Metodika analýzy dat: Od základů po aplikace metod strojového učení

Úvod do programování a jazyk Python 1. část

Jakub Steinbach, Jan Vrba

Ústav matematiky, informatiky a kybernetiky
VŠCHT

30.9.2024

Obsah slajdů I

Obecné informace

Proč se učit Python?

- je velice jednoduchý pro začátečníky
- jeho používání vytváří obecně dobré návyky pro psaní kódu v libovolných jazycích
- univerzální jazyk (web, datová analýza, strojové učení, hry, mobilní aplikace,...)
- v současné době je jeden z nejrozšířenějších jazyků
- multiparadigmatický jazyk
- existuje velká komunita a široká nabídka knihoven

Specifika Pythonu

- je to case-sensitive jazyk
- bloky kódu jsou definovány pomocí odsazení (tělo funkce, definice třídy, atp.)
- řada knihoven a jejich částí je implementována v kompilovaných jazycích které jsou při interpretaci nahrány jako std. Pythoní moduly (např. numpy a scipy jsou napsány v C)
- součástí standartní knihovny je i testovací framework
- repozitář balíčků a knihoven PyPI (Python Package Index) instalovatelných pomocí nástroje *pip*

Hello world!

```
print("Hello, World!")
```

Klíčová slova v Pythonu

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Tabulka: Slova která se nesmí používat pro označení entit v Pythonu

Příkazy v Pythonu

```
x = 7
```

- Python vytvoří objekt třídy Integer, nastaví mu hodnotu 7 a pomocí přiřazovacího příkazu = přiřadí tento objekt identifikátoru x
- v Pythonu nepracujeme přímo s nějakou oblastí paměti (rozdíl s C)
- v Pythonu chceme dodržet max. množství znaků na řádek (79) $\Longrightarrow$ příkazy mohou být i víceřadkové

```
x = 8 * (1 + 2 +
    3 + 4 + 5 +
    6 + 7)
```

```
x = 1 + 2 + 3 + \
    3 + 4 + 5 + \
    6 + 7 + 8
```

Identifikátory

- identifikátor slouží k rozlišení jednotlivých entit (proměnné, objekty, funkce,...)
- povolené znaky jsou písmena $a - z$, $A - Z$, číslice $0 - 9$ a podtržítko _
- identifikátory nemohou začínat číslem
- klíčové slovo nemůže být použito jako identifikátor (viz slajd 7)
- maximální ani minimální délka identifikátoru není specifikována
- norma PEP8 specifikuje jak pojmenovávat třídy, proměnné, funkce, objekty atp.

Jak volit jména identifikátorů?

- **Obecně** - nepoužíváme "l" (el), "O", nebo "I" (i) pro pojmenovávání (možnost záměny v různých sadách fontů)
- **Moduly a balíčky** - krátké jména, malá písmena, slova oddělená podtržítkem (useful_module)
- **Názvy tříd** - CapWords konvence (StudentClass)
- **Funkce a proměnné** - malá písmena, slova oddělená podtržítkem (line_width). mixedStyle povolen pouze pokud tento styl pojmenování převládá v důsledku použití starších knihoven a balíčků (např. threading.py)
- **Konstanty** - kapitálky (MY_CONSTANT)

Datové typy v Pythonu a operace s nimi

Základní proměnné v Pythonu

- celé číslo (integer)
- číslo s plovoucí desetinnou čárkou (float)
- řetězec (string) - sekvence UNICODE znaků

```
b = False  # boolean
x = 6  # integer
y = 6.0  # float
z = 1.3 + 2.4j  # complex
s = "some string"  # string
s = 'this is also string'  # string
```

