Python3_Notes

May 24, 2020

Complete Python Bootcamp: Go from zero to hero in Python ***

Become a Python Programmer and learn one of employer's most requested skills of 2018!

Udmey

Tutorials point ***

1 Python

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL). This tutorial gives enough understanding on Python programming language.

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable.

- Python is Interpreted − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- Python is Interactive − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- Python is Object-Oriented − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- Python is a Beginner's Language − Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

1.1 Python's features include

- Easy-to-learn − Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- Easy-to-read − Python code is more clearly defined and visible to the eyes.
- Easy-to-maintain − Python's source code is fairly easy-to-maintain.
- A broad standard library − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- A broad standard library − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.

- Interactive Mode – Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- Portable – Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- Extendable – You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- Databases – Python provides interfaces to all major commercial databases.
- GUI Programming – Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- Scalable – Python provides a better structure and support for large programs than shell scripting.
- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

2 Python Object and Data Structure Basics

- Numbers.
- Variable Assignment.
- Strings.
- Print Formatting with Strings.
- Lists.
- Dictionaries.
- Tuples.
- Sets and Booleans.
- Files.

2.1 Python Basic Data types

Name	**Type**	**Description**
Integers	int	Whole numbers, such as: 3 ,300,200
Strings	str	Ordered sequence of characters: “hello” ‘Sammy’ “2000” “ ”
Lists	list	Ordered sequence of objects: [10,“hello”,200.3]
Dictionaries	dict	Unordered Key:Value pairs: {“mykey” : “value” , “name” : “Frankie”}
Tuples	tup	Ordered immutable sequence of objects: (10,“hello”,200.3)
Sets	set	Unordered collection of unique objects: {“a”,“b”}

Name	Type	Description
Booleans	bool	Logical value indicating True or False

2.2 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:

2.2.1 Basic Arithmetic

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

[1]:
```
3
```

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

[2]:
```
1
```

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

[3]:
```
4
```

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

[4]:
```
1.5
```

[5]:
```
# Floor Division
7//4
```

[5]:
```
1
```

[6]:
```python
# Mod
10 % 4
```

[6]:
```
2
```

[7]:
```python
# Powers
2**3
```

[7]:
```
8
```

2.2.2 Variable Assignments

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables. The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

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.

[8]:
```python
# 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.

[9]:
```python
# Adding the objects
a+a
```

[9]:
```
10
```

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 :'",<>/?|()!@#$%^&*~-+
4. It's considered best practice (PEP8) that names are lowercase.
5. Avoid using the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names.
6. Avoid using words that have special meaning in Python like "list" and "str"

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

[10]:
```python
# 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
```

Python allows you to assign a single value to several variables simultaneously.

```
[11]: a = b = c = 1
```

2.2.3 Dynamic Typing

Python uses *dynamic typing*, meaning you can reassign variables to different data types. This makes Python very flexible in assigning data types; it differs from other languages that are *statically typed*.

```
[12]: my_dogs = 2
```

```
[13]: my_dogs
```

```
[13]: 2
```

```
[14]: my_dogs = ['Sammy', 'Frankie']
```

```
[15]: my_dogs
```

```
[15]: ['Sammy', 'Frankie']
```

2.2.4 Pros and Cons of Dynamic Typing

Pros of Dynamic Typing

- very easy to work with
- faster development time

Cons of Dynamic Typing

- may result in unexpected bugs!
- you need to be aware of `type()`

2.2.5 Determining variable type with `type()`

You can check what type of object is assigned to a variable using Python's built-in `type()` function. Common data types include: * **int** (for integer) * **float** * **str** (for string) * **list** * **tuple** * **dict** (for dictionary) * **set** * **bool** (for Boolean True/False)

2.3 Strings

Strings are sequences of characters, using the syntax of either single quotes or double quotes. Because strings are ordered sequences it means we can using indexing and slicing to grab sub-sections of the string. Indexing notation uses [ ] notation after the string (or variable assigned the string). Indexing allows you to grab a single character from the string…

Slicing allows you to grab a subsection of multiple characters, a "slice" of the string. This has the following syntax:

`[start:stop:step]`

- start is a numerical index for the slice start
- stop is the index you will go up to (but not include)
- step is the size of the "jump" you take.

Strings are used in Python to record text information, such as names. 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.) String Indexing and Slicing
4.) String Properties
5.) String Methods
6.) Print Formatting
```

2.3.1 Creating a String

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

```
[16]: # Single word
      'hello'
```

```
[16]: 'hello'
```

2.3.2 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.

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

```
[17]: 'Hello World'
```

2.3.3 String Basics

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

```
[18]: len('Hello World')
```

```
[18]: 11
```

Python's built-in len() function counts all of the characters in the string, including spaces and punctuation.

2.3.4 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 then walk through a few examples of indexing.

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

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

```
[20]: 'H'
```

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

```
[21]: '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:

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

```
[22]: 'Hello World'
```

[23]:
```
# Grab everything, but go in step sizes of 2
s[::2]
```

[23]:
```
'HloWrd'
```

[24]:
```
# We can use this to print a string backwards
s[::-1]
```

[24]:
```
'dlroW olleH'
```

2.3.5 String Properties

It's 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:

[25]:
```
# Let's try to change the first letter to 'x'
#s[0] = 'x'
```

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

Something we *can* do is concatenate strings!

[26]:
```
# Concatenate strings!
s + ' concatenate me!'
```

[26]:
```
'Hello World concatenate me!'
```

2.3.6 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:

[27]:
```
# Upper Case a string
s.upper()
```

[27]:
```
'HELLO WORLD'
```

[28]:
```python
# Lower case
s.lower()
```

[28]:
```
'hello world'
```

[29]:
```python
# Split a string by blank space (this is the default)
s.split()
```

[29]:
```
['Hello', 'World']
```

[30]:
```python
# Split by a specific element (doesn't include the element that was split on)
s.split('W')
```

[30]:
```
['Hello ', 'orld']
```

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

2.3.7 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:

[31]:
```python
'Insert another string with curly brackets: {}'.format('The inserted string')
```

[31]:
```
'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!

2.3.8 String Formatting

String formatting lets you inject items into a string rather than trying to chain items together using commas or string concatenation. As a quick comparison, consider:

```
player = 'Thomas'
points = 33

'Last night, '+player+' scored '+str(points)+' points.'  # concatenation

f'Last night, {player} scored {points} points.'          # string formatting
```

There are three ways to perform string formatting. * The oldest method involves placeholders using the modulo `%` character. * An improved technique uses the `.format()` string method. * The newest method, introduced with Python 3.6, uses formatted string literals, called *f-strings*.

Since you will likely encounter all three versions in someone else's code, we describe each of them here.

2.3.9 Formatting with placeholders

You can use %s to inject strings into your print statements. The modulo `%` is referred to as a "string formatting operator".

[32]:
```
print("I'm going to inject %s text here, and %s text here." %('some','more'))
```

```
I'm going to inject some text here, and more text here.
```

You can also pass variable names:

[33]:
```
x, y = 'some', 'more'
print("I'm going to inject %s text here, and %s text here."%(x,y))
```

```
I'm going to inject some text here, and more text here.
```

As another example, `\t` inserts a tab into a string.

[34]:
```
print('I once caught a fish %s.' %'this \tbig')
print('I once caught a fish %r.' %'this \tbig')
```

```
I once caught a fish this       big.
I once caught a fish 'this \tbig'.
```

The `%s` operator converts whatever it sees into a string, including integers and floats. The `%d` operator converts numbers to integers first, without rounding. Note the difference below:

[35]:
```
print('I wrote %s programs today.' %3.75)
print('I wrote %d programs today.' %3.75)
```

```
I wrote 3.75 programs today.
I wrote 3 programs today.
```

2.3.10 Padding and Precision of Floating Point Numbers

Floating point numbers use the format %5.2f. Here, 5 would be the minimum number of characters the string should contain; these may be padded with whitespace if the entire number does not have this many digits. Next to this, .2f stands for how many numbers to show past the decimal point. Let's see some examples:

[36]:
```
print('Floating point numbers: %5.2f' %(13.144))
```

```
Floating point numbers: 13.14
```

[37]:
```
print('Floating point numbers: %25.2f' %(13.144))
```

```
Floating point numbers:                     13.14
```

For more information on string formatting with placeholders visit https://docs.python.org/3/library/stdtypes.html#old-string-formatting

2.3.11 Alignment, padding and precision with `.format()`

Within the curly braces you can assign field lengths, left/right alignments, rounding parameters and more

[38]:
```python
print('{0:8} | {1:9}'.format('Fruit', 'Quantity'))
print('{0:8} | {1:9}'.format('Apples', 3.))
print('{0:8} | {1:9}'.format('Oranges', 10))
```

```
Fruit    | Quantity
Apples   |       3.0
Oranges  |        10
```

2.3.12 Formatted String Literals (f-strings)

Introduced in Python 3.6, f-strings offer several benefits over the older `.format()` string method described above. For one, you can bring outside variables immediately into to the string rather than pass them as arguments through `.format(var)`.

