
- Python 9 - Generovanie výnimiek.ipynb
- Python 10 - Zoznamy a metódy zoznamov.ipynb
- Python 11 - Algoritmy na zoznamoch.ipynb
- Python 12 - Príkaz cyklu while.ipynb

Programovanie v Pythone 1
Vysokoškolský učebný text

Autori:
PaedDr. Ján Guniš, PhD.,
doc. RNDr. Ľubomír Šnajder, PhD.

Vydavateľ: Univerzita Pavla Jozefa Šafárika v Košiciach
Vydavateľstvo ŠafárikPress

Rok vydania: 2021
Náklad: 70
Rozsah strán: 170
Rozsah: 7,23 AH
Umiestnenie publikácie: https://unibook.upjs.sk/img/cms/2021/pf/programovanie-v-pythone-1.pdf
Umiestnenie príloh: https://unibook.upjs.sk/dokumenty/prilohy-programovanie-v-pythone-1.zip
Vydanie: prvé
Tlač: EQUILIBRIA, s. r. o.

Dostupné od: 24.3.2021
ISBN 978-80-8152-969-6 (tlačená publikácia)
ISBN 978-80-574-0009-7 (e-publikácia)