

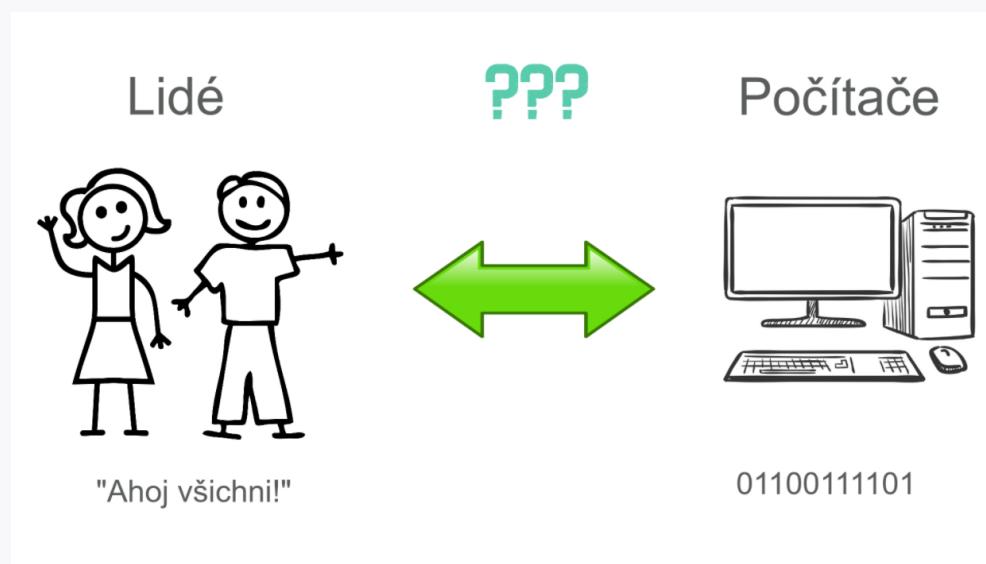


Co je to programování

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / CO JE TO PROGRAMOVÁNÍ

V nejjednodušším slova smyslu jde o **předávání instrukcí počítačům**, aby pro nás vykonali nějaký úkol. Instrukcemi myslíme nějaký **program** (kód nebo návod), který obsahuje zadání, jak má počítač postupovat.

Bohužel, počítače neumí češtinu ani jiný lidský jazyk (zatím). Nerozumí tomu, co říkáme, eventuálně příseme. Počítače rozumí pouze **strojovému kódu** (tedy jedničkám a nulám).



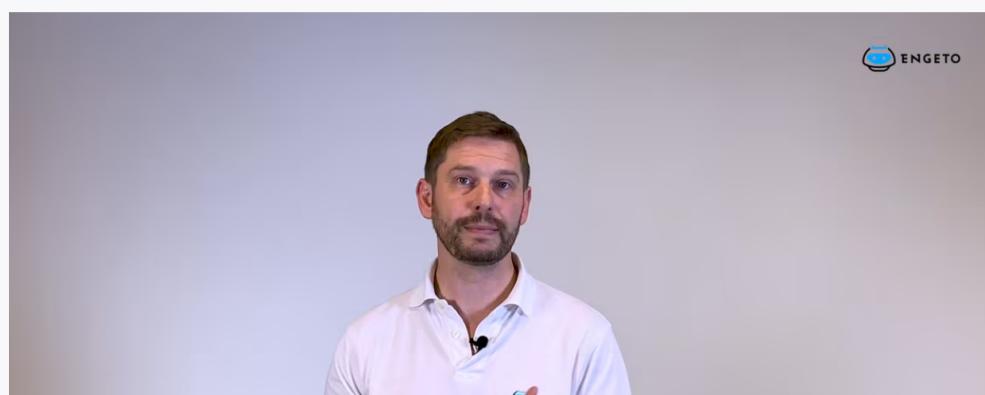
Proto existují tzv. *programovací jazyky*. Ty tvoří **pomyslný most** mezi lidskou řečí a strojovým kódem.

Proč tedy Python

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / PROČ TEDY PYTHON

Jedním z těchto *programovacích jazyků* je právě Python. Jde o jazyk, který v únoru 2021 oslavil svoje [30. narozeniny](#).

Co se základních informací o Pythonu týče, mrkněte se na video níže:





Dále se můžete podívat na [oficiální web \(v angličtině\)](#). Můžete se podívat na sekci [jak začít \(getting started\)](#), která je určené pro nováčky.

Kde získáme Python

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / KDE ZÍSKÁME PYTHON

K instalaci jazyka Python se budeme vracet v další části kurzu, kde si instalaci krok za krokem rozebereme.

Pokud už ale **máš nějaké zkušenosti** a nemůžeš se dočkat, můžeš si Python nainstalovat. Na [tomto odkazu](#) si vybereš verzi podle tvého operačního systému.

Po skončení instalace budeš mít k dispozici následující:

1. **Základní prvky jazyka** - aby Python věděl, co to znamená `print`, `str`, cyklus a podmínka,
2. **Interpret Pythonu** - tedy náš "most", který zajistí, aby počítač porozuměl našemu zápisu. Současně je to prostředí, kde si můžeš zkoušet svůj zápis. Je napsaný [v jazyce C](#) (existují samozřejmě různé implementace př. [Python](#), [PyPy](#), [RustPython](#), [IronPython](#))
3. **Některé pomocné knihovny** - o nich si povíme později.

Co je to interpret

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / CO JE TO INTERPRET

Python je tzv. **interpretovaný jazyk**. Co to vůbec znamená?

Mezi hlavní **výhody** patří možnost spouštět stejný soubor na **různých operačních systémech** (jako Windows, Linux, MacOS). Stačí jazyk nainstalovat a můžete spouštět vaše soubory.

Další výhodou je práce se soubory samotná. Stačí jej otevřít, upravit a můžete ho zase používat i se všemi změnami.

Práce s interpretovaným jazykem má ale i své **nevýhody**. Proces běží **pomaleji**, protože interpret převádí zápis čitelný lidskému oku na nuly a jedničky, kterým rozumí počítače.