[39]:
```python
name = 'Fred'

print(f"He said his name is {name}.")
```

```
He said his name is Fred.
```

Pass `!r` to get the string representation:

[40]:
```python
print(f"He said his name is {name!r}")
```

```
He said his name is 'Fred'
```

2.4 Lists

Lists are ordered sequences that can hold a variety of object types.They use [] brackets and commas to separate objects in the list. Lists support indexing and slicing. Lists can be nested and also have a variety of useful methods that can be called off of them.

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!

```
[41]: # 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:

```
[42]: 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.

```
[43]: len(my_list)
```

```
[43]: 4
```

2.4.1 Indexing and Slicing

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

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

```
[45]: # Grab element at index 0
      my_list[0]
```

```
[45]: 'one'
```

```
[46]: my_list + ['new item']
```

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

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

```
[47]: ['one', 'two', 'three', 4, 5, 'one', 'two', 'three', 4, 5]
```

2.4.2 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:

[48]:
```
# Create a new list
list1 = [1,2,3]
list1
```

[48]:
```
[1, 2, 3]
```

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

[49]:
```
# Append
list1.append('append me!')
list1
```

[49]:
```
[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:

[50]:
```
# Pop off the 0 indexed item
list1.pop(0)
```

[50]:
```
1
```

[51]:
```
# Show
list1
```

[51]:
```
[2, 3, 'append me!']
```

2.4.3 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!

[52]:
```
# 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]
```

[53]:
```
# Show
matrix
```

[53]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

2.4.4 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!

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

```
[55]: first_col
```

```
[55]: [1, 4, 7]
```

2.4.5 Dictionaries

Dictionaries are unordered mappings for storing objects. Previously we saw how lists store objects in an ordered sequence, dictionaries use a key-value pairing instead. This key-value pair allows users to quickly grab objects without needing to know an index location.

Dictionaries use curly braces and colons to signify the keys and their associated values. * {'key1':'value1','key2':'value2'} So when to choose a list and when to choose a dictionary?

Dictionaries: Objects retrieved by key name. Unordered and can not be sorted.

Lists: Objects retrieved by location. Ordered Sequence can be indexed or sliced.

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.

2.4.6 Constructing a Dictionary

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

```
[56]: # Make a dictionary with {} and : to signify a key and a value
      my_dict = {'key1':'value1','key2':'value2'}
      # Call values by their key
      my_dict['key2']
```

```
[56]: 'value2'
```

2.4.7 Nesting with Dictionaries

Hopefully you're 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:

```
[57]: # Dictionary nested inside a dictionary nested inside a dictionary
      d = {'key1':{'nestkey':{'subnestkey':'value'}}}
```

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

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

```
[58]: 'value'
```

2.4.8 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:

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

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

```
[60]: dict_keys(['key1', 'key2', 'key3'])
```

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

```
[61]: dict_values([1, 2, 3])
```

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

```
[62]: dict_items([('key1', 1), ('key2', 2), ('key3', 3)])
```

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.

2.5 Tuples

Tuples are very similar to lists. However they have one key difference – immutability. Once an element is inside a tuple, it can not be reassigned. Tuples use parenthesis: (1,2,3)

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.

2.5.1 Constructing Tuples

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

```
[63]: # Create a tuple
      t = (1,2,3)
      # Check len just like a list
      len(t)
```

```
[63]: 3
```

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

      # Show
      t
```

```
[64]: ('one', 2)
```

2.5.2 Basic Tuple Methods

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

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

```
[65]: 0
```

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

```
[66]: 1
```

2.5.3 Immutability

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

```
[67]: t[0]= 'change'
```

```
          ␣
      ↪---------------------------------------------------------------------------

          TypeError                                 Traceback (most recent call␣
      ↪last)

          <ipython-input-67-1257c0aa9edd> 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.

2.5.4 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 a tuple becomes 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 Sets and Booleans!!

2.6 Sets

Sets are unordered collections of **unique** elements. Meaning there can only be one representative of the same object. We can construct them by using the set() function. Let's go ahead and make a set to see how it works

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

```
[ ]: # Create a list with repeats
     list1 = [1,1,2,2,3,4,5,6,1,1]
     # Cast as set to get unique values
     set(list1)
```

2.7 Booleans

Booleans are operators that allow you to convey **True** or **False** statements. These are very important later on when we deal with control flow and logic!

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).

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

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

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

2.8 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!

2.8.1 IPython Writing a File

This function is specific to jupyter notebooks! Alternatively, quickly create a simple .txt file with sublime text editor.

[68]:
```
%%writefile test.txt
Hello, this is a quick test file.
```

```
Overwriting test.txt
```

2.8.2 Python Opening a file

Let's being by opening the file test.txt that is located in the same directory as this notebook. For now we will work with files located in the same directory as the notebook or .py script you are using.

It is very easy to get an error on this step:

[69]:
```
myfile = open('whoops.txt')
```

```
       ␣
 ↪---------------------------------------------------------------------------

        FileNotFoundError                         Traceback (most recent call␣
 ↪last)

        <ipython-input-69-410403f4f4b4> in <module>
    ----> 1 myfile = open('whoops.txt')


        FileNotFoundError: [Errno 2] No such file or directory: 'whoops.txt'
```

To avoid this error,make sure your .txt file is saved in the same location as your notebook, to check your notebook location, use **pwd**:

[ ]:
```
pwd
```

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

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

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

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:

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

```
[ ]: my_file.close()
```

2.8.3 Writing to a File

By default, 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:

```
[ ]: # Add a second argument to the function, 'w' which stands for write.
     # Passing 'w+' lets us read and write to the file

     my_file = open('test.txt','w+')
```

2.8.4 Use caution!

Opening a file with `'w'` or `'w+'` truncates the original, meaning that anything that was in the original file **is deleted**!

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

```
[70]: # Read the file
      my_file.seek(0)
      my_file.read()
```

```
       ␣
↪---------------------------------------------------------------------------

       NameError                                 Traceback (most recent call␣
↪last)

       <ipython-input-70-946fbe6e2856> in <module>
         1 # Read the file
   ----> 2 my_file.seek(0)
         3 my_file.read()


       NameError: name 'my_file' is not defined
```

2.8.5 Appending to a File

Passing the argument `'a'` opens the file and puts the pointer at the end, so anything written is appended. Like `'w+'`, `'a+'` lets us read and write to a file. If the file does not exist, one will be created.

```
[ ]: my_file = open('test.txt','a+')
     my_file.write('\nThis is text being appended to test.txt')
     my_file.write('\nAnd another line here.')
```

```
[ ]: my_file.seek(0)
     print(my_file.read())
```

```
[ ]: my_file.close()
```

2.8.6 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:

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

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

```
[ ]: for line in open('test.txt'):
         print(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. It's 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.

```
[ ]: with open('myfile.txt',mode ='r') as f:
         content = f.read()
```

```
[ ]: %%writefile newmyfile.txt
     this is first line
     this is the second line
```

```
[ ]: with open('newmyfile.txt',mode = 'a') as f:
         f.write('Four on forth')
```

```
[ ]: with open('myfile.txt',mode ='r') as f:
         content = f.read()
         print(content)
```

3 Python Comparison Operators

3.1 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

In the table below, a=3 and b=4.

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 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

```
[71]: 2 == 2
```

```
[71]: True
```

- Less than or Equal to *

```
[72]: 2 <= 2
```

```
[72]: True
```

```
[73]: 2 <= 4
```

```
[73]: 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

3.2 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 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:

```
[74]: 1 < 2 < 3
```

```
[74]: True
```

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

```
[75]: 1<2 and 2<3
```

```
[75]: True
```

4 Introduction to Python Statements

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.

Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

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.

4.1 Python vs Other Languages

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

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

Version 1 (Other Languages)

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

Version 2 (Python)

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

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:

4.2 Indentation

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements. Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation.

A code block (body of a function, loop etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.

Generally four whitespaces are used for indentation and is preferred over tabs.

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

`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!

4.3 if

Let's begin to learn about control flow. We often only want certain code to execute when a particular condition has been met. For example, if my dog is hungry (some condition), then I will feed the dog (some action). To control this flow of logic we use some keywords: * if * elif * else

Control Flow syntax makes use of colons and indentation (whitespace). Syntax of an if statement

`python if some_condition:     execute some code`

1. if : An if statement consists of a boolean expression followed by one or more statements.
2. if…else statements: An if statement can be followed by an optional else statement, which executes when the boolean expression is FALSE.
3. nested if statements: You can use one if or else if statement inside another if or else if statement(s).

[76]:
```
if True:
    print('It was true!')
```

```
It was true!
```

Let's add in some else logic:

```
[77]: 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
```

4.4 for Loops

Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

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 keys or values.

We've already seen the for statement a little bit in past lectures but now let's 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 your 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.

