Obsah dnesnej prednasky

1. Nemenne objekty a

2. Funkcie vyssej urovne

Sutaz

Project Euler (https://projecteuler.net/)

- Vyriesit co najviac uloh funkcionalne
- Najlepsi dostanu **plny pocet bodov** z Python casti zaverecnej skusky

Elm

Mali ste mat jeden test na prednaske a jeden na cviceni

Minimalne jeden sa nahradi malym projektom.

1. Nemenné (Immutable) objekty

Nemenný objekt sa po vytvorení už nemôže meniť

In [72]:
```
x = 'foo'
print(id(x))
print(id(x.upper()))
print(id(x + 'bar'))
```

```
140314250459208
140314099848448
140314099849568
```

Neznamena to, ze referencia na objekt sa nemoze menit

v cisto funkcionalnom jazyku by sa nemalo diat ani to

In [73]:
```
x = 'foo'
y = x
print(x, id(x))
x = 'bar'
print(x, id(x))# objekt foo sa nezmenil, to len x uz smeruje na iny objekt
print(y, id(y))
```

```
foo 140314250459208
bar 140314250459152
foo 140314250459208
```

Nie je to to iste ako klucove slovo *final* v Jave

Final premenna po vytvoreni nemoze smerovat na iny objekt

Objekt samtny ale moze byt zmeneny

In [ ]:
```
# -- JAVA --
final List<Integer> list = new ArrayList<Integer>();
list = new ArrayList<Integer>(); // toto sa neskompiluje
```

In [ ]:
```
# -- JAVA --
final List<Integer> list = new ArrayList<Integer>();
list.add(1); //toto prejde bez problemov
```

In [ ]:
```
# -- JAVA --
final List<Integer> list = Collections.unmodifiableList(new ArrayList<Integer>(.
```

Imutable znamena, ze hociaka operacia nad objektom vytvori novy objekt namiesto toho aby zmenila ten povodny

In [1]:
```
# retazec je immutable
x = 'foo'
y = x
print(x) # foo
y += 'bar'
print(x) # foo
print(y)
```

```
foo
foo
foobar
```

In [2]:
```
print(id(x))
print(id(y))
```

```
2361298200816
2361332423344
```

In [4]:
```
# zoznam je mutable
x = [1, 2, 3]
y = x
print(x)
y += [3, 2, 1]
print(x)
```

```
[1, 2, 3]
[1, 2, 3, 3, 2, 1]
```

In [5]:
```
print(id(x))
print(id(y))
```

```
2361332410504
2361332410504
```

Pozor, v Pythone sa parametre funkcie predavaju referenciou

Pri mutable objektoch to moze sposobit necakane veci ak neviete, co sa vo funkcii deje

In [76]:
```
def func(val):
    val += 'bar'

x = 'foo' # retazec je immutable, objekt sa nezmeni
print(x)
func(x)
print(x)
```

```
foo
foo
```

In [77]:
```
def func(val):
    val += [3, 2, 1]

x = [1, 2, 3] # zoznam je mutable, zmeni sa premenna mimo bloku funkcie
print(x)
func(x)
print(x)
```

```
[1, 2, 3]
[1, 2, 3, 3, 2, 1]
```

Ak predate immutable objekt funkcii, tak vam ho funkcia urcite nezmeni

String je immutable

Podobne ako vsetky zakladne typy

In [78]:
```
a = 'text'
print(a)
print('Adresa je: {}'.format(id(a)))
```

```
text
Adresa je: 140314297470624
```

In [79]:
```
# Znamena to, ze neviem menit hodnotu
a[0] = 'T'
print(a)
print('Adresa je: {}'.format(id(a)))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-79-309bd94f7dc8> in <module>()
      1 # Znamena to, ze neviem menit hodnotu
----> 2 a[0] = 'T'
      3 print(a)
      4 print('Adresa je: {}'.format(id(a)))

TypeError: 'str' object does not support item assignment
```

List je mutable

In [80]:
```
a = [1,2,3,4,5]
print(a)
print('Adresa je: {}'.format(id(a)))
```

```
[1, 2, 3, 4, 5]
Adresa je: 140314100315464
```

