

Tipy a triky na používanie funkcionálnych vlastností Pythonu

Jakub Ševcech

@sevo_jakub

Funkcionálny jazyk musí mať aspoň niektoré z týchto vlastností

- + **Funkcie sú objekty** (first class objects): všetko, čo viete spraviť s údajmi, viete spraviť aj s funkciami
- + **Rekurzia** sa používa ako riadiaca štruktúra
- + Zameranie na **spracovanie zoznamov**
- Čisto funkcionálne jazyky sa **vyhýbajú vedľajším účinkom**
- Čisto funkcionálne jazyky **odrádzajú od používania príkazov**
- + Používajú sa **vyššie funkcie** (higher order functions)
- Zaujímať sa o to, **čo** počítame a nie o to **ako**

Python nie je funkcionálny jazyk

- Veľmi ľahko sa dá sklúznúť k nečistým výrazom
- Otvorené triedy
- Nie všetky základné typy sú immutable
- Obmedzená veľkosť zásobníka
- Chýba optimalizácia chvostovej rekurzie
- Pre veľa funkcionálnych črt chýba priama podpora
 - Väčšinou sa ale dajú ohackovať

Má ale veľa funkcionálnych vlastností

- Funkcie sú objekty
- Map, reduce a filter v štandardnej knižnici
- Lambda (obmedzená, ale čo už)
- Generátory, iterátory
- Dekorátory
- Closure
- ...

Ak budeme programovať správne, tak veľa týchto vlastností vieme využiť

- Elegantný kód
- Jednoduché testovanie
- Znovupoužiteľnosť kódu
- Distribuovateľnosť / paralelizovateľnosť výpočtu
- Cachovanie výpočtu
- Zrýchlenie a zmenšenie pamäťových nárokov

FP vs. IP

Class is “data with operations attached” while a closure is “operations with data attached.”

David Mertz: Functional Programming in Python

Rozbitie na menšie, **čisté** funkcie zlepšuje testovateľnosť

```
def get_ssh_options(host):
    def get_value(line, key_arg):
        m = re.search(r'^\s*\s\s+(.+)\s*$'
                      % key_arg, line, re.I)

        if m:
            return m.group(1)
        else:
            return ''

    mydict = {}
    for line in (SSH_CONFIG_FILE):
        line = line.strip()
        line = re.sub(r'#.*$', '', line)
        if not line:
            continue
        if get_value(line, 'Host') != host:
            continue
        if get_value(line, 'Host') == '':
            k, v = line.lower().split(None, 1)
            mydict[k] = v
    return mydict
```

Funkcia sparsuje ~/.ssh/config a vráti dict ssh nastavení pre nejaký host

Rozbitie na menšie, **čisté** funkcie zlepšuje testovateľnosť

```
def get_ssh_options(host):
    def get_value(line, key_arg):
        m = re.search(r'^\s*%s\s+(.+)\s*$'
                      % key_arg, line, re.I)
        if m:
            return m.group(1)
        else:
            return ''

    def remove_comment(line):
        return re.sub(r'#.*$', '', line)

    def not_the_host(line):
        return get_value(line, 'Host') != host

    def not_a_host(line):
        return get_value(line, 'Host') == ''

    lines = [line.strip() for line in (SSH_CONFIG_FILE)]
    comments_removed = [remove_comment(line) for line in lines]
    blanks_removed = [line for line in comments_removed if line]
    top_removed = (itertools.dropwhile(not_the_host, blanks_removed))[1:]
    goodpart = itertools.takewhile(not_a_host, top_removed)
    return dict([line.lower().split(None, 1) for line in goodpart])
```


Vedľajšie efekty sa veľmi rýchlo objavajú pri písaní testov

59

The Local Hero

A test case that is dependent on something specific to the development environment it was written on in order to run. The result is the test passes on development boxes, but fails when someone attempts to run it elsewhere.

The Hidden Dependency

Closely related to the local hero, a unit test that requires some existing data to have been populated somewhere before the test runs. If that data wasn't populated, the test will fail and leave little indication to the developer what it wanted, or why... forcing them to dig through acres of code to find out where the data it was using was supposed to come from.