- pro zjištění typu entity - **type**

```
print(type(s))
```

- můžeme použít i vícenásobné přiřazení

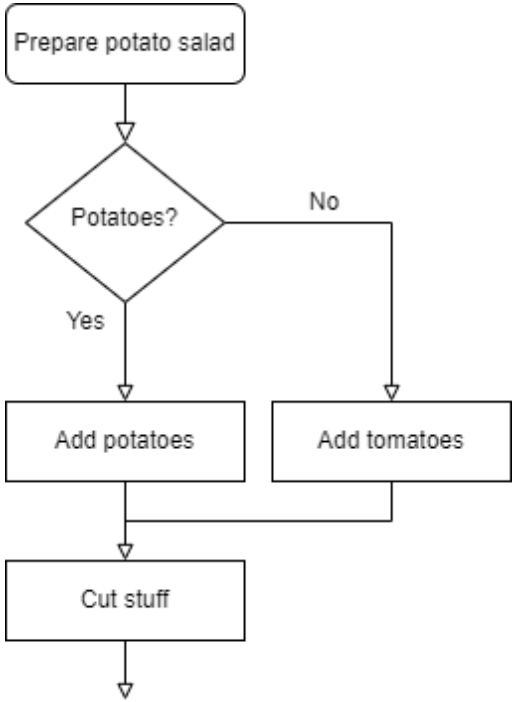
```
x, y, z, s = 6, 6.0, 1.3 + 2.4j, "some string"
```

Aritmetické operace v Pythonu

- sčítání a + b
- odčítání a - b
- násobení a * b
- dělení a / b
- celočíselné dělení a // b
- zbytek po dělení a % b
- mocnění a ** b
- negace ∼a (výsledek je -(a+1))
- unární plus +a
- unární mínus -a
- násobení matic A @ B (od verze 3.5)

Větvení programu

Větvení - "když nemáte brambory, dejte tam rajčata"



- v případě, že na základě nějaké podmínky mají být vykonány různé bloky kódu dochází k tzv. větvení programu
- v Pythonu pomocí odsazením můžeme definovat blok kódu

Logické podmínky v Pythonu

- rovnost

```
a == b
```

- nerovnost

```
a != b
```

- menší

```
a < b
```

- větší

```
a > b
```

- menší nebo rovno

```
a <= b
```

- větší nebo rovno

```
a >= b
```

Logické podmínky a jejich kombinace

- pro testování identity se používá klíčové slovo **is**
- pro skládání logických podmínek je možné použít operátory **and**, **or**
- pro negaci logické podmínky slouží klíčové slovo **not**

Příklady:

- x je větší než b a y je menší než c

$$x > b \ \textbf{and} \ y < c$$

- x je větší než b nebo y je menší než c

$$x > b \ \textbf{or} \ y < c$$

- neplatí, že x je větší než b nebo y je menší než c

$$\textbf{not} \ (x > b \ \textbf{or} \ y < c)$$

Větvení - příklad

- za klíčovým slovem **if** následuje logický výraz a za ním znak :
- následuje odsazený blok kódu který se vykoná v případě že je uvedený logický výraz pravdivý (True)
- klíčové slovo **elif** specifikuje další možnou podmínku
- klíčové slovo **else** uvozuje blok kódu který se vykoná, pokud nebyla splněna žádná z výše uvedených podmínek

```
if some_text == "do A":
    print("lets do A")
elif some_text == "do B":
    print("lets do B")
else:
    print("lets do something else")
```

Podmíněný výraz

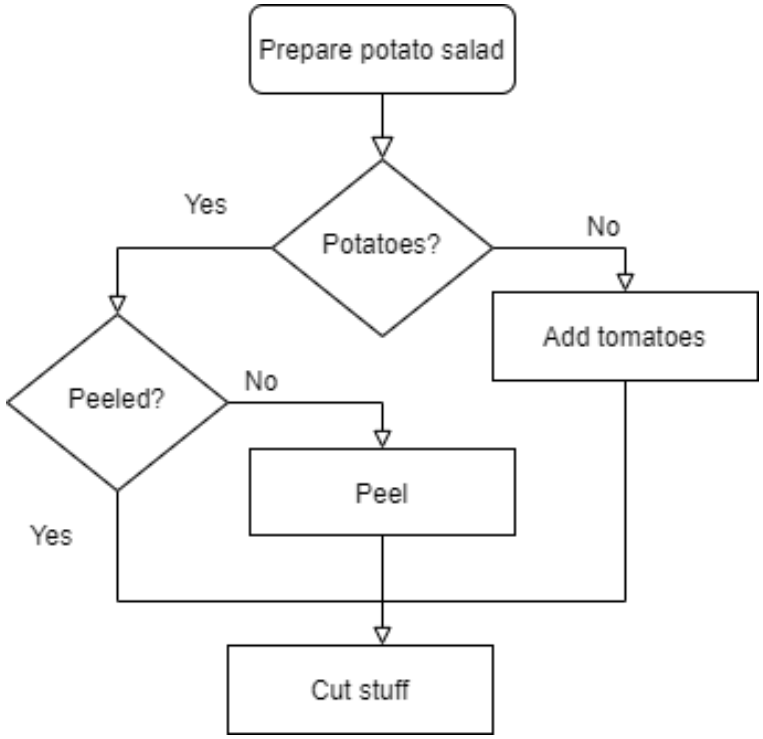
- v Pythonu je možné využít tzv. podmíněný výraz
- proměnné je přiřazena hodnota na základě logické podmínky

```
result = value if condition else other_value
```

```
a = 8
result = 2 * a if a == 5 else 0
print("a >= 8") if >= 8 else print("a < 8")
```

