Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Numbers.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Numbers.ipynb)

Numbers and more in Python!

In this lecture, we will learn about numbers in Python and how to use them.

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
4.) Object Assignment in Python
```

Types of numbers

Python has various "types" of numbers (numeric literals). We'll mainly focus on integers and floating point numbers.

Integers are just whole numbers, positive or negative. For example: 2 and -2 are examples of integers.

Floating point numbers in Python are notable because they have a decimal point in them, or use an exponential (e) to define the number. For example 2.0 and -2.1 are examples of floating point numbers. 4E2 (4 times 10 to the power of 2) is also an example of a floating point number in Python.

Throughout this course we will be mainly working with integers or simple float number types.

Here is a table of the two main types we will spend most of our time working with some examples:

<table border = “1“>

Numbers in Python

Examples Number "Type"

1,2,-5,1000 Integers

1.2,-0.5,2e2,3E2 Floating-point numbers

</table>

Now let's start with some basic arithmetic.

Basic Arithmetic

In [1]:
```
# Addition
2+1
```

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

In [2]:
```
# Subtraction
2-1
```

Out[2]:
```
1
```

In [3]:
```
# Multiplication
2*2
```

Out[3]:
```
4
```

In [4]:
```
# Division
3/2
```

Out[4]:
```
1
```

Python 3 Alert!

Whoa! What just happened? Last time I checked, 3 divided by 2 is equal 1.5 not 1!

The reason we get this result is because we are using Python 2. In Python 2, the / symbol performs what is known as "*classic*" division, this means that the decimal points are truncated (cut off). In Python 3 however, a single / performs "*true*" division. So you would get 1.5 if you had inputed 3/2 in Python 3.

So what do we do if we are using Python 2 to avoid this?

There are two options:

Specify one of the numbers to be a float:

In [11]:
```
# Specifying one of the numbers as a float
3.0/2
```

Out[11]:
```
1.5
```

In [12]:
```
# Works for either number
3/2.0
```

Out[12]:
```
1.5
```

We could also "cast" the type using a function that basically turns integers into floats. This function, unsurprisingly, is called float().

In [14]:
```
# We can use this float() function to cast integers as floats:
float(3)/2
```

Out[14]:
```
1.5
```

We will go over functions in much more detail later on in this course, so don't worry if you are confused by the syntax here. Consider this a sneak preview.

One more "sneak preview" we can use to deal with classic division in Python 2 is importing from a module called **future**.

This is a module in Python 2 that has Python 3 functions, this basically allows you to import Python 3 functions into Python 2. We will go over imports and modules later in the course, so don't worry about fully understanding the import statement right now!

In [15]:
```
from __future__ import division
3/2
```

Out[15]:
```
1.5
```

When you import division from the **future** you won't need to worry about classic division occurring anymore anywhere in your code!

Arithmetic continued

In [16]:
```
# Powers
2**3
```

Out[16]:
```
8
```

In [17]:
```
# Can also do roots this way
4**0.5
```

Out[17]:
```
2.0
```

In [18]:
```
# Order of Operations followed in Python
2 + 10 * 10 + 3
```

Out[18]:
```
105
```

In [19]:
```
# Can use parenthesis to specify orders
(2+10) * (10+3)
```

Out[19]:
```
156
```

Variable Assignments

Now that we've seen how to use numbers in Python as a calculator let's see how we can assign names and create variables.

We use a single equals sign to assign labels to variables. Let's see a few examples of how we can do this.

In [37]:
```
# Let's create an object called "a" and assign it the number 5
a = 5
```

Now if I call *a* in my Python script, Python will treat it as the number 5.

In [38]:
```
# Adding the objects
a+a
```

Out[38]:
```
10
```

What happens on reassignment? Will Python let us write it over?

In [39]:
```
# Reassignment
a = 10
```

In [40]:
```
# Check
a
```

Out[40]:
```
10
```

Yes! Python allows you to write over assigned variable names. We can also use the variables themselves when doing the reassignment. Here is an example of what I mean:

In [41]:
```
# Check
a
```

Out[41]:
```
10
```

In [42]:
```
# Use A to redefine A
a = a + a
```

In [43]:
```
# Check
a
```

Out[43]:
```
20
```

The names you use when creating these labels need to follow a few rules:

1. Names can not start with a number.
2. There can be no spaces in the name, use _ instead.
3. Can't use any of these symbols :'",<>/?|\()!@#$%^&*~-+
3. It's considered best practice (PEP8) that the names are lowercase.

Using variable names can be a very useful way to keep track of different variables in Python. For example:

In [44]:
```
# Use object names to keep better track of what's going on in your code!
my_income = 100

tax_rate = 0.1

my_taxes = my_income*tax_rate
```

In [46]:
```
# Show my taxes!
my_taxes
```

Out[46]:
```
10.0
```

So what have we learned? We learned some of the basics of numbers in Python. We also learned how to do arithmetic and use Python as a basic calculator. We then wrapped it up with learning about Variable Assignment in Python.

Up next we'll learn about Strings!

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Strings.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Strings.ipynb)

Strings

Strings are used in Python to record text information, such as name. Strings in Python are actually a *sequence*, which basically means Python keeps track of every element in the string as a sequence. For example, Python understands the string "hello' to be a sequence of letters in a specific order. This means we will be able to use indexing to grab particular letters (like the first letter, or the last letter).

This idea of a sequence is an important one in Python and we will touch upon it later on in the future.

In this lecture 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
```

Creating a String

To create a string in Python you need to use either single quotes or double quotes. For example:

In [1]:
```
# Single word
'hello'
```

Out[1]:
```
'hello'
```

In [2]:
```
# Entire phrase
'This is also a string'
```

Out[2]:
```
'This is also a string'
```

In [3]:
```
# We can also use double quote
"String built with double quotes"
```

Out[3]:
```
'String built with double quotes'
```

In [4]:
```
# Be careful with quotes!
' I'm using single quotes, but will create an error'
```

```
  File "<ipython-input-4-6565b0b7b5e3>", line 2
    ' I'm using single quotes, but will create an error'
        ^
SyntaxError: invalid syntax
```

The reason for the error above is because the single quote in I'm stopped the string. You can use combinations of double and single quotes to get the complete statement.

In [10]:
```
"Now I'm ready to use the single quotes inside a string!"
```

Out[10]:
```
"Now I'm ready to use the single quotes inside a string!"
```

Now let's learn about printing strings!

Printing a String

Using Jupyter notebook with just a string in a cell will automatically output strings, but the correct way to display strings in your output is by using a print function.

In [11]:
```
# We can simply declare a string
'Hello World'
```

Out[11]:
```
'Hello World'
```

In [12]:
```
# note that we can't output multiple strings this way
'Hello World 1'
'Hello World 2'
```

Out[12]:
```
'Hello World 2'
```

We can use a print statement to print a string.

In [13]:
```
print 'Hello World 1'
print 'Hello World 2'
print 'Use \n to print a new line'
print '\n'
print 'See what I mean?'
```

```
Hello World 1
Hello World 2
Use
 to print a new line


See what I mean?
```

Python 3 Alert!

Something to note. In Python 3, print is a function, not a statement. So you would print statements like this: print('Hello World')

If you want to use this functionality in Python2, you can import form the **future** module.

A word of caution, after importing this you won't be able to choose the print statement method anymore. So pick whichever one you prefer depending on your Python installation and continue on with it.

In [32]:
```
# To use print function from Python 3 in Python 2
from __future__ import print_function

print('Hello World')
```

```
Hello World
```

String Basics

We can also use a function called len() to check the length of a string!

In [33]:
```
len('Hello World')
```

Out[33]:
```
11
```

String Indexing

We know strings are a sequence, which means Python can use indexes to call parts of the sequence. Let's learn how this works.

In Python, we use brackets [] after an object to call its index. We should also note that indexing starts at 0 for Python. Let's create a new object called s and the walk through a few examples of indexing.

In [1]:
```
# Assign s as a string
s = 'Hello World'
```

In [35]:
```
#Check
s
```

Out[35]:
```
'Hello World'
```

In [36]:
```
# Print the object
print(s)
```

```
Hello World
```

Let's start indexing!

In [21]:
```
# Show first element (in this case a letter)
s[0]
```

Out[21]:
```
'H'
```

In [22]:
```
s[1]
```

Out[22]:
```
'e'
```

In [23]:
```
s[2]
```

Out[23]:
```
'l'
```

We can use a : to perform *slicing* which grabs everything up to a designated point. For example:

In [24]:
```
# Grab everything past the first term all the way to the length of s which is len(s)
s[1:]
```

Out[24]:
```
'ello World'
```

In [25]:
```
# Note that there is no change to the original s
s
```

Out[25]:
```
'Hello World'
```

In [26]:
```
# Grab everything UP TO the 3rd index
s[:3]
```

Out[26]:
```
'Hel'
```

Note the above slicing. Here we're telling Python to grab everything from 0 up to 3. It doesn't include the 3rd index. You'll notice this a lot in Python, where statements and are usually in the context of "up to, but not including".

In [27]:
```
#Everything
s[:]
```

Out[27]:
```
'Hello World'
```

We can also use negative indexing to go backwards.

In [28]:
```
# Last letter (one index behind 0 so it loops back around)
s[-1]
```

Out[28]:
```
'd'
```

In [29]:
```
# Grab everything but the last letter
s[:-1]
```

Out[29]:
```
'Hello Worl'
```

We can also use index and slice notation to grab elements of a sequence by a specified step size (the default is 1). For instance we can use two colons in a row and then a number specifying the frequency to grab elements. For example:

In [42]:
```
# Grab everything, but go in steps size of 1
s[::1]
```

Out[42]:
```
'Hello World'
```

In [46]:
```
# Grab everything, but go in step sizes of 2
s[::2]
```

Out[46]:
```
'HloWrd'
```

In [47]:
```
# We can use this to print a string backwards
s[::-1]
```

Out[47]:
```
'dlroW olleH'
```

String Properties

Its important to note that strings have an important property known as immutability. This means that once a string is created, the elements within it can not be changed or replaced. For example:

In [48]:
```
s
```

Out[48]:
```
'Hello World'
```

In [49]:
```
# Let's try to change the first letter to 'x'
s[0] = 'x'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-49-3a9c668aa5ab> in <module>()
      1 # Let's try to change the first letter to 'x'
----> 2 s[0] = 'x'

TypeError: 'str' object does not support item assignment
```

Notice how the error tells us directly what we can't do, change the item assignment!

Something we can do is concatenate strings!

In [50]:
```
s
```

Out[50]:
```
'Hello World'
```

In [52]:
```
# Concatenate strings!
s + ' concatenate me!'
```

Out[52]:
```
'Hello World concatenate me!'
```

In [53]:
```
# We can reassign s completely though!
s = s + ' concatenate me!'
```

In [54]:
```
print(s)
```

```
Hello World concatenate me!
```

In [58]:
```
s
```

Out[58]:
```
'Hello World concatenate me!'
```

We can use the multiplication symbol to create repetition!

In [59]:
```
letter = 'z'
```

In [60]:
```
letter*10
```

Out[60]:
```
'zzzzzzzzzz'
```

Basic Built-in String methods

Objects in Python usually have built-in methods. These methods are functions inside the object (we will learn about these in much more depth later) that can perform actions or commands on the object itself.

We call methods with a period and then the method name. Methods are in the form:

object.method(parameters)

Where parameters are extra arguments we can pass into the method. Don't worry if the details don't make 100% sense right now. Later on we will be creating our own objects and functions!

Here are some examples of built-in methods in strings:

In [61]:
```
s
```

Out[61]:
```
'Hello World concatenate me!'
```

In [62]:
```
# Upper Case a string
s.upper()
```

Out[62]:
```
'HELLO WORLD CONCATENATE ME!'
```

In [65]:
```
# Lower case
s.lower()
```

Out[65]:
```
'hello world concatenate me!'
```

In [67]:
```
# Split a string by blank space (this is the default)
s.split()
```

Out[67]:
```
['Hello', 'World', 'concatenate', 'me!']
```

In [68]:
```
# Split by a specific element (doesn't include the element that was split on)
s.split('W')
```

Out[68]:
```
['Hello ', 'orld concatenate me!']
```

There are many more methods than the ones covered here. Visit the advanced String section to find out more!

Print Formatting

We can use the .format() method to add formatted objects to printed string statements.

The easiest way to show this is through an example:

In [79]:
```
'Insert another string with curly brackets: {}'.format('The inserted string')
```

Out[79]:
```
'Insert another string with curly brackets: The inserted string'
```

We will revisit this string formatting topic in later sections when we are building our projects!

Next up: Lists!

In [ ]:
```

```

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Print Formatting.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Print Formatting.ipynb)

Print Formatting

In this lecture we will briefly cover the various ways to format your print statements. As you code more and more, you will probably want to have print statements that can take in a variable into a printed string statement.

The most basic example of a print statement is:

In [17]:
```
print 'This is a string'
```

```
This is a string
```

Strings

You can use the %s to format strings into your print statements.

In [4]:
```
s = 'STRING'
print 'Place another string with a mod and s: %s' %(s)
```

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

Floating Point Numbers

Floating point numbers use the format %n1.n2f where the n1 is the total minimum number of digits the string should contain (these may be filled with whitespace if the entire number does not have this many digits. The n2 placeholder stands for how many numbers to show past the decimal point. Lets see some examples:

In [12]:
```
print 'Floating point numbers: %1.2f' %(13.144)
```

```
Floating point numbers: 13.14
```

In [13]:
```
print 'Floating point numbers: %1.0f' %(13.144)
```

```
Floating point numbers: 13
```

In [14]:
```
print 'Floating point numbers: %1.5f' %(13.144)
```

```
Floating point numbers: 13.14400
```

In [15]:
```
print 'Floating point numbers: %10.2f' %(13.144)
```

```
Floating point numbers:      13.14
```

In [19]:
```
print 'Floating point numbers: %25.2f' %(13.144)
```

```
Floating point numbers:                     13.14
```

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(). We will learn more about these functions later on in the course, but you should note you can actually pass almost any Python object with these two methods and it will work:

In [23]:
```
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 [24]:
```
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'
```

Multiple Formatting

Pass a tuple to the modulo symbol to place multiple formats in your print statements:

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

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

Using the string .format() method

The best way to format objects into your strings for print statements is using the format method. The syntax is:

```
'String here {var1} then also {var2}'.format(var1='something1',var2='something2')
```

Lets see some examples:

In [27]:
```
print 'This is a string with an {p}'.format(p='insert')
```

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

In [28]:
```
# Multiple times:
print 'One: {p}, Two: {p}, Three: {p}'.format(p='Hi!')
```

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

In [29]:
```
# Several Objects:
print 'Object 1: {a}, Object 2: {b}, Object 3: {c}'.format(a=1,b='two',c=12.3)
```

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

That is the basics of string formatting! Remember that Python 3 uses a print() function, not the print statement!

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Lists.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Lists.ipynb)

Lists

Earlier when discussing strings we introduced the concept of a *sequence* in Python. Lists can be thought of the most general version of a *sequence* in Python. 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
```

Lists are constructed with brackets [] and commas separating every element in the list.

Let's go ahead and see how we can construct lists!

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

We just created a list of integers, but lists can actually hold different object types. For example:

In [4]:
```
my_list = ['A string',23,100.232,'o']
```

Just like strings, the len() function will tell you how many items are in the sequence of the list.

In [6]:
```
len(my_list)
```

Out[6]:
```
4
```

Indexing and Slicing

Indexing and slicing works just like in strings. Let's make a new list to remind ourselves of how this works:

In [7]:
```
my_list = ['one','two','three',4,5]
```

In [10]:
```
# Grab element at index 0
my_list[0]
```

Out[10]:
```
'one'
```

In [11]:
```
# Grab index 1 and everything past it
my_list[1:]
```

Out[11]:
```
['two', 'three', 4, 5]
```

In [13]:
```
# Grab everything UP TO index 3
my_list[:3]
```

Out[13]:
```
['one', 'two', 'three']
```

We can also use + to concatenate lists, just like we did for strings.

In [14]:
```
my_list + ['new item']
```

Out[14]:
```
['one', 'two', 'three', 4, 5, 'new item']
```

Note: This doesn't actually change the original list!

In [15]:
```
my_list
```

Out[15]:
```
['one', 'two', 'three', 4, 5]
```

You would have to reassign the list to make the change permanent.

In [16]:
```
# Reassign
my_list = my_list + ['add new item permanently']
```

In [18]:
```
my_list
```

Out[18]:
```
['one', 'two', 'three', 4, 5, 'add new item permanently']
```

We can also use the * for a duplication method similar to strings:

In [20]:
```
# Make the list double
my_list * 2
```

Out[20]:
```
['one',
 'two',
 'three',
 4,
 5,
 'add new item permanently',
 'one',
 'two',
 'three',
 4,
 5,
 'add new item permanently']
```

In [23]:
```
# Again doubling not permanent
my_list
```

Out[23]:
```
['one', 'two', 'three', 4, 5, 'add new item permanently']
```

Basic List Methods

If you are familiar with another programming language, you might start to draw parallels between arrays in another language and lists in Python. Lists in Python however, tend to be more flexible than arrays in other languages for a two good reasons: they have no fixed size (meaning we don't have to specify how big a list will be), and they have no fixed type constraint (like we've seen above).

