Getting Started With Python

September 24, 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.

```
In [84]: # Create a set
         x = set()

In [85]: # We add to sets with the add() method
         x.add(1)

In [86]: #Show
         x

Out[86]: {1}

In [87]: # Add a different element
         x.add(2)

In [88]: #Show
         x

Out[88]: {1, 2}

In [89]: # Try to add the same element
         x.add(1)

In [90]: #Show
         x

Out[90]: {1, 2}

In [91]: # Create a list with repeats
         l = [1,1,2,2,3,4,5,6,1,1]

In [92]: # Cast as set to get unique values
         set(l)

Out[92]: {1, 2, 3, 4, 5, 6}
```

10.2 Booleans

Python comes with Booleans (with predefined True and False displays that are basically just the integers 1 and 0). It also has a placeholder object called None.

```
In [93]: # Set object to be a boolean
         a = True

In [94]: #Show
         a
```

```
Out[94]: True

In [95]: # Output is boolean
         1 > 2

Out[95]: False

In [96]: # None placeholder
         b = None

In [102]: print 0 == False
          print 1 == True

True
True
```

11 Comparison Operators

These operators allow us to compare variables and output a Boolean value (True or False).

11.1 Table of Comparison Operators

Operator
Description
Example
==
If the values of two operands are equal, then the condition becomes true.
(a == b) is not true.
!=
If values of two operands are not equal, then condition becomes true.
(a != b) is true
<>
If values of two operands are not equal, then condition becomes true.
(a <> b) is true. This is similar to != operator.
>
If the value of left operand is greater than the value of right operand, then condition becomes true.
(a > b) is not true.
<
If the value of left operand is less than the value of right operand, then condition becomes true.
(a < b) is true.
>=
If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.
(a >= b) is not true.
<=
If the value of left operand is less than or equal to the value of right operand, then condition becomes true.
(a <= b) is true.

Examples

```
In [28]: # Equal ==
         print "Is 2 == 2: ", 2==2
         print "Is 2 == 3: ", 2==3
```

```
Is 2 == 2:  True
Is 2 == 3:  False
```

```
In [29]: # Not Equal !=
         print "Is 2 != 2: ", 2!=2
         print "Is 2 != 3: ", 2!=3
```

```
Is 2 != 2:  False
Is 2 != 3:  True
```

```
In [30]: # Not Equal <>
         print "Is 2 <> 2: ", 2<>2
         print "Is 2 <> 3: ", 2<>3
```

```
Is 2 <> 2:  False
Is 2 <> 3:  True
```

```
In [32]: # Less than <
         print "Is 2 < 2: ", 2<2
         print "Is 2 < 3: ", 2<3
```

```
Is 2 < 2:  False
Is 2 < 3:  True
```

```
In [31]: # Less than or equal to <=
         print "Is 2 <= 2: ", 2<=2
         print "Is 2 <= 3: ", 2<=3
```

```
Is 2 <= 2:  True
Is 2 <= 3:  True
```

```
In [33]: # Greater than >
         print "Is 2 > 2: ", 2>2
         print "Is 3 > 2: ", 3>2
```

```
Is 2 > 2:  False
Is 3 > 2:  True
```

```
In [34]: # Greater than or equal to >=
         print "Is 2 >= 2: ", 2>=2
         print "Is 3 >= 2: ", 3>=2
```

```
Is 2 >= 2:  True
Is 3 >= 2:  True
```

12 Chained Comparison Operators

An interesting feature of Python is the ability to *chain* multiple comparisons to perform a more complex test. You can use these chained comparisons as a shorthand for larger Boolean Expressions.

We will also introduce two other important statements in python: **and** and **or**.

```
In [37]: 1 < 2 < 3
```

```
Out[37]: True
```

The above statement checks if 1 was less than 2 **and** if 2 was less than 3. We could have written this using an **and** statement in Python:

```
In [38]: 1<2 and 2<3
```

```
Out[38]: True
```

The **and** is used to make sure two checks have to be true in order for the total check to be true. Let's see another example:

```
In [39]: 1 < 3 > 2
```

```
Out[39]: True
```

The above checks if 3 is larger than both the other numbers, so you could use **and** to rewrite it as:

```
In [40]: 1<3 and 3>2
```

```
Out[40]: True
```

Its important to note that Python is checking both instances of the comparisons. We can also use **or** to write comparisons in Python. For example:

```
In [42]: 1==2 or 2<3
```

```
Out[42]: True
```

```
In [43]: 1==1 or 100==1
```

```
Out[43]: True
```

13 Introduction to Python Statements

13.1 Python vs Other Languages

Let's create a simple statement that says: "If a is greater than b, assign 2 to a and 4 to b"

Take a look at these two if statements (we will learn about building out if statements soon).

