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

```python
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

```python
def greet:
    print "Hallo Welt"
def threetimes(f):
    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**.
**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 $x \rightarrow f(x)$
closures

```python
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 \textit{later}
- immutable/stateless data structures \textit{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.

<table>
<thead>
<tr>
<th>sum over all items of list</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>def</strong> add(x, y): <strong>return</strong> x+y</td>
</tr>
<tr>
<td>reduce(add, range(4))</td>
</tr>
<tr>
<td>reduce(add, range(4), 4)</td>
</tr>
</tbody>
</table>
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

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

equivalent to map

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

**even squares**

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

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

<table>
<thead>
<tr>
<th>set comprehension</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{ x**2 for x in range(10) }</code></td>
</tr>
<tr>
<td><code>#=&gt; returns a set instead of a list</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dict comprehension</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{ x: x**x for x in range(1,11) }</code></td>
</tr>
<tr>
<td><code>#=&gt; {1: 1, 2:4, 3:9, ...}</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>iterator comprehension</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(x for x in range(1,11))</code></td>
</tr>
</tbody>
</table>
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 \textit{infinite} list of all numbers starting from \textit{start} and incrementing by \textit{step}.
Infinite iterators

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

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

\[
\text{repeat } ('a', 10) \Rightarrow \text{repeat 'a' 10 times}
\]

\[
\text{repeat } ('a') \Rightarrow \text{repeat 'a' infinite times}
\]
Finite iterators

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

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

\[ \text{groupby}([1, 2, 1, 'a', 'b', 4.2, 4.3],  
\lambda x : \text{type}(x)) \]  
#\Rightarrow \text{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

```python
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) |
  for m in re.finditer("([A-Z][a-z]*)\(\(d*\)\), \n  "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

\[ \begin{align*}
x s & = [1, 2, 3, 4, 5] \\
y s & = [4, 2, 5, 7, 2] \\
\text{# classical variant} \\
\text{map}(\lambda(x, y): \text{pow}(x, y), \text{zip}(xs, ys)) \\
\text{# shorter:} \\
\text{starmap(pow, zip(xs, ys))}
\end{align*} \]
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, range(2, int(sqrt(i))))) : yield i

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