Advanced Python

Lambdas and filters and decorators! Oh my!

What you need

- Basic proficiency in Python (or just wing it)
- Your laptop with Python 2.7.x installed
- The seminar setup:
  - http://bit.ly/pyseminar

What we’ll cover

- Lambda expressions
- **`map`**, **`reduce`**, and **`filter`**
- Comprehensions
- Decorators
- **`itertools`** / **`functools`**

Lambda expressions

Lambda expressions (short: lambdas) define short anonymous functions that return a value

```
def name(args):
                       ≈   name = lambda args: expression
    return expression
```

Lambda expressions

Lambda expressions (short: lambdas) define short anonymous functions that return a value

```
def name(args):
                           ≈  name = lambda args: expression
    return expression
```

← look at all this pretty code!

Lambda expressions

Exercise: Define a named function **`make_matrix(n, m)`** that returns a lambda function with one parameter x. The lambda function when called will return an n×m matrix (list of lists) with the value x in every cell.

Hints: what happens when you execute `[6] * 10` in Python?

```
# Example usage
>>> mat_func = make_matrix(2, 3)
>>> my_martix = mat_func(1.5)
>>> seminar.mprint(my_matrix)
[1.5, 1.5, 1.5]
[1.5, 1.5, 1.5]
```

Lambda expressions

Exercise: Define a named function **`make_matrix(n, m)`** that returns a lambda function with one parameter x. The lambda function when called will return an n×m matrix (list of lists) with the value x in every cell.

```
# Possible solution
def make_matrix(n, m):
    return lambda x: [[x] * m] * n
```

```
# Example usage
>>> mat_func = make_matrix(2, 3)
>>> my_martix = mat_func(1.5)
>>> seminar.mprint(my_matrix)
[1.5, 1.5, 1.5]
[1.5, 1.5, 1.5]
```

Seminar setup: http://bit.ly/pyseminar

Map and filter

`map(function, iterable)`

run **`function`** on each item in **`iterable`** and return the results as a list

`filter(function, iterable)`

run **`function`** on each item in **`iterable`** and return a list of items that returned a truth value

Map and filter

Simple stuff!

Exercise: Use **`filter`** on **`seminar.lst`** to return a list that contains only the numbers that are divisible by 3

Map and filter

Simple stuff!

Exercise: Use `filter` on `seminar.lst` to return a list that contains only the numbers that are divisible by 3

```
# Solution
filter(lambda x: not x % 3, seminar.lst)
```

Reduce

`reduce(function, iterable)`

apply **`function(x, y)`** on each the first two items in **`iterable`**, then apply **`function(x, y)`** on the result and on the next item in **`iterable`**, then on the result and the next, … etc until only one value remains. Return this value.

Reduce

```
add = lambda x, y: x + y
reduce(add, [1, 2, 3, 4, 5])
→ reduce(add, [(1+2), 3, 4, 5])
→ reduce(add, [((1+2)+3), 4, 5])
→ reduce(add, [(((1+2)+3)+4), 5])
→ reduce(add, [((((1+2)+3)+4)+5)])
→ ((((1+2)+3)+4)+5) == 15
```

Reduce

← code time

Seminar setup: http://bit.ly/pyseminar

Reduce

No interesting exercise here.

Any ideas of your own? :)

Comprehensions

List comprehension is a syntactic construct for creating a list based on any existing iterable.

Comprehensions

```python
lst = []
for x in iterable:
    if fltr(x):
        lst.append(mp(x))
```

Comprehensions

```
lst = []
for x in iterable:
    if fltr(x):
        lst.append(mp(x))
```

Comprehensions

```
lst = []
for x in filter(fltr, iterable):
    lst.append(mp(x))
```

Comprehensions

```
lst = []
for x in filter(fltr, iterable):
    lst.append(mp(x))
```

Comprehensions

```
lst = map(mp, filter(fltr, iterable))
```

Comprehensions

```
lst = [mp(x) for x in iterable if fltr(x)]
```

Comprehensions

← code time!

Comprehensions

Exercise: Using set comprehension find all the different severity levels that exist in **`seminar.log`**

Comprehensions

Exercise: Using set comprehension find all the different severity levels that exist in **`seminar.log`**

```
# Solution
{ event.severity for event in seminar.log }
```

Decorators

Decorators are functions that take another function as a parameter and return a new functions.

Decorators

Decorators are functions that take another function as a parameter and return a new functions.

What? Why?!

Decorators

```python
def time_it(func):
    def inner_func(*args, **kwargs):
        start = datetime.now()
        result = func(*args, **kwargs)
        print func.func_name, datetime.now() - start
        return result
    return inner_func


def slow_function():
    # do some heavy calculations here
    return True
slow_function = time_it(slow_function)
```

Decorators

```
def time_it(func):
    def inner_func(*args, **kwargs):
        start = datetime.now()
        result = func(*args, **kwargs)
        print func.func_name, datetime.now() - start
        return result
    return inner_func


@time_it
def slow_function():
    # do some heavy calculations here
    return True
```

Decorators

Used for logging, debugging, access control, caching, etc…

← more code!

