

C2184 Úvod do programování v Pythonu (2021)

8. Procvičování, life hacks

- Tato lekce obsahuje věci, bez kterých se teoreticky obejdeme, ale mohou být užitečné
- Co byste si měli zapamatovat:
 - Generátorové výrazy (... for ... in ...)
 - Funkce enumerate, reversed, sorted, zip

Generátorové výrazy (*generator expressions*)

- Časté operace s kolekcemi:
 - Mapování = chceme aplikovat funkci na každý prvek
['1', '2', '3', '4', '5', '6'] -> [1, 2, 3, 4, 5, 6]
 - Filtrování = chceme vybrat prvky splňující podmínku
[1, 2, 3, 4, 5, 6] -> [2, 4, 6]
- Generátorové výrazy (... for ... in ...) zjednodušují mapování a filtrování.

Mapování

```
[1]: strings = '1 2 3 4 5 6'.split()
      print(strings)
```

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

```
[2]: # Klasický přístup:
      numbers = []
      for s in strings:
          numbers.append(int(s))
      print(numbers)
```

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

```
[3]: # Generátorový výraz:  
numbers = [int(s) for s in strings]  
print(numbers)
```

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

Filtrování

```
[4]: # Klasický přístup:  
even_numbers = []  
for i in numbers:  
    if i % 2 == 0:  
        even_numbers.append(i)  
print(even_numbers)
```

[2, 4, 6]

```
[5]: # Generátorový výraz:  
even_numbers = [i for i in numbers if i % 2 == 0]  
print(even_numbers)
```

[2, 4, 6]

Filtrování + mapování

```
[6]: even_numbers_squared = [i**2 for i in numbers if i % 2 == 0]  
print(even_numbers_squared)
```

[4, 16, 36]

Slovníky

```
[7]: square_dict = {i: i**2 for i in numbers}  
print(square_dict)
```

{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}

```
[8]: original_dict = {1: 'A', 2: 'B', 3: 'C'}  
inverted_dict = {v: k for k, v in original_dict.items()}  
print(inverted_dict)
```

{'A': 1, 'B': 2, 'C': 3}

- Typ výsledné kolekce je dán typem závorek:
 - [... for ... in ...] vytváří seznam (*list comprehension*)
 - {... for ... in ...} - vytváří množinu (*set comprehension*)

- {...: ... for ... in ...} - vytváří slovník (*dictionary comprehension*)
- Výjimka: (... for ... in ...) vytváří iterátor, nikoliv *n*-tici

```
[9]: (x for x in numbers if x >= 3) # iterátor
```

```
[9]: <generator object <genexpr> at 0x7f1a3c2420b0>
```

```
[10]: tuple(x for x in numbers if x >= 3) # n-tice
```

```
[10]: (3, 4, 5, 6)
```

Iterátory

- **Iterátor** je objekt, který umí postupně vracet nějaké prvky.
- Volání funkce `next` na iterátoru:
 - Vráť další prvek, pokud ještě má
 - Vyhodí výjimku `StopIteration`, pokud už vyčerpал prvky
- Funkce `iter` vytvoří z iterovatelného objektu iterátor

```
[11]: numbers = [1, 2, 3] # seznam
      number_iterator = iter(numbers) # iterátor
```

```
[12]: number_iterator
```

```
[12]: <list_iterator at 0x7f1a3c279b20>
```

```
[13]: next(number_iterator)
```

```
[13]: 1
```

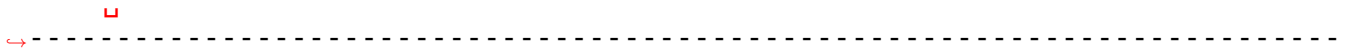
```
[14]: next(number_iterator)
```

```
[14]: 2
```

```
[15]: next(number_iterator)
```

```
[15]: 3
```

```
[16]: next(number_iterator)
```



[illegible]

```
/tmp/ipykernel_11930/1427042954.py in <module>
----> 1 next(number_iterator)
```

StopIteration:

- Cyklus for je vlastně jen syntaktická zkratka pro použití iterátoru

```
[17]: # Cyklus while s iterátorem:
      iterator = iter(numbers)
      while True:
          try:
              i = next(iterator)
          except StopIteration:
              break
      print(i)
```

- 1
- 2
- 3

```
[18]: # Cyklus for:
      for i in numbers:
          print(i)
```

- 1
- 2
- 3

Iterator vs iterable

- Iterátor (*iterator*) = objekt, na kterém lze volat `next`
 - Když se vyčerpá, nelze už prvky procházet! (musíme vytvořit nový iterátor)
 - Neumí `len`, indexování...
- Iterovatelný objekt (*iterable*) = objekt, ze kterého lze vytvořit iterátor pomocí funkce `iter`
 - Kolekce, řetězce, `range`...

(Volání `iter` na iterátoru vrátí samotný iterátor, proto každý iterátor je zároveň iterovatelným objektem).

```
[19]: iter(number_iterator) is number_iterator
```

[19]: True

- Výhoda iterátoru vůči seznamu - prvky nemusí být nikde fyzicky uloženy

```
[20]: total = sum([i**2 for i in range(10_000_000)]) # Spočítané prvky,
      ↪uložíme do seznamu, pak je sečteme
```

```
[21]: total = sum(i**2 for i in range(10_000_000)) # Spočítané prvky,
      ↪rovnou sečteme, nemusí se nikam ukládat
```

- Výhoda iterátoru vůči seznamu - prvky se generují, až když jsou potřeba

```
[22]: square_iterator = (i**2 for i in range(10**80))
      print(next(square_iterator))
      print(next(square_iterator))
      print(next(square_iterator))
```

```
0
1
4
```

Co všechno nám vrací iterátor?

- Funkce iter
- Funkce pro chytré iterování: enumerate, reversed, zip, map, filter...
- Generátorové výrazy bez [] nebo {}: (... for ... in ...)
- Generátorové funkce

Chytré iterování

- Funkce enumerate očísluje prvky

```
[23]: names = ['Bob', 'Alice', 'Cyril']
      for i, name in enumerate(names):
          print(f'{i}. {name}')
```

```
0. Bob
1. Alice
2. Cyril
```

```
[24]: for i, name in enumerate(names, start=1):
      print(f'{i}. {name}')
```

```
1. Bob
2. Alice
3. Cyril
```

- Funkce reversed prochází od konce

```
[25]: for name in reversed(names):
      print(name)
```

```
Cyril
Alice
Bob
```

- Funkce sorted seřadí prvky (vytváří nový seřazený seznam)

```
[26]: for name in sorted(names):
      print(name)
```

```
Alice
Bob
Cyril
```

- Funkce zip prochází více objektů najednou

```
[27]: for name, letter in zip(names, ['Marley', 'Nováková', 'Svoboda']):
      print(name, letter)
```

```
Bob Marley
Alice Nováková
Cyril Svoboda
```

- Pozor, funkce enumerate, reversed, zip nevytváří kolekci, pouze *iterátor* určen k jednorázovému proždění prvků
- Pouze sorted vrací normální seznam

```
[28]: iterator = reversed(names)
      for name in iterator:
          print(name)
```

```
Cyril
Alice
Bob
```

```
[29]: for name in iterator: # Iterátor se již vyčerpá
      print(name)
```

- Iterátor můžeme “vysypat” do seznamu:

```
[30]: names_numbers = list(zip(names, numbers))
      print(names_numbers)
```

```
[('Bob', 1), ('Alice', 2), ('Cyril', 3)]
```

Rekurze

- Když funkce volá sama sebe.

```
[31]: def factorial(n: int) -> int:
      '''Calculate factorial of number n.'''
      if n == 1:
          return 1
      else:
          return n * factorial(n - 1)
```

```
[32]: factorial(5)
```

[32]: 120

- Pozor, hrozí zacyklení (např. volání factorial(0)).
- Nepřímá rekurze: např. funkce a volá funkci b, funkce b volá funkci a.
- Příklad rekurze: prohledávání vnořených seznamů:

```
[33]: 5 in [1, 2, [3, [4], [5, 6], 7], 8, 9]
```