Version 1 (Other Languages)

```
if (a>b){
    a = 2;
    b = 4;
}
```

Version 2 (Python)

```
if a>b:
    a = 2
    b = 4
```

Python gets rid of () and {} by incorporating two main factors: a *colon* and *whitespace*. The statement is ended with a colon, and whitespace is used (indentation) to describe what takes place in case of the statement.

Another major difference is the lack of semicolons in Python. Semicolons are used to denote statement endings in many other languages, but in Python, the end of a line is the same as the end of a statement.

Note: Python is heavily driven by code indentation and whitespace. This means that code readability is a core part of the design of the Python language.

14 if,elif,else Statements

Syntax format for if statements, elif, else statements:

```
if case1:
    perform action1
elif case2:
    perform action2
else:
    perform action 3
```

```
In [46]: if True:
             print 'It was true!'
```

```
It was true!
```

```
In [47]: x = False

         if x:
             print 'x was True!'
         else:
             print 'I will be printed in any case where x is not true'
```

```
I will be printed in any case where x is not true
```

```
In [45]: person = 'Niket'

         if person == 'Niket':
             print 'Welcome Niket!'
         elif person =='Jose':
             print "Welcome Jose!"
         else:
             print "Welcome Programmer"
```

```
Welcome Niket!
```

Note: Nested if statements are each checked until a True boolean causes the nested code below it to run. You should also note that you can put in as many elif statements as you want before you close off with an else.

15 for Loops

A **for** loop acts as an iterator in Python, it goes through items that are in a *sequence* or any other iterable item.

Here's the syntax for a **for** loop in Python:

```
for item in object:
    statements to do stuff
```

```
In [49]: words = ['cat', 'window', 'defenestrate']
         # Iterate over words
         for word in words:
             print word, len(word)
```

```
cat 3
window 6
defenestrate 12
```

```
In [50]: # Iterate over string
         for letter in 'This is a string.':
             print letter
```

```
T
h
i
s

i
s

a

s
t
r
i
n
g
.
```

```
In [57]: tup = (1,2,3,4,5)
         # Iterate over tuple
         for t in tuple:
             print t
```

```
10
12
```

Tuples have a special quality when it comes to **for** loops. If you are iterating through a sequence that contains tuples, the item can actually be the tuple itself, this is an example of *tuple unpacking*. During the **for** loop we will be unpacking the tuple inside of a sequence and we can access the individual items inside that tuple!

```
In [52]: list = [(2,4),(6,8),(10,12)]

In [56]: for tuple in list:
             print tuple

(2, 4)
(6, 8)
(10, 12)


In [60]: # Now with unpacking!
         for (t1,t2) in list:
             print t1

2
6
10


In [64]: dict = {'k1':1,'k2':2,'k3':3}
         # Iterate over dictionary
         # Below Code iterates over only keys
         for item in dict:
             print item

k3
k2
k1


In [66]: # Iterate over dictionary using generator
         # Create a generator
         for k,v in dict.iteritems():
             print k,v

k3 3
k2 2
k1 1
```

15.0.1 Python 3: items()

In Python 3 you should use .items() to iterate through the keys and values of a dictionary.

16 while loops

The **while** statement in Python is one of most general ways to perform iteration. A **while** statement will repeatedly execute a single statement or group of statements as long as the condition is true.