Let's go ahead and explore some more special methods for lists:

In [42]:
```
# Create a new list
l = [1,2,3]
```

Use the **append** method to permanently add an item to the end of a list:

In [43]:
```
# Append
l.append('append me!')
```

In [44]:
```
# Show
l
```

Out[44]:
```
[1, 2, 3, 'append me!']
```

Use **pop** to "pop off" an item from the list. By default pop takes off the last index, but you can also specify which index to pop off. Let's see an example:

In [45]:
```
# Pop off the 0 indexed item
l.pop(0)
```

Out[45]:
```
1
```

In [46]:
```
# Show
l
```

Out[46]:
```
[2, 3, 'append me!']
```

In [47]:
```
# Assign the popped element, remember default popped index is -1
popped_item = l.pop()
```

In [48]:
```
popped_item
```

Out[48]:
```
'append me!'
```

In [49]:
```
# Show remaining list
l
```

Out[49]:
```
[2, 3]
```

It should also be noted that lists indexing will return an error if there is no element at that index. For example:

In [50]:
```
l[100]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-50-3e7ce3111e95> in <module>()
----> 1 l[100]

IndexError: list index out of range
```

We can use the **sort** method and the **reverse** methods to also effect your lists:

In [51]:
```
new_list = ['a','e','x','b','c']
```

In [52]:
```
#Show
new_list
```

Out[52]:
```
['a', 'e', 'x', 'b', 'c']
```

In [53]:
```
# Use reverse to reverse order (this is permanent!)
new_list.reverse()
```

In [54]:
```
new_list
```

Out[54]:
```
['c', 'b', 'x', 'e', 'a']
```

In [55]:
```
# Use sort to sort the list (in this case alphabetical order, but for numbers it will go ascending)
new_list.sort()
```

In [56]:
```
new_list
```

Out[56]:
```
['a', 'b', 'c', 'e', 'x']
```

Nesting Lists

A great feature of of Python data structures is that they support *nesting*. This means we can have data structures within data structures. For example: A list inside a list.

Let's see how this works!

In [57]:
```
# 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 [60]:
```
# Show
matrix
```

Out[60]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Now we can again use indexing to grab elements, but now there are two levels for the index. The items in the matrix object, and then the items inside that list!

In [61]:
```
# Grab first item in matrix object
matrix[0]
```

Out[61]:
```
[1, 2, 3]
```

In [62]:
```
# Grab first item of the first item in the matrix object
matrix[0][0]
```

Out[62]:
```
1
```

List Comprehensions

Python has an advanced feature called list comprehensions. They allow for quick construction of lists. To fully understand list comprehensions we need to understand for loops. So don't worry if you don't completely understand this section, and feel free to just skip it since we will return to this topic later.

But in case you want to know now, here are a few examples!

In [63]:
```
# Build a list comprehension by deconstructing a for loop within a []
first_col = [row[0] for row in matrix]
```

In [64]:
```
first_col
```

Out[64]:
```
[1, 4, 7]
```

We used list comprehension here to grab the first element of every row in the matrix object. We will cover this in much more detail later on!

For more advanced methods and features of lists in Python, check out the advanced list section later on in this course!

In [ ]:
```
```

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Dictionaries.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Dictionaries.ipynb)

Dictionaries

We've been learning about *sequences* in Python but now we're going to switch gears and learn about *mappings* in Python. 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

So what are mappings? 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.

A Python dictionary consists of a key and then an associated value. That value can be almost any Python object.

Constructing a Dictionary

Let's see how we can construct dictionaries to get a better understanding of how they work!

In [1]:
```
# Make a dictionary with {} and : to signify a key and a value
my_dict = {'key1':'value1','key2':'value2'}
```

In [2]:
```
# Call values by their key
my_dict['key2']
```

Out[2]:
```
'value2'
```

Its important to note that dictionaries are very flexible in the data types they can hold. For example:

In [13]:
```
my_dict = {'key1':123,'key2':[12,23,33],'key3':['item0','item1','item2']}
```

In [4]:
```
#Lets call items from the dictionary
my_dict['key3']
```

Out[4]:
```
['item0', 'item1', 'item2']
```

In [5]:
```
# Can call an index on that value
my_dict['key3'][0]
```

Out[5]:
```
'item0'
```

In [7]:
```
#Can then even call methods on that value
my_dict['key3'][0].upper()
```

Out[7]:
```
'ITEM0'
```

We can effect the values of a key as well. For instance:

In [14]:
```
my_dict['key1']
```

Out[14]:
```
123
```

In [15]:
```
# Subtract 123 from the value
my_dict['key1'] = my_dict['key1'] - 123
```

In [16]:
```
#Check
my_dict['key1']
```

Out[16]:
```
0
```

A quick note, Python has a built-in method of doing a self subtraction or addition (or multiplication or division). We could have also used += or -= for the above statement. For example:

In [17]:
```
# Set the object equal to itself minus 123
my_dict['key1'] -= 123
my_dict['key1']
```

Out[17]:
```
-123
```

We can also create keys by assignment. For instance if we started off with an empty dictionary, we could continually add to it:

In [21]:
```
# Create a new dictionary
d = {}
```

In [22]:
```
# Create a new key through assignment
d['animal'] = 'Dog'
```

In [24]:
```
# Can do this with any object
d['answer'] = 42
```

In [25]:
```
#Show
d
```

Out[25]:
```
{'animal': 'Dog', 'answer': 42}
```

Nesting with Dictionaries

Hopefully your starting to see how powerful Python is with its flexibility of nesting objects and calling methods on them. Let's see a dictionary nested inside a dictionary:

In [26]:
```
# Dictionary nested inside a dictionary nested in side a dictionary
d = {'key1':{'nestkey':{'subnestkey':'value'}}}
```

Wow! Thats a quite the inception of dictionaries! Let's see how we can grab that value:

In [29]:
```
# Keep calling the keys
d['key1']['nestkey']['subnestkey']
```

Out[29]:
```
'value'
```

A few Dictionary Methods

There are a few methods we can call on a dictionary. Let's get a quick introduction to a few of them:

In [30]:
```
# Create a typical dictionary
d = {'key1':1,'key2':2,'key3':3}
```

In [35]:
```
# Method to return a list of all keys
d.keys()
```

Out[35]:
```
['key3', 'key2', 'key1']
```

In [36]:
```
# Method to grab all values
d.values()
```

Out[36]:
```
[3, 2, 1]
```

In [33]:
```
# Method to return tuples of all items  (we'll learn about tuples soon)
d.items()
```

Out[33]:
```
[('key3', 3), ('key2', 2), ('key1', 1)]
```

Hopefully you now have a good basic understanding how to construct dictionaries. There's a lot more to go into here, but we will revisit dictionaries at later time. After this section all you need to know is how to create a dictionary and how to retrieve values from it.

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Tuples.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Tuples.ipynb)

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.

You'll have an intuition of how to use tuples based on what you've learned about lists. We can treat them very similarly with the major distinction being that tuples are immutable.

Constructing Tuples

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

In [5]:
```
# Can create a tuple with mixed types
t = (1,2,3)
```

In [6]:
```
# Check len just like a list
len(t)
```

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

In [8]:
```
# Can also mix object types
t = ('one',2)

# Show
t
```

Out[8]:
```
('one', 2)
```

In [4]:
```
# Use indexing just like we did in lists
t[0]
```

Out[4]:
```
'one'
```

In [11]:
```
# Slicing just like a list
t[-1]
```

Out[11]:
```
2
```

Basic Tuple Methods

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

In [12]:
```
# Use .index to enter a value and return the index
t.index('one')
```

Out[12]:
```
0
```

In [13]:
```
# Use .count to count the number of times a value appears
t.count('one')
```

Out[13]:
```
1
```

Immutability

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

In [14]:
```
t[0]= 'change'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-93def5f9b4bd> in <module>()
----> 1 t[0]= 'change'

TypeError: 'tuple' object does not support item assignment
```

Because of this immutability, tuples can't grow. Once a tuple is made we can not add to it.

In [15]:
```
t.append('nope')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-15-799b3447c4d9> in <module>()
----> 1 t.append('nope')

AttributeError: 'tuple' object has no attribute 'append'
```

When to use Tuples

You may be wondering, "Why bother using tuples when they have fewer available methods?" To be honest, tuples are not used as often as lists in programming, but are used when immutability is necessary. If in your program you are passing around an object and need to make sure it does not get changed, then tuple become your solution. It provides a convenient source of data integrity.

You should now be able to create and use tuples in your programming as well as have an understanding of their immutability.

Up next Files!

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Files.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Files.ipynb)

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!

iPython Writing a File

In [6]:
```
%%writefile test.txt
Hello, this is a quick test file
```

```
Overwriting test.txt
```

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 [14]:
```
# Open the text.txt we made earlier
my_file = open('test.txt')
```

In [15]:
```
# We can now read the file
my_file.read()
```

Out[15]:
```
'Hello, this is a quick test file'
```

In [16]:
```
# But what happens if we try to read it again?
my_file.read()
```

Out[16]:
```
''
```

This happens because you can imagine the reading "cursor" is at the end of the file after having read it. So there is nothing left to read. We can reset the "cursor" like this:

In [42]:
```
# Seek to the start of file (index 0)
my_file.seek(0)
```

In [19]:
```
# Now read again
my_file.read()
```

Out[19]:
```
'Hello, this is a quick test file'
```

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

In [26]:
```
# Readlines returns a list of the lines in the file.
my_file.readlines()
```

Out[26]:
```
['Hello, this is a quick test file']
```

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 [39]:
```
# Add a second argument to the function, 'w' which stands for write
my_file = open('test.txt','w+')
```

In [40]:
```
# Write to the file
my_file.write('This is a new line')
```

In [43]:
```
# Read the file
my_file.read()
```

Out[43]:
```
'This is a new line'
```

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 [44]:
```
%%writefile test.txt
First Line
Second Line
```

```
Overwriting test.txt
```

Now we can use a little bit of flow to tell the program to for through every line of the file and do something:

In [45]:
```
for line in open('test.txt'):
    print line
```

```
First Line

Second Line
```

Don't worry about fully understanding this yet, for loops are coming up soon. But we'll break down what we did above. We said that for every line in this text file, go ahead and print that 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 memory.
3.) Notice the indent on the second line for print. This whitespace is required in Python.

We'll learn a lot more about this later, but up next: Sets and Booleans!

In [46]:
```
# Pertaining to the first point above
for asdf in open('test.txt'):
    print asdf
```

```
First Line

Second Line
```

In [ ]:
```

```

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Sets and Booleans.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Sets and Booleans.ipynb)

Set and Booleans ¶

There are two other object types in Python that we should quickly cover. Sets and Booleans.

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

In [1]:
```
x = set()
```

In [3]:
```
# We add to sets with the add() method
x.add(1)
```

In [4]:
```
#Show
x
```

Out[4]:
```
{1}
```

Note the curly brackets. This does not indicate a dictionary! Although you can draw analogies as a set being a dictionary with only keys.

We know that a set has only unique entries. So what happens when we try to add something that is already in a set?

In [5]:
```
# Add a different element
x.add(2)
```

In [6]:
```
#Show
x
```

Out[6]:
```
{1, 2}
```

In [7]:
```
# Try to add the same element
x.add(1)
```

In [9]:
```
#Show
x
```

Out[9]:
```
{1, 2}
```

Notice how it won't place another 1 there. That's because a set is only concerned with unique elements! We can cast a list with multiple repeat elements to a set to get the unique elements. For example:

In [10]:
```
# Create a list with repeats
l = [1,1,2,2,3,4,5,6,1,1]
```

In [12]:
```
# Cast as set to get unique values
set(l)
```

Out[12]:
```
{1, 2, 3, 4, 5, 6}
```

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. Let's walk through a few quick examples of Booleans (we will dive deeper into them later in this course).

In [13]:
```
# Set object to be a boolean
a = True
```

In [16]:
```
#Show
a
```

Out[16]:
```
True
```

We can also use comparison operators to create booleans. We will go over all the comparison operators later on in the course.

In [17]:
```
# Output is boolean
1 > 2
```

Out[17]:
```
False
```

We can use None as a placeholder for an object that we don't want to reassign yet:

In [18]:
```
# None placeholder
b = None
```

Thats it! You should now have a basic understanding of Python objects and data structure types. Next, go ahead and do the assessment test!

In [ ]:
```

```

Objects and Data Structures Assessment Test

Test your knowledge.

Answer the following questions

Write a brief description of all the following Object Types and Data Structures we've learned about:

Numbers:

Strings:

Lists:

Tuples:

Dictionaries:

Numbers

Write an equation that uses multiplication, division, an exponent, addition, and subtraction that is equal to 100.25.

Hint: This is just to test your memory of the basic arithmetic commands, work backwards from 100.25

In [ ]:

Explain what the cell below will produce and why. Can you change it so the answer is correct?

In [ ]:
```
2/3
```

Answer these 3 questions without typing code. Then type code to check your answer.

```
What is the value of the expression 4 * (6 + 5)

What is the value of the expression 4 * 6 + 5

What is the value of the expression 4 + 6 * 5
```

In [ ]:

What is the *type* of the result of the expression 3 + 1.5 + 4?

What would you use to find a number's square root, as well as its square?

In [ ]:

Strings

Given the string 'hello' give an index command that returns 'e'. Use the code below:

In [ ]:
```
s = 'hello'
# Print out 'e' using indexing

# Code here
```

Reverse the string 'hello' using indexing:

In [ ]:
```
s ='hello'

# Reverse the string using indexing

# Code here
```

Given the string hello, give two methods of producing the letter 'o' using indexing.

In [ ]:
```
s ='hello'

# Print out the

# Code here
```

Lists

Build this list [0,0,0] two separate ways.

In [ ]:
```

```

Reassign 'hello' in this nested list to say 'goodbye' item in this list:

In [14]:
```
l = [1,2,[3,4,'hello']]
```

Sort the list below:

In [15]:
```
l = [3,4,5,5,6]
```

Dictionaries

Using keys and indexing, grab the 'hello' from the following dictionaries:

In [10]:
```
d = {'simple_key':'hello'}
# Grab 'hello'
```

In [12]:
```
d = {'k1':{'k2':'hello'}}
# Grab 'hello'
```

In [13]:
```
# Getting a little tricker
d = {'k1':[{'nest_key':['this is deep',['hello']]}]}

#Grab hello
```

In [ ]:
```
# This will be hard and annoying!
d = {'k1':[1,2,{'k2':['this is tricky',{'tough':[1,2,['hello']]}]}]}
```

Can you sort a dictionary? Why or why not?

Tuples

What is the major difference between tuples and lists?

How do you create a tuple?

Sets

What is unique about a set?

Use a set to find the unique values of the list below:

In [ ]:
```
l = [1,2,2,33,4,4,11,22,3,3,2]
```

Booleans

For the following quiz questions, we will get a preview 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.	
<>	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.

What will be the resulting Boolean of the following pieces of code (answer fist then check by typing it in!)

In [ ]:
```
# Answer before running cell
2 > 3
```

In [17]:
```
# Answer before running cell
3 <= 2
```

In [18]:
```
# Answer before running cell
3 == 2.0
```

In [ ]:
```
# Answer before running cell
3.0 == 3
```

In [ ]:
```
# Answer before running cell
4**0.5 != 2
```

Final Question: What is the boolean output of the cell block below?

In [ ]:
```
# two nested lists
l_one = [1,2,[3,4]]
l_two = [1,2,{'k1':4}]

#True or False?
l_one[2][0] >= l_two[2]['k1']
```

Great Job on your first assessment!

Objects and Data Structures Assessment Test

Test your knowledge.

Answer the following questions

Write a brief description of all the following Object Types and Data Structures we've learned about:

For the full answers, review the Jupyter notebook introductions of each topic!

Numbers (http://nbviewer.ipython.org/github/jmportilla/Complete-Python-Bootcamp/blob/master/Numbers.ipynb)

Strings (http://nbviewer.ipython.org/github/jmportilla/Complete-Python-Bootcamp/blob/master/Strings.ipynb)

Lists (http://nbviewer.ipython.org/github/jmportilla/Complete-Python-Bootcamp/blob/master/Lists.ipynb)

Tuples (http://nbviewer.ipython.org/github/jmportilla/Complete-Python-Bootcamp/blob/master/Tuples.ipynb)

Dictionaries (http://nbviewer.ipython.org/github/jmportilla/Complete-Python-Bootcamp/blob/master/Dictionaries.ipynb)

Numbers

Write an equation that uses multiplication, division, an exponent, addition, and subtraction that is equal to 100.25.

Hint: This is just to test your memory of the basic arithmetic commands, work backwards from 100.25

In [10]:
```
# Your answer is probably different
(20000 - (10 ** 2) / 12 * 34) - 19627.75
```

Out[10]:
```
100.25
```

Explain what the cell below will produce and why. Can you change it so the answer is correct?

In [11]:
```
2/3
```

Out[11]:
```
0
```

Answer: Because Python 2 performs classic division for integers. Use floats to perform true division. For example: 2.0/3

Answer these 3 questions without typing code. Then type code to check your answer.

