Iterování

- Protokol umožňuje používat `for` cyklus pro všechny **iterovatelné** obejkty (tzv. **iterable**)
- Kdy je objekt iterovatelný?
  1) Pokud je uložený jako sekvence
  2) Produkuje během iterace 1 hodnotu v 1 okamžik

  [!info] ***iterable*** - objekt podporující volání `iter()` ***iterator*** - objekt vrácený voláním `iter()` podporující volání `next()`

- Abychom mohli podporovat `iter()` metodu je nutné implementovat dunder metodu `__iter__`
- Iterátor, který je je onou funkcí produkovaný musí zase implementovat dunder metodu `__next__` a správně vyvolat vyjímku `StopIteration` (na konci produkce)

```
iterator = iter([1, 2, 3])

next(iterator)
next(iterator)
next(iterator)
# exception StopIteration
next(iterator)
```

- Funkce `zip()`, `enumerate()`, `filter()` vrací *iterable* a rovněž *iterable* přijímají, lze je tedy zanořovat

```
list(zip(enumerate(range(10)), range(10)))
```

Modul `itertools`

- Implementuje sadu užitečných iterátorů
- Preferujeme nad vlastní implementací (rychlost a paměťová úspornost)
- Přehled všech dostupných je zde

`itertools.starmap`

```
import itertools
import operator


# operator.mul přijímá 2 argumenty
itertools.starmap(operator.mul, [[1, 2], [2, 3], [3, 4], [4, 5]])
# výsledek - [2, 6, 12, 20]
```

`itertools.filterfalse`

```python
import itertools


itertools.filterfalse(None, [[], [1, 2, 3], [1]])
# [[]]

itertools.filterfalse(lambda x: x % 3, range(20))
# [0, 3, 6, 9, 12, 15, 18]
```

Comprehensions

- Společně s cykly se jedná o nejčastější využití *iteračního* protokolu
- Jistý způsobem nahrazuje `for` cyklus (jiný zápis i jiné provedení "uvnitř")

```python
# Oba následující příklady dělají totéž

squares = []

for number in numbers:
    squares.append(number ** 2)

# ----------------------------------------

squares = [number ** 2 for number in numbers]

# obecně -------------------------------

new_list = [expression for member in iterable]
```

- Výsledkem je vždy **nový** seznam
- Nepsané pravidlo: pokud je *list comprehension* delší než 2 řádky, raději použijeme bežný *for* cyklus

Používání podmínek v list comprehensions

1) Filtrování

```python
sentence = 'the rocket came back from mars'

vowels = []
for char in sentence:
    if char in 'aeiou':
        vowels.append(char)

# ----------------------------------------

vowels = [i for i in sentence if i in 'aeiou']
```

```
new_list = [expression for member in iterable (if conditional)]
```

2) Úprava prvků

```
numbers = [10, 20, -5, 10]
abs_numbers = []

for number in numbers:
    if number > 0:
        abs_numbers.append(number)
    else:
        abs_numbers.append(abs(number))

# ----------------------------------------

numbers = [10, 20, -5, 10]

abs_numbers = [number if number > 0 else abs(number) for number in numbers]

# ----------------------------------------

new_list = [expression (if conditional) for member in iterable]
```

- Je možné i zanořování (i když občas bývá nepřehledné)
- Samozřejmě záleží na pořadí

```
colors = ['red', 'green', 'blue']
sizes = ['S', 'M', 'L', 'XL']

tshirts = []

for color in colors:
    for size in sizes:
        tshirts.append((color, size))

# ----------------------------------------

tshirts_by_color = [(color, size) for color in colors for size in sizes]

# ----------------------------------------

[expression for target1 in iterable1 if condition1
            for target2 in iterable2 if condition2 ...
            for targetN in iterableN if conditionN]
```

Set comprehensions

- Jak z názvu plyne, jako výsledek navrací množinu

```
sentence = 'the rocket came back from mars'

unique_vowels = {i for i in sentence if i in 'aeiou'}

# {'a', 'e', 'o'}
```

Dictionary comprehensions

- Obdobné => výsledkem bude slovník

```
sentence = 'the rocket came back from mars'

word_len = {word: len(word) for word in sentence.split(' ')}

# {'the': 3, 'rocket': 6, 'came': 4, 'back': 4, 'from': 4, 'mars': 4}
```

Poznámky

- Lokální proměnné nejsou dále dostupné (rozsah platnosti)
- Přináší znatelné zrychlení (až 2x)
  - V rámci interpretu jsou prováděny rychlostí jazyka `C`

Generátorové funkce

- Výpočet nekončí a nevrací hodnotu jako u běžných (`return`), ale funkce se "uspí" a následně "probudí" k výpočtu (`yield`)
- Jsou kompilovány jako *generátory* (spojitost s *iteracemi*)
- Vhodné použít pokud nevím zda bude potřeba celý výsledek

```
# klasická funkce
def calculate_squares(n):
    for i in range(n):
        return i ** 2

# generátorová funkce
def generate_squares(n):
    for i in range(n):
        yield i ** 2

# 683 ns ± 52 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
%timeit calculate_squares(20000)

# 244 ns ± 6.12 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
%timeit generate_squares(20000)
```

- Generátor je tzv. ***single-iteration*** objekt

- Možnost iterovat pouze jedenkrát
- Pokud se chceme k výsledku vrátit, musíme hodnoty ukládat “‘python # klasicky list comprehension (výpočet proběhne okamžitě) [number ** 2 for number in range(10)] # vrací - [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

generátorový výraz

(number ** 2 for number in range(10)) # vrací - <generator object at 0x106b73ac0>

generator = (number ** 2 for number in range(10))

zde se vypočítá první hodnota

next(generator) “‘