Introduction to Scientific Python

Lecture 2: Control, Functions, and Lists

Luke de Oliveira (lukedeo@stanford.edu)

January 14th, 2016

Last time...

- Intro to the course – any questions?
- Notion of a variable

Bookkeeping

- We now have a course website! https://icme.github.io/cme193
- Sign up for Piazza please!
- Do take the 2 min survey about your background!
- Today will be the last slide-based lecture. I will need to cover some basics that are easier in a stand-and-deliver format. Next Tuesday, when we dive into data structures, we will go fully interactive.

Contents

- Control statements

- Functions

- Lists

Control statements

Control statements allow you to do more complicated tasks.

- If
- For
- While

If statements

Using `if`, we can execute part of a program conditional on some statement being true.

```
if traffic_light == ’green’:
    move()
```

Indentation

In Python, blocks of code are defined using indentation.

This means that everything indented after an `if` statement is only executed if the statement is `True`.

If the statement is `False`, the program skips all indented code and resumes at the first line of unindented code

```
if statement:
    # if statement is True, then all code here
    # gets executed but not if statement is False
    print "The statement is true"
    print "Else, this would not be printed"
# the next lines get executed either way
print "Hello, world,"
print "Bye, world!"
```

If-Else statement

We can add more conditions to the If statement using `else` and `elif` (short for else if)

```
if traffic_light == ’green’:
    drive()
elif traffic_light == ’orange’:
    accelerate()
else:
    stop()
```

For loops

Very often, one wants to repeat some action. This can be achieved by a for loop

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

Here, `range(n)` gives us a *list* with integers $0, \ldots, n-1$. More on this later!

While loops

When we not know how many iterations are needed, we can use `while`.

```
i = 1
while i < 100:
    print i**2,
    i += i**2  # a += b is short for a = a + b
# 1 4 36 1764
```

Continue

`continue` continues with the next iteration of the smallest enclosing loop.

```
for num in range(2, 10):
    if num % 2 == 0:
        print "Found an even number", num
        continue
    print "Found an odd number", num
```

from: Python documentation

Continue

continue continues with the next iteration of the smallest enclosing loop.

```
for num in range(2, 10):
    if num % 2 == 0:
        print "Found an even number", num
        continue
    print "Found an odd number", num
```

from: Python documentation

Break

The `break` statement allows us to jump out of the smallest enclosing `for` or `while` loop.

Finding prime numbers:

```
max_n = 10
for n in range(2, max_n):
    for x in range(2, n):
        if n % x == 0:  # n divisible by x
            print n, ’equals’, x, ’*’, n/x
            break
    else:  # executed if no break in for loop
        # loop fell through without finding a factor
        print n, ’is a prime number’
```

from: Python documentation

Break

The `break` statement allows us to jump out of the smallest enclosing `for` or `while` loop.

Finding prime numbers:

```
max_n = 10
for n in range(2, max_n):
    for x in range(2, n):
        if n % x == 0:  # n divisible by x
            print n, ’equals’, x, ’*’, n/x
            break
    else:  # executed if no break in for loop
        # loop fell through without finding a factor
        print n, ’is a prime number’
```

from: Python documentation

Pass

The pass statement does nothing, which can come in handy when you are working on something and want to implement some part of your code later.

```python
if traffic_light == 'green':
    pass  # to implement
else:
    stop()
```

Contents

- Control statements
- Functions
- Lists

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
```

```python
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
```

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:

```python
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

- Control statements
- Functions
- Lists

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’]
```