Základy programování v Pythonu

Jan Laštovička

21. prosince 2020

1 První seminář

1.1 Výrazy

Spusťte příkaz `python3` v terminálu. Pokud Python verze 3 nemáte, můžete jej stáhnout ze stránky `https://www.python.org/downloads/` a nainstalovat. Přivítá vás interpret Pythonu, který za znaky >>> očekává vstup. Zadejte znaky `1` a `2` a stiskněte Return.

```
>>> 12
12
>>>
```

Na první pohled se zdá, že interpret pouze vytiskl zadané znaky `12` a čeká na další vstup. Ve skutečnosti proběhlo několik fází, které si nyní projdeme. Nejdříve interpret přečetl zadané znaky a vytvořil z nich hodnotu, která reprezentuje číslo 12. Pro jednoduchost budeme říkat číslo jak cifrám, které jsme zadali, tak hodnotě, která vznikla jejich přečtením. Čísla jsou speciálním případem výrazů. Dále interpret výraz vyhodnotil. Výsledkem vyhodnocení výrazu je hodnota. Vyhodnocení čísla probíhá triviálně. Výsledkem je to samé číslo, které jsme vyhodnocovali. V našem případě číslo 12. Nakonec je výsledná hodnota vytištěna. Pod námi zadaným vstupem se objevily znaky `12`. Tento proces se opakuje dokud jej nepřerušíme zadáním `exit()` a stiskem Return. Tím interpret ukončíme a vrátíme se do terminálu. Pokud interpret neukončíme, můžeme zadat celé nezáporné číslo výpisem jeho cifer v desítkové soustavě. Pro teď nic jiného neumíme.

Shrňme si práci interpretu. Nejprve je načten vstup (fáze read), poté je vstup vyhodnocen (fáze eval) a nakonec výsledek vytištěn (fáze print). Tento proces se opakuje (fáze loop). Spojíme-li počáteční písmena anglických jmen fází dostaneme zkratku REPL, která tento proces označuje.

Pokusme se zadat před číslo cifru nula.

```
>>> 01
  File "<stdin>", line 1
    01
     ^
```

```
SyntaxError: leading zeros in decimal integer literals are not permitted;
 use an 0o prefix for octal integers
>>>
```

Obdrželi jsme syntaktickou chybu (`SyntaxError`). Syntaktické chyby vznikají ve fázi čtení vstupu a upozorňují nás na to, že porušujeme gramatiku jazyka. V tomto případě gramatika Pythonu zakazuje začít číslo cifrou nula.

Představíme si další typ výrazu. Máme-li dva výrazy v_1 a v_2 můžeme vytvořit výraz

v_1+v_2

nazývaný součet.

Protože čísla `1` a `2` jsou výrazy, můžeme vytvořit výraz `1+2`. Co se stane, když necháme tento výraz vyhodnotit?

```
>>> 1+2
3
```

Jak jste asi předpokládali, obdrželi jsme součet čísel 1 a 2. Pojďme se ale podrobněji podívat, jak k tomu došlo. Ve fázi čtení vstupu se vytvořil výraz součtu v_1+v_2, kde podvýrazy v_1 a v_2 jsou postupně čísla 1 a 2. Vyhodnocení výrazu součtu probíhá tak, že se nejprve vyhodnotí podvýrazy v_1 a v_2. Tím obdržíme dvě hodnoty, které následně sečteme. Protože v_1 a v_2 jsou v našem případě čísla 1 a 2, jejich vyhodnocením, jak již víme, obdržíme opět čísla 1 a 2. Součet je roven číslu 3. Tím končí fáze vyhodnocení. Zbývá výsledek vytisknout.

Pokud u součtu vynecháme výraz v_2 vznikne syntaktická chyba:

```
>>> 1+
  File "<stdin>", line 1
    1+
     ^
SyntaxError: invalid syntax
```

Poznamenejme, že vynechání podvýrazu v_1 k chybně nevede. Tedy vstup `+1` je v jazyce platný, ale nejedná se o součet.

Uvědomme si, že součet je také výraz a může tedy vystupovat u dalšího součtu v roli podvýrazu v_1 nebo v_2. Gramatika tedy umožňuje vytvořit vstup `1+2+3`. Ten dokonce mohl vzniknout dvojím způsobem. Zaprvé jsme nejprve mohli vytvořili součet `1+2` a poté jej v roli v_1 použili v součtu `1+2+3`, kde jako v_2 vystupuje číslo `3`. Zadruhé jsme mohli začít součtem `2+3` a poté vytvořit součet `1+2+3`, kde v_1 by bylo číslo `1` a v_2 součet `2+3`.

Ať už vstup `1+2+3` vznikl prvním nebo druhým způsobem, interpret po zadání vytiskne `6`.

```
>>> 1+2+3
6
```

Platí, že interpret u součtu upřednostnil první způsob vytvoření vstupu. Nejprve tedy sečetl čísla 1 a 2, tím obdržel číslo 3, a poté k výsledku přičetl číslo 3.

Libovolný výraz můžeme uzavřít do kulatých závorek. Přesněji, pokud je v výraz, pak

`(`v`)`

je také výraz. Výraz `(`v`)` se vyhodnotí prostě tak, že se vyhodnotí jediný podvýraz v a jeho výsledek je i výsledkem vyhodnocení výrazu `(`v`)`. Význam závorek je, že umožňují měnit pořadí vyhodnocení výrazů.

Výraz `1+2+3` interpret vyhodnotí stejně jako výraz `(1+2)+3`.

```
>>> (1+2)+3
6
```

Pomocí závorek můžeme změnit pořadí vyhodnocení součtů.

```
>>> 1+(2+3)
6
```

Výsledek je sice stejný jako v prvním případě, ale interpret nejdříve sečetl čísla 2 a 3 a až poté sečetl 1 a 5.

Při zapomenutí otevírací závorky čtení vstupu skončí chybou.

```
>>> 1+2+3)
  File "<stdin>", line 1
    1+2+3)
         ^
SyntaxError: unmatched ')'
```

Při zapomenutí uzavírací závorky interpret očekává její doplnění na dalším řádku.

```
>>> 1+(2+3
... )
6
```

Uzavřít do závorek lze libovolný výraz. Závorky můžeme přidat i tam, kde nepřinesou žádný nový význam.

```
>>> (1)
1
>>> ((1))
1
```

Součet je příklad operátoru. Jedná se o operátor binární, protože má dva podvýrazy. K operátoru součtu je přiřazena operace sčítající čísla. Operace je také binární: vyžaduje dva argumenty nazývané operandy. Rozdíl mezi operátorem a operací spočívá v tom, že operátor určuje jak z existujících výrazů

složit nový výraz. Oproti tomu operace definuje jakým způsobem se při vyhodnocení operátoru spočítá ze vstupních hodnot výsledná hodnota. Například pomocí operátoru + můžeme ze dvou výrazů `1` a `(2+3)` složit výraz `1+(2+3)`. Při jeho vyhodnocení se vykoná operace sčítání, kde v roli operandů vystupují dvě hodnoty reprezentující čísla 1 a 5.

Další binární operátory, jejichž operace pracují s celými čísly, jsou - (rozdíl),* (násobek), `//` (celočíselné dělení) a `%` (zbytek po celočíselném dělení).

Vyzkoušejme si nové operátory.

```
>>> 5-2
3
>>> 5*2
10
>>> 5//2
2
>>> 5%2
1
```

Všimněme si, že pomocí operátoru - můžeme vytvořit záporné číslo.

```
>>> 0-1
-1
```

U operátorů `//` a `%` dochází k chybě dělení nulou v případě, že se druhý podvýraz vyhodnotí na nulu.

```
>>> 4%0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> 4//0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Tato chyba je jiného druhu, než dříve uvedené syntaktické chyby. Nenastává při čtení vstupu, ale až ve fázi vyhodnocení výrazu.

Nově představené operátory se stejně jako operátor sčítání vyhodnocují zleva doprava. Proto se například výraz `2-2-2` vyhodnotí jako `(2-2)-2`.

Operátory + a - mají menší prioritu než operátory *, `//` a `%`. To znamená, že se při čtení vstupu snaží interpret považovat operátory z druhé skupiny jako podvýrazy operátorů z první skupiny. Například výraz `1+2*3` je chápán jako `1+(2*3)`. Uvnitř skupin platí aplikování operátorů zleva doprava. Tedy `10//2%3` je to samé jako `(10//2)%3`

Představíme si jeden operátor, který má pouze jeden podvýraz a proto jej nazýváme unární. Pokud v je výraz, pak

```
-v
```

je také výraz. Aby nedocházelo k záměně s binárním operátorem `-`, označujeme jej `-x`. Operátor při vyhodnocení počítá operaci opačného čísla.

```
>>> -5
-5
```

Zde vstup `-5` je unárním operátorem `-x` s podvýrazem `5`, který se vyhodnotí tak, že se spočítá opačné číslo k číslu 5. Výsledné záporné číslo -5 je následně vytištěno znaky `-5`. Tato skutečnost je zřejměji vyjádřena následovně.

```
>>> -(5)
-5
```

Operátor `-x` má větší prioritu než všechny dosud představené operátory. Proto například výraz `-1-1` je chápán jako `(-1)-1`.

Kolem znaků uvnitř binárních operátorů někdy píšeme mezery, aby jsme výraz zpřehlednili. Tyto mezery nemají na význam výrazu žádný vliv. Pro přehlednost můžeme tedy výraz `-1-1` zapsat jako `-1 - 1`.

1.2 Proměnné

Výrazy slouží k vyjádření výpočtu hodnoty. Připomeňme si například, že záporné číslo neumíme zadat přímo, ale musíme jej vypočítat jako opačné číslo ke kladnému číslu. Kromě výrazu můžeme interpretu zadat příkaz přiřazení hodnoty do proměnné. Pokud i je jméno proměnné a v výraz, pak

i=v

je příkaz přiřazení. Příkaz se vyhodnotí tak, že se nejprve vyhodnotí výraz v a poté vznikne vazba proměnné i na výsledek vyhodnocení.

```
>>> x=1
```

Interpret nic nevytiskl. Pouze vytvořil vazbu proměnné `x` na hodnotu 1. Mluvíme také o tom, že proměnná `x` má hodnotu 1. Seznam vazeb si můžeme představit jako tabulku, kde v prvním sloupci jsou jména proměnných a v druhém k nim navázané hodnoty. Tabulka vazeb nyní vypadá následovně.

x	1

Každé jméno proměnné je výrazem. V případě, že proměnná má vazbu na hodnotu, je výsledkem vyhodnocení navázaná hodnota.

```
>>> x
1
```

Pokud proměnná nemá vazbu, skončí vyhodnocení chybou.

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

Protože proměnná je výrazem, můžeme ji použít jako podvýraz jiného výrazu.

```
>>> x+x
2
```

Oproti tomu přiřazení není výrazem a není tedy možné jej použít jako podvýraz.

```
>>> 1+(a=2)
  File "<stdin>", line 1
    1+(a=2)
       ^
SyntaxError: invalid syntax
```

Přiřazení hodnoty k proměnné, která již má vazbu, způsobí, že vazba se změní na zadanou hodnotu.

```
>>> x=2
```

Tabulka vazeb bude po provedení předchozího příkazu vypadat následovně.

x	2

Chcete-li zrušit všechny vazby proměnných, ukončete interpreter vstupem `exit()` a znovu jej spusťte.

Z důvodu lepší čitelnosti budeme vždy psát kolem znaku přiřazení mezery.

```
>>> x = 2
```

Na pravé straně od znaku přiřazení může být libovolný výraz. Dokážete říci, co způsobí následující příkaz? (Předpokládáme, že proměnná `x` má vazbu na číslo 2.)

```
>>> x = x
```

Nejprve se vyhodnotí výraz na pravé straně od rovnítka. Protože se jedná o proměnnou, která má vazbu, bude výsledkem navázané číslo 2. Poté se změní vazba proměnné `x` na číslo 2. Výsledek nebude tedy mít žádný efekt.

A co tento příkaz?

```
>>> x = x + 1
```

Podobně jako v předchozím případě se nejprve vyhodnotí výraz na pravé straně. Zde ale je výraz `x + 1` jehož hodnota je 3. Poté dojde ke změně vazby proměnné `x` na hodnotu 3. Efekt bude takový, že hodnota proměnné se zvýší o jedna.

Jméno proměnné může obsahovat písmena, číslice a podtržítka. Jména zpravidla píšeme malými písmeny v anglickém jazyce a jednotlivá slova oddělujeme podtržítkem.

```
>>> age = 15
>>> person1_height = 179
```

Proměnná nesmí začínat číslicí. To by vedlo k chybě v syntaxi.

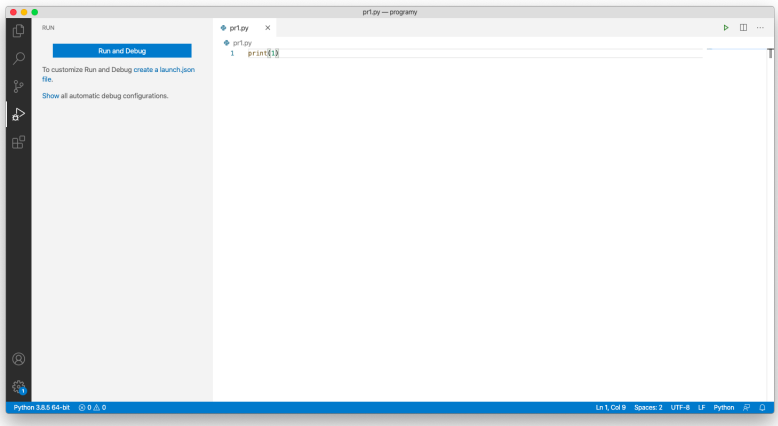
```
>>> 1age
  File "<stdin>", line 1
    1age
     ^
SyntaxError: invalid syntax
```

1.3 Program

V následujících částech budete potřebovat program Visual Studio Code s rozšířením pro Python (`https://code.visualstudio.com`). Vytvořte si nový adresář kam budete ukládat programy. Spusťte Visual Studio Code a dejte otevřít nově vytvořený adresář volbou `Open folder...` Vytvořte nový soubor s obsahem

```
print(1)
```

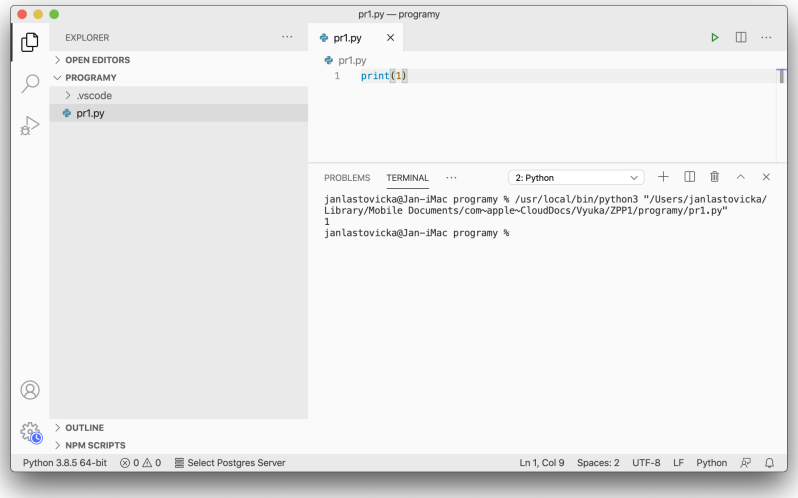
a uložte jej pod názvem `pr1.py`. V položce `Run` lišty aktivit (sloupec vlevo) si nechte vytvořit `launch.json` soubor a vyberte si první možnost (Python File).



Soubor `launch.json` můžete zavřít. Spusťte klikem na zelený trojúhelník program (▷). V otevřeném terminálu je vidět výstup programu:

```
1
```

Okno studia by mělo nyní vypadat následovně.



Obsahem souboru je malý program sestávající z jediného řádku: `print(1)` Jedná se o příkaz tisku. Přesněji pokud v je výraz, pak

`print(`v`)`

je příkaz tisku. Příkaz se vykoná (vyhodnotí) tak, že se nejdříve vyhodnotí výraz v a poté se výsledek vytiskne.

Spuštění programu vykoná příkaz tisku a tím vytiskne číslo 1. Možná vás napadlo, že stejný význam by mělo do programu napsat jen výraz `1`. Spuštění takového programu by nemělo žádný efekt. Nedojde k tisku žádného výstupu a program by měl stejný význam, jako by byl prázdný. Nejprve uveďme, že každý výraz je i příkazem. Tedy náš program obsahující pouze `1` je z pohledu zápisu správný. Tisk výsledku vyhodnocení výrazu se ale provádí pouze v interaktivním (REPL) režimu interpretu. Proto v spuštěných programech potřebujeme k tisku hodnoty použít příkaz `print`.

Podívejme se na mírně složitější program.

```
a = 1
print(a)
```

Program zapsaný v souboru se skládá ze dvou příkazů. Spuštění programu postupně vykoná oba příkazy. Nejprve se vykoná příkaz přiřazení `a = 1`. Tím vznikne vazba proměnné `a` na číslo 0. Poté se vykoná příkaz tisku `print(a)`. Výraz `a` se vyhodnotí na číslo 1, které se příkazem vytiskne. Efekt bude tedy opět tisk čísla 1.

Program se obecně skládá z příkazů a běh každého programu probíhá tak, že se postupně vykonávají jeho příkazy.

Další rozdíl mezi interaktivním módem a spuštěním programu zapsaného v souboru spočívá v tom, že v druhém případě se provede kontrola gramatiky

celého programu před jeho spuštěním. V následujícím příkladu se tedy první příkaz vůbec neprovede, protože druhý řádek obsahuje syntaktickou chybu.

```
print(1)
1+
```

Pokus o spuštění skončí chybou

```
File ".../pr1.py", line 2
    1+
     ^
SyntaxError: invalid syntax
```

Porovnejme výpis po spuštění následujícího programu.

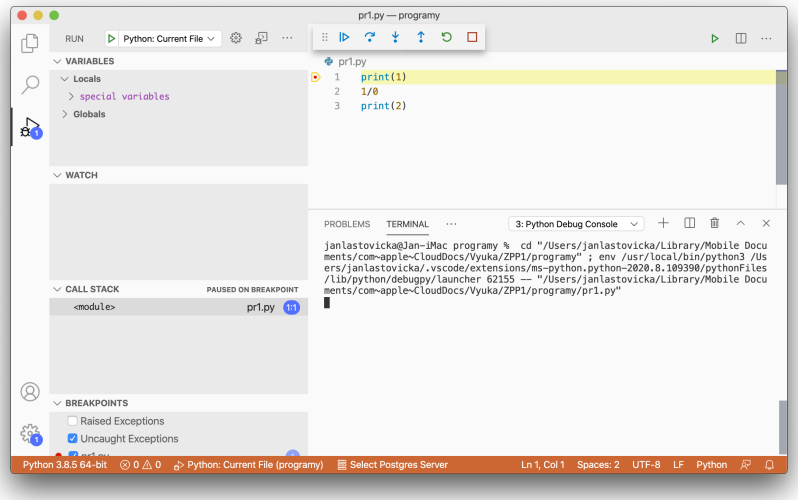
```
print(1)
1/0
print(2)
```

V terminálu se objeví:

```
1
Traceback (most recent call last):
  File ".../pr1.py", line 2, in <module>
    1/0
ZeroDivisionError: division by zero
```

Vidíme, že došlo k vykonání prvního příkazu a vytištění číslice 1. Teprve až vykonání druhého příkazu skončilo chybou. Běh programu se tím zastavil a k vykonání třetího příkazu vůbec nedošlo.

Klikneme-li nalevo od čísla řádku vložíme na řádek zarážku anglicky breakpoint (●). Vložme zarážku na první řádek (● 1 print(1)) a spusťme program pro ladění volbou `Start Debugging` (▷ Python: Current File) v položce `Run` (▶) nabídky aktivit. Studio by nyní mělo vypadat jako na níže uvedeném obrázku.



Vidíme, že vykonávání programu se zastavilo na zarážce ještě před vykonáním prvního příkazu. Nyní máme dvě možnosti. Můžeme nechat program vykonávat další příkazy od zarážky volbou **Continue** ( ), nebo vykonat následující příkaz a znovu se zastavit volbou **Step Over** ( ). Využijeme druhé možnosti, které budeme říkat krok, a vykonáme příkaz `print(1)`. Na výstup se vytisklo číslo 1. Vykonávání programu je nyní zastavené na příkazu `1/0`. Další krok v programu způsobí chybu dělení nulou a studio v programu označilo místo, kde k chybě došlo. Pokus o další krok chybu vytiskne a program ukončí. Proces, kterým jsme nyní prošli, nazýváme krokováním programu. Klikem zarážku odstraníme.

Při zastavení programu studio v sekci **Variables** (vlevo nahoře) zobrazuje aktuální vazby proměnných. Vezměme si program

```
x = 1
y = x + 1
x = y * 3
print(x)
```

a přidejme zarážku na poslední příkaz a spusťme program pro ladění. Program se před tiskem zastavil a ve studiu vidíme, že proměnná `x` má vazbu na číslo 6 a proměnná `y` na 2. Necháme program doběhnout a přesuneme zarážku na první řádek. Krokováním programu můžeme sledovat jak se vazby proměnných vytvářejí a mění.

V kostře programu

```
x = 1
y = 2
?
print(x)
```

```
print(y)
```

chceme na místo `?` doplnit kód tak, aby se prohodily vazby proměnných `x` a `y`. Program by tedy měl vytisknout nejdříve `2` a poté `1`. Myslíte si, že bude následující řešení fungovat?

```
x = 1
y = 2
x = y
y = x
print(x)
print(y)
```

Projdeme si běh programu. První dva příkazy vytvoří vazby `x` na 1 a `y` na 2. Po vykonání třetího příkazu se vazba `x` změní na 2. Všiměme si, že nyní jsme přišli o informaci jaká hodnota byla původně navázána na `x`. Čtvrtý příkaz změní vazbu `y` na aktuální hodnotu proměnné `x`, kterou je hodnota 2. Pátý i šestý příkaz tedy vytiskne číslici `2`.

Řešení spravíme tím, že si před provedením třetího příkazu uložíme hodnotu proměnné `x` do pomocné proměnné `z`.