Syntax of a while loop is:

```
while test:
    code statement
else:
    final code statements
```

```
In [69]: i = 1
         while i <= 5:
             print i
             i+=1

1
2
3
4
5
```

```
In [74]: # Use of else clause in while loop
         i = 1
         while i <= 5:
             print i
             i+=1
         else:
             print 'All Done!'

1
2
3
4
5
All Done!
```

```
In [77]: # Else clause isnt executed if loop is broken in between using break
         i = 1
         while i <= 5:
             print i
```

```
        i+=1
        if i==3:
            break
    else:
        print 'All Done!'
```

```
1
2
```

17 break, continue, pass statements

```
break: Breaks out of the current closest enclosing loop.
continue: Continues to the next iteration of the closest enclosing loop.
pass: Does nothing at all.
```

```
In [1]: # Break statement
        i = 1
        while i <= 5:
            print i
            i+=1
            if i==3:
                break
        else:
            print 'All Done!'
```

```
1
2
```

```
In [8]: # Continue statement
        i = 1
        while i <= 5:
            if i==3:
                i+=1
                continue
            print i
            i+=1
        else:
            print 'All Done!'
```

```
1
2
4
5
All Done!
```

A word of caution!! It is possible to create an infinitely running loop with while statements.

18 range()

range() allows us to create a list of numbers ranging from a starting point *up to* an ending point. We can also specify step size.

```
Docstring:
range(stop) -> list of integers
range(start, stop[, step]) -> list of integers
```

```
In [9]: # if only one parameter is specified it serves as end point
        # Below code generates a list of numbers starting from 0 upto 10
        range(10)
```

```
Out[9]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [11]: # Below code generates a list of numbers starting from 0 upto 5
         range(0,5)
```

```
Out[11]: [0, 1, 2, 3, 4]
```

```
In [12]: # Below code generates a list of numbers starting from 0 upto 10 in steps
         range(0,10,2)
```

```
Out[12]: [0, 2, 4, 6, 8]
```

18.0.2 Python 3 Alert!

You might have been wondering, what happens if I want to use a huge range of numbers? Can my computer store that all in memory?

Great thinking! This is a dilemma that can be solved with the use of a generator. A generator allows the generation of generated objects that are provided at that instance but does not store every instance generated into memory.

This means a generator would not create a list to generate like range() does, but instead provide a one time generation of the numbers in that range. Python 2 has a built-in range generator called xrange(). It is recommended to use xrange() for **for** loops in Python 2.

In Python 3, range() behaves as a generator

```
In [17]: for num in xrange(10):
             print num
```

```
0
1
2
3
4
5
6
7
8
9
```

19 List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

```
In [7]: # List of Squares Without using List Comprehensions
        squares = []
        for x in range(10):
            squares.append(x**2)
        squares

Out[7]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

In [10]: # List of Squares using List Comprehensions
         squares = [x**2 for x in xrange(10)]
         squares

Out[10]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

In [13]: # Grab every letter in string
         lst = [x for x in 'word']
         lst

Out[13]: ['w', 'o', 'r', 'd']

In [11]: # List Comprehensions with if statement
         # List of even numbers in a range
         lst = [x for x in range(11) if x % 2 == 0]
         lst

Out[11]: [0, 2, 4, 6, 8, 10]

In [12]: # Nested List Comprehensions
         lst = [ x**2 for x in [x**2 for x in range(11)]]
         lst

Out[12]: [0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10000]
```

20 Methods

Methods are essentially functions built into objects. To learn about how to create our own objects and methods refer to Object Oriented Programming (OOP) and classes sections.

Methods will perform specific actions on the object and can also take arguments, just like a function. Methods are in the form:

```
object.method(arg1,arg2,etc...)
```

```
In [10]: # Lets take a look at various methods of list
         # Create a simple list
         l = [1,2,3,4,5]
```

Fortunately, with iPython and the Jupyter Notebook we can quickly see all the possible methods using the tab key. The methods for a list are:

- append
- count
- extend
- insert
- pop
- remove
- reverse
- sort

```
In [11]: l.append(6)
         l
```

```
Out[11]: [1, 2, 3, 4, 5, 6]
```

You can always use Shift+Tab in the Jupyter Notebook to get more help about the method. In general Python you can use the help() function:

```
In [12]: help(l.count)
```

```
Help on built-in function count:

count(...)
    L.count(value) -> integer -- return number of occurrences of value
```

21 Functions

A function is a useful construct that groups together a set of statements so they can be run more than once. They can also let us specify parameters that can serve as inputs to the functions.

On a more fundamental level, functions allow us to not have to repeatedly write the same code again and again.

