Introduction to Scientific Python

Lecture 1: Introduction

Luke de Oliveira (lukedeo@stanford.edu)

September 29, 2015

Contents

Course creator: Sven Schmidt

- Borrowed lots of content
- Fourth year PhD student in ICME

Contents

Instructor

Luke de Oliveira

- Second year MS student in ICME, Data Science track, B.S. in Applied Math from Yale
- Background in Machine Learning / Mathematics / Particle Physics
- Use Python for next-gen AI in particle physics at CERN / LHC
- I consult for companies building data science products using lots of stuff you will learn!

Contents

Quick poll

Who has...

- written one line of code?
- written a `for` loop?
- written a `function`?
- heard of recursion?
- heard of object oriented programming?
- unit testing?

Quick poll

Who has...

- written one line of code?
- written a `for` loop?
- written a `function`?
- heard of recursion?
- heard of object oriented programming?
- unit testing?

Quick poll

Who has...

- written one line of code?
- written a `for` loop?
- written a `function`?
- heard of recursion?
- heard of object oriented programming?
- unit testing?

Quick poll

Who has...

- written one line of code?
- written a `for` loop?
- written a `function`?
- heard of recursion?
- heard of object oriented programming?
- unit testing?

Quick poll

Who has...

- written one line of code?
- written a `for` loop?
- written a `function`?
- heard of recursion?
- heard of object oriented programming?
- unit testing?

Quick poll

Who has...

- written one line of code?
- written a `for` loop?
- written a `function`?
- heard of recursion?
- heard of object oriented programming?
- unit testing?

Feedback

If you have comments, like things to be done differently, please let me know and let me know asap.

Email: lukedeo@stanford.edu

Questionnaires at the end of the quarter are nice, but they won't help you! Please bother me with concerns / questions.

Content of course

- Variables
- Functions
- Data types
  - Strings, Lists, Tuples, Dictionaries
- File input and output (I/O)
- Classes
- Exception handling
- Recursion
- Numpy, Scipy
- Some subset (based on interest) of...
  - Matplotlib
  - IPython

Setup of course

- 8 lectures
- Recommended exercises + time to do them
- Office hours! (TBA) Will be focused on examples / usage

More abstract setup of course

My job is to show you the possibilities and some resources (exercises, limitations, design choices etc.)

Your job is to teach yourself Python (with some help!)

If you think you can learn Python by *just* listening to me, you give me way, way, way too much credit.

Exercises

Exercises in second half of the class. Try to finish them in class, else make sure you understand them all before next class.

Feel free (or: You are strongly encouraged) to work in pairs on the exercises.

References

The internet is an excellent source, and Google is a perfect starting point. Usually, typing your question into Google in plain English will lead to an answer on StackOverflow in the first n results, n small.

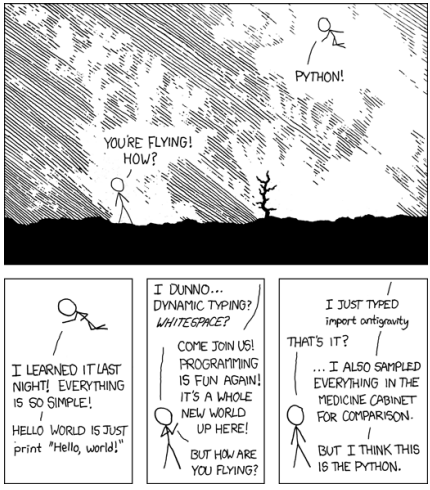
The official documentation is also good, always worth a try (though a bit terse): `https://docs.python.org/2/`.

Other:

- `http://docs.python-guide.org/en/latest/`
- Link: List of free online python books

Contents

Python


YOU'RE FLYING! HOW?
PYTHON!
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

Python

- Relatively easy to learn
- Fast to write code
- Intuitive
- Very versatile (vs Matlab/R)
- Less control, worse performance (vs. C, naive case)
  - Naive Python (read: Python written like C) is *painfully* slow
  - If you're careful, you can do just as well, though!
- Less safety handles, responsibility for user
- Rivals perl for text manipulation!
- Active library variety and development (scikit-learn, Django, Flask, bs4...endless!)

Python

- Duck-typed! If it looks like a ndarray, swims like a ndarray, and quacks like a ndarray, then it probably is an ndarray.
- Dynamic, interpreted! Line-by-line execution when run.

Contents

How to install Python

Many alternatives, but I suggest installing using a prepackaged distribution, such as Anaconda

`http://continuum.io/downloads`

This is very easy to install and also comes with a lot of packages.

See the getting started instructions on the course website for more information.

Packages

Packages enhance the capabilities of Python, so you don't have to program everything by yourself (it's faster too!).

