

# Functions & Scope

- Functions
  - Normal
  - Lambda functions
- Scoping
  - LEGB rule
  - Closures

## Functions

Functions in Python:

- Are nestable
- Can use closures
- Cannot be overloaded
- Can accept **and** return variable number of arguments

## Example: Python function

```
In [1]: def calculate_salary(position, nyears, is_manager=False):
        """Calculates the salary of an employee given some criteria.

        Args:
            position (string): One of {'engineer', 'accounts', 'marketing'}
            nyears (int): non-negative number of years at company
            is_manager (boolean): if manager (default=False)
        Returns:
            salary (int): how much employee is paid per year
        Raises:
            No exceptions
        """
        salary = 0
        if position == 'engineer':
            salary += 200000
        else:
            salary += 100000

        salary += (1000 * nyears)
        if is_manager:
            salary += 25000
        return salary

help(calculate_salary)
```

Help on function calculate\_salary in module \_\_main\_\_:

```
calculate_salary(position, nyears, is_manager=False)
    Calculates the salary of an employee given some criteria.

    Args:
        position (string): One of {'engineer', 'accounts', 'marketing'}
        nyears (int): non-negative number of years at company
        is_manager (boolean): if manager (default=False)
    Returns:
        salary (int): how much employee is paid per year
    Raises:
        No exceptions
```

## Return Values

You don't have to return a value, but the default is None!

```
In [15]: def yell():
        print "HEY!"

ret_val = yell()
print ret_val
```

```
HEY!
None
```

## Function Arguments

Argument types:

- Positional (fixed & variable)
- Keyword (fixed & variable)

### Example: Fixed Positional

```
In [5]: import math

def distance(x1, y1, x2, y2):
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)

# test it out
print distance(0, 0, 4, 4)

# what if we don't give enough arguments?
print distance(0, 0, 2, 3)

5.65685424949
3.60555127546
```

### Example: Fixed Keyword

Only accept a whitelist of keys as variable names, but the order does not matter when calling. Keyword arguments also have default values.

```
In [10]: def meal(fruit=None, sandwich='baloney'):
    cost = 0
    if not fruit:
        fruit = 'apple'  # handle default

    # handle cost of fruit
    if fruit == 'banana':
        cost += 0.50
    else:
        cost += 0.75

    # handle sandwich cost
    if sandwich == 'baloney':
        cost += 3.50
    else:
        cost += 4.00
    return cost

# try it out
print meal()
print meal(sandwich='baloney', fruit='banana')
print meal('apple', 'baloney')  # don't have to specify keyword! but you
    must order
print meal(fruit='apple')
print meal(fruit='banana')
```

4.25  
4.0  
4.25  
4.25  
4.0

## Example: Variable Positional

The order doesn't matter, and we can have as many as we'd like!

```
In [21]: def add_numbers(*args):
    total = 0
    for arg in args:
        total += arg
    return total

print add_numbers(5)
print add_numbers(1, 2, 3)
print add_numbers(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
```

5  
6  
45

## Example: Variable Keyword

The order doesn't matter, and we can have as many as we'd like.

```
In [11]: def output_key_value_pairs(**kwargs):
          for keyword, value in kwargs.items():
              print keyword, "=>", value

          output_key_value_pairs(color='purple', amount='14', total='50')

          color => purple
          amount => 14
          total => 50
```

## Example: Variable Keyword & Positional

```
In [12]: def do_something(*args, **kwargs):
          total = 0
          for arg in args:
              total += arg
          for key, value in kwargs.items():
              print key, "->", value
          return total

          # try it out
          do_something(1, 2, 3, color='purple', dwelling='house')

          color -> purple
          dwelling -> house
```

Out[12]: 6

## Variable, Fixed, Keyword, *and* Positional ?!

Yes. We. Can.

```
In [13]: def do_something(fixed1, fixed2, fixed3=None, fixed4=10, *args,
                        **kwargs):

    print "Fixed positional arguments:", fixed1, fixed2
    print "Fixed keyword arguments:", fixed3, fixed4

    total = 0
    for arg in args:
        total += arg

    for key, value in kwargs.items():
        print key, "->", value
    return total

# try it out
print do_something(1, 2, 3, 4, 5, 6, color='purple', dwelling='house')
# print do_something(1, 2, 3, color='blue')
```