In [81]:
```
# Znamena to, ze neviem menit hodnotu
a[0] = 'T'
print(a)
print('Adresa je: {}'.format(id(a)))
```

```
['T', 2, 3, 4, 5]
Adresa je: 140314100315464
```

Tuple je immutable

In [82]:
```
t1 = (1, 2, 3, 4, 5)
t1
```

Out[82]:
```
(1, 2, 3, 4, 5)
```

In [83]:
```
t1[1]
```

Out[83]:
```
2
```

In [84]:
```
t1[1]=3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-84-ab9dff0930da> in <module>()
----> 1 t1[1]=3

TypeError: 'tuple' object does not support item assignment
```

Nemennost moze komplikovat pracu s objektami

In [85]:
```
t1 = (1, 2, 3, 4, 5)
# Ked chceme update, treba vyrobit novy objekt
t2 = t1[:2] + (17, ) + t1[3:]
t2
```

Out[85]:
```
(1, 2, 17, 4, 5)
```

In [86]:
```
# alebo
l1 = list(t1)
l1[2] = 17
t2 = tuple(l1)
t2
```

Out[86]:
```
(1, 2, 17, 4, 5)
```

In [87]:
```
# vs.
a = [1,2,3,4,5]
a[2] = 17
a
```

Out[87]:
```
[1, 2, 17, 4, 5]
```

Preco je nemennost dobra

Netreba pocitat s tym, ze sa vam moze objekt zmenit

- Je to bezpecnejsie.
- vznika menej chyb
- Lahsie sa debuguje

Lahsie sa testuje

- staci test na jednu funkciu a nie celu skupinu objektov
- Ak testujete fukncniu, ktora meni objekty, tak moze vzniknut viacero testovych pachov

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

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

Toto je dovod, preco ma Test Driven Development (TDD) taky usepch

- Testy sa píšu ešte pred kódom
- Zamýšľate sa ako napísať kód tak, aby bol testovateľný

- Bez toho aby ste o tom vedeli odstraňujete vedľajšie efekty
- Snazite sa o to, aby na sebe funkcie co najmenej zaviseli
- Pripravovanie objektov je pre vas zbytocnou komplikaciou
- Zmena stavu objektu sposobuje, ze musite pisat velmi vela testov aby ste osetrili mnozstvo hranicnych stavov. A kedze sme tvroy lenive, tak nas to prirodzene vedie k tomu, aby sme pisali kod, ktory sa lahko testuje a nepouziva komplikovane obejkty a zavislosti od stavu.

Da sa lahsie zdielat medzi vlaknami a procesmi

- netreba synchronizovat pristup k objektom a stavu

Da sa hashovat

- ak pouzijete premenlivy obejkt ako kluc a zmenite jeho stav, tak aj hodnota hashovacej funkkcie spocitanej z tohto objektu sa zmeni. To znamena, ze by ste po zmene obejktu uz nenasli povodny zaznam v hashovacej tabulke.
- ak pouzijete nemenny objekt ako kluc, tak sa urcite nezmeni a ani hodnota hashovacej funkcie sa urcite nezmeni
- hashovacia funkcia nad nim vzdy vrati rovnaku hodnotu

Objekty mozu byt mensie. Zaberaju menej miesta v pamati a operacie nad nimi mozu byt rychlejsie.

- nepotrebujete reziu na to, aby ste umoznili velke mnzostvo transformacii.

Ale!!!

- Je treba vytvarat velmi vela objektov.
- Garbage collector sa narobi.

In [88]:
```
# inspirovane https://www.youtube.com/watch?v=5qQQ3yzbKp8
employees = ['Jozo', 'Eva', 'Fero', 'Miro', 'Anna', 'Kristina']

output = '<ul>\n'

for employee in employees:
    output += '\t<li>{}</li>\n'.format(employee)
#     print('Adresa outputu je: {}'.format(id(output)))

output += '</ul>'

print(output)
```

```
<ul>
	<li>Jozo</li>
	<li>Eva</li>
	<li>Fero</li>
	<li>Miro</li>
	<li>Anna</li>
	<li>Kristina</li>
</ul>
```

Postupne vytvarame retazec, ktoreho velkost stale rastie a zakazdym sa vytvori novy a novy objekt. Kazdy docasny obejkt sa musi potom odstranit pomocou garbage collectoru. Zostane zachovana referencia len na ten posledny objekt.