For example: Numpy is a package that adds many linear algebra capabilities, more on that later

How to install packages

To install a package that you do not have, use `pip`, which is the Python package manager.

such as

```
$ pip install seaborn
```

Python 3

Python 3 has been around for a while and is slowly gaining traction.

However, many people still use Python 2, so we will stick with that.

Differences are not big, so you can easily switch.

How to use Python

There are two ways to use Python:

command-line mode: talk directly to the interpreter

scripting-mode: write code in a file (called script) and run code by typing

```
$ python scriptname.py
```

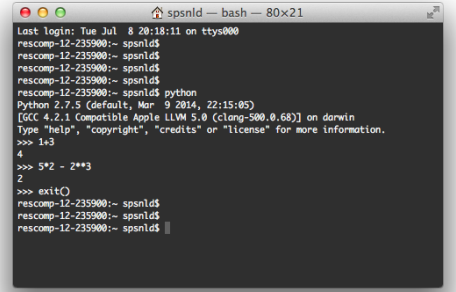
in the terminal.

The latter is what we will focus on in this course, though using the command-line can be useful to quickly check functionality.

The interpreter

We can start the interpreter by typing ‘python’ in the terminal.

Now we can interactively give instructions to the computer, using the Python language.



Scripting mode

A more convenient way to interact with Python is to write a script.

A script contains all code you want to execute. Then you call Python on the script to run the script.

First browse, using the terminal, to where the script is saved

Then call `python scriptname.py`

Scripting mode

Suppose the Python script is saved in a folder `/Documents/Python` called `firstscript.py`.

Then browse to the folder by entering the following command into the terminal

```
$ cd ~/Documents/Python
```

And then run the script by entering

```
$ python firstscript.py
```

Print statement

We can print output to screen using the `print` command

```
print("Hello, world!")
```

Contents

Values

A value is the fundamental thing that a program manipulates.

Values can be “Hello, world!”, `42`, `12.34`, `True`

Values have types. . .

Types

Boolean True/False

String “Hello, world!”

Integer 92

Float 3.1415

Use `type` to find out the type of a variable, as in

```
>> type("Hello, world!")
<type 'str'>
```

Variables

One of the most basic and powerful concepts is that of a *variable*.

A variable *assigns* a name to a value.

```
message = "Hello, world!"
n = 42
e = 2.71

# note we can print variables:
print(n)  # yields 42

# note: everything after pound sign is a comment
```

Try it!

Variables

Almost always preferred to use variables over values:

- Easier to update code
- Easier to understand code (useful naming)

What does the following code do:

```
print 4.2 * 3.5
```

```
length = 4.2
height = 3.5
area = length * height
print(area)
```

Variables

Almost always preferred to use variables over values:

- Easier to update code
- Easier to understand code (useful naming)

What does the following code do:

```
print 4.2 * 3.5
```

```
length = 4.2
height = 3.5
area = length * height
print(area)
```

Keywords

Not allowed to use keywords, they define structure and rules of a language.

Python has 29 keywords, they include:

- and
- def
- for
- return
- is
- in
- class

Integers

Operators for integers

`+ - * / % **`

Note: / uses integer division:

`5 / 2` yields `2`

But, if one of the operands is a float, the return value is a float:

`5 / 2.0` yields `2.5`

Note: Python automatically uses long integers for very large integers.

Floats

A floating point number approximates a real number.

Note: only finite precision, and finite range (overflow)!

Operators for floats

$+$	addition
-	subtraction
*	multiplication
/	division
**	power

Booleans

Boolean expressions:

$==$ equals: `5 == 5` yields `True`

$!=$ does not equal: `5 != 5` yields `False`

$>$ greater than: `5 > 4` yields `True`

$>=$ greater than or equal: `5 >= 5` yields `True`

Similarly, we have $<$ and $<=$.

Logical operators:

`True and False` yields `False`

`True or False` yields `True`

`not True` yields `False`

Booleans

Boolean expressions:

$==$ equals: `5 == 5` yields `True`

$!=$ does not equal: `5 != 5` yields `False`

$>$ greater than: `5 > 4` yields `True`

$>=$ greater than or equal: `5 >= 5` yields `True`

Similarly, we have $<$ and $<=$.

Logical operators:

`True and False` yields `False`

`True or False` yields `True`

`not True` yields `False`

Modules

Not all functionality available comes automatically when starting python, and with good reasons.

We can add extra functionality by importing modules:

```
>> import math
>> math.pi
3.141592653589793
```

Useful modules: `math`, `string`, `random`, and as we will see later `numpy`, `scipy` and `matplotlib`.

More on modules later!

Contents

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.

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