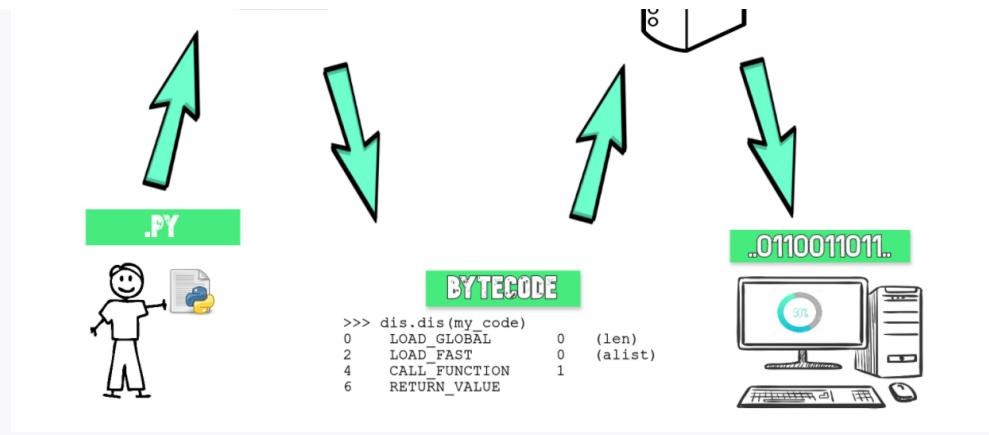
Současně vás Python sám neupozorní, že se mu nelibí nějaký chybný zápis. To zjistíte až se objeví chyba po spuštění.

Jak vypadá samotná interpretace (jen pro zvídavé)

Interpret si můžete představit jako nějaký program. Celý cyklus od zápisu, po jeho provedení probíhá následně:

1. Vytvoříme zdrojový kód v Pythonu (s příponou `.py`)
2. Spustíme jej pomocí interpretu Pythonu (standartní CPython)
3. Interpret řádek po řádku vytvoří tzv. bytecode z vašeho zdrojového kódu
4. Interpret pošle nově vytvořený bytecode do virtuálního stroje CPython
5. Ten vrací strojový kód, tedy nuly a jedničky přímo pro váš počítač





Python 2 nebo Python 3

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / PYTHON 2 NEBO PYTHON 3

Opatrně, není Python jako Python. Některé operační systémy (Linux, MacOS) mají velmi často předem nainstalovaný Python se starší verzí (<2.7). Díky těmto, dnes již [oficiálně neudržovaným verzím](#), totiž běží některé původní procesy. Současně se s těmito verzemi můžete setkat v některých materiálech na webu.

My budeme pracovat s verzemi **Pythonu 3.6 a vyšší**, abychom si společně mohli ukázat všechny funkce a procesy, které ve starších verzích nebyly dostupné.

Pokud byste chtěli vidět seznam největších změn doporučujeme skočit [sem](#).

Pokud byste postupem času potřebovali proces pro **převedení zdrojového kódu** z verze 2 do verze 3, mrkněte na [knihovnu 2to3](#).

Jak začít pracovat

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / JAK ZAČÍT PRACOVAT

V první leci není nutné instalovat ani Python, ani žádné pracovní prostředí. Budeme pracovat společně, v rámci těchto materiálů. K instalaci pracovního prostředí Pythonu a některého z editorů se vrátíme na začátku další lekce.

Pokud už máš nějaké zkušenosti a současně máš nainstalovalovaný Python z předchozí sekce, budeš potřebovat kromě samotného jazyka ještě nějaké **pracovní prostředí**. Teoreticky ti stačí prostý textový editor na tvém počítači, ale existuje spousta šikovnějších nástrojů.

Můžeš si vybrat jednu z těchto možností (ale konkrétních prostředí je mnohem více):

1. **Interpret** - interaktivní prostředí interpreta přímo v příkazovém rádku tvého počítače. Takto můžeš okamžitě spouštět a ověřovat jednoduché příkazy. Tady v materiálech můžeš použít prostředí terminálu v pravé části obrazovky (vedle **Editor kódu** najdeš **Terminál** po spuštění se objeví černá obrazovka se třemi šípkami na začátku posledního řádku **>>>**)
2. **Textový editor** - upravený textový editor, který umí formátovat text, napovídat, atd. Na ukázku se můžeš podívat třeba na [Sublime text](#) nebo [Atom](#).
3. **Vývojařské prostředí** - specializované prostředí obsahující různé nástroje a pomůcky (sofistikovanější, často větší než editor i náročnější na paměť). Podívej se třeba na [PyCharm](#), [Visual Studio Code](#) nebo [replit](#).
4. **Notebooky** - speciální prostředí, které umožňuje využít potenciál interpretu a současně zapisovat poznámky, zobrazovat obrázky, grafy aj. Přečísl si můžeš o [projektu Jupyter](#) nebo [Google Colaboratory](#).



colab

ČÍSELNÉ HODNOTY

Úvod do datových typů

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / ČÍSELNÉ HODNOTY / ÚVOD DO DATOVÝCH TYPŮ

Pro naučení jakéhokoliv programovacího jazyků, si potřebuješ osvojit znalosti z tzv. *tří teoretických pilířů*. Na těchto pilířích stojí téměř všechny programovací jazyky:

1. **Syntaxe** (funkce, podmínky, smyčky, aj.)
2. **Datové typy** (čísla, sekvence, aj.)
3. **Knihovny** (decimal, aj.)

Společně začneme u druhého pilíře, tedy základních datových typů. **Datové typy**, nebo také **datové struktury**. Začneme u něčeho, co je nám blízké a to jsou číselné hodnoty. Python je standardně vybavený dvěma číselnými datovými typy:

1. **int**, celá čísla (z angl. *integer*)
2. **float**, desetinná čísla (z angl. *float*)

Celá čísla

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / ČÍSELNÉ HODNOTY / CELÁ ČÍSLA

Pokud uvidíš ukázkou zápisu v **černém rámečku**, neváhej a vyzkoušej si příkaz v terminálu tady v materiálech (začneš psát za symbol **>>>** a potvrzuješ klávesou **Enter**)



The screenshot shows the Python Academy course interface. At the top, there are tabs for 'INTERPRET', 'EDITOR', 'IDE', and 'IPYNB'. Below these, there are icons for various tools: a terminal window icon, Visual Studio Code, PyCharm, Atom, and Jupyter Notebook. The main content area has a dark header 'DOJO / ONLINE PYTHON AKADEMIE' and '1. Úvod do programování v Pythonu'. It lists three topics: 'Co je to programování', 'Proč tedy Python', and 'Kde získáme Python'. On the right, there's a progress bar showing '71% hotovo z Lekce 1' and a search bar with 'Vyhledávat'.