```
What is the value of the expression 4 * (6 + 5)

What is the value of the expression 4 * 6 + 5

What is the value of the expression 4 + 6 * 5
```

In [16]:
```
4 * (6 + 5)
```

Out[16]:
```
44
```

In [17]:
```
4 * 6 + 5
```

Out[17]:
```
29
```

In [18]:
```
4 + 6 * 5
```

Out[18]:
```
34
```

What is the *type* of the result of the expression 3 + 1.5 + 4?

Answer: Floating Point Number

What would you use to find a number's square root, as well as its square?

In [14]:
```
100 ** 0.5
```

Out[14]:
```
10.0
```

In [12]:
```
10 ** 2
```

Out[12]:
```
100
```

Strings

Given the string 'hello' give an index command that returns 'e'. Use the code below:

In [19]:
```
s = 'hello'
# Print out 'e' using indexing
s[1]
```

Out[19]:
```
'e'
```

Reverse the string 'hello' using indexing:

In [21]:
```
s ='hello'

# Reverse the string using indexing

s[::-1]
```

Out[21]:
```
'olleh'
```

Given the string hello, give two methods of producing the letter 'o' using indexing.

In [22]:
```
s ='hello'

# Print out the

s[-1]
```

Out[22]:
```
'o'
```

In [23]:
```
s[4]
```

Out[23]:
```
'o'
```

Lists

Build this list [0,0,0] two separate ways.

In [25]:
```
#Method 1
[0]*3
```

Out[25]:
```
[0, 0, 0]
```

In [27]:
```
#Method 2
l = [0,0,0]
l
```

Out[27]:
```
[0, 0, 0]
```

Reassign 'hello' in this nested list to say 'goodbye' item in this list:

In [28]:
```
l = [1,2,[3,4,'hello']]
```

In [31]:
```
l[2][2] = 'goodbye'
```

In [32]:
```
l
```

Out[32]:
```
[1, 2, [3, 4, 'goodbye']]
```

Sort the list below:

In [33]:
```
l = [3,4,5,5,6]
```

In [38]:
```
#Method 1
sorted(l)
```

Out[38]:
```
[3, 4, 5, 5, 6]
```

In [40]:
```
#Method 2
l.sort()
l
```

Out[40]:
```
[3, 4, 5, 5, 6]
```

Dictionaries

Using keys and indexing, grab the 'hello' from the following dictionaries:

In [41]:
```
d = {'simple_key':'hello'}
# Grab 'hello'
```

In [42]:
```
d['simple_key']
```

Out[42]:
```
'hello'
```

In [43]:
```
d = {'k1':{'k2':'hello'}}
# Grab 'hello'
```

In [44]:
```
d['k1']['k2']
```

Out[44]:
```
'hello'
```

In [45]:
```
# Getting a little tricker
d = {'k1':[{'nest_key':['this is deep',['hello']]}]}
```

In [51]:
```
# This was harder than I expected...
d['k1'][0]['nest_key'][1][0]
```

Out[51]:
```
'hello'
```

In [52]:
```
# This will be hard and annoying!
d = {'k1':[1,2,{'k2':['this is tricky',{'tough':[1,2,['hello']]}]}]}
```

In [61]:
```
# Phew
d['k1'][2]['k2'][1]['tough'][2][0]
```

Out[61]:
```
'hello'
```

Can you sort a dictionary? Why or why not?

Answer: No! Because normal dictionaries are *mappings* not a sequence.

Tuples

What is the major difference between tuples and lists?

Tuples are immutable!

How do you create a tuple?

In [63]:
```
t = (1,2,3)
```

Sets

What is unique about a set?

Answer: They don't allow for duplicate items!

Use a set to find the unique values of the list below:

In [64]:
```
l = [1,2,2,33,4,4,11,22,3,3,2]
```

In [65]:
```
set(l)
```

Out[65]:
```
{1, 2, 3, 4, 11, 22, 33}
```

Booleans

For the following quiz questions, we will get a preview 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.	
<>	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.

What will be the resulting Boolean of the following pieces of code (answer fist then check by typing it in!)

In [66]:
```
# Answer before running cell
2 > 3
```

Out[66]:
```
False
```

In [67]:
```
# Answer before running cell
3 <= 2
```

Out[67]:
```
False
```

In [68]:
```
# Answer before running cell
3 == 2.0
```

Out[68]:
```
False
```

In [69]:
```
# Answer before running cell
3.0 == 3
```

Out[69]:
```
True
```

In [70]:
```
# Answer before running cell
4**0.5 != 2
```

Out[70]:
```
False
```

Final Question: What is the boolean output of the cell block below?

In [71]:
```
# two nested lists
l_one = [1,2,[3,4]]
l_two = [1,2,{'k1':4}]

#True or False?
l_one[2][0] >= l_two[2]['k1']
```

Out[71]:
```
False
```

Great Job on your first assessment!

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Comparison Operators.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Comparison Operators.ipynb)

Comparison Operators

In this lecture we will be learning about Comparison Operators in Python. These operators will allow us to compare variables and output a Boolean value (True or False).

If you have any sort of background in Math, these operators should be very straight forward.

First we'll present a table of the comparison operators and then work through some examples:

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.

Let's now work through quick examples of each of these.

Equal

In [3]:
```
2 == 2
```
Out[3]:
```
True
```

In [4]:
```
1 == 0
```
Out[4]:
```
False
```

Not Equal

In [5]:
```
2 != 1
```
Out[5]:
```
True
```

In [6]:
```
2 != 2
```
Out[6]:
```
False
```

In [7]:
```
2 <> 1
```
Out[7]:
```
True
```

In [8]:
```
2 <> 2
```
Out[8]:
```
False
```

Greater Than

In [9]:
```
2 > 1
```

Out[9]: True

In [10]:
```
2 > 4
```

Out[10]: False

Less Than

In [11]:
```
2 < 4
```

Out[11]: True

In [12]:
```
2 < 1
```

Out[12]: False

Greater Than or Equal to

In [13]:
```
2 >= 2
```

Out[13]: True

In [14]:
```
2 >= 1
```

Out[14]: True

Less than or Equal to

In [15]:
```
2 <= 2
```

Out[15]: True

In [16]:
```
2 <= 4
```

Out[16]: True

Great! Go over each comparison operator to make sure you understand what each one is saying. But hopefully this was straightforward for you

Next we will cover chained comparison operators

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Chained Comparison Operators.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Chained Comparison Operators.ipynb)

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.

In this lecture we will learn how to chain comparison operators and we will also introduce two other important statements in python: **and** and **or**.

Let's look at a few examples of using chains:

In [1]:
```
1 < 2 < 3
```
Out[1]: True

The above statement check 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 [2]:
```
1<2 and 2<3
```
Out[2]: 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 [3]:
```
1 < 3 > 2
```
Out[3]: True

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

In [4]:
```
1<3 and 3>2
```
Out[4]: 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 [5]:
```
1==2 or 2<3
```
Out[5]: True

Note how it was true, this is because with the **or** operator, we only need one *or* the other two be true. Let's see one more example to drive this home:

In [6]:
```
1==1 or 100==1
```
Out[6]: True

Great! For an overview of this quick lesson: You should have a comfortable understanding of using **and** and **or** statements as well as reading chained comparison code.

Go ahead and go to the quiz for this section to check your understanding!

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Introduction to Python Statements.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Introduction to Python Statements.ipynb)

Introduction to Python Statements

In this lecture we will be doing a quick overview of Python Statements. This lecture will emphasize differences between Python and other languages such as C++.

There are two reasons we take this approach for learning the context of Python Statements:

```
1.) If you are coming from a different language this will rapidly accelerate your understanding of Python.
2.) Learning about statements will allow you to be able to read other languages more easily in the future.
```

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
```

You'll notice that Python is less cluttered and much more readable than the first version. How does Python manage this?

Let's walk through the main differences:

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.

Lastly, to end this brief overview of differences, let's take a closer look at indentation syntax in Python vs other languages:

Indentation

Here is some pseudo-code to indicate the use of whitespace and indentation in Python:

Other Languages

```
if (x)
    if(y)
        code-statement;
else
    another-code-statement;
```

Python

```
if x:
    if y:
        code-statement
else:
    another-code-statement
```

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

Now let's start diving deeper by coding these sort of statements in Python!

Time to code!

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ If, elif, and else Statements.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/If, elif, and else Statements.ipynb)

if,elif,else Statements

if Statements in Python allows us to tell the computer to perform alternative actions based on a certain set of results.

Verbally, we can imagine we are telling the computer:

"Hey if this case happens, perform some action"

We can then expand the idea further with elif and else statements, which allow us to tell the computer:

"Hey if this case happens, perform some action. Else if another case happens, perform some other action. Else-- none of the above cases happened, perform this action"

Let's go ahead and look at the syntax format for if statements to get a better idea of this:

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

First Example

Let's see a quick example of this:

In [1]:
```
if True:
    print 'It was true!'
```

```
It was true!
```

Let's add in some else logic:

In [5]:
```
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
```

Multiple Branches

Let's get a fuller picture of how far if, elif, and else can take us!

We write this out in a nested structure. Take note of how the if,elif,and else line up in the code. This can help you see what if is related to what elif or else statements.

We'll reintroduce a comparison syntax for Python.

In [6]:
```
loc = 'Bank'

if loc == 'Auto Shop':
    print 'Welcome to the Auto Shop!'
elif loc == 'Bank':
    print 'Welcome to the bank!'
else:
    print "Where are you?"
```

```
Welcome to the bank!
```

Note how the 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.

Let's create two more simple examples for the if,elif, and else statements:

In [1]:
```
person = 'Sammy'

if person == 'Sammy':
    print 'Welcome Sammy!'
else:
    print "Welcome, what's your name?"
```

```
Welcome Sammy!
```

In [2]:
```
person = 'George'

if person == 'Sammy':
    print 'Welcome Sammy!'
elif person =='George':
    print "Welcome George!"
else:
    print "Welcome, what's your name?"
```

```
Welcome George!
```

Indentation

It is important to keep a good understanding of how indentation works in Python to maintain the structure and order of your code. We will touch on this topic again when we start building out functions!

In [ ]:
```

```

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ For Loops.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/For Loops.ipynb)

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. Objects that we've learned about that we can iterate over include strings,lists,tuples, and even built in iterables for dictionaries, such as the keys or values.

We've already seen the **for** statement a little bit in past lectures but now lets formalize our understanding.

Here's the general format for a **for** loop in Python:

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

The variable name used for the item is completely up to the coder, so use your best judgment for choosing a name that makes sense and you will be able to understand when revisiting your code. This item name can then be referenced inside you loop, for example if you wanted to use if statements to perform checks.

Let's go ahead and work through several example of **for** loops using a variety of data object types. we'll start simple and build more complexity later on.

Example 1

Iterating through a list.

In [1]:
```
# We'll learn how to automate this sort of list in the next lecture
l = [1,2,3,4,5,6,7,8,9,10]
```

In [2]:
```
for num in l:
    print num
```

```
1
2
3
4
5
6
7
8
9
10
```

Great! Hopefully this makes sense. Now lets add a if statement to check for even numbers. We'll first introduce a new concept here--the modulo.

Modulo

The modulo allows us to get the remainder in a division and uses the % symbol. For example:

In [5]:
```
17 % 5
```

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

This makes sense since 17 divided by 5 is 3 remainder 2. Let's see a few more quick examples:

In [6]:
```
# 3 Remainder 1
10 % 3
```

Out[6]:
```
1
```

In [9]:
```
# 2 Remainder 4
18 % 7
```

Out[9]:
```
4
```

In [10]:
```
# 2 no remainder
4 % 2
```

Out[10]:
```
0
```

Notice that if a number is fully divisible with no remainder, the result of the modulo call is 0. We can use this to test for even numbers, since if a number modulo 2 is equal to 0, that means it is an even number!

Back to the **for** loops!

Example 2

Let's print only the even numbers from that list!

In [11]:
```
for num in l:
    if num % 2 == 0:
        print num
```

```
2
4
6
8
10
```

We could have also put in else statement in there:

In [12]:
```
for num in l:
    if num % 2 == 0:
        print num
    else:
        print 'Odd number'
```

```
Odd number
2
Odd number
4
Odd number
6
Odd number
8
Odd number
10
```

Example 3

Another common idea during a **for** loop is keeping some sort of running tally during the multiple loops. For example, lets create a for loop that sums up the list:

In [13]:
```
# Start sum at zero
list_sum = 0

for num in l:
    list_sum = list_sum + num

print list_sum
```

```
55
```

Great! Read over the above cell and make sure you understand fully what is going on. Also we could have implemented a += to to the addition towards the sum. For example:

In [14]:
```
# Start sum at zero
list_sum = 0

for num in l:
    list_sum += num

print list_sum
```

```
55
```

Example 4

We've used for loops with lists, how about with strings? Remember strings are a sequence so when we iterate through them we will be accessing each item in that string.

In [15]:
```
for letter in 'This is a string.':
    print letter
```

```
T
h
i
s

i
s

a

s
t
r
i
n
g
.
```

Example 5

Let's now look at how a for loop can be used with a tuple:

In [16]:
```
tup = (1,2,3,4,5)

for t in tup:
    print t
```

```
1
2
3
4
5
```

Example 6

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 [17]:
```
l = [(2,4),(6,8),(10,12)]
```

In [18]:
```
for tup in l:
    print tup
```

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

In [19]:
```
# Now with unpacking!
for (t1,t2) in l:
    print t1
```

```
2
6
10
```

Cool! With tuples in a sequence we can access the items inside of them through unpacking! The reason this is important is because many object will deliver their iterables through tuples. Let's start exploring iterating through Dictionaries to explore this further!

Example 7

In [1]:
```
d = {'k1':1,'k2':2,'k3':3}
```

In [2]:
```
for item in d:
    print item
```

```
k3
k2
k1
```

Notice how this produces only the keys. So how can we get the values? Or both the keys and the values?

Here is where we are going to have a Python 3 Alert!

Python 3 Alert!

Python 2: Use .iteritems() to iterate through

In Python 2 you should use .iteritems() to iterate through the keys and values of a dictionary. This basically creates a generator (we will get into generators later on in the course) that will generate the keys and values of your dictionary. Let's see it in action:

In [9]:
```
# Creates a generator
d.iteritems()
```

Out[9]:
```
<dictionary-itemiterator at 0x104365520>
```

Calling the items() method returns a list of tuples. Now we can iterate through them just as we did in the previous examples.

In [10]:
```
# Create a generator
for k,v in d.iteritems():
    print k
    print v
```

```
k3
3
k2
2
k1
1
```

Python 3: items()

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

In [11]:
```
# For Python 3
for k,v in d.items():
    print(k)
    print(v)
```

```
k3
3
k2
2
k1
1
```

You might be wondering why this worked in Python 2. This is because of the introduction of generators to Python during its earlier years. (We will go over generators and what they are in a future section, but the basic notion is that generators don't store data in memory, but instead just yield it to you as it goes through an iterable item).

Originally, Python items() built a real list of tuples and returned that. That could potentially take a lot of extra memory.

Then, generators were introduced to the language in general, and that method was reimplemented as an iterator-generator method named iteritems(). The original remains for backwards compatibility.

One of Python 3's changes is that items() now return iterators, and a list is never fully built. The iteritems() method is also gone, since items() now works like iteritems() in Python 2.

Conclusion

We've learned how to use for loops to iterate through tuples,lists,strings, and dictionaries. It will be an important tool for us, so make sure you know it well and understood the above examples.

More resources (http://www.tutorialspoint.com/python/python_for_loop.htm)

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ While loops.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/While loops.ipynb)

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. The reason it is called a 'loop' is because the code statements are looped through over and over again until the condition is no longer met.

The general format of a while loop is:

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

Let's look at a few simple while loops in action.

In [2]:
```
x = 0

while x < 10:
    print 'x is currently: ',x
    print ' x is still less than 10, adding 1 to x'
    x+=1
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
x is currently:  1
 x is still less than 10, adding 1 to x
x is currently:  2
 x is still less than 10, adding 1 to x
x is currently:  3
 x is still less than 10, adding 1 to x
x is currently:  4
 x is still less than 10, adding 1 to x
x is currently:  5
 x is still less than 10, adding 1 to x
x is currently:  6
 x is still less than 10, adding 1 to x
x is currently:  7
 x is still less than 10, adding 1 to x
x is currently:  8
 x is still less than 10, adding 1 to x
x is currently:  9
 x is still less than 10, adding 1 to x
```

Notice how many times the print statements occurred and how the while loop kept going until the True condition was met, which occurred once x==10. Its important to note that once this occurred the code stopped. Lets see how we could add an else statement:

In [3]:
```
x = 0

while x < 10:
    print 'x is currently: ',x
    print ' x is still less than 10, adding 1 to x'
    x+=1

else:
    print 'All Done!'
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
x is currently:  1
 x is still less than 10, adding 1 to x
x is currently:  2
 x is still less than 10, adding 1 to x
x is currently:  3
 x is still less than 10, adding 1 to x
x is currently:  4
 x is still less than 10, adding 1 to x
x is currently:  5
 x is still less than 10, adding 1 to x
x is currently:  6
 x is still less than 10, adding 1 to x
x is currently:  7
 x is still less than 10, adding 1 to x
x is currently:  8
 x is still less than 10, adding 1 to x
x is currently:  9
 x is still less than 10, adding 1 to x
All Done!
```

break, continue, pass

We can use break, continue, and pass statements in our loops to add additional functionality for various cases. The three statements are defined by:

```
break: Breaks out of the current closest enclosing loop.
continue: Goes to the top of the closest enclosing loop.
pass: Does nothing at all.
```

Thinking about **break** and **continue** statements, the general format of the while loop looks like this:

```
while test:
    code statement
    if test:
        break
    if test:
        continue
else:
```

break and **continue** statements can appear anywhere inside the loop's body,but we will usually put them further nested in conjunction with an **if** statement to perform an action based on some condition.