21.1 Defining Functions using def statement

```
In [13]: def name_of_function(arg1,arg2):
             '''
             This is where the function's Document String (doc-string) goes
             '''
             # Do stuff here
             #return desired result
```

21.2 Examples

```
In [14]: # Function without arguments
         def say_hello():
             print 'hello'

         say_hello()
```

```
hello
```

```
In [21]: # Function with arguments
         def greet(name):
             print 'hello %s' %name

         greet('Niket')
```

```
hello Niket
```

```
In [26]: # Function with default arguments
         def greet(name='World'):
             print 'hello %s' %name

         # Call with one arguments
         greet('Niket')
         # Call with no arguments
         greet()
```

```
hello Niket
hello World
```

```
In [29]: # Using Return statement
         def add(a, b):
             return a + b

In [30]: add(5, 6)

Out[30]: 11

In [31]: import math

         def is_prime(num):
             '''
             Methods checks if num is prime
             if num is prime returns True
             else returns False
             '''
             if num % 2 == 0 and num > 2:
                 return False
             for i in range(3, int(math.sqrt(num)) + 1, 2):
                 if num % i == 0:
                     return False
             return True

In [33]: is_prime(23)

Out[33]: True

In [34]: is_prime(27)

Out[34]: False
```

22 Lambda expressions

One of Pythons most useful (and for beginners, confusing) tools is the lambda expression. lambda expressions allow us to create "anonymous" functions. This basically means we can quickly make ad-hoc functions without needing to properly define a function using def.

Function objects returned by running lambda expressions work exactly the same as those created and assigned by defs. There is key difference that makes lambda useful in specialized roles:

lambda's body is a single expression, not a block of statements.

- The lambda's body is similar to what we would put in a def body's return statement. We simply type the result as an expression instead of explicitly returning it. Because it is limited to an expression, a lambda is less general that a def. We can only squeeze design, to limit program nesting. lambda is designed for coding simple functions, and def handles the larger tasks.

Lets slowly break down a lambda expression by deconstructing a function:

```
In [1]: def square(num):
            result = num**2
            return result

In [2]: square(2)

Out[2]: 4
```

Continuing the breakdown:

```
In [4]: def square(num):
            return num**2

In [5]: square(2)

Out[5]: 4
```

We can actually write this in one line (although it would be bad style to do so)

```
In [6]: def square(num): return num**2

In [7]: square(2)

Out[7]: 4
```

This is the form a function that a lambda expression intends to replicate. A lambda expression can then be written as:

```
In [8]: lambda num: num**2

Out[8]: <function __main__.<lambda>>
```

Note how we get a function back. We can assign this function to a label:

```
In [10]: square = lambda num: num**2

In [11]: square(2)

Out[11]: 4

In [12]: even = lambda x: x%2==0

In [13]: even(3)

Out[13]: False

In [14]: even(4)

Out[14]: True
```

Just like a normal function, we can accept more than one function into a lambda expression:

```
In [15]: add = lambda x,y : x+y

In [16]: add(2,3)

Out[16]: 5
```

I highly recommend reading this blog post at Python Conquers the Universe for a great breakdown on lambda expressions and some explanations of common confusions!

23 Nested Statements and Scope

Now that we have gone over on writing our own functions, its important to understand how Python deals with the variable names you assign. When you create a variable name in Python the name is stored in a *name-space*. Variable names also have a *scope*, the scope determines the visibility of that variable name to other parts of your code.

```
In [17]: x = 25

         def printer():
             x = 50
             return x

         print x
         print printer()
```

```
25
50
```

Interesting! But how does Python know which **x** you're referring to in your code? This is where the idea of scope comes in. Python has a set of rules it follows to decide what variables (such as x in this case) you are referencing in your code. Lets break down the rules:

This idea of scope in your code is very important to understand in order to properly assign and call variable names.

In simple terms, the idea of scope can be described by 3 general rules:

1. Name assignments will create or change local names by default.
2. Name references search (at most) four scopes, these are:
    - local
    - enclosing functions
    - global
    - built-in
3. Names declared in global and nonlocal statements map assigned names to enclosing module and function scopes.

The statement in #2 above can be defined by the LEGB rule.

LEGB Rule.

L: Local — Names assigned in any way within a function (def or lambda)), and not declared global in that function.

E: Enclosing function locals — Name in the local scope of any and all enclosing functions (def or lambda), from inner to outer.

G: Global (module) — Names assigned at the top-level of a module file, or declared global in a def within the file.

B: Built-in (Python) — Names preassigned in the built-in names module : open,range,SyntaxError,...