```
[78]: # We'll learn how to automate this sort of list in the next lecture
      list1 = [1,2,3,4,5,6,7,8,9,10]

      for num in list1:
          print(num)
```

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

```
9
10
```

[79]:
```python
tup = (1,2,3,4,5)

for t in tup:
    print(t)
```

```
1
2
3
4
5
```

[80]:
```python
list2 = [(2,4),(6,8),(10,12)]
```

[81]:
```python
for tup in list2:
    print(tup)
```

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

[82]:
```python
# Now with unpacking!
for (t1,t2) in list2:
    print(t1)
```

```
2
6
10
```

[83]:
```python
d = {'k1':1,'k2':2,'k3':3}
```

[84]:
```python
for item in d:
    print(item)
```

```
k1
k2
k3
```

4.5 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 statements
else:
    final code statements
```

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

[85]:
```python
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. It's important to note that once this occurred the code stopped. Let's see how we could add an else statement:

[86]:
```python
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!
```

4.6 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. Terminates the loop statement and transfers execution to the statement immediately following the loop.
- continue: Goes to the top of the closest enclosing loop.Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
- pass: Does nothing at all. The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

Thinking about break and continue statements, the general format of the while loop looks like this: `python while test:     code statement     if test:           break     if test:           continue   else:     code`

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.

Let's go ahead and look at some examples!

[87]:
```python
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
```

4.7 Useful Operators

There are a few built-in functions and "operators" in Python that don't fit well into any category, so we will go over them in this lecture, let's begin!

4.7.1 range

The range() type returns an immutable sequence of numbers between the given start integer to the stop integer. range() constructor has two forms of definition: * range(stop) * range(start, stop[, step])

range() takes mainly three arguments having the same use in both definitions:

- start - integer starting from which the sequence of integers is to be returned
- stop - integer before which the sequence of integers is to be returned. The range of integers end at stop - 1.
- step (Optional) - integer value which determines the increment between each integer in the sequence

The range function allows you to quickly *generate* a list of integers, this comes in handy a lot, so take note of how to use it! There are 3 parameters you can pass, a start, a stop, and a step size. Let's see some examples:

[88]:
```
range(0,11)
```

[88]:
```
range(0, 11)
```

Note that this is a **generator** function, so to actually get a list out of it, we need to cast it to a list with **list()**. What is a generator? Its a special type of function that will generate information and not need to save it to memory. We haven't talked about functions or generators yet, so just keep this in your notes for now, we will discuss this in much more detail in later on in your training!

```
[89]: # Notice how 11 is not included, up to but not including 11, just like slice␣
       ↪notation!
      list(range(0,11))
```

```
[89]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
[90]: # odd --------- even
      list(range(1,10,2)), list(range(0,10,2))
```

```
[90]: ([1, 3, 5, 7, 9], [0, 2, 4, 6, 8])
```

```
[91]: list(range(10,-10,-1))
```

```
[91]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

4.7.2 enumerate

The enumerate() method adds counter to an iterable and returns it. The returned object is a enumerate object.

The enumerate() method takes two parameters:

- iterable - a sequence, an iterator, or objects that supports iteration
- start (optional) - enumerate() starts counting from this number. If start is omitted, 0 is taken as start.

enumerate is a very useful function to use with for loops. Let's imagine the following situation:

```
[92]: x = ['a','b','c','d']
      a= enumerate(x, start = 10)
      list(a)
```

```
[92]: [(10, 'a'), (11, 'b'), (12, 'c'), (13, 'd')]
```

```
[93]: index_count = 0

      for letter in 'abcde':
          print("At index {} the letter is {}".format(index_count,letter))
          index_count += 1
```

```
At index 0 the letter is a
At index 1 the letter is b
At index 2 the letter is c
At index 3 the letter is d
At index 4 the letter is e
```

Keeping track of how many loops you've gone through is so common, that enumerate was created so you don't need to worry about creating and updating this index_count or loop_count variable

4.7.3 zip

The zip() function take iterables (can be zero or more), makes iterator that aggregates elements based on the iterables passed, and returns an iterator of tuples.

Notice the format enumerate actually returns, let's take a look by transforming it to a list()

```
[94]: list(enumerate('abcde'))
```

```
[94]: [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]
```

It was a list of tuples, meaning we could use tuple unpacking during our for loop. This data structure is actually very common in Python , especially when working with outside libraries. You can use the **zip()** function to quickly create a list of tuples by "zipping" up together two lists.

```
[95]: mylist1 = [1,2,3,4,5]
      mylist2 = ['a','b','c','d','e']

      # This one is also a generator! We will explain this later, but for now let's␣
       ↪transform it to a list
      zip(mylist1,mylist2)
```

```
[95]: <zip at 0x7f63344fa2d0>
```

```
[96]: # Notice the tuple unpacking!

      for i,letter in enumerate('abcde'):
          print("At index {} the letter is {}".format(i,letter))
```

```
At index 0 the letter is a
At index 1 the letter is b
At index 2 the letter is c
At index 3 the letter is d
At index 4 the letter is e
```

4.7.4 in operator

We've already seen the **in** keyword durng the for loop, but we can also use it to quickly check if an object is in a list

```
[97]: 'x' in ['x','y','z']
```

```
[97]: True
```

[98]:
```
'x' in [1,2,3]
```

[98]:
```
False
```

4.7.5 min and max

Quickly check the minimum or maximum of a list with these functions.

[99]:
```
mylist = [10,20,30,40,100]
min(mylist)
```

[99]:
```
10
```

[100]:
```
max(mylist)
```

[100]:
```
100
```

4.7.6 random

Python comes with a built in random library. There are a lot of functions included in this random library, so we will only show you two useful functions for now.

[101]:
```
from random import shuffle
```

[102]:
```
# This shuffles the list "in-place" meaning it won't return
# anything, instead it will effect the list passed
shuffle(mylist)
mylist
```

[102]:
```
[20, 30, 40, 10, 100]
```

4.7.7 input

The input() method reads a line from input, converts into a string and returns it.

The input() method takes a single optional argument: prompt (Optional) - a string that is written to standard output (usually screen) without trailing newline

[103]:
```
input('Enter Something into this box: ')
```

```
Enter Something into this box:  Hi,10
```

[103]:
```
'Hi,10'
```

4.8 List 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:

[104]:
```python
# Grab every letter in string
lst = [x for x in 'word']
lst
```

[104]:
```
['w', 'o', 'r', 'd']
```

5 Methods and Functions

5.1 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 perform specific actions on an 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.

Python Method

1. Method is called by its name, but it is associated to an object (dependent).
2. A method is implicitly passed the object on which it is invoked.
3. It may or may not return any data.
4. A method can operate on the data (instance variables) that is contained by the corresponding class

Basic Method Structure in Python : **Basic Python method**

```python
class class_name:
    def method_name():
        #......
        # method body
        #......
```

Let's take a quick look at what an example of the various methods a list has:

```
[105]: # Create a simple list
       lst = [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:

```
[106]: lst.append(6)
       lst
```

```
[106]: [1, 2, 3, 4, 5, 6]
```

```
[107]: help(lst.count)
```

```
Help on built-in function count:

count(value, /) method of builtins.list instance
    Return number of occurrences of value.
```

5.2 Functions

5.2.1 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).

- Function is block of code that is also called by its name. (independent)
- The function can have different parameters or may not have any at all. If any data (parameters) are passed, they are passed explicitly.
- It may or may not return any data.
- Function does not deal with Class and its instance concept.

5.2.2 def Statements

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

[108]:
```
def name_of_function(arg1,arg2):
    '''
    This is where the function's Document String (docstring) 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 (such as len).

Next come a pair of parentheses 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 docstring, this is where you write a basic description of the function. Using iPython and iPython Notebooks, you'll be able to read these docstrings by pressing Shift+Tab after a function name. Docstrings are not necessary for simple functions, but it's 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.

[109]:
```
def is_prime(num):
    '''
    Naive method of checking for primes.
    '''
    for n in range(2,num):
        if num % n == 0:
            print(num,'is not prime')
            break
    else: # If never mod zero, then prime
        print(num,'is prime!')
```

[110]:
```
is_prime(16)
```

```
16 is not prime
```

5.3 Lambda Expressions, Map, and Filter

Now its time to quickly learn about two built in functions, filter and map. Once we learn about how these operate, we can learn about the lambda expression, which will come in handy when you begin to develop your skills further!

5.3.1 map function

The map() function in Python takes in a function and a list. The **map** function allows you to "map" a function to an iterable object. That is to say you can quickly call the same function to every item in an iterable, such as a list. For example:

[111]:
```
def square(num):
    return num**2
my_nums = [1,2,3,4,5]
map(square,my_nums)

# To get the results, either iterate through map()
# or just cast to a list
list(map(square,my_nums))
```

[111]:
```
[1, 4, 9, 16, 25]
```

[112]:
```
def splicer(mystring):
    if len(mystring) % 2 == 0:
        return 'even'
    else:
        return mystring[0]

mynames = ['John','Cindy','Sarah','Kelly','Mike']
list(map(splicer,mynames))
```

[112]:
```
['even', 'C', 'S', 'K', 'even']
```

5.3.2 filter function

The filter function returns an iterator yielding those items of iterable for which function(item) is true. Meaning you need to filter by a function that returns either True or False. Then passing that into filter (along with your iterable) and you will get back only the results that would return True when passed to the function.