Lets go ahead and look at some examples!

In [6]:
```
x = 0

while x < 10:
    print 'x is currently: ',x
    print ' x is still less than 10, adding 1 to x'
    x+=1
    if x ==3:
        print 'x==3'
    else:
        print 'continuing...'
        continue
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
continuing...
x is currently:  1
 x is still less than 10, adding 1 to x
continuing...
x is currently:  2
 x is still less than 10, adding 1 to x
x==3
x is currently:  3
 x is still less than 10, adding 1 to x
continuing...
x is currently:  4
 x is still less than 10, adding 1 to x
continuing...
x is currently:  5
 x is still less than 10, adding 1 to x
continuing...
x is currently:  6
 x is still less than 10, adding 1 to x
continuing...
x is currently:  7
 x is still less than 10, adding 1 to x
continuing...
x is currently:  8
 x is still less than 10, adding 1 to x
continuing...
x is currently:  9
 x is still less than 10, adding 1 to x
continuing...
```

Note how we have a printed statement when x==3, and a continue being printed out as we continue through the outer while loop. Let's put in a break once x ==3 and see if the result makes sense:

In [7]:
```
x = 0

while x < 10:
    print 'x is currently: ',x
    print ' x is still less than 10, adding 1 to x'
    x+=1
    if x ==3:
        print 'Breaking because x==3'
        break
    else:
        print 'continuing...'
        continue
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
continuing...
x is currently:  1
 x is still less than 10, adding 1 to x
continuing...
x is currently:  2
 x is still less than 10, adding 1 to x
Breaking because x==3
```

Note how the other else statement wasn't reached and continuing was never printed!

After these brief but simple examples, you should feel comfortable using while statements in you code.

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

In [ ]:
```
# DO NOT RUN THIS CODE!!!!
while True:
    print 'Uh Oh infinite Loop!'
```

In [ ]:
```

```

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Range().ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Range().ipynb)

range()

In this short lecture we will be discussing the range function. We haven't developed a very deep level of knowledge of functions yet, but we can understand the basics of this simple (but extremely useful!) function.

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. Lets walk through a few examples:

In [7]:
```
range(0,10)
```

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

In [8]:
```
x =range(0,10)
type(x)
```

Out[8]:
```
list
```

In [3]:
```
start = 0 #Default
stop = 20
x = range(start,stop)
```

In [4]:
```
x
```

Out[4]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Great! Notice how it went *up to* 20, but doesn't actually produce 20. Just like in indexing. What about step size? We can specify that as a third argument:

In [5]:
```
x = range(start,stop,2)
#Show
x
```

Out[5]:
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Awesome! Well thats it...or is it?

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 solve with the use of a generator. For a simplified explanation: 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.

The good news is in Python 3, range() behaves as a generator and you don't need to worry about it. Let's see a quick example with xrange()

In [9]:
```
for num in range(10):
    print num
```

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

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

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

So the main takeaway here is for Python 2, if you are using range() in a way that you don't need to save the results in a list, use xrange() instead. For Python 3, use range() in any instance.

You should now have a good understanding of how to use range() in either version of Python.

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ List Comprehensions.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/List Comprehensions.ipynb)

Comprehensions

In addition to sequence operations and list methods, Python includes a more advanced operation called a list comprehension.

List comprehensions allow us to build out lists using a different notation. You can think of it as essentially a one line for loop built inside of brackets. For a simple example:

Example 1

In [1]:
```
# Grab every letter in string
lst = [x for x in 'word']
```

In [2]:
```
# Check
lst
```

Out[2]:
```
['w', 'o', 'r', 'd']
```

This is the basic idea of a list comprehension. If you're familiar with mathematical notation this format should feel familiar for example: x^2 : x in { 0,1,2...10}

Lets see a few more example of list comprehensions in Python:

Example 2

In [1]:
```
# Square numbers in range and turn into list
lst = [x**2 for x in range(0,11)]
```

In [2]:
```
lst
```

Out[2]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Example 3

Lets see how to add in if statements:

In [5]:
```
# Check for even numbers in a range
lst = [x for x in range(11) if x % 2 == 0]
```

In [6]:
```
lst
```

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

Example 4

Can also do more complicated arithmetic:

In [7]:
```
# Convert Celsius to Fahrenheit
celsius = [0,10,20.1,34.5]

fahrenheit = [ ((float(9)/5)*temp + 32) for temp in Celsius ]

fahrenheit
```

Out[7]:
```
[32.0, 50.0, 68.18, 94.1]
```

Example 5

We can also perform nested list comprehensions, for example:

In [8]:
```
lst = [ x**2 for x in [x**2 for x in range(11)]]
lst
```

Out[8]:
```
[0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10000]
```

Later on in the course we will learn about generator comprehensions. After this lecture you should feel comfortable reading and writing basic list comprehensions.

Statements Assessment Test

Lets test your knowledge!

Use for, split(), and if to create a Statement that will print out words that start with 's':

In [3]:
```
st = 'Print only the words that start with s in this sentence'
```

In [1]:
```
#Code here  l = [word for word in st.split() if word[0] == 's']
```

Use range() to print all the even numbers from 0 to 10.

In [2]:
```
#Code Here  [num for num in xrange(11) if num % 2 == 0]
```

Use List comprehension to create a list of all numbers between 1 and 50 that are divisible by 3.

In [3]:
```
#Code in this cell
[]       [num for num in range(1, 51) if num % 3 == 0]
```

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

Go through the string below and if the length of a word is even print "even!"

In [6]:
```
st = 'Print every word in this sentence that has an even number of letters'
```

In [4]:
```
#Code in this cell  print [word for word in st.split() if len(word) % 2 == 0]
```

Write a program that prints the integers from 1 to 100. But for multiples of three print "Fizz" instead of the number, and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

In [ ]:
```
#Code in this cell
for x in xrange(101):
    if x % 3 == 0 and x % 5 == 0:
        print x, 'FizzBuzz'
    elif x % 3 == 0:
        print x, 'Fizz'
    elif x % 5 == 0:
        print x, 'Buzz'
    else:
        print x
```

Use List Comprehension to create a list of the first letters of every word in the string below:

In [8]:
```
st = 'Create a list of the first letters of every word in this string'
```

In [5]:
```
#Code in this cell  [word[0] for word in st.split()]
```

Great Job!

Statements Assessment Solutions

Use for, split(), and if to create a Statement that will print out words that start with 's':

In [3]:
```
st = 'Print only the words that start with s in this sentence'
```

In [4]:
```
for word in st.split():
    if word[0] == 's':
        print word
```

```
start
s
sentence
```

Use range() to print all the even numbers from 0 to 10.

In [1]:
```
range(0,11,2)
```

Out[1]:
```
[0, 2, 4, 6, 8, 10]
```

Use List comprehension to create a list of all numbers between 1 and 50 that are divisible by 3.

In [1]:
```
[x for x in range(1,50) if x%3 == 0]
```

Out[1]:
```
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48]
```

Go through the string below and if the length of a word is even print "even!"

In [2]:
```
st = 'Print every word in this sentence that has an even number of letters'
```

In [4]:
```
for word in st.split():
    if len(word)%2 == 0:
        print word+" <-- has an even length!"
```

```
word <-- has an even length!
in <-- has an even length!
this <-- has an even length!
sentence <-- has an even length!
that <-- has an even length!
an <-- has an even length!
even <-- has an even length!
number <-- has an even length!
of <-- has an even length!
```

Write a program that prints the integers from 1 to 100. But for multiples of three print "Fizz" instead of the number, and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

In [ ]:
```
for num in xrange(1,101):
    if num % 5 == 0 and num % 3 == 0:
        print "FizzBuzz"
    elif num % 3 == 0:
        print "Fizz"
    elif num % 5 == 0:
        print "Buzz"
    else:
        print num
```

Use List Comprehension to create a list of the first letters of every word in the string below:

In [6]:
```
st = 'Create a list of the first letters of every word in this string'
```

In [7]:
```
[word[0] for word in st.split()]
```

Out[7]:
```
['C', 'a', 'l', 'o', 't', 'f', 'l', 'o', 'e', 'w', 'i', 't', 's']
```

Great Job!

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Methods.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Methods.ipynb)

Methods

We've already seen a few example of methods when learning about Object and Data Structure Types in Python. Methods are essentially functions built into objects. Later on in the course we will learn about how to create our own objects and methods using Object Oriented Programming (OOP) and classes.

Methods will perform specific actions on the object and can also take arguments, just like a function. This lecture will serve as just a brief introduction to methods and get you thinking about overall design methods that we will touch back upon when we reach OOP in the course.

Methods are in the form:

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

You'll later see that we can think of methods as having an argument 'self' referring to the object itself. You can't see this argument but we will be using it later on in the course during the OOP lectures.

Lets take a quick look at what an example of the various methods a list has:

In [2]:
```
# 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

Let's try out a few of them:

append() allows us to add elements to the end of a list:

In [3]:
```
l.append(6)
```

In [4]:
```
l
```

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

Great! Now how about count()? The count() method will count the number of occurrences of an element in a list.

In [7]:
```
# Check how many times 2 shows up in the list
l.count(2)
```

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

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 [17]:
```
help(l.count)
```

```
Help on built-in function count:

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

Feel free to play around with the rest of the methods for a list. Later on in this section your quiz will involve using help and Google searching for methods of different types of objects!

Great! By this lecture you should feel comfortable calling methods of objects in Python!

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Functions.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Functions.ipynb)

Functions

Introduction to Functions

This lecture will consist of explaining what a function is in Python and how to create one. Functions will be one of our main building blocks when we construct larger and larger amounts of code to solve problems.

So what is a function?

Formally, a function is a useful device 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. If you remember back to the lessons on strings and lists, remember that we used a function len() to get the length of a string. Since checking the length of a sequence is a common task you would want to write a function that can do this repeatedly at command.

Functions will be one of most basic levels of reusing code in Python, and it will also allow us to start thinking of program design (we will dive much deeper into the ideas of design when we learn about Object Oriented Programming).

def Statements

Let's see how to build out a function's syntax in Python. It has the following form:

In [3]:
```
def name_of_function(arg1,arg2):
    '''
    This is where the function's Document String (doc-string) goes
    '''
    # Do stuff here
    #return desired result
```

We begin with def then a space followed by the name of the function. Try to keep names relevant, for example len() is a good name for a length() function. Also be careful with names, you wouldn't want to call a function the same name as a built-in function in Python (https://docs.python.org/2/library/functions.html) (such as len).

Next come a pair of parenthesis with a number of arguments separated by a comma. These arguments are the inputs for your function. You'll be able to use these inputs in your function and reference them. After this you put a colon.

Now here is the important step, you must indent to begin the code inside your function correctly. Python makes use of *whitespace* to organize code. Lots of other programing languages do not do this, so keep that in mind.

Next you'll see the doc-string, this is where you write a basic description of the function. Using iPython and iPython Notebooks, you'll be ab;e to read these doc-strings by pressing Shift+Tab after a function name. Doc strings are not necessary for simple functions, but its good practice to put them in so you or other people can easily understand the code you write.

After all this you begin writing the code you wish to execute.

The best way to learn functions is by going through examples. So let's try to go through examples that relate back to the various objects and data structures we learned about before.

Example 1: A simple print 'hello' function

In [4]:
```
def say_hello():
    print 'hello'
```

Call the function

In [5]:
```
say_hello()
```

```
hello
```

Example 2: A simple greeting function

Let's write a function that greets people with their name.

In [6]:
```
def greeting(name):
    print 'Hello %s' %name
```

In [7]:
```
greeting('Jose')
```

```
Hello Jose
```

Using return

Let's see some example that use a return statement. return allows a function to *return* a result that can then be stored as a variable, or used in whatever manner a user wants.

Example 3: Addition function

In [8]:
```
def add_num(num1,num2):
    return num1+num2
```

In [9]:
```
add_num(4,5)
```

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

In [10]:
```
# Can also save as variable due to return
result = add_num(4,5)
```

In [11]:
```
print result
```

```
9
```

What happens if we input two strings?

In [12]:
```
print add_num('one','two')
```

```
onetwo
```

Note that because we don't declare variable types in Python, this function could be used to add numbers or sequences together! We'll later learn about adding in checks to make sure a user puts in the correct arguments into a function.

Lets also start using *break*,*continue*, and *pass* statements in our code. We introduced these during the while lecture.

Finally lets go over a full example of creating a function to check if a number is prime (a common interview exercise).

We know a number is prime if that number is only evenly divisible by 1 and itself. Let's write our first version of the function to check all the numbers from 1 to N and perform modulo checks.

In [8]:
```
def is_prime(num):
    '''
    Naive method of checking for primes.
    '''
    for n in range(2,num):
        if num % n == 0:
            print 'not prime'
            break
    else: # If never mod zero, then prime
        print 'prime'
```

In [9]:
```
is_prime(16)
```

```
not prime
```

Note how we break the code after the print statement! We can actually improve this by only checking to the square root of the target number, also we can disregard all even numbers after checking for 2. We'll also switch to returning a boolean value to get an example of using return statements:

In [15]:
```
import math

def is_prime(num):
    '''
    Better method of checking for primes.
    '''
    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 [16]:
```
is_prime(14)
```

Out[16]:
```
False
```

Great! You should now have a basic understanding of creating your own functions to save yourself from repeatedly writing code!

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Lambda expressions.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Lambda expressions.ipynb)

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 [3]:
```
def square(num):
    return num**2
```

In [4]:
```
square(2)
```

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

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

In [5]:
```
def square(num): return num**2
```

In [6]:
```
square(2)
```

Out[6]:
```
4
```

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

In [7]:
```
lambda num: num**2
```

Out[7]:
```
<function __main__.<lambda>>
```

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

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

In [9]:
```
square(2)
```

Out[9]:
```
4
```

And there you have it! The breakdown of a function into a lambda expression! Lets see a few more examples:

Example 1

Check it a number is even

In [13]:
```
even = lambda x: x%2==0
```

In [14]:
```
even(3)
```

Out[14]:
```
False
```

In [15]:
```
even(4)
```

Out[15]:
```
True
```

Example 2

Grab first character of a string:

In [22]:
```
first = lambda s: s[0]
```

In [23]:
```
first('hello')
```

Out[23]:
```
'h'
```

Example 3

Reverse a string:

In [24]:
```
rev = lambda s: s[::-1]
```

In [25]:
```
rev('hello')
```

Out[25]:
```
'olleh'
```

Example 4

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

In [17]:
```
adder = lambda x,y : x+y
```

In [19]:
```
adder(2,3)
```

Out[19]:
```
5
```

lambda expressions really shine when used in conjunction with map(),filter() and reduce(). Each of those functions has its own lecture, so feel free to explore them if you're very interested in lambda.

I highly recommend reading this blog post at Python Conquers the Universe (https://pythonconquerstheuniverse.wordpress.com/2011/08/29/lambda_tutorial/) for a great breakdown on lambda expressions and some explanations of common confusions!

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Nested Statements and Scope.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Nested Statements and Scope.ipynb)

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.

Lets start with a quick thought experiment, imagine the following code:

In [6]:
```
x = 25

def printer():
    x = 50
    return x

print x
print printer()
```

What do you imagine the output of printer() is? 25 or 50? What is the output of print x? 25 or 50?

In [3]:
```
print x
```

```
25
```

In [8]:
```
print printer()
```

```
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,...

Quick examples of LEGB

Local

In [15]:
```
# x is local here:
f = lambda x:x**2
```

Enclosing function locals

This occurs when we have a function inside a function (nested functions)

In [19]:
```
name = 'This is a global name'

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

    def hello():
        print 'Hello '+name

    hello()

greet()
```

```
Hello Sammy
```

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

Global

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

In [20]:
```
print name
```

```
This is a global name
```

Built-in

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

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

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

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.

Example:

In [1]:
```
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
```

The first time that we print the value of the name x with the first line in the function's body, Python uses the value of the parameter declared in the main block, above the function definition.

Next, we assign the value 2 to x. The name x is local to our function. So, when we change the value of x in the function, the x defined in the main block remains unaffected.

With the last print statement, we display the value of x as defined in the main block, thereby confirming that it is actually unaffected by the local assignment within the previously called function.

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.

Example:

In [12]:
```
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
```

The global statement is used to declare that x is a global variable - hence, when we assign a value to x inside the function, that change is reflected when we use the value of x in the main block.

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

Conclusion

You should now have a good understanding of Scope (you may have already intuitively felt right about Scope which is great!) One last mention is that you can use the globals() and locals() functions to check what are your current local and global variables.

Another thing to keep in mind is that everything in Python is an object! I can assign variables to functions just like I can with numbers! We will go over this again in the decorator section of the course!

Functions and Methods Homework ¶

Complete the following questions:

Write a function that computes the volume of a sphere given its radius.

In [25]:
```
def vol(rad):
    pass
```

Write a function that checks whether a number is in a given range (Inclusive of high and low)

In [7]:
```
def ran_check(num,low,high):
    pass
```

If you only wanted to return a boolean:

In [8]:
```
def ran_bool(num,low,high):
    pass
```

In [9]:
```
ran_bool(3,1,10)
```

Out[9]:
```
True
```

Write a Python function that accepts a string and calculate the number of upper case letters and lower case letters.

```
Sample String : 'Hello Mr. Rogers, how are you this fine Tuesday?'
Expected Output :
No. of Upper case characters : 4
No. of Lower case Characters : 33
```

If you feel ambitious, explore the Collections module to solve this problem!