```
x = 1
y = 2
z = x
x = y
y = z
print(x)
print(y)
```

Krok za krokem si projděte program ve studiu.

Programátor by měl program psát především tak, aby byl srozumitelný pro programátora, který program bude číst. Tím může být buď jiné programátor, nebo jeho tvůrce, který po nějaké době zapomene, jak program fungoval. Z tohoto důvodu je dobré do programu dodávat prázdné řádky, které logicky oddělují části kódu a hlavně komentáře. Pokud na nějakém řádku je znak `#`, tak vše, co je za tímto znakem až do konce řádku, se počítá za komentář a do programu nepatří. Předchozí program můžeme tedy zpřehlednit tak, jak je ukázáno na Obrázku 1.

Dokážete v následující kostře programu chybějící místo doplnit tak, aby se hodnota proměnné `y` dostala do proměnné `x`, hodnota proměnné `z` do proměnné `y` a konečně hodnota proměnné `x` do proměnné `z`? Program by tedy měl postupně vytisknout čísla 2, 3 a 1.

```
x = 1
y = 2
z = 3
?
```

```
# Prohození vazeb proměnných.

# vstupní hodnoty
x = 1
y = 2

z = x # proměnná z je pomocná
x = y
y = z

# tisk výstupních hodnot
print(x)
print(y)
```

Obrázek 1: Program s komentáři.

```
print(x)
print(y)
print(z)
```

Část programu, kde se zadávají hodnoty proměnných, budeme nazývat vstupem, a část, která tiskne výsledek programu, výstupem. Předchozí úkol by tedy šel formulovat takto: Pro tři zadané hodnoty x_1, x_2 a x_3 na vstupu vraťte na výstupu hodnoty x_2, x_3 a x_1.

Úkol 1.1. Jsou dány délky stran obdélníku. Vraťte jeho obvod a obsah.

Úkol 1.2. Pro zadanou dálku hrany spočtěte obsah krychle.

Úkol 1.3. Kolik vteřin trvá časový úsek zadaný počtem vteřin, minut, hodin a dnů?

Úkol 1.4. Rychlost světla ve vakuu je přibližně 299 792 458 m/s. Jedna astronomická jednotka (AU) má 149 597 870 700 m. Přibližně za kolik vteřin urazí světlo ve vakuu zadaný počet astronomických jednotek? Vzdálenost Slunce a Země je přibližně 1 AU, vzdálenost Slunce a Pluta je přibližně 40 AU.

Úkol 1.5. Pro zadaná přirozená čísla n_1 a n_2 vraťte nulu v případě, že jsou obě sudá. Lze problém vyřešit tak, že prostě vždy vrátíme nulu? Co když zadání změníme tak, že program musí vrátit nenulové číslo v opačném případě.

Úkol 1.6. Vraťte číslo jedna, pokud je zadané číslo sudé, jinak vraťte nulu.

Úkol 1.7. Vraťte nulu právě tehdy, když je zadané číslo dělitelné třemi, pěti a sedmi. Lze program zjednodušit tak, aby se počítal zbytek po dělení pouze jednou?

Úkol 1.8. Máme dána dvě trojciferná čísla n_1 a n_2. Úkolem je vrátit číslo, jehož cifry budou nejdříve cifry čísla n_1 a poté cifry čísla n_2.

Úkol 1.9. Vraťte všechny cifry zadaného trojciferného čísla.

Úkol 1.10. Prohoďte první a třetí cifru trojciferného čísla.

Úkol 1.11. Vraťte nulu právě tehdy, když je čtyřciferné číslo palindrom. Číslo je palindrom pokud se čte stejně zleva doprava i zprava doleva. Například číslo 1221 je palindrom, ale číslo 1222 palindrom není.

Úkol 1.12. Jsou dány cifry čtyřciferného čísla v dvojkové soustavě. Převeďte číslo do desítkové soustavy.

Úkol 1.13. Pro číslo menší než 32 vraťte všechny jeho cifry v dvojkové soustavě.

Úkol 1.14. Upravte poslední cifru zadaného čísla tak, aby bylo výsledné číslo dělitelné třemi.

Úkol 1.15. Bod v rovině je určen celočíselnými souřadnicemi x a y. Souřadnice jsou v rozmezí $\langle 0, 100)$. Jaké souřadnice bude mít bod, když jej posuneme v x-ové ose o 10 a v y-ové ose o 20 jednotek a poté dvakrát vzdálíme od počátku?

Úkol 1.16. Pomocí až čtyřciferného čísla $c_1c_2c_3c_4$ reprezentujeme dvě až dvouciferná čísla c_1c_2 a c_3c_4. Například číslo 1234 reprezentuje čísla 12 a 34. V případě, že by cifry c_1 a c_2 chyběly, bere se první číslo rovné nule. Tedy 12 reprezentuje čísla 0 a 12. Nyní máme na vstupu dvě takové až čtyřciferné reprezentace dvojce čísel. Úkolem je vrátit reprezentaci dvojce součtu sobě odpovídajících čísel. Například pro 1234 a 2431 vrátíme 3665, protože $12 + 24 = 36$ a $34 + 31 = 65$. V případě, že by součet nějaké složky vyšel tříciferný, první cifru zahodíme. Tedy například pro 8912 a 9000 vrátíme 7912, protože $89 + 90 = 179$, $12 + 0 = 12$ a u čísla 179 zahodíme cifru 1 a dostaneme 79.

Úkol 1.17. Pomocí až čtyřciferného čísla $c_1c_2c_3c_4$ reprezentujeme desetinné číslo c_1c_2, c_3c_4. Například číslo 1234 reprezentuje desetinné číslo $12, 34$. Cifry přiřazujeme odzadu. Tedy číslo 12 reprezentuje desetinné číslo $0, 12$. Mějme zadány dvě reprezentace desetinných čísel. Cílem je vrátit reprezentaci jejich součtu. Tedy pro 1212 a 2111 vrátíme 3323 protože $12, 12 + 21, 11 = 33, 23$. V případě, že by vznikla při součtu pěticiferná reprezentace, první cifru zahodíme. Tedy pro 9999 a 1 vrátíme 0, protože $99, 99 + 0, 01 = 100$. Reprezentace součtu je 10000 a zahozením první cifry obdržíme nulu.

Úkol 1.18. Pro zadané přirozené číslo vraťte součet všech přirozených čísel, která jsou menší nebo rovno než toto číslo.

2 Druhý seminář

2.1 Operátor umocňování

Na začátek si představíme z pohledu vyhodnocování netradiční binární operátor mocniny **. Levý podvýraz určuje mocněnce a pravý mocnitele. Proto tedy platí:

```
>>> 5 ** 2
25
```

Neboli $5^2 = 25$. Aby byl výsledek celočíselný, omezíme se na situace, kde mocnitel je nezáporné číslo. Záludné je, že mocnina má vyšší prioritu než unární `-x`. Proto překvapivě platí:

```
>>> -1 ** 2
-1
```

To z toho důvodu, že výraz `-1**2` se vyhodnotí stejně jako `-(1**2)`. Nyní již výsledek není překvapením. Další výjimka spočívá v tom, že mocnina se na rozdíl od všech ostatních operátorů vyhodnocuje zprava do leva. Proto

```
>>> 2 ** 2 ** 3
256
```

počítá, že $2^{(2^3)} = 2^8 = 256$, tedy stejně jako `2 ** (2 ** 3)`. To odpovídá konvenci mocnění v matematice, kde platí, že $b^{p^q} = b^{(p^q)}$. Vyhodnocení zleva doprava musíme vynutit uzávorkováním:

```
>>> (2 ** 2) ** 3
64
```

Spočítali jsme, že $(2^2)^3 = 4^3 = 64$.

Zopakujeme si prioritu operátorů. Nejmenší prioritu má součet `+` a rozdíl `-` dále jsou násobek `*`, dělení `//` a zbytek po dělení `*` poté následuje opačné číslo `-x` a největší prioritu má mocnina `**`.

2.2 Pravdivostní hodnoty

Jediný typ hodnot, se kterým jsme dosud pracovali, byla celá čísla. Celých čísel je potencionálně nekonečně mnoho. Mohli jsme vytvořit libovolně velké celé číslo. Nyní si uvedeme nový typ hodnot, který bude naopak velmi malý, pouze dvě hodnoty budou tohoto typu. Zavedeme typ *pravdivostní hodnota*, který bude obsahovat pouze hodnotu *pravda* a hodnotu *nepravda*.

Hodnotu *pravda* získáme přečtením výrazu `True` a hodnotu *nepravda* přečtením výrazu `False`. Pravdivostní hodnoty se tisknou zpět na tyto výrazy. Pravdivostní hodnoty se stejně jako čísla vyhodnocují sami na sebe. Obecně lze říci, že každá hodnota se vyhodnotí sama na sebe. Proto platí následující.

```
>>> True
True
>>> False
False
```

Čísla a pravdivostní hodnoty patří mezi literály. Literál je zápis hodnoty přímo v programu. Proměnné naopak mezi literály nepatří. Jejich hodnota je určena až za běhu programu.

Uvedeme tři operátory nezývané *pravdivostní operátory*, které pracují s pravdivostními hodnotami: `or`, `and` a `not`. Operátory `or` a `and` jsou binární a operátor `not` je unární. Operace přiřazené operátorům jsou dány následujícími tabulkami.

`or`	`True`	`False`
`True`	`True`	`True`
`False`	`True`	`False`

`and`	`True`	`False`
`True`	`True`	`False`
`False`	`False`	`False`

`not`	
`True`	`False`
`False`	`True`

Proto platí:

```
>>> True or False
True
>>> True and False
False
>>> not True
False
```

Pravdivostní operace můžou mít jako své operandy libovolné hodnoty. Například výraz `1 and 2` má hodnotu 2. My se na semináři omezíme pouze na operandy, které jsou pravdivostní hodnoty.

Každý z pravdivostních operátorů má jinou prioritu. Nejmenší prioritu má operátor `or`, dále je `and` a nakonec je operátor `not` s nejvyšší prioritou. Priority pravdivostních operátorů jsou ilustrovány následujícím příkladem, kde oba výrazy mají stejný význam.

```
>>> not False and not True or True
True
>>> ((not False) and (not True)) or True
True
```

K zjednodušení výrazů můžeme použít znalost zákonů logiky. Můžeme například použít zákon dvojí negace a dostaneme následující tvrzení. Pro libovolný výraz v platí, že výraz `not not` v má stejnou hodnotu jako výraz v. Budeme také říkat, že tyto výrazy jsou ekvivalentní. Běžně také budeme používat takzvané De Morganovy zákony. Pro výrazy v_1 a v_2 platí, že výraz

`not` v_1 `and not` v_2

je ekvivalentní výrazu

`not` (v_1 `or` v_2 `)`

a výraz

`not` v_1 `or not` v_2

je ekvivalentní výrazu

`not` (v_1 `and` v_2 `)`.

Pro implikaci nemáme žádný operátor a musí být vyjádřena podle zákonu o náhradě implikace disjunkcí. Tedy chceme-li pro výrazy v_1 a v_2 vyjádřit, že v_1 implikuje v_2, vytvoříme výraz

`not` v_1 `or` v_2.

Úkol 2.1. Za použití zákonů logiky zjednodušte výrazy

1. `not (not x or not y)`,
2. `x and x and x`,
3. `x and y or x and z`.

Ukážeme si další operátory nazývané porovnávací operátory, které se vyhodnocují na pravdivostní hodnoty. Patří mezi ně binární operátory <, <=, >, >=, != (nerovnost) a == (rovnost). Operátory <, <=, >, >= pro číselné argumenty porovnávají hodnoty čísel. Například

```
>>> 1 < 0
False
>>> 2 < 2
False
>>> 2 <= 2
True
>>> 3 > 2
True
>>> 4 >= 5
False
```

Operátory rovnosti a nerovnosti lze použít na libovolné hodnoty. Tedy jak na čísla tak na pravdivostní hodnoty.

```
>>> 100 == 100
True
>>> True != False
True
```

Všimněme si, že rovnost zapisujeme dvěma znaky rovnítka (==), protože jeden znak rovnítka máme vyhrazený pro příkaz přiřazení. Zapomenutí druhého rovnítka vede k časté chybě:

```
>>> 10 = 10
  File "<stdin>", line 1
SyntaxError: cannot assign to literal
```

Zde se ve skutečnosti snažíme přiřadit hodnotu 10 literálu `10`, což není možné.

Operátory porovnání mají menší prioritu než aritmetické operátory (`+`, `*`, ...) a větší prioritu než pravdivostní operátory (`or`, `and` a `not`). Proto platí, že následující dva výrazy se vyhodnotí přesně stejně.

```
>>> 10 + 5 == 15 and -1 == 0 - 1
True
>>> ((10 + 5) == 15) and ((-1) == (0 - 1))
True
```

Operátory porovnání mají obvyklé vlastnosti, které známe z matematiky. Například pro výrazy v_1 a v_2 platí, že výraz

v_1 `!=` v_2

je ekvivalentní výrazu

`not` v_1`==` v_2

a

v_1 `<` v_2

je ekvivalentní výrazu

v_2 `>` v_1.

Zaměřme se nyní na vyhodnocení pravdivostních operátorů `and` a `or`. Platí, že jejich podvýrazy se vyhodnocují jen, pokud je to nutné. Výraz

```
True or 5 + a == 10
```

bude vždy pravdivý bez ohledu na vazbu proměnné `a`. Vyhodnocování výrazu se tedy nebude obtěžovat vyhodnocením podvýrazu `5 + a == 10` a vrátí pravdu. Podobně výraz

```
1 != 1 and a < 5
```

bude vždy nepravdivý.

Říkáme, že vyhodnocení operátorů `or` a `and` je líné. Podívejme se ještě na jeden příklad

```
a == 0 or 20 % a == 0
```

Pokud by se operátor `or` nevyhodnocoval líně, skončilo by vyhodnocení pro `a` rovno 0 chybou při pokusu dělit nulou. Líné vyhodnocování však nejdříve vyhodnotí podvýraz `a == 0` a pokud je pravdivý, vrátí rovnou pravdu. Výraz tedy vrací pravdu, pokud je `a` rovno nule nebo není rovno nule a dělí číslo dvacet.

Poznamenejme, že při uplatňování pravidel logiky musíme být obezřetní v případech, kdy vyhodnocení výrazu může skončit chybou. Předchozí výraz tedy není ekvivalentní výrazu

```
20 % a == 0 or a == 0
```

a to přesto, že u logické spojky nebo nezáleží na pořadí spojovaných výroků. Druhý výraz skončí chybou pro `a` rovno 0.

Úkol nás může vyzvat, že máme rozhodnout, zda platí nějaké tvrzení. Vezměme si například následující úkol.

> Jsou dána dvě celá čísla a a b. Rozhodněte, jestli je a menší než b.

V takovém případě je náš cíl napsat program, který má na vstupu dvě čísla a a b. Program vrátí `True`, jestliže tvrzení platí ($a < b$). V opačném případě musí vrátit `False`. Řešení by mohlo vypadat takto:

```
# vstup
a = 1
b = 2

# porovnání
result = a < b

# výstup
print(result)
```

Úkol 2.2. Pro zadaná přirozená čísla a, b a c rozhodněte, zda platí $a^2 + b^2 = c^2$.

2.3 Větvení programu

Dostáváme se k představení příkazu, který nám umožňuje vykonat určité příkazy jen, pokud je splněna zadaná podmínka. Přesněji pokud máme výraz v a příkazy $p_1, p_2, \ldots, p_n$ pak

```
if v:
  p1
  p2
  ...
  pn
```

je *příkaz větvení*. Příkazy $p_1, \ldots, p_n$ jsou odsazené tabulátorem. První řádek

```
if v:
```

se nazývá *hlavička* a příkazy $p_1, \ldots, p_n$ se nazývají *tělo* příkazu větvení. Předpokládáme, že výraz v má pravdivostní hodnotu. Příkaz větvení se vykoná tak, že se nejdříve vyhodnotí výraz v. Pokud je jeho hodnota pravda, vykonají se postupně výrazy $p_1, \ldots, p_n$ v opačném případě vykonávání skončí.

Ukážeme si příklad použití.

```
a = 5
if a < 10:
  print(a)
```

Byl použit příkaz větvení, kde výraz v je `a < 10`, $n = 1$ a příkaz p_1 je `print(a)`. Spuštění programu nejprve vytvoří vazbu proměnné `a` na hodnotu 5 (první řádek), dále vyhodnotí `a < 10` na hodnotu `True`. Protože hodnota je pravda, vykoná se tělo příkazu větvení. Přesněji se vykoná příkaz `print(a)` a hodnota `a` se vytiskne. Pokud v prvním řádku změníme pravou stranu od rovnítka na `10`, výraz v hlavičce větvení se vyhodnotí na `False` a vykonání těla se neprovede. Program skončí bez tisku jakékoliv hodnoty.

Příkaz větvení můžeme použít na výpočet absolutní hodnoty zadaného čísla.

```
x = -5
if x < 0:
  x = -x
print(x)
```

Všimněte si, že konec příkazu větvení je dán odsazením jeho těla. Následující část programu je tedy příkaz větvení.

```
if x < 0:
  x = -x
```

Příkaz měnící znaménko hodnoty proměnné `x` se provede jen, pokud je číslo záporné. Projděte si krok po kroku program. Krokem projděte program pro hodnoty 0 a 10.

Pokud odstraníme dvojtečku v příkazu větvení, dojde k chybě:

```
   if x < 0
          ^
SyntaxError: invalid syntax
```

Tělo větvení se může skládat z více příkazů. Následující program prohodí hodnoty `a` a `b`, pokud je `b` menší než `a`.

```
a = 4
b = 2
if b < a:
  c = a
  a = b
  b = c
print(a)
print(b)
```

Vyzkoušejte si program pro `a` rovno 2 a `b` rovno 4.

Samozřejmě můžeme použít víc příkazů větvení za sebou.

```
a = 10
if a < 100:
  print(1)
if a >= 0:
  print(2)
```

Co program dělá? Zkuste jej spustit pro hodnoty 120 a -1.

Protože příkaz větvení je opět příkazem, můžeme jej použít v roli příkazu p_i pro nějaké i v těle jiného příkazu větvení.

```
a = 4
if a % 2 == 0:
  if a == 4:
    print(1)
  print(2)
```

Co program vytiskne? Co vytiskne pro hodnoty 2 a 1?

Zde příkaz větvení

```
if a == 4:
  print(1)
```

je použit v těle příkazu větvení:

```
if a % 2 == 0:
  if a == 4:
    print(1)
  print(2)
```

Další chybou by bylo, když by nebyly všechny příkazy v těle příkazu větvení stejně odsazeny. Pokus o spuštění programu

```
a = 0
if a == 0:
  print(1)
    print(2)
```

skončí chybou

```
line 4
    print(2)
    ^
IndentationError: unexpected indent
```

Úkol 2.3. Realizujte znaménkovou funkci, též známou jako funkce signum. Funkce pro kladné hodnoty vrátí číslo 1, pro záporné číslo -1 a pro 0 vrátí 0.

Úkol 2.4. Jsou dány tři celá čísla a, b a c. Rozhodněte, zda a náleží do otevřeného intervalu (b, c).

Úkol 2.5. Úkolem je napsat program, který žákovi přiřadí známku z bodované písemky. Jsou dány bodové hranice pro jednotlivé známky x_4, x_3, x_2 a x_1. Předpokládáme, že platí $x_4 < x_3 < x_2 < x_1$. Dále je dán počet bodů, které žák získal. Aby žák například dostal čtyřku, musí získat aspoň x_4 a méně než x_3 bodů.

Úkol 2.6. Seřaďte tři čísla podle velikosti.

Úkol 2.7. Jsou zadány tři délky (nezáporná čísla). Rozhodněte, zda je možné sestrojit trojúhelník, jehož strany budou mít zadané délky.

Úkol 2.8. Rozhodněte, zda je trojúhelník, u něhož známe délky všech tří stran, pravoúhlý.

Úkol 2.9. Jsou dány velikosti vnitřních úhlů trojúhelníku, vraťte jedna, pokud je trojúhelník ostroúhlý, dva, pokud je pravoúhlý a tři, pokud se jedná o tupoúhlý trojúhelník. Rozhodněte, zda je trojúhelník, u něhož známe délky všech tří stran, pravoúhlý.

Úkol 2.10. Rozhodněte, zda čtyři daná čísla tvoří aritmetickou posloupnost.

Úkol 2.11. Pro bod vraťte číslo kvadrantu. Pravý horní kvadrant má číslo jedna, levý horní dva, levý spodní tři a konečně pravý dolní čtyři.

Úkol 2.12. V souřadnicovém systému je dán bod a obdélník. Bod souřadnicemi p_x a p_y, obdélník souřadnicemi levého horního rohu r_x a r_y dále šířkou w a výškou h. Rozhodněte, zda bod leží uvnitř obdélníku.

Úkol 2.13. Do jakých kvadrantů zasahuje úsečka daná souřadnicemi koncových bodů?

Úkol 2.14. Některá desetinná čísla můžeme zapsat ve tvaru $s \cdot 10^e$, kde s je trojciferné číslo a e je celé číslo. Například $5 = 500 \cdot 10^{-2}$, $0,1 = 100 \cdot 10^{-3}$ a $1200 = 120 \cdot 10^1$. Naopak číslo 1234 v tomto tvaru zapsat nelze. Číslo tohoto tvaru tedy můžeme reprezentovat dvojicí čísel s a e. Napište program, který sečte dvě čísla tohoto tvaru. Výsledek musí být opět zadaného tvaru. Například pro $120 \cdot 10^0$ a $300 \cdot 10^{-2}$ program vrátí $123 \cdot 10^0$ nebo pro $999 \cdot 10^1$ a $100 \cdot 10^{-2}$ program vrátí $100 \cdot 10^2$. V případě potřeby můžete zaokrouhlovat. Například pro $901 \cdot 10^0$ a $100 \cdot 10^0$ program vrátí $100 \cdot 10^1$ nebo pro $100 \cdot 10^5$ a $100 \cdot 10^0$ program vrátí $100 \cdot 10^5$.

3 Třetí seminář

3.1 Klauzule příkazu větvení

Vezměme si následující program, který vrátí číslo jedna, pokud je vstup větší jak dvacet a jinak vrátí číslo dvě.