Společně začneme u druhého pilíře, tedy základních datových typů. Datové typy, nebo také datové struktury. Začneme u něčeho, co je nám blízké a to jsou číselné hodnoty. Python je standardně vybavený dvěma číselnými datovými typy.

1. **int**, celá čísla (z angl. *integer*)
 2. **float**, desetinná čísla (z angl. *float*)

The screenshot shows a navigation sidebar on the left with categories like 'Co je to interpret', 'Python 2 nebo Python 3', and 'Jak začít pracovat'. Below it, under 'ČÍSELNÉ HODNOTY', are sections for 'Úvod do datových typů', 'Celé čísla', 'Desetinná čísla', 'Aritmetické operace', 'Komplikace se stringy', and 'Kvíz'. The 'Celé čísla' section is currently selected. The main content area has a header 'Celé čísla' and a sub-header 'ONLINE PYTHON AKADEMIE / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / ČÍSELNÉ HODNOTY / CELÁ ČÍSLA'. It contains text about writing numbers in black brackets and includes two code examples:

```
>>> 100 + 200 # přepisuj bez šipek součet hodnot  
300 # ... a sleduj, co bude výstupem
```

V zápisu je vhodné **psát mezery**, aby byl přehlednější. Je to pouze obecně platné doporučení. Výstup bude stejný, pokud je nepoužijeme:

```
>>> 100+200 # přepisuj bez šipek součet hodnot  
300 # ... a sleduj, co bude výstupem
```

V zápis je vhodné **psát mezery**, aby byl přehlednější. Je to pouze obecně platné doporučení. Výstup bude stejný, pokud je nepoužijeme:

```
>>> 100+200 # přepisuj bez šipek součet hodnot  
300 # ... a sleduj, co bude výstupem
```

Všimni si symbolu **#**, který jsme použili v obou zápisech výše. Mřížka naznačuje, že jde o **jednořádkový komentář**. Python bude zápis za ní ignorovat, takže ji můžeš využít pro tvoje vlastní vysvětlivky.

Dále si vyzkoušej základní aritmetické operace:

```
>>> 100 + 200  
300  
>>> 300 - 100  
200  
>>> 200 * 100  
20000
```

Znaménko mezi číslami se často označuje jako tzv. *operátor* (čísla na jeho stranách potom jako *operands*). Pokud na toto označení narazíš, tak at říkám cizí.

Všimni si, jak klasické **dělení** nevrátí celé číslo, ale **desetinné číslo**. Není to žádná chyba, ale záměrný účel tohoto operátoru.

```
>>> 700 / 350  
2.0
```

Pokud si chceš ověřit, že jde skutečně o datový typ celých čísel, vyzkoušej pomocí našeho terminálu funkci **type** (více si o funkci povíme za chvíli).

```
>>> type(1)  
<class 'int'>  
>>> type(100)  
<class 'int'>
```

V obou případech jsme dostali zpátky výstup interpreta, jehož formát nám zatím nemusí být jasné. Zásadní je část toho výstupu, kde stojí **int** (tedy integer). Takže jde skutečně o celé číslo a Python to ví.

Desetinná čísla

Základní aritmetické operace jsou stejné jako u celých čísel. Desetinným oddělovačem je **tečka**, čárka slouží pro jiné účely (opět si povíme v dalších kapitolách).

Opět si vyzkoušej použití základních operátorů v prostředí našeho terminálu:

```
>>> 0.1 + 0.1  
0.2  
>>> 0.5 - 0.3  
0.2  
>>> type(0.3)  
<class 'float'>
```

Občas se při práci s desetinnými čísly setkáš s fenoménem známým jako *plovoucí řádová čárka*. Ten je způsobený tím, že některá desetinná čísla nemají odpovídající binární tvar. Proto jsou použity přibližné hodnoty.

```
>>> 0.1 + 0.2  
0.3000000000000004
```

Nemusíš se ničeho obávat, není to chyba na tvé straně, ale obecný fakt. Pokud budeš do budoucna potřebovat práci s přesnými desetinnými čísly, doporučujeme pracovat s knihovnou **decimal** (o práci s knihovnami se budeme bavit v pozdějších lekcích)

Aritmetické operace

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / ČÍSELNÉ HODNOTY / ARITMETICKÉ OPERACE

Seznam všech základních aritm. operátorů najdete v tabulce níže:

Operátor	Proces
+	Sčítání
-	Odčítání
*	Násobení
/	Dělení
//	Celočíselné dělení
%	Modulo (zbytek po dělení)
**	Umocňování

První 4 operace jsou poměrně dobře jasné. Podívejme se nyní na zbývající tři:

Celočíselné dělení

```
>>> 10 // 3  
3  
>>> 11 // 2  
5
```

Interpret vezme výslednou hodnotu a zaokrouhlí ji na nižší celé číslo.

Modulo (~ zbytek po dělení)

```
>>> 10 % 3  
1  
>>> 11 % 3
```

Interpret vezme hodnotu na pravé straně (3) a snaží se ji vložit do hodnoty na levé straně, kolikrát jen může (3 * 3 = 9). Poté vezme hodnotu na levé straně a odečte od ní maximální násobek hodnoty z pravé strany (11 - (3 * 3)). Výsledkem je potom jejich rozdíl -> modulo.

Umocňování

```
>>> 2 ** 2 # hodnota dva na druhou
4
>>> 2 ** 3 # hodnota dva na třetí
8
```

Komplikace s čísly

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / ČÍSELNÉ HODNOTY / KOMPLIKACE S ČÍSLY

```
>>> 2 + 3 * 2
8
```

Různé operátory mají různé priority. Pokud budete používat různé operátory, doporučujeme rozlišovat mezi jejich důležitostí. To potom pomůže čtenáři tohoto zápisu lépe pochopit souvislosti. Okolo operátoru s vyšší prioritou nebudeme psát mezery:

```
>>> 2 + 3*2
8
```

V takovém případě bude čtenáři našeho zápisu na první pohled jasné, která operace má přednost. Dále můžete používat klasické kulaté závorky:

```
>>> 2 + (3 * 2)
8
```

Tabulka s hierarchií operátorů

Pořadí	Operátor	Proces
1.	()	Závorky
2.	**	Umocňování
3.	*	Násobení
4.	/	Dělení
5.	+	Sčítání
6.	-	Odčítání

Kvíz

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / ČÍSELNÉ HODNOTY / KVÍZ

1/4

Kdy použijeme datový typ `float`?