[113]:
```
def check_even(num):
    return num % 2 == 0

nums = [0,1,2,3,4,5,6,7,8,9,10]
filter(check_even,nums)
list(filter(check_even,nums))
```

[113]:
```
[0, 2, 4, 6, 8, 10]
```

5.3.3 lambda expression

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.

[114]:
```
double = lambda x: x * 2

# Output: 10
print(double(5))
```

```
10
```

5.4 Nested Statements and Scope

Now that we have gone over writing our own functions, it's 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.

Let's start with a quick thought experiment; imagine the following code:

[115]:
```python
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?

[116]:
```python
print(x)
```

```
25
```

[117]:
```python
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 — Names 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,…

5.4.1 Quick examples of LEGB

5.4.2 Local

[118]:
```
# x is local here:
f = lambda x:x**2
```

5.4.3 Enclosing function locals

This occurs when we have a function inside a function (nested functions)

[119]:
```
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!

5.4.4 Global

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

[120]:
```
print(name)
```

```
This is a global name
```

5.4.5 Built-in

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

[121]:
```
len
```

[121]:
```
<function len(obj, /)>
```

5.4.6 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:

[122]:
```
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.

5.4.7 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:

[123]:
```
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.

5.4.8 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!

5.5 Function Arguments

we can define a function that takes variable number of arguments. We can define a function using default, keyword and arbitrary arguments. `python default keyword arbitrary arguments *args **kwargs`

5.5.1 default argument

Function arguments can have default values in Python. We can provide a default value to an argument by using the assignment operator (=). Here is an example.

[124]:
```
def greet(name, msg = "Good morning!"):
    print("Hello",name + ', ' + msg)

greet("Kate")
greet("Bruce","How do you do?")
```

```
Hello Kate, Good morning!
Hello Bruce, How do you do?
```

Work with Python long enough, and eventually you will encounter `*args` and `**kwargs`. These strange terms show up as parameters in function definitions. What do they do? Let's review a simple function:

```
[125]: def myfunc(a,b):
           return sum((a,b))*.05

       myfunc(40,60)
```

```
[125]: 5.0
```

This function returns 5% of the sum of **a** and **b**. In this example, **a** and **b** are *positional* arguments; that is, 40 is assigned to **a** because it is the first argument, and 60 to **b**. Notice also that to work with multiple positional arguments in the `sum()` function we had to pass them in as a tuple.

What if we want to work with more than two numbers? One way would be to assign a *lot* of parameters, and give each one a default value.

```
[126]: def myfunc(a=0,b=0,c=0,d=0,e=0):
           return sum((a,b,c,d,e))*.05

       myfunc(40,60,20)
```

```
[126]: 6.0
```

Obviously this is not a very efficient solution, and that's where `*args` comes in.

Function Arguments ## `*args`

When a function parameter starts with an asterisk, it allows for an *arbitrary number* of arguments, and the function takes them in as a tuple of values. Rewriting the above function:

```
[127]: def myfunc(*args):
           return sum(args)*.05

       myfunc(40,60,20)
```

```
[127]: 6.0
```

Notice how passing the keyword "args" into the `sum()` function did the same thing as a tuple of arguments.

It is worth noting that the word "args" is itself arbitrary - any word will do so long as it's preceded by an asterisk. To demonstrate this:

```
[128]: def myfunc(*spam):
           return sum(spam)*.05
```

```
myfunc(40,60,20)
```

[128]:
```
6.0
```

5.5.2 **kwargs

Similarly, Python offers a way to handle arbitrary numbers of *keyworded* arguments. Instead of creating a tuple of values, `**kwargs` builds a dictionary of key/value pairs. For example:

[129]:
```
def myfunc(**kwargs):
    if 'fruit' in kwargs:
        print(f"My favorite fruit is {kwargs['fruit']}")  # review String␣
 ↪Formatting and f-strings if this syntax is unfamiliar
    else:
        print("I don't like fruit")

myfunc(fruit='pineapple')
```

```
My favorite fruit is pineapple
```

[130]:
```
myfunc()
```

```
I don't like fruit
```

5.5.3 *args and **kwargs combined

You can pass `*args` and `**kwargs` into the same function, but `*args` have to appear before `**kwargs`

[131]:
```
def myfunc(*args, **kwargs):
    if 'fruit' and 'juice' in kwargs:
        print(f"I like {' and '.join(args)} and my favorite fruit is␣
 ↪{kwargs['fruit']}")
        print(f"May I have some {kwargs['juice']} juice?")
    else:
        pass

myfunc('eggs','spam',fruit='cherries',juice='orange')
```

```
I like eggs and spam and my favorite fruit is cherries
May I have some orange juice?
```

Placing keyworded arguments ahead of positional arguments raises an exception:

[132]:
```
myfunc(fruit='cherries',juice='orange','eggs','spam')
```

```
  File "<ipython-input-132-fc6ff65addcc>", line 1
    myfunc(fruit='cherries',juice='orange','eggs','spam')
                                          ^
SyntaxError: positional argument follows keyword argument
```

As with "args", you can use any name you'd like for keyworded arguments - "kwargs" is just a popular convention.

That's it! Now you should understand how `*args` and `**kwargs` provide the flexibilty to work with arbitrary numbers of arguments!

5.6 First Class Function

The "first-class" concept only has to do with functions in programming languages. First-class functions means that the language treats functions as values – that you can assign a function into a variable, pass it around etc. It's seldom used when referring to a function, such as "a first-class function". It's much more common to say that "a language has/hasn't first-class function support".So "has first-class functions" is a property of a language.

Properties of first class functions:

- A function is an instance of the Object type.
- You can store the function in a variable.
- You can pass the function as a parameter to another function.
- You can return the function from a function.
- You can store them in data structures such as hash tables, lists etc.

So, languages which support values with function types, and treat them the same as non-function values, can be said to have "first class functions".

[133]:
```python
# first class function
def square(x):
    return x*x

def cube(x):
    return x*x*x

def my_map(func, arg_list):
    result = []
    for i in arg_list:
        result.append(func(i))
    return result

squares = my_map(cube,[1,2,3,4,5])

print(squares)
```

```
[1, 8, 27, 64, 125]
```

5.6.1 Clousers

A closure is a nested function which has access to a free variable from an enclosing function that has finished its execution. Three characteristics of a Python closure are:

- it is a nested funcion
- it has access to a free variable in outer scope
- it is returned from the enclosing function

A free variable is a variable that is not bound in the local scope. In order for closures to work with immutable variables such as numbers and strings, we have to use the nonlocal keyword.

Python closures help avoiding the usage of global values and provide some form of data hiding. They are used in Python decorators.

[134]:
```python
def outer_func():
    message = 'Hi'

    def innner_func():
        print(message)
    return innner_func()

outer_func()
outer_func()
outer_func()
```

```
Hi
Hi
Hi
```

[135]:
```python
def outer_func(msg):
    message = msg

    def innner_func():
        print(message)
    return innner_func

hi_func = outer_func('Hi')
bye_func = outer_func('Bye')

hi_func()
bye_func()
```

```
Hi
Bye
```

[136]:
```python
def make_counter():

    count = 0
    def inner():

        nonlocal count
        count += 1
        return count

    return inner


counter = make_counter()

c = counter()
print(c)

c = counter()
print(c)

c = counter()
print(c)
```

```
1
2
3
```

5.6.2 Functions within functions

Great! So we've seen how we can treat functions as objects, now let's see how we can define functions inside of other functions:

[137]:
```python
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")
```

[138]:
```python
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
```

[139]:
```python
welcome()
```

```
      ␣
↪---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call␣
↪last)

        <ipython-input-139-a401d7101853> 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.

5.6.3 Returning Functions

Now lets learn about returning functions from within functions:

[165]:
```python
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
```

Now let's see what function is returned if we set x = hello(), note how the empty parentheses means that name has been defined as Jose.

[166]:
```python
x = hello()
```

[167]:
```python
x
```

[167]:
```
<function __main__.hello.<locals>.greet()>
```

Great! Now we can see how x is pointing to the greet function inside of the hello function.

```
[168]: print(x())
```

```
        This is inside the greet() function
```

Let's 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 parentheses 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 *This is inside the greet() function.*

5.6.4 Functions as Arguments

Now let's see how we can pass functions as arguments into other functions:

```
[169]: def hello():
           return 'Hi Jose!'

       def other(func):
           print('Other code would go here')
           print(func())
```

```
[170]: 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:

6 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 Polymorphism
- Learning about Special Methods for classes

What Is **Object-Oriented Programming** (OOP)?

Object-oriented Programming, or OOP for short, is a programming paradigm which provides a means of structuring programs so that properties and behaviors are bundled into individual objects.

Python is a multi-paradigm programming language. Meaning, it supports different programming approach. One of the popular approach to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

```
attributes
behavior
```

Lets start the lesson by remembering about the Basic Python Objects. For example:

[171]:
```
lst = [1,2,3]
```

Remember how we could call methods on a list?

[172]:
```
lst.count(2)
```

[172]:
```
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 let's explore Objects in general:

6.0.1 Objects

An **object** is an instance of a class. A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.

