

Základy programování v Pythonu

Jan Laštovička

23. října 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. Nejprve 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. Pojd'me 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 vytvořit 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), `//` (celočíslné 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.

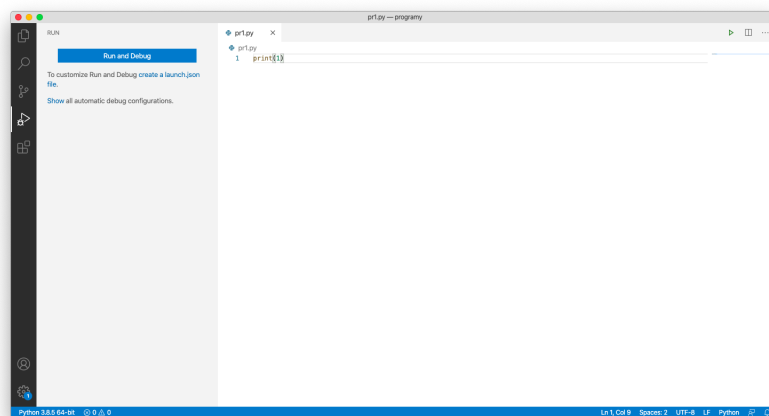
```
>>> 1age
File "<stdin>", line 1
    1age
    ^
SyntaxError: invalid syntax
```


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. Spustěte Visual Studio Code a dejte otevřít nově vytvořený adresář volbou **Open folder...** Vytvořte nový soubor s obsahem

```
print(1)
```

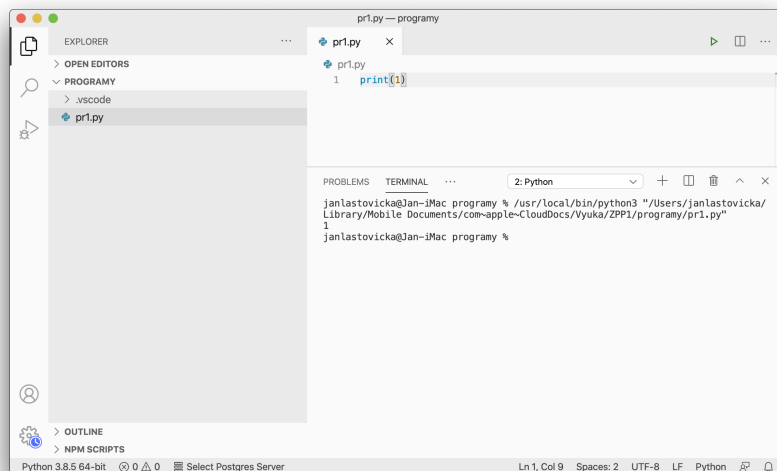
a uložte jej pod názvem `pr1.py`. V položce **Run** lišty aktivit (sloupec vlevo) si nechte vytvořit `launch.json` soubor a vyberte si první možnost (Python File).



Soubor `launch.json` můžete zavřít. Spustěte 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.

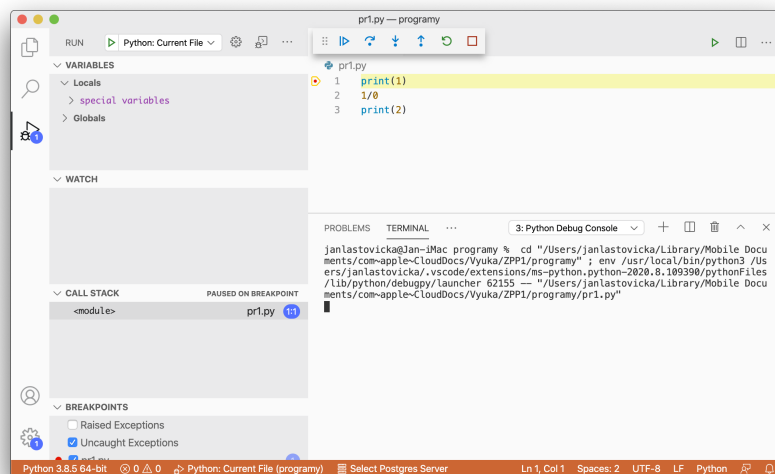
```
print(1)
1/0
print(2)
```

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 číslce 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 (•). Vložíme zarážku na první řádek (• 1 print(1)) a spustíme 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 zářezce ještě před vykonáním prvního příkazu. Nyní máme dvě možnosti. Můžeme nechat program vykonávat další příkazy od zářezky 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 zářezku 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 zářezku 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 zářezku 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 jiný 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 délku hrany spočítejte 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 trojčíselná 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 trojčiferného čísla.

Úkol 1.10. Prohod'te první a třetí cifru trojčiferného čísla.

Úkol 1.11. Vraťte nulu právě tehdy, když je čtyřčiferné 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 čtyřčiferného 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 $(0, 100)$. 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 čtyřčiferného 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 čtyřčiferné 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 tříčiferný, 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 čtyřčiferného 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 (v1 or v2)`

a výraz

`not v1 or not v2`

je ekvivalentní výrazu

`not (v1 and v2).`

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

`not v1 or v2.`

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

$$v_1 \neq v_2$$

je ekvivalentní výrazu

$$\text{not } v_1 == v_2$$

a

$$v_1 < v_2$$

je ekvivalentní výrazu

$$v_2 > v_1.$$

Zaměříme 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.

Ríkáme, 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, \dots, p_n pak

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

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

```
if v:
```

se nazývá *hlavička* a příkazy p_1, \dots, 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, \dots, 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 hlavičky 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. Seřadte 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 $x \leq 20$. 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, \dots, p_n jsou příkazy. Neboli blok je posloupnost příkazů. Nyní pokud v je podmínka, b_1 a b_2 bloky, pak

```
if v:
    b1
else:
    b2
```

je další forma příkazu větvení. Částem

```
if v:
    b1
```

a

```
else:
    b2
```

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 $v_1, v_2 \dots, v_n$ jsou podmínky a $b_1, b_2 \dots, b_n, b_{n+1}$ 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, \dots, 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 $v_1 \dots, v_n$ nemusí být vylučné. 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, \dots, v_n jsou výrazy, pak

```

print(v1, ..., vn)

```

je rozšíření příkazu tisku. Příkaz se vykoná tak, že postupně vyhodnotí výrazy v_1, \dots, 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 prvočíslu, které je buď o dva větší, nebo o dva menší než jiné prvočíslu. 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: řetězec. 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 žlut'oučký kůň úpěl d'ábelské 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, \dots, v_n je výraz

```
print( $v_1$ , ...,  $v_n$ , 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 pravouhlý 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 ukázkou 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
```

směřující 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 Eukleidova 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, \dots, v_n jsou výrazy, pak

$$f(v_1, \dots, v_n)$$

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, \dots, v_n tím získáme hodnoty h_1, \dots, h_n . Poté zavoláme funkci f s argumenty h_1, \dots, 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 přepouští, 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 řetězcí

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 řetězcí 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 = 'koby lamamalybok'

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 porovnáваме odpovídající znaky:

```

string = 'koby lamamalybok'

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. Převed'te řetězec znaků číslic na číslo zapsané těmito číslicemi. Tedy pro řetězec '456' vraťte číslo 456.

Úkol 5.2. Převed'te 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 'Kobyła 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 ' Kobyła má malý bok. ' vraťte 'Kobyła 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 pštrošicí a pštrosáčaty šli do pštrošáčárny.' 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 'mráz' 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 'mráz'. Nápoděda: Podívejte se, jak funguje zbytek po dělení ze záporného čísla.