# Getting Started With Python

August 12, 2017

## 1 Setting up Python Environment

**Installation Steps:**

1. Go to the Jupyter Website.
2. Scroll down and and click the Install the Notebook button.
3. The button will take you to the installation documentation.
4. This in turn should take you to the Anaconda Installation page, go to the Anaconda Installation page and click on the Graphical Installer for your system to download the installer.
5. Follow the directions for the graphical installer you chose, it should be straight-forward, just like installing any other software, keep any default options.
6. Once you've successfully downloaded Anaconda you can begin with the installation commands for Jupyter. All installation commands should be run in the Terminal (for Mac and Linux) or the Command Prompt/Powershell (Windows). (Mac users should just search for terminal in Spotlight search, Windows Users just search for either powershell or cmd in your windows search tool to find your appropriate installation tool). You also have the option of using the Anaconda Command Prompt as shown in the videos.
7. Once the installation is done, in your terminal/command prompt type:

   ```
   jupyter notebook
   ```

8. You should eventually see a new tab open up in your browser for you to begin using Jupyter Notebooks. Don't worry if your tab says something like "Conda [Root]" or "Python Default", either of these options will work fine. You can click on these to start a new.
9. For more information on how to use the Jupyter Notebooks, refer to the other lectures in this section.
10. For more information on the Jupyter Notebook system in general, check out the official documentation.

## 2 Jupyter Notebook

```
In [1]: print 'Shift+Enter to run this cell'
```

```
Shift+Enter to run this cell
```

# 3 Numbers in Python

In this section we'll learn about the following topics:

```
1.) Types of Numbers in Python
2.) Basic Arithmetic
3.) Differences between Python 2 vs 3 in division
```

```
In [2]: #Addition
        2+3
```

```
Out[2]: 5
```

```
In [3]: # Subtraction
        2-3
```

```
Out[3]: -1
```

```
In [4]: # Multiplication
        3*5
```

```
Out[4]: 15
```

```
In [5]: # Division
        10/3
```

```
Out[5]: 3
```

```
In [6]: # To use Division Python 3 in Python 2
        # from __future__ import division
        10/3
```

```
Out[6]: 3
```

```
In [7]: # Modulo - Returns remainder after division
        10%3
```

```
Out[7]: 1
```

```
In [8]: # Divison with Floating point
        10.0/3
```

```
Out[8]: 3.3333333333333335
```

```
In [9]: # Divison with Floating point
        10/3.0
```

```
Out[9]: 3.3333333333333335
```

```
In [10]: # Divison with Floating point
         float(10)/3
```

```
Out[10]: 3.3333333333333335

In [11]: # Floating point Issues and Limitations
         0.1+0.2-0.3

Out[11]: 5.551115123125783e-17

In [12]: # Exponentiation
         2**4

Out[12]: 16

In [13]: # Exponentiation Operators to find roots
         16**0.5

Out[13]: 4.0
```

# 4 Strings in Python

In this section we'll learn about the following:

```
1.) Creating Strings
2.) Printing Strings
3.) Differences in Printing in Python 2 vs 3
4.) String Indexing and Slicing
5.) String Properties
6.) String Methods
7.) Print Formatting
```

```
In [14]: 'hello world'
```

```
Out[14]: 'hello world'
```

```
In [15]: "hello world"
```

```
Out[15]: 'hello world'
```

```
In [16]: "I'm ready to use single quotes inside string"
```

```
Out[16]: "I'm ready to use single quotes inside string"
```

```
In [17]: print 'hello world'
         print 'this is yet another string'
```

```
hello world
this is yet another string
```

```
In [18]: # In Python 3, print is a function, not a statement.
         # So you would print statements like this: print('Hello World')
         # Use print function from Python 3 in Python 2
         # from __future__ import print_function
         print('hello world');
```

```
hello world
```

