CME 193: Introduction to Scientific Python

Winter 2017

Lecture 2: Functions and Lists

Blake Jennings

`stanford.edu/~bmj/cme193`

Contents

Course outline

- Exercises each lecture; aggregate portfolio due at the end of the four weeks.
- Homework 1 required
- Option: Homework 2 or project

Homework

- I will post Homework 1 by Tuesday
- Everyone will need to complete Homework 1
- Due the following Tuesday **(1/31)**.
- Homework 2 will also be posted that day, which will be due 2/9.

Project

- Chance for you to work on something you enjoy, to use Python for something that interests you
- Some ideas/sources for data will be on the course website
- Projects need to be done individually
- If you want to do a project, please submit a brief proposal (1-2 paragraphs) on what you want to do - due by Tuesday (1/31)
- Final project will be due 2/9 as well

Contents

Simple example

Example: Suppose we want to find the circumference of a circle with radius 2.5. We could write

```
radius = 2.5
circumference = 2 * math.pi * radius
```

What if we want to compute this value for many different sized circles?

Functions

Functions let us re-use the same code to complete the same task many times.

Functions are used to abstract components of a program.

Much like a mathematical function, they take some input and then do something to find the result.

Python (and any package we will use) have many built-in functions.

Functions: def

Start a function definition with the keyword `def`

Then comes the function name, with arguments in braces, and then a colon

```
def func(arg1, arg2):
```

Functions: def

Start a function definition with the keyword `def`

Then comes the function name, with arguments in braces, and then a colon

```
def func(arg1, arg2):
```

Functions: body

Then comes, indented, the body of the function

Use `return` to specify the output

```
return result
```

```
def calc_circumference(radius):
    circumference = 2 * math.pi * radius
    return circumference
```

Functions: body

Then comes, indented, the body of the function

Use `return` to specify the output

`return result`

```
def calc_circumference(radius):
    circumference = 2 * math.pi * radius
    return circumference
```

How to call a function

Calling a function is simple (i.e. run/execute):

```
>> func(2.3, 4)
```

Quick question

What is the difference between `print` and `return`?

Exercise

1. Write a function that prints ‘Hello, world!’
2. Write a function that returns ‘Hello, `name`!’, where `name` is a variable

Exercise solution

```
def hello_world():
    print 'Hello, world!'

def hello_name(name):
    return 'Hello, ' + name + '!'
```

Return

By default, Python returns None

Once Python hits `return`, it will return the output and jump out of the function

```
def loop():
    for x in xrange(10):
        print x
        if x == 3:
            return

def hello_name(name):
    print 'Hello,', name + '!'

def hello_world():
  print 'Hello, world!'

hello_world()
hello_name('cme193 class')

loop()
```

What does this function do?

Return

By default, Python returns None

Once Python hits `return`, it will return the output and jump out of the function

```
def loop():
    for x in xrange(10):
        print x
        if x == 3:
            return

def hello_name(name):
    print 'Hello,', name + '!'

def hello_world():
  print 'Hello, world!'

hello_world()
hello_name('cme193 class')

loop()
```

What does this function do?

Everything is an object

Everything is Python is an object, which means we can pass functions:

```
def twice(f, x):
    ''' Apply f twice '''
    return f(f(x))
```

Scope

Variables defined within a function (local), are only accessible within the function.

```
x = 1

def add_one(x):
    x = x + 1  # local x
    return x

y = add_one(x)
# x = 1, y = 2
```

Functions within functions

It is also possible to define functions within functions, just as we can define variables within functions.

```
def function1(x):
    def function2(y):
        print y + 2
        return y + 2

    return 3 * function2(x)

a = function1(2)    # 4
print a             # 12
b = function2(2.5)  # error: undefined name
```

Global keyword

We could (but should not) change global variables within a function

```
x = 0

def incr_x():
    x = x + 1  # does not work

def incr_x2():
    global x
    x = x + 1  # does work
```

Question: What is the difference between the last two functions?