[33]: False

```
[34]: def deep_search(deep_list: list, item: object) -> bool:
      '''Decide if item is present in deep_list or in any of its_
      ↪elements, recursively.'''
      for x in deep_list:
          if x == item:
              return True
          elif isinstance(x, list) and deep_search(x, item):
              return True
      return False
```

```
[35]: deep_search([1, 2, [3, [4], [5, 6], 7], 8, 9], 5)
```

[35]: True

```
[36]: deep_search([1, 2, [3, [4], [5, 6], 7], 8, 9], 0)
```

[36]: False

- Příklad rekurze: generování všech permutací
 - Nula prvků - lze seřadit pouze jedním způsobem: []
 - Více prvků - vybereme první prvek a skombinujeme se všemi permutacemi zbylých prvků
 - Např. permutace prvků 1, 2, 3, 4:

- * 1 + všechny permutace 2, 3, 4
- * 2 + všechny permutace 1, 3, 4
- * 3 + všechny permutace 1, 2, 4
- * 4 + všechny permutace 1, 2, 3

- Pouze ukázka, v praxi využijte funkci `permutations` z modulu `itertools`

```
[37]: from typing import List
```

```
def permutations(items: list) -> List[list]:  
    if len(items) == 0:  
        return [[]]  
    result = []  
    for head in items:  
        remaining_items = [x for x in items if x != head]  
        for tail in permutations(remaining_items):  
            new_permutation = [head] + tail  
            result.append(new_permutation)  
    return result
```

```
[38]: permutations([1, 2, 3, 4])
```

```
[38]: [[1, 2, 3, 4],  
       [1, 2, 4, 3],  
       [1, 3, 2, 4],  
       [1, 3, 4, 2],  
       [1, 4, 2, 3],  
       [1, 4, 3, 2],  
       [2, 1, 3, 4],  
       [2, 1, 4, 3],  
       [2, 3, 1, 4],  
       [2, 3, 4, 1],  
       [2, 4, 1, 3],  
       [2, 4, 3, 1],  
       [3, 1, 2, 4],  
       [3, 1, 4, 2],  
       [3, 2, 1, 4],  
       [3, 2, 4, 1],  
       [3, 4, 1, 2],  
       [3, 4, 2, 1],  
       [4, 1, 2, 3],  
       [4, 1, 3, 2],  
       [4, 2, 1, 3],  
       [4, 2, 3, 1],  
       [4, 3, 1, 2],  
       [4, 3, 2, 1]]
```

Generické typy

- Upřesnění k přednášce **6. Funkce**
- Generický typ = typ, který lze upřesnit (parametrizovat) pomocí hranatých závorek
 - Např. generický typ `list` (seznam) -> parametrizovaný generický typ `list[int]` (seznam celých čísel)
- Python <= 3.8
 - Parametrizované generické typy v typových anotacích se píšou velkým písmenem (např. `List[int]`) a je třeba je importovat se z modulu `typing`
- Python >= 3.9:
 - Základní vstavané typy (malým písmenem) lze použít i jako generické typy (např. `list[int]`)
 - Fungují i původní typy z modulu `typing`, ale považují se za zastaralé
 - V modulu `typing` zůstává: `Union[X, Y]`, `Optional[X]`, `Any...` (od Pythonu 3.10: `Union[X, Y] == X|Y`, `Optional[X] == X|None`)

```
[39]: # Python <= 3.8
from typing import List, Dict, Tuple

def foo(a: List[int], b: Dict[str, float]) -> Tuple[bool, int]:
    return (True, 9)
```

```
[ ]: # Python >= 3.9
def foo(a: list[int], b: dict[str, float]) -> tuple[bool, int]:
    return (True, 9)
```

Rozšiřující učivo (nepovinné)

Anonymní funkce `lambda`

- Vytvoření funkce beze jména
- Příklad: chceme seřadit studenty podle příjmení – použijeme `sorted` s parametrem `key`

```
[40]: students = [('Alice', 'Nováková'), ('Cyril', 'Svoboda'), ('Bob', 'Marley')]
sorted(students) # Řadí podle křestního jména
```

```
[40]: [('Alice', 'Nováková'), ('Bob', 'Marley'), ('Cyril', 'Svoboda')]
```

