Python s KISKem

Tento interaktivní workbook slouží jako berličky pro tento víkend plný Pythonu. Každý máte vlastní kopii a lze si do ní dělat poznámky. Do workbooku se zapisuje pomocí Markdownu. Základní ovládání si ukážeme. Stejné věci platí i u druhého workbooku s cvičeními.

Cvičení jsou ryze dobrovolná, ale doporučuji si je vypracovat všechna. Pokud nestiháte, nic se neděje, můžete si je dopracovat doma. Po skončení vám nasdílím i workbook se všemi vypracovanými věcmi, ale berte to jako inspiraci nebo berličky, když už opravdu nebudete vědět. Opisováním přijdete o veškerou srandu i strasti při řešení.

Nebojte se googlit, když se zaseknete. Já to dělám v práci dnes a denně. Programátor, který říká, že negooglí, lže. Všichni to děláme, i na banální věci. 😉

Pamatujte, žádná otázka není hloupá, pouze ta, která není položena.

Obsah

Užitečné odkazy

Dokumentace Python | https://docs.python.org/

Pokročilé používání argumentů u funkcí | https://realpython.com/python-kwargs-and-args/

Python s W3Schools (mají i tutoriál na Pandas, Numpy, MatPlotLib) | https://www.w3schools.com/python/default.asp

Předmět CS50P na Harvardu (videa z roku 2022) | https://www.youtube.com/c/cs50/videos

Google | https://www.google.com/search?q=how+to+python

StackOverflow | https://stackoverflow.com/

RealPython | https://realpython.com/

Pandas User Guide | https://pandas.pydata.org/docs/user_guide/index.html

Numpy Docs | https://numpy.org/doc/stable/

Co je Python?

Python je interpretovaný programovací jazyk. Interpretovaný jazyk zjednodušeně znamená, že řádky jsou zpracovávány jeden po druhém. Toto se liší od jazyků kompilovaných, kde se programy zpracovávají celé najednou (například C, C++). Současná verze Pythonu je už tak schopná, že se v Pythonu dá naprogramovat téměř všechno.

Základní užití Pythonu:

- data science
- webový vývoj
- systémové skriptování
- matematické výpočty

Další využití Pythonu

- hry
- vývoj softwaru, aj.

Proč Python

Jeden z důvodů je, že Python je používaný téměř všude, v menší či větší míře. Dále také proto, že jeho syntaxe, ke které se dostaneme později, je velmi podobná angličtině. Jazyk jako takový je velmi flexibilní a umožňuje programování v různých paradigmatech, ale tím my se zabývat nebudeme. Další velkou výhodou je, že Python běží na všech platformách.

Google Colaboratory a Replit

S Pythonem je možné pracovat různými způsoby. Je možné psát kód přímo v příkazové řádce pomocí interaktivní konzole. Také můžeme používat komplexní vývojová prostředí jako PyCharm, Visual Studio Code aj., ale díky tomu, že je jazyk interpretovaný, existují nástroje jako Jupyter Notebook nebo prostředí, ve kterém se nacházíme právě teď (**Replit**).

Není potřeba nic instalovat a nastavovat, stačí si vytvořit nový sešit (projekt) a můžeme psát po blocích Python kód, který je možné rovnou spustit. Spouštět bloky můžeme jeden po druhém, všechny naráz, nebo jen jeden.

Bloky si umí předávat data mezi sebou, ale pouze od shora dolů. To znamená, že data z bloku 1 můžeme použít v bloku 2 a níže, ale ne naopak.

Základní pojmy

Skript nebo program

Soubor příkazů, které vykonávají nějakou logiku. Například zpracování dat ze souboru, ale také jen jednořádkové příkazy.

Proměnné

Základní stavební kámen programů v Pythonu. Slouží na uložení hodnoty v paměti, se kterou je možné později pracovat.

Datový typ

Určuje, jaká data jsou uložena v proměnné. Základní datové typy v Pythonu jsou

- String – řetězec znaků
- Numeric – celá čísla (int), desetinná čísla (float)
- Boolean – pravda/nepravda – true/false
- Sequence – seznam (list), rozsah (range)

Typů je více, ale toto jsou ty základní, se kterými budeme pracovat většinu času.

I když je Python dynamický ohledně datových typů, zápis proměnných je pro jednotlivé datové typy různý. Viz další spustitelný blok.

Pravidla pro psaní kódu

Při programování existují předepsané způsoby, jak by se měl kód zapisovat, a je dobré je dodržovat. Poté totiž bude kódu rozumět každý, nebo pro to budete mít velký předpoklad. Dodržování těchto pravidel ale pomáhá i vám, protože když přijdete ke kódu po delší době, budete vědět, co jste tím mysleli.

Pro tento Bootcamp budeme používat tato pravidla:

Názvy proměnných, funkcí, parametrů

V Pythonu se používá zápis nazývaný *snake_case*, a to z důvodu, že se vše píše malými písmeny a slova jsou oddělená podtržítkem. Dají se samozřejmě používat i jiné formáty zápisu; záleží, jak se domluví lidé v týmu. Tento Bootcamp ale vychází z pravidel, která se nazývají PEP. Ne doslovně, protože to je šikana i pro samotné vývojáře, ale čerpám z nich základní pravidla.

```
# ☑
promenna = True
promenna_na_vice_slov = True

# ❌
promenna-na-vice-slov = False
promenna na vice slov = False
...
```

Výjimkou jsou proměnné, u kterých se nemá měnit jejich hodnota – nazýváme je konstanty. Ty píšeme velkými písmeny a slova oddělujeme podtržítkem. Google Colab nám bohužel nehlídá, že se takovou proměnnou snažíme někde v programu přepsat. Komplexní vývojářské nástroje toto umožňují.

```
# ☑
KONSTANTA = 3.14

# ❌
Konstanta = 3.14
```

Funkce jsou takové speciální, mají totiž v definici závorky a nějaké proměnné v nich. Tyto proměnné nazýváme parametry, ale pravidly se řídí oboje: jak název funkce, tak názvy parametrů.

```
# ☑
def moje_funkce():
  pass

def moje_funkce_s_parametry(parametr_jedna, parametr_dva):
  pass

# ❌
def moje funkce():
  pass

def moje-funkce():
  pass
```

Odsazování kódu

Toto je základní pravidlo Pythonu, a proto si jej prosím pečlivě zapamatujte. V Pythonu odsazení kódu funguje jako členění bloků kódu a špatné odsazení může způsobit v lepším případě špatnou funkcionalitu, v tom horším a pravděpodobnějším se skript nepustí.

Pro odsazování používáme tabulátor – tolikrát, kolikrát potřebujeme blok odsadit. Dostatečně si to osaháme během praktických příkladů, ale tady je krátký přehled, jak to funguje. Dají se používat i mezery, ale to se uklikáte.

```
# deklarace proměnných
x = 10                                  # bez odsazení
y = 5                                   # bez odsazení

# odsazení při definici funkce
def secti_dve_cisla(cislo1, cislo2):    # bez odsazení
  return cislo1 + cislo2                # jeden tabulátor

# vnořené odsazení
print("Toto bude cyklus")               # bez odsazení

for cislo in range(10):
  if cislo == 1:                        # jeden tabulátor
    print("Hele, to je jednička, jako ty!") # dva tabulátory
  else:                                 # jeden tabulátor
    print(cislo)                        # dva tabulátory

print("Tady už cyklus skončil")         # bez odsazení
```

Angličtina v kódu

Prozatím jsem pro prezentaci základů používal češtinu, ale od teď všechny názvy budeme psát a pojmenovávat v angličtině. Procvičíme si slovíčka, ale při zápisu některých bloků kódu taky uvidíme jednoduchou syntax, která vychází z angličtiny.

Další blok kódu je spustitelný, abyste viděli, jak vypadá zápis programu, který vypíše čísla od 1 do 10. Jen upozornění: range funguje jako interval od-po.

> `range(1, 10)` => *v rozsahu od 1 (včetně) po 10 (mimo 10). K indexům se dostaneme později.* 😉

```
for number in range(1,10):
  print(number)

# Lze přeložit jako:
# For number in range from 1 to 10 excluding, print the number.
```

Proměnné

Jak už bylo napsáno, proměnná je místo v paměti, kam si uložíme hodnotu. Představme si to jako kousek papírku, na který si napíšeme poznámku, a tu si vložíme do šuplíku ve stole.

Proměnná musí být nějak pojmenovaná a vzpomeňte si, jak by pojmenování mělo vypadat. V Pythonu je potřeba při založení (dále inicializaci) proměnné přiřadit hodnotu. Do proměnné se dá uložit jakýkoliv datový typ, deklarace funkce je svým způsobem také inicializace proměnné.

Pozor ale na datové typy. Při inicializaci musíme znát, jak se který typ zapisuje. Pokud se podíváte kousek výš, kde byly ukázány datové typy, tak je vidět, jak taková inicializace probíhá.