Scope questions

```
def f1():
    global x
    x = x + 1
    return x

def f2():
    return x + 1

def f3():
    x = 5
    return x + 1

x = 0
print f1() # output? x?
print x
print f2() # output? x?
print x
print f3() # output? x?
print x
```

Default arguments

It is sometimes convenient to have default arguments

```
def func(x, a=1):
    return x + a

print func(1)     # 2
print func(1, 2)  # 3
```

The default value is used if the user doesn't supply a value.

More on default arguments

Consider the function prototype: `func(x, a=1, b=2)`

Suppose we want to use the default value for a, but change b:

```
def func(x, a=1, b=3):
    return x + a - b

print func(2)        # 0
print func(5, 2)     # 4
print func(3, b=0)   # 4
```

Docstring

It is important that others, including *you-in-3-months-time* are able to understand what your code does.

This can be easily done using a so called 'docstring', as follows:

```
def nothing():
    """ This function doesn't do anything. """
    pass
```

We can then read the docstring from the interpreter using:

```
>> help(nothing)
```

This function doesn't do anything.

Docstring

It is important that others, including *you-in-3-months-time* are able to understand what your code does.

This can be easily done using a so called 'docstring', as follows:

```
def nothing():
    """ This function doesn't do anything. """
    pass
```

We can then read the docstring from the interpreter using:

```
>> help(nothing)
```

This function doesn't do anything.

Question

```
def nothing():
    """ This function doesn't do anything. """
    pass
```

Question: what does `nothing()` return?

Lambda functions

An alternative way to define short functions:

```
cube = lambda x: x*x*x

print cube(3)
```

Pros:

- One line / in line
- No need to necessarily name a function

Can be useful in the exercises.

Most useful if you only need a function once, so quick to write.

Contents

Importing a module

We can import a module by using `import`

E.g. `import math`

We can then access everything in `math`, for example the square root function, by:

```
math.sqrt(2)
```

Importing as

We can rename imported modules

E.g. `import math as m`

Now we can write `m.sqrt(2)`

In case we only need some part of a module

We can import only what we need using the `from ... import ...` syntax.

E.g. `from math import sqrt`

Now we can use `sqrt(2)` directly.

Import all from module

To import all functions, we can use `*`:

E.g. `from math import *`

Again, we can use `sqrt(2)` directly.

Note that this is considered bad practice! It makes it hard to understand where functions come from and what if several modules come with functions with same name.

Writing your own modules

It is perfectly fine to write and use your own modules. Simply import the name of the file you want to use as module.

E.g.

```
def helloworld():
    print 'hello, world!'

print 'this is my first module'
```

```
import firstmodule

firstmodule.helloworld()
```

What do you notice?

Only running code when main file

By default, Python executes all code in a module when we import it.

However, we can make code run only when the file is the main file:

```
def helloworld():
    print 'hello, world!'

if __name__ == "__main__":
    print 'this only prints when run directly'
```

Try it!

Contents

Lists

- Group variables together
- Specific order
- Access items using square brackets: [ ]

However, do not confuse a list with the mathematical notion of a vector.

Accessing elements

- First item: [0]
- Last item: [-1]

```
myList = [5, 2.3, 'hello']

myList[0]     # 5
myList[2]     # 'hello'
myList[3]     # ! IndexError
myList[-1]    # 'hello'
myList[-3]    # ?
```

Note: can mix element types!

Slicing and adding

- Lists can be sliced: [2:5]
- Lists can be multiplied
- Lists can be added

Look into this: confusing!

```
myList = [5, 2.3, 'hello']

myList[0:2]     # [5, 2.3]

mySecondList = ['a', '3']

concatList = myList + mySecondList
# [5, 2.3, 'hello', 'a', '3']
```

Multiplication

We can even multiply a list by an integer

```
myList = ['hello', 'world']

myList * 2
# ['hello', 'world', 'hello', 'world']

2 * myList
# ['hello', 'world', 'hello', 'world']
```

Lists are mutable

Lists are mutable, this means that individual elements can be changed.

```
myList = ['a', 43, 1.234]

myList[0] = -3
# [-3, 43, 1.234]

x = 2
myList[1:3] = [x, 2.3]  # or: myList[1:] = [x, 2.3]
# [-3, 2, 2.3]

x = 4
# What is myList now?
```

list is unchanged after changing x

Copying a list

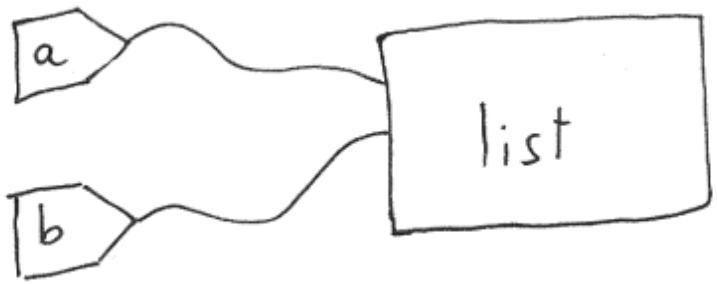
How to copy a list?

```
a = [1, 2, 3]
b = a  # let's copy a
print b

b[1] = 5  # now we want to change an element

print b
# [1, 5, 2]

print a
# [1, 5, 2]
```

.

What just happened?

Variables in python really are tags or references:



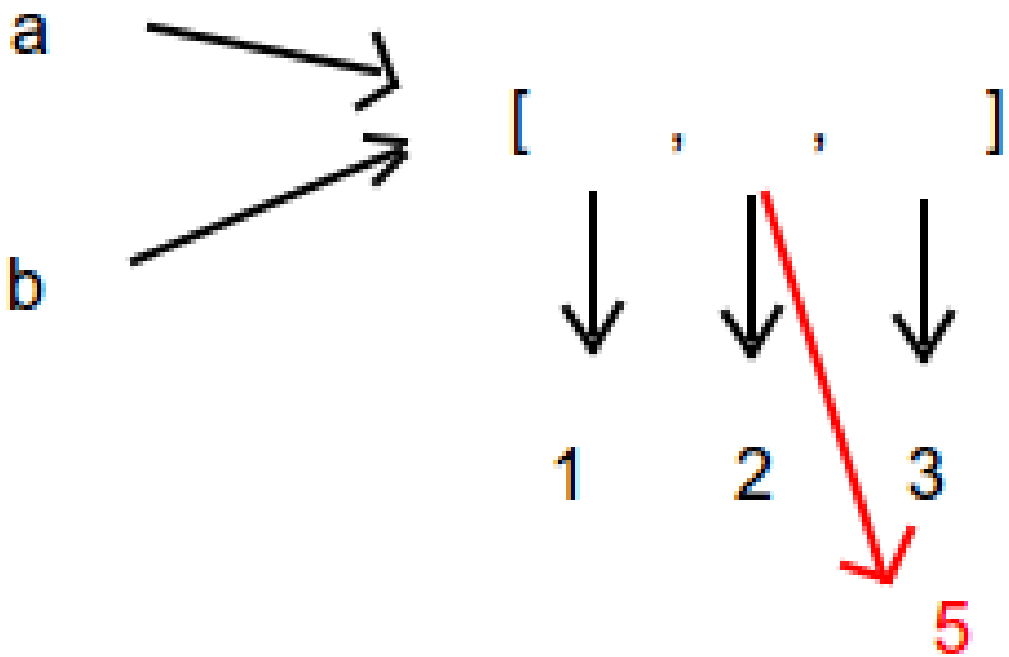
Everything in python is an object, and variables are just references to those objects.

So `b = a` means: `b` is same tag as `a`.

Image from `http://henry.precheur.org/python/copy_list.html`

What just happened?

Here is a diagram of of what happened in the last example.



Assignment operators

Python uses an assignment operator to set up a reference to an object.

The command `x = 'Hello, world!'` does two things: (1) creates a string object for `'Hello, world!'` and points variable `x` to this object.