Ako zabezpecit nemennost objektov?

- konvencia
- vynutit si ju

S vela vecami si mozeme pomoct kniznicou Pyrsistent

In [89]:
```
import pyrsistent as ps
```

List / Vektor

In [90]:
```
v1 = ps.pvector([1, 2, 3, 4])
v1 == ps.v(1, 2, 3, 4)
```

Out[90]: True

In [91]:
```
v1[1]
```

Out[91]: 2

In [92]:
```
v1[1:3]
```

Out[92]:
```
pvector([2, 3])
```

In [93]:
```
v1[1] = 3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-93-00087ab71557> in <module>()
----> 1 v1[1] = 3

TypeError: 'pvectorc.PVector' object does not support item assignment
```

In [95]:
```
v3 = v1.set(1, 5)
print(v3, id(v3))
print(v1, id(v1))
```

```
pvector([1, 5, 3, 4]) 140314100300968
pvector([1, 2, 3, 4]) 140314100300464
```

Map / dict

In [96]:
```
m1 = ps.pmap({'a':1, 'b':2})
m1 == ps.m(a=1, b=2)
```

Out[96]:
```
True
```

In [97]:
```
m1['a']
```

Out[97]:
```
1
```

In [98]:
```
m1.b # toto s dict nejde
```

Out[98]:
```
2
```

In [99]:
```
print(m1.set('a', 3))
print(m1)
```

```
pmap({'b': 2, 'a': 3})
pmap({'b': 2, 'a': 1})
```

In [100]:
```
print(id(m1), id(m1.set('a', 3)))
```

```
140314100301112 140314100301256
```

Transformacia mutable <=> immutable

In [101]:
```
ps.freeze([1, {'a': 3}])
```

Out[101]:
```
pvector([1, pmap({'a': 3})])
```

In [102]:
```
ps.thaw(ps.v(1, ps.m(a=3)))
```

Out[102]:
```
[1, {'a': 3}]
```

... a dalsie immutable struktury

https://github.com/tobgu/pyrsistent (https://github.com/tobgu/pyrsistent)

- PVector, similar to a python list
- PMap, similar to dict
- PSet, similar to set
- PRecord, a PMap on steroids with fixed fields, optional type and invariant checking and much more
- PClass, a Python class fixed fields, optional type and invariant checking and much more
- Checked collections, PVector, PMap and PSet with optional type and invariance checks and more
- PBag, similar to collections.Counter
- PList, a classic singly linked list
- PDeque, similar to collections.deque
- Immutable object type (immutable) built on the named tuple
- freeze and thaw functions to convert between pythons standard collections and pyrsistent collections.
- Flexible transformations of arbitrarily complex structures built from PMaps and PVectors.

Da sa nieco spravit s tou spotrebou pamati?