Důležité je zmínit, že každý jazyk, nejen Python, má svoje rezervovaná slova, která by se pro názvy proměnných neměla používat. Například: `int`, `float`, `set`, `def`. Těmto výrazům se říká klíčová slova `keywords` a při zvýrazňění syntaxe jsou vždy označeny jinou barvou než základní barva kódu (pro nás bílá).

Vypisování hodnot v Pythonu

Určitě jste postřehli používání příkazu `print( něco uvnitř )`

Pomocí příkazu `print()` můžeme vypsat hodnotu proměnné, logické nebo aritmetické operace, anebo výsledek funkce s návratovou hodnotou (k tomu se dostaneme).

Ze začátku pomocí příkazu `print()` budeme vypisovat pouze řetězce nebo hodnoty proměnných. Používání `print()` je jednoduché:

Pokud chceme vypsat jen nějaký informativní řetězec, nemusíme jej ukládat do samostatné proměnné, prostě jej vepíšeme jako parametr do příkazu `print()`.

```
print("toto je nějaká věta, která bude vypsána z Python kódu")
```

Pokud budeme chtít vypsat hodnotu proměnné, tak místo řetězce do parametru předáme danou proměnnou. Předání znamená, že tam napíšeme její název a Python se postará o zbytek.

```
my_variable = 12

# předám svoji proměnnou do funkce print jako parametr
print(my_variable)

# ve výstupu programu bude: 12
```

Také můžeme vypsat více proměnných zároveň.

```
a = 5
b = 6
c = 8

# jako parametry uvedeme jednotlivé proměnné a oddělíme je čárkou
print(a, b, c)

# ve výstupu programu bude: 5 6 8
# Pozor, ve výpisu nebudou čárky!
```

Formátované výpisy

Naprosto úžasná věc Pythonu je možnost výpis formátovat. Je to jednoduché a účinné. Dejme tomu, že máme tři proměnné:

- name – Josef
- age – 30
- nickname – Chosé

A chceme z těchto tří proměnných poskládat větu.

> Ahoj, jmenuji se Josef, je mi 30 let a kamarádi mi dali přezdívku Chosé.

Ve formátovaném stringu je možné dělat jednořádkové operace (např. aritmetické) nebo můžeme dokonce zavolat funkci, která vrací hodnotu, a tu rovnou vepsat do stringu.

```
name = "Josef"
age = 30
nickname = "Chosé"

# Můžeme použít prastarý zápis, ale můžeme takto spojovat pouze řetězce,
# a tak musíme všechy číselné hodnoty převést na řetězec atd.
print("Ahoj, jmenuji se " + name + ", je mi " + str(age) +
      " let a kamarádi mi dali přezdívku " + nickname)

# Nebo pomocí formátovaného zápisu
print(f"Ahoj, jmenuji se {name}, je mi {age} let a kamarádi mi dali přezdívku
{nickname}")

# Jen pro info, funguje to i pro víceřádkové řetězce, ke kterým se dostaneme
print(f'''
  Ahoj, jmenuji se {name},
  je mi {age} let,
  a kamarádi mi dali přezdívku {nickname}
''')
```

```
# Ve formátovaném zápisu můžeme volat i funkce
def add(a, b):
  return a + b

print(f"součet čísel 2 a 5 je: {7 + 5}")
print("součet čísel 2 a 5 je: " + str(7 + 5))
```

Funkce

Funkce je blok kódu, který můžeme použít na více místech. Slučuje nám logiku několika příkazů do jednoho. Předtím, než můžeme vlastní fukci použít, musíme ji nadefinovat. K tomu slouží klíčové slovo `def`, následuje název funkce, pak kulaté závorky a mezi nimi můžou a nemusí být parametry. Jeden, dva nebo deset. Záleží, kolik jich pro danou funkci potřebujeme. A nakonec dvojtečka. Na novém řádku a s patřičným odsazením začínáme definovat tělo funkce.

```
# Definice funkce s adekvátním názvem
def sum_and_print(x, y):
  print(x + y)

# Volání funkce
sum_and_print(2, 5)
```

Zde jsme si nadefinovali funkci `sum_and_print`, která vypíše výsledek pomocí příkazu `print`. Když s výsledkem funkce chceme pracovat dál, využijeme ve funkci klíčové slovo `return`. Tomuto se říká návratová hodnota.

```
# Definice funkce s návratovou hodnotou
def sum(x, y):
  return x + y

# Následně si můžeme hodnotu, kterou nám fuknce vrátí, uložit do proměnné result

result = sum(2, 5)

print(result)
```

Je to o něco delší, ale můžeme pak výsledek používat na více místech a nemusíme funkci volat pokaždé, když ten výsledek potřebujeme. V případě, že z funkce pouze **vypisujeme** hodnotu, nemůžeme tuto hodnotu dále v programu používat. Takové funkce se snažte definovat co nejméně.

Lambda

Lambda nebo také jednořádková funkce. Je to speciální zápis anonymní fuknce, které se nějak nejmenuje. Má vždy jeden parametr, pro který definujeme, co se s ním má stát.

Použití je například v Pandas u metody `assign`, která nám vytvoří automaticky počítanou hodnotu pro daný sloupec.

```
# syntaxe zápisu lambda funkce
lambda parametr: co se má stát s parametrem
```

Keyword argumenty ve funkcích

Při definici funkce můžeme definovat tzv. `keyword argumenty`. Z pravidla by to měly být vždy poslední argumenty v definici funkce. K čemu slouží? Funkce může mít mnoho parametrů a abychom je vždy nemusely zadávat všechny, tak si je pojmenujeme (`keyword`). Když poté chceme funkci zavolat, musíme paraterm explicitně vyjmenovat a nastavit mu hodnotu.

```
# pseudo funkce, takto se nekreslí
def draw_triangle(a, b, c, isfilled=False, pattern=None):
  pass

# Můžeme funkci zavolat bez specifikování keyword argumentu protože už má
defaultní hodnotu
draw_triangle(1, 2, 3) # vykreslí jen čáry

# Defualtní hodnotu ale můžeme zaměnit za aktuální, když při volání funkce
# použijeme tento keyword argument a hodnotu mu nastavíme
draw_triangle(1, 2, 3, isfilled=True, pattern="stripes") # vykreslí  trojúhelník,
který bude vyšrafovaný
```

Platnost proměnných aneb `scope`

Abychom v kódu měli pořádek, mají některé proměnné pouze omezenou platnost. V programování se tomu říká `scope`. Pokud například v naší fukci nadefinujeme proměnnou jméno, tak s ní můžeme pracovat pouze v dané funkci. Mimo ni nám to fungovat nebude.

```
def greet()
  name = "Patrik"
  print(name)

# pokus použít proměnnou z funkce mimo ni
print(name)    # NameError: name 'name' is not defined
```

To stejné platí i pro cykly, podmínky a jiné. **Pozor**, v podmínkách toto neplatí. Napříč `if` a `else` částmi je stejný scope.

```
name = "Jan"

if name == "Patrik":
  print(name)
  surname = "Procházka"
else:
  print(name)
  print(surname)  # vyhodí error, že proměnná neexistuje.
```

To znamená, že můžeme v různých úrovních používat stejné pojmenovaní proměnných.

```
name = "Patrik"

def greet():
  name = "Jan"
  print(f"Jméno ve funkci je {name}")

# zavoláme funkci greet, která incializuje vlastní proměnnou name, ale nezmění tu
v globálním scopu
greet()

# Můžeme vidět, že proměnná name v globálním scopu se nezměnila
print(f"Jméno mimo funkci je {name}")

# Výstup v konzoli bude:
# Jméno ve funkci je Jan
# Jméno mimo funkci je Patrik
```

Vstup uživatele do programu

Protože Python funguje v konzoli, je možné naprogramovat interakci s uživatelem. Oproti jiným jazykům je toto v Pythonu opravdu jednoduché. Python nabízí funkci `input()`, která zastaví program a čeká na vstup uživatele, který musí být potvrzen entrem. Funkce `input` nám vrací to, co zadal uživatel, a když si to uložíme do proměnné, můžeme s tím nadále v našem programu pracovat.

Funkci `input` můžeme zadat řetězec znaků jako parametr a tento řetězec se poté ukáže na řádku, který čeká na vstup od uživatele. Můžeme tak uživatele informovat o tom, co od něj očekáváme.

```
name = input("Jak se jmenuješ?: ")

print(f"Uživatel se údajně jmenuje: {name}")
```

Datové typy

Datových typů jsme se již dotkli, ale teď se na ně podíváme podrobněji, protože v práci s daty jsou nedílnou součástí. Například při používání knihovny `pandas` můžete ušetřit spoustu paměti a času, pokud pro dataset, se kterým pracujete, určíte typy pro jednotlivé sloupce.