In Python, *everything is an object.* Remember from previous lectures we can use type() to check the type of object something is:

[173]:
```
print(type(1))
print(type([]))
print(type(()))
print(type({}))
```

```
<class 'int'>
<class 'list'>
```

```
<class 'tuple'>
<class '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.

6.0.2 Class

A class is a blueprint for the object. User defined objects are created using the class keyword. Classes are used to create new user-defined data structures that contain arbitrary information about something. The class is a blueprint that defines the nature of a future object. From classes we can construct instances. An instance is a specific object created from a particular class. In the case of an animal, we could create an Animal() class to track properties about the Animal like the name and age. For example, above we created the object lst which was an instance of a list object.

It's important to note that a class just provides structure—it's a blueprint for how something should be defined, but it doesn't actually provide any real content itself. Let see how we can use class:

[174]:
```python
# Create a new object type called Sample
class Sample:
    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.

6.0.3 Constructors in Python

The **constructor** method is used to initialize data. It is run as soon as an object of a class is instantiated. Also known as the **init** method, it will be the first definition of a class.

Class functions that begins with double underscore **(___)** are called **special functions** as they have special meaning.Of one particular interest is the **init()** function. This special function gets called whenever a new object of that class is instantiated. This type of function is also called **constructors** in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

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. 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:

[175]:
```
class Dog:
    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:

[176]:
```
sam.breed
```

[176]:
```
'Lab'
```

[177]:
```
frank.breed
```

[177]:
```
'Huskie'
```

Note how we don't have any parentheses 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:

[178]:
```
class Dog:

    # Class Object Attribute
    species = 'mammal'
    def __init__(self,breed,name):
        self.breed = breed
        self.name = name
```

[179]:
```
sam = Dog('Lab','Sam')
```

[180]:
```
sam.name
```

[180]:
```
'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.

[181]:
```
sam.species
```

[181]:
```
'mammal'
```

6.0.4 Class Variables

Class variables are defined within the class construction. Because they are owned by the class itself, class variables are shared by all instances of the class. They therefore will generally have the same value for every instance unless you are using the class variable to initialize a variable.

Defined outside of all the methods, class variables are, by convention, typically placed right below the class header and before the constructor method and other methods.

[182]:
```
class Employee():

    raise_amount =1.04
    def __init__(self,first,last,pay):
        self.first  = first
        self.last = last
        self.pay = pay
        self.email = first+'.'+last+'@company.com'

    def fullname(self):
        return self.first+' '+self.last

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amount)
        return self.pay
```

[183]:
```
emp_1 = Employee('Sudhir','Kumar',200)
emp_2 = Employee('Latha','Shet',200)

print(emp_1.__dict__)
print(emp_2.fullname())
print(emp_1.apply_raise())
```

```
{'first': 'Sudhir', 'last': 'Kumar', 'pay': 200, 'email':
'Sudhir.Kumar@company.com'}
Latha Shet
208
```

6.0.5 Instance Variables

Instance variables are owned by instances of the class. This means that for each object or instance of a class, the instance variables are different.

Unlike class variables, instance variables are defined within methods.

[184]:
```
print(emp_1.first)
print(emp_1.email)
```

```
Sudhir
Sudhir.Kumar@company.com
```

6.1 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 a key concept of the OOP paradigm. They are essential to 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.

Let's go through an example of creating a Circle class:

[185]:
```
class Circle:
    pi = 3.14

    # Circle gets instantiated with a radius (default is 1)
    def __init__(self, radius=1):
        self.radius = radius
        self.area = radius * radius * Circle.pi

    # Method for resetting Radius
    def setRadius(self, new_radius):
        self.radius = new_radius
        self.area = new_radius * new_radius * self.pi

    # Method for getting Circumference
    def getCircumference(self):
        return self.radius * self.pi * 2
```

```
c = Circle()

print('Radius is: ',c.radius)
print('Area is: ',c.area)
print('Circumference is: ',c.getCircumference())
```

```
Radius is:  1
Area is:  3.14
Circumference is:  6.28
```

In the ___init___ method above, in order to calculate the area attribute, we had to call Circle.pi. This is because the object does not yet have its own .pi attribute, so we call the Class Object Attribute pi instead. In the setRadius method, however, we'll be working with an existing Circle object that does have its own pi attribute. Here we can use either Circle.pi or self.pi. Now let's change the radius and see how that affects our Circle object:

[186]:
```
c.setRadius(2)

print('Radius is: ',c.radius)
print('Area is: ',c.area)
print('Circumference is: ',c.getCircumference())
```

```
Radius is:  2
Area is:  12.56
Circumference 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.

6.2 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).

Inheritance is when a class uses code constructed within another class. If we think of inheritance in terms of biology, we can think of a child inheriting certain traits from their parent. That is, a child can inherit a parent's height or eye color. Children also may share the same last name with their parents. Classes called child classes or subclasses inherit methods and variables from parent classes or base classes. Let's see an example by incorporating our previous work on the Dog class:

[187]:
```
class Animal:
    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!")
```

[188]:
```
d = Dog()
```

```
Animal created
Dog created
```

[189]:
```
d.whoAmI()
```

```
Dog
```

[190]:
```
d.eat()
```

```
Eating
```

[191]:
```
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.

The **benefits** of inheritance are:

1. It represents real-world relationships well.

2. It provides reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
3. It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

6.3 Polymorphism

We've learned that while functions can take in different arguments, methods belong to the objects they act on. In Python, *polymorphism* refers to the way in which different object classes can share the same method name, and those methods can be called from the same place even though a variety of different objects might be passed in.

Polymorphism is an important feature of class definition in Python that is utilized when you have commonly named methods across classes or subclasses. This allows functions to use objects of any of these polymorphic classes without needing to be aware of distinctions across the classes.

Polymorphism can be carried out through inheritance, with subclasses making use of base class methods or overriding them. The best way to explain this is by example:

[192]:
```python
class Dog:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return self.name+' says Woof!'

class Cat:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return self.name+' says Meow!'

niko = Dog('Niko')
felix = Cat('Felix')

print(niko.speak())
print(felix.speak())
```

```
Niko says Woof!
Felix says Meow!
```

Here we have a Dog class and a Cat class, and each has a `.speak()` method. When called, each object's `.speak()` method returns a result unique to the object.

There a few different ways to demonstrate polymorphism. First, with a for loop:

[193]:
```python
for pet in [niko,felix]:
    print(pet.speak())
```

```
Niko says Woof!
Felix says Meow!
```

Another is with functions:

[194]:
```python
def pet_speak(pet):
    print(pet.speak())

pet_speak(niko)
pet_speak(felix)
```

```
Niko says Woof!
Felix says Meow!
```

In both cases we were able to pass in different object types, and we obtained object-specific results from the same mechanism.

A more common practice is to use abstract classes and inheritance. An abstract class is one that never expects to be instantiated. For example, we will never have an Animal object, only Dog and Cat objects, although Dogs and Cats are derived from Animals:

[195]:
```python
class Animal:
    def __init__(self, name):    # Constructor of the class
        self.name = name

    def speak(self):              # Abstract method, defined by convention only
        raise NotImplementedError("Subclass must implement abstract method")


class Dog(Animal):

    def speak(self):
        return self.name+' says Woof!'

class Cat(Animal):

    def speak(self):
        return self.name+' says Meow!'

fido = Dog('Fido')
isis = Cat('Isis')

print(fido.speak())
print(isis.speak())
```

```
Fido says Woof!
Isis says Meow!
```

Real life examples of polymorphism include: * opening different file types - different tools are needed to display Word, pdf and Excel files * adding different objects - the + operator performs arithmetic and concatenation

6.4 Encapsulation of Data

Another important advantage of OOP consists in the encapsulation of data. We can say that object-oriented programming relies heavily on encapsulation. The terms encapsulation and abstraction (also data hiding) are often used as synonyms. They are nearly synonymous, i.e. abstraction is achieved though encapsulation. Data hiding and encapsulation are the same concept, so it's correct to use them as synonyms. Generally speaking encapsulation is the mechanism for restricting the access to some of an object's components, this means that the internal representation of an object can't be seen from outside of the objects definition. Access to this data is typically only achieved through special methods: Getters and Setters. By using solely get() and set() methods, we can make sure that the internal data cannot be accidentally set into an inconsistent or invalid state.

Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation. In Python, we denote private attribute using underscore as prefix i.e single " _ " or double " __".

[196]:
```
class Computer:

    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))

    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()
```

```
Selling Price: 900
```

[197]:
```
# change the price
c.__maxprice = 1000
c.sell()
```

```
Selling Price: 900
```

[198]:
```
# using setter function
c.setMaxPrice(1000)
c.sell()
```

```
Selling Price: 1000
```

6.5 Special Methods

Finally let's 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 let's create a Book class:

[199]:
```
class Book:
    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")
```

[200]:
```
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 __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

Mozilla's Post

Tutorial's Point

Official Documentation

Special Method

6.5.1 class method

A class method is a method that is bound to a class rather than its object. It doesn't require creation of a class instance, much like staticmethod.

The difference between a static method and a class method is: * Static method knows nothing about the class and just deals with the parameters * Class method works with the class since its parameter is always the class itself.

The class method can be called both by the class and its object.

