Introduction to Scientific Python

Lecture 2: Functions and Lists

Luke de Oliveira (lukedeo@stanford.edu)

September 30, 2015

Contents

Google Surveys!

- Received input from most of you – please help me help you!
- It seems like a lot of you are interested in Machine Learning...
- How to get the most out of this course

Proposal

- 1-2 paragraph pdf outlining your project ideas
- Email me (lukedeo@stanford.edu) with the subject line:

  `[CME193 PROJECT]LastName-FirstName`.

  I have a filter set up for these.
- Due next Thursday before class (don't stress, this is fun!)

Project

- Content: up to you!
- Some ideas on course website
- Work on your own
- Submit source code and brief write-up (basically updated proposal) via Email
- Due 5pm last day of *this class*, NOT the last day of the quarter.
- See course website
- Ask if things are unclear!

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 = math.pi * radius
```

Functions

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.

Rule of thumb: a functions should do one *obvious* thing!

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 = 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 = math.pi * radius
    return circumference
```

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
```

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
```

What does this function do?

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):
    # string formatting: more on this later.
    return 'Hello, {}!'.format(name)
```

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 f2() # output? x?
print f3() # output? 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 name a function

Try to use these for the homework if you can.

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

```
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?
```

Copying a list

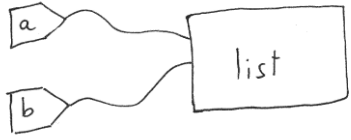
How to copy a list?

```
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:



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

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

Copying a list

Instead: we want:

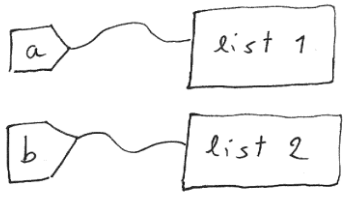


Figure : `b = list(a)` or `b = a[:]`

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)
```

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 variables do not?

We have not changed the tag, only the contents of the list. The variable `l`, that is attached to the list, becomes local. The elements however, do not!

What happens in this case?

```
def list_function(l):
    l = [2, 3, 4]
    return l

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

Why does the list change, but variables do not?

We have not changed the tag, only the contents of the list. The variable `l`, that is attached to the list, becomes local. The elements however, do not!

What happens in this case?

```
def list_function(l):
    l = [2, 3, 4]
    return l

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

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)` / `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)` / `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`

List comprehensions

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

[i * 2 for i in ints]
# [2, 6, 20]

[[i, j] for i in ints for j in ints if i != j]
# [[1, 3], [1, 10], [3, 1], [3, 10], [10, 1], [10, 3]]

[(x, y) for x in xrange(3) for y in xrange(x+1)]
# ...
```

Note how we can have a lists as elements of a list!

List comprehensions

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

[i * 2 for i in ints]
# [2, 6, 20]

[[i, j] for i in ints for j in ints if i != j]
# [[1, 3], [1, 10], [3, 1], [3, 10], [10, 1], [10, 3]]

[(x, y) for x in xrange(3) for y in xrange(x+1)]
# ...
```

Note how we can have a lists as elements of a list!

Implementing map using list comprehensions

Let's implement `map` using list comprehensions

```
def my_map(f, xs):
    return [f(x) for x in xs]
```

Implement filter by yourself in one of the exercises.

Implementing map using list comprehensions

Let's implement `map` using list comprehensions

```
def my_map(f, xs):
    return [f(x) for x in xs]
```

Implement filter by yourself in one of the exercises.