Datové typy dělíme na primitivní a komplexní. Komplexní datový typ je většinou složen z více primitivních anebo jeho obsah je více hodnot stejného i rozdílného primitivního typu.

Primitivní datové typy

- **string** – řetězec znaků, při deklaraci hodnoty používáme `""`, můžeme použít i `''`, ale standard jsou dvojité uvozovky
- **int** – celočíselná hodnota jako například `12`
- **float** – desetinné číslo `1.2` – POZOR: desetinná čísla píšeme VŽDY s tečkou
- **boolean** – logická hodnota (v informatice bráno jako 1 nebo 0) – v Pythonu `True` nebo `False`

```
string = "Řetězec znaků"
integer = 12
decimal = 1.2
boolean = True
```

Komplexní datové typy

Python obsahuje minimálně 9 komplexních datových typů. Ukážeme prozatím jen dva, u dalších jen zmíním jejich význam a některé jsou vynechané úplně.

- **list** – seznam různých hodnot, které se mohou lišit datovým typem. Mixování vícero datových typů v jednom listu je ale tzv. anti-pattern. Není doporučeno to používat, program je pak nepředvídatelný.
- **range** – speciální druh listu hodnot, které nám generuje Python v zadaném rozmezí. Je to trochu matoucí, protože `range()` je zároveň funkce, ale i datový typ.
- **set** – další speciální druh listu, který ale obsahuje pouze unikátní hodnoty. Set lze vytvořit z listu, k tomu se ale dostaneme.
- **dict** – slovník, v informatice též jako dictionary, je datová struktura, která data ukládá ve formátu `klíč-hodnota`, v informatice jako `key-value pair`.
- **tuple** – také list, ale je `immutable`. To znamená, že po jeho inicializaci nemůžeme měnit hodnoty uvnitř.

```
my_list = [1, 1, 2, 2, 3, 3]
my_range = range(1, 10)
my_set = {1, 2, 3}
my_dict = { "name": "Patrik", "age": 30 }
my_tuple = (1, 1, 2, 2, 3, 3)
```

Mezitypové převádění

Mezitypové převádění je často používané, protože občas můžeme mít číslo zapsané jako `string`. Už u logických hodnot jsme si ukázali, jak můžeme jiné datové typy převést na typ `boolean`, a to pomocí funkce `bool()`, kdy jako parametr předáme hodnotu nebo proměnnou, kterou chceme na daný datový typ převést.

Na mezitypové převody existují tyto funkce:

- `int()` – pro převod na celé číslo
- `float()` – pro převod na desetinné číslo
- `str()` – pro převod na řetězec
- `bool()` – pro převod na logickou hodnotu

```
string = "Ahoj"
string_number = "10"
integer = 12
decimal = 1.2

# převod čísel na string je jednoduchý, jde to vždy
print(str(integer))
print(str(decimal))

# při převodu stringu na číslo ale musíme vědět, zda string obsahuje hodnotu,
kterou můžeme na číslo převést.
print(int(string_number))
print(float(string_number))

# error
print(int(string))
```

String aka řetězec znaků

V Pythonu je řetězec považován za primitivní datový typ, ale je tomu opravdu tak? Python umožňuje s řetezci dělat psí kusy. Můžeme například řetězce sčítat, chcete-li spojovat. To ale není všechno, my v Pythonu můžeme řetězce i násobit 😅. Dále pomocí cyklu můžeme vypsat jednotlivé znaky v řetězci.

Důležitý je tu pojem znak – jedno písmeno – protože string v Pythonu není nic jiného než seznam jednotlivých znaků. To znamená, že řetězec můžeme vnímat i jako `list`.

```
# Harakiri s řetězci

          # ["A", "h", "o", "j"]
my_string = "Ahoj"

# Výpis jednotlivých znaků z řetezce 🏏
for letter in my_string:
  print(letter)

my_second_string = "světe"

# Sčítání řetězců? Není problém 👶
print(my_string + my_second_string)

# Násobení řetězců 🚀
print("Čus " * 3)
```

Víceřádkové řetězce

Někdy je potřeba do proměnné uložit delší řetězec, který je třeba vizuálně oddělený prázdným řádkem. V Pythonu není problém, jen musíme při deklaraci používat dvojité uvozovky troje místo jedněch. Poté co v Python kódu zadáme do těchto trojitých uvozovek bude i na výstupu.

```
long_and_formatted_string = '''Ahoj světe,

snažím se učit Python a myslím, že mi to jde.

Díky!'''

# Formátování je stejné, jako jsme to napsali při deklaraci.
print(long_and_formatted_string)
```

Zjištění délky řetězce

Dále se velmi často může hodit délka řetezce. Jak už asi tušíte, v Pythonu to není žádný problém. Máme na to pomocnou funkci `len()`, která nám řekne, jak je daný řetězec dlouhý. A nejen řetězec, `len()` lze použít třeba i na list, a že si toho užijeme.

```
my_string = "Ahoj!"
print(len(my_string))     # 5
```

Ověření, zda je znak nebo jiný řetězec obsažen v jiném

Další operace, kterou lze na řetězci dělat je "vyhledávání v nich". V uvozovkách je kvůli nadsázce, není to totiž jako vyhledávání na Googlu. Tuto operaci lze dělat v podmínkách, ke kterým se pomale dostáváme, ale také i ve funkci `print()` na které si to ukážeme.

Tuto operaci můžeme i negovat pomoci klíčového slova `not`

```
substring = "Ahoj"

long_string = "Ahoj světe, učím se Python"
another_string = "Zatím to jde ok..."

print(substring in long_string)        # True
print(substring in another_string)     # False

# Můžeme operaci znegovat pomocí klíčového slova `not`
print(substring not in another_string)  # True
```

Numerické datové typy

Opakování je matka moudrosti, a proto znovu opakuji, že máme tři numerické datové typy v Pythonu. My budeme používat pouze dva, a to:

`int` – celočíselné hodnoty

`float` – desetinná čísla

Čísla jsou ve většině programovacích jazycích jednoduché a přímočaré. V Pythonu můžeme převádět mezi typy `int` a `float`, jak se nám zlíbí (ukážeme si později). Je však potřeba dávat pozor na to, že když desetinné číslo převedeme na celé číslo, nefunguje tam žádné zaokrouhlování a hodnota za desetinnou tečkou se odmaže.

```
my_decimal = 12.67

print(int(my_decimal))          # 12
```

Logická hodnota – Boolean

Další jednoduchý datový typ a tentokrát napříč programovacími jazyky. Výsledky logických operací jsou typu `boolean` a logické výrazy vždy končí ve stavu `True` nebo `False`.

Menší trik, který se používá s hodnotami boolean v programovacích jazycích, je vyhodnocování, jestli proměnná primitivního typu obsahuje hodnotu. Dost často se toto používá, pokud potřebujete pracovat s proměnnou, ve které je nějaká hodnota. K praktickému používání se dostaneme u podmínek.

```
print(bool("Ahoj"))          # True
print(bool(1))               # True

# Zatímco
print(bool(""))              # False
print(bool(0))               # False

# A platí to i u komplexních datových typů
print(bool(()))              # False
print(bool([]))              # False
print(bool({}))              # False
```

Aritmetické operace

V Python kódu je možné počítat jak jednoduché operace pomocí příkazů, tak i složitější matematické úlohy většinou za pomoci knihoven třetích stran.

My si ukážeme ty jednoduché. Mezi ně řadíme:

- sčítání (+)
- odčítání (-)
- násobení (*)
- dělení (/)
- modulo (zbytek po celočíselném dělení) (%)
- mocniny (**)
- celočíselné dělení (//)

```
a = 10
b = 4

print(a+b)    # 14
print(a-b)    # 6
print(a*b)    # 40
print(a/b)    # 2.5
print(a%b)    # 2
print(a**b)   # 10000
print(a//b)   # 2
```

Pěkně přímočaré, není potřeba nic navíc. Pokud chceme například do proměnné x, která má hodnotu 5, při inicializaci příčíst hodnotu proměnné y, tak to uděláme takto:

```
x = 5
y = 4

x = x + y        # po tomto příkazu bude hodnota uložená v x 9
```

Můžeme vidět, že zápis tak jednoduché věci je docela zdlouhavý, a proto existují i zkrácené zapisy pro aritmetické operace.

`x = x + y` lze zapsat i jako `x += y`. Platí to pro všechny operace, které jsme si teď ukázali.

Je také možné si všimnout, že výsledek dělení je vždy typu decimal (float).

Logické operace a jiné operátory

Programování jako takové je nabité logikou, a proto snad v každém programovacím jazyku existují logické operátory. Logické operátory mají využití v podmínkách, které nás čekají níže.

Nebudeme se zaobírat správnými pojmenováními, abychom se z toho nezbláznili.