Vnořené podmínky

- v pythonu je limit na počet vnořených bloků nastaven na 20
- v případě že máme víc než 5 vnořených bloků (nejenom podmínek, ale i smyček) děláme asi něco špatně



Vnořené podmínky - salát a pass

```
if potatoes:
    if not potatoes_peeled:
        print("peeling potatoes")
else:
    print("adding tomatoes")
print("cutting stuff")
```

ekvivaletní zápis

```
if potatoes:
    if not potatoes_peeled:
        print("peeling potatoes")
    else:
        pass
else:
    print("adding tomatoes")
print("cutting stuff")
```

Kontejnerové datové typy v Pythonu a operace s nimi

Základní kontejnerové typy

- seznam (list) - uspořádaná měnitelná posloupnost

```
some_list = [6, 6.0, "some string"]
```

- n-tice (tuple) - uspořádaná neměnná posloupnost

```
my_tuple = (6, 6.0, "some string")
```

- množina (set) - neuspořádaná kolekce unikátních neměnitelných hodnot

```
my_set = {6, 6.0, "some string"}
```

- slovník (dictionary) - neuspořádaná (od verze 3.7 uspořádaná) kolekce dvojic klíč - hodnota

```
my_dictionary = {"x" : 6, "s" : "some str"}
```

Práce se seznamem

- vytvoření

```
my_list = [1, "text", 6.0, [2, 3]]
```

- výběr prvku

```
my_list[1]
```

- změna prvku

```
my_list[0] = 7
```

- seznam prvních dvou prvků

```
my_list[0:2]
```

- změna vícero prvků

```
my_list[0:2] = ["some", 3.5 ]
```

- zjištění délky seznamu

```
len(my_list)
```

Práce se seznamem

- výběr posledních dvou prvků

```
my_list[-2:]
```

- vložení prvku na konec seznamu

```
my_list.append(100)
```

- vložení prvku na konkrétní pozici

```
my_list.insert(1, "item")
```

- smazání prvku na vybrané pozici

```
my_list.pop(1)
```

- smazání prvku podle hodnoty

```
my_list.remove(7)
```

- seřazení seznamu od největšího prvku

```
my_list.sort(reverse=True)
```

Práce se seznamem

- spojování seznamů

```
my_list + ["another", "list"]
my_list.extend(["another", "list"])
```

- obrácení pořadí prvků

```
my_list.reverse()
```

- zjištění počtu prvků určité hodnoty

```
my_list.count("another")
```

- pozice prvního výskytu prvku určité hodnoty

```
my_list.index("another")
```

- odstranění všech prvků ze seznamu

```
my_list.clear()
```

Práce s n-ticí

- vytvoření

```
my_tuple = ("car", "bus", "train")
```

- přístup k prvku

```
my_tuple[1]
```

- rozbalení

```
vehicle1, vehicle2, vehicle3 = my_tuple
```

- spojování

```
my_tuple + ("train",)
```

- zjištění počtu prvku s konkrétní hodnotou

```
my_tuple.count("train")
```

- nalezení pozice prvku s konkrétní hodnotou

```
my_tuple.index("train")
```

Práce s n-ticí

- násobení n-tice

```
my_set * 2
```

Stejným způsobem můžeme násobit i seznam!