In [11]:
```
def up_low(s):
    pass
```

Write a Python function that takes a list and returns a new list with unique elements of the first list.

```
Sample List : [1,1,1,1,2,2,3,3,3,3,4,5]
Unique List : [1, 2, 3, 4, 5]
```

In [13]:
```
def unique_list(l):
    pass
```

In [14]:
```
unique_list([1,1,1,1,2,2,3,3,3,3,4,5])
```

Out[14]:
```
[1, 2, 3, 4, 5]
```

Write a Python function to multiply all the numbers in a list.

```
Sample List : [1, 2, 3, -4]
Expected Output : -24
```

In [17]:
```
def multiply(numbers):
    pass
```

In [18]:
```
multiply([1,2,3,-4])
```

Out[18]:
```
-24
```

Write a Python function that checks whether a passed string is palindrome or not.

Note: A palindrome is word, phrase, or sequence that reads the same backward as forward, e.g., madam or nurses run.

In [19]:
```
def palindrome(s):
    pass
```

In [20]:
```
palindrome('helleh')
```

Out[20]:
```
True
```

Hard:

Write a Python function to check whether a string is pangram or not.

```
Note : Pangrams are words or sentences containing every letter of the alphabet at least once.
For example : "The quick brown fox jumps over the lazy dog"
```

Hint: Look at the string module

In [21]:
```
import string

def ispangram(str1, alphabet=string.ascii_lowercase):
    pass
```

In [22]:
```
ispangram("The quick brown fox jumps over the lazy dog")
```

Out[22]:
```
True
```

In [23]:
```
string.ascii_lowercase
```

Out[23]:
```
'abcdefghijklmnopqrstuvwxyz'
```

Great Job!

Functions and Methods Homework Solutions

Write a function that computes the volume of a sphere given its radius.

In [25]:
```
def vol(rad):
    return (4.0/3)*(3.14)*(rad**3)
```

Write a function that checks whether a number is in a given range (Inclusive of high and low)

In [7]:
```
def ran_check(num,low,high):
    #Check if num is between low and high (including low and high)
    if num in range(low,high+1):
        print " %s is in the range" %str(num)
    else :
        print "The number is outside the range."
```

If you only wanted to return a boolean:

In [8]:
```
def ran_bool(num,low,high):
    return num in range(low,high+1)
```

In [9]:
```
ran_bool(3,1,10)
```

Out[9]:
```
True
```

Write a Python function that accepts a string and calculate the number of upper case letters and lower case letters.

```
Sample String : 'Hello Mr. Rogers, how are you this fine Tuesday?'
Expected Output :
No. of Upper case characters : 4
No. of Lower case Characters : 33
```

If you feel ambitious, explore the Collections module to solve this problem!

In [11]:
```
def up_low(s):
    d={"upper":0, "lower":0}
    for c in s:
        if c.isupper():
            d["upper"]+=1
        elif c.islower():
            d["lower"]+=1
        else:
            pass
    print "Original String : ", s
    print "No. of Upper case characters : ", d["upper"]
    print "No. of Lower case Characters : ", d["lower"]
```

In [12]:
```
s = 'Hello Mr. Rogers, how are you this fine Tuesday?'
up_low(s)
```

```
Original String :  Hello Mr. Rogers, how are you this fine Tuesday?
No. of Upper case characters :  4
No. of Lower case Characters :  33
```

Write a Python function that takes a list and returns a new list with unique elements of the first list.

```
Sample List : [1,1,1,1,2,2,3,3,3,3,4,5]
Unique List : [1, 2, 3, 4, 5]
```

In [13]:
```
def unique_list(l):
    # Also possible to use list(set())
    x = []
    for a in l:
        if a not in x:
            x.append(a)
    return x
```

In [14]:
```
unique_list([1,1,1,1,2,2,3,3,3,3,4,5])
```

Out[14]:
```
[1, 2, 3, 4, 5]
```

Write a Python function to multiply all the numbers in a list.

```
Sample List : [1, 2, 3, -4]
Expected Output : -24
```

In [4]:
```
def multiply(numbers):
    total = 1
    for x in numbers:
        total *= x
    return total
```

In [7]:
```
multiply([1,2,3,-4])
```

Out[7]:
```
-24
```

Write a Python function that checks whether a passed string is palindrome or not.

Note: A palindrome is word, phrase, or sequence that reads the same backward as forward, e.g., madam or nurses run.

In [5]:
```
def palindrome(s):

    s = s.replace(' ','') # This replaces all spaces " " with no space ''. (Fixes issues with strings that have spaces)
    return s == s[::-1] # Check through slicing
```

In [7]:
```
palindrome('nurses run')
```

Out[7]:
```
True
```

In [8]:
```
palindrome('abcba')
```

Out[8]:
```
True
```

Hard:

Write a Python function to check whether a string is pangram or not.

```
Note : Pangrams are words or sentences containing every letter of the alphabet at least once.
For example : "The quick brown fox jumps over the lazy dog"
```

Hint: Look at the string module

In [21]:
```
import string

def ispangram(str1, alphabet=string.ascii_lowercase):
    alphaset = set(alphabet)
    return alphaset <= set(str1.lower())
```

In [22]:
```
ispangram("The quick brown fox jumps over the lazy dog")
```

Out[22]:
```
True
```

In [23]:
```
string.ascii_lowercase
```

Out[23]:
```
'abcdefghijklmnopqrstuvwxyz'
```

In [ ]:
```

```

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Object Oriented Programming.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Object Oriented Programming.ipynb)

Object Oriented Programming

Object Oriented Programming (OOP) tends to be one of the major obstacles for beginners when they are first starting to learn Python.

There are many,many tutorials and lessons covering OOP so feel free to Google search other lessons, and I have also put some links to other useful tutorials online at the bottom of this Notebook.

For this lesson 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

Lets start the lesson by remembering about the Basic Python Objects. For example:

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

Remember how we could call methods on a list?

In [3]:
```
l.count(2)
```

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

What we will basically be doing in this lecture is exploring how we could create an Object type like a list. We've already learned about how to create functions. So lets explore Objects in general:

Objects

In Python, *everything is an object*. Remember from previous lectures we can use type() to check the type of object something is:

In [4]:
```
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 [2]:
```
# 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.

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 [3]:
```
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 [11]:
```
sam.breed
```

Out[11]:
```
'Lab'
```

In [9]:
```
frank.breed
```

Out[9]:
```
'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 [4]:
```
class Dog(object):

    # Class Object Attribute
    species = 'mammal'

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

In [5]:
```
sam = Dog('Lab','Sam')
```

In [6]:
```
sam.name
```

Out[6]:
```
'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 [7]:
```
sam.species
```

Out[7]:
```
'mammal'
```

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 [8]:
```
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
```

Great! Notice how we used self. notation to reference attributes of the class within the method calls. Review how the code above works and try creating your own method

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 [9]:
```
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 [10]:
```
d = Dog()
```

```
Animal created
Dog created
```

In [25]:
```
d.whoAmI()
```

```
Dog
```

In [26]:
```
d.eat()
```

```
Eating
```

In [27]:
```
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.

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 [11]:
```
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, self.pages)

    def __len__(self):
        return self.pages

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

In [12]:
```
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.

Great! After this lecture you should have a basic understanding of how to create your own objects with class in Python. You will be utilizing this heavily in your next milestone project!

For more great resources on this topic, check out:

Jeff Knupp's Post (https://www.jeffknupp.com/blog/2014/06/18/improve-your-python-python-classes-and-object-oriented-programming/)

Mozilla's Post (https://developer.mozilla.org/en-US/Learn/Python/Quickly_Learn_Object_Oriented_Programming)

Tutorial's Point (http://www.tutorialspoint.com/python/python_classes_objects.htm)

Official Documentation (https://docs.python.org/2/tutorial/classes.html)

Object Oriented Programming

Homework Assignment

Problem 1

Fill in the Line class methods to accept coordinate as a pair of tuples and return the slope and distance of the line.

In [18]:
```
class Line(object):

    def __init__(self,coor1,coor2):
        pass

    def distance(self):
        pass

    def slope(self):
        pass
```

```
class Line(object):

    def __init__(self, coor1, coor2):
        self.coor1 = coor1
        self.coor2 = coor2

    def slope(self):
        x1, y1 = self.coor1
        x2, y2 = self.coor2
        return float(y2 - y1) / (x2 - x1)

    def distance(self):
        x1, y1 = self.coor1
        x2, y2 = self.coor2
        return (float(x2 - x1) ** 2 + float(y2 - y1) ** 2) ** 0.5
```

In [ ]:
```
# EXAMPLE OUTPUT

coordinate1 = (3,2)
coordinate2 = (8,10)

li = Line(coordinate1,coordinate2)
```

In [20]:
```
li.distance()
```

Out[20]:
```
9.433981132056603
```

In [21]:
```
li.slope()
```

Out[21]:
```
1.6
```

Problem 2

Fill in the class

In [31]:
```
class Cylinder(object):

    def __init__(self,height=1,radius=1):
        pass

    def volume(self):
        pass

    def surface_area(self):
        pass
```

```
class Cylinder(object):

    pi = 3.14

    def __init__(self, height = 1, radius = 1):
        self.height = height
        self.radius = radius

    def volume(self):
        return self.pi * (self.radius ** 2) * self.height

    def surface_area(self):
        return 2 * self.pi * (self.radius ** 2)
                + 2 * self.pi * self.radius * self.height
```

In [32]:
```
# EXAMPLE OUTPUT
c = Cylinder(2,3)
```

In [34]:
```
c.volume()
```

Out[34]:
```
56.52
```

In [35]:
```
c.surface_area()
```

Out[35]:
```
94.2
```

In [ ]:
```

```

Object Oriented Programming

Homework Assignment

Problem 1

Fill in the Line class methods to accept coordinate as a pair of tuples and return the slope and distance of the line.

In [18]:
```
class Line(object):

    def __init__(self,coor1,coor2):
        self.coor1 = coor1
        self.coor2 = coor2

    def distance(self):
        x1,y1 = self.coor1
        x2,y2 = self.coor2
        return ( (x2-x1)**2 + (y2-y1)**2 )**0.5

    def slope(self):
        x1,y1 = self.coor1
        x2,y2 = self.coor2
        return float((y2-y1))/(x2-x1)
```

In [19]:
```
coordinate1 = (3,2)
coordinate2 = (8,10)

li = Line(coordinate1,coordinate2)
```

In [20]:
```
li.distance()
```

Out[20]:
```
9.433981132056603
```

In [21]:
```
li.slope()
```

Out[21]:
```
1.6
```

Problem 2

Fill in the class

In [31]:
```
class Cylinder(object):

    def __init__(self,height=1,radius=1):
        self.height = height
        self.radius = radius

    def volume(self):
        return self.height * (3.14)* (self.radius)**2

    def surface_area(self):
        top = (3.14)* (self.radius)**2
        return 2*top + 2*3.14*self.radius*self.height
```

In [32]:
```
c = Cylinder(2,3)
```

In [34]:
```
c.volume()
```

Out[34]:
```
56.52
```

In [35]:
```
c.surface_area()
```

Out[35]:
```
94.2
```

In [ ]:

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Errors and Exceptions Handling.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Errors and Exceptions Handling.ipynb)

Errors and Exception Handling

In this lecture we will learn about Errors and Exception Handling in Python. You've definitely already encountered errors by this point in the course. For example:

In [1]:
```
print 'Hello
```

```
  File "<ipython-input-1-23e01f0d17c8>", 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 (https://docs.python.org/2/library/exceptions.html). now lets learn how to handle errors and exceptions in our own code.

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: To get a better understanding of all this lets check out an example: We will look at some code that opens and writes a file:

In [11]:
```
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 statement
   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 [14]:
```
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 statement
   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 [13]:
```
try:
    f = open('testfile','r')
    f.write('Test write this')
except:
    # This will check for any exception and then execute this print statement
   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.
```

For example:

In [16]:
```
try:
   f = open("testfile", "w")
   f.write("Test write statement")
finally:
   print "Always execute finally code blocks"
```

```
Always execute finally code blocks
```

We can use this in conjunction with except. Lets see a new example that will take into account a user putting in the wrong input:

In [33]:
```
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 [35]:
```
askint()
```

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

In [36]:
```
askint()
```

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

---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-36-6ee53d339e7e> in <module>()
----> 1 askint()

<ipython-input-33-728ec4c542c2> in askint()
      7         finally:
      8             print "Finally, I executed!"
----> 9         print val

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

Notice how we got an error when trying to print val (because it was never properly assigned) Lets remedy this by asking the user and checking to make sure the input type is an integer:

In [39]:
```
def askint():
        try:
            val = int(raw_input("Please enter an integer: "))
        except:
            print "Looks like you did not enter an integer!"
            val = int(raw_input("Try again-Please enter an integer: "))
        finally:
            print "Finally, I executed!"
        print val
```

In [40]:
```
askint()
```

```
Please enter an integer: f
Looks like you did not enter an integer!
Try again-Please enter an integer: f
Finally, I executed!

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-40-6ee53d339e7e> in <module>()
----> 1 askint()

<ipython-input-39-e540976abf48> in askint()
      4         except:
      5             print "Looks like you did not enter an integer!"
----> 6             val = int(raw_input("Try again-Please enter an integer: "))
      7         finally:
      8             print "Finally, I executed!"

ValueError: invalid literal for int() with base 10: 'f'
```

Hmmm...that only did one check. How can we continually keep checking? We can use a while loop!

In [41]:
```
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
```

Errors and Exceptions Homework -

Problem 1

Handle the exception thrown by the code below by using try and except blocks.

In [1]:
```
for i in ['a','b','c']:
    print i**2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-908335551eea> in <module>()
      1 for i in ['a','b','c']:
----> 2     print i**2

TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

```
try:
    for i in ['a', 'b', 'c']:
        print i ** 2
except TypeError:
    print "This is a type error!"
else:
    print "Cannot be executed..."
```

Problem 2

Handle the exception thrown by the code below by using **try** and **except** blocks. Then use a **finally** block to print 'All Done.'

In [2]:
```
x = 5
y = 0

z = x/y
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-2-3effb78be709> in <module>()
      2 y = 0
      3
----> 4 z = x/y

ZeroDivisionError: integer division or modulo by zero
```

```
try:
    x = 5
    y = 0
    z = x / y
except ZeroDivisionError:
    print "This is zero division error!"
else:
    print "Cannot be executed anyway..."
finally:
    print "All Done."
```

Problem 3

Write a function that asks for an integer and prints the square of it. Use a while loop with a try,except, else block to account for incorrect inputs.

In [24]:
```
def ask():
    pass()
```

In [25]:
```
ask()
```

```
Input an integer: null
An error occurred! Please try again!
Input an integer: 2
Thank you, you number squared is:  4
```

```
def ask():
    while True:
        try:
            val = int(raw_input("Please enter an integer: "))
        except:
            print "looks like you did not enter an integer!"
            continue
        else:
            print "Yes. That's an integer!"
            print "Square of your input is:", val ** 2
            break
        finally:
            print "Finally executed!"
```

Great Job!

Errors and Exceptions Homework - Solution ¶

Problem 1

Handle the exception thrown by the code below by using try and except blocks.

In [1]:
```
try:
    for i in ['a','b','c']:
        print i**2
except:
    print "An error occurred!"
```

```
An error occurred!
```

Problem 2

Handle the exception thrown by the code below by using **try** and **except** blocks. Then use a **finally** block to print 'All Done.'

In [4]:
```
x = 5
y = 0
try:
    z = x/y
except ZeroDivisionError:
    print "Can't divide by Zero!"
finally:
    print 'All Done!'
```