```
Class.classmethod()
Or even
Class().classmethod()
```

```
@classmethod
def set_raise_amount(cls, amount):
    cls.raise_amount = amount
```

[201]:
```
class Employee():

    raise_amount =1.04
    def __init__(self,first,last,pay):
        self.first  = first
        self.last = last
        self.pay = pay
        self.email = first+'.'+last+'@company.com'

    def fullname(self):
        return self.first+' '+self.last

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amount)
        return self.pay

    @classmethod
    def set_raise_amount(cls, amount):
        cls.raise_amount = amount

    @classmethod
    def from_string(cls,emp_str):
        " class method used as constructor"
        first,last,pay = emp_str.split('-')
        return cls(first,last,pay)
    @staticmethod
    def is_workday(day):
        if day.weekday() ==5 or day.weekday() ==6:
            return False
        return True
```

[202]:
```
emp_1 = Employee('Sudhir','Kumar',2000)
emp_2 = Employee('Latha','Shet',4000)

print(emp_1.raise_amount)
Employee.set_raise_amount(1.1)
print(emp_1.raise_amount)
```

```
1.04
1.1
```

class method can be used as alternative constructor. we use class method in order to provide multiple ways of creating our objects.

```
@classmethod
def from_string(cls,emp_str):
    " class method used as constructor"
    first,last,pay = emp_str.split('-')
    return cls(first,last,pay)
```

[203]:
```
# class method used as constructor
emp_str_1 = 'John-Doe-2000'
new_emp = Employee.from_string(emp_str_1)
print(new_emp.fullname())
```

```
John Doe
```

6.5.2 Static method

Static methods, much like class methods, are methods that are bound to a class rather than its object. They do not require a class instance creation. So, they are not dependent on the state of the object.

The difference between a static method and a class method is: * Static method knows nothing about the class and just deals with the parameters. * Class method works with the class since its parameter is always the class itself.

They can be called both by the class and its object.

```
Class.staticmethodFunc()
or even
Class().staticmethodFunc()
```

```
@staticmethod
```

```
def is_workday(day):
    if day.weekday() ==5 or day.weekday() ==6:
        return False
    return True
```

[204]:
```
import datetime
my_day = datetime.date(2018,11,21)
Employee.is_workday(my_day)
```

[204]:
```
True
```

6.5.3 if name == 'main'

stackoverflow

Whenever the Python interpreter reads a source file, it does two things:

- it sets a few special variables like **name**, and then
- it executes all of the code found in the file.

Let's see how this works and how it relates to your question about the **name** checks we always see in Python scripts.

7 Code Sample

Let's use a slightly different code sample to explore how imports and scripts work. Suppose the following is in a file called foo.py.

```
# Suppose this is foo.py.

print("before import")
import math

print("before functionA")
def functionA():
    print("Function A")

print("before functionB")
def functionB():
    print("Function B {}".format(math.sqrt(100)))

print("before __name__ guard")
if __name__ == '__main__':
    functionA()
    functionB()
print("after __name__ guard")
```

7.0.1 Special Variables

When the Python interpeter reads a source file, it first defines a few special variables. In this case, we care about the **name** variable.

7.1 When Your Module Is the Main Program

If you are running your module (the source file) as the main program, e.g.

```
python foo.py
```

the interpreter will assign the hard-coded string "**main**" to the **name** variable, i.e.

```
# It's as if the interpreter inserts this at the top
# of your module when run as the main program.
__name__ = "__main__"
```

7.1.1 When Your Module Is Imported By Another

On the other hand, suppose some other module is the main program and it imports your module. This means there's a statement like this in the main program, or in some other module the main program imports:

```
# Suppose this is in some other main program.
import foo
```

The interpreter will search for your foo.py file (along with searching for a few other variants), and prior to executing that module, it will assign the name "foo" from the import statement to the **name** variable, i.e.

```
# It's as if the interpreter inserts this at the top
# of your module when it's imported from another module.
__name__ = "foo"
```

7.1.2 Executing the Module's Code

After the special variables are set up, the interpreter executes all the code in the module, one statement at a time. You may want to open another window on the side with the code sample so you can follow along with this explanation.

Always

1. It prints the string "before import" (without quotes).
2. It loads the math module and assigns it to a variable called math. This is equivalent to replacing import math with the following (note that **import** is a low-level function in Python that takes a string and triggers the actual import):

```
# Find and load a module given its string name, "math",
# then assign it to a local variable called math.
math = __import__("math")
```

3. It prints the string "before functionA".
4. It executes the def block, creating a function object, then assigning that function object to a variable called functionA.
5. It prints the string "before functionB".
6. It executes the second def block, creating another function object, then assigning it to a variable called functionB.
7. It prints the string "before **name** guard".

Only When Your Module Is the Main Program

8. If your module is the main program, then it will see that **name** was indeed set to "**main**" and it calls the two functions, printing the strings "Function A" and "Function B 10.0".

Only When Your Module Is Imported by Another

8. (instead) If your module is not the main program but was imported by another one, then **name** will be "foo", not "**main**", and it'll skip the body of the if statement.

Always

9. It will print the string "after **name** guard" in both situations.

7.1.3 Why Does It Work This Way?

You might naturally wonder why anybody would want this. Well, sometimes you want to write a .py file that can be both used by other programs and/or modules as a module, and can also be run as the main program itself. Examples:

- Your module is a library, but you want to have a script mode where it runs some unit tests or a demo.
- Your module is only used as a main program, but it has some unit tests, and the testing framework works by importing .py files like your script and running special test functions. You don't want it to try running the script just because it's importing the module.
- Your module is mostly used as a main program, but it also provides a programmer-friendly API for advanced users.

Beyond those examples, it's elegant that running a script in Python is just setting up a few magic variables and importing the script. "Running" the script is a side effect of importing the script's module.

[205]:
```python
# Suppose this is foo.py.

print("before import")
import math

print("before functionA")
def functionA():
    print("Function A")

print("before functionB")
def functionB():
    print("Function B {}".format(math.sqrt(100)))

print("before __name__ guard")
if __name__ == '__main__':
    functionA()
    functionB()
print("after __name__ guard")
```

```
before import
before functionA
before functionB
before __name__ guard
Function A
Function B 10.0
after __name__ guard
```

7.2 Modules and Packages

In this section we briefly: * code out a basic module and show how to import it into a Python script * run a Python script from a Jupyter cell * show how command line arguments can be passed into a script

Check out the video lectures for more info and resources for this.

The best online resource is the official docs: https://docs.python.org/3/tutorial/modules.html#packages

But I really like the info here: https://python4astronomers.github.io/installation/packages.html

7.2.1 Writing modules

[206]:
```
%%writefile file1.py
def myfunc(x):
    return [num for num in range(x) if num%2==0]
list1 = myfunc(11)
```

```
Overwriting file1.py
```

file1.py is going to be used as a module.

Note that it doesn't print or return anything, it just defines a function called *myfunc* and a variable called *list1*. ### Writing scripts

[207]:
```
%%writefile file2.py
import file1
file1.list1.append(12)
print(file1.list1)
```

```
Overwriting file2.py
```

file2.py is a Python script.

First, we import our **file1** module (note the lack of a .py extension) Next, we access the *list1* variable inside **file1**, and perform a list method on it. `.append(12)` proves we're working with a Python list object, and not just a string. Finally, we tell our script to print the modified list. ### Running scripts

[208]:
```
! python file2.py
```

```
[0, 2, 4, 6, 8, 10, 12]
```

Here we run our script from the command line. The exclamation point is a Jupyter trick that lets you run command line statements from inside a jupyter cell.

[209]:
```
import file1
print(file1.list1)
```

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

The above cell proves that we never altered **file1.py**, we just appended a number to the list *after* it was brought into **file2**.

7.2.2 Passing command line arguments

Python's `sys` module gives you access to command line arguments when calling scripts.

[210]:
```
%%writefile file3.py
import sys
import file1
num = int(sys.argv[1])
print(file1.myfunc(num))
```

```
Overwriting file3.py
```

Note that we selected the second item in the list of arguments with `sys.argv[1]`. This is because the list created with `sys.argv` always starts with the name of the file being used.

[211]:
```
! python file3.py 21
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Here we're passing 21 to be the upper range value used by the *myfunc* function in **list1.py**

7.2.3 Understanding modules

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 the math module, we simply import the name of the module:

[212]:
```python
# import the library
import math
```

[213]:
```python
# use it (ceiling rounding)
math.ceil(2.4)
```

[213]:
```
3
```

7.2.4 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:

[214]:
```python
print(dir(math))
```

```
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

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:

[215]:
```python
help(math.ceil)
```

```
Help on built-in function ceil in module math:

ceil(x, /)
    Return the ceiling of x as an Integral.

