Lecture IV
Functions & Scope

- Defining Functions
- Returning
- Parameters
- Docstrings
- Functions as Objects
- Scope
- Dealing with Globals
- Nesting Scopes

Defining Functions

- Functions in Python are defined using the `def` keyword. The types of parameters or returned values are not specified.
- The code inside a function is compiled but not executed until the function is called, so type and undefined variable errors are not caught.
- Functions are not defined until execution reaches the `def` keyword and cannot be used before that.
- Functions can be nested inside each other.

python

Defining Functions

- Examples:

  -
    ```
    def greet():
         print 'hello!'

    def say(x):
         print x

    greet()
    say('How are you?')
    say('These are Python functions!')
    greet()
    ```

Returning

- Functions can return any object.
- A function not containing a return statement or containing a return statement without a value returns `None` by default.
- Functions can return multiple values by returning a tuple or list containing them.

Returning

- Examples:

  -
    ```
    def get5():
        return 5

    def getNothing():
        pass

    print get5()
    print getNothing()

    def add(x, y):
        return x + y

    print add(5, 6)
    print add(1, 211)
    ```

Returning

- Examples:

  -
    ```
    def getHigherAndLower(x):
        return x-1, x+1

    def getPowers(x, n):
        return [x**i for i in range(1, n+1)]

      a, b = getHigherAndLower(5)
      a → 4     b → 6

      w, x, y, z = getPowers(3, 4)
      w → 3     x → 9     y → 27     z → 81

      x, y, z = getPowers(5, 3)
      x → 5     y → 25     z → 125
    ```

python

Function Parameters

- Functions can take any number of parameters, either positional or keyword.
- Normal parameters are both positional and keyword. The user of the function can call it by specifying either a sequence of parameters or a set of parameter name/value pairs.
- Functions can also accept a variable number of parameters, keyword or positional, using special syntax.
- Parameters can take default values.

python

Function Parameters

- Examples:

  -
    ```
    def greet(name, is_formal):
            if is_formal:
                print 'Greetings,', name
            else:
                print 'Hey,', name

      greet('Jack', False)
      greet('Mr. Doe', True)
      greet('Mr. Doe', is_formal=True)
      greet(name='Mr. Doe', is_formal=True)
      greet(is_formal=True, name='Mr. Doe')
      greet(is_formal=False, name='Jack')
    ```

Function Parameters

- Examples:

  -
    ```
    def greet(name, is_formal=True):
        if is_formal:
            print 'Greetings,', name
        else:
            print 'Hey,', name

    greet('Jack', False)
    greet('Mr. Doe', True)
    greet('Mr. Doe', is_formal=True)
    greet(name='Mr. Doe', is_formal=True)
    greet(is_formal=True, name='Mr. Doe')
    greet(is_formal=False, name='Jack')
    greet('Jack')
    greet('Mr. Doe')
    ```

Function Parameters

- Examples:

  -
    ```
    def calculate(x, power=1, multiplier=1, extra=0):
        return multiplier * (x ** power) + extra

      print calculate(5)                          → 5
      print calculate(5, multiplier=3)            → 15
      print calculate(5, extra=4)                 → 9
      print calculate(5, power=2, extra=3)        → 28
      print calculate(3, extra=10, power=2)       → 19
    ```

Function Parameters

- Examples:

  -
    ```
    def add(*nums):
         sum = 0
         for i in nums:
             sum += I
         return sum

    print add(1, 4, 10)         → 15
    print add(3)                → 3
    print calculate(4.5, 6)     → 10.5

    x = [3, 30, 300]
    print add(*x)               → 333
    print add(*[5, 3, 2])       → 10
    ```

python

Function Parameters

- Examples:

  -
    ```
    def showGradeTable(name, **subjects):
         print name
         sum = 0
         for subject, mark in subjects.items():
             print '%s: %s' % (subject, mark)
             sum += mark
         print 'Average:', float(sum) / len(subjects)

    showGradeTable('Mary', c=68, math=90, english=83,
                      statistics=87)

    showGradeTable('Jim', arabic=50, art=95,
    marketing=80)
    ```

python

Lambdas

- Lambdas are an alternative way of defining functions. A lambda expression returns a function object which you can use directly or assign to a variable. Examples:

  -
    ```
    f = lambda x: x*x
    g = lambda x, y: (x + y) / 2.0
    print f(5)                    → 25
    print f(10)                   → 100
    print g(5, 6)                 → 5.5
    print (lambda a: 2*a)(5)      → 10
    ```

Functions as Objects

- Remember that everything in Python is an object. That includes functions. You can assign them to variables, return them, pass them to other functions, etc. Examples:

  -
    ```
    def f(x):
            return 5 + x

    g = f

    print f(4)    → 9

    print g(4)    → 9
    ```

Functions as Objects

- Examples:

  -
    ```
    x = ['abcz', 'dy', 'fex']

    print sorted(x)
    x → ['abcz', 'dy', 'fex']

    print sorted(x, key=lambda s: len(s))
    x → ['dy', 'fex', 'abcz']

    print sorted(x, key=lambda s: s[::-1])
    x → ['fex', 'dy', 'abcz']

    print sorted(x, key=lambda s: s[1])
    x → ['abcz', 'fex', 'dy']
    ```

Functions as Objects

- Examples:

  -
    ```
    def getAdder(x):
            def adder(n):
                 return n + x
            return adder

       f = getAdder(5)
       g = getAdder(10)
       h = getAdder(30)

       print f(10)    → 14
       print g(10)    → 20
       print h(10)    → 40
    ```

Scope

- When variables come into existence (by being assigned a value), they are bound to the current scope.
- For all code outside functions and classes, the scope is "global". That is, variables defined there are visible everywhere in the current module.
- For code inside functions or classes, the scope is local to the current function or class.

Scope

- Example:

  -
    ```
    x = 10
    def f():
        y = 5
        print x    → 10
        print y    → 5
    def g():
        print x    → 10
        print y    → ERROR
    print x    → 10
    print y    → ERROR
    ```

Scope

- Variables defined in local scopes eclipse global variables.
- It does not matter at which point in the function the variable is defined - if it has the same name as a global, it will override it everywhere in the function.
- Globals cannot be modified from inside functions by default. To assign to them, the function must explicitly define the variables it wants to change using the `global` keyword.

python

Scope

- Examples:

  -
    ```
    x = 10
    def f():
        print x      → 10
    def g():
        x = 20
        print x      → 20
    def h():
        print x      → ERROR
        x = 20
    f()
    g()
    h()
    print x      → 10
    ```

python

Scope

- Examples:

  -
    ```
    x = 10
    def f():
        x = 20
    def g():
        global x
        x = 30

    print x    → 10
    f()
    print x    → 10
    g()
    print x    → 30
    ```

Nesting Scope

- Python scopes can be nested. Functions defined inside other functions have their own scope while being able to access (but not modify!) variables in their parent scope.
- The outer function, in such case, can't access variables from the inner function's scope.

Nesting Scope

- Examples:

  -
    ```
    def f():
        x = 5
        def g():
            x = 10
            y = 20
            print x    → 10
            print y    → 20
        print x    → 5
        print y    → ERROR
    f()
    ```

python

Nesting Scope

- Examples:

  -
    ```
    def f():
         x = 5
         def g():
             global x      → this does nothing here.
             x = 10
             y = 20
             print x       → 10
             print y       → 20
         print x           → 5
         print y           → ERROR
     f()
    ```