23.1 Quick examples of LEGB

23.1.1 Local

```
In [18]: # x is local here:
         f = lambda x:x**2

In [20]: name = 'This is a global name'

         def greet():
             # Enclosing function
             name = 'Niket'

             def hello():
                 print 'Hello '+name

             hello()

         greet()
```

```
Hello Niket
```

Note how Niket was used, because the hello() function was enclosed inside of the greet function!

23.1.2 Global

Luckily in Jupyter a quick way to test for global variables is to see if another cell recognizes the variable!

```
In [21]: print name
```

```
This is a global name
```

23.1.3 Built-in

These are the built-in function names in Python (don't overwrite these!)

```
In [22]: len
```

```
Out[22]: <function len>
```

23.2 Local Variables

When you declare variables inside a function definition, they are not related in any way to other variables with the same names used outside the function - i.e. variable names are local to the function. This is called the scope of the variable. All variables have the scope of the block they are declared in starting from the point of definition of the name.

```
In [23]: x = 50

         def func(x):
             print 'x is', x
             x = 2
             print 'Changed local x to', x

         func(x)
         print 'x is still', x
```

```
x is 50
Changed local x to 2
x is still 50
```

23.3 The global statement

If you want to assign a value to a name defined at the top level of the program (i.e. not inside any kind of scope such as functions or classes), then you have to tell Python that the name is not local, but it is global. We do this using the global statement. It is impossible to assign a value to a variable defined outside a function without the global statement.

You can use the values of such variables defined outside the function (assuming there is no variable with the same name within the function). However, this is not encouraged and should be avoided since it becomes unclear to the reader of the program as to where that variable's definition is. Using the global statement makes it amply clear that the variable is defined in an outermost block.

```
In [24]: x = 50

         def func():
             global x
             print 'This function is now using the global x!'
             print 'Because of global x is: ', x
             x = 2
             print 'Ran func(), changed global x to', x

         print 'Before calling func(), x is: ', x
         func()
         print 'Value of x (outside of func()) is: ', x
```

```
Before calling func(), x is:  50
This function is now using the global x!
Because of global x is:  50
Ran func(), changed global x to 2
Value of x (outside of func()) is:  2
```

You can specify more than one global variable using the same global statement e.g. global x, y, z.

24 Object Oriented Programming

For this section we will construct our knowledge of OOP in Python by building on the following topics:

- Objects
- Using the *class* keyword
- Creating class attributes
- Creating methods in a class
- Learning about Inheritance
- Learning about Special Methods for classes

```
In [13]: l = [1,2,3]
```

Remember how we could call methods on a list?

```
In [15]: l.count(1)

Out[15]: 1
```

24.1 Objects

In Python, *everything is an object*. We can use type() to check the type of object.

```
In [18]: print type(1)
         print type([])
         print type(())
         print type({})
```

```
<type 'int'>
<type 'list'>
<type 'tuple'>
<type 'dict'>
```

So we know all these things are objects, so how can we create our own Object types? That is where the *class* keyword comes in. ## class The user defined objects are created using the class keyword. The class is a blueprint that defines a nature of a future object. From classes we can construct instances. An instance is a specific object created from a particular class. For example, above we created the object 'l' which was an instance of a list object.

Let see how we can use **class**:

```
In [19]: # Create a new object type called Sample
         class Sample(object):
             pass

         # Instance of Sample
         x = Sample()

         print type(x)
```

```
<class '__main__.Sample'>
```

By convention we give classes a name that starts with a capital letter. Note how x is now the reference to our new instance of a Sample class. In other words, we **instantiate** the Sample class.

Inside of the class we currently just have pass. But we can define class attributes and methods.

An **attribute** is a characteristic of an object. A **method** is an operation we can perform with the object.

For example we can create a class called Dog. An attribute of a dog may be its breed or its name, while a method of a dog may be defined by a .bark() method which returns a sound.

Let's get a better understanding of attributes through an example.