```
Can't divide by Zero!
All Done!
```

Problem 3

Write a function that asks for an integer and prints the square of it. Use a while loop with a try,except, else block to account for incorrect inputs.

In [24]:
```
def ask():

    while True:
        try:
            n = input('Input an integer: ')
        except:
            print 'An error occurred! Please try again!'
            continue
        else:
            break


    print 'Thank you, you number squared is: ',n**2
```

In [25]:
```
ask()
```

```
Input an integer: null
An error occurred! Please try again!
Input an integer: 2
Thank you, you number squared is:  4
```

Great Job!

6. Built-in Exceptions

Exceptions should be class objects. The exceptions are defined in the module `exceptions`. This module never needs to be imported explicitly: the exceptions are provided in the built-in namespace as well as the `exceptions` module.

For class exceptions, in a `try` statement with an `except` clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which *it* is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

The built-in exceptions listed below can be generated by the interpreter or built-in functions. Except where mentioned, they have an "associated value" indicating the detailed cause of the error. This may be a string or a tuple containing several items of information (e.g., an error code and a string explaining the code). The associated value is the second argument to the `raise` statement. If the exception class is derived from the standard root class `BaseException`, the associated value is present as the exception instance's `args` attribute.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition "just like" the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The built-in exception classes can be subclassed to define new exceptions; programmers are encouraged to derive new exceptions from the `Exception` class or one of its subclasses, and not from `BaseException`. More information on defining exceptions is available in the Python Tutorial under User-defined Exceptions.

The following exceptions are only used as base classes for other exceptions.

exception **BaseException**

The base class for all built-in exceptions. It is not meant to be directly inherited by user-defined classes (for that, use `Exception`). If `str()` or `unicode()` is called on an instance of this class, the representation of the argument(s) to the instance are returned, or the empty string when there were no arguments.

New in version 2.5.

args

The tuple of arguments given to the exception constructor. Some built-in exceptions (like `IOError`) expect a certain number of arguments and assign a special meaning to the elements of this tuple, while others are usually called only with a single string giving an error message.

exception **Exception**

All built-in, non-system-exiting exceptions are derived from this class. All user-defined exceptions should also be derived from this class.

Changed in version 2.5: Changed to inherit from `BaseException`.

exception **StandardError**
The base class for all built-in exceptions except `StopIteration`, `GeneratorExit`, `KeyboardInterrupt` and `SystemExit`. `StandardError` itself is derived from `Exception`.

exception **ArithmeticError**
The base class for those built-in exceptions that are raised for various arithmetic errors: `OverflowError`, `ZeroDivisionError`, `FloatingPointError`.

exception **BufferError**
Raised when a buffer related operation cannot be performed.

exception **LookupError**
The base class for the exceptions that are raised when a key or index used on a mapping or sequence is invalid: `IndexError`, `KeyError`. This can be raised directly by `codecs.lookup()`.

exception **EnvironmentError**
The base class for exceptions that can occur outside the Python system: `IOError`, `OSError`. When exceptions of this type are created with a 2-tuple, the first item is available on the instance's `errno` attribute (it is assumed to be an error number), and the second item is available on the `strerror` attribute (it is usually the associated error message). The tuple itself is also available on the `args` attribute.

New in version 1.5.2.

When an `EnvironmentError` exception is instantiated with a 3-tuple, the first two items are available as above, while the third item is available on the `filename` attribute. However, for backwards compatibility, the `args` attribute contains only a 2-tuple of the first two constructor arguments.

The `filename` attribute is `None` when this exception is created with other than 3 arguments. The `errno` and `strerror` attributes are also `None` when the instance was created with other than 2 or 3 arguments. In this last case, `args` contains the verbatim constructor arguments as a tuple.

The following exceptions are the exceptions that are actually raised.

exception **AssertionError**
Raised when an `assert` statement fails.

exception **AttributeError**
Raised when an attribute reference (see Attribute references) or assignment fails. (When an object does not support attribute references or attribute assignments at all, `TypeError` is raised.)

exception **EOFError**
Raised when one of the built-in functions (`input()` or `raw_input()`) hits an end-of-file condition (EOF) without reading any data. (N.B.: the `file.read()` and `file.readline()` methods return an empty string when they hit EOF.)

exception **FloatingPointError**

Raised when a floating point operation fails. This exception is always defined, but can only be raised when Python is configured with the `--with-fpectl` option, or the `WANT_SIGFPE_HANDLER` symbol is defined in the `pyconfig.h` file.

exception **GeneratorExit**
Raised when a generator's `close()` method is called. It directly inherits from `BaseException` instead of `StandardError` since it is technically not an error.

New in version 2.5.

Changed in version 2.6: Changed to inherit from `BaseException`.

exception **IOError**
Raised when an I/O operation (such as a `print` statement, the built-in `open()` function or a method of a file object) fails for an I/O-related reason, e.g., "file not found" or "disk full".

This class is derived from `EnvironmentError`. See the discussion above for more information on exception instance attributes.

Changed in version 2.6: Changed `socket.error` to use this as a base class.

exception **ImportError**
Raised when an `import` statement fails to find the module definition or when a `from ... import` fails to find a name that is to be imported.

exception **IndexError**
Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not a plain integer, `TypeError` is raised.)

exception **KeyError**
Raised when a mapping (dictionary) key is not found in the set of existing keys.

exception **KeyboardInterrupt**
Raised when the user hits the interrupt key (normally `Control-C` or `Delete`). During execution, a check for interrupts is made regularly. Interrupts typed when a built-in function `input()` or `raw_input()` is waiting for input also raise this exception. The exception inherits from `BaseException` so as to not be accidentally caught by code that catches `Exception` and thus prevent the interpreter from exiting.

Changed in version 2.5: Changed to inherit from `BaseException`.

exception **MemoryError**
Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated value is a string indicating what kind of (internal) operation ran out of memory. Note that because of the underlying memory management architecture (C's `malloc()` function), the interpreter may not always be able to completely recover from this situation; it nevertheless raises an exception so that a stack traceback can be printed, in case a run-away program was the cause.

exception **NameError**

Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.

exception **NotImplementedError**
This exception is derived from `RuntimeError`. In user defined base classes, abstract methods should raise this exception when they require derived classes to override the method.

New in version 1.5.2.

exception **OSError**
This exception is derived from `EnvironmentError`. It is raised when a function returns a system-related error (not for illegal argument types or other incidental errors). The `errno` attribute is a numeric error code from `errno`, and the `strerror` attribute is the corresponding string, as would be printed by the C function `perror()`. See the module `errno`, which contains names for the error codes defined by the underlying operating system.

For exceptions that involve a file system path (such as `chdir()` or `unlink()`), the exception instance will contain a third attribute, `filename`, which is the file name passed to the function.

New in version 1.5.2.

exception **OverflowError**
Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for long integers (which would rather raise `MemoryError` than give up) and for most operations with plain integers, which return a long integer instead. Because of the lack of standardization of floating point exception handling in C, most floating point operations also aren't checked.

exception **ReferenceError**
This exception is raised when a weak reference proxy, created by the `weakref.proxy()` function, is used to access an attribute of the referent after it has been garbage collected. For more information on weak references, see the `weakref` module.

New in version 2.2: Previously known as the `weakref.ReferenceError` exception.

exception **RuntimeError**
Raised when an error is detected that doesn't fall in any of the other categories. The associated value is a string indicating what precisely went wrong.

exception **StopIteration**
Raised by an iterator's `next()` method to signal that there are no further values. This is derived from `Exception` rather than `StandardError`, since this is not considered an error in its normal application.

New in version 2.2.

exception **SyntaxError**
Raised when the parser encounters a syntax error. This may occur in an `import` statement, in an `exec` statement, in a call to the built-in function `eval()` or `input()`, or when reading the initial script or standard input (also interactively).

Instances of this class have attributes **filename**, **lineno**, **offset** and **text** for easier access to the details. `str()` of the exception instance returns only the message.

exception **IndentationError**
Base class for syntax errors related to incorrect indentation. This is a subclass of `SyntaxError`.

exception **TabError**
Raised when indentation contains an inconsistent use of tabs and spaces. This is a subclass of `IndentationError`.

exception **SystemError**
Raised when the interpreter finds an internal error, but the situation does not look so serious to cause it to abandon all hope. The associated value is a string indicating what went wrong (in low-level terms).

You should report this to the author or maintainer of your Python interpreter. Be sure to report the version of the Python interpreter (`sys.version`; it is also printed at the start of an interactive Python session), the exact error message (the exception's associated value) and if possible the source of the program that triggered the error.

exception **SystemExit**
This exception is raised by the `sys.exit()` function. When it is not handled, the Python interpreter exits; no stack traceback is printed. If the associated value is a plain integer, it specifies the system exit status (passed to C's **exit()** function); if it is `None`, the exit status is zero; if it has another type (such as a string), the object's value is printed and the exit status is one.

Instances have an attribute **code** which is set to the proposed exit status or error message (defaulting to `None`). Also, this exception derives directly from `BaseException` and not `StandardError`, since it is not technically an error.

A call to `sys.exit()` is translated into an exception so that clean-up handlers (`finally` clauses of `try` statements) can be executed, and so that a debugger can execute a script without running the risk of losing control. The `os._exit()` function can be used if it is absolutely positively necessary to exit immediately (for example, in the child process after a call to `os.fork()`).

The exception inherits from `BaseException` instead of `StandardError` or `Exception` so that it is not accidentally caught by code that catches `Exception`. This allows the exception to properly propagate up and cause the interpreter to exit.

Changed in version 2.5: Changed to inherit from `BaseException`.

exception **TypeError**
Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

exception **UnboundLocalError**
Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. This is a subclass of `NameError`.

New in version 2.0.

exception **UnicodeError**

Raised when a Unicode-related encoding or decoding error occurs. It is a subclass of `ValueError`.

`UnicodeError` has attributes that describe the encoding or decoding error. For example, `err.object[err.start:err.end]` gives the particular invalid input that the codec failed on.

encoding

The name of the encoding that raised the error.

reason

A string describing the specific codec error.

object

The object the codec was attempting to encode or decode.

start

The first index of invalid data in `object`.

end

The index after the last invalid data in `object`.

New in version 2.0.

exception **UnicodeEncodeError**

Raised when a Unicode-related error occurs during encoding. It is a subclass of `UnicodeError`.

New in version 2.3.

exception **UnicodeDecodeError**

Raised when a Unicode-related error occurs during decoding. It is a subclass of `UnicodeError`.

New in version 2.3.

exception **UnicodeTranslateError**

Raised when a Unicode-related error occurs during translating. It is a subclass of `UnicodeError`.

New in version 2.3.

exception **ValueError**

Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as `IndexError`.

exception **VMSError**

Only available on VMS. Raised when a VMS-specific error occurs.

exception **WindowsError**

Raised when a Windows-specific error occurs or when the error number does not correspond to an `errno` value. The `winerror` and `strerror` values are created from the return values of the `GetLastError()` and `FormatMessage()` functions from the Windows Platform API. The `errno` value maps the `winerror` value to corresponding `errno.h` values. This is a subclass of `OSError`.

New in version 2.0.

Changed in version 2.5: Previous versions put the `GetLastError()` codes into `errno`.

exception **ZeroDivisionError**
Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

The following exceptions are used as warning categories; see the `warnings` module for more information.

exception **Warning**
Base class for warning categories.

exception **UserWarning**
Base class for warnings generated by user code.

exception **DeprecationWarning**
Base class for warnings about deprecated features.

exception **PendingDeprecationWarning**
Base class for warnings about features which will be deprecated in the future.

exception **SyntaxWarning**
Base class for warnings about dubious syntax.

exception **RuntimeWarning**
Base class for warnings about dubious runtime behavior.

exception **FutureWarning**
Base class for warnings about constructs that will change semantically in the future.

exception **ImportWarning**
Base class for warnings about probable mistakes in module imports.

New in version 2.5.

exception **UnicodeWarning**
Base class for warnings related to Unicode.

New in version 2.5.

6.1. Exception hierarchy

The class hierarchy for built-in exceptions is:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
```

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Modules and Packages.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Modules and Packages.ipynb)

Modules and Packages

There's no code here because it didn't really make sense for the section. Check out the video lectures for more info and the resources for this.

Here is the best source the official docs! https://docs.python.org/2/tutorial/modules.html#packages (https://docs.python.org/2/tutorial/modules.html#packages)

But I really like the info here: https://python4astronomers.github.io/installation/packages.html (https://python4astronomers.github.io/installation/packages.html)

Here's some extra info to help:

Modules in Python are simply Python files with the .py extension, which implement a set of functions. Modules are imported from other modules using the import command.

To import a module, we use the import command. Check out the full list of built-in modules in the Python standard library here.

The first time a module is loaded into a running Python script, it is initialized by executing the code in the module once. If another module in your code imports the same module again, it will not be loaded twice but once only - so local variables inside the module act as a "singleton" - they are initialized only once.

If we want to import module math, we simply import the module:

In [5]:
```
# import the library
import math
```

In [6]:
```
# use it (ceiling rounding)
math.ceil(2.4)
```

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

Exploring built-in modules

Two very important functions come in handy when exploring modules in Python - the dir and help functions.

We can look for which functions are implemented in each module by using the dir function:

In [8]:
```
print(dir(math))
```

```
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
```

When we find the function in the module we want to use, we can read about it more using the help function, inside the Python interpreter:

In [10]:
```
help(math.ceil)
```

```
Help on built-in function ceil in module math:

ceil(...)
    ceil(x)

    Return the ceiling of x as an int.
    This is the smallest integral value >= x.
```

Writing modules

Writing Python modules is very simple. To create a module of your own, simply create a new .py file with the module name, and then import it using the Python file name (without the .py extension) using the import command.

Writing packages

Packages are name-spaces which contain multiple packages and modules themselves. They are simply directories, but with a twist.

Each package in Python is a directory which MUST contain a special file called **__init__.py**. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.

If we create a directory called foo, which marks the package name, we can then create a module inside that package called bar. We also must not forget to add the **__init__.py** file inside the foo directory.

To use the module bar, we can import it in two ways:

In [ ]:
```
# Just an example, this won't work
import foo.bar
```

In [ ]:
```
# OR could do it this way
from foo import bar
```

In the first method, we must use the foo prefix whenever we access the module bar. In the second method, we don't, because we import the module to our module's name-space.

The **__init__.py** file can also decide which modules the package exports as the API, while keeping other modules internal, by overriding the **__all__** variable, like so:

In [ ]:
```
__init__.py:

__all__ = ["bar"]
```

python4astronomers

One of the key features of Python is that the actual core language is fairly small. This is an intentional design feature to maintain simplicity. Much of the powerful functionality comes through external modules and packages.

The main work of installation so far has been to supplement the core Python with useful modules for science analysis.

Module

A module is simply a file containing Python definitions, functions, and statements. Putting code into modules is useful because of the ability to import the module functionality into your script or IPython session, for instance:

```
import astropy
import astropy.table
data = astropy.table.Table.read('my_table.fits')
```

You'll see `import` in virtually every Python script and soon it will be second nature.

Question:

Importing modules and putting the module name in front is such a bother, why do I need to do this?

Answer:

It keeps everything modular and separate. For instance many modules have a `read()` function since this is a common thing to do. Without using the `<module>.<function>(...)` syntax there would be no way to know which one to call.

Tip

Sometimes it is convenient to make an end-run around the `<module>.` prefixing. For instance when you run `ipython --pylab` the interpreter does some startup processing so that a number of functions from the numpy and matplotlib modules are available *without* using the prefix.

Python allows this with this syntax:

```
from <module> import *
```

That means to import every function and definition from the module into the current namespace (in other words make them available without prefixing). For instance you could do:

```
from astropy.table import *
data = Table('my_table.fits')
```

A general rule of thumb is that `from <module> import *` is OK for interactive analysis within IPython but you should avoid using it within scripts.

Package

A package is just a way of collecting related modules together within a single tree-like hierarchy. Very complex packages like NumPy or SciPy have hundreds of individual modules so putting them into a directory-like structure keeps things organized and avoids name collisions. For example here is a partial list of sub-packages available within SciPy

scipy.fftpack	Discrete Fourier Transform algorithms
scipy.stats	Statistical Functions
scipy.lib	Python wrappers to external libraries
scipy.lib.blas	Wrappers to BLAS library

scipy.lib.lapack	Wrappers to LAPACK library
scipy.integrate	Integration routines
scipy.linalg	Linear algebra routines
scipy.sparse.linalg	Sparse Linear Algebra
scipy.sparse.linalg.eigen	Sparse Eigenvalue Solvers
scipy.sparse.linalg.eigen.arpack	Eigenvalue solver using iterative methods.

Exercise: Import a package module and learn about it

Import the Linear algebra module from the SciPy package and find out what functions it provides.

Finding and installing other packages

If you've gotten this far you have a working scientific Python environment that has *most* of what you will ever need. Nevertheless it is almost certain that you will eventually find a need that is not met within your current installation. Here we learn **where** to find other useful packages and **how** to install them.

Package resources

Google

Google "python blah blah" or "python astronomy blah blah"

Resource lists

There are a number of sites specifically devoted to Python for astronomy with organized lists of useful resources and packages.

- Astropython.org resources
- Comfort at 1 AU
- Astronomical Python

Good vs. bad resources

When you find some package on the web, look for a few things:

- Good modern-looking documentation with examples
- Installs easily without lots of dependencies (or has detailed installation instructions)
- Actively developed

PyPI

The Python Package Index is the main repository for 3rd party Python packages (about 14000 packages and growing). An increasing number of astronomy related packages are available on PyPI, but this list misses a lot of available options.

The advantage of being on PyPI is the ease of installation using `pip install <package_name>`.

Exercise: Find packages for coordinate manipulations

Find one or more Python packages that will transform coordinates from Galactic to FK5 ecliptic.

Hint: tags are helpful at astropython.org and don't forget the "next" button at the bottom.

Package installation

There are two standard methods for installing a package.

pip install

The `pip install` script is available within our scientific Python installation and is very easy to use (when it works). During the installation process you already saw many examples of `pip install` in action. Features include:

- If supplied with a package name then it will query the PyPI site to find out about that package. Assuming the package is there then `pip install` will automatically download and install the package.

- Will accept a local tar file (assuming it contains an installable Python package) or a URL pointing to a tar file.
- Can install in the user package area via `pip install <package or URL> --user` (but see discussion further down)

python setup.py install

Some packages may fail to install via `pip install`. Most often there will be some obvious (or not) error message about compilation or missing dependency. In this case the likely next step is to download the installation tar file and untar it. Go into the package directory and look for files like:

```
INSTALL
README
setup.py
setup.cfg
```

If there is an INSTALL or README file then hopefully you will find useful installation instructions. Most well-behaved python packages do the installation via a standard `setup.py` script. This is used as follows:

```
python setup.py --help  # get options
python setup.py install # install in the python area (root / admin req'd)
python setup.py install --user # install to user's package area
```

More information is available in the Installing Python Modules page.

Where do packages get installed?

An important option in the installation process is where to put the package files. We've seen the `--user` option in `pip install` and `python setup.py install`. What's up with that? In the section below we document how this works. See the discussion in Multiple Pythons on your computer for a reason you might want to do this, but first please read this warning:

Warning

We strongly recommend against installing packages with `--user` unless you are an expert and really understand what you are doing, . This is because the local user version will always take precedence and can thus potentially disrupt other Python installations and cause hard-to-understand problems. Big analysis packages like CIAO, STSci_Python or CASA are carefully tested assuming the integrated environment they provide. If you start mucking this up then all bets are off.