Základními operátory jsou větší, menší, rovná se, a jejich kombinace. Pokud takovou operaci napíšeme v kódu, výsledkem takové operace je vždy hodnota `boolean`, tedy `True` nebo `False`. Pokud z této operace dostanete něco jiného, tak je někde chyba.

```
x = 5
y = 4

print(x < y)       # False
print(x > y)       # True
print(x == y)      # False
print(x <= y)      # False
print(x >= y)      # True
print(x != y)      # True
```

Logické operace

Logické operace jsou věda sama o sobě. Pamatujete si výrokovou logiku?

V Pythonu se nachází 3 logické operátory

- logické "a" (`and`)
- logické "nebo" (`or`)
- logická negace (`not`)

Zapomoci logických operací můžeme řetězit podmínky v našem programu. V matematice jsme například určovali hodnotu čísla v podmínách nějak takto: `0 < x < 10` toto ale nemůže v pythonu napsat, protože by se nám program nesputil a vyhodil nám chybu. Jsou to dvě podmínky v jednom. Když chceme takovou podmínku napsat v Pythonu, musíme obě podmínky spojit logickým "ano" `and`.

Když si podmínku `0 < x < 10` můžeme rozložit na dvě podmínky:

- `x` musí být větší než `0`
- A ZÁROVEŇ je `x` menší jak `10`

Obě tyto podmínky už umíme zapsat v Pythonu a jen je spojíme logickým "a" `and`.

```
x = 9

print( x > 0 and x < 10 )    # True
```

Pokud podmínky řetězíme, tak je potřeba brát v potaz tyto pravidla:

> Abychom dostali `True` za pomoci `and` , tak musí být všechny podmínky spojené logickými operátory `True`. `True and True and True` je výsledně `True`, zatím co `True and False and True` je výsledně `False`

> Při používání logické operace `or` nám stačí pro výsledné `True`, aby byla alespoň jedna podmínka ze všech s výsledkem `True`. `True or False or False` je výsledně `True`, zatím co `False or False or False` je výsledně `False`.

Operátory `is` a `in`

Tyto dva operátory s sebou přináší trochu zamotání všeho, co jsme se doteď naučili. Respektive jeden z nich a to `is`. Operátor `is` porovnává dva objekty, jestli jsou stejné.

```
x = 5
y = 5
print(x is y)    # True

i = [1, 2, 3]
j = [1, 2, 3]
print(i is j)    # False ?!

a = "ahoj"
b = "ahoj"
print(a is b)    # True ?!
```

Jednoduché datové typy lze porovnávat pomoci operátu `is` ale **NEDOPORUČUJI TO!** Je to proti konvencím a operátor `is` je velmi specifický a jeho použití také. U komplexních datových typů se porovnává místo v paměti ne hodnoty jako takové.

Zatím co operátor `in` se vám bude hodit a používá se v Pythonu často. Lze s jaho pomocí vyhledávat v řetězcích i komplexních datových typek.

```
x = {1,2,3}     # Set
print(1 in x)   # True

x = [1,2,3]     # List
print(1 in x)   # True

x = (1,2,3)     # Tuple
print(1 in x)   # True
```

Operátor `in` se dá používat i pro práci se slovníky (`dict`), ale k tomu možná později.

Operátory `is` nebo `in` lze kombinovat a otáčet podmínku pomoci `not`.

- není - `is not`
- není v - `not in`

Krásně to reflektuje angličtinu, ale dělá binec v syntaxi programovacího jazyka 🙃

Podmínky

Podmínky jsou nedílnou součástí programování. Slouží k větvení programů podle zadaných podmínek, které jsme si ukazovali. Podmínku definujeme pomocí klíčového slova `if` a následuje podmínka, která musí ve výsledku dát hodnotu `boolean`. V jádru jsou podmínky velmi jednoduché, jako když položíte otázku, na kterou se odpovídá ano či ne.

Po podmínce následuje dvojtečka, stejně jako u definice funkce. Prvně píšeme blok kódu, který se vykoná, když je podmínka vyhodnocena pozitivně. Může, ale nemusí následovat blok `else`, který zachytí ostatní případy vyhodnocení podmínky. A opět blok kódu, který se má v tomto případě vykonat. **POZOR!** Bloky kódu v obou větvích podmínky musí být řádně odsazeny, jinak se nám Python kód nespustí!

```
i_love_kisk = True

# Definice podmínky a "otázky", na kterou se dá odpovědět ano či ne
if i_love_kisk == True:
  # Blok kódu, pokud je odpověď kladná
  print("Tak se vidíme na KISK párty!")
else:
  # Klíčové slovo else se vykoná ve všech ostatních případech
  print("Přijď na KISK párty a změníš názor 🚀")
```

Více podmínek aneb `if...else if...else`

Python nám také umožňuje spojit více podmínke dohromady, když potřebuje vyhodnotit více stavů. Za první `if` blok přidáme blok s další podmínkou za pomoci klíčového slova `elif`. Toto je jedna z mála věcí, která je na Python syntaxi divná, většina programovacích jazyků má klauzuli `else if`, ale Python chtěl být speciální 🙃.

```
my_age = 18

if my_age < 18:
  print("Alkohol by mi prodat neměli")
elif my_age == 18:
  print("Je to těsný, ale můžu si v klidu kupovat")
else:
  print("Naprostá pohodička")
```

Když skládáte více podmínek, musíte si dávat pozor na to, jak je skládáte. Protože se můžete i u tak jednoduchého příkladu dostat do bodu, že nebudete mít ošetřeny všechny stavy.

```
brand = "Peugeot"
year = 2000

if brand == "Peugeot":
  print("Francouz... ale aspoň je pohodlnej")
# Tato podmínka nikdy nenastane, protože všechny Peugoty zachytí podmínka první.
elif brand == "Peugeot" and year < 2000:
  print("Kámo, čas na změnu")
elif year < 2012 and brand is not "Peugeot":
  print("Hmmm...")
else:
  print("Užívej moderní káru")
```

Říka se tomu problém největší specifity. To znamená, že podmínka, která je nejvíce specifická by měla být první, respektive měly by podmínky být poskládány od nejsložitějších po nejjednodušší.

Ternární operátory

Ternární operátor je zkrácený zápis podmínek na jeden řádek (většinou). Toto zase musím v Pythonu pochválit. Ze syntaktického hlediska je to zase jako věta v angličtině. Vezměme v potaz úvodní příklad ohledně náklonosti ke KISKu.

```
i_love_kisk = True

if i_love_kist == True:
  print("Tak se vidíme na KISK párty!")
else:
  print("Přijď na KISK párty a změníš názor 🚀")
```

Takovou jednoduchou podmínku, která má pouze `if else` skladbu můžeme zapsat i pomoci ternárního operátoru.

```
print("Tak se vidíme na KISK párty!") if i_love_kist == True else print("Přijď na
KISK párty a změníš názor 🚀")
```

Možná je to trochu nepřehledná, ale v podstatě je to poskládané takto:

> `Příkaz, co se má provést, POKUD je tento výrok pravdivý NEBO se provede tento příkaz`

Cykly

Cyklus nám může posloužit, když potřebujeme něco udělat několikrát. Máme dva druhy cyklu – `for` a `while`. Mějme na paměti, že cyklus se nemusí vždy opakovat tolikrát, kolikrát bylo zamýšleno. Můžeme v cyklu mít podmínku, která nám například najde nějaký prvek, a poté zavolat příkaz `break`, který cyklus ukončí, a následuje další příkaz po cyklu.

```
fruits = ["apple", "kiwi", "banana"]

for fruit in fruits:
  print(fruit)

print("Konec cyklu")

for x in range(2, 6):
  print(x)
```

Cyklus `for`

Tento cyklus slouží k iterování v daném rozmezí opakování. Můžeme použít range, nebo pomoci cyklu `for` procházet datové struktury jako `list`, `tuple`, nebo `set`.

Syntaxe definice cyklu `for` je zase podobná angličtině. Například pokud chceme iterovat nad listem, tak definici v anglické větě víceméně přepíšeme do Pythonu.

```
for fruit in fruit_basket do something
```

```
fruit_basket = ["apple", "orange", "banana", "watermelon"]

for fruit in fruit_basket:
  print(fruit)
```

Za klíčovým slovem `for` definujeme proměnnou, která se bude používat v bloku kódu, který se vykonává při každé iteraci cyklu. Můžeme si ji pojmenovat jak chceme, ale mějte na paměti, že pojmenováváme proměnné smysluplně.

Pomoci cyklu můžeme i procházet řetězce písmeno po písmenu.

```
my_string = "Patrik"

for char in my_string:
  print(char)
```

Specialita cyklu `for` je, že ho můžeme vnořovat do sebe. Můžeme tedy mít cyklus v cyklu.

```
adjectives = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

# Pro jednu interakci vnějšího cyklu musí být ukončeno vše uvnitř cyklu
for adj in adjectives:
  for fruit in fruits:
    print(adj, fruit)
```

Stejně jako u fukncí a podmínek, ani cykly nemohou být inicializovány s prázdným tělem. Opět ale můžeme využít klíčové slovo `pass`.

```
for i in range(1, 10):
  pass
```

Cyklus `while`

Tento cyklus má v inicializaci podmínku, která vyhodnocuje, zda se má kód v těle cyklu ještě provádět. Většinu případů dokáže, kdy potřebuje cyklus, dokáže zastát cyklus `for`, ale někdy je z logiky věci lepší použít cyklus `while`.

```
fruits = ["apple", "banana", "kiwi", "watermelon"]

index = 0

while index < len(fruits):
  if fruits[i] == "kiwi":
    print("Aaaaa, kyselý!!")
    break
  else:
    print("Mňam")

  index += 1

print("Mám rád sladké, no")
```

Pokud u cyklu while používáme v incializační podmínce index, musíme ho vždy ručně modifikovat, jinak by se nám mohlo stát, že se nám spustí nekonečná smyčka.

`while` cyklus je dobrý například pokud potřebujeme souvisle zpracovávat vstup od uživatele a pouze některé interakce by měly program ukončit.

```
# Takový programátorský hack, kdy udržet while cyklus bežet donekonečna
while True:
  # načteme vstup uživatele
  user_input = input("Tak co?: ")

  # vyhodnotíme, zda uživatel nezadal nějaké exit slovo
  if str.lower(user_input) in ["konec", "q", "pryč"]:
    print("Tak čau!!")
    # ukončíme while cyklus
    break

  # Provokujeme dál...
  print(f"{user_input}... Aha, a dál?")
```

Specialitou `while` cyklu je možnost za něj navázet `else`, protože při inicializaci cyklu zadáváme podmínku

```
number = 1
while number < 10:
  print(number)
  number += 1
else:
  print(f"{number} už není méně než 10")
```

Datové struktury

nebo také komplexní datové typy. Narozdíl od jednoduchých datových typů jsou ty komplexní uloženy v paměti rozdílně.

Když inicializujeme list pomocí příkazu `x = [1, 2, 3]`, tak se nám do `x` uloží **odkaz do paměti**, kde je uloženo pole. Z toho důvodu jsme dostávali `False`, když jsme zkoušeli porovnávat dvě zdánlivě identická pole pomocí operátoru `is`. Každé pole ukazovalo do jiného kousku paměti, a to nám porovnával operátor `is`.

Na toto je potřeba si dávat pozor, když budeme s datovými strukturami pracovat. S tím, co jsme se zatím naučili, víme, že můžeme z proměnné `a` kopírovat hodnotu do proměnné `b` tím, že napíšeme `a = b`, ale co se stane, když zkusíme to stejné u nějaké datové struktury?

```
a = [1, 2, 3]        # Incializujeme list
b = a                # "Zkopírujeme" list a do proměnné b

a[0] = 5             # Změníme první prvek listu a na číslo 5

print(b)             # Co se nám vypíše?
print(a is b)        # True
```

Vypíše se nám `[5, 2, 3]`. A proč? Protože když jsme zkoušeli zkopírovat list z `a` do `b`, tak jsme proměnné `b` nepředali hodnoty z listu, ale pouze adresu na místo v paměti, kde je uložen list `a`. To znamená, že jsme v tu chvíli měli v proměnných `a` i `b` uloženou stejnou adresu, a proto nám vyjde porovnání `a is b` jako `True`.

S tímto chováním přichází spousta věcí, které si musíme hlídat, když chceme s takovými strukturami pracovat, a proto programovací jazyky nabízí spoustu fukcí, které nám práci s těmito strukturami ulehčí. Představíme si ty základní funkce, které můžeme používat, a k čemu slouží.

Společné znaky datových struktur

Mějme list `[1, 2, 3, 4, 5]`. Už jsme si řekli, že datové struktury nám umožní ukládat více hodnot do jedné proměnné, které spolu mohou, ale nemusí souviset. Ale co když z takového listu chceme přečíst pouze jednu hodnotu, nebo ji změnit? Od toho existuje `index`. Index je ukazatel na pozici v listu. Index nabývá hodnot od 0 do ..., to záleží na velikosti listu, ale z pravidla je to `počet prvků v listu - 1`. Mínus jedna protože začínáme od hodnoty 0, ale délka je `1` pokud je v listu pouze jeden prvek, který se nachází na `indexu` **0**.

> V Pythonu můžeme zadat i zápornou hodnotu indexu a tato funkcionalita se může hodnit -> na indexu `-1` najdeme poslední položku seznamu atd.

`[1, 2, 3, 4, 5, 6]` List

`_0__1__2__3__4__5_` Indexy prvků

V Pythonu i jiných programovacích jazycích se používá k přístupu k hodnotám v listu pomoci hranatých závorek `[]`, které jsou hned za názvem proměnné. Do hranatých závorek uvádíme hodnotu `indexu`, na který chceme přistoupit.

```
my_list = [1, 2, 3, 4, 5, 6]

# ☑
my_list[3]

# ❌
my_list [3]


# NEGATIVNÍ INDEX

# Když chceme přistoupit na poslední položku v seznamu,
# tak máme dvě možnosti:
my_list[len(my_list) - 1]       # 👴 moc dlouhé

my_list[-1]                     # 👍
```

Tento přístup platí i pro `Tuple` a `Dictionary`. Ale aby to nebylo jednoduché, tak u `Dictionary` je to trochu jiné, ale k tomu se dostaneme. **U struktury typu `Set` takto k hodnotám přistupovat nemůžeme!**

Samozřejmě do sebe můžeme jednotlivé jednotlivé listy zanořovat. I u těchto vnořených listů můžeme přistupovat pomoci indexu k jednotlivým prvkům. Začínáme vždy s indexem vnějšího a pokračujeme dál.

```
my_class = [["Pepa", "Jan", "Patrik"], ["Jana", "Anna", "Lenka"]]

# ☑
my_class[0][2]              # Patrik

# ❌
my_class[0,2]               # Error
```

Přistupování k hodnototám v zadanám rozsahu indexu. Stejně jako u `range()` platí, že první číslo platí a druhé určuje hranici, ale už je vynecháno.

```
# indexy   0  1  2  3  4  5  6  7  8
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
# vypíšeme _____[3, 4, 5]____________
my_list[2:5]
```

List

List je často používaná datová struktura a nabízí spoustu metod pro práci s ním.

Přídání prvku do listu

Metoda `append()`

Append je jedna z metod, pomoci které můžeme přidávat prvky do listu. Metoda `append()` má jeden parametr. V tomto parametru předáme objekt, který chceme do listu přidat. Objekt, protože pomoci metody `append` můžeme přidat primitivní hodnotu, ale i další list, či jinou datovou strukturu.

```
# Přidání nového prvku do listu
girls = ["Jana", "Anna"]
girls.append("Lenka")

my_class = [["Pepa", "Jan", "Patrik"]]
my_class.append(girls)

# Přídání dalši ho listu do listu listů 😅
my_class.append([1, 2, 3])

# Výsledný list
[['Pepa', 'Jan', 'Patrik'], ['Jana', 'Anna', 'Lenka'], [1, 2, 3]]
```

Metoda `insert()`

Metoda insert přijímá dva parametry, první parametr je index, na který chceme do listu něco vložit a druhý je objekt, co chceme vložit.

Je možné vkládat i datové struktury, ale pokud třeba chcete spojit dvě pole, bude se hodit další metoda `extend`.

```
# indexy  _0, 1, 2, 3, 4, 5, 6, 7, 8_
my_list = [1, 2, 3, 5, 6, 7, 8, 9, 0]
my_list.insert(3, 4)

# Výsledný list
[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
```

Metoda `extend`

Extend slouží na slučování nejen listů. Umožňuje totiž sloučit list s tuplem nebo list se setem. Používat ji nebudeme, ale hodit se může.

```
numbers = [1, 2, 3, 4, 5]
more_numbers = [6, 7, 8, 9, 0]

numbers.extend(more_numbers)

# Výsledný list
[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
```

Mazání prvků z listu

Metoda `pop` a `remove`

Pro mazání z listů v Pythonu existují dvě metody, a to `pop` a `remove`. Obě metody příjimají jeden parametr, ale chování je rozdílné.