Po niektorych operaciach sa objekty dost podobaju

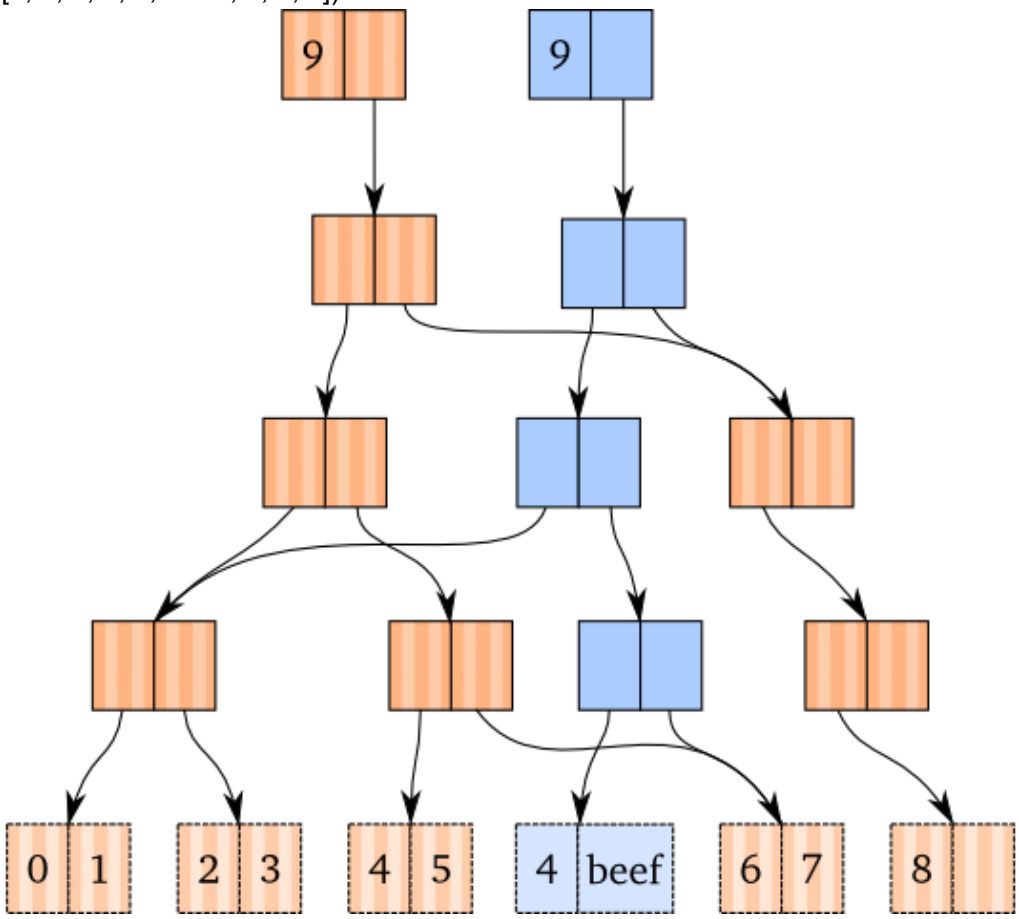
In [103]:
```
v1 = ps.v(0, 1, 2, 3, 4, 5, 6, 7, 8)
print(v1)
v2 = v1.set(5, 'beef')
print(v2)
```

```
pvector([0, 1, 2, 3, 4, 5, 6, 7, 8])
pvector([0, 1, 2, 3, 4, 'beef', 6, 7, 8])
```

Zdielanie casti datovej struktury

pvector([0, 1, 2, 3, 4, 5, 6, 7, 8])

pvector([0, 1, 2, 3, 4, 'beef', 6, 7, 8])



http://hypirion.com/musings/understanding-persistent-vector-pt-1
(http://hypirion.com/musings/understanding-persistent-vector-pt-1)

Nanestastie, Python toto nepodporuje

Niektore funkcionalne jazyky ako napriklad Clojure ale ano.

2. Higher order functions

Funkcional v LISPe (a inych funkcionalnych jazykoch) je funkcia, ktora ma ako argument funkciu alebo funkciu vracia

- FUNCALL - vykonanie funkcie s argumentami
- MAPCAR - zobrazenie
- REMOVE-IF/REMOVE-IF-NOT - filter
- REDUCE - redukcia
- ...

V Pythone a inych jazykoch

- **Funkcia vyssej urovne** (Higher order function) - je funkcia, ktora dostava funkciu ako parameter
- **Generator** - je funkcia, ktora vracia funkciu

Funkcie vyssej urovne sa daju velmi dobre pouzit na spracovanie zoznamu

Minimalne je to ich najcastejsie pouzitie. Casto sa ale pouzivaju aj na ine struktury: napr.: strom

Najcastejsie operacie so zoznamom:

- zobrazenie
- filter
- redukcia

Zobrazenie

Aplikovanie funkcie/transformacie na vsetky prvky zoznamu a vytvorenie noveho zoznamu z transformovanych prvkov

In [7]:
```
def process_item(x):
    return x*x
item_list = [1,2,3,4,5,6]
```

In [8]:
```
# impertivny zapis
collection = []
for item in item_list:
    partial_result = process_item(item)
    collection.append(partial_result)
collection
```

Out[8]:
```
[1, 4, 9, 16, 25, 36]
```

In [10]:
```
#  C-like zapis
collection = [0] * len(item_list) # nahrada mallocu
index = 0
while index < len(item_list):
    partial_result = process_item(item_list[index])
    collection[index] = partial_result
    index += 1
collection
```

Out[10]:
```
[1, 4, 9, 16, 25, 36]
```

Zobrazenie je tak casta operacia, ze ma zmysel

spravit nejaku abstrakciu, aby som to nemusel implementovat stale odznova.

