Functional functions in python

map, zip, fold, apply, filter, ...

Lukas Prokop

1st of March 2016

A 10-minutes presentation

Table of contents

Background

Background

The functions we will discuss follow directly from the *Lambda calculus*.

What's that? You know *Turing machines* as computational models? The Church-Turing thesis claims that anything computable by a Turing machine can be computed using Lambda Calculus.

In most simple words: Lambda Calculus is "model everything as function".

Functional programming languages implement the Lambda Calculus. This is why Lambda Calculus is popular among Haskellers, Clojurers, ...

Python

Python is multi-paradigmatic (you can program object-oriented, imperative, functional, etc)[1]

So we can use tools of functional programming in python, right? Yes, but in a limited way (missing lazy evaluation, tail-call optimization, monads, immutable data structures, etc).

Advantages:

1. Easy testable (follows from referential transparency)
2. Mathematically reasonable (useful for formal verification)

Disadvantages:

1. Less popular, therefore high bus factor
2. Purely functional is impossible

[1] If you really want to...

First-class citizens

Python has functions with first-class citizenship. So we can use functions taking functions as arguments (*higher-order functions*). Example:

```
>>> a = [2, -3, 5, 4]
>>> a.sort(key=lambda v: abs(v))
>>> a
[2, -3, 4, 5]
```

Wait? Did I say `lambda`? Yes, as in Lambda Calculus.

List comprehensions

Why lists?

Lambda Calculus likes lists. Why?

We *don't* use mutable data structures where we iterate over all values and manipulate one after another. Or pass them as pointer to a function.

We *do* create a list of values. We pop the top element and apply a function to it. The remaining values constitute a new list. No pointers.

So essentially, this results directly from recursion over iteration a.

List comprehensions in python

A very convenient tool to create new lists.

In mathematics, we have

$$\{\{a, b\} : a \neq b, a, b \in V\}$$

In python, we have:

```
1 V = {1, 3, 5, 6}
2 E = [{a, b} for a in V for b in V if a != b]
```

List comprehensions in python

Simple expressions:

```
ints = [x for x in range(1, 20)]
```

We can do what we will come to know as `map()`:

```
f = lambda v: v**2
mapped = [f(x) for x in range(1, 20)]
```

We can do what we will come to know as `filter()`:

```
import re
# abigail's regex
pred = lambda v: not re.match(r'1?$|^(11+?)\1+$', '1' * v)
filtered = [x for x in range(20) if pred(x)]
```

List comprehensions in python

What about generators?

```
f = lambda x: x * 2
gen = (f(x) for x in range(1, 20))
```

What about dictionary comprehensions?

```
compr1 = dict((x, f(x)) for x in range(1, 20))

# How thinks this works?
compr2 = {x: f(x) for x in range(1, 20)}
```

List comprehensions in python

It does.

PEP 274, Barry Warsaw, 2001 [2]

```
>>> {i : chr(65+i) for i in range(4)}
{0: 'A', 1: 'B', 2: 'C', 3: 'D'}
>>> {(k, v): k+v for k in range(4) for v in range(4)}
{(3, 3): 6, (3, 2): 5, (3, 1): 4, (0, 1): 1, ...}
```

[2] Withdrawn for Python 2.3 but included in Python 2.7 and Python 3.0

List comprehensions in python

And finally set comprehensions:

```
1  a = {x*2 for x in range(20)}
```

Functions

all (conjunction)

`all()` returns true, if – and only if – all values are true.

```
ints = {2, 3, 7, 23}
is_prime = lambda v: not re.match(r'1?$|^(11+?)\1+$', '1' * v)
print(all([is_prime(v) for v in ints]))
```

What about an empty sequence?

```
>>> print(all([]))
True
```

any (disjunction)

`any()` returns true, if – and only if – some value is true.

```
ints = {2, 4, 6, 8}
print(any([is_prime(v) for v in ints]))
```

What about an empty sequence?