Sadly seen this far too many times with ancient .dlls which depend on nebulous and varied .ini files which are constantly out of sync on any given production system, let alone extant on your machine without extensive consultation with the three developers responsible for those dlls. Sigh.

[share](#)

[edited Feb 29 '12 at 8:40](#)

[community wiki](#)
[2 revs, 2 users 91%](#)
[annakata](#)

[show 1 more comment](#)

Vedľajšie efekty sa veľmi rýchlo objavia pri písaní testov

58 Chain Gang

A couple of tests that must run in a certain order, i.e. one test changes the global state of the system (global variables, data in the database) and the next test(s) depends on it.

You often see this in database tests. Instead of doing a rollback in `teardown()`, tests commit their changes to the database. Another common cause is that changes to the global state aren't wrapped in `try/finally` blocks which clean up should the test fail.

[share](#)

[edited Feb 29 '12 at 8:41](#)

[community wiki](#)
[4 revs, 2 users 90%](#)
[Aaron Digulla](#)


[show 1 more comment](#)

Čo všetko môže spôsobovať vedľajšie efekty?

- Globálne premenné
- I/O
- Zdieľaný zdroj
- Uchovávanie stavu (v objekte)
- Mutable objekty + predávanie referencie
- ...

Niektorým sa nedá vyhnúť. Čisto funkcionálny program by nevedel zapísať výsledky.

Držte ich v oddelenej časti programu.



Ak budete písať čisté funkcie, tak sa vám budú ľahko používať „Higher order funkcie“

Nemusíte riešiť závislosti a stav.

Výsledok závisí len od vstupu.

Veľmi dobré na spracovanie kolekcií.

„Higher order functions“ zlepšují čitelnost a znovupoužitelnost kódu

```
collection = []  
for item in item_list:  
    partial_result = process_item(item)  
    collection.append(partial_result)
```

```
from operator import add  
from functools import reduce
```

```
collection = map(process_item, item_list)
```

„Higher order functions“ zlepšují čitelnost a znovupoužitelnost kódu

	<pre>from operator import add from functools import reduce</pre>
<pre>collection = [] for item in item_list: partial_result = process_item(item) collection.append(partial_result)</pre>	<pre>collection = map(process_item, item_list)</pre>
<pre>total = 0 for item in item_list: total += item</pre>	<pre>total = reduce(add, item_list)</pre>

„Higher order functions“ zlepšují čitelnost a znovupoužitelnost kódu

	<pre>from operator import add from functools import reduce</pre>
<pre>collection = [] for item in item_list: partial_result = process_item(item) collection.append(partial_result)</pre>	<pre>collection = map(process_item, item_list)</pre>
<pre>total = 0 for item in item_list: total += item</pre>	<pre>total = reduce(add, item_list)</pre>
<pre>collection = [] for item in item_list: if condition(item): collection.append(item)</pre>	<pre>collection = filter(condition, item_list)</pre>

„Higher order functions“ zlepšují čitelnost a znovupoužitelnost kódu

	<pre>from operator import add from functools import reduce</pre>
<pre>collection = [] for item in item_list: partial_result = process_item(item) collection.append(partial_result)</pre>	<pre>collection = map(process_item, item_list)</pre>
<pre>total = 0 for item in item_list: total += item</pre>	<pre>total = reduce(add, item_list)</pre>
<pre>collection = [] for item in item_list: if condition(item): collection.append(item)</pre>	<pre>collection = filter(condition, item_list)</pre>
<pre>collection = [] for item in item_list: if condition(item): new = modify(item) collection.add_to(new) else: new = modify_differently(item) collection.add_to(new) result = [] for thing in collection: new = process(thing) result.add_to(new)</pre>	<pre>def modify_conditionally(item): if condition(item): return modify(item) else: return modify_differently(item) collection = map(modify_conditionally, item_list) result = map(process, collection)</pre>

Python umožňuje rôzne zápisy so **skoro** rovnakým výsledkom

```
total = 0
for x in item_list:
    total += x
```

```
from functools import reduce
def add(x, y):
    return x + y

total = reduce(add, item_list)
```

```
from operator import add
total = reduce(add, item_list)
```

```
total = reduce(lambda x, y: x+y, item_list)
```

```
total = sum(item_list)
```