Decorators

Decorators can accept parameters

```
@authorization_required("ROLE_ADMIN")
def admin_dashboard(request):
    ...
```

In this case you define 3 functions

- `def authorization_required(role)`
  - `def decorator(func)`
    - `def inner_func(*args, **kwargs)`

Decorators

← code again!

Decorators

Exercise: Write a decorator throttle with a parameter max that will only let a function run up to max times, after max times just print “DANGER!”

```
# Example usage
>>> @throttle(2)
... def beetlejuice():
...     return "Beetlejuice!"
...
>>> beetlejuice()
'Beetlejuice!'
>>> beetlejuice()
'Beetlejuice!'
>>> beetlejuice()
DANGER!
>>> beetlejuice()
DANGER!
```

Decorators

Exercise: Write a decorator
throttle with a parameter
will only let a function ru
max times, after max time
print “DANGER!”

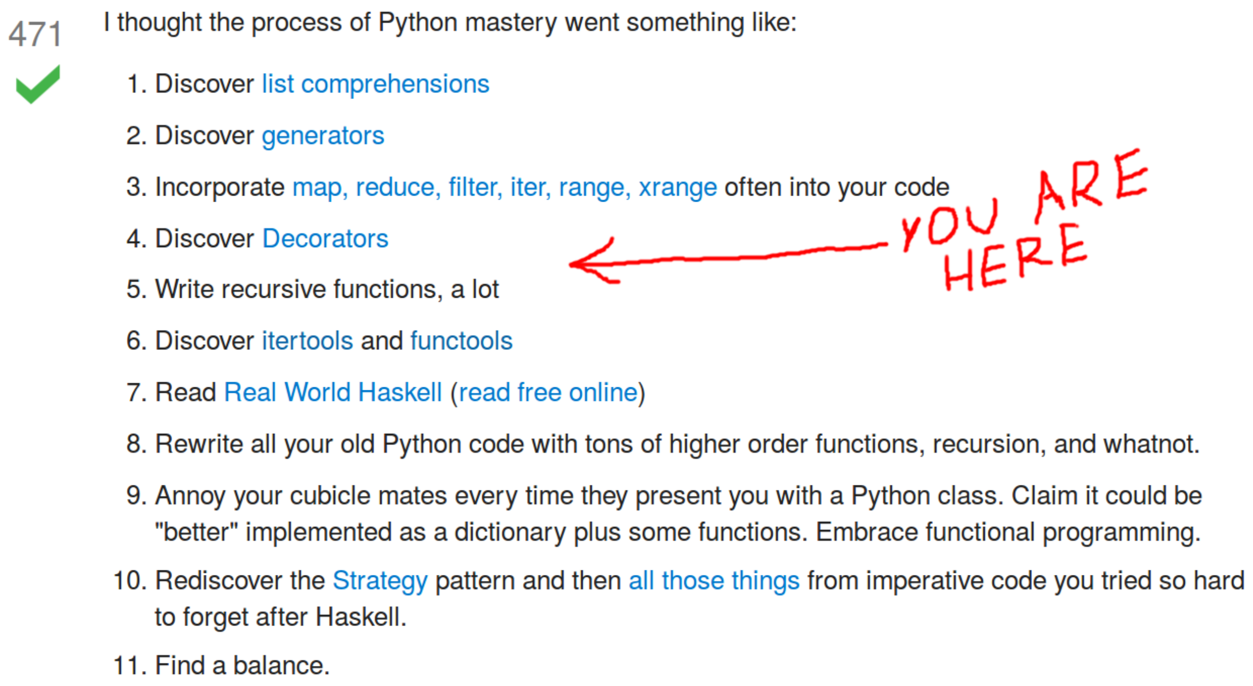
```
# Solution
def throttle(max):
    def decorator(func):
        func.__throttle__ = 0

        def inner_func(*args, **kwargs):
            if func.__throttle__ < max:
                func.__throttle__ += 1
                return func(*args, **kwargs)
            print "DANGER!"
        return inner_func
    return decorator
```

Seminar setup: http://bit.ly/pyseminar

itertools / functools

According to the following serious StackOverflow answer:



471

I thought the process of Python mastery went something like:

1. Discover list comprehensions
2. Discover generators
3. Incorporate map, reduce, filter, iter, range, xrange often into your code
4. Discover Decorators
5. Write recursive functions, a lot
6. Discover itertools and functools
7. Read Real World Haskell (read free online)
8. Rewrite all your old Python code with tons of higher order functions, recursion, and whatnot.
9. Annoy your cubicle mates every time they present you with a Python class. Claim it could be "better" implemented as a dictionary plus some functions. Embrace functional programming.
10. Rediscover the Strategy pattern and then all those things from imperative code you tried so hard to forget after Haskell.
11. Find a balance.

YOU ARE HERE

http://stackoverflow.com/a/2576240/241456

Seminar setup: http://bit.ly/pyseminar

`itertools` / `functools`

`itertools`

This module implements a number of iterator building blocks inspired by constructs from APL, Haskell, and SML.

`functools`

The functools module is for higher-order functions: functions that act on or return other functions. In general, any callable object can be treated as a function for the purposes of this module.

`itertools` / `functools`

Or: where I get lazy and tell you to RTFM

We’re done!

Thank you