Zobrazenie pomocou funkcie vyssej urovne je prehladnejsie

In [11]:
```
def process_item(x):
    return x*x
item_list = [1,2,3,4,5,6]
```

In [12]:
```
# funkcionalny zapis
collection = map(process_item, item_list)
list(collection)
```

Out[12]:
```
[1, 4, 9, 16, 25, 36]
```

Nezaujima ma ako je `map` implementovane.

Funkcia `map` predstavuje abstrakciu. Ak niekto zmeni implementaciu map, tak ma to niejak neovplyvni. Ak map a ani `process_item` nema ziadne vedlajsie vplyvy (su to ciste funkcie), tak su na sebe uplne nezavisle a mzoem ich menit bez toho aby som menil zvysok kodu.

Viem co chcem dosiahnut a nezaujima ma ako sa to vykona. Deklarujem co chcem dostat a nemusim imperativne hovorit ako to chcem dostat.

Dalsi priklad pouzitia funkcie map

In [19]:
```
def fahrenheit(T):
    return ((float(9)/5)*T + 32)

def celsius(T):
    return (float(5)/9)*(T-32)

temperatures = (36.5, 37, 37.5, 38, 39)
F = list(map(fahrenheit, temperatures))
C = list(map(celsius, map(fahrenheit, temperatures)))
print(F)
print(C)
```

```
[97.7, 98.60000000000001, 99.5, 100.4, 102.2]
[36.5, 37.00000000000001, 37.5, 38.00000000000001, 39.0]
```

Alebo este iny

In [14]:
```
list(map(len, open('data/morho.txt')))
```

Out[14]:
```
[8, 1, 42, 41, 43, 41]
```

In [15]:
```
list(map(print, open('data/morho.txt')))
```

```
Mor ho!


Zleteli orly z Tatry, tiahnu na podolia,

ponad vysokÃ© hory, ponad rovnÃ© polia;

preleteli cez Dunaj, cez tÃº Å¡Ãru vodu,

sadli tam za pomedzÃm slovenskÃ©ho rodu.
```

Out[15]:
```
[None, None, None, None, None, None]
```

Funkcia *map* odstranuje potrebu udrzovat si stav

- nepotrebujem ziadnu kolekciu, ktora je v nejakom case ciastocne naplnena
- nepotrebujem ziadny index, ktory sa inkrementuje
- nestaram sa o to, ako map funguje
  - iterativne, rekurziou, paralelne, distribuovane, pomocou indexu?
- nestaram sa o vnutornu strukturu kolekcie
  - staci aby sa cez nu dalo iterovat (o tomto si povieme viac nabuduce)

Funkcia map by mohla byt implementovana napriklad takto

In [ ]:
```
def my_map(f, seq): # Takto by to mohlo byt v pythone 2 a nie 3. Tam map vracia 
    result = []
    for x in seq:
        result.append(f(x))
    return result
```

Filter

Dalsia velmi casta operacia

Zo zoznamu sa vytvara novy zoznam s tymi prvkami, ktore splnaju podmienku

In [124]:
```
item_list = [1,2,3,4,5,6]
def condition(x):
    return(x % 2 == 0)
```

In [125]:
```
collection = []
for item in item_list:
    if condition(item):
        collection.append(item)
collection
```

Out[125]:
```
[2, 4, 6]
```

Filter pomocou funkcie vyssej urovne

In [ ]:
```
item_list = [1,2,3,4,5,6]
def condition(x):
    return(x % 2 == 0)
```