## 4.1 String Basics

```
In [19]: len('Hello World')
```

```
Out[19]: 11
```

## 4.2 String Indexing

```
In [20]: s = "Hello World"

         # Check
         s

Out[20]: 'Hello World'

In [21]: # Show first element
         print s[0]

H


In [22]: # Grab everything past the first term all the way to the length of s which
         s[1:]

Out[22]: 'ello World'

In [23]: # Note that there is no change to the original s
         s

Out[23]: 'Hello World'

In [24]: # Grab everything UP TO the 3rd index
         s[:3]

Out[24]: 'Hel'

In [25]: #Everything
         s[:]

Out[25]: 'Hello World'

In [26]: #Last letter (one index behind 0 so it loops back around)
         s[-1]

Out[26]: 'd'

In [27]: # Grab everything, but go in steps size of 1
         s[::1]

Out[27]: 'Hello World'

In [28]: # Grab everything, but go in step sizes of 2
         s[::2]

Out[28]: 'HloWrd'

In [29]: # We can use this to print a string backwards
         # Reverse a string
         s[::-1]

Out[29]: 'dlroW olleH'
```

### 4.3 String Properties

```
In [30]: s

Out[30]: 'Hello World'

In [81]: # Let's try to change the first letter to 'x'
         # Strings are immutable
         # Below code gives error
         # s[0] = 'x'

In [32]: s

Out[32]: 'Hello World'

In [33]: # Concatenate strings!
         s + ' concatenate me!'

Out[33]: 'Hello World concatenate me!'

In [34]: # We can reassign s completely though!
         s = 'Hello World'
         s = s + ' concatenate me!'
         print s

Hello World concatenate me!


In [35]: letter = 'a'
         # we can use multiplication symbol to create repetition
         letter*5

Out[35]: 'aaaaa'
```

### 4.4 Basic Strings Built in Methods

```
In [36]: s

Out[36]: 'Hello World concatenate me!'

In [37]: # Uppercase
         s.upper()

Out[37]: 'HELLO WORLD CONCATENATE ME!'

In [38]: # Lowercase
         s.lower()

Out[38]: 'hello world concatenate me!'

In [39]: # Split a string by blank space (this is the default)
         s.split()
```

```
Out[39]: ['Hello', 'World', 'concatenate', 'me!']

In [40]: # Split by a specific element (doesn't include the element that was split
         s.split('W')

Out[40]: ['Hello ', 'orld concatenate me!']
```

# 5 Print Formatting

```
In [41]: print 'This is a string'
```

```
This is a string
```

## 5.1 Strings

```
In [42]: s = 'STRING'
         print 'Place another string with a mod and s: %s' %(s)
```

```
Place another string with a mod and s: STRING
```

## 5.2 Floating point numbers

```
In [43]: print 'Floating point numbers: %1.2f' %(13.144)
```

```
Floating point numbers: 13.14
```

```
In [44]: print 'Floating point numbers: %1.0f' %(13.144)
```

```
Floating point numbers: 13
```

```
In [45]: print 'Floating point numbers: %1.5f' %(13.144)
```

```
Floating point numbers: 13.14400
```

```
In [46]: print 'Floating point numbers: %10.2f' %(13.144)
```

```
Floating point numbers:      13.14
```

## 5.3 Conversion Format Methods

It should be noted that two methods %s and %r actually convert any python object to a string using two separate methods: str() and repr()

```
In [47]: print 'Here is a number: %s. Here is a string: %s' %(123.1,'hi')
```

```
Here is a number: 123.1. Here is a string: hi
```

```
In [48]: print 'Here is a number: %r. Here is a string: %r' %(123.1,'hi')
```

```
Here is a number: 123.1. Here is a string: 'hi'
```

### 5.4 Multiple Formatting

```
In [49]: print 'First: %s, Second: %1.2f, Third: %r' %('hi!',3.14,22)
```

```
First: hi!, Second: 3.14, Third: 22
```

### 5.5 Using the string .format() method

```
In [50]: print 'This is a string with an {p}'.format(p='insert')
```

```
This is a string with an insert
```

```
In [51]: # Multiple times:
         print 'One: {p}, Two: {p}, Three: {p}'.format(p='Hi!')
```

```
One: Hi!, Two: Hi!, Three: Hi!
```

```
In [52]: # Several Objects:
         print 'Object 1: {a}, Object 2: {b}, Object 3: {c}'.format(a=1,b='two',c=1
```

```
Object 1: 1, Object 2: two, Object 3: 12.3
```

# 6 Lists

Unlike strings, they are mutable, meaning the elements inside a list can be changed!