Práce s množinou

- vytvoření množiny (pozor, množina obsahuje pouze unikátní prvky)

```
my_set = {1, 2, 3, 4, 4}
```

- zjištění jestli je prvek v množině

```
3 in my_set
```

- iterace přes všechn prvky množiny

```
for item in my_set:
    print(item)
```

- přidání prvku do množiny

```
my_set.add(5)
```

- odstranění prvku z množiny

```
my_set.remove(1)
```

Práce s množinou

- sjednocení množin

```
my_set.union({2, 6})
```

- rozšíření množiny o prvky ze sjednocení

```
my_set.update({2, 6})
```

- průnik množin

```
my_set.intersection({2, 6})
```

- odstranění všech prvků které nejsou přítomny ve druhé množině

```
my_set.intersection_update({2, 6})
```

- odstranění vších prvků z množiny

```
my_set.clear()
```

Práce se slovníkem

- vytvoření slovníku

```
my_dict = {
"name": "Akira",
"age": 8,
"occupation": "test subject",
"ID": 28
}
```

- výběr hodnoty

```
my_dict["name"]
my_dict.get("name")
```

- seznam klíčů ve slovníku

```
my_dict.keys()
```

- seznam hodnot ve slovníku

```
my_dict.values()
```

Práce se slovníkem

- změna jedné hodnoty ve slovníku

```
my_dict["age"] = 9
```

- změna vícero hodnot

```
my_dict.update({"age": 10,
"occupation": "Tokyo destroyer"})
my_dict.get("name")
```

- přidání hodnoty

```
my_dict["nationality"] = "Japanese"
```

- odstranění hodnoty

```
my_dict.pop("nationality")
del my_dict["age"]
```

Smyčky

Smyčka while

- smyčka **while** umožňuje vykonávat blok kódu dokud není splněna požadovaná podmínka
- proměnné relevantní k podmínce musí být před spuštěním smyčky existovat

```
iteration_number = 1
while iteration_number < 10:
    print("this is iteration: ", iteration_number)
    iteration_number += 1
```

Předčasné ukončení smyčky - break

- klíčové slovo **break** umožní okamžitě ukončit nadřezanou smyčku

```
iteration_number = 1
while iteration_number < 10:
    if iteration_number == 4:
        break
    print("this is iteration: ", iteration_number)
    iteration_number += 1
```

Zastavení aktuální iterace a spuštění další - continue

- klíčové slovo **continue** okamžitě ukončí aktuální iteraci a smyčka pokračuje následující iterací

```
iteration_number = 1
while iteration_number < 10:
    iteration_number += 1
    if iteration_number == 4:
        continue
    print("this is iteration: ", iteration_number)
```

Smyčka while - vykonání kódu při nesplnění podmínky pro pokračování smyčky - else

```
iteration = 1
while iteration < 10:
    iteration += 1
else:
    print("iteration is not < 10")
```

- blok uvozený klíčovým slovem **else** se vykoná jakmile není splněná podmínka pro spuštění další iterace příslušné smyčky
- př: prohledáváme seznam a v případě že ho prohledáme celý a nic nenalezneme můžeme využít blok else (úspora 1 proměnné)

Smyčka For

- smyčka **for** umožňuje iterovat jakoukoliv sekvenci (resp. objekt s funkcí **__iter__()**, viz později) a přistoupit k jednotlivým prvkům
- příklady sekvencí: řetezec, seznam, n-tice, slovník

```
for  letter in "alphabet":
    print(letter)
for item in [1, "car", [4, 5]]
    print(item)
for item in (1, 2, 3)
    print(item)
dict = {"color": blue, "number": 5}
for key in  dict:
    print(key, ":", dict[key])
```

Smyčka For a slovník - keys(), values()

- smyčkou **for** lze iterovat hodnoty slovníku nebo klíče
- funkce **values()** vytvoří seznam hodnot
- funkce **keys()** vytvoří seznam klíčů

```
dict = {"key1": "value1", "key2": "value2"}
for key in dict.keys():
    print(key)

for value in dict.values():
    print(value)
```

Smyčka For a slovník - items()

- smyčkou **for** lze iterovat i přes dvojice klíč - hodnota
- funkce **items()** vytvoří seznam entic klíč - hodnota

```
dict = {"key1": "value1", "key2": "value2"}
for  item in dict.items():
    print(item)

for key, value in dict.items():
    print(key, ":", value)
```

Smyčka For a její ukončení

- pomocí příkazu **break** lze předčasně ukončit nadřazenou smyčku **for**

```
for item in [1,2,3,4]:
    print("pre break item: ", item)
    if item == 3:
        break
    print("post break item: ", item)
```