WITH `--user`

Packages get installed in a local user-owned directory when you do something like either of the following:

```
pip install --user aplpy
python setup.py install --user
```

This puts the packages into:

Mac	~/Library/Python/2.x/lib/python/site-packages
Linux	~/.local/lib/python-2.x/site-packages
Windows	%APPDATA%/Python/Python2x/site-packages

Note

On Mac if you did not use Anaconda or the EPD Python Framework then you may see user packages within `~/.local/lib` as for linux. This depends on whether Python is installed as a MacOS Framework or not.

WITHOUT `--user`

If you use Anaconda or a non-root Python installation then there is no issue with permissions on any platform since the entire installation is local to a directory you own.

However, installing to a system-wide Python installation will require root or admin privilege. Installing this way has the benefit of making the package available for all users of the Python installation, but has the downside that it is more difficult to back out changes if required. In general we recommend using *only* the system package manager (e.g. `yum`) to install packages to the system Python. This will ensure integrity of your system Python, which is important even if you are the only user.

How do I find a package once installed?

Finding the file associated with a package or module is simple, just use the `help` command in IPython:

```
import scipy
help scipy
```

This gives something like:

```
NAME
    scipy

FILE
    /usr/local/lib/python2.6/site-packages/scipy/__init__.py

DESCRIPTION
    SciPy: A scientific computing package for Python
    ================================================

    Documentation is available in the docstrings and
    online at http://docs.scipy.org.
    ...
```

Uninstalling packages

There is no simple and fully consistent way to do this unless you use solutions like Anaconda or Canopy. The Python community is working on this one. In most simple cases, however, you can just delete the module file or directory that is revealed by the technique shown above.

Getting help on package installation

If you attempt to install a package but it does not work, your basic options are:

- Dig in your heels and start reading the error messages to see why it is unhappy. Often when you find a specific message it's time to start googling by pasting in the relevant parts of the message.
- Send an email to the AstroPy mailing list astropy@scipy.org. Include:
    - Package you are trying to install
    - URL for downloading the package tar file
    - Your platform (machine architecture and exact OS version)
    - Exactly what you typed
    - Entire output from the `python setup.py install` process

Do NOT just write and say “I tried to install BLAH and it failed, can someone help?”

Where does Python look for modules?

The official reference on Modifying Python’s Search Path gives all the details. In summary:

When the Python interpreter executes an import statement, it looks for modules on a search path. A default value for the path is configured into the Python binary when the interpreter is built. You can determine the path by importing the sys module and printing the value of `sys.path`:

```
$ python
Python 2.2 (#11, Oct  3 2002, 13:31:27)
[GCC 2.96 20000731 (Red Hat Linux 7.3 2.96-112)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', '/usr/local/lib/python2.3', '/usr/local/lib/python2.3/plat-linux2',
 '/usr/local/lib/python2.3/lib-tk', '/usr/local/lib/python2.3/lib-dynload',
 '/usr/local/lib/python2.3/site-packages']
>>>
```

Within a script it is possible to adjust the search path by modify `sys.path` which is just a Python list. Generally speaking you will want to put your path at the front of the list using insert:

```
import sys
sys.path.insert(0, '/my/path/python/packages')
```

You can also add paths to the search path using the PYTHONPATH environment variable.

Multiple Pythons on your computer

This is a practical problem that you are likely to encounter. Straight away you probably have the system Python (/usr/bin) and the Anaconda Python. Then if you install PyRaf, CIAO, and CASA you will get one more Python installation for each analysis package (there are good reasons for this). **In general, different Python installations cannot reliably share packages or resources**. Each installation should be considered as its own local Python universe.

Installing within each Python

Now that you know about all the great packages within our Scientific Python installation, you might want to start using them in your PyRAF or CASA or CIAO analysis.

If you start digging into Python you will likely come across the technique of setting the `PYTHONPATH` environment variable to extend the list of search paths that Python uses to look for a module. Let’s say you are using CIAO Python and want to use SciPy functions. You might be tempted to set `PYTHONPATH` to point to the directory in EPD where the SciPy modules live. This will fail because the EPD Python modules were compiled and linked assuming they’ll be run with EPD Python. With effort you might find a way to make this work, but in general it’s not a workable solution.

What *will* often work is to follow the package installation procedure for each desired package within each Python installation. This assumes that you have write permission into the directories where the analysis package files live. Simply enter the appropriate analysis environment, then do then following:

- At the command line do `which python` to verify that `python` is the correct one from the analysis environment.
- Navigate to http://pypi.python.org/pypi/pip#downloads
- Download the latest version of pip (*pip-X.Y.tar.gz*)

- Untar that file, go in the tar directory, and do `python setup.py install`
- Do `rehash` (for csh) then `which pip` to make sure the new `pip` got installed into your analysis environment path.
- Now you can do `pip install <package>` or `python setup.py install` for each desired package within that analysis environment.

It's worth noting that the original example of SciPy will not install with `pip`. It requires a very tricky installation from source, so unless SciPy ships with your favorite analysis environment you are out of luck with that one.

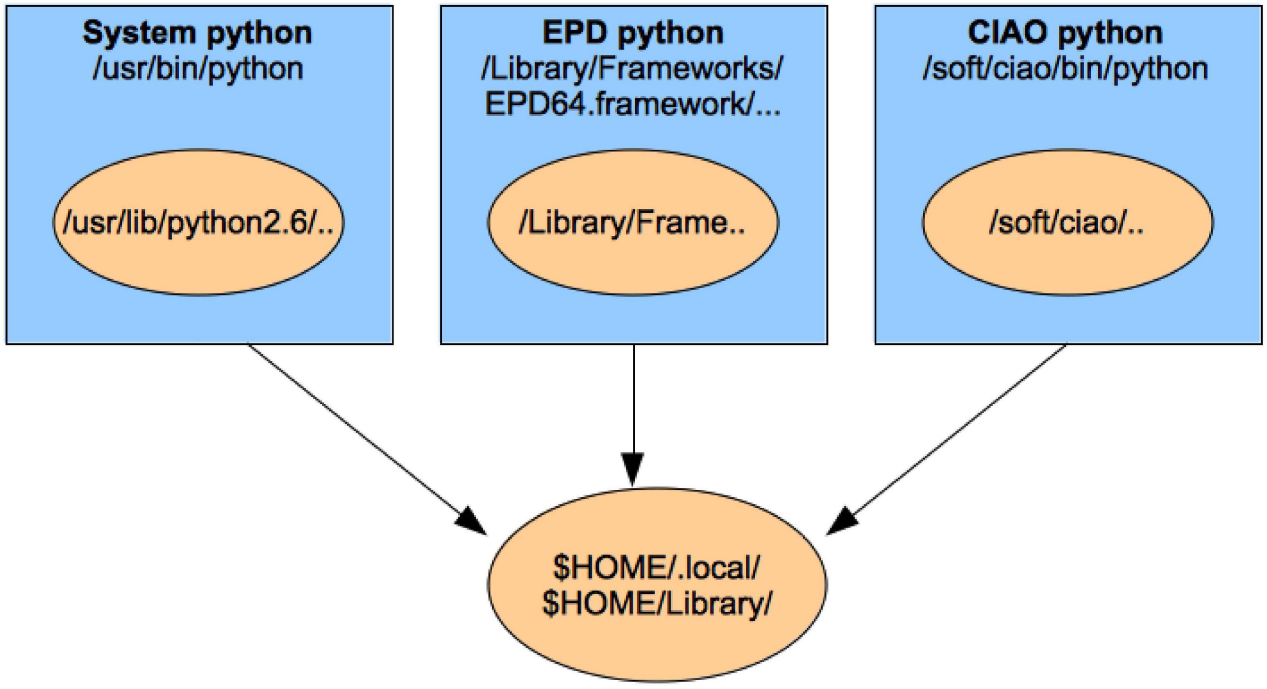
If you do *not* have write access to the analysis package directories, then you need to use the `--prefix` option in `pip` to install in a local area and then set a corresponding `PYTHONPATH`.

Can we share packages?

In some cases you can successfully share between Pythons. Although we don't recommend doing this it is nevertheless useful to illustrate how this works.

> **Warning**
>
> This technique is prone to breaking things in strange ways and we do not recommend it.

The first rule is that they need to be the same major version, i.e. all 2.6 or 2.7. This is because Python always includes a major version like `python2.6/` in the default search path so Python 2.7 will never find 2.6 packages. The second rule is to install packages using the `--user` option in `pip install` or `setup.py install`. This results in the situation shown below where each Python can find common packages in the local user area:



Be sure to test that the package you installed works within the other Python environments.

Virtualenv

Virtualenv is a very useful tool for creating isolated Python environments. As seen in the linux non-root install it provides a way to make a virtual clone of an existing Python environment. This clone can then be used as the package installation location.

One use case is wanting to install a new or experimental version of a package without overwriting the existing production version in your baseline environment.

For a good introductory tutorial see http://iamzed.com/2009/05/07/a-primer-on-virtualenv/.

Anaconda environments

The Anaconda Python distribution, in conjunction with the `conda` package manager, makes it easy maintain multiple Python environments under one tree. This is extremely useful if you need to install different versions of packages, perhaps for testing or for running a particular application which has certain package requirements. See Python Packages and Environments with conda for an introduction.

Final exercises

Exercise [intermediate]: Fully install APLpy

Go to the APLpy install page and read the instructions. Manually install all of the Python package dependencies with the `--user` option or try the auto-install script available there.

For extra credit install the Montage C library as discussed on the APLpy install page. Then try to run the example Making a publication quality plot that was shown in the introductory talk. The necessary input files are in the `install_examples.tar` file.

Exercise [intermediate]: Install HDF5 and PyTables

Install HDF5 and PyTables. This will let you read HDF5 tables in Python. HDF5 is a data file format which can store and manipulate extremely large or complex datasets in a scalable manner. It is the baseline for some data-heavy facilities such as LOFAR.

Exercise [expert]: Install SciPy and all dependencies from source

Attempt to follow the instructions for building from source in the Installing SciPy page. (No binary downloads!). This will be useful if you want to use the very latest development version of Python or else want to use the system-dependent build optimization so your numerical libraries are the fastest possible. For most people this is not needed.

If you can do this then consider yourself an expert on Python installation.

Page Source Back to Top

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Map.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Map.ipynb)

map()

map() is a function that takes in two arguments: a function and a sequence iterable. In the form: map(function, sequence)

The first argument is the name of a function and the second a sequence (e.g. a list). map() applies the function to all the elements of the sequence. It returns a new list with the elements changed by function.

When we went over list comprehension we created a small expression to convert Fahrenheit to Celsius. Let's do the same here but use map.

We'll start with two functions:

In [4]:
```
def fahrenheit(T):
    return ((float(9)/5)*T + 32)
def celsius(T):
    return (float(5)/9)*(T-32)

temp = [0, 22.5, 40,100]
```

Now lets see map() in action:

In [7]:
```
F_temps = map(fahrenheit, temp)

#Show
F_temps
```

Out[7]:
```
[32.0, 72.5, 104.0, 212.0]
```

In [8]:
```
# Convert back
map(celsius, F_temps)
```

Out[8]:
```
[0.0, 22.5, 40.0, 100.0]
```

In the example above we haven't used a lambda expression. By using lambda, we wouldn't have had to define and name the functions fahrenheit() and celsius().

In [10]:
```
map(lambda x: (5.0/9)*(x - 32), F_temps)
```

Out[10]:
```
[0.0, 22.5, 40.0, 100.0]
```

Great! We got the same result! Using map is much more commonly used with lambda expressions since the entire purpose of map() is to save effort on having to create manual for loops.

map() can be applied to more than one iterable. The iterables have to have the same length.

For instance, if we are working with two lists-map() will apply its lambda function to the elements of the argument lists, i.e. it first applies to the elements with the 0th index, then to the elements with the 1st index until the n-th index is reached.

For example lets map a lambda expression to two lists:

In [12]:
```
a = [1,2,3,4]
b = [5,6,7,8]
c = [9,10,11,12]

map(lambda x,y:x+y,a,b)
```

Out[12]:
```
[6, 8, 10, 12]
```

In [13]:
```
# Now all three lists
map(lambda x,y,z:x+y+z, a,b,c)
```

Out[13]:
```
[15, 18, 21, 24]
```

We can see in the example above that the parameter x gets its values from the list a, while y gets its values from b and z from list c. Go ahead and play with your own example to make sure you fully understand mapping to more than one iterable.

Great job! You should now have a basic understanding of the map() function.

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Reduce.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Reduce.ipynb)

reduce()

Many times students have difficulty understanding reduce() so pay careful attention to this lecture. The function reduce(function, sequence) continually applies the function to the sequence. It then returns a single value.

If seq = [s1, s2, s3, ... , sn], calling reduce(function, sequence) works like this:

- At first the first two elements of seq will be applied to function, i.e. func(s1,s2)
- The list on which reduce() works looks now like this: [function(s1, s2), s3, ... , sn]
- In the next step the function will be applied on the previous result and the third element of the list, i.e. function(function(s1, s2),s3)
- The list looks like this now: [function(function(s1, s2),s3), ... , sn]
- It continues like this until just one element is left and return this element as the result of reduce()

Lets see an example:

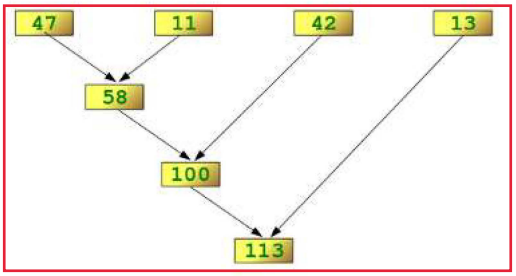
In [16]:
```
lst =[47,11,42,13]
reduce(lambda x,y: x+y,lst)
```

Out[16]:
```
113
```

Lets look at a diagram to get a better understanding of what is going on here:

In [9]:
```
from IPython.display import Image
Image('http://www.python-course.eu/images/reduce_diagram.png')
```

Out[9]:



Note how we keep reducing the sequence until a single final value is obtained. Lets see another example:

In [20]:
```
#Find the maximum of a sequence (This already exists as max())
max_find = lambda a,b: a if (a > b) else b
```

In [21]:
```
#Find max
reduce(max_find,lst)
```

Out[21]:
```
47
```

Hopefully you can see how useful reduce can be in various situations. Keep it in mind as you think about your code projects!

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Filter.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Filter.ipynb)

filter

The function filter(function, list) offers a convenient way to filter out all the elements of an iterable, for which the function returns True.

The function filter(function(),l) needs a function as its first argument. The function needs to return a Boolean value (either True or False). This function will be applied to every element of the iterable. Only if the function returns True will the element of the iterable be included in the result.

Lets see some examples:

In [4]:
```
#First let's make a function
def even_check(num):
    if num%2 ==0:
        return True
```

Now let's filter a list of numbers. Note: putting the function into filter without any parenthesis might feel strange, but keep in mind that functions are objects as well.

In [3]:
```
lst =range(20)

filter(even_check,lst)
```

Out[3]:
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

filter() is more commonly used with lambda functions, this because we usually use filter for a quick job where we don't want to write an entire function. Lets repeat the example above using a lambda expression:

In [5]:
```
filter(lambda x: x%2==0,lst)
```

Out[5]:
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Great! You should now have a solid understanding of filter() and how to apply it to your code!

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Zip.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Zip.ipynb)

zip

zip() makes an iterator that aggregates elements from each of the iterables.

Returns an iterator of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted. With a single iterable argument, it returns an iterator of 1-tuples. With no arguments, it returns an empty iterator.

zip() is equivalent to:

```
def zip(*iterables):
    # zip('ABCD', 'xy') --> Ax By
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
                return
            result.append(elem)
        yield tuple(result)
```

zip() should only be used with unequal length inputs when you don't care about trailing, unmatched values from the longer iterables.

Lets see it in action in some examples:

Examples

In [1]:
```
x = [1,2,3]
y = [4,5,6]

# Zip the lists together
zip(x,y)
```

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

Note how tuples are returned. What if one iterable is longer than the other?

In [2]:
```
x = [1,2,3]
y = [4,5,6,7,8]

# Zip the lists together
zip(x,y)
```

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

Note how the zip is defined by the shortest iterable length. Its generally advised not to zip unequal length iterables unless your very sure you only need partial tuple pairings.

What happens if we try to zip together dictionaries?

In [4]:
```
d1 = {'a':1,'b':2}
d2 = {'c':4,'d':5}

zip(d1,d2)
```

Out[4]:
```
[('a', 'c'), ('b', 'd')]
```

This makes sense because simply iterating through the dictionaries will result in just the keys. We would have to call methods to mix keys and values:

In [6]:
```
zip(d2,d1.itervalues())
```

Out[6]:
```
[('c', 1), ('d', 2)]
```

Great! Finally lets use zip a to switch the keys and values of the two dictionaries:

In [7]:
```
def switcharoo(d1,d2):
    dout = {}

    for d1key,d2val in zip(d1,d2.itervalues()):
        dout[d1key] = d2val

    return dout
```

In [8]:
```
switcharoo(d1,d2)
```

Out[8]:
```
{'a': 4, 'b': 5}
```

Great! You can use zip to save a lot of typing in many situations! You should now have a good understanding of zip() and some possible use cases.