- A. Float použijeme, pokud potřebujeme pracovat s tzv. komplexními čísly
- B. Float použijeme, pokud potřebujeme pracovat s celými čísly
- C. Float použijeme, pokud potřebujeme pracovat s desetinnými čísly
- D. Float použijeme při práci s římskými číslicemi

TEXTOVÉ HODNOTY

String

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / TEXTOVÉ HODNOTY / STRING

String, tedy řetězec je různě dlouhé uskupení znaků (písmen, čísel, speciálních symbolů). Dále se označuje jako sekvence, kterou jakmile jednou vytvoříme **nelze změnit** (z angl. *immutable*).

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

Písmena v předchozí ukázce jsou zapsaná pomocí dvojitých uvozovek na začátku a konci textu.

```
>>> type("Python")  
<class 'str'>
```

Opět se ujistíme, že jde skutečně o datový typ **string** (stačí si všimnout pouze klíčového **str**)

Jak zapsat string

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / TEXTOVÉ HODNOTY / JAK ZAPSAT STRING

Pro vytvoření nové hodnoty typu **str** můžeme použít:

1. Jednoduché uvozovky
2. Dvojité uvozovky
3. Trojité uvozovky

```
>>> 'Marian'  
'Marian'  
>>> "Marian"  
'Marian'  
>>> """Marian"""  
'Marian'  
>>> '''Marian'''  
'Marian'
```

Běžně se používají **první dvě varianty**. Je to jenom na tobě, která se ti bude líbit více. Důležité je použít stejně uvozovky jak **na začátku, tak na konci**.

String můžeš vytvořit současně z různých znaků:

```
>>> "Python"
'Python'
>>> "Python3"
'Python3'
>>> "Python3.9"
'Python3.9'
>>> type("12345")
<class 'str'>
>>> type("@#!")
<class 'str'>
```

Komplikace se stringy

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / TEXTOVÉ HODNOTY / KOMPLIKACE SE STRINGY

Může nastat situace, kdy budeš potřebovat doplnit písmena ve stringu o uvozovku:

```
>>> 'It's friday'
  File "<stdin>", line 1
    'It's friday'
    ^
SyntaxError: invalid syntax
```

To, co vidíš v předchozí ukázce je výstup, který oznamuje, že máš v zápisu stringu chybu.

SyntaxError konkrétně naznačuje, že jde o prohřešek proti základním předpisům jazyka Python.

Pro opravení takového zápisu, si ukážeme několik řešení:

```
>>> "It's friday"
"It's friday"
```

Pokud použijeme dvojité ohraničující uvozovky a jednoduché vnitřní, příkaz bude fungovat správně.

Pokud budeš chtít ale použít i ty, problém může přetrvat:

```
>>> "It's "kind of" friday"
  File "<stdin>", line 1
    "It's "kind of" friday"
    ^
SyntaxError: invalid syntax
```

V takovém případě budeme muset použít speciální znak \ (zpětné lomítko). Pokud v rámci stringu interpret narazí na symbol \ počítá s tím, že další znak bude mít speciální význam.

```
>>> "It\'s \"kind of\" friday"
'It\'s "kind of" friday'
```

První lomítko nám ve výstupu zůstalo ale pokud hodnotu vložíme třeba do funkce **print** (o které si povíme za chvíli víc) uvidíme výstup bez lomítek.

```
>>> print("It\'s \"kind of\" friday")
It's "kind of" friday
```

Použití speciálních symbolů souvisejících se zpětným lomítkem je víc. Jsou to tzv. *escape characters*. V tabulce níž najdeš soupis těch nejčastějších:

Speciální znak	Význam
\'	Apostrof
\n	Zpětné lomítko
\n	Nový řádek
\r	<i>Return carriage</i>
\t	Tabulátor
\b	<i>Backspace</i>
\f	<i>Form feed</i>

Speciální symboly uvidíš, pokud si nyní vyzkoušíš zápis s třemi uvozovkami na začátku a na konci:

```
>>> """
... Ahoj,
... vsem!
...
'\nAhoj, \nvsem!\n'
```

Už při zápisu si můžeš všimnout jak ti interpret dovolí pokračovat na dalších řádcích (pomocí třech teček `...`). To je současně i podstatou třech uvozovek na začátku a na konci. Umožní ti zapsat hodnotu typu `str` na více řádků.

```
>>> """
... Ahoj,
... jste jednou!
...
'\nAhoj, \njste jednou!\n'
```

V obou výstupech si můžeš všimnout znaku `\n`. Tím interpret indikuje, že si zapisoval na nový řádek. Současně tento *escape character* najdeš v předešlé tabulce.

Spojování stringů

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / TEXTOVÉ HODNOTY / SPOJOVÁNÍ STRINGŮ

Operátor `+` u číselných typů provede součet. Můžeš jej použít i u stringů. Tady ovšem ne za účelem sčítání, ale **spojování stringů**:

```
>>> "Prvni" + "Druhy"
'PrvniDruhy'
>>> "123" + "456"
'123456'
>>> "+" + "-" + "*" + "/"
'+-*/'
```

Opakování stringů

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / TEXTOVÉ HODNOTY / OPAKOVÁNÍ STRINGŮ

Podobně jako u spojování má operátor `*` jiný účel u číselných typů a u stringů. Pokud zapíšete string, poté `*` a za ní celé číslo, zadaný string zopakujete tolíkrát, jaké bylo zadané číslo.

```
>>> "M" * 10
'MMMMMMMMM'
```

```
>>> "opakujeme" * 4  
'opakujemeopakujemeopakujemeopakujeme'
```

Indexování

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / TEXTOVÉ HODNOTY / INDEXOVÁNÍ

Znaky, ze kterých je string složený mají **pořadí**. Toto pořadí je určené celým číslem, tzv. **index**. Pomocí tohoto indexu máme možnost vybrat konkrétní znak.