We can check if two variables point to the same object with the `is` command: `a is b` returns a boolean.

Copying a list

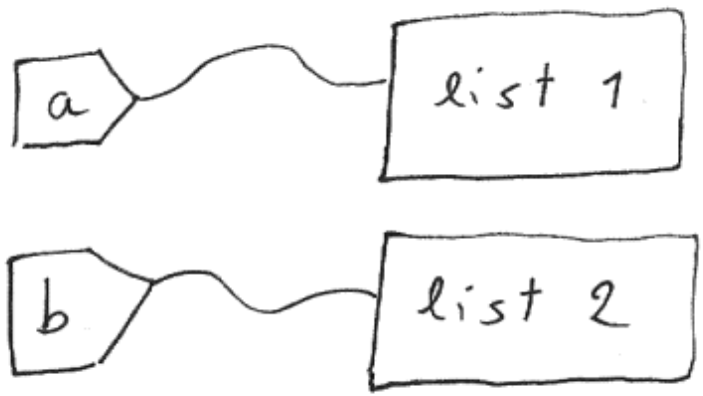


Image from `http://henry.precheur.org/python/copy_list.html`

Copying a list

```
a = [1, 2, 3]
b = a
c = list(a)
print id(a), id(b), id(c)
a[0] = 4
print a, b, c
```

Functions modify lists

Consider the following function:

```
def set_first_to_zero(xs):
    xs[0] = 0

l = [1, 2, 3]
print l
set_first_to_zero(l)
print l
```

What is printed?

`[1, 2, 3], [0, 2, 3]`

Why does the list change, but variables do not?

Functions modify lists

Consider the following function:

```
def set_first_to_zero(xs):
    xs[0] = 0

l = [1, 2, 3]
print l
set_first_to_zero(l)
print l
```

What is printed?

`[1, 2, 3]`, `[0, 2, 3]`

Why does the list change, but variables do not?

Why does the list change, but ints do not?

When passing in an argument that is mutable, we are just passing in the reference to that object. The reference becomes local but the object itself is not.

When passing in an argument that is immutable, we are passing a copy of the object, so everything is local.

More control over lists

- `len(xs)`
- `xs.append(x)`
- `xs.count(x)`
- `xs.insert(i, x)`
- `xs.sort()` and `sorted(xs)`: what's the difference?
- `xs.remove(x)`
- `xs.pop()` or `xs.pop(i)`
- `x in xs`

All these can be found in the Python documentation, google: 'python list'

Or using `dir(xs)` or `dir([])`

More control over lists

- `len(xs)`
- `xs.append(x)`
- `xs.count(x)`
- `xs.insert(i, x)`
- `xs.sort()` and `sorted(xs)`: what's the difference?
- `xs.remove(x)`
- `xs.pop()` or `xs.pop(i)`
- `x in xs`

All these can be found in the Python documentation, google: 'python list'

Or using `dir(xs)` or `dir([])`

Looping over elements

It is very easy to loop over elements of a list using `for`, we have seen this before using `range`.

```
someIntegers = [1, 3, 10]

for integer in someIntegers:
    print integer,
# 1 3 10

# What happens here?
for integer in someIntegers:
    integer = integer*2
```

Looping over elements

Using `enumerate`, we can loop over both element and index at the same time.

```
myList = [1, 2, 4]

for index, elem in enumerate(myList):
    print '{0}) {1}'.format(index, elem)

# 0) 1
# 1) 2
# 2) 4
```

Map

We can apply a function to all elements of a list using `map`

```
l = range(4)
print map(lambda x: x*x*x, l)
# [0, 1, 8, 27]
```

Filter

We can also filter elements of a list using `filter`

```
l = range(8)

print filter(lambda x: x % 2 == 0, l)
# [0, 2, 4, 6]
```

List comprehensions

A very powerful and concise way to create lists is using *list comprehensions*

```
print [i**2 for i in range(5)]
# [0, 1, 4, 9, 16]
```

This is often more readable than using `map` or `filter`