CME 193: Introduction to Scientific Python

Spring 2017

Lecture 1

Lecture 1 Contents

- Course Outline
- Introduction
- Python Basics
- Installing Python

Quick Poll

Who has written one line of code?

...a for loop?

…a function?

Who has heard of recursion?

...object oriented programming?

...unit testing?

Who has programmed in Python before?

Course Outline

Instructors

Blake Jennings

bmj@stanford.edu

- 2nd year MS student in ICME
- From Dallas, Texas
- BS in Software Engineering from UT Austin
- Python as primary language for past 3 years, including:
    - 9M as software engineer at rewardStyle
    - Last summer as Apple data analyst intern in Austin
    - Current work in Stanford Kudla Fund

Jacob Perricone

jacobp2@stanford.edu

- 1st year MS Student in ICME
- Raised in New York City
- BSE in Operations Research and Financial Engineering with Certificates in Computer Science, Statistics and Machine Learning from Princeton University
- Python or Matlab is always my language of choice - nearly all work conducted at Princeton and Stanford has been in Python or Matlab
- Expert in numpy, pandas, scikit-learn

Course Content

- Variables Functions Data types
- Strings, Lists, Tuples, Dictionaries
- File input and output (I/O)
- Classes, object oriented programming
- Exception handling
- Recursion
- Numpy, Scipy, Pandas and Scikit-learn
- Jupyter, Matplotlib and Seaborn
- Unit tests
- List subject to change depending on time, interests

Course setup

- 8 total sessions
- 45 min lecture
- 5 min break
- 30 min interactive - demos and exercises
- First part of class will be traditional lecture, second part will give you time to work on exercises in class
- **Required deliverables:**
    - Exercises
    - Two homework assignments

More abstract setup of course

- My job is to show you the possibilities and some resources
- Ultimately, your job is to teach yourself Python
- In order for it to stick, you will need to put in considerable effort

Exercises

We will work on exercises second half of the class. **Try to finish as much as possible during class time**. They will help you understand topics we just talked about. If you do not finish in class, please try to look at and understand them prior to the next class meeting.

Feel free (or: you are strongly encouraged) to work in pairs on the exercises. It’s acceptable to hand in the same copy of code for your exercise solutions if you work in pairs, but you must mention your partner.

Exercises

(continued)

- At the end of the course, you will be required to hand in your solutions for the exercises you attempted.
- This is to show your active participation in class.
- You are expected to do at least 70% of the assigned exercises. Feel free to skip some problems, for example if you lack some required math background knowledge.
- Don’t worry about this now, just save all the code you write.

Feedback

- If you have comments or would like things to be done differently, please let me know as you think of them. Can tell me in person, via email or Canvas.
- Questionnaires at the end of the quarter are nice, but they won’t help you.

Workload

- The only way to learn Python, is by writing a a lot of Python.
- Good news: Python is fun to write. Put in the effort these 4 weeks and reap the rewards.
- From past experience: If you are new to programming, consider this a hard 3 unit class where you will have to figure out quite a bit on your own. However, if you have a solid background in another language, this class should be pretty easy.

To new programmers

- If you have never programmed before, be warned that this will be difficult.
- The problem: 4 weeks, 8 lectures, 1 unit. We will simply go too fast.
- Alternative: spend some time learning on your own (Codecademy / Udacity etc). There are so many excellent resources online these days. We offer this class every quarter.

Important course information

- **Website**: stanford.edu/~bmj/cme193 (https://stanford.edu/~bmj/cme193)
- **Canvas**: Use Canvas for discussing problems, turning in homework, etc. Also, I will view participation of Canvas discussion as participation on the course.
- **Office hours**: Directly after class, or by appointment in Huang basement