    This is the smallest integer >= x.
```

7.2.5 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.

7.2.6 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:

[218]:
```
ls
```

```
cap.py*       MyMainPackage/
__pycache__/                  test_cap.py*
dfdf.txt*     mymodule.py*
Python3_Advanced_pr.ipynb*  testfile*
file1.py      my_new_file.txt*  Python3_Assignment.ipynb*
test.txt*
file2.py      myprogramm.py*    Python3_Notes.ipynb*
two.py*
file3.py      newmyfile.txt*    Python3_Practice.ipynb*
image/        notebook.tex*     readme.md*
myfile.txt*   one.py*           simple1.py*
```

```
Just an example, this won't work
import foo.bar

# 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:

```
__init__.py:

__all__ = ["bar"]
```

7.3 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:

[221]:
```
print('Hello')
```

```
Hello
```

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. Now let's learn how to handle errors and exceptions in our own code.

7.3.1 try and except

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

```
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 let's check out an example: We will look at some code that opens and writes a file:

[222]:
```
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 let's see what would happen if we did not have write permission (opening only with 'r'):

[223]:
```python
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:

[224]:
```python
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:

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

```
Always execute finally code blocks
```

We can use this in conjunction with except. Let's see a new example that will take into account a user providing the wrong input:

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

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

```
[227]: askint()
```

```
Please enter an integer:  1

Finally, I executed!
1
```

```
[228]: askint()
```

```
Please enter an integer:  '2'

Looks like you did not enter an integer!
Finally, I executed!


          ␣
 ↪---------------------------------------------------------------------------


        UnboundLocalError                         Traceback (most recent call␣
 ↪last)


        <ipython-input-228-cc291aa76c10> in <module>
    ----> 1 askint()



        <ipython-input-226-c97dd1c75d24> 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). Let's remedy this by asking the user and checking to make sure the input type is an integer:

[229]:
```
def askint():
    try:
        val = int(input("Please enter an integer: "))
    except:
        print("Looks like you did not enter an integer!")
        val = int(input("Try again-Please enter an integer: "))
    finally:
        print("Finally, I executed!")
    print(val)
```

[230]:
```
askint()
```

```
Please enter an integer:  uowq

Looks like you did not enter an integer!

Try again-Please enter an integer:  10

Finally, I executed!
10
```

Hmmm…that only did one check. How can we continually keep checking? We can use a while loop!

[231]:
```
def askint():
    while True:
        try:
            val = int(input("Please enter an integer: "))
        except:
            print("Looks like you did not enter an integer!")
            continue
        else:
            print("Yep that's an integer!")
            break
        finally:
            print("Finally, I executed!")
        print(val)
```

[232]:
```
askint()
```

```
Please enter an integer:  'asl'
```

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

Please enter an integer:  od

Looks like you did not enter an integer!
Finally, I executed!

Please enter an integer:  8

Yep that's an integer!
Finally, I executed!
```

So why did our function print "Finally, I executed!" after each trial, yet it never printed `val` itself? This is because with a try/except/finally clause, any continue or break statements are reserved until *after* the try clause is completed. This means that even though a successful input of **3** brought us to the else: block, and a break statement was thrown, the try clause continued through to finally: before breaking out of the while loop. And since print(val) was outside the try clause, the break statement prevented it from running.

Let's make one final adjustment:

[233]:
```
def askint():
    while True:
        try:
            val = int(input("Please enter an integer: "))
        except:
            print("Looks like you did not enter an integer!")
            continue
        else:
            print("Yep that's an integer!")
            print(val)
            break
        finally:
            print("Finally, I executed!")
```

[234]:
```
askint()
```

```
Please enter an integer:  1

Yep that's an integer!
1
Finally, I executed!
```

Great! Now you know how to handle errors and exceptions in Python with the try, except, else, and finally notation!

7.4 Built in function

7.4.1 map()

map() is a built-in Python function that takes in two or more arguments: a function and one or more iterables, in the form:

`map(function, iterable, ...)`

map() returns an *iterator* - that is, map() returns a special object that yields one result at a time as needed. We will learn more about iterators and generators in a future lecture. For now, since our examples are so small, we will cast map() as a list to see the results immediately.

When we went over list comprehensions we created a small expression to convert Celsius to Fahrenheit. Let's do the same here but use map:

[235]:
```
def fahrenheit(celsius):
    return (9/5)*celsius + 32

temps = [0, 22.5, 40, 100]
```

Now let's see map() in action:

[236]:
```
F_temps = map(fahrenheit, temps)

#Show
list(F_temps)
```

[236]:
```
[32.0, 72.5, 104.0, 212.0]
```

7.4.2 map() with multiple iterables

map() can accept more than one iterable. The iterables should be the same length - in the event that they are not, map() will stop as soon as the shortest iterable is exhausted.

For instance, if our function is trying to add two values **x** and **y**, we can pass a list of **x** values and another list of **y** values to map(). The function (or lambda) will be fed the 0th index from each list, and then the 1st index, and so on until the n-th index is reached.

Let's see this in action with two and then three lists:

[237]:
```
a = [1,2,3,4]
b = [5,6,7,8]
c = [9,10,11,12]

list(map(lambda x,y:x+y,a,b))
```

[237]:
```
[6, 8, 10, 12]
```

[238]:
```
# Now all three lists
list(map(lambda x,y,z:x+y+z,a,b,c))
```

[238]:
```
[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.

7.4.3 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()

Let's see an example:

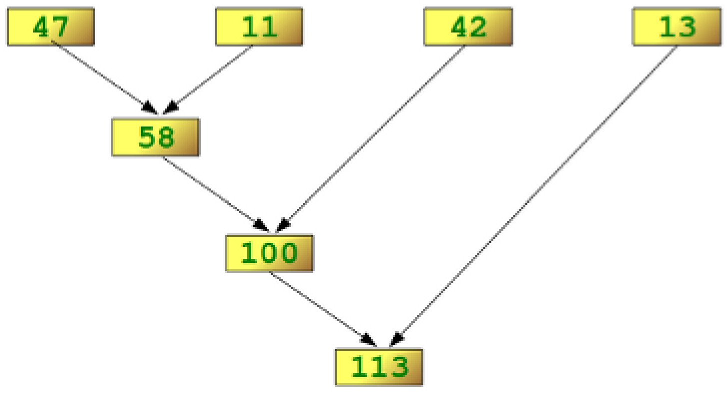
[239]:
```
from functools import reduce

lst =[47,11,42,13]
reduce(lambda x,y: x+y,lst)
```

[239]:
```
113
```

Lets look at a diagram to get a better understanding of what is going on here:

[240]:
```
from IPython.display import Image
Image('http://www.python-course.eu/images/reduce_diagram.png')
```

[240]:



Note how we keep reducing the sequence until a single final value is obtained. Lets see another example:

[241]:
```
#Find the maximum of a sequence (This already exists as max())
max_find = lambda a,b: a if (a > b) else b
```

[242]:
```
#Find max
reduce(max_find,lst)
```

[242]:
```
47
```

Hopefully you can see how useful reduce can be in various situations. Keep it in mind as you think about your code projects!

7.4.4 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,list) 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.

Like map(), filter() returns an *iterator* - that is, filter yields one result at a time as needed. Iterators and generators will be covered in an upcoming lecture. For now, since our examples are so small, we will cast filter() as a list to see our results immediately.

Let's see some examples:

[243]:
```
#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 parentheses might feel strange, but keep in mind that functions are objects as well.

[244]:
```
lst =range(20)

list(filter(even_check,lst))
```

[244]:
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

filter() is more commonly used with lambda functions, because we usually use filter for a quick job where we don't want to write an entire function. Let's repeat the example above using a lambda expression:

[245]:
```
list(filter(lambda x: x%2==0,lst))
```

[245]:
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

7.4.5 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.

Let's see it in action in some examples:

[246]:
```python
x = [1,2,3]
y = [4,5,6]

# Zip the lists together
list(zip(x,y))
```

[246]:
```
[(1, 4), (2, 5), (3, 6)]
```

7.4.6 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
```

enumerate() becomes particularly useful when you have a case where you need to have some sort of tracker. For example:

[247]:
```python
lst = ['a','b','c']
for count,item in enumerate(lst):
    if count >= 2:
        break
    else:
        print(item)
```

```
a
b
```

enumerate() takes an optional ``start'' argument to override the default value of zero:

[248]:
```python
months = ['March','April','May','June']

list(enumerate(months,start=3))
```

[248]:
```
[(3, 'March'), (4, 'April'), (5, 'May'), (6, 'June')]
```

7.4.7 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:

```
[249]: lst = [True,True,False,True]
```

```
[250]: all(lst)
```

```
[250]: False
```

Returns False because not all elements are True.