```
x = 10

if x > 20:
  y = 1
if x <= 20:
  y = 2

print(y)
```

Všimněme si, že bude platit právě jedna z podmínek `x > 20` a `x <= 20`. Pokud známe pravdivost první podmínky, nemá smysl vyhodnocovat druhou podmínkou - její pravdivost již známe.

Rozšíříme si příkaz větvení, aby jednoduše vyjádřil podobné situace. Nejdříve si zavedeme pojem bloku. *Blok* je

p_1
...
p_n

kde $p_1, \ldots, p_n$ jsou příkazy. Neboli blok je posloupnost příkazů. Nyní pokud v je podmínka, b_1 a b_2 bloky, pak

`if` v`:`
  b_1
`else:`
  b_2

je další forma příkazu větvení. Částem

`if` v`:`
  b_1

a

`else:`
  b_2

se říká *klauzule* příkazu větvení. Každá klauzule má hlavičku a tělo. Hlavička začíná klíčovým slovem a končí dvojtečkou. Tělo je odsazený blok příkazů. Tedy příkaz větvení může mít jednu nebo dvě klauzule. Vykonání příkazu větvení s klauzulí `else` probíhá tak, že se nejprve vyhodnotí podmínka v. Pokud je pravdivá, vykoná se blok b_1, jinak se vykoná blok b_2.

Úvodní příklad je tedy možné úsporněji zapsat následovně.

```
x = 10

if x > 20:
  y = 1
else:
  y = 2

print(y)
```

Podívejme se na program počítající znaménkovou funkci:

```
n = 10

if n > 0:
  signum = 1
if n < 0:
  signum = -1
if n == 0:
  signum = 0

print(signum)
```

S použitím klauzule `else` příkazu větvení jej můžeme přepsat následovně.

```
n = 10

if n > 0:
  signum = 1
else:
  if n < 0:
    signum = -1
  else:
    signum = 0

print(signum)
```

Můžeme si představit, že program se dělí do tří větví. Zde musíme použít vnořené větvení, protože zatím příkaz větvení umožňuje rozdělit program do dvou větví.

Rozšíříme si příkaz větvení tak, aby mohl program rozdělit do libovolného počtu větví. Pokud $v_1, v_2 \ldots, v_n$ jsou podmínky a $b_1, b_2 \ldots, b_n, b_{n+1}$ jsou bloky, pak

```
if v1:
  b1
elif v2:
  b2
```

```
...
elif v_n:
  b_n
else:
  b_{n+1}
```

je další forma příkazu větvení.

Vidíme, že jsme umožnili přidávat klauzule `elif` mezi klauzule `if` a `else`. Příkaz se vykoná tak, že se postupně vyhodnocují podmínky $v_1, v_2, \ldots, v_n$ až se narazí na první pravdivou. Řekněme, že první pravdivá podmínka je v_i. Pak se vykoná blok b_i a vykonávání příkazu skončí. Pokud by žádná podmínka nebyla pravdivá, vykoná se blok b_{n+1}.

Znaménkovou funkci lze za použití klauzule `elif` zapsat pouze jedním příkazem větvení takto:

```
n = 10

if n > 0:
  signum = 1
elif n < 0:
  signum = -1
else:
  signum = 0

print(signum)
```

Poznamenejme, že klauzule `else` je nepovinná a že podmínky $v_1 \ldots, v_n$ nemusí být výlučné. Program, který pro záporná čísla vytiskne jedničku a pro čísla menší než deset dvojku lze zapsat následovně.

```
n = -2

if n < 0:
  print(1)
elif n < 10:
  print(2)
```

Zde dojde jen k tisku čísla jedna, přestože podmínka `n < 10` klauzule `elif` je pravdivá. Pro `n` rovno 10 program nic nevytiskne.

Část o příkazu větvení ukončíme shrnutím. Příkaz musí začínat klauzulí `if`, následovat může několika klauzulemi `elif` a může končit jednou klauzulí `else`.

3.2 Iterace

Představme si, že chceme nějaký kus kódu opakovat pětkrát pouze se změnou jisté hodnoty v kódu. Například chceme postupně vytisknout všechna nezáporná čísla menší než pět. Můžeme to provést následujícím programem.

```
print(0)
print(1)
print(2)
print(3)
print(4)
```

Lze úkol splnit tak, aby příkaz tisku byl při každém kroku stejný? Odpověď je pozitivní:

```
i = 0
print(i)
i = 1
print(i)
i = 2
print(i)
i = 3
print(i)
i = 4
print(i)
```

Touto technikou můžeme blok příkazů opakovat, ale počet opakování musí být předem zadán. Vidíme, že v bloku máme k dispozici číslo opakování v proměnné `i`. Co ale když počet opakování neznáme předem? Chceme například vytisknout všechna nezáporná celá čísla menší než zadané číslo. Za tímto účelem zavedeme příkaz cyklu `for`. Jedná se podobně jako příkaz větvení o složený příkaz. Složené příkazy jsou příkazy, které se skládají s klauzulí, jejichž těla obsahují opět příkazy. Mějme jméno proměnné i, výraz v jehož hodnota je celé nezáporné číslo a blok b, pak

```
for i in range(v):
  b
```

je příkaz cyklu `for`. Tento příkaz se vykoná tak, že se nejdříve vyhodnotí výraz v a tím se získá číslo n udávající počet opakování. Dále se n krát vykonají příkazy bloku b. Před každým vykonáním bloku se nastaví hodnota proměnné i na číslo opakování, které se počítá od nuly. Tedy při prvním opakování bude i nastaveno na 0, při druhém na 1 a tak dále.

Výše uvedený program lze tedy přepsat následovně.

```
for i in range(5):
  print(i)
```

Počet opakování již může být proměnnou:

```
n = 5

for i in range(n):
  print(i)
```

Všimněte si, že proměnná i je nastavena před každým vykonáním těla cyklu. Její změnou tedy nelze ovlivnit počet opakování. Následující program například desetkrát vytiskne nulu bez ohledu na příkaz `i = i + 1`.

```
for i in range(10):
  print(0)
  i = i + 1
```

Pokud proměnná i měla před vykonáním cyklu nějakou vazbu, je tato vazba změněna. Navíc z toho jak vykonání cyklu probíhá plyne, že po skončení cyklu bude mít proměnná i vazbu na číslo poslední iterace. Například následující program desetkrát vytiskne nulu a poté devítku.

```
i = 5
for i in range(10):
  print(0)
print(i)
```

Pokud počet opakování n vyjde rovný nule. Příkaz cyklu `for` se rovnou ukončí. Například program

```
 for i in range(0):
  print(1)
```

nic neudělá.

Co když budeme chtít vytisknout všechna sudá čísla menší nebo rovno než zadané číslo? Můžeme postupovat dvojím způsobem. Zaprvé je možné vložit příkaz větvení do příkazu cyklu:

```
n = 10

for i in range(n):
  k = i + 1
  if k % 2 == 0:
    print(k)
```

Zadruhé můžeme upravit počet opakování:

```
n = 10

for i in range(n // 2):
  print((i + 1) * 2)
```

Následující program vytiskne všechny dělitele zadaného čísla.

```
# Vytiskne všechny dělitele zadaného čísla.
n = 100
```

```
for i in range(n):
  k = i + 1
  if n % k == 0:
    print(k)
```

Předchozí program stačí mírně upravit a obdržíme program rozhodující o tom, zda je dané číslo prvočíslem.

```
# Rozhodne, zda je číslo prvočíslo.
n = 7

divisor_count = 0
for i in range(n):
  k = i + 1
  if n % k == 0:
    divisor_count = divisor_count + 1

is_prime = divisor_count == 2

print(is_prime)
```

V těle cyklu může být další cyklus. Předchozí program rozhodující o tom, zda je číslo prvočíslo, můžeme upravit tak, aby vytiskl všechna prvočísla menší nebo rovno než zadané číslo:

```
# Vytiskne všechna prvočísla menší nebo rovno než zadané číslo.
m = 1000

for j in range(m):
  n = j + 1
  divisor_count = 0
  for i in range(n):
    k = i + 1
    if n % k == 0:
      divisor_count = divisor_count + 1
  is_prime = divisor_count == 2
  if is_prime:
    print(n)
```

Před zadáním úkolů si rozšíříme příkaz tisku o možnost vytisknout více hodnot na jeden řádek. Pokud $v_1, \ldots, v_n$ jsou výrazy, pak

`print(`v_1`,`$,\ldots,$`,`v_n`)`

je rozšíření příkazu tisku. Příkaz se vykoná tak, že postupně vyhodnotí výrazy $v_1, \ldots, v_n$ a hodnoty vytiskne za sebe oddělené mezerou. Například

```
>>> print(1+1, 2, True or True)
2 2 True
```

Speciálním případem je použití příkazu tisku bez výrazů v_i. Příkaz `print()` pouze vytiskne prázdný řádek.

```
>>> print()

>>>
```

Úkol 3.1. Je dán první člen a_1 a rozdíl mezi sousedními členy d aritmetické posloupnosti. Vytiskněte prvních n členů této posloupnosti.

Úkol 3.2. Sečtěte prvních n členů aritmetické posloupnosti dané prvním členem a_1 a rozdílem sousedních členů d. Nesmíte použít součtový vzorec.

Úkol 3.3. Vytiskněte všechny dělitele dvou zadaných přirozených čísel.

Úkol 3.4. Rozhodněte, zda jsou dvě zadaná přirozená čísla nesoudělná.

Úkol 3.5. Je dáno přirozené číslo n. Vytiskněte všechna přirozená čísla menší než n, která jsou s n nesoudělná.

Úkol 3.6. Vytiskněte všechny trojce a, b, c přirozených čísel menších než zadané číslo, pro které platí $a^2 + b^2 = c^2$.

Úkol 3.7. Jsou dána přirozená čísla n a $m > 2$. Rozhodněte, zda existují přirozená čísla a, b, c menší nebo rovno než n a přirozené číslo k větší než 2 a menší nebo rovno než m taková, že $a^k + b^k = c^k$. (Podle Velké Fermatovy věty musí být odpověď vždy záporná.)

Úkol 3.8. Prvočíselné dvojče je prvočíslo, které je buď o dva větší, nebo o dva menší než jiné prvočíslo. Vytiskněte každé prvočíselné dvojče menší nebo rovno než zadané číslo.

Úkol 3.9. Spočítejte faktoriál $n!$ zadaného čísla n. Faktoriál nuly je jedna $0! = 1$ a pro faktoriál nenulového n platí $n! = n \cdot (n-1)!$.

Úkol 3.10. Je dáno přirozené číslo n. Vytiskněte prvních n prvků Fibonacciho posloupnosti. První dva prvky posloupnosti jsou nula a jedna. Každý další prvek je součtem dvou předchozích prvků.

Úkol 3.11. Je dáno přirozené číslo n a celá čísla a_0 a q. Vypište prvních n členů geometrické posloupnosti začínající prvkem a_0 a s kvocientem q.

Úkol 3.12. Vraťte součet prvních n členů geometrické posloupnosti začínající celým číslem a_0 s kvocientem q bez použití součtového vzorce.

3.3 Tisk řetězce znaků

Nejprve si zavedeme nový typ hodnot: řetězce. Řetězce jsou hodnoty, které uchovávají posloupnosti znaků. Můžeme je vložit do programu podobně jako čísla ve formě literálů tak, že do apostrofů umístíme znaky řetězce. Přesněji pokud s je posloupnost znaků, pak

`'`s`'`

je řetězec. Podobně jako číslo i řetězec je výraz a jelikož se jedná o hodnotu, vyhodnocuje se sám na sebe. Proto například platí

```
>>> 'Python'
'Python'
```

Výraz `'Python'` vytvoří řetězec obsahující šest znaků: `P`, `y`, `t`, `h`, `o`, `n`. Můžeme vkládat i znaky s diakritikou:

```
'Příliš žluťoučký kůň úpěl ďábelské ódy.'
```

Jak vidíme, řetězec může obsahovat i mezery. Některé znaky můžeme vložit pouze pomocí takzvané escape sekvence. Například znak apostrofu vložíme znaky `\'`. Tedy `'\''` je řetězec délky jedna obsahující apostrof. Dále máme sekvenci `\n` pro nový řádek a sekvenci `\\` pro zpětné lomítko. Nový řádek je tedy jeden znak v řetězci.

```
>>> print('a\nb')
a
b
```

Speciálním případem je řetězec `''`. Jedná se o řetězec, který neobsahuje žádný znak.

Alternativně lze zadat řetězec umístěním jeho znaků do uvozovek:

```
>>> "Python"
'Python'
```

Tisk řetězce používá přednostně apostrofy, ale v případě, že řetězec obsahuje apostrof a neobsahuje uvozovky použijí se k tisku uvozovky:

```
>>> '\''
"'"
```

Při použití uvozovek k zadání řetězce je samozřejmě potřeba uvozovky v řetězci zadávat escape sekvencí:

```
>>> "Karel řekl: \"Vida, prší.\""
'Karel řekl: "Vida, prší."'
```

Zatím si nepředstavíme žádné operátory pracující s řetězci. Řetězce budeme používat pouze k tisku.

Pokud chceme v příkazu tisku zamezit tisku nového řádku použijeme následující jeho formu. Pro výrazy $v_1, \ldots, v_n$ je výraz

```
print(v1, ..., vn, end='')
```

forma příkazu tisku, která při vykonání po tisku hodnot nevytiskne nový řádek. Například:

```
>>> print(1, end='')
1>>>
```

Následující program vytiskne čtverec hvězdiček o hraně n.

```
# Vytiskne čtverec hvězdiček o zadané hraně.
n = 10

for i in range(n):
  for j in range(n):
    print('*', end='')
  print('')
```

Výstup programu je:

```
**********
**********
**********
**********
**********
**********
**********
**********
**********
**********
```

Úkol 3.13. Pro zadanou délku vytiskněte čtverec hvězdiček s prázdným vnitřkem. Například pro 5 program vytiskne

```
*****
*   *
*   *
*   *
*****
```

Úkol 3.14. Pro zadané přirozené číslo n vytiskněte pravoúhlý rovnoramenný trojúhelník hvězdiček s rameny délky n.

```
*
**
***
****
*****
```

Úkol 3.15. Pro zadaný počet pater vytiskněte trojúhelník hvězdiček podobný níže uvedenému. Například pro pět pater vypadá trojúhelník takto:

```
    *
   ***
  *****
 *******
*********
```

Úkol 3.16. Pro zadané přirozené číslo n, vytiskněte z hvězdiček diamant, který bude mít $n \cdot 2 + 1$ pater. Například pro číslo tři má diamant sedm pater a vypadá následovně.

```
   *
  ***
 *****
*******
 *****
  ***
   *
```

4 Čtvrtý seminář

4.1 Rozšířený příkaz přiřazení

Vezměme si jednoduchý program, který číslo zvětší o jedna.

```
n = 5
n = n + 1
print(n)
```

Zde příkaz `n = n + 1` zvýší hodnotu proměnné `n` o jedna. Zvýšení hodnoty proměnné o libovolnou hodnotu můžeme provést úsporněji následujícím příkazem. Pokud i je jméno proměnné a v je výraz, tak

i += v

je rozšířený příkaz přiřazení. Vykonání příkazu proběhne stejně, jako bychom napsali:

i = i + v

Výše uvedený program můžeme tedy přehledněji napsat následovně.

```
n = 5
n += 1
print(n)
```

Podobně lze použít rozšířený příkaz přiřazení pro všechny binární operátory. Přesněji je-li i jméno proměnné, o jeden z operátorů `+`, `-`, `*`, `//` nebo `%` a v výraz, pak

$$i\ o{=}\ v$$

je rozšířený příkaz přiřazení. Vykonání příkazu proběhne stejně, jako by byl místo něj uveden příkaz:

$$i = i\ o\ v$$

Co vytiskne následující program?

```
n = 1
n += 1
n **= 3
n //= 2
n -= 1
n *= 2
n %= 4
print(n)
```

4.2 Podmínečné opakování

Vraťme se k tisku cifer trojciferného čísla:

```
n = 123

n1 = n

c = n1 % 10
n1 = n1 // 10
print(c)

c = n1 % 10
n1 = n1 // 10
print(c)

c = n1 % 10
n1 = n1 // 10
print(c)
```

S použitím rozšířeného příkazu přiřazení lze program přepsat takto:

```
n = 123

n1 = n

c = n1 % 10
n1 //= 10
print(c)

c = n1 % 10
n1 //= 10
print(c)

c = n1 % 10
n1 //= 10
print(c)
```

K dalšímu zjednodušení můžeme použít příkaz for cyklu:

```
n = 123

n1 = n
for i in range(3):
  c = n1 % 10
  n1 //= 10
  print(c)
```

Co když budeme chtít program upravit tak, aby vytiskl všechny cifry zadaného čísla? Narazíme na problém, že neumíme získat počet cifer čísla. Program bychom rádi upravili tak, aby tělo cyklu probíhalo, dokud bude číslo `n1` nenulové. Za tímto účelem zavedeme následující příkaz.

Pokud v je podmínka a b blok, pak

```
while v:
  b
```

je příkaz podmíněného opakování, nebo-li příkaz `while` cyklu. Podobně jako příkaz `for` cyklu, se jedná o složený příkaz s jednou klauzulí `while`. Příkaz se vykoná tak, že se opakuje následující. Nejdříve se vyhodnotí podmínka v. Pokud je podmínka pravdivá, vykoná se blok b, jinak se vykonávání příkazu ukončí.

S použitím příkazu podmíněného opakování můžeme náš program napsat takto:

```
n = 123

n1 = n
```

```
while n1 != 0:
  c = n1 % 10
  n1 //= 10
  print(c)
```

Cyklus zde nejdříve zkontroluje, zda `n1` je nenulové. Pokud by `n1` bylo nula, pak by program skončil. Jinak do proměnné `c` uloží poslední cifru `n1`, z `n1` odstraní poslední cifru a cifru `c` vytiskne. Vše funguje díky tomu, že z čísla `n1` postupně ubývají cifry až nakonec získáme nulu.

Všimněte si, že program nefunguje pro číslo nula. Dokážete ho spravit?

Cyklus `while` do programů může přinést nový druh chyb. Dosud programy vždy skončily. Nyní se může stát, že program bude počítat navždy a nikdy neskončí. Triviální příklad takového programu je:

```
while True:
  print(0)
```

Program po spuštění bude donekonečna tisknout nuly. Říkáme, že program *cyklí*. Jeho činnost musíte v terminálu ukončit kombinací kláves `Ctrl`+`C`.

Ne vždy je pád do nekonečné smyčky takto průzračný. Vezměme si na ukázku následující program.

```
n = 10

n1 = n
while n1 != 0:
  print(n1)
  n1 -= 2
```

Zdá se, že program tiskne čísla menší nebo rovno než zadané číslo a přitom každé druhé vynechává. Co se ale stane, když jej spustíme pro devítku? Program vytiskl:

```
9
7
5
3
1
```

Pak přeskočil nulu a pokračuje dále

```
-1
-3
-5
-7
```

směřujíce k zápornému nekonečnu. Program tedy někdy skončí a jindy cyklí. Chyba je v podmínce `n1 != 0`. Dokážete ji spravit tak, aby program vždy skončil?

Vezměme si obecný příklad `for` cyklu:

```
for i in range(v):
  b
```

Zde i je jméno proměnné, v výraz a b blok. Předpokládejme, že blok b nemění hodnotu proměnné i a nepoužívá proměnnou `n`. Pak cyklus `for` můžeme přepsat pomocí cyklu `while` následovně.

```
n = v
i = 0
while i < n:
  b
  i += 1
```

Například program

```
for i in range(5):
  print(i)
```

lze ekvivalentně zapsat takto:

```
n = 5
i = 0
while i < n:
  print(i)
  i += 1
```

Cyklus `while` nemůžeme obecně přepsat na cyklus `for` a to z toho důvodu, že cyklus `for` narozdíl od cyklu `while` vždy skončí. Konečnost vykonávání `for` cyklu je velikou výhodou. Proto se budeme snažit tam, kde je to možné, upřednostnit `for` cyklus před `while` cyklem. Lze říci, že `for` cyklus používáme, když známe dopředu počet opakování.

Vraťme se k rozhodování, zda je číslo prvočíslem:

```
n = 5

is_prime = True
for i in range(n - 2):
  j = i + 2
  if n % j == 0:
    is_prime = False

print(is_prime)
```

Pro číslo tisíc program už po první iteraci ví, že není prvočíslo (podmínka `1000 % 2 == 0` je pravdivá), ale přesto pokračuje zbytečně dál. Pojďme program upravit tak, aby skončil, jakmile zjistí, že číslo není prvočíslem. Nejprve přepíšeme `for` cyklus `while` cyklem:

```
n = 5

is_prime = True
j = 2
while j < n:
  if n % j == 0:
    is_prime = False
  j += 1

print(is_prime)
```

Nyní již stačí upravit podmínku cyklu:

```
n = 5

is_prime = True
j = 2
while j < n and is_prime:
  if n % j == 0:
    is_prime = False
  j += 1

print(is_prime)
```

Následuje několik úkolů na procvičení podmíněného opakování.

Úkol 4.1. Vraťte ciferný součet přirozeného čísla.

Úkol 4.2. Vraťte cifraci přirozeného čísla. Cifrace čísla n je číslo n v případě, že n je jednociferné. V opačném případě je to cifrace ciferného součtu čísla n. Například pro 99, nejprve spočítáme ciferný součet $9 + 9 = 18$, protože 18 není jednociferné číslo, proces opakujeme. Ciferný součet čísla 18 je $1 + 8 = 9$ a to je i výsledek cifrace.

Úkol 4.3. Vytiskněte rozklad přirozeného čísla na prvočísla.

Úkol 4.4. Je dáno přirozené číslo. Vraťte číslo jehož cifry jsou cifry daného čísla čtené pozpátku.

Úkol 4.5. Rozhodněte, zda je libovolné přirozené číslo palindrom.

Úkol 4.6. Vraťte celou část logaritmu přirozeného čísla o daném základu.

Úkol 4.7. Rozhodněte, zda jsou dvě čísla nesoudělná. Ukončete program, jakmile zjistíte netriviálního dělitele.

Úkol 4.8. Je dán počet cifer n. Vytiskněte všechna n-ciferná čísla.

Úkol 4.9. Vraťte počet cifer zadaného přirozeného čísla.

Úkol 4.10. Jsou dána přirozená čísla n a k, kde $k \leq n$. Vraťte nejmenšího dělitele čísla n, který je větší nebo rovno než číslo k.

Úkol 4.11. Je dáno číslo n. Vytiskněte prvních n dokonalých čísel. Číslo k je dokonalé, jestliže součet dělitelů k menších než k je roven k. Například šestka je dokonalé číslo, protože $6 = 1 + 2 + 3$.

Úkol 4.12. Pomocí Eukeidova algoritmu spočítejte největšího společného dělitele dvou čísel.

Úkol 4.13. Za použití řešení předchozího úkolu spočítejte nejmenší společný násobek dvou čísel.

Úkol 4.14. Vraťte celou část odmocniny nezáporného čísla.

Úkol 4.15. Upravte rozhodování prvočíselnosti tak, aby program zkoušel jen čísla menší než odmocnina z daného čísla. Využijte výsledek z předchozího úkolu.

5 Pátý seminář

5.1 Přerušení iterace

Někdy by bylo výhodné přerušit iteraci způsobenou příkazem cyklu `for`. Vraťme se opět k testu prvočíselnosti.