Smyčka For a ukončení aktuální iterace

- pomocí příkazu **continue** lze předčasně ukončit aktuální iteraci

```
for item in [1,2,3,4]:
    print("pre break item: ", item)
    if item == 3:
        continue
    print("post break item: ", item)
```

Smyčka For a funkce range()

- funkce **range(start, stop, step)** vrací požadovanou sekvenci čísel od hodnoty **start** do hodnoty **stop - 1** s krokem daným hodnotou **step**
- parametry **stop, step** jsou volitelné, sekvence pak začíná od 0 s krokem 1

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

for i in range(2, 5):
    print(i)

for i in range(2,15,3):
    print(i)
```

Smyčka For a příkazy else a pass

- smyčka **for** nemůže být prázdná, pokud nechceme zatím programovat obsah, lze použít klíčové slovo **pass** a vyhnout se tak chybě při spuštění

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

- klíčové slovo **else** specifikuje blok kódu, který se vykoná při v případě že je smyčka úspěšně vykonána (není předčasně ukončena příkazem **break**)

```
for i in range(5):
    print(i)
else:
    print("finished")
```

Vnořené smyčky For

- **for** smyčky lze vnořovat do sebe
- př. generování zápasů v Scheveningském systému (šachy, MtG, Heartstone,...)

```
for a in ["Carlsen", "Liren", "Caruana"]:
    for b in ["Aronian", "Giri", "Grischuk"]:
        print(a, "–", b)
```

Funkce

Čtení ze standartního vstupu - input()

- funkce **input()** umožní uživateli prostřednictvím std. vstupu (klávesnice) zadat nějaký řetězec
- ukončení zadávání je pomocí klávesnice Enter
- funkce vrátí uživateli řetězec, který obsahuje vstup zadaný z klávesnice

```
obtained_string = input("Type something and press Ente
print("You typed: ", obtained_string)
print("Give me more:")
more_data = input()
print("You gave me ", more_data)
```

Vybrané základní funkce Pythonu

- **abs(number)** - vrátí absolutní hodnotu čísla
- **bin(number)** - vrátí binární reprezentaci čísla
- **chr(unicode)** - vrátí znak s příslučným kódem
- **dir(object)** - vrátí seznam metod a vlastností objektu
- **eval(expression)** - vyhodnotí a spustí daný výraz (NEPOUŽÍVAT!!)
- **help(command)** - do std. výstupu zobrazí help příslušného příkazu (objektu,...)

Vybrané základní funkce Pythonu

- **len(object)** - vrátí délku objektu (např. počet znaků řetězce)
- **max(iterable)** - vrátí maximální hodnotu iterovatelného objektu
- **min(iterable)** - vrátí nejmenší hodnotu iterovatelného objektu
- **sum(iterable)** - vrátí součet iterovatelného objektu
- **ord(integer)** - převede znak na odpovídající hodnotu Unicode
- **round(number)** - vrátí zaokrouhlené číslo
- **type(object)** - vrátí typ daného objektu

Funkce definované uživatelem

- funkce je blok kódu který se vykoná v okamžiku jejího zavolání a lze jej volat opakovaně
- proměnné vytvořené uvnitř funkce existují jenom v rámci této funkce
- z funkce nelze přistupovat k proměnným existujícím mimo ní (pokud nejsou globální, nebo nejsou na úrovni modulu)
- když v programu píšeme obdobnou věc podruhé, měli bysme se zamyslet jestli nepoužít funkci
- pro definici funkce se používá klíčové slovo **def**
- funkce se volá jejím jménem + ()

```
def hello_world():
    print("Hello World from a function!")
hello_world()
```

Funkce - argumenty

- funkci lze předat argumenty, tedy informace s kterými pak může dále pracovat
- do závorek za jejím jménem lze definovat libovolné argumenty, oddělené čárkou

```
def introduce(name, planet):
    print("I␣am␣", name, "␣from␣",
          planet, "␣planet")
introduce("Spock", "Vulcan")
```

Funkce - návratová hodnota

- výstupem funkce může být libovolný objekt, několik hodnot, atp.
- za klíčovým slovem **return** jsou uvedeny výstupní hodnoty
- vícero hodnot jsou vráceny jako n-tice, pokud není specifikováno jinak

