Python: Functional Programming

Kai Dührkop

Lehrstuhl für Bioinformatik
Friedrich-Schiller-Universität Jena
kai.duehrkop@uni-jena.de

11.-15. August 2014

Section 1

Functional Programming

- procedural/imperative
  - list of instructions
  - examples: C, Pascal, bash
- declarative
  - description of the problem
  - examples: SQL, HTML, make
- object-oriented
  - stateful objects with methods
  - examples: Smalltalk, Java, C#

- functional
  - decompose problem into a set of functions
  - functions transform input to output
  - functions have no internal state
  - examples: Haskell, Scheme, Clojure

What is an object–oriented programming language?

- objects
- classes
- polymorphism

- encapsulation

What is an object–oriented programming language?

- objects *What is with primitives in java?*
- classes
- polymorphism
- encapsulation

What is an object–oriented programming language?

- objects *What is with primitives in java?*
- classes *What is with prototypes in javascript?*
- polymorphism
- encapsulation

What is an object–oriented programming language?

- objects *What is with primitives in java?*
- classes *What is with prototypes in javascript?*
- polymorphism *Lisp/Scheme have much higher level of polymorphism*
- encapsulation

What is an object–oriented programming language?

- objects *What is with primitives in java?*
- classes *What is with prototypes in javascript?*
- polymorphism *Lisp/Scheme have much higher level of polymorphism*
- encapsulation *modules allow encapsulation, too*

What is a functional programming language?

- functions as first class member
- higher-order functions
- anonymous functions, closures, lambda calculus
- lazyness
- immutable/stateless data structures
- list transformations

functions as first class member

```
def greet:
  print "Hallo Welt"
gruesse = greet
gruesse() # => "Hallo Welt"
viele_gruesse = [gruesse, gruesse, greet]
for gruss in viele_gruesse:
  gruss()
#=> Hallo Welt
#=> Hallo Welt
#=> Hallo Welt
```

Functions are just objects. You can assign them to variables, put them into collections, pass them as parameters.

higher-order functions

```
def greet:
  print "Hallo Welt"
def threetimes(f):
  f()
  f()
  f()

threetimes(greet)
#=> Hallo Welt
#=> Hallo Welt
#=> Hallo Welt
```

higher-order functions

```
def greeter(name):
  def g:
    print "Hallo %s" % name
  return g
greet_world = greeter("Welt")
greet_world() #=> Hallo Welt
```

Functions that get functions as parameters and/or have functions as return type are called **high-order functions**

anonymous functions, lambda calculus

```
def greet(name):
  print "Hello %s" % name
# is equivalent to
greet = lambda name: print "Hello %s" % name

# from earlier example
threetimes(lambda: print "Hallo Welt")
```

lambda allow to define functions without explicitly giving them a name. Use lambda to pass functions directly to other functions. lambda calculus $\lambda x.f(x)$ equivalent to mathematical notation $x \to f(x)$

closures

```
def stack():
  list = []
  def push(x): list.append(x)
  def pop(): return list.pop()
  return (push, pop)

push, pop = stack()
push(5)
pop() #=> 5
```

Anonymous functions that are able to access and modify local variables inside their scope are called **closures**. Be careful: They may lead to memory leaks!

What is a functional programming language?

- functions as first class member
- higher-order functions
- anonymous functions, closures, lambda calculus
- lazyness *later*
- immutable/stateless data structures *nothing special*
- **list transformations**

Section 2

List Transformations

map, filter, reduce

- **map(f,seq)** calls **f** for each item of **seq**
- more than one sequence allowed

calculate x^3 for all items of list

```
def cube(x): return x**3

map(cube, range(4))                    #[0,1,8,27]
```

add items of three lists

```
def add(x, y, z): return x+y+z

map(add, range(4), range(4), range(4))
#[0,3,6,9]
```

- **filter(predicate,seq)** returns a **sequence** of all items for which **predicate** is **True**

get odd numbers from list

```
def odd(x): return x%2 != 0

filter(odd, range(8))          #[1,3,5,7]
```

- **reduce(f,seq)** calls **f** for first two items of **seq**, then on result and next item, etc.
- with start value: called on start value and first, then on result and second, etc.

sum over all items of list

```
def add(x, y): return x+y

reduce(add, range(4))                   #6
reduce(add, range(4), 4)                #10
```

all and any

```
xs = [2,4,6,8]

# check if all values in a collection are true
all(map(lambda x: x % 2 == 0, xs))

# check if any value in a collection is true
any(map(lambda x: x > 5, xs))

# equivalent
ys = map(lambda x: x % 2 == 0, xs)
all(ys) == reduce(lambda a,b: a and b, ys)
any(ys) == reduce(lambda a,b: a or b, ys)
```

Section 2

List Transformations

- **list comprehension** is more readable than **map**

squares

```
[x**2 for x in range(10)]
```

equivalent to map

```
map(lambda x: x**2, range(10))
```

- conditional **list comprehension** with **if** clause
- equivalent to **filter** and subsequent **map**

even squares

```
[x**2 for x in range(10) if x%2 == 0]
```

equivalent to filter + map

```
map(lambda x: x**2,\
    filter(lambda x: x%2 == 0, range(10)))
```

combined list comprehension

```
[x+y for x in [1,2] for y in [1,2]]
#[2, 3, 3, 4]
```

nested list comprehension

```
[[x+y for x in [1,2]] for y in [1,2]]
#[[2, 3], [3, 4]]
```