```
n = 7

is_prime = True
for i in range(n - 2):
  j = i + 2
  if n % j == 0:
    is_prime = False

print(is_prime)
```

Pokud chceme cyklus přerušit v momentě, kdy narazíme na netriviálního dělitele, museli bychom jej nyní přepsat na `while` cyklus. Nepříjemné by ale bylo, že bychom ztratili záruku konečného vykonávání, kterou cyklus `for` přináší. Proto si zavedeme nový příkaz `break` nazývaný *příkaz přerušení cyklu*, který lze použít v těle `for` cyklu. Příkaz přerušení cyklu se vykoná tak, jak ostatně název napovídá, že se okamžitě ukončí nejvnitřnější cyklus, ve kterém se vykonávání nachází.

Program můžeme tedy s úspěchem přepsat takto:

```
n = 7

is_prime = True
for i in range(n - 2):
```

```
  j = i + 2
  if n % j == 0:
    is_prime = False
    break

print(is_prime)
```

Nyní se po nalezení netriviálního dělitele cyklus ukončí.

Příkaz ukončení cyklu se musí nacházet v těle cyklu. Proto spuštění programu

```
print(1)
break
print(2)
```

končí chybou `SyntaxError: 'break' outside loop`. Všimněte si, že se jedná o chybu zápisu, která se odhalí ještě před spuštěním programu.

Vezměme si nyní následující program, který tiskne dvojce nezáporných čísel menších než zadané číslo.

```
n = 10

for i in range(n):
  for j in range(n):
    print(i, j)
```

Co když budeme chtít program upravit tak, aby první číslo bylo menší nebo rovno než druhé číslo. První verze by byla následující.

```
n = 10

for j in range(n):
  for i in range(n):
    if i <= j:
      print(i, j)
```

Za použití větvení tiskneme jen některé dvojce. Tato varianta však není efektivní, protože zbytečně procházíme spoustu dvojic čísel. Program můžeme vylepšit vhodně umístěným příkazem přerušení cyklu:

```
n = 10

for j in range(n):
  for i in range(n):
    if j < i:
      break
    print(i, j)
```

Kód funguje, protože `break` vyskočí z `for` cyklu s iterační proměnnou `i`. Je ale použití příkazu přerušení cyklu nutné? Neumíme program napsat lépe bez něj? Odpověď je kladná. Můžeme upravit počet opakování vnořeného cyklu:

```
n = 10

for j in range(n):
  for i in range(j + 1):
    print(i, j)
```

Tím jsme zvýšili čitelnost programu. Proto si zavedeme pravidlo, že `break` budeme používat pouze v případech, kdy bychom museli jinak přepsat `for` cyklus na `while` cyklus.

5.2 Volání funkce

Zastavme se na chvíli u příkazu tisku. Vezměme si například

```
>>> print(1)
1
```

Ve skutečnosti tento příkaz volá funkci `print` s argumentem 1. Obecněji je-li f jméno funkce a $v_1, \ldots, v_n$ jsou výrazy, pak

f`(`v_1`, ...,` v_n`)`

je *volání funkce*. Každé volání funkce je výrazem. Za názvem funkce f nepíšeme mezeru, ale za každou čárkou oddělující podvýrazy ano. Výraz se vyhodnotí tak, že se postupně vyhodnotí výrazy $v_1, \ldots, v_n$ tím získáme hodnoty $h_1, \ldots, h_n$. Poté zavoláme funkci f s argumenty $h_1, \ldots, h_n$. Volání funkce vrátí hodnotu, která je i hodnotou výrazu volání funkce.

Následující příkazy jsou výrazy volající funkci `print`.

```
print(1, 2, 3)
print()
print(1, end='')
```

Poslední volání obsahuje takzvaný pojmenovaný argument `end`, který volání připouští, ale naše definice jej zatím nepostihuje. Použití pojmenovaného argumentu si dovolíme pouze u funkce `print`.

Volání funkce je výraz, musí tedy mít nějakou hodnotu. Podívejme se, jakou hodnotu má volání funkce `print`.

```
>>> print(print(1))
1
None
```

Tisk jedničky provedlo volání `print(1)`, které vrátilo hodnotu `None` vytištěnou následujícím voláním funkce `print`. Hodnotu `None` budeme nazývat *prázdná hodnota*. Prázdná hodnota je jediná hodnota svého typu (*typu prázdné hodnoty*). Připomeňme, že známe hodnoty různého typu. Dosud známé typy jsou čísla, pravdivostní hodnoty, řetězce a nyní ještě typ prázdné hodnoty. Prázdnou hodnotu dáváme tam, kde chceme sdělit, že zde ve skutečnosti žádná hodnota není. To je příklad hodnoty volání (*návratové hodnoty*) funkce `print`. V interaktivním režimu platí, že prázdnou hodnotu interpret ve fázi tisku netiskne. Tedy:

```
>>> None
>>>
```

Jméno proměnné nesmí kolidovat s názvem funkce. Proto následující program skončí chybou.

```
print = 1
print(1)
```

Chybě `TypeError: 'int' object is not callable` je třeba rozumět tak, že po změně hodnoty proměnné `print` na číslo jedna, přestalo být `print` funkcí a není již možné tuto funkci zavolat. Názvy funkcí poznáte tak, že se ve studiu zabarví modře. Zakážeme si tyto názvy používat jako názvy proměnných. Tedy `print` je zakázaný název proměnné.

5.3 Práce s řetězci

Dosud umíme pouze řetězce vytvářet tak, že jejich znaky obklopíme uvozovkami. Například výraz `'Python'` má hodnotu řetězec se znaky `Python`. Nyní si ukážeme, jak s řetězci pracovat.

Funkce `len` bere jako svůj argument řetězec a vrací jeho délku. Například:

```
>>> len('Python')
6
>>> len('')
0
```

Jednoprvkový řetězec je *znak*. Tedy například řetězec `'P'` je i znakem.

Pokud v_s a v_i jsou výrazy, pak

v_s`[`v_i`]`

je *výraz indexačního operátoru*. Výraz se vyhodnotí tak, že se nejprve vyhodnotí výraz v_s a tím se získá hodnota h_s, pak se vyhodnotí výraz v_i a tím se získá hodnota h_i. Jestliže hodnota h_s je řetězec a hodnota h_i celé nezáporné číslo menší než délka řetězce h_s, pak hodnota indexačního operátoru je $(h_i + 1)$-tý znak řetězce h_s. Říkáme taky znak řetězce h_s na indexu h_i.

Následují příklady použití.

```
>>> 'Python'[0]
'P'
>>> s = 'Python'
>>> s[0]
'P'
>>> s[5]
'n'
```

Hodnota výrazu `s[0]` je tedy první znak řetězce `s`.

Pokus získat znak na indexu větším nebo rovném než je délka řetězce vede k chybě.

```
>>> s[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

S pomocí iterace můžeme napsat program tisknoucí všechny znaky zadaného řetězce.

```
string = 'Python'

for i in range(len(string)):
  print(string[i])
```

Binární operátor + lze použít k spojování řetězců. Přesněji pokud v_1 a v_2 jsou výrazy jejichž hodnoty jsou řetězce s_1 a s_2, pak hodnota v_1 + v_2 je řetězec vzniklý spojením řetězců s_1 a s_2. Proto

```
>>> 'Py' + 'thon'
'Python'
```

Operátor + slouží jak k sčítání čísel, tak k spojování řetězců. Oba operandy však musí být buď řetězce, nebo čísla. Proto získání hodnot následujících výrazů skončí chybou:

```
>>> '1' + 1
TypeError: can only concatenate str (not "int") to str
>>> 1 + '1'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Výpis chyb je zkrácený pouze na chybovou hlášku.

Následující program otočí pořadí znaků v zadaném řetězci.

```
string = 'Python'

reverse = ''
```

```
for i in range(len(string)):
  reverse = string[i] + reverse

print(reverse)
```

Operátory == a != lze použít k porovnávání řetězců. Přitom dva řetězce jsou stejné, pokud mají stejnou délku a znaky na odpovídajících indexech se rovnají. Proto

```
>>> string = 'Python'
>>> string == 'Python'
True
>>> 'p' != 'P'
True
>>> 'python' == 'Python'
False
```

Všimněte si, že porovnávání je citlivé na velikost písmen. Ve skutečnosti můžeme porovnávat na rovnost čísla a řetězce.

```
>>> 1 == '1'
False
>>> '1' != 1
True
```

Platí, že hodnoty různých typů se nerovnají.

Otočení řetězce v kombinaci s porovnáváním řetězců můžeme použít k rozhodnutí, zda je program palindrom

```
string = 'kobylamamalybok'

reverse = ''
for i in range(len(string)):
  reverse = string[i] + reverse

is_palindrom = string == reverse

print(is_palindrom)
```

Program lze napsat i bez otáčení řetězce prostě tak, že porovnáváme odpovídající znaky:

```
string = 'kobylamamalybok'

is_palindrom = True
string_len = len(string)
i = 0
```

```
n = string_len // 2
while i < n and is_palindrom:
  if string[i] != string[string_len - 1 - i]:
    is_palindrom = False
  i += 1

print(is_palindrom)
```

Každému znaku je jednoznačně přiřazené nezáporné číslo. Představíme si dvě funkce, které převádí mezi sebou znaky a čísla. Funkce `chr` vrací znak k zadanému číslu a funkce `ord` vrací číslo zadaného znaku. Čísla některých znaků určuje ASCII tabulka. Velká písmena anglické abecedy se začínají číslovat od 65. Tedy například znak `A` má číslo 65, znak `'L'` číslo 76 a poslední znak `'Z'` číslo 90:

```
>>> ord('A')
65
>>> ord('L')
76
>>>  ord('Z')
90
```

Malá písmena anglické abecedy se číslují od 97:

```
>>> ord('a')
97
>>> ord('m')
109
>>> ord('z')
122
```

Tedy malé písmeno má o 32 větší číslo než stejné velké písmeno.

Znaky odpovídající číslicím jsou seřazené podle hodnoty a číslují se od 48:

```
>>> ord('0')
48
>>> ord('1')
49
>>> ord('5')
53
>>> ord('9')
57
```

Pro převedení číslice reprezentované jako znak na jednociferné číslo stejné hodnoty musíme od čísla číslice odečíst 48:

```
>>> ord('0') - 48
0
```

Podívejme se na program, který převede řetězec napsaný malými písmeny na velká písmena:

```
string = 'python'

upper_case = ''
for i in range(len(string)):
  upper_case += chr(ord(string[i]) - 32)

print(upper_case)
```

Následují úkoly na práci s řetězci.

Úkol 5.1. Převeďte řetězec znaků číslic na číslo zapsané těmito číslicemi. Tedy pro řetězec `'456'` vraťte číslo `456`.

Úkol 5.2. Převeďte zadané číslo na řetězec znaků jeho číslic. Například pro číslo `123` vraťte řetězec `'123'`.

Úkol 5.3. Rozhodněte, zda je česká věta zadaná jako řetězec palindrom. Řetězec neobsahuje diakritická znaménka. Při kontrole ignorujte velikost písmen, mezery a interpunkci. Tedy řetězec `'Kobyla ma maly bok.'` je palindrom.

Úkol 5.4. Vytiskněte všechny slova obsažená v řetězci, kde sousední slova jsou oddělená mezerou. Například pro `'jablko banán hruška'` vytiskněte:

```
jablko
banán
hruška
```

Úkol 5.5. Odstraňte nadbytečné mezery z řetězce obsahující českou větu. Například pro řetězec `'  Kobyla   má     malý bok.  '` vraťte `'Kobyla má malý bok.'`.

Úkol 5.6. Vytiskněte indexy všech výskytů řetězce p v řetězci s. Například pro řetězec p roven `štros` a s roven
`'Pštros s pštrosicí a pštrosáčaty šli do pštrosačárny.'` vytiskněte:

```
1
10
22
41
```

Co program vytiskne, když bude p rovno prázdnému řetězci? Je to správně?

Úkol 5.7. Pro řetězec s a indexy i_s a i_e vraťte podřetězec řetězce s všech znaků s indexem i, kde $i_s \leq i < i_e$. Například pro s rovno `'Python'`, i_s rovno 2 a i_e rovno 5 vraťte `'tho'`. Co když nebudete předpokládat, že $i_s \leq i_e$?

Úkol 5.8. Rozhodněte, zda dva řetězce obsahují stejné znaky, když máte povoleno porovnávat pouze řetězce délky jedna (znaky).

Úkol 5.9. Pro zadané řetězce s, s_p, s_t, kde s_p není prázdný, vraťte řetězec, který vznikne z řetězce s tak, že se všechny výskyty řetězce s_p nahradí řetězcem s_t. Program pro s rovno `'Dal jsem jablko do košíku.'`, s_p rovno `'jablko'` a s_t rovno `'hrušku'` vrátí `'Dal jsem hrušku do košíku.'`. Nahrazení `'11'` za `'2'` v řetězci `'111111'` musí vrátit `'222'`. Program musí být schopný i odstraňovat podřetězce. Tedy například při nahrazení `' '` v `'1 23 4 56'` řetězcem `''` vrátí `'123456'`.

Úkol 5.10. Převeďte kladné číslo menší jak sto na zápis v římských číslicích. Nepoužívejte odčítací pravidla. Tedy 4 je `'IIII'`.

Úkol 5.11. Vraťte ke kladnému číslu menšímu jak devadesát jeho zápis v římských číslicích. Používejte odčítací pravidla. Například pro 14 vraťte `'XIV'`.

Úkol 5.12. Vraťte hodnotu řetězce obsahujícího zápis čísla za použití římských číslic I, V, X a L bez odčítacích pravidel. Tedy pro `'XIIII'` vraťte 14.

Úkol 5.13. Převeďte řetězec obsahující zápis přirozeného čísla menšího než devadesát římskými číslicemi na číslo. Zápis může používat odečítací pravidla. Například pro `'XLIX'` vraťte 49.

Úkol 5.14. Pro přirozené číslo vraťte řetězec cifer jeho zápisu v dvojkové soustavě. Například pro 15 vraťte `'1111'`.

Úkol 5.15. Převeďte řetězec obsahující zápis čísla v dvojkové soustavě na číslo. Tedy pro `'10101'` vraťte 21.

Úkol 5.16. Vytiskněte ASCII znaky a jejich kódy od 32. do 126. znaku včetně.

Úkol 5.17. Je dán řetězec s a nezáporné číslo n. Zašifrujte řetězec s obsahující pouze malá písmena tak, že každé písmeno posunete v ASCII kódu o n pozic doprava. Při pokusu vyjít ven z malých písmen se vraťte na začátek. Tedy považujte `'a'` za následníka `'z'`. Například řetězec `'mraz'` pro n rovno 2 zakódujte na `'otcb'`.

Úkol 5.18. Pokud je potřeba, upravte předchozí program tak, aby mohl zakódované slovo dešifrovat. Tedy aby pro `'otcb'` a -2 vrátil zpět `'mraz'`. Nápověda: Podívejte se, jak funguje zbytek po dělení ze záporného čísla.

6 Šestý seminář

6.1 Konstanty

Začneme převodem slova napsaného malými písmeny na velká písmena.

```
string = 'python'

upper_case = ''
for i in range(len(string)):
```

```
  upper_case += chr(ord(string[i]) - 32)

print(upper_case)
```

V programu se vyskytuje záhadná hodnota 32. Když jsme program psali, věděli jsme co znamená. Při čtení programu za delší dobu však její význam nemusí být zřejmý. Jedna možnost, jak program učinit čitelnější, by byla doplnit komentář, který by hodnotu vysvětloval. My se však vydáme druhou možností, která nás nabádá k tomu si hodnotu pojmenovat.

Zavedeme si tedy proměnnou `letters_distance` s hodnotou 32.

```
string = 'python'

letters_distance = 32
upper_case = ''
for i in range(len(string)):
  upper_case += chr(ord(string[i]) - letters_distance)

print(upper_case)
```

Z názvu je zřejmé, že proměnná vyjadřuje vzdálenost mezi písmeny. Z povahy programu už nás trkne, že se jedná o vzdálenost čísel malých a velkých písmen.

Proměnná `letters_distance` je výjimečná v tom, že se po běhu programu nemění její hodnota. Takovým proměnným budeme říkat *konstanty* a budeme je psát velkými písmeny.

```
string = 'python'

LETTERS_DISTANCE = 32
upper_case = ''
for i in range(len(string)):
  upper_case += chr(ord(string[i]) - LETTERS_DISTANCE)

print(upper_case)
```

Ještě můžeme učinit jeden krok k zlepšení čitelnosti programu, kterým je doplnit výpočet hodnoty konstanty.

```
string = 'python'

LETTERS_DISTANCE = ord('a') - ord('A')
upper_case = ''
for i in range(len(string)):
  upper_case += chr(ord(string[i]) - LETTERS_DISTANCE)

print(upper_case)
```

Z výpočtu je zřejmé, jakou hodnotu konstanta uchovává.

Zavedeme si pravidlo, které nás povede k tomu si pro hodnoty v programu zavádět konstanty.

6.2 Desetinná čísla

Dosud jsme se zabývali pouze celými čísly. V této části si představíme způsob práce s desetinnými čísly. Například číslo $0,2$ s desetinnou čárkou zadáme pomocí desetinné tečky.

```
>>> 0.1
0.1
```

Aritmetické operátory pracující také s desetinnými čísly.

```
>>> 0.1 + 0.1
0.2
>>> 0.5 * 0.2
0.1
```

Celé číslo může být zadáno také jako desetinné číslo s použitím desetinné tečky a nuly v desetinné části.

```
 >>> 1.0
1.0
```

Pokud je jeden z operandů aritmetické operace celé číslo a druhý desetinné číslo, je celé číslo převedeno na desetinné číslo.

```
>>> 2 * 0.1
0.2
```

Výsledek operace s desetinnými čísly je vždy desetinné číslo.

```
>>> 2 * 0.5
1.0
```

Zavedeme si nový binární aritmetický operátor dělení (`/`). Jeho výsledkem je vždy desetinné číslo.

```
>>> 4 / 2
2.0
>>> 0.1 / 0.2
0.5
```

Desetinné číslo také dostaneme při použití operátoru mocniny se záporným mocnitelem.

```
>>> 2 ** -3
0.125
```

Při zadávání desetinných čísel lze použít i vědeckou notaci. Číslo ve tvaru $a \times 10^b$ zapíšeme jako

aeb

Použitím vědecké notace můžeme v kilogramech pohodlně vyjádřit jak nejmenší hmotnost atomu $1,67 \times 10^{-27}$ výrazem `1.67e-27` tak hmotnost sluce $1,9891 \times 10^{30}$ výrazem `1.9891e30`.

Při tisku interpret používá vědeckou notaci pouze pro čísla s větší absolutní hodnotou exponentu.

```
>>> 1e-2
0.01
>>> 1.23e2
123.0
>>> 10000000000000000.0
1e+16
>>> 0.00001
1e-05
```

Počítání s desetinnými čísly je pouze přibližné. Proto s překvapením zjistíme, že podmínka `0.1 + 0.1 + 0.1 == 0.3` není splněná.

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

Důvodem je, že interpret uchovává desetinná čísla v binární podobě a to pouze určitý počet číslic. Problém se vyjasní, když si představíme, že chceme uchovat číslo 1/3 jako desetinné číslo s určitým počtem platných míst. Můžeme například říci, že 1/3 se přibližně rovná číslu $0,333333$, které si označíme a. Jistě nikoho nepřekvapí, že $a + a + a$ není rovno jedné. V binární soustavě nastává analogický problém. Proto

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

Desetinná čísla budeme porovnávat pouze přibližně. Za tímto účelem si představíme funkci `abs`, která vrací absolutní hodnotu zadaného čísla. Funkce jako argument bere jak celá tak desetinná čísla.

```
>>> abs(-0.1)
0.1
>>> abs(0.1)
0.1
>>> abs(3)
3
>>> abs(-3)
3
```

S pomocí funkce `abs` můžeme napsat program, který rozhodne, zda se dvě desetinná čísla přibližně rovnají.

```
a = 0.3
b = 0.1 + 0.1 + 0.1

print(abs(a - b) < 1e-10)
```

Vidíme, že a je přibližně rovno b jestliže je absolutní hodnota jejich rozdílu menší než určitá tolerance. Zde je tolerance 10^{-10}. Absolutní hodnotě rozdílu čísel a, b budeme říkat *vzdálenost čísel* a, b. Proto můžeme říci, že vzdálenost čísel a, b je menší než tolerance.

Jak nás poučuje předchozí část, měli bychom pro toleranci vytvořit konstantu.

```
a = 0.3
b = 0.1 + 0.1 + 0.1

PRECISSION = 1e-10
print(abs(a - b) < PRECISSION)
```

Pro různé programy může být samozřejmě tolerance různá. Jiná bude u programu stroje operující lidské srdce a jiná u tachometru kola.

Interpret používá pro práci s desetinnými čísly formát čísel s plovoucí desetinnou čárkou. Tento formát se skládá ze tří částí: znaménka, platných číslic a exponentu. O omezení počtu platných číslic jsme si již řekli. Zbývá dodat, že i exponent je omezen. Proto existuje největší desetinné číslo

```
1.7976931348623157e+308
```

a nejmenší kladné desetinné číslo

```
5e-324
```
.

Existence největšího desetinného čísla vede k tomu, že pokud by operace měla vrátit číslo větší než největší možné, tak se vrátí nekonečno nebo dojde k chybě.

```
>>> 1e308 * 2
inf
>>> 2.0 ** 10000
OverflowError: (34, 'Result too large')
```

Stojí za zmínku, že celá čísla horní omezení velikosti teoreticky nemají.

```
 >>> 2 ** 10000
19950631168807583848...
```

Pokud by výsledek byl blíže nule než nejmenší kladné desetinné číslo, propadne se tento výsledek na nulu.

```
 >>> 5e-324 / 2
0.0
>>> (-5e-324) / 2
-0.0
```

Poznamenejme, že máme zápornou a kladnou nulu.

Při počítání s nekonečnem můžeme narazit na zvláštní hodnotu `nan`.