STRING:

'AUTOBUS'

INDEXY OD ZAČÁTKU:

0	1	2	3	4	5	6
---	---	---	---	---	---	---

A	U	T	O	B	U	S
---	---	---	---	---	---	---

INDEXY OD KONCE:

-7	-6	-5	-4	-3	-2	-1
----	----	----	----	----	----	----

INDEXOVÁNÍ STRINGŮ

Ihned za konkrétní hodnotu napišeme hranatou závorku a do ní celé číslo indexu, který potřebujeme.

```
>>> "autobus"[1]  
'u'
```

Příklad výše můžeš přečíst jako *Ze stringu autobus mi vypiš znak na indexu 1*. Z toho vyplývá, že indexování obecně začíná celým číslem **0**, tedy první znak.

```
>>> "autobus"[0]  
'a'  
>>> "autobus"[1]  
'u'  
>>> "autobus"[2]  
't'  
>>> "autobus"[3]  
'o'
```

Pokud budeš potřebovat indexovat **od konce**, můžeš pracovat s negativním indexem. Poslední hodnotu získáš pomocí indexu **-1**, předposlední hodnotu pomocí indexu **-2**, atd.

```
>>> "autobus"[-1]  
's'  
>>> "autobus"[-2]  
'u'  
>>> "autobus"[-3]  
'b'
```

Slicing

Pokud budete potřebovat pracovat pouze s částí datového typu **str**, můžete jej *rozkrájet* (z angl. *slicing*).

```
>>> "autobus"[0:4]
'auto'
```

Stejně jako u **indexování** použijeme hranatou závorku. Tentokrát ji doplníme **dvojtečkou** a **dalším celým číslem**. První hodnota je potom **počáteční index**, druhá hodnota je **konečný index**.

Opatrně **druhý index** je braný **nepřímo**, takže abys získal znak z indexu **6**, musíš napsat číslo **7**:

```
>>> "autobus"[0:6]
'autobu'
>>> "autobus"[1:7]
'autobus'
```

Dále je možné zápis zkrátit. Pokud si budeš zápisem s pomocí indexů jistý, můžesh vyzkoušet následující:

1. **"autobus"[:2]** - vynecháš první index a začneš dvojtečkou (původně **[0:2]**)
2. **"autobus"[2:]** - vynecháš druhý index a končíš dvojtečkou (původně **[2:7]**)

```
>>> "autobus"[0:2]      # první dvě písmena
'au'
>>> "autobus":[2]       # první dvě písmena, zápis jedním číslem
'au'
>>> "autobus":[2:7]     # od třetího písmena do konce
'tobus'
>>> "autobus":[2:]      # od třetího písmena do konce, zápis jedním číslem
'tobus'
```

Striding

Nakonec operace známá jako **přeskakování** (z angl. *striding*), umožňuje získat každý n-tý údaj ze stringu.

Doplníme **třetí celočíselnou hodnotu** do hranaté závorky, oddělenou dvojtečkou:

```
>>> "autobus"[0:7:2]
'atbs'
```

Hodnota **2** zapsaná v hranaté závorce výš říká, že vezme nejprve index **0**, a potom každý druhý index (tedy indexy **2, 4, 6**).

```
>>> "autobus":[1:7:2]"
```

Upravíme hodnoty v hranatých závorkách, ale platí pořad stejně pravidlo. Vezmeme nejprve index **1** (tedy **"u"**), a potom každý druhý index (tentokrát indexy **3, 5**).

Opět je možné zápis **zkrátit**. Vynech hodnoty indexů pro **počátek** a **konec**. Zápis pouze dvě dvojtečky a poslední hodnotu pro **přeskakování**:

```
>>> "autobus":[::2]"
'atbs'
```

Dokonce můžeš použít **zápornou hodnotu** pro přeskakování pozpátku:

```
>>> "autobus":[-1] # obracene poradi, zacne 's', jeden znak za druhym
'subotua'
>>> "autobus":[-2] # obracene poradi, zacne 's', pote kazdy druhy znak
'sbta'
>>> "autobus":[-3] # obracene poradi, zacne 's', pote kazdy treti znak
'soa'
```

Kvíz

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / TEXTOVÉ HODNOTY / KVÍZ

1/4

Jaký bude výsledek této operace?: `'5' * 4`

A. `20`

B. `'54'`

C. `'5555'`

D. `'20'`

PROMĚNNÉ V PYTHONU

Pamatuj si informace

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / PROMĚNNÉ V PYTHONU / PAMATUJ SI INFORMACE

Proměnná (angl. *variable*) je v podstatě místo, kam schováme naše hodnoty, abychom je mohli později (opakováně) použít.

Vytvoření proměnné

Pro vytvoření proměnné musíš dodržet tento postup:

```
>>> jmeno = "Lukas"
```

1. `jmeno` - na levé straně od rovnítka je **jméno proměnné**. Jak pojmenovat proměnnou si povíme za chvíli,
2. `=` symbol, který **přiřadí zadanou hodnotu** ke jménu proměnné,
3. `"Lukas"` - je **hodnota**, kterou si chceme uložit na později (nebo ji použít více než 1x). V tomto případě je to *string*, ale můžeš uložit také číselné hodnoty aj.

Pokud potřebuješ hodnotu použít, Python ji najde pomocí jména, které jsme si před chvílí vytvořili:

```
>>> jmeno  
'Lukas'
```

Jak pojmenovat proměnnou

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / PROMĚNNÉ V PYTHONU / JAK POJMENOVAT PROMĚNNOU

Pravidla pro pojmenování

Jméno proměnné (někdy také odkaž) **může obsahovat** tyto znaky:

1. **Písmenné** znaky,

2. **Číselné** znaky,

3. **Podtržitka**

```
>>> jmeno = "Lukas"
>>> jmeno2 = "Matous"
>>> moje_jmeno = "Jan"
```

Existují ovšem znaky, které jméno **obsahovat nesmí**:

1. Jméno proměnné nesmí **začínat číselným znakem**.

2. Jméno proměnné nesmí **obsahovat speciální znaky** (kromě podtržítka)

3. Jméno proměnné nesmí **obsahovat mezery**

```
>>> 2jmeno = "Marek"
      File "<stdin>", line 1
          2jmeno = "Marek"
          ^
SyntaxError: invalid syntax
```

```
>>> moje jmeno = "Tomas"
      File "<stdin>", line 1
          moje jmeno = "Tomas"
          ^
SyntaxError: invalid syntax
```

```
moje@jmeno = "Zdenek"
      File "<stdin>", line 1
      SyntaxError: can't assign to operator
```

Další doporučení

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / PROMĚNNÉ V PYTHONU / DALŠÍ DOPORUČENÍ

Tady jsou některá další doporučení, které mají uživatelům pomáhat při zapisování a čtení.