- also **set** and **dictionary comprehension**

set comprehension

```
{ x**2 for x in range(10) }
#=> returns a set instead of a list
```

dict comprehension

```
{ x: x*x for x in range(1,11) }
#=> {1: 1, 2:4, 3:9, ...}
```

iterator comprehension

```
(x for x in range(1,11))
```

Section 3

Iterators and Generators

Iterators

- iterate over (possibly infinite) values
- lazy
- many functions (zip, map, filter) exists as lazy iterator functions (izip, imap, ifilter)
- package **itertools**

Infinite iterators

```
for i in count(1001, 2):
  if isPrimzahl(i):
    print i
    break
```

count(start, step) returns *infinite* list of all numbers starting from *start* and incrementing by *step*

Infinite iterators

```
cycle([1,2,3,4]) #=> 1,2,3,4,1,2,3,4,1,2,3,4,1,...
# repeats the given sequence indefinitely often

chain([1,2,3,4], [5,6,7,8], [9,10])
#=> 1,2,3,4,5,6,7,8,9,10
# iterate over concatenated sequences

repeat('a', 10) #=> repeat 'a' 10 times
repeat('a') #=> repeat 'a' infinite times
```

Finite iterators

```
dropwhile(lambda x: x < 5, [1,2,3,4,5,6,7,8] )
# skip elements in list
# while predicate is true

takewhile(lambda x: x < 5, [1,2,3,4,5,6,7,8] )
# extract elements from list
# while predicate is true

groupby([1, 2, 1, 'a', 'b', 4.2, 4.3],\
lambda x: type(x))
#=> iterator over (typename, value-iterator)
```

Combinatoric iterators

```
# cartesian product
product(['a', 'b', 'c'], [1,2,3])

# all permutations
list(permutations([1,2,3]))
#=> [(1, 2, 3), (1, 3, 2), (2, 1, 3),
#    (2, 3, 1), (3, 1, 2), (3, 2, 1)]

# all possibilities to draw 3 elements
# from the collection
list(combinations([1,2,3,4], 3))
#=> [(1, 2, 3), (1, 2, 4),
#    (1, 3, 4), (2, 3, 4)]
```

Regexp

```
import re
re.finditer("some regexp", "some string")
#=> iterator over occurences
#   of regexp in string

# parse a molecular formula
{m.group(1): int(m.group(2) or 1) \
 for m in re.finditer("([A-Z][a-z]*)(\d*)", \
 "C6H12O6")}
#=> {"C": 6, "H": 12, "O": 6 }
```

Slicing Iterators

```
# unfortunately, there is no
# iterator[start:to] syntax
iterator = count(0,2)
# take first 10 elements
islice(iterator, 10)
# take elements from 10 to 20
islice(iterator, 10, 20)
# take each second element from 10 to 20
islice(iterator, 10, 20, 2)
```

starmap and izip

```
xs = [1,2,3,4,5]
ys = [4,2,5,7,2]
# classical variant
map(lambda (x,y): pow(x,y), zip(xs, ys))

# shorter:
starmap(pow, zip(xs,ys))
```

Writing own Iterators (Generators)

```
def primes(start=2):
    for i in count(start, 1):
        for j in range(2, int(sqrt(i))):
            if i % j == 0:
                break
        else:
            yield i

for i in primes(100)
    print i # iterates over all primes from 100
```

Writing own Iterators (Generators)

```python
def primes(start=2):
  for i in count(start, 1):
    if all(imap(lambda j: i % j !=0, \
       xrange(2,int(sqrt(i))))): yield i

# return all primes smaller than 100
primes = takewhile(lambda x: x < 100, primes())
```

Imperative style

```
xs = [20, 27, 22, 38, 32, 21]
ys = [17, 20, 12, 18, 22, 39]
for i in xrange(len(xs)):
    print xs[i], ys[i]
```

functional style

```
xs = [20, 27, 22, 38, 32, 21]
ys = [17, 20, 12, 18, 22, 39]
for x, y in zip(xs, ys):
    print x, y
```

Imperative style

```
for i in range(n):
  for j in range(m):
    pass
```

functional style

```
for i, j in product(xrange(n), xrange(m)):
  pass
```

Imperative style

```
for i in range(n):
  for j in range(i, n):
    pass
```

functional style

```
for i, j in combination(xrange(n), 2):
  pass
```

Imperative style

```
occurences = 0
for i in list:
  if predicate(i):
    occurences += 1
```

functional style

```
occurences = len(filter(predicate, list))
# or lazy
occurences = sum(1 for x in ifilter(predicate, list)
```

Take home messages

- **map**, **filter**, **reduce**, **all**, **any**
- iterators and generators (especially **combinate**, **product**)
- **list/set/dict/iterator comprehension**: "filter+map"
- **lambda** for small functions