`pop` nám odstraní prvek ze zadaného indexu. Je možné parametr nezadat a tím neurčit index na kterého chceme odstranit položku. Výchozí chování způsobí, že bude smazána poslední položka v listu.

```
numbers = [1, 2, 3, 4]
numbers.pop()

# stav listu
[1, 2, 3]

numbers.pop(1)

# konečný stav listu
[1, 3]
```

Zatím co `remove` odstraní z pole prvek, který má stejnou hodnotu jako parametr. **POZOR!**, odstraňuje **pouze** první prvek, na který narazí.

```
fruits = ["banana", "apple", "kiwi", "watermelon", "banana"]
fruits.remove("banana")

# Výsledný list - všimněte si, že byla odstraněna
# pouze první hodnota, která je shodná s parametrem.
["apple", "kiwi", "watermelon", "banana"]
```

Metoda `clear`

Jak už název napovídá, tak tato metoda smaže všechny prvky v listu.

```
my_list = [1, 2, 3, 4, 5, 6]
my_list.clear()

# Výsledek
[]
```

Další užitečné metody pro list

Metoda `sort`

Seřadí pole vzestupně, pokud chceme sestupně, tak musíme nastavit parametr `reverse` na `True`.

```
my_numbers = [9, 0, 1, 6, 2, 3, 6, 5, 4]
my_numbers.sort()

# Výsledek
[0, 1, 2, 3, 4, 5, 6, 6, 9]

# Řazení sestupně pomoci parametru reverse
my_numbers.sort(reverse=True)

# Výsledek
[9, 6, 6, 5, 4, 3, 2, 1, 0]
```

Metoda `reverse`

Tato metoda jednoduše přehází hodnoty v listu do opačného pořadí.

```
numbers = [1, 2 ,3]
numbers.reverse()

# Výsledek
[3, 2, 1]
```

Metoda `count`

Tato metoda spočítá kolikrát se hodnota zadaná jako parametr objevuje v listu. Tento počet výskytů je jako návratová hodnota a proto ji musíme uložit do nějaké proměnné, jinak o výsledek této funkce přijdeme.

```
fruits = ["banana", "apple", "kiwi", "banana", "watermelon"]

number_of_bananas = fruits.count("banana")

print(number_of_bananas)     #  2
```

Set

Jak už bylo řečeno, tak `set` je datová struktura, která obsahuje pouze unitkátní hodnoty. Práce se setem ale není tak jednoduchá jako s listem. Set také nedispunuje takovým množstvím metod jako list. Na ukázku jich tu máme jen pár. Další zádrhel setu je, že nemůže přistupovat k prvkům jako u listu přes hranaté závorky a index, na který chceme přistoupit. Musíme buď projít celý set pomoci cyklu a nebo použijeme podmínku pro zjištění, zda se prvek, který chceme, nachází v setu.

K čemu je tedy takový set dobrý? Dost často potřebuje získat pouze unikátní hodnoty v datech (například z listu) a než abychom si museli psát složitou fukci, tak si z listu jednoduše vytvoříme `set`.

Set se nedá seřadit ani upravovat nějak napřímo. Pro jeho editaci musíme striktně pracovat jen s metodami.

```
billion_numbers = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5]

distinct_values = set(billion_numbers)

print(distinct_values)

# Unikátní hodnoty všechny pohromadě
{1, 2, 3, 4, 5}

# Výpis hodnot ze setu po jedné
for value in distinct_values:
  print(value)

# Nebo si můžeme pomoci operátoru in zjistit, zda se hodnota v setu nachází
print(6 in distinct_values)   # False
```

Jak můžeme vidět, pokud si vypíšeme `set` pomoci funkce `print`, tak hodnoty jsou ve složených závorkách a ne v hranatých. **Pozor** ale když si chcete vytvořit pouze prázdný set a hodnoty do něj přidat později, tak ho vytvoříme pomoci tzv. `constructoru`.

```
# Správně ☑
my_empty_set = set()

# Špatně ❌
my_empty_set = {}
```

`{}` tento zápis nám vytvoří prázdný `dictionary`, k tomu se dostaneme následně.

Přídávání hodnot do setu

Pro přidání do setu nám poslouží dvě metody - `add` a `update`.

Rozdíl mezi nimi je, že `add` nám přidá novou hodnotu, **jednu**. `update` dokáže přidat do setu hodnoty z dalšího pole a rovnou z nich vybere pouze unikátní hodnoty.

```
my_healthy_food = set(["banana", "apple"])

my_healthy_food.add("kiwi")

# Výsledek
{'apple', 'kiwi', 'banana'}

# Přidání listu pomoci metody update
vegetables = ["carrot", "carrot", "potatoe", "tomatoe"]

my_healthy_food.update(vegetables)

# Výsledek
{'carrot', 'tomatoe', 'potatoe', 'apple', 'kiwi', 'banana'}
```

Mazání hodnot ze setu

Stejně jako `list`, tak i `set` má metodu `remove`, která funguje naprosto stejně.

Stejně tak metoda `clear`, která smaže hodnoty ze setu.

Tuple

Tuplem se úplně zabývat nebude, je to specifická datová struktura podobná listu. Můžeme na položky přistupovat pomoci hranatých závorek. Položky se v tuplu nachází v pořadí v jakém byly inicializovány, ale narozdíl od listu je modifikace tuplu složitá a ze správnosti věci by se dít neměla. `Tuple` nám může sloužit například jako konstantní výčet hodnot.

Dictionary

Datová struktura, která obsauje objekty ve formátu `klíč: hodnota`, respektive `key-value pair`. Co to znamená? Rychlé přistupování k objektům pomoci klíče. Můžeme iterovat nad klíči nebo můžeme iterovat nad hodnotami. `Dictionary` je mocná datová struktura, která se používá hojně ve všech programovacích jazycích.

Jeho zápis a práce s ním je ale složitější než například s listem, protože je o něco komplexnější než `list`, `set` nebo `tuple`.

Jak již bylo zmíněno, `dictionary` inicializuje pomoci složených závorek.

```
my_car = {
  # následně vytvoříme key-value pair
  "brand": "Peugeot",  # jednotlivé key-value pairy od sebe oddělujeme čárkou
  "year": 2000,
  "is_electric": False,
  "defects": ["old", "rusty", "slow"]
}
```

Jak můžeme vidět, do hodnot můžeme v rámci jednoho objektu ukládat různé datové typy, i ty komplexní. Co se týče klíčů, tak ty mohou být `string` nebo `integer` (celé číslo).

Přistupování k hodnotám `dictionary` je možné pomoci hranatých závorek, do kterých zadáme klíč, pro který chceme znát jeho hodnotu. Proto `key-value pair`. Každý klíč má přiřazenou hodnotu. Klíče nemůže existovat bez hodnoty a naopak.

```
print(my_car["brand"]) # Peugeot

# Lze také použít metodu get
print(my_car.get("brand")) # Peugeot
```

Pomoci stejného zápisu můžeme měni hodnoty pro zadaný klíč

```
my_car["year"] = 2001

# Lze využít metodu update
my_car.update({"year": 2001})
```

Pokud potřebujeme z `dictionary` dostat data, máme několik způsobů.

Metoda `keys`

Pokud chceme dostat z `dictionary` jeho klíče, můžeme na objektu zavolat metodu `keys()`.

```
print(my_car.keys())

# Výstup
dict_keys(['brand', 'model', 'year'])
```

Metoda `values`

Pokud ale chceme z `dictionary` dostat všechny jeho hodnoty, tak se nám bude hodit metoda `values()`

```
print(my_car.value())

# Výstup
dict_values(['Peugeot', 2000, False, ['old', 'rusty', 'slow']])
```

Metoda `items`

Tato metoda je trochu specifická, vrací nám list tuplů, kde první hodnota je klíč, druhá hodnota je příslušná hodnota. Díky tomu, že je to tuple, tak se `key-value pairy` nedají editovat.

```
print(my_car.items())

# Výstup
dict_items([('brand', 'Peugeot'), ('year', 2000), ('is_electric', False),
('defects', ['old', 'rusty', 'slow'])])
```

Tyto tři metody mají specifickou vlastnost. Můžete si všimnout, že to není obyčejný list, ale `dict_keys`, `dict_values` a `dict_items`. Tyto tři speciální "struktury" slouží jako pohled, ajťácky `view`, na list hodnot. Nad těmito strukturami se dá normálně iterovat pomoci cyklů, ale nejde k nim přistupovat pomoci hrantých závorek a indexu, jak je tomu u listů.

```
# Procházení jednotlivých klíču v cyklu ☑
for key in my_car.keys():
  print(key)

# Nefunguje ❌
print(my_car.keys()[1])      # error
```

Kombinování datových struktur dohromady

Ano, aby to bylo jednoduché, tak můžeme jednotlivé struktury kombinovat do sebe.

List v listu už jsme viděli, ale také můžeme mít list objektů (`dictionaries`), list tuplů atd. Některé kombinace například nedávají smysl.

