
KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Jan Laštovička
https://www.inf.upol.cz/lide/jan-lastovicka
jan.lastovicka@upol.cz
17. listopadu 12, 771 46 Olomouc

Úvod do programovacích stylů ◇ poznámky k přednášce

10. Iterátory

verze z 27. listopadu 2024

1 Iterátory

Na hodnoty, které se skládají z lineárně uspořádaných prvků, můžeme pohlížet jako na **posloupnosti**. Například pole `[1, 2, 3]` můžeme chápat jako posloupnost hodnot 1, 2 a 3. Posloupnosti se také nazývají iterovatelné hodnoty.

Průchod posloupností reprezentujeme hodnotou, která se nazývá iterátor. Přesnou definici iterovatelné hodnoty a iterátoru se dozvíme později. Iterátor k iterovatelné hodnotě *`iterable`* získáme zavoláním vestavěné funkce `iter`:

```
iter(iterable) => iterator
```

Například:

```
>>> i = iter([1, 2, 3])
```

Voláním vestavěné funkce `next` na iterátor postupně získáváme prvky posloupnosti:

```
next(iterator) => element
```

Vyzkoušíme:

```
>>> next(i)
1
>>> next(i)
2
>>> next(i)
3
```

Pokud další prvek posloupnosti neexistuje, funkce `next` vyvolá výjimku `StopIteration`. Tedy:

```
 >>> next(i)
StopIteration
```

Přirozeně pro jednu iterovatelnou hodnotu můžeme vytvořit více iterátorů:

```
>>> array = [1, 2, 3]
>>> i1 = iter(array)
>>> i2 = iter(array)
>>> next(i1)
1
>>> next(i2)
1
>>> next(i1)
2
>>> next(i1)
3
>>> next(i1)
StopIteration
>>> next(i2)
2
>>> next(i2)
3
>>> next(i2)
StopIteration
```

Iterátor sám je iterovatelná hodnota představující zbytek dosud neprojité posloupnosti:

```
>>> i1 = iter([1, 2, 3])
>>> next(i1)
1
>>> i2 = iter(i1)
>>> next(i2)
2
```

Pokračování v průchodu `i1` mění průchod `i2`:

```
>>> next(i1)
3
>>> next(i2)
StopIteration
```

Iterátory nelze použít ve funkcionálním programovacím stylu a to z toho důvodu, že při každém zavoláním funkce `next` dojde ke změně iterátoru. Jak víme, změny hodnot jsou ve funkcionálním stylu zakázané.

Řetězec je iterovatelná hodnota. Na řetězec se můžeme dívat jako na posloupnost znaků, které jej tvoří. Například:

```
>>> i = iter("py")
```

```
>>> next(i)
'p'
>>> next(i)
'y'
>>> next(i)
StopIteration
```

Vestavěná funkce:

```
range(start, stop)
```

vrací iterovatelnou hodnotu, která představuje posloupnost celých čísel větších nebo rovno než *start* a menších než *stop*. Volání

```
range(stop)
```

je ekvivalentní s

```
range(0, stop)
```

Například máme:

```
>>> i = iter(range(1, 3))
>>> next(i)
1
>>> next(i)
2
>>> next(i)
StopIteration
```

Vestavěná funkce `list` pro posloupnost vrátí pole prvků posloupnosti. Například:

```
>>> list(range(3))
[0, 1, 2]
>>> list("python")
['p', 'y', 't', 'h', 'o', 'n']
```

Příkaz `for` má tvar:

```
for variable in iterable:
    block
```

kde *variable* je proměnná, *iterable* iterovatelná hodnota a *block* blok kódu. Příkaz se vykoná následovně:

1. Získá se iterátor *iterator* iterovatelné hodnoty *iterable*.
2. Dokud volání `next(`*iterator*`)` nevyvolá výjimku `StopIteration`, tak
   (a) se nastaví proměnná *variable* na návratovou hodnotu funkce `next`
   (b) a vykoná se *block*.

Například:

```
>>> for i in range(3):
        print(i)
0
1
2
```

Otevřený soubor je průchod posloupností, jejíž prvky jsou řádky posloupnosti. Například vezměme soubor `file.txt` s obsahem:

```
line1
line2
line3
```

Následujícím příkazem vytiskneme textové reprezentace všech řádků souboru:

```
>>>> with open("file.txt") as file:
         for line in file:
            print(repr(line))


'line1\n'
'line2\n'
'line3\n'
```

2 Funkce vyšších řádů pro posloupnosti

Vestavěná funkce `map` očekává funkci *function* a posloupnosti *iterable*. Funkce `map` vrátí průchod posloupností návratových hodnot volání funkce *function* na prvky posloupnosti *iterable*.

Například:

```
>>> i = map(lambda x: x + 1, [1, 2, 3])
>>> next(i)
2
>>> next(i)
3
```

```
>>> next(i)
4
>>> next(i)
StopIteration
```

Vestavěná funkce `filter` očekává predikát `predicate` s jedním parametrem a posloupnost `iterable`. Funkce vrací průchod posloupností těch prvků posloupnosti `iterable`, které splňují predikát `predicate`. Například:

```
>>> i = filter(lambda x: x % 2 == 0, [1, 2, 3, 4, 5, 6])
>>> next(i)
2
>>> next(i)
4
>>> next(i)
6
>>> next(i)
StopIteration
```

3 Posloupnost a její průchod jako objekty

Iterátor je objekt, který rozumí zprávám `__next__` a `__iter__` bez argumentů. Zaslání zprávy `__next__` iterátoru, vrátí další jeho prvek. Zaslání zprávy `__iter__` iterátoru pouze vrátí příjemce.

Příklad třídy iterátorů skládajících se ze samých jedniček:

```
class Ones:
    def __next__(self):
        return 1

    def __iter__(self):
        return self
```

Funkce `next` zašle zprávu `__next__` svému jedinému argumentu. Tedy:

```
>>> i = Ones()
>>> next(i)
1
>>> next(i)
1
>>> next(i)
1
```

Instance třídy `Ones` tedy představují průchod nekonečnou posloupností jedniček.

Iterovatelná hodnota je objekt, který rozumí zprávě `__iter__` zaslané bez argumentů. Návratová hodnota zprávy musí být iterátor.

Příklad třídy slov řetězců:

```
class Words:
    def __init__(self, string):
        self.string = string

    def __iter__(self):
        return iter(self.string.split(" "))
```

Použití:

```
>>> for word in Words("ahoj světe"):
        print(word)
ahoj
světe
```

4 Generátory

Příkaz produkující prvek posloupnosti (zkráceně **příkaz `yield`**) nabývá tvaru:

```
yield element
```

kde *`element`* je další prvek posloupnosti. Příkaz může být použit pouze v těle funkce.

Funkce, která obsahuje v těle příkaz `yield`, se nazývá **generator**. Příkaz návratu (příkaz `return`) v těle generátoru smí vracet pouze hodnotu `None`.

Příklad generátoru:

```
def get_numbers():
    i = 0
    while i < 3:
        yield i
        i = i + 1
```

Poté, co se zavolá generátor, vykonávání těla generátoru se pozastaví před vykonáním prvního příkazu, a volání vrátí iterátor. Například volání funkce:

```
>>> i = get_numbers()
```

pouze vytvoří iterátor `i`.

Při vyžádání dalšího prvku funkcí `next` se začne vykonávat tělo generátoru od pozastaveného místa až po příkaz `yield`. Dalším prvkem posloupnosti bude

hodnota určená příkazem `yield`. Vykonávání těla generátoru se pozastaví na následujícím příkazu. Tedy:

```
>>> next(i)
0
```

Vykonávání těla generátoru je pozastaveno na řádku:

```
i = i + 1
```

Popsaným způsobem získáme další prvky posloupnosti:

```
>>> next(i)
1
>>> next(i)
2
```

Skončení vykonávání těla generátoru povede k vyvolání výjimky `StopIteration`:

```
>>> next(i)
StopIteration
```

5 Generující výraz

Iterátor lze vytvořit pomocí **generujícího výrazu**, který má tvar:

```
(element clauses)
```

kde *`element`* je výraz určující prvky iterátoru a část *`clauses`* určuje hodnoty proměnných použitých v *`element`*.

Část *`clauses`* obsahuje za sebe napsané klauzule `if` a `for`. První klauzule je vždy `for` klauzule.

Klauzule `for` má tvar:

```
for variable in iterable
```

kde *`variable`* je proměnná a *`iterable`* iterovatelná hodnota.

Klauzule `if` má tvar:

```
if condition
```

kde *`condition`* je podmínka.

Například:

```
(x + 1 for x in [1, 4, 2, 3] if x % 2 == 0)
```

Hodnotou generujícího výrazu je iterátor, kde prvky jsou určeny následovně:

Vyhodnocení `for` klauzule:

```
for variable in iterable
```

probíhá tak, že se pro každý prvek iterovatelné hodnoty *`iterable`* vyhodnotí zbývající klauzule, kde proměnná *`variable`* bude aktuálním prvkem posloupnosti.

Například klauzule:

```
for x in [1, 4, 2, 3]
```

Vyhodnotí zbytek klauzulí, kde proměnná `x` bude postupně nabývat hodnot 1, 4, 2 a 3.

Při vyhodnocování klauzule `if` se vyhodnotí její podmínka. Pokud je podmínka splněna, pokračuje se následující klauzulí. Jinak se přeruší postupné vyhodnocování klauzulí způsobené naposledy vyhodnocovanou `for` klauzulí.

Pokud se podaří vyhodnotit všechny klauzule až do konce, pak další hodnota iterátoru je hodnota výrazu *`element`*.

Proto:

```
>>> i = (x + 1 for x in [1, 4, 2, 3] if x % 2 == 0)
>>> next(i)
5
>>> next(i)
3
>>> next(i)
StopIteration
```

Pokud je výraz generátoru použit jako argument volání funkce, můžeme vynechat vnější kulaté závorky:

```
>>> list(x ** 2 for x in range(5))
[0, 1, 4, 9, 16]
```

Klauzule `for` může záviset na proměnných uvedených dříve zapsanou `for` klauzulí:

```
>>> list([x, y] for x in range(4) for y in range(x))
[[1, 0], [2, 0], [2, 1], [3, 0], [3, 1], [3, 2]]
```

Uzavřením popisu generování do hranatých místo kulatých závorek rovnou vytvoříme pole:

```
>>> [[x, y] for x in range(4) for y in range(x)]
[[1, 0], [2, 0], [2, 1], [3, 0], [3, 1], [3, 2]]
```

Poslední příklad:

```
>>> [[x, y] for x in range(1, 10) for y in range(2, x) if x % y == 0]
[[4, 2], [6, 2], [6, 3], [8, 2], [8, 4], [9, 3]]
```

6 Eratosthenovo síto

Naším cílem je pomocí Eratosthenova síta vytvořit posloupnost všech prvočísel. Začneme tím, že si vytvoříme průchod posloupností všech nezáporných celých čísel:

```
def get_numbers():
    i = 0
    while True:
        yield i
        i = i + 1
```

Vyzkoušíme:

```
>>> i = get_numbers()
>>> next(i)
0
>>> next(i)
1
>>> next(i)
2
...
```

Pomocná funkce:

```
def is_divisor(a, b):
    return (b % a) == 0
```

rozhoduje, zda `a` dělí `b`. Například:

```
>>> is_divisor(2, 8)
True
>>> is_divisor(2, 7)
False
```

Tedy dvojka je dělí osmičku, ale nedělí sedmičku.

Funkce:

```
def remove_multiples(n, iterable):
    return filter(lambda m: not is_divisor(n, m), iterable)
```

pro posloupnost `iterable` vrací průchod posloupností `iterable`, ze které odstraňuje prvky dělitelné `n`. Takhle například získáme průchod posloupností všech lichých čísel:

```
>>> i = remove_multiples(2, get_numbers())
>>> next(i)
1
>>> next(i)
3
>>> next(i)
5
...
```

Vezměme průchod posloupností všech čísel a odeberme první dva prvky (nulu a jedničku):

```
>>> numbers = get_numbers()
>>> next(numbers)
0
>>> next(numbers)
1
```

Opakováním následujícího dostáváme postupně všechna prvočísla.

1. Odebereme prvek z `numbers` a označme si jej `prime`:

   ```
   prime = next(numbers)
   ```

2. Nastavme `numbers` na průchod posloupností `numbers`, ze které jsme odstranili všechna čísla, která jsou dělitelná číslem `prime`:

   ```
   numbers = remove_multiples(prime, numbers)
   ```

3. Číslo `prime` je další prvočíslo.

Otestujeme:

```
>>> prime = next(numbers)
>>> numbers = remove_multiples(prime, numbers)
>>> prime
2
>>> prime = next(numbers)
>>> numbers = remove_multiples(prime, numbers)
>>> prime
```

```
3
>>> prime = next(numbers)
>>> numbers = remove_multiples(prime, numbers)
>>> prime
5
...
```

Právě popsaný postup vtělíme do generátoru:

```
def get_primes():
    numbers = get_numbers()
    next(numbers)
    next(numbers)
    while True:
        prime = next(numbers)
        numbers = remove_multiples(prime, numbers)
        yield prime
```

Zavoláním generátoru získáme průchod posloupností všech prvočísel:

```
>>> primes = get_primes()
>>> next(primes)
2
>>> next(primes)
3
>>> next(primes)
5
```

Otázky a úkoly na cvičení

Ve zdrojových souborech k přednášce je přiložena databáze filmů rozdělená do souborů. Soubory jsou ve formátu TSV (Tab-Separated Values). Každý řádek souboru obsahuje položky oddělené tabulátorem. Údaje o filmech a režisérech se nalézají ve třech souborech:

1. `movie.tsv` Řádek popisuje film
2. `person.tsv` Řádek popisuje osobu.
3. `director.tsv` Řádek určuje režiséra filmu. (Film může mít více režisérů)

Položky řádku v `movie.tsv`:

1. identifikátor filmu
2. název filmu

3. rok vydání
4. délka v minutách

Položky řádku v `person.tsv`:

1. identifikátor osoby
2. jméno osoby
3. rok narození

Položky řádku v `director.tsv`:

1. identifikátor filmu
2. identifikátor osoby

Najděte odpověď na následující tři otázky:

1. Které osoby se narodily v zadaném roce?
2. Jaké filmy režíroval zadaný režisér?
3. Které filmy mají více jak jednoho režiséra?