```
>>> inf = 1e+308 * 2
>>> inf / inf
nan
```

Hodnota `nan` je zkratka za *Not A Number* a je hodnotou neurčitých výrazů. Jako například ∞/∞. Označme si hodnotu `nan`.

```
>>> nan = inf / inf
```

Výsledky operací, kde aspoň jeden z operandů je `nan`, jsou opět `nan`.

```
>>> nan + 1
nan
>>> nan * nan
nan
```

Hodnota `nan` se nerovná ničemu dokonce ani sama sobě.

```
>>> nan == 1
False
>>> nan == nan
False
```

Přibližné výpočty s desetinnými čísly mají za důsledek to, že pokud bude sčítanec a o mnoho řádů větší než sčítanec b, pak může být součet a a b roven a. Například

```
>>> 10e20 + 1 == 10e20
True
```

Podobné problémy nastávají i u ostatních aritmetických operací.

Následují úkoly na procvičení práce s desetinnými čísly.

Úkol 6.1. Přibližně převeďte teplotu zadanou jako celé nebo desetinné číslo ve stupních Fahrenheita na stupně Celsia podle vztahu

$$c = \frac{5(f - 32)}{9},$$

kde f je teplota ve stupních Fahrenheita a c ve stupních Celsia. Výslednou hodnotu uveďte jako desetinné číslo.

Úkol 6.2. Použijte vztah zadaný v předchozím úkolu a napište program převádějící teplotu zadanou ve stupních Celsia na stupně Fahrenheita.

Úkol 6.3. Vytiskněte prvních n členů Fibonacciho posloupnosti. První člen je roven nule, druhý jedné a každý další je roven součtu dvou předchozích. Posloupnost tedy začíná $0, 1, 1, 2, 3, 5, 8, \ldots$

Úkol 6.4. Pro dané přirozené číslo $n > 1$ vypočítejte přibližnou hodnotu zlatého řezu rovnou poměru

$$\frac{a_n}{a_{n-1}},$$

kde a_i je i-tý člen Fibonacciho posloupnosti.

Úkol 6.5. Je dána přesnost d, řekněme 10^{-10}. Předchozí program nám dává posloupnost stále se zlepšujících odhadů zlatého řezu. Upravte jej tak, aby výpočet skončil ve chvíli, kdy se sousední členy odhadů zlatého řezu budou lišit o méně než d.

Úkol 6.6. Pro dané přirozené číslo n vraťte součet

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}.$$

Úkol 6.7. Pro dané přirozené číslo k vraťte n takové, že součet z předchozího úkolu bude větší nebo roven než k. Teoreticky pro každé k takové n existuje, prakticky však už pro malé k třeba 15 je n veliké.

Úkol 6.8. Spočítejte přibližnou hodnotu druhé odmocniny desetinného čísla a pomocí Newtonovy metody. Začněte libovolným odhadem x_0. Můžete položit x_0 rovno a. Dále počítejte prvky posloupnosti podle vztahu

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{a}{x_k}\right)$$

Jako výsledek pro zadané n vraťte prvek x_n.

Úkol 6.9. Upravte předchozí program tak, aby výpočet skončil ve chvíli, kdy vzdálenost druhé mocniny odhadu a čísla a bude menší nebo rovna než zadané d. Skončete tedy ve chvíli kdy

$$|x_k^2 - a| \leq d.$$

Úkol 6.10. Spočítejte přibližně hodnotu čísla π, když víte, že čtvrtina π je rovna

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

Program bude brát přirozené číslo n a do výpočtu zahrnete jen n členů výpočtu.

Úkol 6.11. Upravte předchozí program tak, aby se výpočet zastavil ve chvíli, kdy vzdálenost dvou následujících odhadů bude menší než zadaná přesnost.

7 Sedmý seminář

7.1 Funkce

Začněme motivačním příkladem. Vezměme si program, který počítá absolutní hodnotu čísla.

```
num = -4

if num < 0:
  num *= -1

print(num)
```

Pokud chceme program spustit s jiným vstupem, musíme změnit hodnotu proměnné `num`. To je nešikovné zejména pro testování. Program bychom rádi otestovali aspoň pro kladnou hodnotu, zápornou hodnotu a nulu. To by znamenalo vždy hodnotu změnit a program spustit. Chceme tedy náš kód vykonat vícekrát s různým vstupem. Za tímto účelem by bylo vhodné mít možnost si část programu pojmenovat, určit co jsou vstupní proměnné a co je výstupní hodnota. Právě k tomu slouží uživatelské funkce, které si v této části představíme.

Připomeňme si, že již umíme funkce volat tak, že určíme jméno funkce a argumenty volání.

```
>>> len('abc')
3
>>> abs(-2)
2
>>> print(1, 2)
1 2
```

Funkce určuje počet a typ svých argumentů. Například funkce `len` bere jeden argument typu řetězec. Volání funkce je výraz a jako každý výraz musí mít hodnotu. Například volání `len('abc')` má hodnotu 3. Této hodnotě říkáme návratová hodnota volání funkce. Říkáme také, že volání funkce vrací hodnotu.

Funkce, které máme k dispozici od startu programu, nazýváme *vestavěné funkce*. Tedy například `abs` je vestavěná funkce.

Dosud jsme používali pouze vestavěné funkce. Nyní si ukážeme, jak definovat funkce vlastní (nazývané *uživatelské funkce*). Pokud je f jméno funkce, $p_1, \ldots, p_n$ jsou názvy proměnných a b blok příkazů, pak

```
def f(p1, ..., pn):
  b
```

je *příkaz definující uživatelskou funkci*. Příkaz definující uživatelskou funkci je tedy složeným příkazem s jedinou klauzulí `def`. Proměnným $p_1, \ldots, p_n$ budeme říkat *parametry funkce* a bloku b tělo funkce. Příkaz se vykoná tak, že definuje (uživatelskou) funkci f.

Například

```
def print_sum(a, b):
  print(a + b)
```

definuje uživatelskou funkci `print_sum`, která má dva parametry `a`, `b`.

Jména funkcí nemůžou být jména vestavěných funkcí ani klíčová slova. Připomeňme, že klauzule složených příkazů začínají klíčovým slovem. Například `if`, `def` nebo `for` jsou klíčová slova. Navíc si musíme dát pozor, aby jména funkcí byla různá od jmen proměnných, které používáme.

Uživatelskou funkci můžeme zavolat výrazem volání funkce:

f`(`v_1`, ...,` v_m`)`

Nejprve se získají hodnoty $h_1 \ldots, h_m$ výrazů $v_1, \ldots, v_m$. Tím obdržíme m argumentů volání. Dále se zkontroluje, zda počet parametrů n funkce f se rovná počtu argumentů m. Pokud ne dojde k typové chybě (`TypeError`). Nyní víme, že $n = m$. Následně se nastaví pozičně odpovídající hodnoty parametrů $p_1, \ldots, p_n$ na argumenty $h_1, \ldots, h_n$. Nakonec se vykoná tělo funkce f.

Argumenty jsou tedy hodnoty, se kterými funkci voláme, a parametry jsou názvy proměnných, kterým se hodnoty nastavují.

Například vyhodnocení výrazu

```
print_sum(1 + 1, 2 + 2)
```

získá hodnoty 2 a 4 výrazů `1 + 1` a `2 + 2`. Protože funkce `print_sum` bere dva parametry a my ji voláme s dvěma argumenty, můžeme pokračovat a nastavit hodnotu proměnné `a` na 2 a hodnotu proměnné `b` na 4. Nakonec vykonáme tělo funkce

```
print(a + b)
```

To bude mít za následek tisk čísla 6.

Spuštění programu uloženého v souboru f z příkazové řádky příkazem

`python3 -i` f

přepne po vykonání programu interpret do interaktivního režimu.

Například pokud soubor `print_sum.py` obsahuje

```
def print_sum(a, b):
  print(a + b)
```

můžete jej spustit `python3 -i print_sum.py` a testovat funkci:

```
>>> print_sum(2, 3)
5
>>> print_sum(1, 0)
1
```

Pokud zadáme špatný počet argumentů, volání funkce skončí chybou.

```
>>> print_sum(1)
TypeError: print_sum() missing 1 required positional argument: 'b'
>>> print_sum(1, 1, 1)
TypeError: print_sum() takes 2 positional arguments but 3 were given
```

Návratová hodnota volání uživatelské funkce je zatím prázdná hodnota `None`. Návratovou hodnotu můžeme určit následujícím příkazem. Je-li v výraz, pak

```
return v
```

je *příkaz návratu*. Lze jej použít pouze v těle definice funkce. Slovo `return` je nové klíčové slovo. Příkaz se vykoná tak, že se ukončí vykonávání těla funkce a vrátí se hodnota výrazu v. Tedy hodnota výrazu v bude návratovou hodnotou volání funkce.

Zavedeme si omezení, že příkaz návratu musí být posledním příkazem, který by tělo funkce bez něj vykonalo. Například následující definice omezení porušuje.

```
def test(n):
  if n == 0:
    return 1
  return 2
```

Můžeme ji ale upravit následovně:

```
def test(n):
  if n == 0:
    return 1
  else:
    return 2
```

Často se budeme setkávat s definicemi funkcí, které jsou tohoto tvaru:

```
def f(p1, ..., pn):
  b
  return v
```

Vraťme se k našemu příkladu programu počítajícího absolutní hodnotu. Program můžeme vyjádřit funkcí:

```
def my_abs(num):
  if num < 0:
    num *= -1
  return num
```

Po načtení programu můžeme testovat:

```
>>> my_abs(-1)
1
>>> my_abs(1)
1
>>> my_abs(0)
0
```

Test můžeme učinit součástí programu:

```
def my_abs(num):
  if num < 0:
    num *= -1
  return num

print(my_abs(-1))
print(my_abs(1))
print(my_abs(0))
```

Vidíme, že program se skládá ze dvou částí. V první definujeme funkci, která určuje počet a jména parametrů, výpočet (tělo funkce) a návratovou hodnotu. V druhé části tuto funkci voláme a tiskneme návratové hodnoty. Můžeme tedy náš program zavolat vícekrát a zkontrolovat, zda se chová správně.

Od tohoto okamžiku budou naše programy mít tuto formu. Nejprve tedy bude uvedena definice funkce a pak příklady volání.

Zatím se omezíme na definici pouze jedné funkce vyjadřující náš program. To nám zatím stačí. K funkcím se ale budeme vracet v dalších seminářích.

7.2 Seznamy

Začneme programem, který tiskne čísla od nuly do n.

```
def print_range(n):
  for i in range(n):
    print(i)
```

Program funguje tak, že čísla se tisknou na výstup:

```
>>> print_range(5)
0
1
2
3
4
```

Tiskem sice posíláme čísla na výstup, ale přicházíme o ně. Co když ale chceme tato čísla postupně sbírat a dále s nimi pracovat? Za tímto účelem si zavedeme nový typ hodnot *seznam*. Jak název napovídá, seznam obsahuje hodnoty a určuje jejich pořadí. Seznam se v mnohém podobá řetězci. Hlavním rozdílem je, že oproti řetězci seznam může obsahovat libovolné hodnoty. Zatím se omezíme na seznamy, které obsahují čísla. Seznam můžeme vytvořit následovně. Jsou-li $v_1 \ldots, v_n$ výrazy, pak

`[`v_1`, ..., `v_n`]`

je *výraz popisující seznam*. Výrazy $v_1, \ldots, v_n$ určují *prvky* seznamu. Hodnotou výrazu je seznam, jehož prvky jsou hodnot výrazů $v_1 \ldots, v_n$. Oproti množině seznamy určují pořadí svých prvků a navíc prvky můžou opakovat.

Příklady seznamů následují.

```
>>> [0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
>>> [1 + 1, 2 * 3]
[2, 6]
>>> [1, 2 - 1, 1 + 1, 2 // 2]
[1, 1, 2, 1]
>>> []
[]
```

Poslední seznam je takzvaný *prázdný seznam*, který neobsahuje žádný prvek.

Seznam podobně jako řetězec má svoji délku a prvky seznamu se nacházejí na určitém indexu. Například `[4, 2, 7]` je seznam délky tři a na indexu nula se nalézá prvek 4, na indexu jedna prvek 2 a na indexu dva prvek 7. Prázdný seznam má nulovou délku.

Operátor + slouží také ke spojování seznamů.

```
>>> [1] + [2]
[1, 2]
>>> l1 = [2, 3]
>>> l2 = [4]
>>> l1 + l2
[2, 3, 4]
>>> l1 = l1 + [4]
>>> l1
[2, 3, 4]
```

Poznamenejme, že příkaz

p += v

není pro seznamy ekvivalentní příkazu

p = p + v

Jak následující kód ukazuje, příkaz += má z našeho pohledu zvláštní chování a proto jej zatím nebudeme používat.

```
 >>> l1 = [1, 2]
>>> l2 = l1
>>> l1 += [3]
>>> l1
[1, 2, 3]
>>> l2
[1, 2, 3]
```

Přesněji není jasné, proč se změnou hodnoty proměnné `l1` změnila i hodnota proměnné `l2`. Následující chování je již v pořádku:

```
>>> l1 = [1, 2]
>>> l2 = l1
>>> l1 = l1 + [3]
>>> l1
[1, 2, 3]
>>> l2
[1, 2]
```

Nyní již můžeme náš úvodní program přepsat tak, aby vracel seznam nezáporných čísel menších než n.

```
def list_range(n):
  result_list = []
  for i in range(n):
    result_list = result_list + [i]
  return result_list
```

V interaktivním režimu dostáváme

```
>>> list_range(5)
[0, 1, 2, 3, 4]
```

Nepřekvapí nás, že funkce `len` a indexační operátor fungují také se seznamy.

```
>>> l = [3, 2 , 1]
>>> len(l)
3
>>> len([4, 4])
2
>>> l[0]
3
>>> [1, 2, 3][0]
1
>>> [1][0]
1
```

Můžeme napsat funkci, která sečte hodnoty všech prvků seznamu. Vstupní proměnnou bychom nejraději pojmenovali `list`. To ale nemůžeme, protože `list` je jméno vestavěné funkce.

```
def list_sum(input_list):
  result = 0
  for i in range(len(input_list)):
    item = input_list[i]
    result += item
  return result
```

Máme například:

```
>>> list_sum([1, 2, 3])
6
```

Součet prvků prázdného seznamu nám vychází roven nule:

```
>>> list_sum([])
0
```

Dva seznamy se rovnají, když mají stejnou délku a prvky na odpovídajících si indexech se rovnají. K porovnávání seznamů můžeme využít operátory rovnosti a nerovnosti. Dostáváme

```
>>> [1, 2, 3] == [1, 2, 3]
True
>>> [1, 2, 3] != [1, 3, 2]
True
>>> [1, 1] == [1]
False
>>> [] == []
True
```

7.3 Úkoly

Úkol 7.1. Napište funkci, která pro n vrací seznam prvních n prvků Fibonnaciho posloupnosti. Pro definici se podívejte na úkol 6.3.

Úkol 7.2. Uměli byste předchozí úkol napsat tak, aby se při výpočtu používaly poslední dva prvky tvořeného seznamu?

Úkol 7.3. Napište funkci, která pro daný seznam čísel spočítá produkt jeho prvků. Produkt získáte tak, že vynásobíte všechny prvky seznamu. Co by měla funkce vrátit pro prázdný seznam?

Úkol 7.4. Napište funkci `reverse`, která pro seznam s vrátí seznam jehož prvky budou prvky seznamu s v opačném pořadí. Tedy

```
>>> reverse([2, 3, 4])
[4, 3, 2]
```

Úkol 7.5. Naprogramujte funkci `sublist`, která bere seznam s a indexy i_f a i_t. Funkce vrátí podseznam seznamu s obsahující prvky, jejichž index je větší nebo rovno než i_f a menší než i_t. Například

```
>>> sublist([5, 4, 3, 2, 1], 0, 3)
[5, 4, 3]
```

Co funkce vrátí, když $i_t \leq i_f$?

Úkol 7.6. Bez použití operátorů `==` a `!=` na porovnávání seznamů napište funkci, která rozhodne, zda jsou dva seznamy rovny.

Úkol 7.7. Napište funkci, která rozhodne, zda je hodnota prvkem seznamu.

Úkol 7.8. Napište funkci, která odstraní všechny výskyty zadané hodnoty ze seznamu.

Úkol 7.9. Upravte předchozí funkci tak, aby odstranila pouze první výskyt hodnoty.

Úkol 7.10. Napište funkci, která zjistí, zda je seznam uspořádaný od nejmenších hodnot k největším.

Úkol 7.11. Rozhodněte, zda jeden seznam začíná prvky druhého seznamu.

Úkol 7.12. Odstraňte ze seznamu všechny prvky, které jsou dělitelné zadaným číslem.

Úkol 7.13. Za použití Eratosthenova síta vraťte všechna prvočísla menší nebo rovno než dané n. Eratostenovo síto je algoritmus, který probíhá tak, že začnete se seznamem s čísel od dvou do n. Poté opakujete následující. První prvek seznamu s je prvočíslo a můžete jej přidat na výstup. Odstraňte ze seznamu s všechny prvky, které jsou dělitelné číslem a. Opakování ukončete, až bude seznam s prázdný.

8 Osmý seminář

8.1 Rozdělení programu do funkcí

Pojďme naprogramovat funkci odstraňující duplicity ze zadaného seznamu. Funkci pojmenujeme `remove_duplicates` a měla by fungovat následovně.

```
>>> remove_duplicates([1, 2, 1, 2, 2])
[1, 2]
>>> remove_duplicates([])
[]
>>> remove_duplicates([1, 2, 3])
[1, 2, 3]
```

Odstraňování duplicit budeme provádět tak, že do proměnné řekněme `result` budeme přidávat jen ty prvky vstupního seznamu, které se v ní ještě nenalézají. Proměnnou, jejíž hodnotu vracíme budeme nazývat *výstupní*. Začneme kostrou programu.

```
def remove_duplicates(input_list):
  result = []
  # ...
  return result
```

Přidáme procházení prvků seznamu `input_list`:

```
def remove_duplicates(input_list):
  result = []
  for i in range(len(input_list)):
    element = input_list[i]
    is_member = False
    # ...
    if not is_member:
      result = result + [element]
  return result
```

Zbývá dodat test, zda je aktuální prvek `element` přítomen v seznamu `result`.

```
1  def remove_duplicates(input_list):
2    result = []
3    for i in range(len(input_list)):
4      element = input_list[i]
5      is_member = False
6      for j in range(len(result)):
7        list_element = result[j]
8        if list_element == element:
9          is_member = True
10         break
11     if not is_member:
12       result = result + [element]
13   return result
```

Iterace na řádku 3 postupně prochází prvky vstupního seznamu. Iterace na řádku 6 kontroluje, zda je aktuální prvek přítomen ve výsledku. Pokud není, větvení na řádku 11 aktuální prvek přidá k výsledku.

Po chvilkové úvaze zjistíme, že řádky 5 až 10 zjišťují, zda je prvek přítomný v seznamu. Nevýhodou našeho řešení je, že tato skutečnost není na první pohled zřejmá. Chtěli bychom tuto část kódu pojmenovat a již víme, že za tímto účelem máme použít uživatelské funkce:

```
def is_element_member(element, input_list):
  is_member = False
  for i in range(len(input_list)):
    list_element = input_list[i]
    if list_element == element:
      is_member = True
      break
  return is_member
```

Funkci bychom měli otestovat:

```
>>> is_element_member(1, [1, 2, 1, 4])
True
>>> is_element_member(5, [1, 2, 1, 4])
False
```

```
>>> is_element_member(5, [])
False
>>> is_element_member(5, [5])
True
```

Nyní již můžeme nahradit ve funkci `remove_duplicates` problematický kód voláním funkce:

```
def is_element_member(element, input_list):
  is_member = False
  for i in range(len(input_list)):
    list_element = input_list[i]
    if list_element == element:
      is_member = True
      break
  return is_member

def remove_duplicates(input_list):
  result = []
  for i in range(len(input_list)):
    element = input_list[i]
    if not is_element_member(element, result):
      result = result + [element]
  return result
```

Podařilo se nám kód programu učinit přehlednější tím, že jsme jej rozdělili do více funkcí.

Jak je možné, že při volání funkce `is_element_member` nepřijdeme o hodnotu proměnné `input_list` funkce `remove_duplicates`, když prvně jmenovaná funkce má parametr také pojmenovaný `input_list`?

Při volání funkce se nejprve vytvoří nové prostředí. Nazýváme jej *lokální prostředí*. Připomeňme, že prostředí je tabulka, která přiřazuje proměnným jejich hodnoty. Hodnoty parametrů funkce se nastavují na argumenty v lokálním prostředí a tělo funkce se v tomto lokálním prostředí vykonává. Lokální prostředí jsou příčinou, proč změna hodnoty proměnné neovlivní hodnotu stejně pojmenované proměnné mimo tělo funkce.

Ukážeme si příklad.