Fixed positional arguments: 1 2  
Fixed keyword arguments: 3 4  
color -> purple  
dwelling -> house  
11

## Method overloading in Python

Cannot be explicitly done, must be a single function.

```
In [ ]: def some_function(switch=None):
        if switch == 'value':
            # ... etc
        pass
        # ... etc
        pass
```

## Lambda Functions

Anonymous Python functions created at runtime.

```
In [17]: # with a single argument
square = lambda x: x**2
print square(4)
print square(9)

# with multiple arguments
subtract = lambda x, y: x - y
print subtract(10, 3)
```

16  
81  
7

## Why would we want lambda functions?

- They are useful in cases where a full-blown function (or series of many) would clutter your module and/or aren't needed outside a function
- When a function is only needed *once*
- For sorting, map, filter, reduce (we'll cover those later!)

## Why wouldn't we?

- Cannot be serialized
- Cannot be reused

## Scoping & Closures

What is `foo`? What is the value of `A()`?

To answer these questions we'll have to talk about how Python handles scope, and how that allows us to use function closures.

```
In [4]: foo = 0

# a convoluted class
def A():
    foo = 1
    def B():
        foo = 2
        def C():
            return foo
        return C()
    return B()

# print A() # ?
# print foo # ?
```

## Not in Kansas Anymore

```
In [58]: # defining things at module level! ouch
for i in range(10):
    variable = 'apple'

# not in Java or C land anymore...
print i
print variable

9
apple
```

## Scoping in Python: LEGB rules

Python follows the LEGB rule for scoping (<http://www.amazon.com/dp/0596513984>), which means when Python encounters a name, the search is conducted as follows in order:

1. **L**ocal
  - A. Names assigned in a `def`, and not declared `global`
2. **E**nclosing function locals
  - A. Name in scope of any (and all) enclosing functions
  - B. Goes inner to outer
3. **G**lobal (module)
  - A. Names at top level of module file
  - B. a global declaration in a `def`
4. **B**uilt-in (Python)
  - A. `open`
  - B. `for`
  - C. `etc`

## Back to the Question!

Take two minutes individually and come up with your answer. Don't code it out - try to think through it.

```
In [19]: foo = 0
```

```
#####  
# Remember: 1) Local, then 2) Enclosing functions,  
#           then 3) Global, then 4) Built-in  
#####  
def A():  
    foo = 1  
    def B():  
        foo = 2  
        def C():  
            return foo  
        return C()  
    return B()  
  
print A() # ?  
print foo # ?
```

```
2  
0
```



# Closures

Because we've seen that functions can remember variables that were in their scope, we can create closures in Python.

Closures in any language allow us to create function "factories" - functions inside of functions.

(Insert Inception joke here)

## Example: Closure

What does this code do? What variable & LEGB rule enables this closure?

```
In [25]: def adder_factory(base):  
         def inner(x):  
             return base + x  
         return inner  
  
adder = adder_factory(10)  
print adder(1)  
print adder(1)
```

```
11  
11
```

## Example: Closure with Lambda

We're still returning an inner function which keeps base in its scope.

```
In [27]: def adder_factory_lambda(base):  
         return lambda x: base + x  
  
adder = adder_factory_lambda(7)  
print adder(2)
```

```
9
```

**So, yes, you can do this...**

```
In [49]: inception = lambda base: \
           lambda x: \
               base + x

# try it out
add = inception(100)
print add(5)
print add(7)

105
107
```

But **don't!**

## Lab: Coding Exercises

Fill in the method definitions in the file `exercises/functions.py`.

Make sure you can pass tests with:

```
$ py.test tests/test_functions.py::FunctionExercises:<function_name> # test single function
$ py.test tests/test_functions.py::FunctionExercises                  # test all at once
```

## Wrap-Up

- **Functions**
  - Normal
    - Fixed & Variable
    - Positional & Keyword
  - lambda functions
- **Scoping**
  - LEGB rule
  - Closures
    - Usage
    - Using lambdas