„Higher order functions“ zlepšují paralelizovatelnost kódu

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    pool = Pool(processes=4)           # start 4 worker processes
    result = pool.apply_async(f, (10,)) # evaluate "f(10)" asynchronously in a single process
    print pool.map(f, range(10))      # prints "[0, 1, 4, ..., 81]"
```

„Higher order functions“ zlepšují paralelizovatelnost kódu

```
import collections
import itertools
import multiprocessing

class SimpleMapReduce(object):

    def __init__(self, map_func, reduce_func, num_workers=None):
        self.map_func = map_func
        self.reduce_func = reduce_func
        self.pool = multiprocessing.Pool(num_workers)

    def partition(self, mapped_values):
        partitioned_data = collections.defaultdict(list)
        for key, value in mapped_values:
            partitioned_data[key].append(value)
        return partitioned_data.items()

    def __call__(self, inputs, chunksize=1):
        map_responses = self.pool.map(self.map_func, inputs, chunksize=chunksize)
        partitioned_data = self.partition(itertools.chain(*map_responses))
        reduced_values = self.pool.map(self.reduce_func, partitioned_data)
        return reduced_values
```

„Higher order functions“ zlepšují paralelizovatelnost kódu

```
import operator
import glob

input_files = glob.glob('*.*rst')
```

```
mapper = SimpleMapReduce(file_to_words, count_words)
word_counts = mapper(input_files)
word_counts.sort(key=operator.itemgetter(1))
word_counts.reverse()
```

```
print(word_counts[:20])
```

```
def count_words(item):
    word, occurrences = item
    return (word, sum(occurrences))
```

```
def file_to_words(filename):
    STOP_WORDS = set(['a', 'an', 'and', 'are', 'as', ... ])
    TR = string.maketrans(string.punctuation, ' ' * len(string.punctuation))

    print multiprocessing.current_process().name, 'reading', filename
    output = []

    with open(filename, 'rt') as f:
        for line in f:
            if line.lstrip().startswith('..'): # Skip rst comment lines
                continue
            line = line.translate(TR) # Strip punctuation
            for word in line.split():
                word = word.lower()
                if word.isalpha() and word not in STOP_WORDS:
                    output.append( (word, 1) )

    return output
```

„Higher order functions“ zlepšují distribuovatelnost kódu - Spark

```
def isprime(n):  
    n = abs(int(n))  
    if n < 2:  
        return False  
    if n == 2:  
        return True  
    if not n & 1:  
        return False  
    for x in range(3, int(n**0.5)+1, 2):  
        if n % x == 0:  
            return False  
    return True
```

```
>>> sc  
<pyspark.context.SparkContext at 0x7fec84f8a150>  
>>> nums = sc.parallelize(xrange(1000000))  
>>> primes = nums.filter(isprime) # 8.51e-05s  
>>> print nums.filter(isprime).count() # 3.91s  
78498  
>>> nums.filter(isprime).take(5) # 0.054s  
[2, 3, 5, 7, 11]  
>>> nums.filter(isprime).takeOrdered(5, key = lambda x: -x) # 2.93s  
[999983, 999979, 999961, 999959, 999953]
```

Immutable objekty zjednodušujú debugovanie

- Ak je chyba, tak v konkrétnej funkcii
- Žiadna iná funkcia nemení objekt predaný referenciou
- Konvencia / vynútenie immutability

Immutable objekt sa nemôže meniť po jeho vytvorení

```
x = 'foo'
y = x
print(x) # foo
y += 'bar'
print(x) # foo
```

```
x = [1, 2, 3]
y = x
print(x) # [1, 2, 3]
y += [3, 2, 1]
print(x) # [1, 2, 3, 3, 2, 1]
```