```
1 def f1(b):
2   b = b + 1
3   return b
4
5 def f2(b):
6   f1(b)
7   return b
8
9 print(f2(2))
```

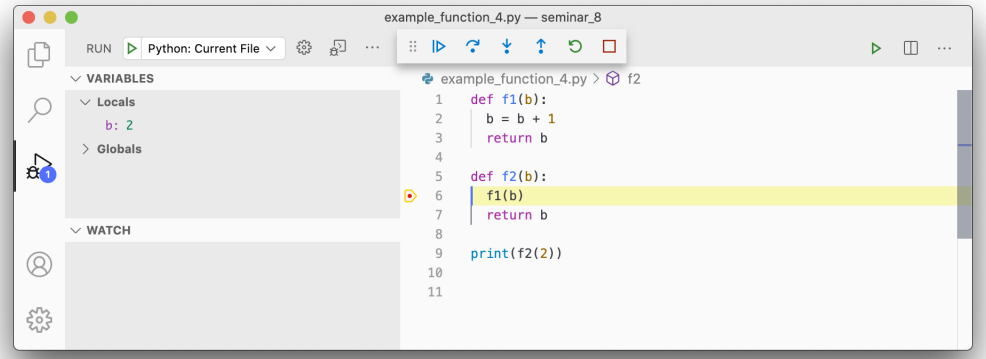
Co program vytiskne? Rozebereme si jeho vykonání. Příkazy na řádcích 1 a 5 pouze definují funkce. Samotný program se spustí vyhodnocením výrazu

```
f2(2)
```

Vytvoří se první lokální prostředí a v něm se nastaví hodnota proměnné `b` na 2:

b	2

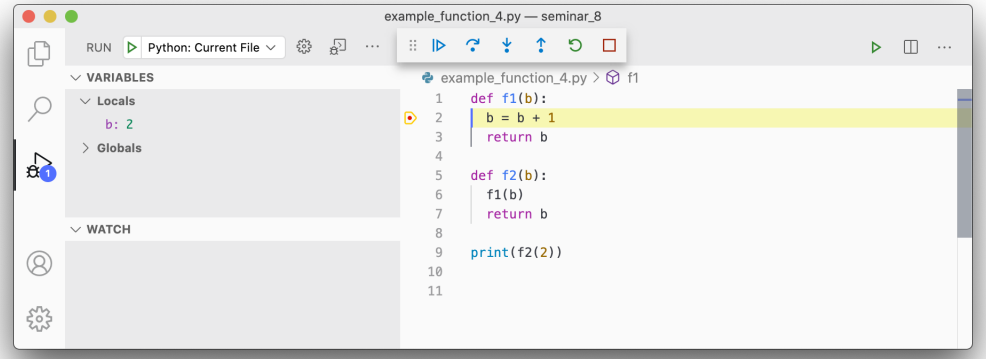
Do lokálního prostředí lze nahlédnout zastavením vykonávání na řádku 5. Lokální prostředí je zobrazené vlevo v části `Locals`.



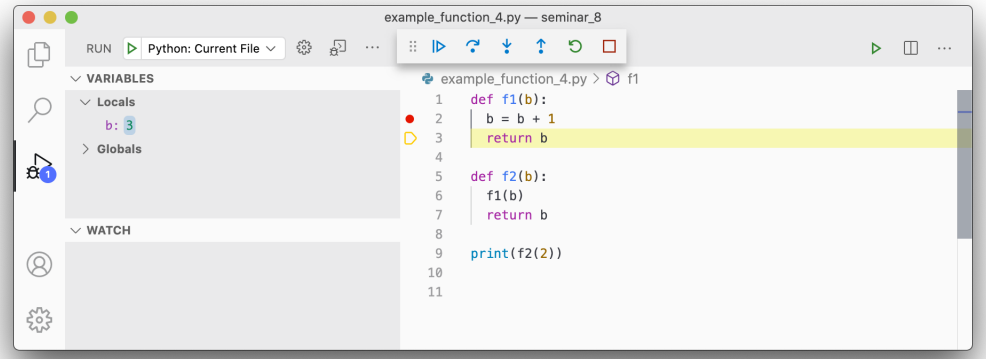
Dále dojde k volání funkce `f1`:

```
f1(b)
```

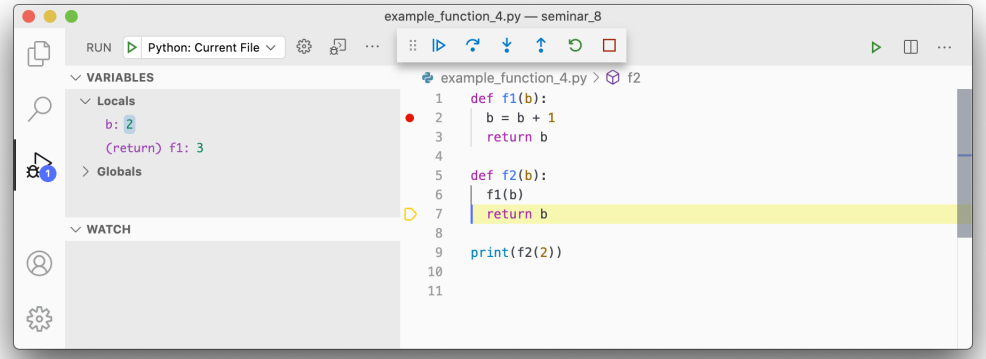
To vytvoří druhé lokální prostředí také s hodnotou 2.



Další příkaz změní hodnotu proměnné `b` v aktuálním druhém lokálním prostředí:



Následuje návrat z volání funkce `f1` do volání funkce `f2` a zde přestává platit druhé lokální prostředí. První lokální prostředí je nyní aktuální a v něm je hodnota `b` stále 2.



Dvojku tedy volání funkce `f2` vrátí a proto program vytiskl číslo dva.

Druhou možností, jak se na volání uživatelských funkcí dívat, je si zavést pojem *rozsah proměnné*. Rozsah proměnné je část kódu, kde se můžeme dotázat na její hodnotu. Říkáme, že proměnná je *platná* ve svém rozsahu. O prvním přiřazení hodnoty do proměnné v těle funkce říkáme, že tuto proměnnou *definuje*. To platí pouze, pokud proměnná není parametrem. Vezměme si například následující program.

```
1  def f1(a):
2      b = 2
3      a = a + 1
4      b = a + b
5      print(a + b)
6
```

```
7  f1(3)
```

Zde řádek dva definuje proměnnou `b`. Řádek tři proměnnou `a` nedefinuje, protože `a` je parametrem funkce. Ani řádek čtyři nedefinuje proměnnou - ta je již definovaná.

Parametry funkce a proměnné definované v jejím těle jsou nazývány *lokální proměnné* funkce. Tedy předchozí funkce má lokální proměnné `a`, `b`.

Rozsahem lokálních proměnných funkce je celé tělo funkce. Například v programu

```
1  def f1(a):
2    b = 2
3    print(a + b)
4
5  f1(3)
```

jsou rozsahem parametru `a` i proměnné `b` řádky dva a tři. Proto při vyhodnocení výrazu `print(a + b)` bude hodnota proměnné `a` tři (argument funkce) a hodnota proměnné `b` dva. Spuštění programu

```
def f1():
  print(a)
  a = 2

f1()
```

skončí chybou

```
UnboundLocalError: local variable 'a' referenced before assignment.
```

Rozsahem proměnné `a` je sice celé tělo funkce `f1`, ale na řádku dva ještě proměnná nemá hodnotu. Všimněte si rozdílu s chybou v programu:

```
def f1():
  print(a)

f1()
```

Zde program končí námi známou chybou

```
NameError: name 'a' is not defined.
```

Vrátíme se k příkladu s funkcemi `f1` a `f2`:

```
1  def f1(b):
2    b = b + 1
3    return b
4
5  def f2(b):
6    f1(b)
7    return b
8
9  print(f2(2))
```

Můžeme jej nyní vysvětlit s pomocí rozsahů proměnných. Parametr `b` funkce `f1` splývá s proměnnou `b` definovanou na řádku dva. Můžeme tedy říci, že řádek dva pouze mění proměnnou `b`. Připomeňme, že parametr funkce je proměnná. Rozsah proměnné `b` jsou řádky $2, 3$.

Rozsah parametru `b` funkce `f2` jsou řádky $6, 7$. Průnik rozsahů parametru `b` funkce `f1` a rozsahu parametru `b` funkce `f2` je prázdný. Můžeme si tedy představit, že se jedná o dvě různé proměnné stejného jména. Přejmenování proměnné jen v jedné funkci nijak neovlivní funkci druhou.

Změna proměnné `b` na řádku dva neovlivní hodnotu proměnné `b` v těle funkce `f2`. Získání hodnoty proměnné `b` na řádku sedm je mimo rozsah proměnné `b` v těle funkce `f1`.

Nyní by již mělo být jasné, proč následující program skončí chybou:

```
NameError: name 'b' is not defined.
```

```
1  def f1():
2    b = 2
3
4  def f2():
5    f1()
6    print(b)
7
8  print(f2())
```

Proměnná `b` definovaná na druhém řádku má rozsah pouze řádek dva. Naproti tomu proměnná `b` na řádku šest není ani parametrem ani není v těle funkce `f2` definována.

Vysvětlete, proč i následující program končí chybou:

```
NameError: name 'a' is not defined.
```

```
1  def f1():
2    print(a)
3
4  def f2():
5    a = 2
6    f1()
7
8  f2()
```

Víme, že po spuštění programu existuje prostředí, které určuje hodnoty proměnných. Tomuto prostředí říkáme *globální prostředí*. Pokud se hodnota proměnné nenajde v lokálním prostředí, hledá se její hodnota v globálním prostředí.

Proměnnou, jejíž hodnotu měníme mimo definice funkcí, nazýváme *globální*. O prvním přiřazení hodnoty do globální proměnné říkáme, že proměnnou *definuje*. Rozsahem globální proměnné je část kódu nalézající se za její definicí.

Pokud by v nějaké části programu byla platná jak lokální tak globální proměnná stejného jména, má přednost lokální proměnná.

Podívejme se na následující program.

```
1 c1 = 1
2
3 def f1():
4   c1 = 0
5   print(c1)
6
7 print(c1)
8 f1()
9 print(c1)
```

Máme zde globální proměnnou `c1` jejíž rozsah jsou řádky dva a dále. Dále se zde nachází lokální proměnná `c1`, která má rozsah řádky čtyři a pět. Změna proměnné `c1` na řádce čtyři tedy změní jen lokální proměnnou. Hodnota globální proměnné zůstane stejná. Proto program vytiskne:

```
1
0
1
```

Z globálního prostředí si v tělech funkcí dovolíme používat pouze konstanty.

8.2 Seznamy různých hodnot

Dosud jsme pracovali pouze se seznamy jejichž prvky byly celá čísla. Nic nám nebrání udělat seznam z libovolných hodnot. Můžeme tedy vytvořit seznam řetězců:

```
['jedna', 'dvě', 'tři']
```

pravdivostních hodnot:

```
[True, False, True]
```

nebo desetinných čísel:

```
[0.1, 12.4, 1e-20]
```

Dokonce můžeme i vytvořit seznam, kde každý prvek je jiného typu:

```
['Hráč jedna', 45, True]
```

Jelikož seznam je také hodnota, může být i on prvkem jiného seznamu.

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

Všimněte si dvojnásobného použití indexačního operátory v posledním příkladě. Seznam `l` by mohl reprezentovat následující matici čísel.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Napíšeme program tisknoucí matici (bez obklopujících závorek).

```
def print_cell(element, is_last):
    print(element, end='')
    if not is_last:
      print('␣', end='')

def print_row(row):
  for j in range(len(row)):
    element = row[j]
    is_last = j == len(row) - 1
    print_cell(element, is_last)
  print()

def print_matrix(matrix):
  for i in range(len(matrix)):
    row = matrix[i]
    print_row(row)
```

Zkouška:

```
>>> print_matrix([[1, 2, 3], [4, 5, 6]])
1 2 3
4 5 6
```

8.3 Úkoly

Úkol 8.1. Pro řetězec obsahující slova oddělená mezerou vraťte seznam těchto slov.

Úkol 8.2. Rozhodněte, zda je jeden seznam podseznamem druhého.

Úkol 8.3. Transponujte zadanou matici. Transponovaná matice vznikne záměnou řádků a sloupců. Například transpozicí matice

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

vznikne matice

$$\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}.$$

Úkol 8.4. Vytvořte jednotkovou matici zadané velikosti. Jednotková matice má na hlavní diagonále jedničky a jinde nuly. Například jednotková matice velikosti tři vypadá takto:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Úkol 8.5. Je dán seznam čísel l a přirozené číslo n. Vytvořte seznam délky n, kde prvek na indexu i je seznam obsahující všechna čísla z l jejichž zbytek po dělení číslem n je i. Například pro seznam `[1, 2, 3, 3, 8, 5, 4]` číslo `3` vraťte `[[3, 3], [1, 4], [2, 8, 5]]`.

Úkol 8.6. Vytvořte počáteční stav šachovnice u hry česká dáma. Šachovnici osm krát osm reprezentujte jako seznam rádků, kde řádek je seznam polí. Nulou označte prázdné pole, jedničkou bílý kámen a dvojkou černý kámen.

Úkol 8.7. Napište funkci, která vytiskne šachovnici. Prázdná pole tiskněte znakem tečka, bílý kámen písmenem malé o, černý kámen hvězdičkou. Za znakem pole nechávejte mezeru. Doplňte z obou stran čísla řádků a písmen sloupců. Tedy počáteční pozice české dámy se vytiskne takto:

```
  a b c d e f g h
8 . o . o . o . o 8
7 o . o . o . o . 7
6 . o . o . o . o 6
5 . . . . . . . . 5
4 . . . . . . . . 4
3 * . * . * . * . 3
2 . * . * . * . * 2
1 * . * . * . * . 1
  a b c d e f g h
```

Úkol 8.8. Napište funkci, která změní zadané pole šachovnice na danou hodnotu.

Úkol 8.9. Umíte předchozí funkce pracující s šachovnicí upravit tak, aby šlo konstantou nastavit velikost šachovnice? Například devět krát devět. Dále můžete konstantou určit počet řad obsazených na začátku kameny hráče. Pro českou dámu to jsou tři řady.

Úkol 8.10. Rozdělte řešení úkolů 3.5 (nesoudělná čísla) a 3.6 (Pythagorejské trojice) do funkcí.

Úkol 8.11. Rozdělte vhodně řešení úkolu 7.13 (Eratosthenovo síto) do funkcí.

9 Devátý seminář

Opakování: zlomky

```
def make_fract(num, den):
  if den == 0:
    return num // den
  else:
    return [num, den]

#print(make_fract(1, 2))

def get_num(fract):
  return fract[0]

# print(get_num(make_fract(1, 2)))

def get_den(fract):
  return fract[1]

# print(get_den(make_fract(1, 2)))

def print_fract(fract):
  print('make_fract(', end='')
  print(get_num(fract), end=',␣')
  print(get_den(fract), end='')
  print(')')

# print_fract(make_fract(1, 2))

def are_fracts_equal(fract1, fract2):
  num1 = get_num(fract1)
  den1 = get_den(fract1)
  num2 = get_num(fract2)
  den2 = get_den(fract2)
  return  num1 * den2 == num2 * den1

# print(are_fracts_equal(make_fract(1, 2), make_fract(1, 2)))
# print(are_fracts_equal(make_fract(1, 2), make_fract(2, 4)))
# print(are_fracts_equal(make_fract(1, 2), make_fract(1, 4)))

def add_fracts(fract1, fract2):
  num1 = get_num(fract1)
  den1 = get_den(fract1)
  num2 = get_num(fract2)
  den2 = get_den(fract2)
```

```
  num_result = num1 * den2 + num2 * den1
  den_result = den1 * den2
  return make_fract(num_result, den_result)

#print_fract(add_fracts(make_fract(1, 2), make_fract(1, 2)))
#print_fract(add_fracts(make_fract(1, 3), make_fract(1, 2)))

def mult_fracts(fract1, fract2):
  num1 = get_num(fract1)
  den1 = get_den(fract1)
  num2 = get_num(fract2)
  den2 = get_den(fract2)
  num_result = num1 * num2
  den_result = den1 * den2
  return make_fract(num_result, den_result)

# print_fract(mult_fracts(make_fract(1, 2), make_fract(2, 1)))

def compute_gcd(n, m):
  while m != 0:
    tmp = m
    m = n % m
    n = tmp
  return n

# print(compute_gcd(6, 8))
# print(compute_gcd(10, 15))
# print(compute_gcd(8, 9))

def reduce_fract(fract):
  num = get_num(fract)
  den = get_den(fract)
  gcd = compute_gcd(num, den)
  return make_fract(num // gcd, den // gcd)

# print_fract(reduce_fract(make_fract(5, 10)))
def print_and_add_fracts(fract1, fract2):
  print_fract(add_fracts(fract1, fract2))

# f1 = make_fract(1, 2)
# f2 = make_fract(2, 1)
# mult_f = mult_fracts(f1, f2)
# red_f = reduce_fract(mult_f)
# print_fract(red_f)

def add_inverse_fract(fract):
```

```
  num = get_num(fract)
  den = get_den(fract)
  return make_fract(-num, den)

# f1 = make_fract(1, 2)
# print_fract(add_fracts(f1 , add_inverse_fract(f1)))

def sub_fracts(fract1, fract2):
  return add_fracts(fract1, add_inverse_fract(fract2))

# print_fract(sub_fracts(make_fract(1, 1), make_fract(1, 3)))

def mult_inverse_fract(fract):
  num = get_num(fract)
  den = get_den(fract)
  return make_fract(den, num)

# f1 = make_fract(1, 2)
# print_fract(mult_fracts(f1 , mult_inverse_fract(f1)))

def div_fracts(fract1, fract2):
  return mult_fracts(fract1, mult_inverse_fract(fract2))

# print_fract(div_fracts(make_fract(1, 1), make_fract(1, 3)))
```

10 Desátý seminář

10.1 Rekurze

Již víme, že z těla uživatelské funkce můžeme volat jinou uživatelskou funkci. To se například děje v následujícím programu počítajícím součet čtverců dvou čísel:

```
1 def square(x):
2   return x * x
3
4 def sum_of_squares(x, y):
5   return square(x) + square(y)
```

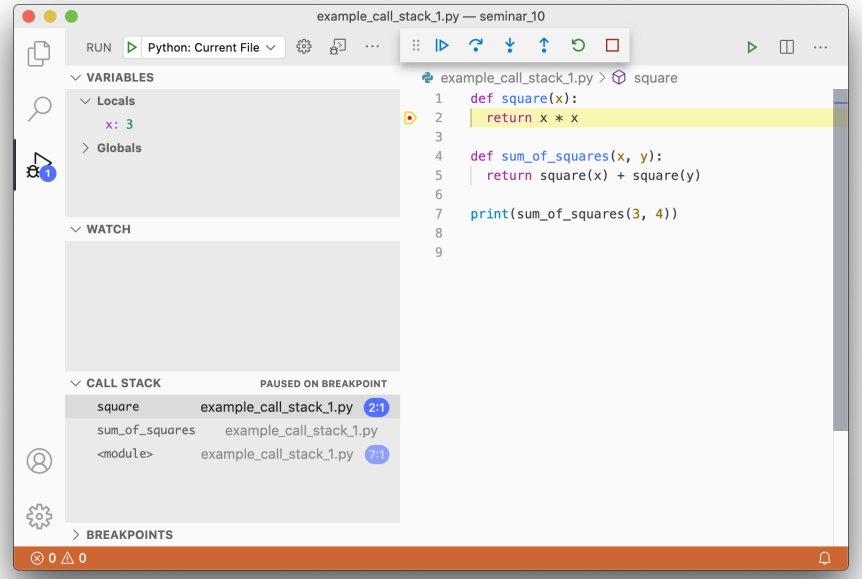
Funkce `sum_of_squares` pro čísla x a y spočítá

$$x^2 + y^2.$$

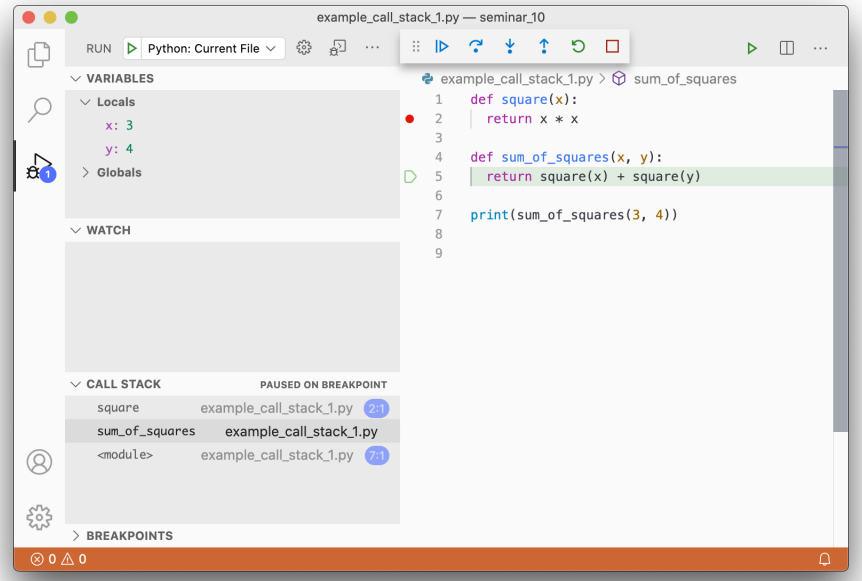
Skutečně, pro x rovno 3 a y rovno 4 dostáváme

```
>>> sum_of_squares(3, 4)
25
```

Pokud si na druhý řádek dáme zarážku a spustíme program pro ladění, můžeme se v levé dolní části pojmenované `CALL STACK` podívat na to, jaká těla funkcí se vykonávají:



První funkce `square` je aktuálně vykonávaná funkce. Pod ní se nalézá funkce `sum_of_squares`, která funkci `square` volala. Kliknutím na `sum_of_squares` se zvýrazní řádek, kde k volání došlo:



Tedy funkce `sum_of_squares` volá funkci `square`.

Abychom lépe viděli, co se při vykonávání odehrává, doplníme si do funkcí tisk hlášek. Jednu hlášku vytiskneme před provedením výpočtu a druhou těsně před vrácením hodnoty.

```
def square(x):
  print('Calling square with', x)
  result = x * x
  print('Result of calling square with', x, 'is', result)
  return result

def sum_of_squares(x, y):
  print('Calling sum_of_squares with', x, 'and', y)
  result =  square(x) + square(y)
  print('Result of calling sum_of_squares with', x, 'and', y,
'is', result)
  return result
```

Hlášky po zavolání funkce vypadají následovně.