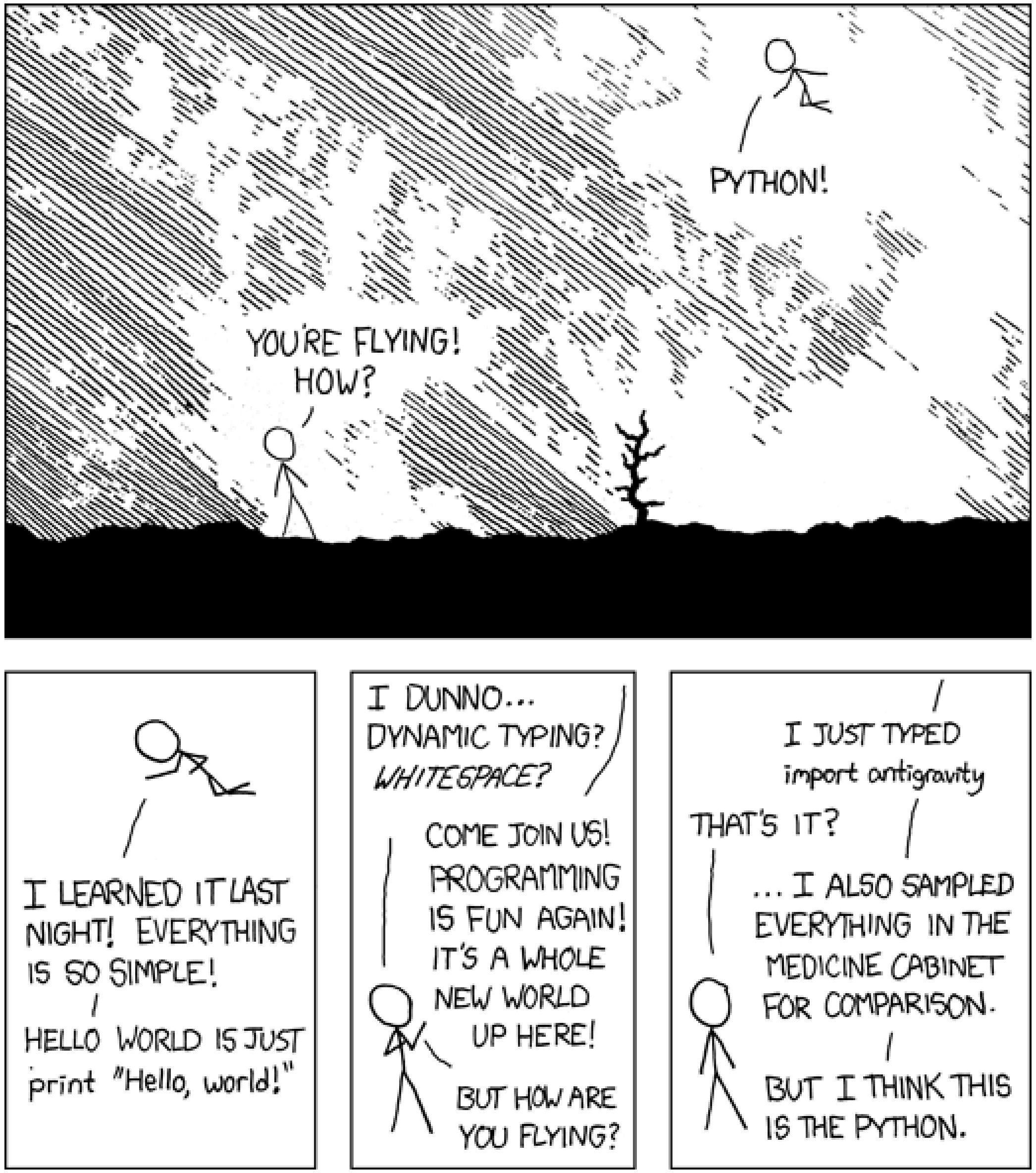
References

- The internet is an excellent source, and Google is a perfect starting point.
- Stackoverflow is the most popular online community of coding QA.
- The official documentation is also good: https://docs.python.org/2/ (https://docs.python.org/2/).
- Course website and Canvas has a list of useful references - I will also try to update them with specific material with each lecture.

Last words before we get to it

- Do the work - utilize the class time
- Make friends
- Fail often
- Fail gently
- Be resourceful

Introduction


PYTHON!
YOU'RE FLYING! HOW?
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.

```
In [61]: print "Hello, world!"
```

```
Hello, world!
```

Python Basics

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!")
  ```

  which returns <**type 'str'>**

- Unlike C/C++ and Java, variables can change types. Python keeps track of the type internally (strongly-typed).

Variables

- One of the most basic and powerful concepts is that of a variable.
- A variable assigns a name to a value.

```
In [62]: 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
```

```
42
```

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
```

```
In [63]: length = 4.2
         height = 3.5
         area = length * height
         print area

         14.7
```

Keywords

- Not allowed to use keywords for naming, they define structure and rules of a language.
- Python has 29 keywords, they include:
    - `def`
    - `for`
    - `and`
    - `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`
- > greater than
  - `5 > 4` yields `True`
- Similarly, we have < and <=.
- `!`= does not equal
  - `5 != 5` yields `False`
- >= greater than or equal
  - `5 >= 5` yields `True`

Logical Operators

- `True and False`
  - evaluates to `False`
- `not True`
  - evaluates to `False`
- `True or False`
  - evaluates to `True`

Statements, expressions and operators

- A statement is an instruction that Python can execute, such as `x=3`
- Operators are special symbols that represent computations, like addition, the values they *operate* on are called operands
- An expression is a combination of values, variable and operators, like `x+3`

Modules

- Not all functionality available comes automatically when starting Python, and with good reasons.
- We can add extra functionality by importing modules:

```
import math
```

- Then we can use things like `math.pi`
- Useful modules: math, string, random, and as we will see later numpy, scipy and matplotlib.
- More on modules later!

```
In [45]: import math
         math.pi

Out[45]: 3.141592653589793
```

Control-flow

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. The indentation within the block needs to be consistent.

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

In [47]:

```python
statement = True # Changing this to False will change the control flow!

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!"
```

```
The statement is true
Else, this would not be printed
Hello, world,
Bye, world!
```

Indentation

- Whitespace is meaningful in Python: especially indentation and placement of newlines.
- Use a newline to end a line of code.
- Use a backslash when must go to next line prematurely.
- No braces to mark blocks of code in Python...
- Use consistent indentation instead.
- The first line with less indentation is outside of the block.
- The first line with more indentation starts a nested block
- Often a colon appears at the start of a new block. (E.g. for function and class definitions.)

`if-else` statement

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

Consider the following example

```
In [49]: if traffic_light == 'green':
             drive()
         elif traffic_light == 'orange':
             accelerate()
         else:
             stop()
```

```
Drive
```

```
In [50]: traffic_light = 'orange'

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

```
Accelerate
```

```
In [51]: traffic_light = 'blue'

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

         Stop!
```

`for` loops

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

```
In [52]: for i in range(5):
             print i**2,
```

```
0 1 4 9 16
```

`for` loops

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

`while` loops

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

In [53]:
```python
i = 1
while i < 100:
    print i**2,
    i += i**2  # a += b is short for a = a + b
```

```
1 4 36 1764
```

`continue`

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

```
In [54]: for num in range(2, 10):
             if num % 2 == 0:
                 print "Found an even number", num
                 continue
             print "Found an odd number", num
```

```
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

`break`

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

```
In [55]: 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'
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

`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.

In [56]:
```python
if traffic_light == 'green':
    pass  # to implement
else:
    stop()
```

```
Stop!
```

Installing Python

Anaconda

- Mac OS and Windows
- Anaconda Python is a Python distribution maintained by Continuum Analytics
- The distribution contains the most widely used Python implementation and up-to-date versions of the most important modules for data science, analytics, and scientific computing.
- Anaconda is easy to download and install and is free to use in personal, academic, and commercial environments.
- By default, Anaconda Python installs into a user local directory and usually does not impact a system provided version of Python. In some cases, having multiple version of Python installed causes problems.

Jupyter Notebook

```
# this is a code cell
print "hello from Jupyter notebook"
# hit ctrl-return to execute code block
# hit shift-return to execute code block and move to next cell
```

This is a Markdown cell

- Markdown is a light weight way to annotate text for nice presentation on the web or in PDF
- For example the `*` will create a bulletted list
- We can easily do *italics*

Resources

- http://jupyter.org/ (http://jupyter.org/)
- https://try.jupyter.org/ (https://try.jupyter.org/)

Using the Python interpreter

- An *interpreter* is a program that reads and executes commands
- It is also sometimes called a REPL or read-evaluate-print-loop
- One way to interact with Python is to use the interpreter
- This is useful for interactive work, learning, and simple testing

Start the Python interpreter on Mac OS

- Open `Terminal.app`. This is located at `/Applications/Utilities/Terminal.app` or may be found using Spotlight Search.
- This is the Bash prompt where commands are entered after `$`
- Type `python` and hit enter to start the interpreter
- This a great way to experiment and learn Python
- To exit the interpreter and return to bash:
    - Enter `>>> exit()`
    - Use the keyboard command `ctrl-d`

Start the Python interpreter from Jupyter

It is possible to access a Python interpreter from inside of the Jupyter notebook. This can be a very quick and handy way to experiment with small bits of Python code.

- From the Jupyter home screen, select the "Terminal" from the "New" dropdown menu.

Scripting model

- A Python script is a text file containing Python code
- Python script file names typically end in `.py`

Let’s create our first script

1. Create a text file named `firstscript.py` with your favorite text editor
2. Insert the following Python code into `firstscript.py`:

```
print("Hello from Python.")
print("I am your first script!")
```

1. Open your favorite terminal emulator (`Terminal.app` on Mac OS)
2. Navigate to the directory containing `firstscript.py` with the `cd` command.
3. Execute the command `$ python firstscript.py`

Why scripts?

Let's write a simple Python script to compute the first `n` numbers in the Fibonacci series. As a reminder, each number in the Fibonacci series is the sum of the two previous numbers. Let `F(i)` be the `i`th number in the series. We define `F(0) = 0` and `F(1) = 1`, then `F(i) = F(i-1) + F(i-2)` for `i >= 2`. Numbers `F(0)` to `F(n)` can be computed with the following Python code:

```
n = 10

if n >= 0:
    fn2 = 0
    print(fn2,end=',')
if n >= 1:
    fn1 = 1
    print(fn1,end=',')
for i in range(2,n+1):
    fn = fn1 + fn2
    print(fn,end=',')
    fn2 = fn1
    fn1 = fn
print()
```

Note, the above code is a preview of Python syntax that we will review in this course.

Fibonacci (continued)

Now, paste this code into a file named `fib.py`. Execute the file with the command `$ python fib.py`. The result should like:

```
$ python fib.py
0,1,1,2,3,5,8,13,21,34,55,
```

To see the utility of scripts, we need to add a bit more code. Change the first line of `fib.py` to be:

```
import sys
n = int(sys.argv[1])
```

This will instruct the script to obtain the value of `n` from the command line:

```
$ python fib.py 0
0,

$ python fib.py 5
0,1,1,2,3,5,

$ python fib.py 21
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,
```

We have increased the utility of our program by making it simple to run from the command line with different input arguments.