C2184 Úvod do programování v Pythonu

7. 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

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

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

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

Filtrování

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

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

Filtrování + mapování

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

Slovníky

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

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

- 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

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

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

Iterátory

- **Iterátor** je objekt, který umí postupně vracet nějaké prvky.
- Volání funkce `next` na iterátoru:
  - Vrátí další prvek, pokud ještě má

  - Vyhodí výjimku `StopIteration`, pokud už vyčerpal prvky
- Funkce `iter` vytvoří z iterovatelného objektu iterátor

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

```
[ ]: number_iterator
```

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

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

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

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

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).

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

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

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

```
[ ]: 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

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

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

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

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

- Funkce `reversed` prochází od konce

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

- Funkce `sorted` seřadí prvky (vytváří nový seřazený seznam)

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

- Funkce `zip` prochází více objektů najednou

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

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

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

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

- Iterátor můžeme “vysypat” do seznamu:

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

Rekurze

- Když funkce volá sama sebe.

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

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

- 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:

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

[ ]:
```
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
```

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

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

- 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`

[ ]:
```python
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
```

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

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`...
- Python 3.10:
    - `Union[X, Y]` -> `X | Y`
    - `Optional[X]` -> `X | None`

[ ]:
```python
# 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
# Python <= 3.8
from __future__ import annotations

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

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

Rozšiřující učivo (nepovinné)

Pojmenované n-tice – `NamedTuple`

- Nevýhoda n-tic: musíme si pamatovat, který prvek má jaký význam :(
- S normálními n-ticemi:

[ ]:
```python
address1 = ('Kamenice', 5, 'Brno')
address2 = ('Žerotínovo náměstí', 9, 'Brno')

def print_address(address: tuple[str, int, str]) -> None:
    print('Street:', address[0])
    print('Number:', address[1])
    print('City:', address[2], end='\n\n')

print_address(address1)
print_address(address2)
```

[ ]:
```python
print(address1)
```

- S pojmenovanými n-ticemi:

[ ]:
```python
from typing import NamedTuple

class Address(NamedTuple):
    street: str
    number: int
    city: str

address1 = Address('Kamenice', 5, 'Brno')
address2 = Address('Žerotínovo náměstí', 9, 'Brno')

def print_address(address: Address) -> None:
    print('Street:', address.street)
    print('Number:', address.number)
    print('City:', address.city, end='\n\n')
```

```
print_address(address1)
print_address(address2)
```

[ ]:
```
print(address1)
```

- Kazda nove vytvorena trida pojmenovanych n-tic (napr. `Address`) je podtypem normalni n-tice

[ ]:
```
isinstance(address1, tuple)
```

[ ]:
```
len(address1)
```

[ ]:
```
address1[2]
```

Anonymní funkce lambda

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

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

- S pojmenovanou funkcí:

[ ]:
```
def surname(person):
    return person[1]

sorted(students, key=surname)  # Řadí podle příjmení
```

- S lambda funkcí:

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

Funkce `map` a `filter`

- Funkce `map` mapuje

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

- Funkce `filter` filtruje

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

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

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

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

```
[ ]: 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*

```
[ ]: from typing import Iterator

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

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

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

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

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

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

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

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

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

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

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

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