```
>>> sum_of_squares(3, 4)
Calling sum_of_squares with 3 and 4
Calling square with 3
Result of calling square with 3 is 9
Calling square with 4
```

```
Result of calling square with 4 is 16
Result of calling sum_of_squares with 3 and 4 is 25
25
```

Vidíme, jak se dvakrát při volání funkce `sum_of_squares` volala funkce `square`. Volání uživatelské funkce při vykonávání uživatelské funkce se nazývá *zanořené*. Můžeme také mluvit o *úrovni zanoření*. Volání uživatelské funkce z globálního prostředí má úroveň zanoření jedna. Volání uživatelské funkce v rámci vykonávání těla uživatelské funkce má o jedna větší úroveň než její volání. Například volání `square` v rámci `sum_of_squares` při vyhodnocení výrazu `sum_of_squares(3, 4)` má úroveň zanoření dva. Jistě si dokážete představit program, jehož vykonávání by probíhalo v úrovni zanoření tři.

Zdůrazněme, že úroveň zanoření je vlastnost stavu interpretu při vykonávání programu. Při vykonávání programu se tedy úroveň zanoření mění.

Podívejme se na další příklad. Předpokládejme na chvíli, že chceme sečíst dvě nezáporná celá čísla, ale umíme pouze přičíst jedničku nebo odečíst jedničku od kladného čísla. Tedy chceme napsat funkci `add`, která bude brát dvě čísla jako argumenty a vracet jejich součet. Například

```
>>> add(2, 3)
5
>>> add(1, 5)
6
>>> add(6, 0)
6
```

Všimněte si, že pokud je druhý argument nula, může funkce rovnou vrátit první číslo. V opačném případě můžeme k prvnímu číslu přičíst jedničku, od druhého jedničku odečíst (tím nezměníme hodnotu součtu) a celý proces opakovat. Vyjádřeno prostředky, které máme k dispozici, dostáváme:

```
def add(n, m):
  while m != 0:
    n += 1
    m -= 1
  return n
```

Sečtení dvou čísel, kde druhé je veliké, může chvíli trvat. Zkuste například vyhodnotit `add(1, 10**7)`. Pokud by druhý argument byl záporný, výpočet neskončí nikdy. Tedy vyhodnocování výrazu `add(1, -1)` bude probíhat do nekonečna.

Když bychom ale slepě přepsali výše popsaný proces sčítání, tak fráze „a celý proces opakovat“ znamená zavolání funkce `add`, a obdržíme:

```
def add(n, m):
  if m == 0:
    return n
  else:
    return add(n + 1, m - 1)
```

Zde je podezřelé, že funkci `add` voláme v její definici. Z pohledu vykonávání to však problém není, protože příkaz definice funkce `add` pouze vytvoří uživatelskou funkci `add`. V momentě zavolání funkce `add` například příkazem `add(2, 2)` již je funkce `add` k dispozici a může tedy být zavolána z těla funkce jako každá jiná funkce.

Aby jsme si udělali lepší představu o tom, jak funkce `add` funguje, přidáme si do jejího těla tisk argumentů a návratové hodnoty:

```
1  def add(n, m):
2    print('Calling␣add␣with', n, 'and', m)
3    if m == 0:
4      result = n
5    else:
6      result = add(n + 1, m - 1)
7    print('Result␣of␣calling␣add␣with', n, 'and', m, 'is', result)
8    return result
```

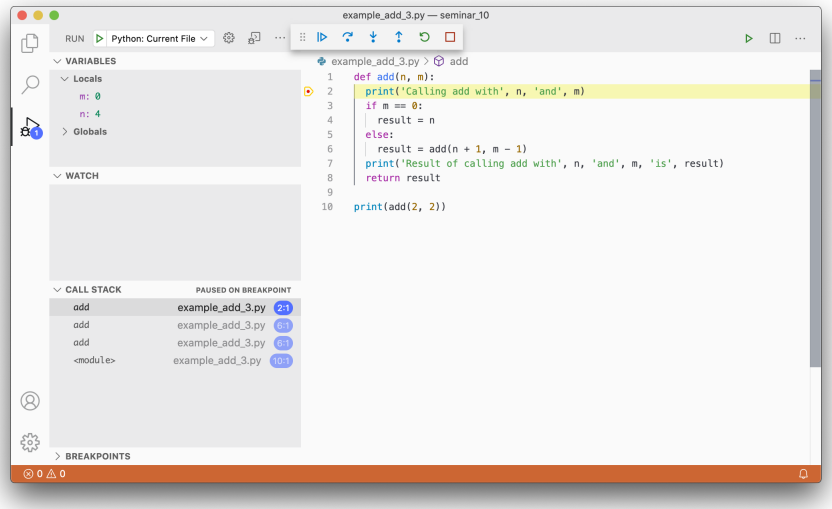
Nyní vyhodnocení výrazu `add(2, 2)` vytiskne

```
Calling add with 2 and 2
Calling add with 3 and 1
Calling add with 4 and 0
Result of calling add with 4 and 0 is 4
Result of calling add with 3 and 1 is 4
Result of calling add with 2 and 2 is 4
```

Vidíme, že první dvě volání prošly přes **`else`** větev a tím se spustilo zavolání funkce `add` na 6. řádku. V třetím volání bylo `m` rovno 0 a vykonala se tedy první větev, která rozhodla, že výsledek bude čtyři. V tuto chvíli probíhá vykonávání třech volání funkce `add`. Tedy funkce `add` s argumenty 2 a 2 volá funkci `add` s argumenty 3 a 1 a ta volá funkci `add` s argumenty 4 a 0. Na osmém řádku se vrátí výsledek a skončí poslední (třetí) volání. První a druhé volání pouze vytisknou hlášku a předají vrácený výsledek.

Říkáme, že funkce je *rekurzivní*, pokud ve svém těle volá samu sebe. Funkce `add` je rekurzivní. O volání funkce v těle funkce řekneme, že se jedná o *rekurzivní volání*, pokud se jedná o volání téže funkce v jejímž těle se volání nachází. Na 6. řádku se nalézá rekurzivní volání. (Voláme funkci `add` v těle funkce `add`).

Další možnost, jak získat představu o rekurzivních volání je projít si jednotlivá rekurzivní volání v režimu pro ladění. V části `CALL STACK` vidíme zanoření rekurzivních volání.



S rekurzí se pojí nový druh chyb:

```
>>> add(1, 1000)
RecursionError: maximum recursion depth exceeded in comparison
```

Logicky je sice program v pořádku, ale interpret Pythonu umožňuje pouze úroveň zanoření tisíc. Ve skutečnosti je tato hodnota o pár jednotek nižší, protože sám interpret pro svoji činnost nějakou úroveň potřebuje. Všimněte si, že volání funkce s druhým záporným argumentem nyní nespadne do nekonečné smyčky, ale skončí chybou:

```
>>> add(1, -1)
RecursionError: maximum recursion depth exceeded in comparison
```

Interpret nepozná, zda se jedná o nekonečnou smyčku a nebo o příliš náročný výpočet.

Některé programy se pomocí rekurze vytvářejí velice snadno. Vezměme si příklad faktoriálu.

$$n! = \begin{cases} 1, & \text{pro } n = 0; \\ n \cdot (n-1)!, & \text{jinak.} \end{cases}$$

Tradiční definice by se neobešla bez cyklu:

```
def factorial(n):
  result = 1
  for i in range(n):
    result *= i + 1
  return result
```

Zkouška:

```
>>> factorial(5)
120
```

S použitím rekurze lze definici faktoriálu přepsat do programu přímočaře:

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n - 1)
```

Výpočet hodnoty faktoriálu podle matematické definice se nyní podobá výpočtu prováděném programem. Ukážeme si to na příkladu faktoriálu tří. Podle definice faktoriálu nejdříve dostáváme

$$3! = 3\cdot(3-1)! = 3\cdot 2! = 3\cdot 2\cdot(2-1)! = 3\cdot 2\cdot 1! = 3\cdot 2\cdot 1\cdot(1-1)! = 3\cdot 2\cdot 1\cdot 0! = 3\cdot 2\cdot 1\cdot 1,$$

dále máme

$$3 \cdot 2 \cdot 1 \cdot 1 = 3 \cdot 2 \cdot 1 = 3 \cdot 2 = 6.$$

Pro zkoumání programu přidáme tisk hlášek:

```
def factorial(n):
  print('Calling␣factorial␣with', n)
  if n == 0:
    result = 1
  else:
    result = n * factorial(n - 1)
  print('Result␣of␣calling␣factorial␣with', n,  'is', result)
  return result
```

Výpočet faktoriálu vytiskne:

```
>>> factorial(3)
Calling factorial with 3
Calling factorial with 2
Calling factorial with 1
Calling factorial with 0
Result of calling factorial with 0 is 1
Result of calling factorial with 1 is 1
Result of calling factorial with 2 is 2
Result of calling factorial with 3 is 6
```

První část odpovídá výpočtu od 3! až po $3\cdot 2\cdot 1\cdot 1$. Druhá část počítá násobení $3 \cdot 2 \cdot 1 \cdot 1$. Návratová hodnota volání funkce se zde na rozdíl od předchozího příkladu mění. To je způsobené tím, že se před vrácením hodnoty provede výpočet používající návratovou hodnotu rekurzivního volání. Konkrétněni na řádce

```
return n * factorial(n - 1)
```

je návratová hodnota výsledek vynásobení n a návratové hodnoty rekurzivního volání. Pokud funkce přímo vrací hodnotu rekurzivního volání, říkáme, že se jedná o *koncovou rekurzi*. Výše uvedená rekurzivní funkce `add` používala koncovou rekurzi. Dokážete ji přepsat tak, aby se o koncově rekurzivní funkci nejednalo? Oproti ní rekurzivní verze faktoriálu koncově rekurzivní není. Můžeme jí ale přepsat tak, aby koncově rekurzivní byla:

```
def factorial_iter(n, result):
  if n == 0:
    return result
  else:
    return factorial_iter(n - 1, result * n)

def factorial(n):
  return factorial_iter(n, 1)
```

Museli jsme ale zavést pomocnou funkci `factorial_iter` a pomocný argument `result`, kde se postupně skládá výsledek. Vlastně pomocí rekurze simulujeme první verzi napsanou pomocí cyklů. Pokud je funkce napsaná jen pomocí cyklů bez rekurze, říkáme, že je *iterativní*. Tedy simulujeme iterativní verzi pomocí rekurze. Proto má pomocná funkce název `factorial_iter`.

Další kanonický příklad na rekurzi je výpočet Fibonacciho posloupnosti. Uveďme nejdříve řešení pomocí cyklů:

```
def fibonacci(n):
  a = 0
  b = 1
  for i in range(n):
    c = a + b
    a = b
    b = c
  return a
```

Funkce počítá $(n+1)$-tý prvek Fibonacciho posloupnosti.

```
>>> print(fibonacci(10))
55
```

Zkusme se opřít o matematickou definici:

$$F(n) = \begin{cases} 0, & \text{pro } n = 0; \\ 1, & \text{pro } n = 1; \\ F(n-1) \cdot F(n-2)!, & \text{jinak.} \end{cases}$$

Vyjádřeno programem dostáváme

```
def fibonacci(n):
  if n == 0:
    return 0
  elif n == 1:
```

```
    return 1
  else:
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Program funguje korektně. Problém je, že již pro docela malé vstupy trvá výpočet překvapivě dlouho. Zkuste si vyhodnotit `fibonacci(45)`.

Program můžeme sice přepsat tak, aby byl efektivnější, ale tím ztratíme eleganci předchozího řešení a vrátíme se v podstatě k původní verzi:

```
def fibonacci_iter(n, a, b):
  if n == 0:
    return a
  else:
    return fibonacci_iter(n - 1, b, a + b)

def fibonacci(n):
  return fibonacci_iter(n, 0, 1)
```

10.2 Úkoly

Úkol 10.1. Pomocí rekurze vynásobte dvě nezáporná celá čísla. Nemůžete použít operátor násobení. Opřete se o následující definici.

$$n \cdot m = \begin{cases} 0, & \text{pro } m = 0; \\ n + (n \cdot (m-1)), & \text{jinak} \end{cases}$$

Úkol 10.2. Napište předchozí funkci iterativně.

Úkol 10.3. Upravte předchozí funkci tak, aby používala koncovou rekurzi.

Úkol 10.4. Pomocí rekurze spočítejte m-tou mocninu čísla n. Čísla n a m jsou celá nezáporná. Nemůžete použít operátor mocnění (`**`).

Úkol 10.5. Co předchozí funkce vrátí jako výsledek 0^0. Je to správně?

Úkol 10.6. Zkuste ve výpočtu mocniny použít funkci násobení z prvního úkolu. Porovnejte možnosti velikosti druhého argumentu oproti předchozí verzi.

Úkol 10.7. Přepište výpočet mocniny tak, aby byl koncově rekurzivní.

Úkol 10.8. Napište iterativní výpočet mocniny.

Úkol 10.9. Umíte spočítat mocninu pouze za použití přičtení a odečtení jedničky? Tedy bez obecného sčítání, odčítání, násobení a samozřejmě mocniny.

11 Jedenáctý seminář

11.1 Podseznamy

Vezmeme-li seznam, například `[1, 2, 3]`, můžeme *indexovat mezery* mezi prvky. Mezera před prvním prvkem bude mít index nula, mezera mezi prvním a druhým prvkem bude mít index jedna, a tak dále. Délka seznamu bude indexem mezery za posledním prvkem.

Podseznam l_2 seznamu l_1 můžeme určit dvojicí indexů mezer i a j seznamu l_1. Podseznam l_2 budou tvořit právě ty prvky mezi mezerami i a j.

Jistě byste uměli napsat funkci `sublist`, která by brala seznam a dva indexy mezer a vracela by jimi určený podseznam:

```
>>> sublist([1, 2, 3], 1, 3)
[2, 3]
>>> sublist([1, 2, 3], 0, 2)
[1, 2]
>>> sublist([1, 2, 3], 0, 0)
[]
```

Definice funkce by mohla vypadat následovně.

```
def sublist(input_list, index_from, index_to):
  result = []
  for i in range(index_to - index_from):
    element = input_list[i + index_from]
    result = result + [element]
  return result
```

Podseznam seznamu lze získat jednodušeji pomocí takzvaných řezů. Pokud v, i a j jsou výrazy, pak

$$v\texttt{[}i\texttt{:}j\texttt{]}$$

je *výraz řezu* (anglicky *slice*). Hodnota výrazu v je seznam. Hodnoty výrazů i a j jsou indexy mezer seznamu v. Hodnotou výrazu řezu je podseznam seznamu v určený indexy i a j. Výše uvedená volání funkce `sublist` lze přepsat pomocí řezů:

```
>>> l = [1, 2, 3]
>>> l[1:3]
[2, 3]
>>> l[0:2]
[1, 2]
>>> l[0:0]
[]
```

Samozřejmě v ve výrazu řezu může být přímo seznam:

```
>>> [1, 2, 3][1:3]
[2, 3]
```

Pokud první index i ve výrazu řezu vynecháme, uvažuje se index nula. Pokud vynecháne druhý index j uvažuje se délka seznamu v. Můžeme vynechat i oba indexy. Například:

```
>>> l = [1, 2, 3]
>>> l[:2]
[1, 2]
>>> l[1:]
[2, 3]
>>> l[:]
[1, 2, 3]
```

11.2 Proměnlivost seznamů

Představme si, že chceme v seznamu nahradit prvek na indexu jiným prvkem. Můžeme to provést následující funkcí:

```
def replace_element(input_list, index, element):
  result_list = []
  for i in range(len(input_list)):
    if i == index:
      result_element = element
    else:
      result_element = input_list[i]
    result_list = result_list + [result_element]
  return result_list
```

S pomocí řezů můžeme funkci napsat jednodušeji:

```
def replace_element(input_list, index, element):
  return input_list[:index] + [element] + input_list[index + 1:]
```

Vezmeme si seznam

```
>>> l1 = [1, 2, 3]
```

zavedeme si jinou proměnnou s hodnotou `l1`:

```
>>> l2 = l1
```

pokud provedeme nahrazení prvku:

```
>>> l1 = replace_element(l1, 1, 5)
```

Vidíme, že se změnila jen hodnota `l1`:

```
>>> l1
[1, 5, 3]
>>> l2
[1, 2, 3]
```

Změnit prvek seznamu můžeme i následujícím příkazem. Pokud l, i a v jsou výrazy, pak

l`[`i`] = `v

je *příkaz změny prvku seznamu*. Hodnotou l je seznam, hodnotou i je index prvku seznamu l, hodnota v je libovolná. Příkaz *změní* prvek seznamu l na indexu i na hodnotu v. Příklad volání:

```
 >>> l = [1, 2, 3]
 >>> l[1] = 5
 >>> l
[1, 5, 3]
```

Oproti předchozímu případu dojde opravdu ke změně prvku seznamu. Předchozí případ vytvořil nový seznam, který měl jen jeden prvek jiný. Skutečně:

```
>>> l1 = [1, 2, 3]
>>> l2 = l1
>>> l1[1] = 5
>>> l1
[1, 5, 3]
>>> l2
[1, 5, 3]
```

Narozdíl od předchozího případu, došlo i ke změně seznamu `l2`. Ve skutečnosti se změnil jen jeden seznam, ale hodnota proměnné `l1` je totožná s hodnotou proměnné `l2`. Jak porovnávat totožnost?

Pokud v_1 a v_2 jsou výrazy, pak

v_1 `is` v_2

je výraz *porovnání totožnosti*. Totožnost nás zajímá pouze u seznamů - ty jediné umíme měnit, proto budeme předpokládat, že hodnoty v_1 a v_2 jsou seznamy. Hodnota výrazu porovnání totožnosti je pravda, pokud seznamy v_1 a v_2 jsou totožné, jinak je hodnota nepravda.

Podívejme se na příklad:

```
>>> l1 = [1, 2, 3]
>>> l2 = l1
>>> l3 = [1, 2, 3]
>>> l1 is l2
True
>>> l1 is l3
False
```

Vidíme, že seznamy `l1` a `l3` nejsou totožné a to i přesto, že jsou si všechny seznamy rovny:

```
>>> l1 == l2
True
>>> l2 == l3
True
```

Ve skutečnosti `l1` a `l2` jsou různé názvy pro tu samou hodnotu, `l3` je jiná hodnota, která je pouze rovna (ekvivalentní) seznamům `l1` a `l2`. Proto změna `l1` změní i hodnotu `l2` ale nezmění hodnotu `l3`:

```
>>> l1[1] = 5
>>> l1
[1, 5, 3]
>>> l2
[1, 5, 3]
>>> l3
[1, 2, 3]
```

Pokud bychom chtěli hodnotu `l1` změnit a neovlivnit hodnotu `l2`, musíme udělat kopii hodnoty `l1`. Toho lze docílit například řezem:

```
>>> l1 = [1, 2, 3]
>>> l2 = l1[:]
>>> l1[1] = 5
>>> l1
[1, 5, 3]
>>> l2
[1, 2, 3]
```

Problém může nastat, pokud hodnoty seznamu jsou zase seznamy:

```
l1 = [[1, 2], [3, 4]]
l2 = l1[:]
l1[0][0] = 5
>>> l1
[[5, 2], [3, 4]]
>>> l2
[[5, 2], [3, 4]]
```

Vidíme, že kopie seznamu není dostačující. Ke kopii prvků seznamu nedošlo:

```
>>> l1 is l2
False
>>> l1[0] is l2[0]
True
```

Problém vyřešíme funkcí kopírující i prvky seznamu:

```
def copy_matrix(matrix):
  new_matrix = []
  for i in range(len(matrix)):
    row = matrix[i]
    new_row = row[:]
    new_matrix = new_matrix + [new_row]
  return new_matrix
```

Test potvrdí dostatečnost kopírování:

```
>>> l1 = [[1, 2], [3, 4]]
>>> l2 = copy_matrix(l1)
>>> l1 == l2
True
>>> l1 is l2
False
>>> l1[0] is l2[0]
False
>>> l1[0][0] = 5
>>> l1
[[5, 2], [3, 4]]
>>> l2
[[1, 2], [3, 4]]
```

Možnost změny seznamu umožňuje funkcím měnit své argumenty. Vezměme si například funkci

```
def set_element(input_list, index, element):
  input_list[index] = element
```

Potom máme:

```
>>> l = [1, 2, 3]
>>> l
[1, 2, 3]
>>> set_element(l, 1, 5)
>>> l
[1, 5, 3]
```

Zavolání funkce způsobilo změnu seznamu `l`.

Podívejme se na kód, který přidá prvek nakonec seznamu:

```
>>> l1 = [1, 2, 3]
>>> l2 = l1
>>> l1 = l1 + [4]
>>> l1
[1, 2, 3, 4]
>>> l2
[1, 2, 3]
```

Seznam `l2` zůstal bez změny. Pro přidání prvku si můžeme napsat funkci:

```
def append_element(input_list, element):
  return input_list + [element]
```

Zkouška:

```
>>> l1 = [1, 2, 3]
>>> l2 = l1
>>> l1 = append_element(l1, 4)
>>> l1
```

```
[1, 2, 3, 4]
>>> l2
[1, 2, 3]
```

Zavedeme si nový příkaz měnící seznam. Pokud l je proměnná, jejíž hodnota je seznam, a v je výraz, jehož hodnota je seznam, pak

l += v

je *příkaz přidání prvků nakonec seznamu*. Příkaz přidá prvky seznamu v nakonec seznamu l. Dojde tedy ke změně seznamu l. Test:

```
>>> l1 = [1, 2, 3]
>>> l2 = l1
>>> l1 += [4]
>>> l1
[1, 2, 3, 4]
>>> l2
[1, 2, 3, 4]
```

Tato verze funkce přidání prvku nakonec seznamu tedy seznam mění:

```
def append_element(input_list, element):
  input_list += [element]
```

Což vidíme zde:

```
>>> l1 = [1, 2, 3]
>>> l2 = l1
>>> append_element(l1, 4)
>>> l1
[1, 2, 3, 4]
>>> l2
[1, 2, 3, 4]
```

11.3 Úkoly

Úkol 11.1. Napište iterativně funkci `replace`, která bere seznam l a dvě hodnoty v_1 a v_2. Funkce nahradí v seznamu l každý výskyt hodnoty v_1 za hodnotu v_2. Funkce nesmí změnit seznam l. Například:

```
>>> replace([1, 2, 1, 3], 1, 5)
[5, 2, 5, 3]
```

Úkol 11.2. Přepište předchozí funkci tak, aby nic nevracela a přímo měnila seznam l. Například:

```
>>> l = [1, 2, 1, 3, 1]
>>> replace(l, 1, 5)
>>> l
[5, 2, 5, 3, 5]
```

Úkol 11.3. Vyřešte první dva úkoly za použití rekurze.

Úkol 11.4. Jsou vaše řešení koncově rekurzivní?

Úkol 11.5. Dokážete dvě předešlé rekurzivní verze upravit tak, aby každé rekurzivní volání zmenšilo délku uvažovaného seznamu na polovinu?

Úkol 11.6. Napište funkci `reverse`, která prohodí prvky zadaného seznamu. Funkce nic nevrací a přímo mění seznam zadaný jako argument.

Úkol 11.7. Přepište předchozí verzi tak, aby vracela seznam s prohozenými prvky a neměnila svůj argument.

Úkol 11.8. Umíte napsat verze `reverse` rekurzivně?

12 Dvanáctý seminář

12.1 Sekvence

Začneme funkcí, která rozhodne, zda je hodnota prvkem seznamu.