In [126]:
```
collection = filter(condition, item_list)
list(collection)
```

Out[126]:
```
[2, 4, 6]
```

Dalsi priklad pouzitia funkcie *Filter*

In [ ]:
```
fibonacci = [0,1,1,2,3,5,8,13,21,34,55]
def is_even(x):
    return x % 2 == 0

list(filter(is_even, fibonacci))
```

Redukcia

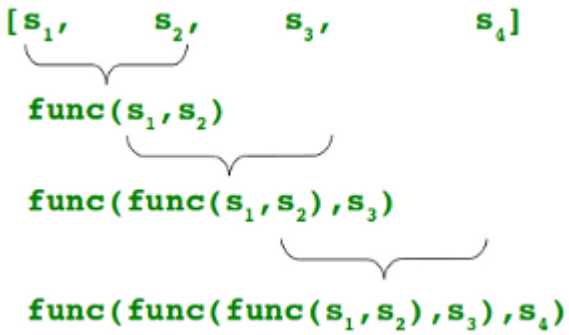
reduce(func, seq, init)

func(a, b)

Opakovane aplikuje funkciu na sekvenciu.

func prijma dva argumenty: hodnotu akumulatora a jeden prvok mnoziny

Atributom *func* moze byt prvok sekvencie alebo navratova hodnota inej *func*



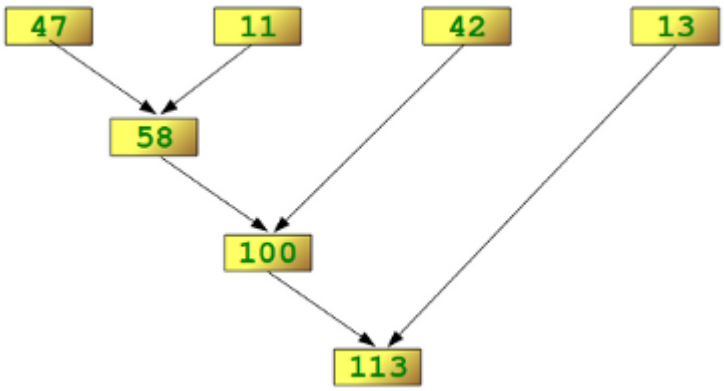
Typicky priklad je suma prvkov zoznamu

In [127]:
```
item_list = [47,11,42,13]
def add(a,b):
    return(a+b)
```

In [128]:
```
from functools import reduce

reduce(add, item_list)
```

Out[128]: 113



In [ ]:
```
total = 0 # Takto by to bolo imperativne
for item in item_list:
    total = add(total, item)
total
```

Dalsi priklad - nasobenie prvkov zoznamu

In [129]:
```
from functools import reduce
def mul(a,b):
    return a * b

reduce(mul, [1,2,3,4,5])
```

Out[129]:
```
120
```

Vela funkcii uz je predpripravenych

In [ ]:
```
from operator import add
```

In [ ]:
```
from operator import mul
```

Da sa spracovavat aj nieco ine ako cisla

In [130]:
```
from functools import reduce
from operator import add

print(reduce(add, open('data/morho.txt')))
```

```
Mor ho!

Zleteli orly z Tatry, tiahnu na podolia,
ponad vysoké hory, ponad rovné polia;
preleteli cez Dunaj, cez tú šíru vodu,
sadli tam za pomedzím slovenského rodu.
```

Da sa napriklad pracovat s mnozinami

In [131]:
```
from operator import or_
reduce(or_, ({1}, {1, 2}, {1, 3}))  # union
```

Out[131]:
```
{1, 2, 3}
```

In [132]:
```
from operator import and_
reduce(and_, ({1}, {1, 2}, {1, 3})) # intersection
```

Out[132]:
```
{1}
```

Lambda funkcia

anonymna funkcia

In [133]:
```
my_sum = lambda x, y: x + y
my_sum(1,2)
```

Out[133]:
```
3
```

- obemdzenie na jediny riadok
- nepotrebuje return

Lambda je celkom prakticka ako parameter funkcie vyssej urovne

In [134]:
```
item_list = [1,2,3,4,5]
print(list(map(lambda x: x**2, item_list)))
```

```
[1, 4, 9, 16, 25]
```