```
def func(val):
    val += [3, 2, 1]

x = [1, 2, 3]
print(x) # [1, 2, 3]
func(x)
print(x) # [1, 2, 3, 3, 2, 1]
```

Prečo je nemennosť dobrá?

- Netreba počítat' s tým, že sa vám môže hodnota meniť.
- Je to bezpečnejšie, vzniká menej chýb a ľahšie sa debuguje.
- Da sa ľahko zdieľať medzi vláknami.
- Da sa hashovať
- Ak je zoznam nemenný, tak sa da paralelizovať jeho spracovávanie.
- Objekty môžu byť menšie. Zaberajú menej miesta v pamäti a operácie nad nimi sú rýchlejšie.

Ale

- Je treba vytvárať veľmi veľa objektov.
- Garbage collector sa narobí.

Nemennosť zabezpečiť konvenciou alebo vynútiť

```
import pyrsistent as ps
```

```
>>> v1 = ps.v(1, 2, 3)
>>> v2 = v1.append(4)
>>> v3 = v2.set(1, 5)
>>> v1
pvector([1, 2, 3])
>>> v2
pvector([1, 2, 3, 4])
>>> v3
pvector([1, 5, 3, 4])
```

```
>>> m1 = ps.m(a=1, b=2)
>>> m2 = m1.set('c', 3)
>>> m3 = m2.set('a', 5)
>>> m1
pmap({'a': 1, 'b': 2})
>>> m2
pmap({'a': 1, 'c': 3, 'b': 2})
>>> m3
pmap({'a': 5, 'c': 3, 'b': 2})
```

```
>>> from pyrsistent
import PClass, field
>>> class
AClass(PClass):
...     x = field()
...
>>> a = AClass(x=3)
>>> a
AClass(x=3)
>>> a.x
3
```

```
>>> ps.freeze([1, {'a': 3}])
pvector([1, pmap({'a': 3})])
>>> ps.thaw(ps.v(1, ps.m(a=3)))
[1, {'a': 3}]
```

Iterátory sa implicitne používajú vo for cykloch na prechádzanie listov a asociatívnych polí.

```
cities = ["Paris", "Berlin", "Hamburg", "Frankfurt"]
for location in cities:
    print("location: " + location)
```

Iterátor slúži ako odkaz na kontajner. Umožňuje iterovať cez prvky bez toho, aby sme vedeli aká je vnútorná štruktúra kolekcie.

Generátor je špeciálny typ funkcie, ktorá vracia iterátor.

```
def city_generator():
    yield("Paris")
    yield("Berlin")
    yield("Hamburg")
    yield("Frankfurt")

cities = city_generator()
next(cities)
```

Generátor sa dá použiť na vytvorenie nekonečnej dátovej štruktúry

```
def fibonacci():  
    """Fibonacci numbers generator"""  
    a, b = 0, 1  
    while True:  
        yield a  
        a, b = b, a + b  
  
f = fibonacci()  
  
counter = 10  
for x in f:  
    print(x)  
    counter -= 1  
    if (counter < 0): break  
  
list(fibonacci()) # toto radšej netreba spustat
```

Ak výstup funkcie závisí len od vstupu, tak sa dá volanie cachovať.

V niektorých funkcionálnych jazykoch sa o to stará interpreter

V pythone manuálne alebo dekorátorom

Memoization dekorátorom

```
# global_counter = 0
def fib(n):
    # global global_counter
    # global_counter += 1
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def memoize(f):
    memo = {}
    def helper(x):
        if x not in memo:
            memo[x] = f(x)
        return memo[x]
    return helper

