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:
  - https://github.com/daniboy/pyseminar

What we'll cover

- Lambda expressions
- `map`, `reduce`, and `filter`
- Comprehensions
- `@decorators`

Seminar setup: https://github.com/daniboy/pyseminar

Lambda expressions

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

```
def name(args):
    return expression
```

≈

```
name = lambda args: expression
```

Seminar setup: https://github.com/daniboy/pyseminar

Lambda expressions

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

```
def name(args):
    return expression
```

≈

```
name = lambda args: expression
```

← look at all this pretty code!

Seminar setup: https://github.com/daniboy/pyseminar

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.

Hint: 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]
```

Seminar setup: https://github.com/daniboy/pyseminar

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: https://github.com/daniboy/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

Seminar setup: https://github.com/daniboy/pyseminar

Map and filter

← code time!

Seminar setup: https://github.com/daniboy/pyseminar

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

Seminar setup: https://github.com/daniboy/pyseminar

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)
```

Seminar setup: https://github.com/daniboy/pyseminar

Reduce

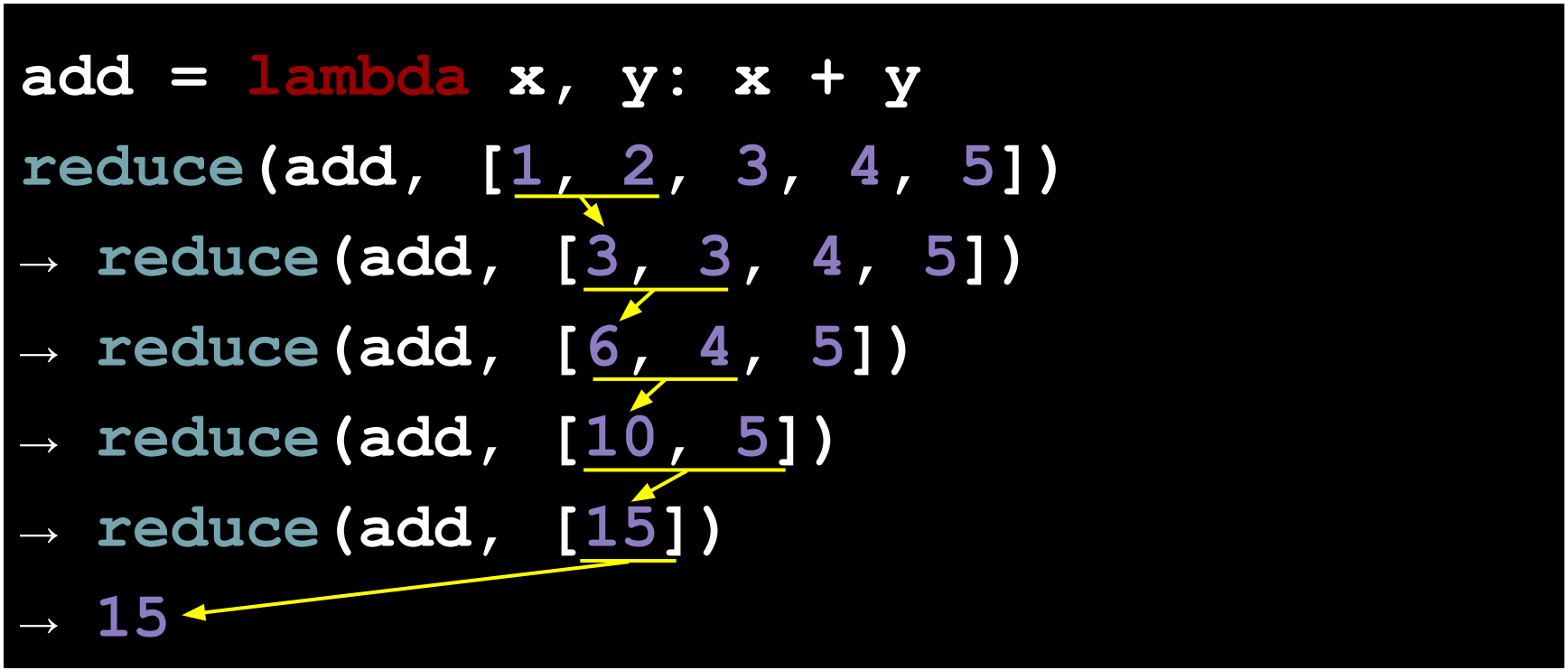
`reduce(function, iterable)`

apply **`function(x, y)`** on each of 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.