In this section we will learn about:

```
1.) Creating lists
2.) Indexing and Slicing Lists
3.) Basic List Methods
4.) Nesting Lists
5.) Introduction to List Comprehensions
```

```
In [54]: # Assign a list to an variable named my_list
         my_list = [1,2,3]

In [56]: # lists can actually hold different object types
         my_list = ['A string',23,100.232,'o']

In [57]: len(my_list)

Out[57]: 4
```

## 6.1 Indexing and Slicing

```
In [58]: my_list = ['one','two','three',4,5]

In [59]: # Grab element at index 0
         my_list[0]

Out[59]: 'one'

In [60]: # Grab index 1 and everything past it
         my_list[1:]

Out[60]: ['two', 'three', 4, 5]

In [61]: # Grab everything UP TO index 3
         my_list[:3]

Out[61]: ['one', 'two', 'three']

In [62]: my_list + ['new item']

Out[62]: ['one', 'two', 'three', 4, 5, 'new item']

In [63]: # Doesn't change actual list
         my_list

Out[63]: ['one', 'two', 'three', 4, 5]

In [64]: # Reassign
         my_list = my_list + ['add new item permanently']
```

```
In [65]: my_list

Out[65]: ['one', 'two', 'three', 4, 5, 'add new item permanently']

In [66]: # We can also use the * for a duplication method similar to strings
         my_list * 2

Out[66]: ['one',
          'two',
          'three',
          4,
          5,
          'add new item permanently',
          'one',
          'two',
          'three',
          4,
          5,
          'add new item permanently']

In [67]: # Again doubling not permanent
         my_list

Out[67]: ['one', 'two', 'three', 4, 5, 'add new item permanently']
```

### 6.2 Basic List Methods

```
In [68]: # Create a new list
         l = [1,2,3]

In [69]: # Append
         l.append('append me!')

In [70]: # Show
         l

Out[70]: [1, 2, 3, 'append me!']

In [71]: # Pop off the 0 indexed item
         l.pop(0)

Out[71]: 1

In [72]: # Show
         l

Out[72]: [2, 3, 'append me!']

In [73]: # Assign the popped element, remember default popped index is -1
         popped_item = l.pop()
```

```
In [74]: popped_item

Out[74]: 'append me!'

In [75]: # Show remaining list
         l

Out[75]: [2, 3]

In [80]: # It should also be noted that lists indexing will return an error...
         # if there is no element at that index
         # l[100]

In [77]: new_list = ['a','e','x','b','c']

In [78]: #Show
         new_list

Out[78]: ['a', 'e', 'x', 'b', 'c']

In [79]: # Use reverse to reverse order (this is permanent!)
         new_list.reverse()

In [80]: new_list

Out[80]: ['c', 'b', 'x', 'e', 'a']

In [81]: # reverse order (this isn't permanent)
         new_list[::-1]

Out[81]: ['a', 'e', 'x', 'b', 'c']

In [82]: # Use sort to sort the list (in this case alphabetical order, but for numb
         new_list.sort()

In [83]: new_list

Out[83]: ['a', 'b', 'c', 'e', 'x']
```