Styl pojmenování

Pojmenování ve tvém zápisu by mělo být **konzistentní**, proto si vyber jeden styl a toho se drž:

camelCase (počáteční písmeno malé, každé první písmeno nového slova velké):

```
>>> mojeJmeno = "Matous"
>>> novaHodnota = 1234
```

snake_case (slova zapsaná malými písmeny, oddělená podtržítkem):

```
>>> moje_jmeno = "Matous"  
>>> nova_hodnota = 1234
```

Konstanty

Pokud se hodnota proměnné nebude měnit (~zůstane konstantní), zapisujeme celé jméno velkými písmeny:

```
>>> TIHOVE_ZRYCHLENI = 9.81  
>>> PI = 3.141
```

Komplikované čtení

Pokud budete chtít jméno proměnné **označit jedním písmenem**, doporučujeme **nepoužívat** **I** (malé e), **O** (velké ó), **I** (velké í). Může totiž snadno dojít k jejich záměně s jiným písmenem.

Dále existují různé druhy písma, kde jsou písmena k nerozeznání od **nuly** a **jedničky**.

Práce s proměnnými

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / PROMĚNNÉ V PYTHONU / PRÁCE S PROMĚNNÝMI

Proměnná je pouze odkaz na hodnotu (jako je *string*, *integer*, *float*), proto je můžeš používat stejně, jak jsme si doposud ukazovali:

integer

```
>>> jedno_cislo = 5  
>>> druhe_cislo = 6  
>>> jedno_cislo + druhe_cislo  
11
```

string

```
>>> jmeno = "Karel"  
>>> prijmeni = " Novak"  
>>> jmeno + prijmeni  
'Karel Novak'
```

Kvíz

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / PROMĚNNÉ V PYTHONU / KVÍZ

1/4

V průběhu programu **ukládáme hodnoty** pro jejich pozdější využití. Jak to děláme?

- A. Hodnotu uložíme do proměnné pomocí dvojtečky

B. Hodnotu přiřadíme proměnné pomocí znaménka rovná se (=)

C. Za hodnotu napíšeme šipku a název proměnné

D. Za chodu programu nelze ukládat hodnoty

SEKVENČNÍ DATOVÉ TYPY

Úvod k sekvencím

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / SEKVENČNÍ DATOVÉ TYPY / ÚVOD K SEKVENCÍM

Doposud jsme si ukázali jak pracovat s proměnnou, která obsahuje **jedno číslo** (`int`, `float`), nebo **jeden řetězec** textových znaků (`str`).

Pojďme si nyní ukázat, že Python umí pracovat i s údaji, které obsahují více různých informací jako několik čísel, nebo několik textových hodnot.

Takové hodnoty potom budeme označovat jako tzv. **sekvenční datové typy** (tedy v jedné proměnné bude několik oddělených údajů). Obecně Python nabízí tyto tři základní sekvenční typy:

1. `list` (z angl. *list*, česky *seznam*),
2. `tuple` (z angl. *tuple*, česky *n-tice*),
3. `range` (z angl. *range*, česky *rozsah*) - na něj přijde řada později.

List

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / SEKVENČNÍ DATOVÉ TYPY / LIST

List je opravdu datový typ, který je doslova tvořen seznamem údajů. Tyto údaje jsou oddělené datovým oddělovačem čárkou.

```
>>> muj_seznam = ["Matous", "Marek", "Lukas", "Jan"]
```

V příkladu si můžeme všimnout některých **charakteristických rysů** pro `list`:

1. **Hranaté závorky** na začátku a na konci listu,
2. **stringy**, které náš list obsahuje,
3. **čárky**, které oddělují jednotlivé hodnoty,
4. **proměnná**, do které si nově napsaný list schovám (`muj_seznam`).

Opět si můžeš pomocí funkce `type` ověřit datový typ. Není nutné chápát celkový význam výstupu funkce `type`. Stačí si povšimnout výrazu `list` ve výstupu.

```
>>> type(["Matous", "Marek", "Lukas", "Jan"])
<class 'list'>
```

Jak vytvořit list

Nejprve si ukážeme možnosti, jak **vytvořit prázdný list**, kam si budeš moc v budoucnu ukládat svoje hodnoty:

1. Možnost, pomocí **prázdných hranatých závorek**,
2. Možnost, pomocí zabudované funkce **list()**.

```
>>> prvni_seznam = []
>>> druhý_seznam = list()
>>> type(prvni_seznam)
<class 'list'>
>>> type(druhý_seznam)
<class 'list'>
```

Opět použijeme funkci **type** pro ověření, že výsledné hodnoty jsou skutečně typu **list**.

Pokud potřebuješ vytvořit **neprázdný list**, můžeš údaje zapsat přímo do hranaté závorky (jako první úkazka v této kapitole):

```
>>> treti_seznam = [2, 4, 6, 8, 10]
>>> ctyrty_seznam = [1.0, 3.0, 5.0, 7.0, 9.0]
>>> type(treti_seznam)
<class 'list'>
>>> type(ctyrti_seznam)
<class 'list'>
```

Jak pracovat s listem

Hodnoty, které **list** obsahuje můžeš zpřístupnit pomocí jejich **pořadí**, tedy indexů. Tento princip funguje stejně jako jsme si ukázali u stringů.

```
>>> muj_seznam = ["Matous", "Marek", "Lukas", "Jan"]
>>> muj_seznam[0]
'Matous'
>>> muj_seznam[1]
'Marek'
>>> muj_seznam[-1]
'Jan'
```

Tedy index **0** představuje **první hodnotu** a index **-1** **poslední hodnotu**.

Tuple

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / SEKVENČNÍ DATOVÉ TYPY / TUPLE

Tuple je na první pohled velice podobný **listu** (seznamu):

```
>>> muj_tupl = ("Matous", "Marek", "Lukas", "Jan")
```

Pro srovnání s **list**:

```
>>> muj_seznam = ["Matous", "Marek", "Lukas", "Jan"]
```

V příkladu si můžeme všimnout některých **charakteristických rysů** pro **tuple**:

1. **Kulaté závorky** na začátku a na konci tuplu,
2. **stringy**, které nás list obsahují,
3. **čárky**, které oddělují jednotlivé hodnoty,
4. **proměnná**, do které si nově napsaný tupl schováme (`muj_tupl`).

Možná se tedy ptáš, proč je nutné mít jak `list`, tak `tuple`, když jsou tak podobné. Hlavním rozdílem je **změnitelnost**.

Sekvenční typ	Změnitelnost	Vysvětlení
<code>list</code> (~seznam)	<i>mutable</i> (~změnitelný)	Můžeš přidávat a odebírat hodnoty
<code>tuple</code> (~n-tice)	<i>immutable</i> (~nezměnitelný)	Jakmile jej vytvoříš, nelze změnit

Z tabulky uvedené výše vyplývá, že pokud chceš pracovat se sekvencí, u které budeš v průběhu **měnit její obsah**, použiješ `list` (~seznam).