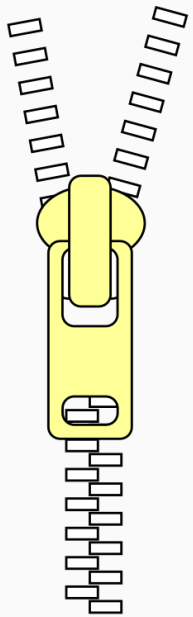
```
>>> print(any([]))
False
```

zip (convolution)

```
"""
class zip(object)
|  zip(iter1 [,iter2 [...]]) --> zip object
|
|  Return a zip object whose .__next__()
|  method returns a tuple where the i-th
|  element comes from the i-th iterable
|  argument.  The .__next__() method
|  continues until the shortest iterable in
|  the argument sequence is exhausted and
|  then it raises StopIteration.
"""
```

zip (convolution)

```
>>> zip("hello", "world")
<zip object at 0x7fb624099dc8>
>>> list(zip("hello", "world"))
[('h', 'w'), ('e', 'o'), ('l', 'r'), ('l', 'l'), ('o', 'd')]
```



It really reminds of a zipper

"A device for temporarily joining two edges of fabric together"

zip (convolution)

It also works for arbitrary many iterables:

```
>>> list(zip({0,3,6}, {1,4,7}, {2,5,8}))
[(0, 1, 8), (3, 4, 2), (6, 7, 5)]
```

min/max/sum

Pretty straightforward: Returns the minimum or maximum value or the sum.

```
>>> min([2, 56, 3])
2
>>> max([2, 56, 3])
56
>>> sum([2, 56, 3])
61
```

min/max/sum

You can also define a key to select an criterion.

```
>>> min([('lukas', 1), ('horst', 9), ('thomas', 3)],
...      key=lambda v: v[1])
('lukas', 1)
>>> max([('lukas', 1), ('horst', 9), ('thomas', 3)],
...      key=lambda v: v[1])
('horst', 9)
```

`sum()` does not, but can take an initial value.

```
>>> sum([1, 4, 6], 100)
111
```

map

```
"""
class map(object)
 |  map(func, *iterables) --> map object
 |
 |  Make an iterator that computes the
 |  function using arguments from each
 |  of the iterables.  Stops when the
 |  shortest iterable is exhausted.
"""
```

map

Apply a function to every value of an iterable:

```
>>> map(lambda v: v + 1, [1, 3, 5, 10])
<map object at 0x7f5939cbef98>
>>> list(map(lambda v: v + 1, [1, 3, 5, 10]))
[2, 4, 6, 11]
```

Corresponds to Visitor pattern in OOP.

filter

```
"""
class filter(object)
 |  filter(function or None, iterable)
 |  --> filter object
 |
 |  Return an iterator yielding those items
 |  of iterable for which function(item)
 |  is true. If function is None,
 |  return the items that are true.
"""
```

filter

```
>>> import re
>>> is_prime = lambda v: not \
...     re.match(r'1?$|^(11+?)\1+$', '1' * v)
>>> filter(is_prime, range(1, 20))
<filter object at 0x7fc3b0750ac8>
>>> list(filter(is_prime, range(1, 20)))
[2, 3, 5, 7, 11, 13, 17, 19]
```

apply

```
>>> apply(lambda v: v + 1, [3])
4
```

Deprecated since version 2.3: Use `function(*args, **keywords)` instead of `apply(function, args, keywords)` ("unpacking").

In Python 3.0 `apply` is undefined.

fold/reduce

Left-folding is called `reduce` in python

```
>>> # computes ((((1+2)+3)+4)+5)
>>> reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
15
```

Python 3.0: Removed `reduce()`. Use `functools.reduce()` if you really need it; however, 99 percent of the time an explicit for loop is more readable.

Guido on those functions

"About 12 years ago, Python aquired lambda, reduce(), filter() and map(), courtesy of (I believe) a Lisp hacker who missed them and submitted working patches. But, despite of the PR value, I think these features should be cut from Python 3000. Update: lambda, filter and map will stay (the latter two with small changes, returning iterators instead of lists). Only reduce will be removed from the 3.0 standard library. You can import it from functools." —Guido van Rossum, 10th of March 2005

Conclusions

Conclusions

Python fundamentally provides some simple primitives which can make your code more concise.

- The `operator` package provides operator methods as functions (unbounded)
- The `functools` package specifically provides features of functional programming
- The `itertools` package provides primitives for generators and combinatorics

The Python documentation call them functional packages.

Curring is provided as `functools.partial`, but I didn't cover that.

Conclusions

Shall I use them?

- If it fits your intuition kindly.
- Don't overdo functional programming in Python!
- Decide yourself whether a function call or list comprehension is more convenient.

Thanks

Thanks for your attention!