```
def is_member(input_list, element):
  result = False
  for i in range(len(input_list)):
    input_element = input_list[i]
    if input_element == element:
      result = True
  return result
```

Dostáváme:

```
>>> is_member([1, 2, 3], 1)
True
>>> is_member([1, 2, 3], 5)
False
```

Jaké požadavky funkce klade na argument `input_list`? Musíme být schopni získat délku argumentu pomocí vestavěné funkce **`len`** a zjistit hodnotu na indexu indexačním operátorem. To ale splňuje nejen seznam, ale i řetězec. Funkce bude zázračně fungovat i na řetězce:

```
>>> is_member('abc', 'b')
True
>>> is_member('abc', 'e')
False
```

Bylo by vhodné přejmenovat některé proměnné funkce `is_member` tak, aby nepoužívaly slovo **`list`**. Zavedeme si pojem *sekvence*. Sekvence je hodnota, která má délku a je možné získat hodnotu na indexu menším, než délka sekvence. Hodnotě na indexu říkáme *položka sekvence* (anglicky *item*). Délku sekvence vrací funkce **`len`** a hodnotu na indexu indexační operátor. Seznamy i řetězce jsou sekvence:

```
>>> l = [5, 2, 7]
>>> len(l)
3
>>> l[0]
5
>>> s = 'abc'
>>> len(s)
3
>>> s[1]
'b'
```

Zvolíme vhodnější jména pro proměnné ve funkci `is_member`:

```
def is_member(sequence, item):
  result = False
  for i in range(len(sequence)):
    sequence_item = sequence[i]
    if sequence_item == item:
      result = True
      break
  return result
```

Funkce při nalezení položky v sekvenci nastaví proměnnou `result`, přeruší cyklus a vrátí hodnotu `result`. Mohli bychom stejně dobře rovnou vrátit `True`. Ve funkcích, kde dojde ke zvýšení čitelnosti, si dovolíme vrátit hodnotu i v místě, které není posledním příkazem funkce. Funkci `is_member` vyjádříme přehledněji.

```
def is_member(sequence, item):
  for i in range(len(sequence)):
    sequence_item = sequence[i]
    if sequence_item == item:
      return True
  return False
```

Následující funkce v sekvenci nahradí položku na indexu zadanou položkou.

```
def replace(sequence, index, item):
  sequence[index] = item
```

Funkce se spoléhá na to, že lze měnit položky sekvence. Víme, že některé sekvence nepřipouští změnu svých položek. Proto funkce bude fungovat pro seznamy, ale ne pro řetězce.

```
>>> s1 = [1, 2, 3]
>>> replace(s1, 1, 5)
>>> s1
[1, 5, 3]
>>> s2 = 'abc'
>>> replace(s2, 1, 'd')
TypeError: 'str' object does not support item assignment
```

Zkusíme funkci přepsat pomocí řezů:

```
def replace(sequence, index, item):
  return sequence[:index] + [item] + sequence[index + 1:]
```

Předpokládáme, že řez by měl fungovat pro každou sekvenci, protože vyžaduje jen získání položky na indexu. Opravdu, řez pro řetězce funguje:

```
>>> s = 'abcd'
>>> s[1:]
'bcd'
>>> s[2:3]
'c'
```

Dále předpokládáme, že sekvence můžeme spojovat. Funkce ale stále řetězce nepřipouští:

```
>>> s1 = [1, 2, 3]
>>> s1 = replace(s1, 1, 5)
>>> s1
[1, 5, 3]
>>> s2 = 'abc'
>>> s2 = replace(s2, 1, 'd')
TypeError: can only concatenate str (not "list") to str
```

Důvodem je, že řez na řetězci vrátí řetězec a ten nemůžeme spojit se seznamem. Můžeme spojovat pouze sekvence stejného typu. Změníme funkci tak, aby místo položky očekávala další sekvenci:

```
def replace(sequence, index, subsequence):
  return sequence[:index] + subsequence + sequence[index + 1:]
```

Nyní již funkce funguje jak pro seznamy tak pro řetězce.

```
>>> s1 = [1, 2, 3]
>>> s1 = replace(s1, 1, [5])
>>> s1
[1, 5, 3]
>>> s2 = 'abc'
>>> s2 = replace(s2, 1, 'd')
>>> s2
'adc'
```

Obecně funkce očekává sekvence, které podporují řezy a spojování.
Každou sekvenci můžeme převést na seznam následující funkcí.

```
def my_list(sequence):
  result = []
  for i in range(len(sequence)):
    result += [sequence[i]]
  return result
```

Můžeme si dovolit použít příkaz na změnu seznamu += a to z toho důvodu, že měníme seznam, který jsme si sami vytvořily (nejedná se o argument). Zkouška:

```
>>> my_list('abc')
['a', 'b', 'c']
>>> my_list([1, 2])
[1, 2]
```

Stejný efekt má i vestavěná funkce **list**:

```
>>> list('abc')
['a', 'b', 'c']
>>> list([1, 2])
[1, 2]
```

12.2 Číselné sekvence

Následující program postupně vytiskne čísla od nuly do devíti.

```
for i in range(10):
  print(i)
```

Ve skutečnosti část programu **range(10)** je volání funkce **range**. Funkce vrací *číselnou sekvenci*. Jedná se o sekvenci, kde položky jsou čísla od nuly do zadaného čísla, které už v sekvenci není.

```
>>> r1 = range(10)
>>> len(r1)
10
>>> r1[1]
1
```

Samotná číselná sekvence se tiskne za použití funkce **range**.

```
>>> range(10)
range(0, 10)
```

Nový první argument v tisku je počáteční položka v číselné sekvenci. Vyhodnocením výrazu **range(0, 10)** obdržíme sekvenci rovnou **range(10)**. Nula je výchozí počáteční prvek sekvence.

```
>>> range(10) == range(0, 10)
True
```

Volbou počáteční položky můžeme například vytvořit číselnou sekvenci od deseti do dvaceti:

```
>>> r = range(10, 20)
>>> len(r)
10
>>> r[0]
```

```
10
>>> r[5]
15
```

Můžeme také získávat řezy číselných sekvencí:

```
>>> r1 = range(100)
>>> r2 = r1[50:]
>>> r2
range(50, 100)
>>> r2[0]
50
>>> len(r2)
50
```

Číselné sekvence ale není možné spojovat a ani nelze měnit jejich položky:

```
>>> range(10) + range(10, 20)
TypeError: unsupported operand type(s) for +: 'range' and 'range'
>>> r = range(10)
>>> r[1] = 2
TypeError: 'range' object does not support item assignment
```

Samozřejmě můžeme převést číselnou sekvenci na seznam pomocí funkce **list**:

```
>>> list(range(50, 100))
[50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
91, 92, 93, 94, 95, 96, 97, 98, 99]
```

Výhoda číselných sekvencí oproti seznamům je, že číselná sekvence zabírá oproti ekvivalentnímu seznamu nepatrné místo v paměti - stačí si pamatovat krajní hodnoty číselné sekvence.

Všimněte si délky vykonání posledního příkazu převádějícího číselnou sekvenci na seznam.

```
>>> r1 = range(10**8)
>>> r1
range(0, 100000000)
>>> r2 = list(r1)
```

12.3 Iterace přes sekvence

Vezměme si program tisknoucí prvky seznamu:

```
l = [10, 3, 8]
for i in range(len(l)):
  print(l[i])
```

Víme, že volání **range(len(l))** vytvoří číselnou sekvenci od nuly do délky seznamu `l`. Ukážeme si, že místo tohoto výrazu můžeme použít libovolný výraz, jehož hodnota je sekvence. Rozšíříme si příkaz iterace. Pokud i je jméno proměnné, s výraz, jehož hodnota je sekvence, a b blok, pak

for i **in** s**:**
  b

je *příkaz iterace přes sekvenci*. Příkaz se vykonává podobně jako původní příkaz pro iteraci s tím rozdílem, že proměnná i postupně nabývá hodnot všech položek v sekvenci s. Blok b se tedy vykoná pro každou položku sekvence.

Předchozí program můžeme úsporněji napsat takto:

```
l = [10, 3, 8]
for element in l:
  print(element)
```

Sekvenci můžeme vytvořit libovolným výrazem. Například spojením seznamů:

```
for element in ([10, 3] + [8]):
  print(element)
```

Protože řetězce jsou sekvence, můžeme iterovat i přes ně:

```
s = 'abcdef'
for char in s:
  print(char)
```

12.4 Volba kroku v řezu a číselné sekvenci

V řezech sekvence můžeme zvolit délku kroku. Pokud s, i, j, k jsou výrazy, pak

s`[`i`:`j`:`k`]`

je *řez s volbou kroku*. Hodnotou s je sekvence, hodnoty i a j jsou indexy mezer sekvence s a hodnota k je přirozené číslo. Hodnota řezu s volbou kroku je sekvence (stejného typu jako s) každé k-té položky od indexu i po index j. Výchozí hodnotou výrazu k je jedna. Například:

```
>>> l = list(range(10))
>>> l[0:10:2]
[0, 2, 4, 6, 8]
>>> l[1:10:2]
[1, 3, 5, 7, 9]
>>> l[::3]
[0, 3, 6, 9]
>>> l[::]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Krok lze zadat jako třetí argument ve funkci **range**:

```
>>> range(0, 10, 2)
range(0, 10, 2)
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
```

Řez s krokem číselné sekvence s krokem je zase číselná sekvence s krokem:

```
>>> l = range(0, 10, 2)
>>> l
range(0, 10, 2)
>>> l[:5:2]
range(0, 10, 4)
>>> list(l[:5:2])
[0, 4, 8]
```

12.5 Změna řezu seznamu

Na levé straně od příkazu = můžeme použít i řez. Například nahrazení druhého a třetího prvku lze provést následovně:

```
>>> l = [1, 2, 3, 4, 5]
>>> l[1:3] = [6, 7]
>>> l
[1, 6, 7, 4, 5]
```

Délka řezu se nemusí rovna délce seznamu, který chceme místo řezu vložit:

```
>>> l = [1, 2, 3, 4, 5]
>>> l[1:3] = [6, 7, 8]
>>> l
[1, 6, 7, 8, 4, 5]
```

Dokonce můžeme i řez ze seznamu smazat:

```
>>> l = [1, 2, 3, 4, 5]
>>> l[1:3] = []
>>> l
[1, 4, 5]
```

Pokud použijeme řez s krokem, musí se rovnat délka řezu délce náhrady:

```
>>> l = [0, 1, 2, 3, 4, 5, 6]
>>> l[::2] = [7, 8, 9, 10]
>>> l
[7, 1, 8, 3, 9, 5, 10]
>>> l[::2] = [7, 8, 9, 10, 11]
ValueError: attempt to assign sequence of size 5 to
extended slice of size 4
```

Na pravé straně od příkazu přiřazení může být libovolná sekvence:

```
>>> l = list(range(10))
>>> l[2:4] = range(3)
>>> l
[0, 1, 0, 1, 2, 4, 5, 6, 7, 8, 9]
>>> l[4:4] = 'abc'
>>> l
[0, 1, 0, 1, 'a', 'b', 'c', 2, 4, 5, 6, 7, 8, 9]
```

Pokud se dolní i horní mez řezu rovnají, dojde k vložení sekvence do seznamu.

12.6 Záporný krok

Krok v číselných sekvencí může být i záporný. To vede na klesající číselné sekvence:

```
>>> list(range(10, 0, -1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> list(range(5, -5, -2))
[5, 3, 1, -1, -3]
```

Záporný krok lze použít i v řezech. Zde se však indexy mezer posunují o jedna doprava. Proto:

```
>>> l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[10:0:-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> l[10:0:-2]
[9, 7, 5, 3, 1]
>>> l[3:0:-1]
[3, 2, 1]
>>> l[10:0:-2] = range(5)
>>> l
[0, 4, 2, 3, 4, 2, 6, 1, 8, 0]
```

Pokud chceme do řezu se záporným krokem uvést i první prvek, musíme druhou mez vynechat:

```
>>> l = list(range(10))
>>> l[5::-1]
[5, 4, 3, 2, 1, 0]
>>> l[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

12.7 Záporný index

Položky sekvence lze číslovat odzadu zápornými čísly. Poslední prvek má index -1, před poslední -2 a tak dále. Například:

```
>>> l = list(range(10))
>>> l[-1]
9
>>> l[-2]
8
```

I mezery lze číslovat zápornými čísly. Mezera mezi posledním a předposlením prvkem má index -1. Proto můžeme použít záporné index mezer i v řezech:

```
>>> l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[-5:-1]
[5, 6, 7, 8]
>>> l[-1:-5:-1]
[9, 8, 7, 6]
>>> l[-1:-5:-1] = [10, 11, 12, 13]
>>> l
[0, 1, 2, 3, 4, 5, 13, 12, 11, 10]
```

12.8 Úkoly

Úkol 12.1. Napište funkci `my_slice`, která bere sekvenci a tři čísla i, j, k a vrací seznam položek sekvence určený řezem od i do j po k. Ve funkci nemůžete použít řez.

Úkol 12.2. Přepište předchozí funkci tak, aby používala `range` jen s jedním argumentem.

Úkol 12.3. Napište funkci, která rozhodne, zda seznam obsahuje prvky nějaké číslené sekvence. Například seznam `[1, 3, 5]` lze zapsat jako `list(range(1,6,2))`, ale seznam `[1, 3, 6]` netvoří prvky žádné číselné sekvence.

Úkol 12.4. Napište funkci, která převede seznam celých čísel na číselnou sekvenci. Funkce skončí chybou, pokud převod nelze provést. Chybu můžete způsobit vyhodnocením výrazu `1/0`. Využijte funkci z předchozího úkolu.

Úkol 12.5. Číselnou sekvenci bez kroku můžeme vyjádřit dvouprvkový seznamem skládajícím se z dolní a horní meze. Napište funkci `my_range`, která vytvoří vaší číselnou sekvenci a funkce `my_range_from` a `my_range_to`, které vrací dolní a horní mez vaší číselné sekvence.

Úkol 12.6. Napište funkce `my_range_len` a `my_range_item`, kde první vrátí délku vaší číselné sekvence a druhá prvek na daném indexu.

Úkol 12.7. Napište funkci `my_range_list`, která převede vaší číselnou sekvenci na seznam.

Úkol 12.8. Napište funkci, která spojí dvě vaše číselné sekvence. Pokud spojení nelze provést, funkce skončí chybou.

Úkol 12.9. Rozšiřte vaše číselné sekvence o zadání kladného kroku.

Úkol 12.10. Umožněte i záporný krok.

Úkol 12.11. Vytvořte podobně vaše geometrické sekvence.

13 Třináctý seminář

13.1 Vstup

Funkce **`print`** vytiskne řetězec na výstup a odřádkuje. Obráceně funkce **`input`** přečte jednu řádku ze vstupu a vrátí ji jako řetězec. Představme si program pojmenovaný jako `input.py`.

```
print(input())
```

Program spuštěný příkazem `python3 input.py` bude čekat na náš vstup. Zadáme například `jablko` a stiskneme klávesu `Return`. Program vytiskne `jablko` na výstup a ukončí se.

```
% python3 example_input_1.py
jablko
jablko
%
```

Symbol procenta znázorňuje výzvu příkazové řádky.

Funkci **`input`** můžeme dát výzvu jako argument:

```
name = input('Zadejte jméno: ')
print('Vaše jméno je ' + name + '.')
```

Program se zeptá na jméno a vytiskne větu, která jméno obsahuje. Například:

```
Zadejte jméno: Jan
Vaše jméno je Jan.
```

13.2 Převody typů

Co když budeme chtít napsat program, který očekává číslo a vytiskne číslo o jedna větší? Zkusme následující program.

```
number = input('Zadejte číslo: ')
print(number + 1)
```

Program po zadání čísla skončí chybou. Proč?

```
Zadejte číslo: 12
TypeError: can only concatenate str (not "int") to str
```

Důvodem je, že funkce **input** vrací vždy řetězec. Převést řetězec cifer v desítkové soustavě na číslo již umíme. Jednodušeji můžeme použít funkci **int**. Program opravíme:

```
number1 = int(input('Zadejte číslo: '))
print(number1 + 1)
```

a vyzkoušíme:

```
Zadejte číslo: 12
13
```

Představme si, že máme proměnnou `age`, jejíž hodnota je číslo vyjadřující věk uživatele, a chceme vytisknout větu informující uživatele o jeho věku. Naivní pokus nebude fungovat:

```
age = 24
print('Váš věk je ' + age + ' let.')
```

Program skončí chybou:

```
TypeError: can only concatenate str (not "int") to str
```

Důvodem je, že řetězec můžeme spojovat pouze s řetězcem. Převod čísla na řetězec jeho cifer v desítkové soustavě také umíme, ale opět lze převod jednodušeji uskutečnit funkcí **str**. Opravená verze:

```
age = 24
print('Váš věk je ' + str(age) + ' let.')
```

správně vytiskne:

```
Váš věk je 24 let.
```

Funkce **str** převede libovolnou hodnotu na řetězec. Můžeme tedy například napsat

```
output = ''
output += str('text')
output += '␣'
output += str(True)
output += '␣'
output += str([1, 2, 3])
print(output)
```

Na výstupu se objeví:

```
text True [1, 2, 3]
```

Kromě funkce **str** převádí hodnoty na řetězec také funkce **repr**. Rozdíl mezi nimi je v tom, že funkce **str** vrací řetězec čitelný pro člověka a funkce **repr** řetězcovou reprezentaci hodnoty. To je řetězec, který obsahuje výraz, jehož hodnota je rovna reprezentované hodnotě. Rozdíl je patrný například u řetězce:

```
>>> str('jahoda')
'jahoda'
>>> repr('jahoda')
"'jahoda'"
```

Vidíme, že druhá funkce obalila řetězec apostrofy a to z toho důvodu, že obsah řetězce můžeme dát na vstup a získat zpět hodnotu:

```
>>> 'jahoda'
'jahoda'
```

13.3 Formátování

Vraťme se k tisku věty obsahující číslo:

```
number = 12
print('Výsledek je ' + str(number) + '.')
```

Jednodušeji lze program zapsat pomocí *formátovacího řetězce*. Jeho zápis je stejný jako zápis obyčejného řetězce s tím rozdílem, že před apostrof umístíme znak `f`. Formátovací řetězec může obsahovat výrazy obklopené složenými závorkami. Například:

```
f'Jedna␣plus␣jedna␣je␣{1␣+␣1}.'
```

Hodnota formátovacího řetězce se získá tak, že se nahradí všechny výrazy v řetězci jejich hodnotami převedenými na řetězec pomocí funkce `str`. Předchozí program lze tedy úsporněji napsat takto:

```
number = 12
print(f'Výsledek je {number}.')
```

Protože se k převodu používá funkce **str**, bude řetězec vložený do formátovacího řetězce bez uvozovek:

```
score = 12
name = 'Petr'
result = f'{name} má {score} bodů.'
print(result)
```

Program vytiskne:

```
Petr má 12 bodů.
```

Do složených závorek může být vložen libovolný výraz. Například program

```
number = 12
print( f'Číslo {number} krát dva se rovná {number * 2}.')
```

vytiskne

```
Číslo 12 krát dva se rovná 24.
```

13.4 Metody řetězců

Jistě byste dokázali napsat funkci, která převede řetězec na velká písmena. Řetězce ale již mají metodu, která převod uskuteční. Metoda je funkce, která je vlastněna jistým typem. Například typ řetězec má metodu `upper`. Podobně jako funkce mají i metody výraz pro jejich zavolání. Pokud v a $a_1, \ldots, a_n$ jsou výrazy a m je jméno metody, pak

$$v\texttt{.}m\texttt{(}a_1\texttt{,}\ldots\texttt{,}v_n\texttt{)}$$

je *výraz volání metody*. Typ hodnoty v musí mít metodu m. Hodnotu v nezýváme *příjemce* (anglicky self). Hodnotou výrazu je výsledek volání metody. Například:

```
>>> name = 'praha'
>>> name.upper()
'PRAHA'
```

Čísla ale metodu `upper` nemají, proto zavolání metody skončí chybou.

```
>>> number = 1
>>> number.upper()
AttributeError: 'int' object has no attribute 'upper'
```

Podobně řetězce mají metodu `lower`, která převede řetězec na malá písmena:

```
>>> 'Olomouc'.lower()
'olomouc'
```

Ukážeme si ještě metodu `replace`, která bere dva řetězce s_1 a s_2 a v řetězci nahradí všechny výskyty řetězce s_1 za řetězec s_2. Například:

```
>>> string = 'Dám si čaj.'
>>> string.replace('čaj', 'kávu')
'Dám si kávu.'
```

Výraz v ve volání metody může být další volání metody. Takto je možné přirozeně volání metod řetězit:

```
>>> string = 'Mám pět knih.'
>>> string.replace('pět', 'šest').upper()
'MÁM ŠEST KNIH.'
```

Víme, že funkce **list** převede řetězec na seznam. Například:

```
>>> list('abc')
['a', 'b', 'c']
```

Obráceně převedení seznamu znaků na řetězec lze provést metodou `join`:

```
>>> ''.join(['a', 'b', 'c'])
'abc'
```

Příjemce zprávy je řetězec, který se vkládá mezi řetězce v seznamu. Můžeme tak spojit nejen znaky, ale i seznam řetězců za použití oddělovače:

```
>>> '-'.join(['ten', 'co', 'se', 'nebojí'])
'ten-co-se-nebojí'
```

Ve skutečnosti metoda opačná k `join` je metoda `split`, která rozdělí řetězec podle oddělovače a vrátí seznam částí:

```
>>> 'ten-co-se-nebojí'.split('-')
['ten', 'co', 'se', 'nebojí']
```

13.5 Nápověda

Funkce `help(`i`)` zobrazí nápovědu vestavěné funkce i. Například

```
>>> help(len)
```

zobrazí:

```
len(obj, /)
    Return the number of items in a container.
```

Lomítko v hlavičce můžete ignorovat.

Nápovědu ukončíte stiskem klávesy `Q`. Funkce umožňuje i zobrazit nápovědu metod. Zde je potřeba tečkou oddělit typ a název metody. Například:

```
>>> help(str.upper)
```

Zobrazí:

```
upper(self, /)
    Return a copy of the string converted to uppercase.
```

Typ `str` je řetězec.

Další nápovědu lze získat na oficiálních stránkách. Například vestavěné funkce jsou popsány zde: `https://docs.python.org/3/library/functions.html`. Popis typů včetně jejich metod lze nalézt zde: `https://docs.python.org/3/library/stdtypes.html`. Nakonec popis jazyka se nachází tady: `https://docs.python.org/3/reference/index.html`