Naopak pokud budeš chtít jako programátor napsat takovou sekvenci, kterou si **nepřeješ změnit** (a dát to naučenoměřenou sobě nebo ostatním programátorům), použiješ `tuple`. Podívej se na ukázku níže:

```
>>> nejvetsi_mesta = ("Praha", "Brno", "Ostrava", "Plzen", "Liberec", "Olomouc")
```

V tuplu `nejvetsi_mesta` jsou všechna města v České republice, která mají více než 100 000 obyvatel. Pro nás je toto zásadní hodnota a nechceme, aby do této proměnné kdokoliv přidal nějaký další údaj. Na základě této potřeby jsme vybrali `tuple`.

Jak vytvořit tuple

Vzhledem k faktu, že je `tuple` nezměnitelný, není vytvoření prázdného tuplu moc výhodné. Nicméně, pokud bychom to vážně potřebovali, postupujeme jako u listu:

1. Pomocí **prázdných kulatých závorek**,
2. Pomocí zabudované **funkce `tuple()`**.

```
>>> prvni_tupl = ()
>>> druhý_tupl = tuple()
>>> type(prvni_tupl)
<class 'tuple'>
>>> type(druhý_tupl)
<class 'tuple'>
```

Ovšem takové datové struktury nejsou příliš užitečné (obvykle vyžadujeme různé hodnoty), a proto se zaměříme na vytvoření **neprázdných tuplů**:

```
>>> treti_tupl = ("Praha", "Berlin", "Varsava", "Bratislava", "Vidēn")
>>> ctvrty_tupl = 1.3, 3.6, 1.8, 0.4, 1.9
>>> type(treti_tupl)
<class 'tuple'>
>>> type(ctvrty_tupl)
<class 'tuple'>
```

Ačkoliv varianta bez kulatých závorek není zcela běžná, můžeš se s ní setkat.

Jak pracovat s tuplem

Stejně jako `list` můžeš i v tuplu získat hodnoty z konkrétních indexů, případně z jej *rozkrájet* (~slicing):

```
>>> treti_tupl = ("Praha", "Berlin", "Varsava", "Bratislava", "Vidēn")
```

```
>>> treti_tupl[0]
'Praha'
>>> treti_tupl[-1]
'Vidēn'
>>> treti_tupl[0:2]
('Praha', 'Berlīn')
>>> treti_tupl[-2]
'Bratislava'
```

Kvíz

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / SEKVENČNÍ DATOVÉ TYPY / KVÍZ

1/3

Který z níže uvedených datových typů je **list** (~seznam):

A. **"Martin"**

B. **1234**

C. **[1, 2, 3, 4]**

D. **(1, 2, 3, 4)**

E. **1, 2, 3, 4**

ZABUDOVANÉ FUNKCE

Úvod do funkcí

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / ZABUDOVANÉ FUNKCE / ÚVOD DO FUNKCÍ

Obecně řečeno Python disponuje několika typy funkcí. Nás budou zajímat hlavně tyto:

1. **Zabudované funkce** (z anglicky *built-in functions*),
2. **Uživatelské funkce** (z anglicky *user-defined functions*) - ty přijdou na řadu později.

Funkce jsou v podstatě pomocné nástroje, které ti umožní snazší a efektivnější práci (např. zadávat hodnoty do tvého programu nebo měnit datovou strukturu jedné hodnoty najinou).

To, že nesou označení **zabudované** znamená, že je máš k dispozici ihned po instalaci. Tedy v každém souboru (s příponou **.py**), který do budoucna vytvoříš.

Použití zabudované funkce

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / ZABUDOVARÉ FUNKCE / POUŽITÍ ZABUDOVARÉ FUNKCE

V uplynulých kapitolách jsme již některé **zabudované funkce** viděli. Byla to zrovna funkce `type`. Pojdme si nyní ukázat jak se taková zabudovaná funkce používá:

```
>>> type("abc")
<class 'str'>
>>> type(3.1416)
<class 'float'>
>>> type(["m", "h"])
<class 'list'>
>>> muj_typ = type("@")
>>> muj_typ
<class 'str'>
```

Všimni si, že její spuštění má jistá **pravidla**:

1. spustím tuto funkci pomocí jejího jména, `type`,
2. ihned za jménem následují **kulaté závorky**,
3. do kulatých závorek zapisujeme **hodnoty**, se kterými chceme pracovat,
4. zabudované funkce umějí často pracovat s **různými datovými typy** (`str`, `float`, `list`)
5. výsledek, který získám použitím funkce si můžeme schovat do proměnné.

Výpis zabudovaných funkcí

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / ZABUDOVARÉ FUNKCE / VÝPIS ZABUDOVARÉ FUNKCÍ

Účelem tohoto materiálu není žádné šprtání všech jmen funkcí. Ani zkušenější programátoři si nepamatují všechny. Důležité je vědět, **kam se podívat**, až je budeš potřebovat.

Níže najdeš tabulku s **některými** zabudovanými funkcemi (ne se všemi!)

Jméno funkce	Účel funkce
<code>type</code>	Vrací datový typ zadané hodnoty
<code>round</code>	Zaokrouhlí zadanou hodnotu na stanovený počet desetinných míst
<code>abs</code>	Vrací absolutní hodnotu
<code>int</code>	Vrací <code>integer</code> ze zadaného stringu nebo číselného údaje
<code>float</code>	Vrací <code>float</code> ze zadaného stringu nebo číselného údaje
<code>str</code>	Vrací <code>string</code> ze zadané hodnoty
<code>list</code>	Vrací nový objekt, sekvenční datový typ <code>list</code>
<code>tuple</code>	Vrací nový objekt, sekvenční datový typ <code>tuple</code>
<code>help</code>	Vratí návod k zadánemu objektu
<code>print</code>	Vypisuje zadné hodnoty jako výstupy
<code>input</code>	Umožnuje ukládat vstupy od uživatele
<code>len</code>	Vrací délku zadané hodnoty
<code>max</code>	Vrací největší hodnotu
<code>min</code>	Vrací nejmenší hodnotu

sum

Vrací součet všech hodnot

pow

Vrací zadanou hodnotu umocněnou o zadaný exponent

Pokud máš se zabudovanými funkciemi nějaké zkušenosti, nebo tě zajímá, které další bys mohl v rámci Pythonu využít, mrkn na [oficiální tabulku všech zabudovaných funkcí](#).