In [135]:
```
item_list = ["auto", "macka", "traktor"]
list(map(lambda x: x.upper(), item_list))
```

Out[135]:
```
['AUTO', 'MACKA', 'TRAKTOR']
```

Spracovanie zoznamu (list comprehension)

In [136]:
```
print(list(map(lambda x: x**2, [1,2,3,4,5])))
print([x**2 for x in [1,2,3,4,5]])
```

```
[1, 4, 9, 16, 25]
[1, 4, 9, 16, 25]
```

In [24]:
```
print(set(map(lambda x: x**2, [1,2,3,4,5])))
print({x**2 for x in [1,2,3,4,5]})
```

```
{1, 4, 9, 16, 25}
{1, 4, 9, 16, 25}
```

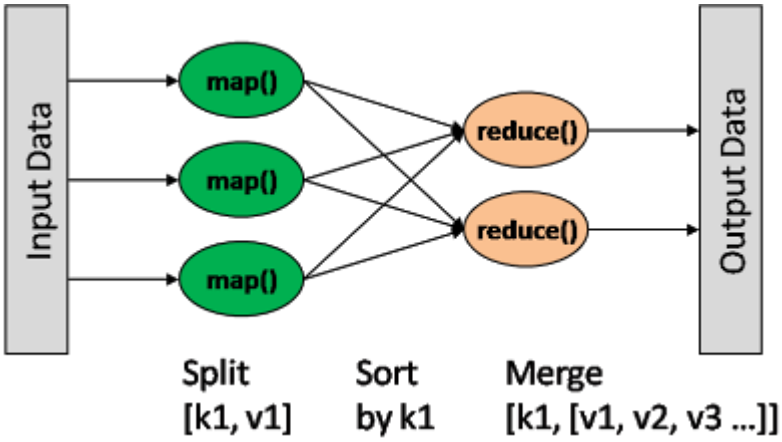
In [27]:
```
print(list(map(lambda x: (x, x**3), filter(lambda x: x % 2 == 0, [1,2,3,4,5]))))
print([(x, x**3) for x in [1,2,3,4,5] if x % 2 == 0])
```

```
[(2, 8), (4, 64)]
[(2, 8), (4, 64)]
```

Na co je to cele dobre - MapReduce

- je programovací model (framework) vyvinutý a patentovaný spoločnosťou Google, Inc. v roku 2004
- hlavným cieľom jeho vývoja bolo uľahčiť programátorom vytváranie dsitribuovaných aplikácií, ktoré spracovávajú veľké objemy dát

- zložité výpočty nad veľkým objemom dát musia byť vykonávané paralelne a dsitribuovane, niekedy až na stovkách alebo tisíckach počítačov súčasne
- pri takomto spracovaní sa treba okrem samotného výpočtu sústrediť napríklad aj na
  - rovnomerné rozdelenie záťaže všetkým dostupným počítačom
  - kontrolovanie výpadkov a porúch spolu s ich následným riešením
- MapReduce prináša ďalšiu vrstvu abstrakcie medzi výpočet, ktorý sa má realizovať paralelne a jeho vykonanie na konkrétnom hardvéri
- Keď napíšem program správne, tak sa nemusím starať na koľkých počítačoch bude bežať



GOTO priklad z netu

Celkom pekny priklad na jednoduchu MapReduce ulohu v Pythone.

Klasicky Word count priklad

http://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/ (http://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/)

--- pseudokod --- function map(String name, String document):

// name: document name

// document: document contents

for each word w in document:

```
    emit (w, 1)
```

function reduce(String word, Iterator partialCounts):

// word: a word

// partialCounts: a list of aggregated partial counts

sum = 0

for each pc in partialCounts:

```
    sum += pc
```

emit (word, sum)

GOTO Spark

Nieco na dalsie studium

- Balicek Operator - https://docs.python.org/3/library/operator.html (https://docs.python.org/3/library/operator.html)
- Balicek Itertools - https://docs.python.org/3/library/itertools.html (https://docs.python.org/3/library/itertools.html)
- Balicek Functools - https://docs.python.org/3/library/functools.html (https://docs.python.org/3/library/functools.html)