Za nejčastěji používané bych považoval list objektů a vnořené objekty.

```
# list objektů
my_family = [
    { "name": "Jana", "role": "wife" },
    { "name": "Viktor", "role": "brother"},
  ]

print(my_family)
# [{'name': 'Jana', 'role': 'wife'}, {'name': 'Viktor', 'role': 'brother'}]

# Zanořené objekty do sebe
my_device = {
  "name": "Macbook Pro"
  "display": {
    "size": 13,
    "unit_of_measure": "inch",
    "is_retina": True,
    "resolution": {
      "width": 2560,
      "height": 1600
    }
  }
}

# Jak vypíšeme hodnotu z vlastnosti display, rozlišení, výška?

print(my_device["display"]["resolution"]["height"]) # 1600
```

Knihovny aneb `modules`

Python jako takový nabízí funkcionalitu, ale můžeme používat i vestavěné moduly, které nám například pomohou s matematickými výpočty – modul `math`. Také existuje modul, který nám pomáhá s výběrem náhodných čísel – `random`. Pak je tu modul `collections`, který si představovat nebudeme, ale pro informaci nabízí práci s datovými strukturami na steroidech.

I `pandas`, `numpy` nebo `matplotlib` jsou moduly. Tyto tři už jsou mnohem rozsáhlejší než výše zmiňované dva.

Jak se s moduly pracuje? Jejich import by měl být vždy na začátku kódu. K připojení modulu do kódu slouží klíčové slovo `import`, případně `from` specifický `import`.

```
import random

print(random.randint(1, 10))    # Vypíše náhodné číslo od 1 do 10
```

Můžeme vidět, že se používá tečka pro přístup k metodě modulu. Jedná se o "tečkovou notaci" (`dot notation`).

Pokud víme, že potřebujeme jen jednu metodu z celého modulu, můžeme import optimalizovat.

```
from random import randint

print(randint(1, 10))    # Vypíše náhodné číslo od 1 do 10
```

Ve všech možných příkladech si můžeme všimnout, že `pandas` se referencuje jako `pd`. Každému importovanému modulu můžeme přiřadit alias pomocí klíčového slova `as`.

```
# nastavíme alias pomocí klíčového slova "as"
import pandas as pd

# všude v kódu poté používáme alias "pd"
pd.DataFrame()
```

NumPy a Pandas

Tyto dvě knihovny jsou jedny z nejpoužívanějších k práci nejen s daty. Python je ale využíván i pro vývoj větších systémů, kde je potřeba zajistit, že kód nemá žádné vedlejší efekty, tak se pro práci s daty nepoužívají vestavěné datové typy, ale datové struktury z knihovny NumPy, případně z jiných knihoven, které se na to specializují.

Například v klasickém Python listu můžeme mít různé datové struktury a typy najednou. To je strašně neefektivní pokud potřebujeme dělat jakékoliv matematické operace. Takový list zazýváme nehomogenní. NumPy má datové typy, které vám zaručí homogennost dat, které v této kolekci naleznete a proto jsou operace nad touto strukturou rychlé.

V NumPy se náhrada za Python list jmenuje Array (mimochodem, array (pole) se používá ve většine programovacích jazyků).

Klávesové zkraty, které se můžou hodit

- `alt + shift + šipka nahoru` kopíruje řádek nad současný řádek
- `alt + shift + šipka dolu` kopíruje řádek pod současný řádek
- `ctrl + /` - zakomentování současného řádku a zároveň zakomentování výběru

N-dimenzionální pole

Než se ponoříme do práce s NumPy a Pandas, tak je dobré se rozšířit pojmy, které je dobré znát.

N-dimenzionální cokoliv, ale většinou se jedná o pole/array/list. N zastává počet dimenzí.

```
# 1d list
one_d_list = [1, 2, 3, 4, 5, 6]

# 2d list, často také nazýván matice
two_d_list = [[1, 2, 3], [4, 5, 6]]

# 3d list najdete níže
```

NumPy datové typy

NumPy narozdíl od Pythonu nabízí více datových typů, které nám umožní mít ještě přesnější matematické výpočty, pokud potřebujeme mnoho desetinných čísel.

Jak jsme si již řekli, tak Python nám nabízí vestavěné datové typy:

- string
- int, float, complex
- bool
- set, tuple, range, dict, list

Tyto datové typy jsou dostatečné na základní operace nad daty, ale pokud je otřeba složitějších matematických operací nad desetinnými čísly, tak je dobré využít knihovny NumPy a typů z ní. Například:

- `np.longdouble`, který je nepřesnější, co Python nabízí, ale také paměťově nejnáročnější.

Dobré je si uvědomit i limitace, které přináší reprezentace desetinných čísel počítačem.

Více informací zde Floating Point Arithmetic: Issues and Limitations

Statistické funkce v NumPy

NumPy nabízí spoustu statistických funckí.

- aritmetický průměr `mean` (`nanmean` ignoruje NaN hodnoty)
- vážený průměr `average`
- medián `median` (`nanmedian` ignoruje NaN hodnoty)
- odchylky
- histogram

Odkaz na na dokumentaci, kde se dozvíte více k jednotlivým funkcím.

Pandas

Jedna z nejpoužívanějších knihoven pro práci s daty. Proč tomu tak je? Pandas totiž nabízí spoustu funkcionalit, které práci s daty ulehčují. Například vlastní datové struktury jako `Series` a `Dataframe`. Podíváme se na obě, protože jsou nedílenou součástí práce s Pandas. Zároveň `Series` je podřazená `DataFrame`, a tak je potřeba je znát obě.

Jak si představit workflow s NumPy, Pandas a Matplotlib?

Máme data, která si potřebujeme rerezentovat, analyzovat, upravit v Pythonu. Pomoci Pandas si načteme `CSV` soubor, vytvoříme si z načtených dat `DataFrame`. Zpracujeme data pomoci funkcionalit, co nám Pandas nabízí. Tyto fukncionality můžeme dolpnit možnostmi, které nabízí NumPy, a provádět matematické, statistické nebo jiné operace jednoduše, bez odborné znalosti matematiky nebo statistiky. Pořád ale musíte chápat, co která funkce dělá a jaký je její use-case.

Když máme data zpracovaná, tak na řadu přichází Matplotlib. Je to knihovna, která nám poskytuje několik možností, jak vizualizovat data. Nabízí různé druhy grafů. Tyto různé druhy grafů mají specifickou potřebu formátu dat a zde se nám zase hodí Pandas, protože díky Pandas je manipulace s daty jednoduchá.

Zní to jednoduše, ale je za tím hodně práce. Nebojte se řešit věci pokus omyl. Ctrl + Z se hodí!

Pandas - Series

Datový typ `Series` si můžeme představit jako `list` z Pythonu, ale nabízí funkcionalitu navíc. Index v `Series` nemusí být vždy jako u listu, tedy numerická řada čísel, začínající od `0`. Můžeme indexovat písmeny, nebo čím je potřeba. Cokoliv `Series` předáme jako parametr pro `index`, to bude sloužit pro indexování. Je nutné ale myslet na to, že pokud chceme mít vlastní index, je potřeba mít připravenou dostatečné množství hodnot pro index, abychom mohli zaindexovat každou hodnotu.

```
import numpy as np
import pandas as pd


# Defulatně se index chová jako index pro pole
s = pd.Series(np.random.randn(5))
print(s)

# ale můžeme index určit sami. Je potřeba mít ale dostatečný počet hodnot
s = pd.Series(np.random.randn(5), index=["a", "b", "c", "d", "e"])
print(s)
```

Objekt `Series` můžeme vytvořit i z `dictionary`. Dejme tomu, že máme jako `dictionary` uloženy data ze soutěže, kde soutěžící sbírali body.

```
results = {
    "Patrik": 20,
    "Jana": 30,
    "Josef": 34,
    "Barbora": 33,
    "Jakub": 32
}

s = pd.Series(results)

print(s)
```

I v tomto případě můžem poskytnout parametru `index` vlastní hodnoty. Víme, že bylo přihlášeno 6 závodníků, ale dostavilo se jen 5, kteří opravdu závodili. Jaký bude výsledek?

```
registered = ["Patrik", "Jana", "Josef", "Daniel", "Barbora", "Jakub"]
results = {
    "Patrik": 20,
    "Jana": 30,
    "Josef": 34,
    "Barbora": 33,
    "Jakub": 32
}

s = pd.Series(results, index=registered)

print(s)
```

Pokud máme v `index` hodnoty, pro které nemáme data, tak u nich bude `NaN`.

NaN (not a number) je v Pandas standardní označení chybějící hodnoty.