```
def discriminant(a, b, c):
    return b ** 2 - 4 * a * c

d = discriminant(1, 4, 1)

def get_extremes(values_list):
    return min(values_list), max(values_list)

min_value, max_value = get_extremes([1,2,3,4,5])
something = get_extremes([1,2,3,4,5])
print(type(something))
```

Funkce - pass

- tělo funkce musí obsahovat nějaký kód, jinak nepůjde program spustit
- minimální tělo funkce lze vytvořit pomocí klíčového slova **pass**
- tato konstrukce se využívá i když chceme napsat v Pythonu interface

```
def convert_data(data):
    pass
```

Funkce - korektní přístupy ke komplexním výstupům funkce

- využití seznamu
- využití slovníku
- využití data class (později)

```
def test_return(a, b):
    return a, b
```

```
def test_return(a, b):
    return [a, b]
```

```
def test_return(a, b):
    return {"a": a, "b": b}
```

Funkce - *args

- funkci v Pythonu je možné předat různý počet argumentů
- např. součet několika čísel, atd.

```
def my_multiply(*args):
    total = 1
    for arg in args:
        total *= arg
    return total

print(my_multiply(1,2,3,4))

def some_args(first_arg, *args):
    print("first arg: ", first_arg)
    for arg in args:
        print("*args: ", arg)
some_args("first", "second", "third")
```

Funkce - pojmenované parametry

- lze používat názvy argumentů, potom nezáleží na pořadí parametrů

```
def print_name(first_name, last_name):
    print(first_name, "␣", last_name)

print_name(last_name="Turing", first_name="Alan")
```

Funkce - **kwargs

- funkci lze předat libovolný počet parametrů s klíčovými slovy pomocí **
- parametry jsou pak dostupné v rámci funkce v podobě slovníku
- konvence je označovat tento libovolný počet klíčových slov jako **kwargs

```
def print_name(**names):
    print(names["first_name"],"␣",names["last_name"])

print_name(last_name="Turing", first_name="Alan")
```

Funkce - defaultní hodnoty parametrů

- funkce v Pythonu mohou pracovat s defaultními hodnotami parametrů
- v případě, že uživatel nedodá požadovanou hodnotu parametru, je použita defaultní hodnota

```
def print_favourite_color(color="red"):
    print("My favourite colos is ",color)

print_favourite_color()
print_favourite_color("blue")
print_favourite_color(color="#00FF00")
```

Lambda funkce

- lambda funkce jsou anonymní funkce (nemají jméno)
- v případě že chceme použít funkci pouze po krátkou dobu
- používají se v případě že chceme předat funkci jako argument
- časté použití s funkcí map nebo filter, callback funkce v GUI

Př: identity function

```
def identity(x):
    return x

lambda x: x
```

Př: druhá mocnina (nekorektní a zbytečná)

```
a = lambda x: x ** 2
a(4)
x = (lambda x: x ** 2)(2)
print(x)
```

Lambda funkce - více argumentů a funkce vyššího řádu

- lambda funkce může pracovat s vícero argumenty
- lambda funkce může mít jako argument i funkci, takové funkce jsou označovány jako funkce vyššího řádu (higher-order function)

Př: vícenasobný argument

```
(lambda x,y: x * y)(3,4)
```

Př: funkce vyššího řádu

```
ho_func = lambda x, func: x + func(x)
ho_func(3, lambda x: x * 2)
```

V Pythonu jsou funkce vyšího řádu funkce **map()**, **filter()**, **reduce()**.

Funkce map

map(function, iterable[, iterable2, ...])

- funkce **map()** umožní provést operaci s prvky iterovatelných objektů bez použití smyčky
- funkce **map()** vytvoří map objekt (iterátor), který vznikne aplikací transformační funkce na každou položku iterovatelného objektu
- častá je kombinace s **lambda** funkcí

```
numbers = [1, 2, 3]
doubled = map(lambda x: x * 2, numbers)
print(list(doubled))
floated = map(float, numbers)
print(list(floated))
```

Funkce filter()

- funkce **filter(function, iterable)** vybere z dané sekvence prvky pro které je hodnota testovací funkce *function* True
- výstupem je iterátor obsahující vybrané prvky
- časté je použití v kombinaci s lambda funkcí