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Enumerate.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Enumerate.ipynb)

enumerate()

In this lecture we will learn about an extremely useful built-in function: enumerate(). Enumerate allows you to keep a count as you iterate through an object. It does this by returning a tuple in the form (count,element). The function itself is equivalent to:

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

Example

In [1]:
```
lst = ['a','b','c']

for number,item in enumerate(lst):
    print number
    print item
```

```
0
a
1
b
2
c
```

enumerate() becomes particularly useful when you have a case where you need to have some sort of tracker. For example:

In [3]:
```
for count,item in enumerate(lst):
    if count >= 2:
        break
    else:
        print item
```

```
a
b
```

Great! You should now have a good understanding of enumerate and its potential use cases.

all() and any() ¶

all() and any() are built-in functions in Python that allow us to conveniently check for boolean matching in an iterable. all() will return True if all elements in an iterable are True. It is the same as this function code:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

any() will return True if any of the elements in the iterable are True. It is equivalent to the following function code:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

Let's see a few examples of these functions. They should be fairly straightforward:

In [4]:
```
lst = [True,True,False,True]
```

In [5]:
```
all(lst)
```

Out[5]:
```
False
```

Returns False because not all elements are True.

In [6]:
```
any(lst)
```

Out[6]:
```
True
```

Returns True because at least one of the elements in the list is True

There you have it, you should have an understanding of how to use any() and all() in your code.

complex()

complex() returns a complex number with the value real + imag*1j or converts a string or number to a complex number.

If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If imag is omitted, it defaults to zero and the constructor serves as a numeric conversion like int and float. If both arguments are omitted, returns 0j.

If you are doing math or engineering that requires complex numbers (such as dynamics,control systems, or impedance of a circuit) this is a useful tool to have in Python.

Lets see some examples:

In [2]:
```
# Create 2+3j
complex(2,3)
```

Out[2]:
```
(2+3j)
```

In [4]:
```
complex(10,1)
```

Out[4]:
```
(10+1j)
```

We can also pass strings:

In [6]:
```
complex('12+2j')
```

Out[6]:
```
(12+2j)
```

Thats really all there is to this useful function. Keep it in mind if you are ever dealing with complex numbers in Python!

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Decorators.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Decorators.ipynb)

Decorators

Decorators can be thought of as functions which modify the *functionality* of another function. They help to make your code shorter and more "Pythonic".

To properly explain decorators we will slowly build up from functions. Make sure to restart the Python and the Notebooks for this lecture to look the same on your own computer. So lets break down the steps:

Functions Review

In [1]:
```
def func():
    return 1
```

In [2]:
```
func()
```

Out[2]:
```
1
```

Scope Review

Remember from the nested statements lecture that Python uses Scope to know what a label is referring to. For example:

In [6]:
```
s = 'Global Variable'

def func():
    print locals()
```

Remember that Python functions create a new scope, meaning the function has its own namespace to find variable names when they are mentioned within the function. We can check for local variables and global variables with the local() and globals() functions. For example:

In [7]:
```
print globals()
```

```
{'_dh': [u'/Users/marci/Udemy-Complete-Python-Bootcamp'], '__': '', '_i': u"s = 'Global Variable'\n\ndef func():\n
```

Here we get back a dictionary of all the global variables, many of them are predefined in Python. So let's go ahead and look at the keys:

In [8]:
```
print globals().keys()
```

```
['_dh', '__', '_i', 'quit', '__builtins__', 's', '_ih', '__builtin__', '_2', 'func', '__name__', '___', '_', '_sh', '_
```

Note how **s** is there, the Global Variable we defined as a string:

In [10]:
```
globals()['s']
```

Out[10]:
```
'Global Variable'
```

Now lets run our function to check for any local variables in the func() (there shouldn't be any)

In [11]:
```
func()
```

```
{}
```

Great! Now lets continue with building out the logic of what a decorator is. Remember that in Python **everything is an object**. That means functions are objects which can be assigned labels and passed into other functions. Lets start with some simple examples:

In [12]:
```
def hello(name='Jose'):
    return 'Hello '+name
```

In [13]:
```
hello()
```

Out[13]:
```
'Hello Jose'
```

Assign a label to the function. Note that e are not using parentheses here because we are not calling the function hello, instead we are just putting it into the greet variable.

In [14]:
```
greet = hello
```

In [15]:
```
greet
```

Out[15]:
```
<function __main__.hello>
```

In [16]:
```
greet()
```

Out[16]:
```
'Hello Jose'
```

This assignment is not attached to the original function:

In [17]:
```
del hello
```

In [18]:
```
hello()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-18-a803225a2f97> in <module>()
----> 1 hello()

NameError: name 'hello' is not defined
```

In [19]:
```
greet()
```

Out[19]:
```
'Hello Jose'
```

Functions within functions

Great! So we've seen how we can treat functions as objects, now lets see how we can define functions inside of other functions:

In [26]:
```
def hello(name='Jose'):
    print 'The hello() function has been executed'

    def greet():
        return '\t This is inside the greet() function'

    def welcome():
        return "\t This is inside the welcome() function"

    print greet()
    print welcome()
    print "Now we are back inside the hello() function"
```

In [27]:
```
hello()
```

```
The hello() function has been executed
         This is inside the greet() function
         This is inside the welcome() function
Now we are back inside the hello() function
```

In [29]:
```
welcome()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-29-efaf77b113fd> in <module>()
----> 1 welcome()

NameError: name 'welcome' is not defined
```

Note how due to scope, the welcome() function is not defined outside of the hello() function. Now lets learn about returning functions from within functions:

Returning Functions

In [34]:
```
def hello(name='Jose'):

    def greet():
        return '\t This is inside the greet() function'

    def welcome():
        return "\t This is inside the welcome() function"

    if name == 'Jose':
        return greet
    else:
        return welcome
```

In [39]:
```
x = hello()
```

Now lets see what function is returned if we set x = hello(), note how the closed parenthesis means that name ahs been defined as Jose.

In [40]:
```
x
```

Out[40]:
```
<function __main__.greet>
```

Great! Now we can see how x is pointing to the greet function inside of the hello function.

In [42]:
```
print x()
```

```
        This is inside the greet() function
```

Lets take a quick look at the code again.

In the if/else clause we are returning greet and welcome, not greet() and welcome().

This is because when you put a pair of parentheses after it, the function gets executed; whereas if you don't put parenthesis after it, then it can be passed around and can be assigned to other variables without executing it.

When we write x = hello(), hello() gets executed and because the name is Jose by default, the function greet is returned. If we change the statement to x = hello(name = "Sam") then the welcome function will be returned. We can also do print hello()() which outputs now you are in the greet() function.

Functions as Arguments

Now lets see how we can pass functions as arguments into other functions:

In [43]:
```
def hello():
    return 'Hi Jose!'

def other(func):
    print 'Other code would go here'
    print func()
```

In [45]:
```
other(hello)
```

```
Other code would go here
Hi Jose!
```

Great! Note how we can pass the functions as objects and then use them within other functions. Now we can get started with writing our first decorator:

Creating a Decorator

In the previous example we actually manually created a Decorator. Here we will modify it to make its use case clear:

In [46]:
```
def new_decorator(func):

    def wrap_func():
        print "Code would be here, before executing the func"

        func()

        print "Code here will execute after the func()"

    return wrap_func

def func_needs_decorator():
    print "This function is in need of a Decorator"
```

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

```
This function is in need of a Decorator
```

In [50]:
```
# Reassign func_needs_decorator
func_needs_decorator = new_decorator(func_needs_decorator)
```

In [51]:
```
func_needs_decorator()
```

```
Code would be here, before executing the func
This function is in need of a Decorator
Code here will execute after the func()
```

So what just happened here? A decorator simple wrapped the function and modified its behavior. Now lets understand how we can rewrite this code using the @ symbol, which is what Python uses for Decorators:

In [52]:
```
@new_decorator
def func_needs_decorator():
    print "This function is in need of a Decorator"
```

In [53]:
```
func_needs_decorator()
```

```
Code would be here, before executing the func
This function is in need of a Decorator
Code here will execute after the func()
```

Great! You've now built a Decorator manually and then saw how we can use the @ symbol in Python to automate this and clean our code. You'll run into Decorators a lot if you begin using Python for Web Development, such as Flask or Django!

Decorators Homework (Optional)

Since you won't run into decorators until further in your coding career, this homework is optional. Check out the Web Framework Flask (http://flask.pocoo.org/). You can use Flask to create web pages with Python (as long as you know some HTML and CSS) and they use decorators a lot! Learn how they use view decorators (http://flask.pocoo.org/docs/0.10/patterns/viewdecorators/). Don't worry if you don't completely understand everything about Flask, the main point of this optional homework is that you have an awareness of decorators in Web Frameworks, that way if you decide to become a "Full-Stack" Python Web Developer, you won't find yourself perplexed by decorators. You can also check out Django (https://www.djangoproject.com/) another (and more popular) web framework for Python which is a bit more heavy duty.

Also for some additional info:

A framework is a type of software library that provides generic functionality which can be extended by the programmer to build applications. Flask and Django are good examples of frameworks intended for web development.

A framework is distinguished from a simple library or API. An API is a piece of software that a developer can use in his application. A framework is more encompassing: your entire application is structured around the framework (i.e. it provides the framework around which you build your software).

Great job!

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Iterators and Generators.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Iterators and Generators.ipynb)

Iterators and Generators

In this section of the course we will be learning about the difference between iteration and generation in Python and how to construct our own Generators with the *yield* statement. Generators allow us to generate as we go along, instead of holding everything in memory.

We've touch on this topic in the past when discussing the range() function in Python 2 and the similar xrange(), with the difference being the xrange() was a generator.

Lets explore a little deep. We've learned how to create functions with **def** and the **return** statement. Generator functions allow us to write a function that can send back a value and then later resume to pick up where it left off. This type of function is a generator in Python, allowing us to generate a sequence of values over time. The main difference in syntax will be the use of a **yield** statement.

In most aspects, a generator function will appear very similar to a normal function. The main difference is when a generator function is compiled they become an object that support an iteration protocol. That means when they are called in your code the don't actually return a value and then exit, the generator functions will automatically suspend and resume their execution and state around the last point of value generation. The main advantage here is that instead of having to compute an entire series of values upfront and the generator functions can be suspended, this feature is known as *state suspension*.

To start getting a better understanding of generators, lets go ahead and see how we can create some.

In [6]:
```
# Generator function for the cube of numbers (power of 3)
def gencubes(n):
    for num in range(n):
        yield num**3
```

In [10]:
```
for x in gencubes(10):
    print x
```

```
0
1
8
27
64
125
216
343
512
729
```

Great! Now since we have a generator function we don't have to keep track of every single cube we created.

Generators are best for calculating large sets of results (particularly in calculations that involve loops themselves) in cases where we don't want to allocate the memory for all of the results at the same time.

As we've noted in previous lectures (such as range()) many Standard Library functions that return lists in Python 2 have been modified to return generators in Python 3 because generators.

Lets create another example generator which calculates fibonacci (https://en.wikipedia.org/wiki/Fibonacci_number) numbers:

In [1]:
```
def genfibon(n):
    '''
    Generate a fibonnaci sequence up to n
    '''
    a = 1
    b = 1
    for i in range(n):
        yield a
        a,b = b,a+b
```

In [2]:
```
for num in genfibon(10):
    print num
```

```
1
1
2
3
5
8
13
21
34
55
```

What is this was a normal function, what would it look like?

In [3]:
```
def fibon(n):
    a = 1
    b = 1
    output = []

    for i in range(n):
        output.append(a)
        a,b = b,a+b

    return output
```

In [4]:
```
fibon(10)
```

Out[4]:
```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Notice that if we call some huge value of n (like 100000) the second function will have to keep track of every single result, when in our case we actually only care about the previous result to generate the next one!

next() and iter() built-in functions

A key to fully understanding generators is the next function() and the iter() function.

The next function allows us to access the next element in a sequence. Lets check it out:

In [20]:
```
def simple_gen():
    for x in range(3):
        yield x
```

In [21]:
```
# Assign simple_gen
g = simple_gen()
```

In [22]:
```
print next(g)
```

```
0
```

In [23]:
```
print next(g)
```

```
1
```

In [24]:
```
print next(g)
```

```
2
```

In [25]:
```
print next(g)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-25-d013a5691c47> in <module>()
----> 1 print next(g)

StopIteration:
```

After yielding all the values next() caused a StopIteration error. What this error informs us of is that all the values have been yielded.

You might be wondering that why don't we get this error while using a for loop? The for loop automatically catches this error and stops calling next.

Lets go ahead and check out how to use iter(). You remember that strings are iterables:

In [26]:
```
s = 'hello'

#Iterate over string
for let in s:
    print let
```

```
h
e
l
l
o
```

But that doesn't mean the string itself is an *iterator*! We can check this with the next() function:

In [27]:
```
next(s)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-bc0566bea448> in <module>()
----> 1 next(s)

TypeError: str object is not an iterator
```

Interesting, this means that a string object supports iteration, but we can not directly iterate over it as we could with a generator function. The iter() function allows us to do just that!

In [28]:
```
s_iter = iter(s)
```

In [29]:
```
next(s_iter)
```

Out[29]:
```
'h'
```

In [30]:
```
next(s_iter)
```

Out[30]:
```
'e'
```

Great! Now you know how to convert objects that are iterable into iterators themselves!

The main takeaway from this lecture is that using the yield keyword at a function will cause the function to become a generator. This change can save you a lot of memory for large use cases. For more information on generators check out:

Stack Overflow Answer (http://stackoverflow.com/questions/1756096/understanding-generators-in-python)

Another StackOverflow Answer (http://stackoverflow.com/questions/231767/what-does-the-yield-keyword-do-in-python)

In [ ]:
```

```

Complete-Python-Bootcamp (/github/jmportilla/Complete-Python-Bootcamp/tree/master)
/ Iterators and Generators Homework.ipynb (/github/jmportilla/Complete-Python-Bootcamp/tree/master/Iterators and Generators Homework.ipynb)

Iterators and Generators Homework

Problem 1 ¶

Create a generator that generates the squares of numbers up to some number N.

In [10]:
```
def gensquares(N):
    pass
```

In [11]:
```
for x in gensquares(10):
    print x
```

```
0
1
4
9
16
25
36
49
64
81
```

```
def gensquares(N):
    for x in range(N):
        yield x ** 2
g = iter(gensquares(10))
g.next()
g.next()
```

Problem 2

Create a generator that yields "n" random numbers between a low and high number (that are inputs). Note: Use the random library. For example:

In [6]:
```
import random

random.randint(1,10)
```

Out[6]: 6

In [7]:
```
def rand_num(low,high,n):
    pass
```

In [9]:
```
for num in rand_num(1,10,12):
    print num
```

```
6
7
6
9
9
4
3
7
8
1
6
1
```

```
import random
def rand_num(low, high, n):
    for x in range(n):
        yield random.randint(low, high)

for num in rand_num(1, 10, 12):
    print num

g = iter(rand_num(1, 10, 12))
g.next()
g.next()
```

Problem 3

Use the iter() function to convert the string below

In [1]:
```
s = 'hello'

#code here
```

```
s = 'hello'
s_iter = iter(s)
s_iter.next()
s_iter.next()
```

Problem 4

Explain a use case for a generator using a yield statement where you would not want to use a normal function with a return statement.

Extra Credit!

Can you explain what *gencomp* is in the code below? (Note: We never covered this in lecture! You will have to do some googling/Stack Overflowing!)

In [18]:
```
my_list = [1,2,3,4,5]

gencomp = (item for item in my_list if item > 3)

for item in gencomp:
    print item
```

```
4
5
```

Hint google: generator comprehension is!

Great Job!

```
myList = [1, 2, 3, 4, 5]
gencomp = (item for item in myList if item > 3)

for item in gencomp:
    print item

gencomp_iter = iter(gencomp)
gencomp_iter.next()
gencomp_iter.next()
```

Iterators and Generators Homework - Solution

Problem 1

Create a generator that generates the squares of numbers up to some number N.

In [10]:
```
def gensquares(N):
    for i in range(N):
        yield i ** 2
```

In [11]:
```
for x in gensquares(10):
    print x
```

```
0
1
4
9
16
25
36
49
64
81
```

Problem 2

Create a generator that yields "n" random numbers between a low and high number (that are inputs). Note: Use the random library. For example:

In [6]:
```
import random

random.randint(1,10)
```

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

In [7]:
```
def rand_num(low,high,n):

    for i in range(n):
        yield random.randint(low, high)
```

In [9]:
```
for num in rand_num(1,10,12):
    print num
```

```
6
7
6
9
9
4
3
7
8
1
6
1
```

Problem 3

Use the iter() function to convert the string below

In [17]:
```
s = 'hello'

s = iter(s)

print next(s)
```

```
h
```

Problem 4

Explain a use case for a generator using a yield statement where you would not want to use a normal function with a return statement.

If the output has the potential of taking up a large amount of memory and you only intend to iterate through it, you would want to use a generator. (Multiple answers are acceptable here!)

Extra Credit!

Can you explain what bonus is in the code below? (Note: We never covered this in lecture!)

In [18]:
```
my_list = [1,2,3,4,5]

gencomp = (item for item in my_list if item > 3)

for item in gencomp:
    print item
```

```
4
5
```

Hint google: generator comprehension is!

Great Job!