CME 193: Introduction to Scientific Python

Lecture 2: Functions and lists

Sven Schmit

`stanford.edu/~schmit/cme193`

Contents

- Functions
- Lists
- Exercises

Functions

Functions are used to abstract components of a program.

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

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

But 2.5 is rather arbitrary, we really want to be able to calculate the circumference of a circle with arbitrary radius r

```
def calc_circumference(radius):
    # Calculates the circumference of
    # a circle with radius r
    circumference = math.pi * radius
    return circumference
```

Functions

Functions are used to abstract components of a program.

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

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

But 2.5 is rather arbitrary, we really want to be able to calculate the circumference of a circle with arbitrary radius r

```
def calc_circumference(radius):
    # Calculates the circumference of
    # a circle with radius r
    circumference = math.pi * radius
    return circumference
```

Functions

Start a function definition with the keyword `def`

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

```
def functionName(var1, var2):
```

Then comes, indented, the body of the function

Optionally, one or more variables can be returned using the `return` keyword (more on this later)

```
return result1, result2
```

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

Functions

Start a function definition with the keyword def

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

```
def functionName(var1, var2):
```

Then comes, indented, the body of the function

Optionally, one or more variables can be returned using the return keyword (more on this later)

```
return result1, result2
```

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

Functions

Start a function definition with the keyword `def`

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

```
def functionName(var1, var2):
```

Then comes, indented, the body of the function

Optionally, one or more variables can be returned using the `return` keyword (more on this later)

```
return result1, result2
```

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

Functions

Start a function definition with the keyword `def`

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

```
def functionName(var1, var2):
```

Then comes, indented, the body of the function

Optionally, one or more variables can be returned using the `return` keyword (more on this later)

```
return result1, result2
```

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

How to call a function

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

```
»> function1(2.3)
```

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)
```

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(2.3)  # 4.3
print a  # 12.9
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
```

This is confusing, play with all of the above concepts, until you understand them!

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.

Functions as arguments

Note that we can also pass functions as arguments.

We will see how `map` and `filter` take in both a function and a list and applied the function to every element in the list!

Lambda functions

An alternative way to define short functions:

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

print cube(3)
```

Contents

- Functions

- Lists

- Exercises

Lists

- Group variables together
- Some sort of ordering: left to right
- Access items using square brackets: [ ]

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]     # ?
```

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]
# [3, 2, 2.3]

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

Copying a list

How to copy a list?

```
list1 = ['a', 'b', 'c']
list2 = list1  # let's copy
print list2

list2[1] = 42  # now we want to change a value

print list2
# ['a', 42, 'c']

print list1
# ['a', 42, 'c']
```

.

What just happened?

Assigning a variable to a list, such as `x = [1, 2, 3]` does not mean `x` equals `[1, 2, 3]`.

Instead, `x` points to the location in memory where `[1, 2, 3]` is stored.

Thus, when we set `y = x`, `y` points to the same location.

By reassigning `y[i] = c`, we thus edit the list both `x` and `y` are pointing to!

If we want to copy a list, use `y = list(x)`

What just happened?

Assigning a variable to a list, such as `x = [1, 2, 3]` does not mean `x` equals `[1, 2, 3]`.

Instead, `x` points to the location in memory where `[1, 2, 3]` is stored.

Thus, when we set `y = x`, `y` points to the same location.

By reassigning `y[i] = c`, we thus edit the list both `x` and `y` are pointing to!

If we want to copy a list, use `y = list(x)`

What just happened?

Assigning a variable to a list, such as `x = [1, 2, 3]` does not mean `x` equals `[1, 2, 3]`.

Instead, `x` points to the location in memory where `[1, 2, 3]` is stored.

Thus, when we set `y = x`, `y` points to the same location.

By reassigning `y[i] = c`, we thus edit the list both `x` and `y` are pointing to!

If we want to copy a list, use `y = list(x)`

Functions modify lists

Consider the following function:

```
def set_first_to_zero(l):
    l[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(l):
    l[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 list itself, only the contents of the list. The variable l, that points to the elements, becomes local. The elements however, do not!

What happens in this case?

```
def list_function(l):
    l = [1.1, 1.2, 1.3]
    return l

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

More control over lists

- Find length of list: `len(myList)`
- Add element at the end: `myList.append(x)`
- Insert at index i: `myList.insert(i, x)`
- Sort list: `myList.sort()`, or `sorted(myList)`. What is the difference?
- Remove first element with value x: `myList.remove(x)`
- Remove item at index i and return it: `myList.pop([i])` If no index given, ‘pops’ last element of list

All these can be found in the Python documentation! Simply Google: ‘python list’

More control over lists

- Find length of list: `len(myList)`
- Add element at the end: `myList.append(x)`
- Insert at index i: `myList.insert(i, x)`
- Sort list: `myList.sort()`, or `sorted(myList)`. What is the difference?
- Remove first element with value x: `myList.remove(x)`
- Remove item at index i and return it: `myList.pop([i])` If no index given, ‘pops’ last element of list

All these can be found in the Python documentation! Simply Google: ‘python list’

More control over lists

- Find length of list: `len(myList)`
- Add element at the end: `myList.append(x)`
- Insert at index i: `myList.insert(i, x)`
- Sort list: `myList.sort()`, or `sorted(myList)`. What is the difference?
- Remove first element with value x: `myList.remove(x)`
- Remove item at index i and return it: `myList.pop([i])` If no index given, ‘pops’ last element of list

All these can be found in the Python documentation! Simply Google: ‘python list’

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]

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

pairs = [[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]]
```

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

List comprehensions

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

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

pairs = [[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]]
```

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

Contents

- Functions
- Lists
- Exercises

Exercises