```
numbers = [1, 2, 3, 4, 5]
odd = filter(lambda x: x % 2 != 0, numbers)
even = filter(lambda x: x % 2 == 0, numbers)
print(list(odd))
print(type(even))
print(list(even))
```

Řetězce a práce s nimi

Formátování textu - f-Strings (Python 3.6+)

- uvození řetězce prefixem *f*
- **{ }** je označován jako placeholder a lze do něj napsat formátovací předpis

```
name = "Spock"
age = 55
print(f"My␣name␣is␣{name},␣I'm␣{age}")
print(f"My␣name␣is␣{name},␣I'm␣{age:b}") # binary
print(f"Hot␣is␣{500.246579:.2f}␣Kelvins") #floating
print(f"Hot␣is␣{500.246579:e}␣Kelvins") #scientific
print(f"Hot␣is␣{0b1110100:d}␣Kelvins") #decimal
for x in range(1, 11):
    print(f"{x:2d}␣{x␣**␣2:3d}␣{x␣**␣3:4d}")
```

Práce s řetězcem - join()

- metoda **string.join(iterable)** vytvoří řetězec, spojených z prvků v iterable, které jsou spojené přes řetězec **string**
- **iterable** musí obsahovat řetězce

```
to_join = ("Name","Surname")
print(" ".join(to_join))
some_dict("Name": "Adam", "Surname": "Bernau")
print("separator".join(some_dict))
```

Práce s řetězcem - split()

- metoda **split(separator)** rozdělí řetězec v místech specifikovaných **separator** a vrátí seznam řetězců

```
to_split = "Along␣came␣a␣man␣carrying␣a␣large␣python."
print(to_split.split("␣"))
print(to_split.split("a"))
```

Práce s řetězcem - replace()

- funkce **replace(old, new [, count])** nahradí podřetězce **old** novým subřetězcem **new**
- v případě, že je specifikován parametr **count**, provede se nejvýše **count** nahrazení
- návratovou hodnotou řetězec s nahrazenými podřetězci

```
cat_string = "Cat is cute. Cat does meow."
wut_string = cat_string.replace("Cat", "Dog")
dog_string = cat_string.replace("Cat", "Dog", 1)
```

Práce s řetězcem - zfill()

- metoda **string.zfill(len)** přidá na začátek řetězece **string** nuly do délky specifikované **len**
- pokud je na začátku řetězce znaménko (+-) jsou nuly přidány až za něj
- použití při práci s binární reprezantací čísel, formátování čísel s fixní délkou
- návratovou hodnotou je doplněný řetězec

```
print("text".zfill(10))
print("+123".zfill(6))
```

Práce s řetězcem - lower(), upper()

- metoda **string.lower()** převede velká písmena v řetězci na malá
- metoda **string.upper()** převede malá písmena v řetězci na velká
- návratovou hodnotou obou metod je převedený řetězec

```
some_string = "AlpHAb3t"
print(some_string.lower())
print(some_string.upper())
```

Práce s řetězcem - strip(), rstrip(), lstrip()

- metoda **string.strip([char])** odstraní z řetězce mezery zleva i zprava
- pokud je specifikován paramter **char** odstraní všechny znaky zleva i zprava
- analogicky fungují metody **rstrip([char]), lstrip([char])** které odstraní požadované znaky pouze zleva resp. zprava
- návratovou hodnotou je upravený řetězec

```
some_text = "    no space  "
print(some_text.strip())
another_text = "aaaaabbbaa"
print(another_text.strip("a"))
print(another_text.rstrip("a"))
print(another_text.lstrip("a"))
```

Práce s řetězcem - find()

- metoda **string.find(substring[, start[, end]])** najde pozici prvního výskytu podřetězce **substring** v řetězci **string**
- volitelné argumenty **start** a **stop** specifikují pozice odkud má prohledávání začít a kde má skončit
- návratovou hodnotou je index počátku výskytu hledaného podřetězce

```
some_text = "alphabet␣alphabet"
print(some_text.find("a"))
print(some_text.find("a", 1, 5))
```

Práce s řetězcem - count()

- metoda **string.count(substring [,start, stop])** vrátí počet výskytů podřetězce **substring** v řetězci string
- volitelné parametry **start, stop** specifikují pozici začátku a konce prohledávání

```
test_string = "abababaaaab"
print(test_string.count("aa"))
print(test_string.count("b", 2, 5)
```