

# CME 193: Introduction to Scientific Python

## Winter 2017

### Lecture 1: Course details and Python basics

Blake Jennings

`stanford.edu/~bmj/cme193`

# Contents

- Administration
- Introduction
- Basics
- Variables
- Control statements
- Installation and Exercises

# Instructor

Blake Jennings

- From Dallas, Texas
- Undergraduate degree in Software Engineering from UT Austin
- 2nd year MS student in ICME (Computational Finance track)
- Python as primary language for past 3 years, including:
  - 9M as software engineer at rewardStyle (Dallas software/fashion startup)
  - last summer as Apple data analyst intern in Austin
  - current work in Stanford Kudla Fund and Stanford Open Credit Project

# 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?

# 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
  - 40 min lecture
  - 40 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

- Portfolio
- Final project or homeworks

## 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 portfolio if you work in pairs, but you must mention your partner.

# Portfolio

At the end of the course, you will be required to hand in an aggregate portfolio of your solutions for the exercises you attempted.

This is to show your active participation in class

You are expected to do at least 2/3rd 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.

# Misc

Website `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 (most likely in Huang basement)

# References

The internet is an excellent source, and Google is a perfect starting point.

The official documentation is also good:

<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.

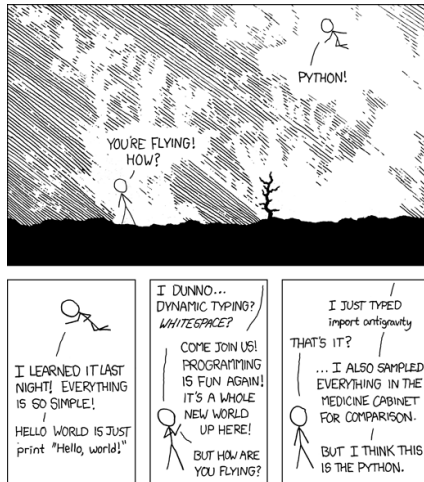
# Last words before we get to it

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

# Contents

- Administration
- **Introduction**
- Basics
- Variables
- Control statements
- Installation and Exercises

# Python



# Python

- High level language
- Relatively easy to learn
- Fast to write code
- Intuitive, English-like readability
- Very versatile (vs Matlab/R)
- Less control, worse performance (vs. C)
- Less safety handles, responsibility for user

# Contents

- Administration
- Introduction
- **Basics**
- Variables
- Control statements
- Installation and Exercises



# How to install Python

Many alternatives, but I suggest installing using a prepackaged distribution, like Anaconda or manage a version using Brew (if using macOS).

Anaconda <https://store.continuum.io/cshop/anaconda/>

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

Homebrew `brew.sh`

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

## Using Python on Stanford machines

If you don't want to or have trouble installing Python locally, you can always use Stanford Farmshare machines or use Cloud9.

More information can be found here: [web.stanford.edu/group/farmshare](http://web.stanford.edu/group/farmshare) and here: <https://c9.io>.

For example, to connect to one of the computers, use command:

```
$ ssh sunetid@corn.stanford.edu
```

X11 forwarding is good for opening up a good editor from the machine locally - I will post a useful link to help with this.

# 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 numpy
```

# 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 - it is also the version installed on the Farmshare computers.

Differences are not too big, so you can easily switch. The differences should not be a problem in the class.

# How to use Python

There are two ways to use Python:

command-line/interactive mode: talk directly to the interpreter

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

```
$ python scriptname.py
```

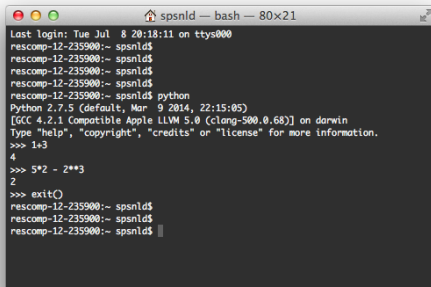
in the terminal.

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

# 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.

A terminal window titled 'spsnld — bash — 80x21' with standard macOS window controls. The terminal shows a user logging in on 'Tue Jul 8 20:18:11 on ttys000'. The user is at a shell prompt 'rescomp-12-235900:~ spsnld\$' and enters several empty commands. Then, they type 'python', which starts 'Python 2.7.5 (default, Mar 9 2014, 22:15:05) [GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin'. The user enters 'Type "help", "copyright", "credits" or "license" for more information.' followed by '>>> 1+3', which outputs '4'. Then '>>> 5\*2 - 2\*\*3' outputs '2'. Finally, '>>> exit()' returns to the shell prompt 'rescomp-12-235900:~ spsnld\$'.

```
spsnld — bash — 80x21
Last login: Tue Jul 8 20:18:11 on ttys000
rescomp-12-235900:~ spsnld$
rescomp-12-235900:~ spsnld$
rescomp-12-235900:~ spsnld$
rescomp-12-235900:~ spsnld$
rescomp-12-235900:~ spsnld$ python
Python 2.7.5 (default, Mar 9 2014, 22:15:05)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 1+3
4
>>> 5*2 - 2**3
2
>>> exit()
rescomp-12-235900:~ spsnld$
rescomp-12-235900:~ spsnld$
rescomp-12-235900:~ spsnld$
```

## 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
```

## Interactive scripting mode

We may also enter the interpreter mode after running the script.

We run `python -i scriptname.py`

We enter command-line mode with all the variables/environment saved from the script, so a good way to examine a script/debug.

Can exit interactive mode by typing `quit()` or hitting `ctr+d`

# Print statement

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

```
print "Hello, world!"
```

Note that one major difference between Python 2 and 3 is the `print` functionality. In Python 3, we would have to write: `print("Hello, world!")`

# Contents

- Administration
- Introduction
- Basics
- **Variables**
- Control statements
- Installation and Exercises

# 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'>
```

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.

```
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 for naming, 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`

# 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

```
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
```

```
»» math.pi
```

```
3.141592653589793
```

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

More on modules later!



# Contents

- Administration
- Introduction
- Basics
- Variables
- **Control statements**
- Installation and Exercises

# 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

```
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!"
```

# 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)

```
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  $0, \dots, n - 1$ . More on this later!

# While loops

When we do 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()
```

## Final words

No need to remember the details: everything is well documented online!

However, knowing about the tools that exist will help you look them up!

Always try the Python documentation first if you forgot something.

# Contents

- Administration
- Introduction
- Basics
- Variables
- Control statements
- **Installation and Exercises**

# Installation and Exercises

You first need to download **Python 2**.

Again, I recommend using conda

(<https://store.continuum.io/cshop/anaconda/>)

Please see the "Getting started" section on the course website  
[stanford.edu/~bmj/cme193](https://stanford.edu/~bmj/cme193).



# Exercises

Located on the course website.