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 Programming
List Transformations
Iterators and Generators
Common Idioms

- **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"
vielse_gruesse = [gruesse, gruesse, greet]
for gruss in vielse_gruesse:
    gruss()
# => Hallo Welt
# => 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

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

threetimes(greet)
#⇒ Hallo Welt
#⇒ Hallo Welt
#⇒ Hallo Welt
#⇒ Hallo Welt```

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

def greeter(name):
    def g:
        print "Hallo %s" % name
    return g

greet_world = greeter("Welt")
greet_world() #→ Hallo Welt
anonymous functions, lambda calculus

```python
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 $x \rightarrow f(x)$
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

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

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

### add items of three lists

```python
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

```python
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.

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

```
reduce(add, range(4))  # 6
reduce(add, range(4), 4) # 10
```
all and any

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

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

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

# equivalent
ys = map(lambda x: x % 2 == 0, x)
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**

```python
squares

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

equivalent to map

generate lambda = lambda x: x**2

map(lambda x: x**2, range(10))
```
- conditional **list comprehension** with **if** clause
- equivalent to **filter** and subsequent **map**

**even squares**

\[
[x^{**2} \text{ for } x \text{ in } \text{range}(10) \text{ if } x \% 2 == 0]
\]

**equivalent to filter + map**

\[
\text{map}(\lambda x: x^{**2}, \text{filter}(\lambda x: x \% 2 == 0, \text{range}(10)))
\]
combined list comprehension

\[
[ x+y \text{ for } x \text{ in } [1,2] \text{ for } y \text{ in } [1,2] ]
\]

# \([2, 3, 3, 4]\)

nested list comprehension

\[
[[ x+y \text{ for } x \text{ in } [1,2] \text{ for } y \text{ in } [1,2] ]
\]

# \([ [2, 3], [3, 4] ]\)
also set and dictionary comprehension

set comprehension

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

dict comprehension

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

iterator comprehension

```python
(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

```python
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

\[
\text{cycle}([1,2,3,4]) \implies 1,2,3,4,1,2,3,4,1,2,3,4,1,\ldots
\]
\# repeats the given sequence indefinitely often

\[
\text{chain}([1,2,3,4],[5,6,7,8],[9,10]) \implies 1,2,3,4,5,6,7,8,9,10
\]
\# iterate over concatenated sequences

\[
\text{repeat}('a', 10) \implies \text{repeat} 'a' 10 \text{ times}
\]
\[
\text{repeat}('a') \implies \text{repeat} 'a' \text{ infinite times}
\]
Finite iterators

\texttt{dropwhile(\textbf{lambda }x: x < 5, [1,2,3,4,5,6,7,8] ) }
\# \textit{skip} elements in list
\# \textit{while} predicate is true

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

\texttt{groupby([1, 2, 1, 'a', 'b', 4.2, 4.3], \textbf{lambda }x: \textbf{type}(x))}
\#\Rightarrow \textit{iterator over (typename, value–iterator)}
Combinatoric iterators

```python
# 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)]
```
import re
re.finditer("some regexp", "some string")
#→ iterator over occurrences
# of regexp in string

# parse a molecular formula
{m.group(1): int(m.group(2) or 1) \n 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

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

# shorter:
\text{starmap} (\text{pow}, \text{zip}(xs, ys))
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
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

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

functional style

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

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

Functional style

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

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

functional style

```python
for i, j in combinations(xrange(n), 2):
    pass
```
Imperative style

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

Functional style

```python
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 combine, product)
- list/set/dict/iterator comprehension: "filter+map"
- lambda for small functions