### 6.3 Nesting Lists

```
In [84]: # Let's make three lists
         lst_1=[1,2,3]
         lst_2=[4,5,6]
         lst_3=[7,8,9]

         # Make a list of lists to form a matrix
         matrix = [lst_1,lst_2,lst_3]

In [85]: # Show
         matrix
```

```
Out[85]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

In [86]: # Grab first item in matrix object
         matrix[0]

Out[86]: [1, 2, 3]

In [87]: # Grab first item of the first item in the matrix object
         matrix[0][0]

Out[87]: 1
```

### 6.4 List Comprehensions

```
In [90]: # Build a list comprehension by deconstructing a for loop within a []
         # We used list comprehension here to grab the first element of every row i
         first_col = [row[0] for row in matrix]

In [89]: first_col

Out[89]: [1, 4, 7]
```

# 7 Dictionaries

If you're familiar with other languages you can think of these Dictionaries as hash tables.

This section will serve as a brief introduction to dictionaries and consist of:

```
1.) Constructing a Dictionary
2.) Accessing objects from a dictionary
3.) Nesting Dictionaries
4.) Basic Dictionary Methods
```

Mappings are a collection of objects that are stored by a key, unlike a sequence that stored objects by their relative position. This is an important distinction, since mappings won't retain order since they have objects defined by a key.

## 7.1 Constructing a Dictionary

```
In [91]: # Make a dictionary with {} and : to signify a key and a value
         my_dict = {'key1':'value1','key2':'value2'}

In [92]: # Call values by their key
         my_dict['key2']

Out[92]: 'value2'

In [93]: my_dict

Out[93]: {'key1': 'value1', 'key2': 'value2'}

In [94]: my_dict = {'key1':123,'key2':[12,23,33],'key3':['item0','item1','item2']}

In [95]: #Lets call items from the dictionary
         my_dict['key3']

Out[95]: ['item0', 'item1', 'item2']

In [96]: # Can call an index on that value
         my_dict['key3'][0]

Out[96]: 'item0'

In [97]: #Can then even call methods on that value
         my_dict['key3'][0].upper()

Out[97]: 'ITEM0'

In [98]: my_dict['key1']

Out[98]: 123

In [99]: # Subtract 123 from the value
         my_dict['key1'] = my_dict['key1'] - 123
```

```
In [100]: #Check
          my_dict['key1']

Out[100]: 0

In [101]: # Set the object equal to itself minus 123
          my_dict['key1'] -= 123
          my_dict['key1']

Out[101]: -123

In [102]: # Create a new dictionary
          d = {}

In [103]: # Create a new key through assignment
          d['animal'] = 'Dog'

In [104]: # Can do this with any object
          d['answer'] = 42

In [105]: #Show
          d

Out[105]: {'animal': 'Dog', 'answer': 42}
```

### 7.2 Nesting with Dictionaries

```
In [106]: # Dictionary nested inside a dictionary nested in side a dictionary
          d = {'key1':{'nestkey':{'subnestkey':'value'}}}

In [107]: # Keep calling the keys
          d['key1']['nestkey']['subnestkey']

Out[107]: 'value'
```

### 7.3 A Few Dictionary Methods

```
In [108]: # Create a typical dictionary
          d = {'key1':1,'key2':2,'key3':3}

In [109]: # Method to return a list of all keys
          d.keys()

Out[109]: ['key3', 'key2', 'key1']

In [110]: # Method to grab all values
          d.values()

Out[110]: [3, 2, 1]

In [111]: # Method to return tuples of all items  (we'll learn about tuples soon)
          d.items()

Out[111]: [('key3', 3), ('key2', 2), ('key1', 1)]
```

# 8 Tuples

In Python tuples are very similar to lists, however, unlike lists they are *immutable* meaning they can not be changed. You would use tuples to present things that shouldn't be changed, such as days of the week, or dates on a calendar.

