České vysoké učení technické v Praze FIT

Programování v Pythonu

Jiří Znamenáček

Příprava studijního programu Informatika je podporována projektem financovaným z Evropského sociálního fondu a rozpočtu hlavního města Prahy.

Praha & EU: Investujeme do vaší budoucnosti

Python – Iterátory

Úvod

Struktury, po kterých se dá iterovat, patří ke zcela základním stavebním kamenům jazyka Python. A iterovat se v Python'u dá opravdu skoro po všem, jak dokumentuje pár následujících příkladů:

```
for znak in "ahoj":
    ...

for prvek in [1, 2, 3, 4, 5]:
    ...

for klíč in {'one': 1, 'two': 2, }:
    ...

for řádka in open('soubor.txt'):
    ...
```

V pozadí za touto pro jazyk zcela základní funkcionalitou stojí tzv. *iterátory*.

Chování iterátorů

Prozkoumejme na příkladu řetězce, co činí iterátor iterátorem:

I. Smyčka *for-in* se stará o „výrobu“ iterátorů tak říkajíc „za běhu“. Například pro řetězec je výsledné chování známé a následující:

```
>>> text = "ahoj"
>>> for znak in text:
...     print(znak)
a
h
o
j
```

II. Ve skutečnosti se na pozadí postupně provedou následující kroky:

```
# Vstupní objekt (zde řetězec), po kterém chceme iterovat:
>>> txt = "ahoj"

# Řetězce mají k dispozici „magickou metodu“ __iter__()..
>>> txt.__iter__
<method-wrapper '__iter__' of str object at 0x00D682A0>
# ..která se používá pro konstrukci iterátoru:
>>> it = iter(txt)
>>> it
<str_iterator object at 0x00DAE030>

# Na iterátorech je pak definována „magická metoda“ __next__()..
>>> it.__next__
<method-wrapper '__next__' of str_iterator object at 0x00DAE030>
# ..která vrací následující prvek:
>>> next(it)
'a'
>>> next(it)
'h'
>>> next(it)
'o'
>>> next(it)
'j'

# Je-li iterátor vyčerpán, obdržíme při pokusu o získání jeho dalš
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    next(it)
StopIteration
```

→ Tedy funkce `iter()` zavolá metodu `__iter__()` daného objektu, zatímco funkce `next()` volá metodu `__next__()` vytvořeného iterátoru.

Smyčka *for-in* tedy na pozadí pomocí funkce `iter()` vytvořila z „předhozeného“ objektu iterátor, po kterém následně pomocí volání `next()` postupně procházela prvek po prvku, dokud nenarazila na výjimku `StopIteration`, na které ukončila svoji činnost.

→ V Python'u 2.x se metoda iterátoru pro vrácení dalšího prvku jmenovala zcela nekonzistentně pouze *next()* (tedy bez podtržítek) a volala se přímo na příslušné instanci.

III. Je zřejmé, že aby se objekt choval jako iterátor, musí definovat metodu *__next__()* pro vrácení dalšího prvku sekvence. Ve většině případů je tato metoda definována přímo na příslušném objektu, pak tudíž stačí, aby metoda *__iter__()* vracela přímo tento objekt.

→ Někdy se může hodit, aby iterování po objektu definoval objekt jiný.

V případě konečného iterátoru se musí objekt postarat i o vrácení výjimky *StopIteration* ve správnou chvíli.

Nekonečný iterátor

Vytvořit iterátor na objektové úrovni tedy zahrnuje trochu víc práce. Pro mnoho aplikací si však vystačíme s použitím generátorů, které se o tvorbu iterátorů starají automaticky. Pro srovnání se podívejme na příklad *nekonečného* generátoru a iterátoru pro generování sudých čísel:

I. Zavedení jako **generátor**:

```
>>> def sudá_čísla():
...     číslo = 0
...     while True:
...         číslo += 2
...         yield číslo

>>> g = sudá_čísla()
>>> next(g)
2
>>> next(g)
4
>>> next(g)
6
```

II. Zavedení jako **iterátor**:

```
>>> class SudáČísla:
...     "Iterátor generující sudá čísla."
...
...     def __init__(self):
...         self.číslo = 0
...
...     def __iter__(self):
...         return self
...
...     def __next__(self):
...         self.číslo += 2
...         return self.číslo

>>> sc = SudáČísla()
>>> sc
<__main__.SudáČísla object at 0x00DC62B0>

>>> it = iter(sc)
>>> it
<__main__.SudáČísla object at 0x00DC62B0>

>>> next(it)
2
>>> next(it)
4
>>> next(it)
6
```

V porovnání velmi jednoduchá generátorová notace v sobě zahrnuje celou mašinérii, kterou je jinak pro vytvoření obdobně se chovajícího objektu potřeba vytvořit. V případě nekonečného generátoru sice odpadá vyvolání výjimky `StopIteration` po vyčerpání dat pro iteraci, ale zavedení obslužných metod iterace (`__iter__()` a `__next__()`) a uložení lokálních proměnných a aktuálního stavu (pomocí atributu `self.číslo`) si musíme zařídit sami.

Konečný iterátor

Nyní už jenom ve zkratce proveďme stejné srovnání pro *konečný* generátor a iterátor sloužící k výpisu sekvence v opačném pořadí:

→ Přímo podle dokumentace.

I. Zavedení jako **generátor**:

```
>>> def reverse(data):
...     for index in range(len(data)-1, -1, -1):
...         yield data[index]
...

>>> for char in reverse('Ahoj!'):
...     print(char)
!
j
o
h
A
```

II. Zavedení jako **iterátor**:

```
>>> class Reverse:
...     "Iterator for looping over a sequence backwards"
...
...     def __init__(self, data):
...         self.data = data
...         self.index = len(data)
...
...     def __iter__(self):
...         return self
...
...     def __next__(self):
...         if self.index == 0:
...             raise StopIteration
...         self.index = self.index - 1
...         return self.data[self.index]

>>> for char in Reverse('Ahoj!'):
...     print(char)
!
j
o
h
A
```

→ Oproti nekonečnému iterátoru zde tedy přibylo hlídání konce sekvence a vyhození výjimky pro tento případ.