Ukázky použití funkcí

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / ZABUDOVAÑE FUNKCE / UKÁZKY POUŽITÍ FUNKCÍ

Nakonec této kapitolky si některé ze zabudovaných funkcí vyzkoušíme, ať se ti s nimi později snáze pracuje.

help

Funkce `help` ti umožní zobrazit nápovědu. Samozřejmě můžeš pořád hledat na internetu, ale občas je dobré mít poznámky i offline. Pojďme si společně vypsat nápovědu pro datový typ `str`:

```
>>> help(str)
Help on class str in module builtins:
class str(object)
...
...
```

Nápověda ti může být ze začátku **méně sedět**. Jednak kvůli angličtině, jednak je poměrně podrobná. Vůbec se toho neboj. Teprve časem ti začne dokumentace dávat lépe smysl. Pro teď ti stačí tyto materiály.

round

Tato funkce umí zaokrohlit desetinné čísla. První zadanou hodnotou je desetinné číslo, druhou zadanou hodnotou je desetinný řád:

```
>>> round(0.33333, 2)
0.33
>>> round(0.987654, 4)
0.9877
>>> round(2.675, 2)
2.67
```

V posledním příkladu si opět můžete všimnout výsledku způsobeného problematickou *plovoucí rádovou čárkou*.

abs

Funkce `abs` vrací absolutní hodnotu zadánoho čísla:

```
>>> abs(-1)
1
>>> abs(-1.1234)
1.1234
```

int

Funkce `int` umí převádět **některé** datové typy na celá čísla:

```
>>> int(1.11)
1
>>> int("11")
11
```

float

Funkce `float` umí převádět **některé** datové typy na desetinná čísla:

```
>>> float(12)
12.0
>>> float("1.12")
1.12
```

str

Funkce `str` umí převádět **některé** datové typy na string:

```
>>> str(12)
"12"
>>> str(1.12)
"1.12"
```

list

Funkce `list` umí vytvořit (také převést) některé datové typy na list. Pokud jí zadáte např. `string`, sama si jej rozdělí (po indexu) na hodnoty v seznamu:

```
>>> list("abc")
['a', 'b', 'c']
>>> list()
[]
>>> muj_tupl = ("a", "b")
>>> list(muj_tupl)
['a', 'b']
```

tuple

Funkce `tuple` pracuje velice podobně jako funkce `list`, ale jejím výsledkem bude vždycky `tuple`:

```
>>> tuple("abc")
('a', 'b', 'c')
>>> tuple()
()
>>> muj_seznam = ["a", "b"]
>>> list(muj_seznam)
('a', 'b')
```

print

Funkce `print` je veliký pomocník, kterého budeme používat velice často. Slouží k tomu, abychom byli schopni vypsat si hodnoty v proměnných a stringové texty.

```
>>> print(12)
12
>>> print("Ahoj, tady Matous")
Ahoj, tady Matous
>>> desetinne_cislo = 2.718
>>> print(desetinne_cislo)
```

```
>>> print(desetinne_cislo)
2.718
```

Dokonce můžeš kombinovat vypisování několika údajů (různých datových typů):

```
>>> jmeno = "Matous"
>>> vek = 99
>>> print("Jmenuji se", jmeno, ". Je mi", vek, "let.")
```

input

Pokud budeš potřebovat zadat do svého zápisu nějakou hodnotu, funkce `input` ti může pomoci. Nejen, že ti umožní zapsat hodnotu, ale současně si ji můžeš schovat do proměnné.

Jenom myslí na to, že ať už napišeš čísla nebo písmena, funkce `input` z nich ve výsledku udělá `string`, takže je pak musíš ručně převést:

```
>>> input()
Ted pisu      # bude ocekavat zapis
'Ted pisu'
>>> jmeno = input("Moje jmeno:")
Moje jmeno: Matous
>>> print(jmeno)
Matous
>>> cislo = input("Zadej cislo:")
Zadej cislo: 111
>>> type(cislo)
<class 'str'>
```

len

Jakmile do funkce `len` zapíšeš nějaký sekvenční datový typ (i string), vrátí ti jeho délku:

```
>>> len("abc")
3
>>> len([])
0
>>> len(list("abcd"))
4
```

min

Funkce `min` vrací nejmenší údaj ze dvou a více hodnot (případně sekvence). Pokud se nejmenší údaj objeví vícekrát, funkce vrátí první, který najde:

```
>>> min(1, 2, 3)
1
```

max

Funkce `max` vrací největší údaj ze dvou a více hodnot (případně sekvence). Pokud se největší údaj objeví vícekrát, funkce vrátí první, který najde:

```
>>> max(11, 12, 13)
```

sum

Funkce `sum` vrátí součet všech číselných hodnot v sekvenci:

```
>>> cela_cisla = (1, 2, 3, 4)
>>> sum(cela_cisla)
10
```

pow

Tato funkce `pow` potřebuje dvě vstupní hodnoty. První hodnota je číselný údaj, druhá hodnota a hodnota exponentu:

```
>>> pow(2, 2)    # hodnota dva na druhou
4
>>> pow(100, 2) # hodnota sto na druhou
10000
```

Kvíz

PYTHON #1: ÚVOD DO PROGRAMOVÁNÍ / ÚVOD DO PROGRAMOVÁNÍ V PYTHONU / ZABUDOVARÉ FUNKCE / KVÍZ

1/3

Jak se nazývá **built-in funkce**, která vypíše informaci na obrazovku?

A. `vypis`

B. `draw`

C. `write`

D. `print`

OPAKOVÁNÍ

Úkol 1: Převaděč jednotek

SPUSTIT ÚKOL

Už nebudeš muset v hlavě kalkulovat převody imperiálních jednotek na metrické. Napiš si na to program!

⌚ 15 min. • ⚡ lehké

Úkol 2: Nakupujeme auto

SPUSTIT ÚKOL

Úkol 3: **Slicing string**

SPUSTIT ÚKOL

Vytvoř program, který bude využívat string slicing k extrahování specifických částí slova "indexování".

⌚ 10 min. ●●● těžší

Úkol 4: **Spojování, opakování, zjištění délky stringů**

SPUSTIT ÚKOL

V tomto úkolu si procvičíš operace na datovém typu string - spojování, opakování a zjištění délky.

⌚ 3 min. ●●● těžší

DALŠÍ LEKCE

>_ Terminál