24.2 Attributes

The syntax for creating an attribute is:

```
self.attribute = something
```

There is a special method called:

```
__init__()
```

This method is used to initialize the attributes of an object. For example:

```
In [20]: class Dog(object):
             def __init__(self,breed):
                 self.breed = breed

         sam = Dog(breed='Lab')
         frank = Dog(breed='Huskie')
```

Lets break down what we have above.The special method

```
__init__()
```

is called automatically right after the object has been created:

```
def __init__(self, breed):
```

Each attribute in a class definition begins with a reference to the instance object. It is by convention named self. The breed is the argument. The value is passed during the class instantiation.

```
 self.breed = breed
```

Now we have created two instances of the Dog class. With two breed types, we can then access these attributes like this:

```
In [22]: sam.breed

Out[22]: 'Lab'
```

```
In [23]: frank.breed

Out[23]: 'Huskie'
```

Note how we don't have any parenthesis after breed, this is because it is an attribute and doesn't take any arguments.

In Python there are also *class object attributes*. These Class Object Attributes are the same for any instance of the class. For example, we could create the attribute *species* for the Dog class. Dogs (regardless of their breed,name, or other attributes will always be mammals. We apply this logic in the following manner:

```
In [24]: class Dog(object):

             # Class Object Attribute
             species = 'mammal'

             def __init__(self,breed,name):
                 self.breed = breed
                 self.name = name

In [25]: sam = Dog('Lab','Sam')

In [26]: sam.name

Out[26]: 'Sam'
```

Note that the Class Object Attribute is defined outside of any methods in the class. Also by convention, we place them first before the init.

```
In [27]: sam.species

Out[27]: 'mammal'
```

24.3 Methods

Methods are functions defined inside the body of a class. They are used to perform operations with the attributes of our objects. Methods are essential in encapsulation concept of the OOP paradigm. This is essential in dividing responsibilities in programming, especially in large applications.

You can basically think of methods as functions acting on an Object that take the Object itself into account through its *self* argument.

Lets go through an example of creating a Circle class:

```
In [28]: class Circle(object):
             pi = 3.14

             # Circle get instantiated with a radius (default is 1)
             def __init__(self, radius=1):
                 self.radius = radius

             # Area method calculates the area. Note the use of self.
```

```
            def area(self):
                return self.radius * self.radius * Circle.pi

            # Method for resetting Radius
            def setRadius(self, radius):
                self.radius = radius

            # Method for getting radius (Same as just calling .radius)
            def getRadius(self):
                return self.radius


        c = Circle()

        c.setRadius(2)
        print 'Radius is: ',c.getRadius()
        print 'Area is: ',c.area()
```

```
Radius is:  2
Area is:  12.56
```

```
In [29]: c.radius
```

```
Out[29]: 2
```

24.4 Inheritance

Inheritance is a way to form new classes using classes that have already been defined. The newly formed classes are called derived classes, the classes that we derive from are called base classes. Important benefits of inheritance are code reuse and reduction of complexity of a program. The derived classes (descendants) override or extend the functionality of base classes (ancestors).

Lets see an example by incorporating our previous work on the Dog class:

```
In [30]: class Animal(object):
            def __init__(self):
                print "Animal created"

            def whoAmI(self):
                print "Animal"

            def eat(self):
                print "Eating"


        class Dog(Animal):
            def __init__(self):
                Animal.__init__(self)
                print "Dog created"
```

```
            def whoAmI(self):
                print "Dog"

            def bark(self):
                print "Woof!"
```

```
In [31]: d = Dog()

Animal created
Dog created


In [32]: d.whoAmI()

Dog


In [33]: d.eat()

Eating


In [34]: d.bark()

Woof!
```

In this example, we have two classes: Animal and Dog. The Animal is the base class, the Dog is the derived class.

The derived class inherits the functionality of the base class.

- It is shown by the eat() method.

The derived class modifies existing behavior of the base class.

- shown by the whoAmI() method.

Finally, the derived class extends the functionality of the base class, by defining a new bark() method.

24.5 Special Methods

Finally lets go over special methods. Classes in Python can implement certain operations with special method names. These methods are not actually called directly but by Python specific language syntax. For example Lets create a Book class:

```
In [35]: class Book(object):
             def __init__(self, title, author, pages):
                 print "A book is created"
                 self.title = title
                 self.author = author
                 self.pages = pages

             def __str__(self):
                 return "Title:%s , author:%s, pages:%s " %(self.title, self.author

             def __len__(self):
                 return self.pages

             def __del__(self):
                 print "A book is destroyed"

In [36]: book = Book("Python Rocks!", "Jose Portilla", 159)

         #Special Methods
         print book
         print len(book)
         del book

A book is created
Title:Python Rocks! , author:Jose Portilla, pages:159
159
A book is destroyed


The __init__(), __str__(), __len__() and the __del__() methods.
```

These special methods are defined by their use of underscores. They allow us to use Python specific functions on objects created through our class.