In this section, we will get a brief overview of the following:

```
1.) Constructing Tuples
2.) Basic Tuple Methods
3.) Immutability
4.) When to Use Tuples.
```

## 8.1 Constructing Tuples

The construction of a tuples use () with elements separated by commas. For example:

```
In [1]: # Can create a tuple with mixed types
        t = (1,2,3)

In [2]: # Check len just like a list
        len(t)

Out[2]: 3

In [3]: # Show
        t

Out[3]: (1, 2, 3)

In [6]: # Can also mix object types
        t = ('one',2,'three')

        # Show
        t

Out[6]: ('one', 2, 'three')

In [5]: # Use indexing just like we did in lists
        t[0]

Out[5]: 'one'

In [7]: # Slicing just like a list
        t[:-1]

Out[7]: ('one', 2)
```

## 8.2 Basic Tuple Methods

Tuples have built-in methods, but not as many as lists do. Lets look at two of them

```
In [8]: # Use .index to enter a value and return the index
        t.index('one')
```

```
Out[8]: 0
```

```
In [9]: # Use .count to count the number of times a value appears
        t.count('one')
```

```
Out[9]: 1
```

## 8.3 Immutability

It can't be stressed enough that tuples are immutable. To drive that point home:

```
In [82]: # Below code gives you error
         # t[0]= 'change'
```

```
In [83]: # Below code gives you error
         # t.append('nope')
```

# 9 Files

Python uses file objects to interact with external files on your computer. These file objects can be any sort of file you have on your computer, whether it be an audio file, a text file, emails, Excel documents, etc. Note: You will probably need to install certain libraries or modules to interact with those various file types, but they are easily available. (We will cover downloading modules later on in the course).

Python has a built-in open function that allows us to open and play with basic file types. First we will need a file though. We're going to use some iPython magic to create a text file!

## 9.1 iPython Writing a File

```
In [20]: %%writefile test.txt
         Hello, this is a quick test file
         This is some more text

Overwriting test.txt
```

## 9.2 Python Opening a file

We can open a file with the open() function. The open function also takes in arguments (also called parameters). Lets see how this is used:

```
In [21]: # Open the text.txt we made earlier
         my_file = open('test.txt')

In [22]: # We can now read the file
         my_file.read()

Out[22]: 'Hello, this is a quick test file\nThis is some more text'

In [23]: # But what happens if we try to read it again?
         my_file.read()

Out[23]: ''

This happens because you can imagine the reading "cursor" is at the end of the file

In [28]: # Seek to the start of file (index 0)
         my_file.seek(0)

In [26]: # Now read again
         my_file.read()

Out[26]: 'Hello, this is a quick test file\nThis is some more text'

In order to not have to reset every time, we can also use the readlines method. Use
everything will be held in memory. We will learn how to iterate over large files la

In [30]: # Readlines returns a list of the lines in the file.
         my_file.seek(0)
         my_file.readlines()

Out[30]: ['Hello, this is a quick test file\n', 'This is some more text']
```

### 9.3 Writing to a File

By default, using the open() function will only allow us to read the file, we need to pass the argument 'w' to write over the file. For example:

```
In [71]: # Add a second argument to the function, 'w' which stands for write
         my_file = open('test.txt','w+')

In [72]: # Write to the file
         my_file.write('This is a new line')

In [73]: # Read the file
         my_file.seek(0)
         my_file.read()

Out[73]: 'This is a new line'
```

### 9.4 Iterating through a File

Lets get a quick preview of a for loop by iterating over a text file. First let's make a new text file with some iPython Magic:

```
In [74]: %%writefile test.txt
         First Line
         Second Line

Overwriting test.txt


In [78]: for line in open('test.txt'):
             print line

First Line

Second Line
```

Its important to note a few things here:

```
1.) We could have called the 'line' object anything (see example below).
2.) By not calling .read() on the file, the whole text file was not stored in memor
3.) Notice the indent on the second line for print. This whitespace is required in
```

```
In [79]: # Pertaining to the first point above
         for asdf in open('test.txt'):
             print asdf

First Line

Second Line
```

# 10 Sets and Booleans

## 10.1 Sets

Sets are an unordered collection of *unique* elements. We can construct them by using the set() function. Let's go ahead and make a set to see how it works