Pokud potřebujeme z objektu `Series` dostat jen data, bez indexů a ostatních příkras, můžeme se k nim dostat pomoci vlastnosti `array` ze `Series` objektu.

```
registered = ["Patrik", "Jana", "Josef", "Daniel", "Barbora", "Jakub"]
results = {
    "Patrik": 20,
    "Jana": 30,
    "Josef": 34,
    "Barbora": 33,
    "Jakub": 32
}

s = pd.Series(results,
              index=registered,
              name="Výsledky") # Pomoci parametru name si můžeme pojemenovat data
při výpisu

print(s)

# Nad tímto polem můžeme zavolat další funkce, které nám nabízí Pandas.
arr = s.array
print(arr)
print(arr.mean()) # Průměrná hodnota z pole
print(arr.median()) # Medián z pole
print(arr.argmax()) # Index položky s maximální hodnotou - vrací index prvního
výskytu maximální hodnoty
print(arr.argmin()) # Index položky s minimální hodnotou - vrací index prvního
výskytu minimální hodnoty

# Také si toto pole můžeme z Pandas objektu převést na NumPy objekt a používat zde
zase funkce, které nabízí NumPy
arr_as_numpy = s.array.to_numpy(copy=True) # v NumpPy se datový typ array jmenuje
ndarray
print(arr_as_numpy)
for e in arr_as_numpy:
  print(e)

# Také je možnost Series objekt převést přímo na Python list
arr_as_list = s.to_list()
print(arr_as_list)
```

Pandas - DataFrame

DataFrame je hlavní datová struktura Pandas. Je nadřazená `Series`. `DataFrame` můžeme chápat jako datový typ `dictionary` ale na steroidech.

Když si pomoci Pandas načtete soubor CSV, výsledek této funkce bude uložen jako `DataFrame`.

Jedná se o dvoudimenziální datovou strukturu, je to matice hodnot. Nejjednodušší bude si reprezentaci `DataFrame` představit jako Excel soubor. Sloupce jsou nějak pojmenované a každý řádek má také své číslo. To je matice. Hodnotu pak máme uloženou například na pozici `A2`.

Pokud budeme pokračovat s našimi soutěžícími, ale uděláme ze soutěže turnaj o více disciplínách, už nemůžeme použít pro reprezentaci dat objekt `Series`. Potřebujeme `DataFrame`.

```
registered = ["Patrik", "Jana", "Josef", "Daniel", "Barbora", "Jakub"]
results = {
    # Klíč slouží jako sloupec a zároveň jako název sloupce
    "Running": pd.Series(np.random.randint(1, 40, 6), index=registered),
    "Jumping": pd.Series(np.random.randint(1, 40, 6), index=registered),
    "Swimming": pd.Series(np.random.randint(1, 40, 6), index=registered),
}

df = pd.DataFrame(results)

print(df)
```

Práce se sloupci v DataFrame

Jak už víme z předchozích lekcí, můžeme s datovými strukturami různě manipulovat.

Můžeme sloupce přidávat, mazat, počítat s nimi.

```
registered = ["Patrik", "Jana", "Josef", "Daniel", "Barbora", "Jakub"]
results = {
    # Klíč slouží jako sloupec a zároveň jako název sloupce
    "Running": pd.Series(np.random.randint(1, 40, 6), index=registered),
    "Jumping": pd.Series(np.random.randint(1, 40, 6), index=registered),
    "Swimming": pd.Series(np.random.randint(1, 40, 6), index=registered),
}

df = pd.DataFrame(results)

df["Result"] = df["Running"] + df["Jumping"] + df["Swimming"] # Vytvoříme nový
sloupec, který obsahuje součet všech disciplín

print(df)

df.pop("Swimming")

# Je potřeba přepočítat výsledky
df["Result"] = df["Running"] + df["Jumping"]

print(df)
```

Metoda `assing`

Metoda `assign` nám umožňuje vytvářet nové sloupce, kterým jako hodnotu můžeme předat přes lambda funkci. V předchozím příkladu jsme si ukázali, jak se dá vypočítat sloupec result, je to super, ale tento sloupec pak není reaktivní a nepřepočítá se, pokud nějaký sloupec z dat vymažme. Pokud jej ale přidáme pomoci metody `assign`, tak se přepočítá při každém vypsání `DataFrame`.

Metoda vrací kopii `DataFrame` s novými sloupci.

```
registered = ["Patrik", "Jana", "Josef", "Daniel", "Barbora", "Jakub"]
results = {
    # Klíč slouží jako sloupec a zároveň jako název sloupce
    "Running": pd.Series(np.random.randint(1, 40, 6), index=registered),
    "Jumping": pd.Series(np.random.randint(1, 40, 6), index=registered),
    "Swimming": pd.Series(np.random.randint(1, 40, 6), index=registered),
}

df_2 = pd.DataFrame(results)

df_2.assign(Result=lambda x: x.Running + x.Jumping + x.Swimming)

print(df_2) # Proč tam sloupec není? Protože jsme si neuložili návratovou hodnotu
funkce assign do proměnné

res = df_2.assign(Result=lambda x: x.Running + x.Jumping + x.Swimming)
print(res)
```

Výpis hodnot pomoci metod `Head` a `Tail`

Funkce `print` je možný způsob, jak vypisovat hodnoty z DataFrame, ale v Google Colab můžeme využít hezkého zobrazení tabulek, když využijeme metody `head` a `tail`.

Metoda `Head`

Slouží pro vypsání počtu záznmů od začátku datasetu.

Metoda `Tail`

Slouží pro vypsání počtu záznamů od konce datasetu.

Když metodám nedáme parametr, vypisují defaultně 5 řádků.

Operace nad řádky a sloupci

Jako další funkcionalitu Pandas nabízí možnost agregovat nad celým `DataFrame` nebo nad jednotlivými sloupci. Neplést prosím s metodou `assign`.

- `sum()` - metoda sečte hodnoty ve sloupci
- `mean()` - metoda vypočítá aritmetický průměr z hodnot ve sloupci

Máme možnost ale tyto metody zavolat i nad jednotlivými sloupci.

```
df["some_colum"].sum()
df["some_colum"].mean()
# výsledek volání je hodnota pro daný sloupec
```

Možnost určit operaci pomoci metody `agg()`.

Agregaci, kterou chceme provést, je možné také zadat pomoci metody `agg()`, které jako parametr předáme odkaz na funkci nebo název předdefinované funkce.

```
def my_function():
  return "Bazinga!!!"

# Sum pomoci metody agg
df.agg(np.sum)
df.agg("sum")

# Mean pomoci metody agg
df.agg(np.mean)
df.agg("mean")

# Vlastní funkce
df.agg(my_function)
```

Více informací v dokumentaci.

Výběr sloupců pro zpracování nebo výpis

Pokud chceme pracovat nebo vypsat jen jednotlivé sloupce, tak je můžeme definovat pomoci přístupu přes hranaté závorky a názvu sloupcu, se kterým máme v úmyslu pracovat.

```
# Jeden sloupec ze všech
print(df["column_1"])

# více sloupců zároveň
print(df["column_1", "column_2", "column_5"])
```

Řazení dat podle hodnot

Pandas nabízí možnost řadit hodnoty v `DataFrame` podle zvoleného sloupce.

```
# Definujeme podle jakého sloupce chceme řadit
df1.sort_values(by="column_name")

# Můžeme řadit i nad více sloupci zároveň
df1.sort_values(by=["col_1", "col_2"])
```

Práce s chybějímící nebo `NaN` hodnotami je v řazení v Pandas v pohodě, protože nám s tím Pandas pomůže. U metody `sort_values` existuje parametr `na_position`, který určuje, kam takové hodnoty ve výsledku zařadit. Možnosti jsou na začátek nebo na konec dat. Ve výchozím nastavení je to na konec dat, ale lze přes paramater `na_position="first"` nastavit, že mají být na začátku.

```
# Seřadí hodnoty a všechny nečíslené nebo chybějící hodnoty budou na začátku
výsledku.
df1.sort_values(by="column_name", na_position="first")
```

Práce s CSV

Co je `CSV`. Jedná se o textovou reprezentaci dat, jednotlivé hodnoty jsou od sebe oddělené oddělovačem. Oddělovač může být **čárka**, **středník**, **tabulátor**, aj.

Většinou CSV soubor obsahuje na prvním řádku názvy sloupců a data jsou až od druhého řádku. Výhodou je, že tento soubor otevře jakýkolik textový editor. Excel nám umožňuje CSV reprezentaci převést do klasické Excel podoby pomoci vestavěné funkcionality v záložce "Data".

```
ID,Jméno,Přijmení,Věk,Email
1,Patrik,Procházka,29,
2,Jana,Procházková,24,
3,Joe,Doe,49,joedoe@email.com
```

V učebních materiálech najdete CSV soubor s daty o populaci amerických měst. Stejně tak v každém Google Colab sešitu najdete v záložce soubory, složku `sample_data`, ve které jsou nějaká vzorová data ve formátu CSV.

Načtení CSV souboru do Pythonu

Python samotný nabízí funkcionalitu, jak pracovat se soubory ze souborového systému počítač. Tento přístup je ale zdlouhavý a v případě mnoha dat může být i neefektivní. `Pandas` nabízí funkcionalitu, která nám umožní data jednoduše nahrát a vytvořit z nich `DataFrame` se kterým můžeme ihned pracovat.

```
import pandas as pd

data_from_csv = pd.read_csv("/content/sample_data/california_housing_test.csv") #
platí pro Google Colab! V každém prostředí fungují importy jinak.

print(data_from_csv)
```