For more great resources on this topic, check out:
Jeff Knupp's Post
Mozilla's Post
Tutorial's Point
Official Documentation

25 Errors and Exception Handling

```
In [38]: print 'Hello


          File "<ipython-input-38-23e01f0d17c8>", line 1
        print 'Hello
                    ^
    SyntaxError: EOL while scanning string literal
```

Note how we get a SyntaxError, with the further description that it was an EOL (End of Line Error) while scanning the string literal. This is specific enough for us to see that we forgot a single quote at the end of the line. Understanding these various error types will help you debug your code much faster.

This type of error and description is known as an Exception. Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal.

You can check out the full list of built-in exceptions here.

25.1 try and except

The basic terminology and syntax used to handle errors in Python is the **try** and **except** statements. The code which can cause an exception to occue is put in the *try* block and the handling of the exception is the implemented in the *except* block of code. The syntax form is:

```
try:
   You do your operations here...
   ...
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ...
else:
   If there is no exception then execute this block.
```

We can also just check for any exception with just using except.

```
In [41]: try:
             f = open('testfile','w')
             f.write('Test write this')
         except IOError:
             # This will only check for an IOError exception and then execute this
            print "Error: Could not find file or read data"
         else:
            print "Content written successfully"
            f.close()
```

```
Content written successfully
```

Now lets see what would happen if we did not have write permission (opening only with 'r'):

```
In [42]: try:
             f = open('testfile','r')
             f.write('Test write this')
         except IOError:
             # This will only check for an IOError exception and then execute this
            print "Error: Could not find file or read data"
         else:
            print "Content written successfully"
            f.close()
```

```
Error: Could not find file or read data
```

Great! Notice how we only printed a statement! The code still ran and we were able to continue doing actions and running code blocks. This is extremely useful when you have to account for possible input errors in your code. You can be prepared for the error and keep running code, instead of your code just breaking as we saw above.

We could have also just said except: if we weren't sure what exception would occur. For example:

```
In [43]: try:
             f = open('testfile','r')
             f.write('Test write this')
         except:
             # This will check for any exception and then execute this print stater
            print "Error: Could not find file or read data"
         else:
            print "Content written successfully"
            f.close()
```

```
Error: Could not find file or read data
```

Great! Now we don't actually need to memorize that list of exception types! Now what if we kept wanting to run code after the exception occurred? This is where **finally** comes in. ## finally The finally: block of code will always be run regardless if there was an exception in the try code block. The syntax is:

```
try:
   Code block here
   ...
   Due to any exception, this code may be skipped!
finally:
   This code block would always be executed.
```

```
In [44]: try:
             f = open("testfile", "w")
             f.write("Test write statement")
         finally:
             print "Always execute finally code blocks"
```

```
Always execute finally code blocks
```

```
In [45]: def askint():
             try:
                 val = int(raw_input("Please enter an integer: "))
             except:
                 print "Looks like you did not enter an integer!"

             finally:
                 print "Finally, I executed!"
             print val
```

```
In [46]: askint()
```

```
Please enter an integer: 5
Finally, I executed!
5
```

```
In [47]: askint()
```

```
Please enter an integer: test
Looks like you did not enter an integer!
Finally, I executed!
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)

<ipython-input-47-6ee53d339e7e> in <module>()
----> 1 askint()


<ipython-input-45-9a2b39f1a383> in askint()
      7     finally:
      8         print "Finally, I executed!"
----> 9     print val


UnboundLocalError: local variable 'val' referenced before assignment
```

```
In [48]: def askint():
             while True:
                 try:
                     val = int(raw_input("Please enter an integer: "))
                 except:
                     print "Looks like you did not enter an integer!"
                     continue
                 else:
                     print 'Yep thats an integer!'
                     break
                 finally:
                     print "Finally, I executed!"
                 print val

In [49]: askint()

Please enter an integer: f
Looks like you did not enter an integer!
Finally, I executed!
Please enter an integer: test
Looks like you did not enter an integer!
Finally, I executed!
Please enter an integer: 5
Yep thats an integer!
Finally, I executed!
```

In [ ]: