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 – Funkcionální prvky v Python'u

Úvod

„Skutečný“ funkcionální jazyk nemá žádné proměnné – jednou přiřazené hodnoty jsou takovými už navždy. Podobně chování funkcí je zcela určeno pouze jejich vstupními parametry a navíc funkce nemění své okolí.

Python nesplňuje žádnou z těchto podmínek. Dokonce byste asi měli docela problém napsat program bez použití proměnných. Ale svým protěžováním iterovatelných sekvenčních typů se pojetí funkcinálních jazyků docela blíží. Dokonce pro práci s nimi nabízí kromě generátorové notace i několik speciálních funkcí.

V dalším se podíváme na několik základních (nejen čistě) funkcionálních prvků jazyka Python.

Generátorová notace

Generátorová notace patří mezi jednu z nejtypičtějších pythoních konstrukcí. Umožňuje velmi snadno na místě jednoho řádku za pomoci členů sekvence vyrábět sekvence (seznamy, množiny a slovníky) jiné.

Pro *seznam* má v principu dvě následující podoby (přičemž *PRVEK* je nejčastěji nějakou funkcí prvků procházené *SEKVENCE*):

```
[ PRVEK for I in SEKVENCE ]
[ PRVEK for I in SEKVENCE if PODMÍNKA ]
```

→ Varianty pro množinu, resp. slovník, se liší pouze ve vnějších závorkách, resp. tvaru prvního členu (nepovinná část s podmínkou pro přehlednost vynechána):

```
{ PRVEK for I in SEKVENCE }
{ KLÍČ:HODNOTA for I in SEKVENCE }
```

map() a spol.

K typicky funkcionálním operacím patří operace prováděné nad celou sekvenční strukturou najednou. Přitom platí, že je-li možné výsledek operace určit dříve, než je prozkoumána sekvence celá, také se tak stane (takže se nepočítají zbytečné kroky).

I. Funkce `any(ITERABLE)` vrací `True`, je-li *alespoň jeden* prvek z iterovatelné struktury vyhodnocen jako pravdivý. Pro prázdnou strukturu vrací `False`:

```
>>> any('ahoj')
True

>>> any([''']*5)
False
```

II. Funkce `all(ITERABLE)` vrací `True`, jsou-li *všechny* prvky z iterovatelné struktury vyhodnoceny jako pravdivé. Pro prázdnou strukturu vrací také `True`:

```
>>> all('ahoj')
True

>>> all(['a', 'h', 0, ''])
False
```

III. Funkce `min(SEKVENCE)` vrací *nejmenší* prvek ze sekvence:

```
>>> min([5, 2, 4, 3, 6])
2
```

→ Pro více vstupních argumentů vrací nejmenší z nich. Nepovinný parametr *key* umožňuje určit řadicí funkci.

IV. Funkce `max(SEKVENCE)` vrací *největší* prvek ze sekvence:

```
>>> max([5, 2, 4, 3, 6])
6
```

→ Pro více vstupních argumentů vrací největší z nich. Nepovinný parametr *key* umožňuje určit řadicí funkci.

V. Funkce `sum(SEKVENCE)` vrací součet prvků v sekvenci:

```
>>> sum([5, 2, 4, 3, 6])
20
```

VI. Funkce `map(FUNKCE, ITERABLE)` vrací iterátor, který postupně mapuje funkci na jednotlivé prvky iterovatelné struktury:

```
>>> map(ord, 'ahoj')
<map object at 0x01406710>
>>> list( map(ord, 'ahoj') )
[97, 104, 111, 106]
```

Často je jednodušší a průhlednější napsat příslušný generátorový výraz *[f(x) for x in xs]*.

VII. Funkce `filter(FUNKCE, ITERABLE)` vrací iterátor, který postupně vrací ty prvky iterovatelné struktury, pro něž zadaná funkce poskytuje `True`:

```
>>> filter(lambda x: x % 2, [5, 2, 4, 3, 6])
<filter object at 0x014085F0>
>>> list( filter(lambda x: x % 2, [5, 2, 4, 3, 6]) )
[5, 3]
```

Pro funkci různou od `None` je často je jednodušší a průhlednější napsat příslušný generátorový výraz *[x for x in xs if f(x)]*.

VIII. Funkce `zip(*ITERABLES)` vrací iterátor, který skládá dohromady do n-tic navzájem si odpovídající prvky z jednotlivých vstupních iterovatelných struktur:

```
>>> zip( 'ahoj', [5, 2, 3, 4] )
<zip object at 0x01400AF8>
>>> list( zip( 'ahoj', [5, 2, 3, 4] ) )
[('a', 5), ('h', 2), ('o', 3), ('j', 4)]
```

→ *zip()* se vyčerpá na nejkratší dostupné sekvenci (potřebujete-li opačné chování, použijte `itertools.zip_longest()`). Pro jednu vstupní sekvenci vrací jednoprvkové n-tice.

Asi už nepřekvapí, že:

```
map(f, filter(p, xs))
  <=>
[f(x) for x in xs if p(x)]
```

Modul „functools“

Další funkcionální prostředky obzvláště pro práci s funkcemi (*higher-order functions*) poskytuje modul *functools*. Podívejme se na výběr z nich:

I. Funkce `reduce(FUNKCE, ITERABLE)` destruktivně aplikuje zadanou binární funkci postupně na dvojce prvků z iterovatelné struktury a vrací výsledek. Příklad je názornější:

```
>>> from functools import reduce

# Vlastně provede ((((5 + 2) + 4) + 3) + 6):
>>> reduce(lambda x, y: x + y, [5, 2, 4, 3, 6])
20
```

→ Případný třetí parametr je předřazen prvkům zpracovávané sekvence jako *inicializační*. Z případné výsledné sekvence o délce 1 se vrací právě jen tento první prvek.

II. Funkce `partial(func, *args, **keywords)` vrací novou funkci (resp. *callable*), kterou vyrobí ze zadané tak, že některé její argumenty „zafixuje“ předanými hodnotami. Příklad bude opět názornější:

```
>>> from functools import partial

# Vlastně zafixuje volání funkce 'int()' ve tvaru 'int(argument, b
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'

>>> basetwo('10010')
18
```

→ Přímo podle dokumentace.

Lambda-funkce

Lambda-funkce jsou anonymní (tj. bezejmenné) funkce definované typicky v rámci jednoho řádku a skládající se pouze z jednoho výrazu, který je vyhodnocován ve chvíli volání funkce:

```
lambda [ARGUMENT[Y]]: VÝRAZ
```

Pythonovské lambda-funkce jsou v porovnání s prakticky libovolnou jinou implementací velmi omezené – mohou obsahovat pouze jeden výraz, a to ještě bez složitějších, natožpak dokonce víceřádkových konstrukcí.

Ve většině případů ale zastanou obdobnou práci „obyčejné“ funkce v Python'u, protože jsou tzv. *first-class object/citizen*. (Což především znamená, že je možné je předávat jako parametry do jiných funkcí, vracet je jako návratové hodnoty z funkcí a ukládat je v datových strukturách.)

Generátory

Naprosto typickým prvkem funkcionálního programování je vytváření (potenciálně i nekonečných) struktur, z nichž se vrací pouze ta část, která je aktuálně vyžadována, a nic dalšího se (zatím) nepočítá.

I. V Python'u se konstruktorům takových objektů říká **generátory**. Vyrábějí se z obyčejných funkcí výměnou `return` za `yield`. Ukažme si nejdříve příklad:

```
>>> def generátor():
...     """Tento generátor je možné zavolat celkem dvakrát."""
...     yield 'první volání'
...     yield 'druhé volání'

>>> for i in generátor():
...     print(i)
první volání
druhé volání
```

Vysvětlení: Každé `yield` „zmrazí“ funkci v aktuálním stavu. Při příštím zavolání generátoru se řízení vrátí přesně do tohoto stavu (narozdíl od funkce, která se „spustí“ znovu od začátku).

Ukažme si poněkud lepší příklad:

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

# ..při použití..
for char in reverse('Ahoj!'):
    print(char)

# ..vrací:
!
j
o
h
A
```

→ Upraveno podle dokumentace.

II. Výhoda generátorů tkví v tom, že mohou být potenciálně i nekonečné, aniž bychom se museli bát o dostupnou paměť – počítá (a vrací) se z nich jenom to, co je v danou chvíli skutečně potřeba:

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

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

→ Samozřejmě si pak musíte dát pozor, abyste se nechytili v nekonečné smyčce ^_^

Generátory (poznámky)

Mezi použitím globální funkce `next()` pro vyvolání dalšího *yield*u a smyčkou `for in` pro proiterování generátoru je několik rozdílů:

- `next()` očekává na vstupu konkrétní instanci generátoru, nikoli jeho konstruktor => kód `next( g() )` bude vracet pokaždé první *yield*; pro nejspíše zamýšlené fungování tedy budete chtít provést něco takovéhoto:

```
g = generátor()
next(g)
```

- podobně zavolání `next()` na ukončený generátor (tj. ten, který už nemá co vrátit) vyvolá výjimku `StopIteration`

  Což je mimochodem právě ta výjimka, která ukončuje vykonávání smyčky *for in*.

Narozdíl od `next()` si tohle všechno smyčka `for in` hlídá sama – na vstup jí můžete poslat rovnou konstruktor generátoru a uvedená výjimka je logicky interpretována jako ukončení cyklu.

Generátorové výrazy

Bude-li výstup generátoru vzápětí „zlikvidován“ v nějaké globální sekvenční funkci (jako třeba *sum()* apod.), může se (zvláště z hlediska paměťových nároků) vyplatit použít místo plného generátoru nebo případné ekvivalentní

generátorové notace tzv. *generátorový výraz*.

Generátorové výrazy se podobají generátorové notaci pro seznamy, jen místo `[]` používají `()`. Pár příkladů přímo z dokumentace:

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> from math import pi, sin
>>> sine_table = {x: sin(x*pi/180) for x in range(0, 91)}

>>> unique_words = set(word  for line in page  for word in line.sp

>>> valedictorian = max((student.gpa, student.name) for student in

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```