- S pojmenovanou funkcí:

```
[41]: def surname(person):  
        return person[1]  
  
sorted(students, key=surname) # Řadí podle příjmení
```

```
[41]: [('Bob', 'Marley'), ('Alice', 'Nováková'), ('Cyril', 'Svoboda')]
```

- S lambda funkcí:

```
[42]: sorted(students, key = lambda person: person[1]) # Řadí podle  
      ↪ příjmení
```

```
[42]: [('Bob', 'Marley'), ('Alice', 'Nováková'), ('Cyril', 'Svoboda')]
```

Funkce map a filter

- Funkce map mapuje

```
[43]: strings = ['1', '2', '3']  
for i in map(int, strings):  
    print(i+1)
```

```
2  
3  
4
```

- Funkce filter filtruje

```
[45]: numbers = [1, 2, 3, 4, 5, 6]  
for i in filter(lambda i: i % 2 == 0, numbers):  
    print(i)
```

```
2  
4  
6
```

- Pozor, obě funkce vrací iterátor
- Funkce map a filter lze vždy přepsat na generátorový výraz a opačně (podle chuti)

```
[46]: iterator1 = map(int, '1 2 3'.split())  
iterator2 = (int(s) for s in '1 2 3'.split())
```

```
[47]: list1 = list(map(int, '1 2 3'.split()))  
list2 = [int(s) for s in '1 2 3'.split()]
```

```
[48]: list1 = list(filter(lambda i: i % 2 == 0, [1, 2, 3]))  
list2 = [i for i in [1, 2, 3] if i % 2 == 0]
```

Generátorové funkce (*generator functions*)

- Funkce, kterých návratovou hodnotou je *generator iterator* (typ iterátoru)
- *Generator iterator* generuje hodnoty až když jsou potřeba (ne už při zavolání funkce)
 - Jednu hodnotu vyžádáme pomocí funkce `next`
 - Více hodnot pomocí `for` cyklu
- Generované hodnoty se v těle funkce uvádějí slovem `yield` (místo `return`)
- Slovo *generator* někdy označuje *generator function*, někdy *generator iterator*

```
[49]: from typing import Iterator

# Generator function
def echo(word: str) -> Iterator[str]:
    while len(word) > 0:
        yield word
        word = word[1:]
```

```
[50]: iterator = echo('Hello') # Generator iterator
iterator
```

```
[50]: <generator object echo at 0x7f1a3c1ca200>
```

```
[51]: next(iterator) # Vygenerujeme 1. hodnotu
```

```
[51]: 'Hello'
```

```
[52]: for s in iterator: # Vygenerujeme zbylé hodnoty
      print(s)
```

```
ello
llo
lo
o
```

```
[53]: for s in iterator: # Iterátor se už vyčerpал, nevypíše se nic
      print(s)
```

```
[54]: next(iterator) # Iterátor se už vyčerpал, vyhodí se chyba typu
      ↪ StopIteration
```

```
↪ -----
```

```

StopIteration                                Traceback (most recent call last)
--
/tmp/ipykernel_11930/3617280921.py in <module>
----> 1 next(iterator) # Iterátor se už vyčerpал, vyhodí se chyba
StopIteration

```

StopIteration:

```
[55]: iterator = echo('Hello') # Nový iterátor, opět můžeme generovat
      next(iterator)
```

```
[55]: 'Hello'
```

- *Generator iterator* může generovat i nekonečnou posloupnost (není to problém, protože hodnoty se generují, až když je potřeba)

```
[56]: def even_numbers() -> Iterator[int]:
      i = 2
      while True:
          yield i
          i += 2
```

```
[57]: iterator = even_numbers()
      iterator
```

```
[57]: <generator object even_numbers at 0x7f1a3c1ca350>
```

```
[58]: for x in iterator:
      print(x)
      if x >= 10:
          break
```

```

2
4
6
8
10

```

```
[59]: # Generujeme dál
      for x in iterator:
          print(x)
          if x >= 20:
              break
```

```

12

```

14
16
18
20