Seminar setup: https://github.com/daniboy/pyseminar

Reduce

```
add = lambda x, y: x + y
reduce(add, [1, 2, 3, 4, 5])
→ reduce(add, [3, 3, 4, 5])
→ reduce(add, [6, 4, 5])
→ reduce(add, [10, 5])
→ reduce(add, [15])
→ 15
```



Seminar setup: https://github.com/daniboy/pyseminar

Reduce

← it’s that code thing again

Seminar setup: https://github.com/daniboy/pyseminar

Reduce

Exercise: use **`reduce`** to convert a list of booleans to a *string* of 0's and 1's

```
# Example:
# convert this tuple
(False, True, True, False, True)
# to this string
'01101'
```

hint 1: you can pass a 3rd parameter to **`reduce`**, this parameter is used as the first "**`x`**" before the first item in the tuple

hint 2: `'hello' if True else 'world!'` vs `'hello' if False else 'world!'`

Seminar setup: https://github.com/daniboy/pyseminar

Reduce

Exercise: use **`reduce`** to convert a list of booleans to a *string* of 0's and 1's

```
# Example:
# convert this tuple
(False, True, True, False, True)
# to this string
'01101'
```

```
# Possible solution
my_tuple = (False, True, True, False, True)
func = lambda x, y: x + ('1' if y else '0')
reduce(func, my_tuple, '')
```

Seminar setup: https://github.com/daniboy/pyseminar

Comprehensions

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

Seminar setup: https://github.com/daniboy/pyseminar

Comprehensions

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

Seminar setup: https://github.com/daniboy/pyseminar

Comprehensions

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

Seminar setup: https://github.com/daniboy/pyseminar

Comprehensions

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

Seminar setup: https://github.com/daniboy/pyseminar

Comprehensions

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

Seminar setup: https://github.com/daniboy/pyseminar

Comprehensions

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

Seminar setup: https://github.com/daniboy/pyseminar

Comprehensions

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

Seminar setup: https://github.com/daniboy/pyseminar

Comprehensions

← code time!

Seminar setup: https://github.com/daniboy/pyseminar

Comprehensions

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

Seminar setup: https://github.com/daniboy/pyseminar

Comprehensions

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

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

Seminar setup: https://github.com/daniboy/pyseminar

Decorators

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

Seminar setup: https://github.com/daniboy/pyseminar

Decorators

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

What? Why?!

Seminar setup: https://github.com/daniboy/pyseminar

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


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

Seminar setup: https://github.com/daniboy/pyseminar

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
```

Seminar setup: https://github.com/daniboy/pyseminar

Decorators

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

← more code!

Seminar setup: https://github.com/daniboy/pyseminar

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):`

Seminar setup: https://github.com/daniboy/pyseminar

Decorators

← code again!

Seminar setup: https://github.com/daniboy/pyseminar

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!
```

Seminar setup: https://github.com/daniboy/pyseminar

Decorators

Exercise: Write a decorator throttle with a parameter that will only let a function run max times, after max times 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: https://github.com/daniboy/pyseminar

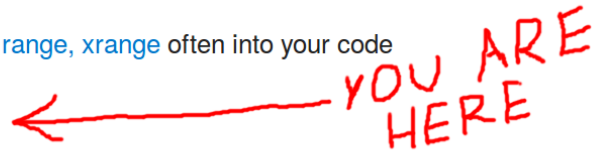
Where to now?

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.



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

Seminar setup: https://github.com/daniboy/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.

Seminar setup: https://github.com/daniboy/pyseminar

`itertools` / `functools`

Now it’s your turn, learn for yourself :)

Seminar setup: https://github.com/daniboy/pyseminar

We’re done!

Thank you

email: rodaniel@cs.ubc.ca twitter: @danielrozenberg