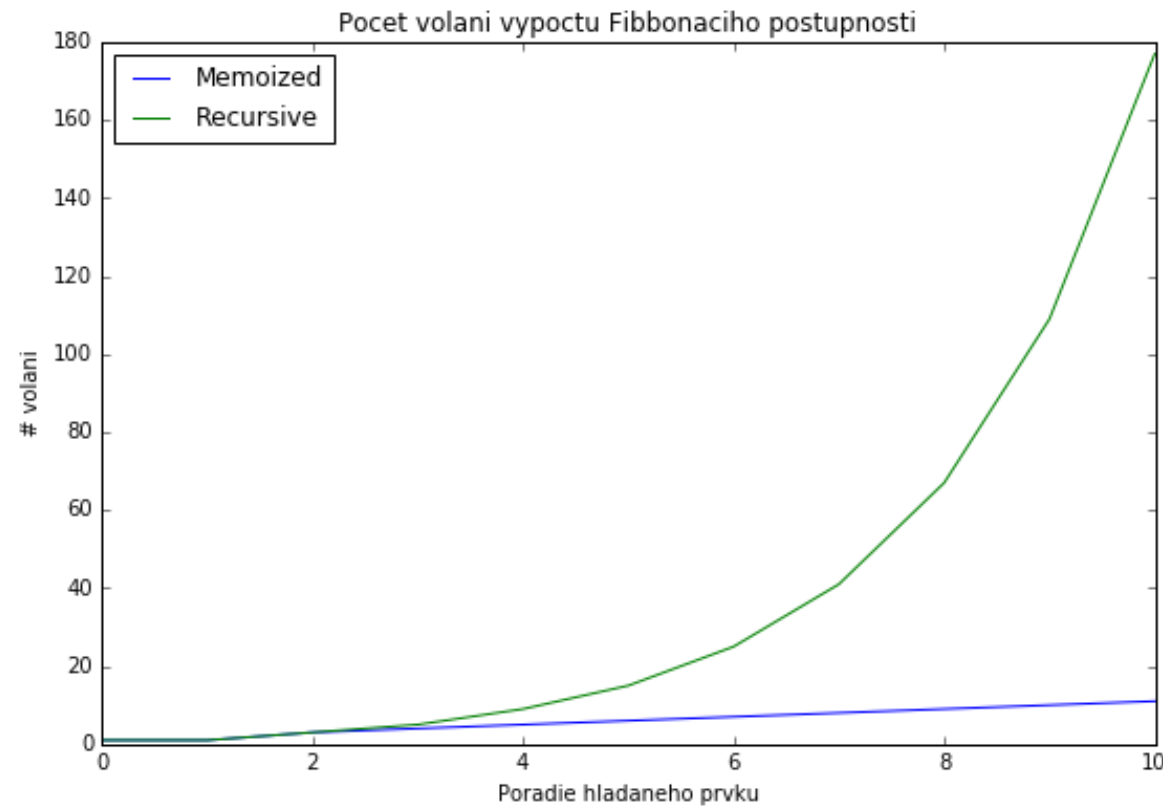
fib = memoize(fib)

fib(15)
```

Dá sa rozšíriť pre ľubovoľný
počet atribútov

x musí byť immutable

Memoization môže znížiť počet volaní funkcie



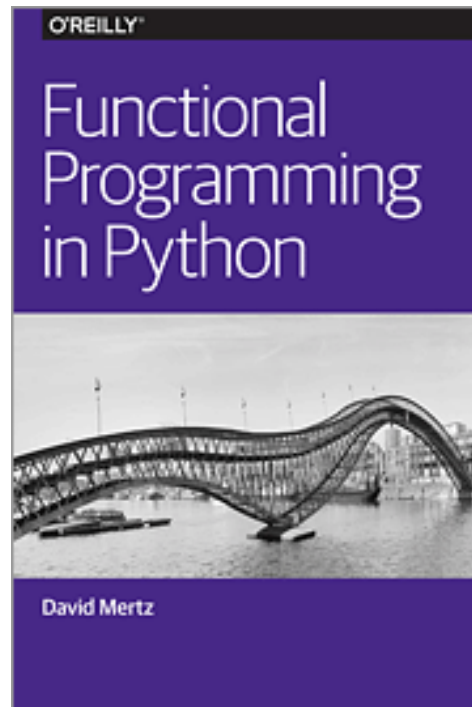
Ďalšie zaujímavé funkcie

- Partial funkcie namiesto preťažovania funkcií¹
- Zdieľanie častí immutable kolekcií na šetrenie miestom^{2,3}
- Generator = > Lazy evaluation
- Dekoratory
- Closure

1) <http://www.ibm.com/developerworks/library/l-pydisp/>

2) <https://github.com/tobgu/pyrsistent>

3) <http://hypirion.com/musings/understanding-persistent-vector-pt-1>



Functional Programming in Python

By [David Mertz](#)

Publisher: O'Reilly

Released: June 2015