```
[251]: any(lst)
```

```
[251]: True
```

Returns True because at least one of the elements in the list is True

7.4.8 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.

Let's see some examples:

[252]:
```
# Create 2+3j
complex(2,3)
```

[252]:
```
(2+3j)
```

We can also pass strings:

[253]:
```
complex('12+2j')
```

[253]:
```
(12+2j)
```

8 Decorators

Decorators are very powerful and useful tool in Python since it allows programmers to modify the behavior of function,methods or class. Decorators allow us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it.

Decorators can be thought of as functions which modify the *functionality* of another function. They help to make your code shorter and more ``Pythonic''.

In Python, functions are the first class objects, which means that --

- Functions are objects; they can be referenced to, passed to a variable and returned from other functions as well.
- Functions can be defined inside another function and can also be passed as argument to another function.

To properly explain decorators we will slowly build up from functions. Make sure to run every cell in this Notebook for this lecture to look the same on your own computer.So let's break down the steps:

8.0.1 Functions Review

[254]:
```
def func():
    return 1
func()
```

[254]:
```
1
```

Scope Review

Remember from the nested statements lecture that Python uses Scope to know what a label is referring to. For example:

[255]:
```
s = 'Global Variable'

def check_for_locals():
    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 locals() and globals() functions. For example:

[272]:
```
print(globals())
```

```
{'__name__': '__main__', '__doc__': 'Automatically created module for IPython
interactive environment', '__package__': None, '__loader__': None, '__spec__':
None, '__builtin__': <module 'builtins' (built-in)>, '__builtins__': <module
'builtins' (built-in)>, '_ih': ['', '# Addition\n2+1', '# Subtraction\n2-1', '#
Multiplication\n2*2', '# Division\n3/2', '# Floor Division\n7//4', '# Mod \n10 %
4', '# Powers\n2**3', '# Let\'s create an object called "a" and assign it the
number 5\na = 5', '# Adding the objects\na+a', "# Use object names to keep
better track of what's going on in your code!\nmy_income = 100\n\ntax_rate =
0.1\n\nmy_taxes = my_income*tax_rate", 'a = b = c = 1', 'my_dogs = 2',
'my_dogs', "my_dogs = ['Sammy', 'Frankie']", 'my_dogs', "# Single
word\n'hello'", "# We can simply declare a string\n'Hello World'", "len('Hello
World')", "# Assign s as a string\ns = 'Hello World'", '# Show first element (in
this case a letter)\ns[0]', '# Grab everything but the last letter\ns[:-1]', '#
Grab everything, but go in steps size of 1\ns[::1]', '# Grab everything, but go
in step sizes of 2\ns[::2]', '# We can use this to print a string
backwards\ns[::-1]', "# Let's try to change the first letter to 'x'\n#s[0] =
'x'", "# Concatenate strings!\ns + ' concatenate me!'", '# Upper Case a
string\ns.upper()', '# Lower case\ns.lower()', '# Split a string by blank space
(this is the default)\ns.split()', "# Split by a specific element (doesn't
include the element that was split on)\ns.split('W')", "'Insert another string
with curly brackets: {}'.format('The inserted string')", 'print("I\'m going to
inject %s text here, and %s text here." %(\'some\',\'more\'))', 'x, y =
\'some\', \'more\'\nprint("I\'m going to inject %s text here, and %s text
here."%(x,y))', "print('I once caught a fish %s.' %'this \\tbig')\nprint('I once
caught a fish %r.' %'this \\tbig')", "print('I wrote %s programs today.'
%3.75)\nprint('I wrote %d programs today.' %3.75)   ", "print('Floating point
numbers: %5.2f' %(13.144))", "print('Floating point numbers: %25.2f'
%(13.144))", "print('{0:8} | {1:9}'.format('Fruit', 'Quantity'))\nprint('{0:8} |
{1:9}'.format('Apples', 3.))\nprint('{0:8} | {1:9}'.format('Oranges', 10))",
'name = \'Fred\'\n\nprint(f"He said his name is {name}.")', 'print(f"He said his
name is {name!r}")', '# Assign a list to an variable named my_list\nmy_list =
[1,2,3]', "my_list = ['A string',23,100.232,'o']", 'len(my_list)', "my_list =
['one','two','three',4,5]", '# Grab element at index 0\nmy_list[0]', "my_list +
['new item']", '# Make the list double\nmy_list * 2', '# Create a new
list\nlist1 = [1,2,3]\nlist1', "# Append\nlist1.append('append me!')\nlist1", '#
Pop off the 0 indexed item\nlist1.pop(0)', '# Show\nlist1', "# Let's make three
lists\nlst_1=[1,2,3]\nlst_2=[4,5,6]\nlst_3=[7,8,9]\n\n# Make a list of lists to
form a matrix\nmatrix = [lst_1,lst_2,lst_3]", '# Show\nmatrix', '# Build a list
comprehension by deconstructing a for loop within a []\nfirst_col = [row[0] for
row in matrix]', 'first_col', "# Make a dictionary with {} and : to signify a
key and a value\nmy_dict = {'key1':'value1','key2':'value2'}\n# Call values by
their key\nmy_dict['key2']", "# Dictionary nested inside a dictionary nested
inside a dictionary\nd = {'key1':{'nestkey':{'subnestkey':'value'}}}", "# Keep
calling the keys\nd['key1']['nestkey']['subnestkey']", "# Create a typical
dictionary\nd = {'key1':1,'key2':2,'key3':3}", '# Method to return a list of all
keys \nd.keys()', '# Method to grab all values\nd.values()', "# Method to return
tuples of all items  (we'll learn about tuples soon)\nd.items()", '# Create a
tuple\nt = (1,2,3)\n# Check len just like a list\nlen(t)', "# Can also mix
object types\nt = ('one',2)\n\n# Show\nt", "# Use .index to enter a value and
return the index\nt.index('one')", "# Use .count to count the number of times a
value appears\nt.count('one')", "t[0]= 'change'",
"get_ipython().run_cell_magic('writefile', 'test.txt', 'Hello, this is a quick
test file.\\n')", "myfile = open('whoops.txt')", '# Read the
file\nmy_file.seek(0)\nmy_file.read()', '2 == 2', '2 <= 2', '2 <= 4', '1 < 2 <
3', '1<2 and 2<3', "if True:\n    print('It was true!')", "x = False\n\nif x:\n
print('x was True!')\nelse:\n    print('I will be printed in any case where x is
not true')", "# We'll learn how to automate this sort of list in the next
lecture\nlist1 = [1,2,3,4,5,6,7,8,9,10]\n\nfor num in list1:\n    print(num)",
'tup = (1,2,3,4,5)\n\nfor t in tup:\n    print(t)', 'list2 =
[(2,4),(6,8),(10,12)]', 'for tup in list2:\n    print(tup)', '# Now with
unpacking!\nfor (t1,t2) in list2:\n    print(t1)', "d = {'k1':1,'k2':2,'k3':3}",
'for item in d:\n    print(item)', "x = 0\n\nwhile x < 10:\n    print('x is
currently: ',x)\n    print(' x is still less than 10, adding 1 to x')\n
x+=1", "x = 0\n\nwhile x < 10:\n    print('x is currently: ',x)\n    print(' x
is still less than 10, adding 1 to x')\n    x+=1\n    \nelse:\n    print('All
Done!')", "x = 0\n\nwhile x < 10:\n    print('x is currently: ',x)\n    print('
x is still less than 10, adding 1 to x')\n    x+=1\n    if x==3:\n
print('Breaking because x==3')\n        break\n    else:\n
print('continuing…')\n        continue", 'range(0,11)', '# Notice how 11 is
not included, up to but not including 11, just like slice
notation!\nlist(range(0,11))', '# odd --------- even\nlist(range(1,10,2)),
list(range(0,10,2))', 'list(range(10,-10,-1))', "x = ['a','b','c','d']\na=
enumerate(x, start = 10)\nlist(a)", 'index_count = 0\n\nfor letter in
\'abcde\':\n    print("At index {} the letter is
{}".format(index_count,letter))\n    index_count += 1',
"list(enumerate('abcde'))", "mylist1 = [1,2,3,4,5]\nmylist2 =
['a','b','c','d','e']\n\n# This one is also a generator! We will explain this
later, but for now let's transform it to a list\nzip(mylist1,mylist2)", '#
Notice the tuple unpacking!\n\nfor i,letter in enumerate(\'abcde\'):\n
print("At index {} the letter is {}".format(i,letter))', "'x' in ['x','y','z']",
"'x' in [1,2,3]", 'mylist = [10,20,30,40,100]\nmin(mylist)', 'max(mylist)',
'from random import shuffle', '# This shuffles the list "in-place" meaning it
won\'t return\n# anything, instead it will effect the list
passed\nshuffle(mylist)\nmylist', "input('Enter Something into this box: ')", "#
Grab every letter in string\nlst = [x for x in 'word']\nlst", '# Create a simple
list\nlst = [1,2,3,4,5]', 'lst.append(6)\nlst', 'help(lst.count)', "def
name_of_function(arg1,arg2):\n    '''\n    This is where the function's Document
```

```
String (docstring) goes\n    '''\n    # Do stuff here\n    # Return desired
result", "def is_prime(num):\n    '''\n    Naive method of checking for primes.
\n    '''\n    for n in range(2,num):\n        if num % n == 0:\n
print(num,'is not prime')\n            break\n    else: # If never mod zero,
then prime\n        print(num,'is prime!')", 'is_prime(16)', 'def square(num):\n
return num**2\nmy_nums = [1,2,3,4,5]\nmap(square,my_nums)\n\n# To get the
results, either iterate through map() \n# or just cast to a
list\nlist(map(square,my_nums))', "def splicer(mystring):\n    if len(mystring)
% 2 == 0:\n        return 'even'\n    else:\n        return mystring[0]\n
\nmynames =
['John','Cindy','Sarah','Kelly','Mike']\nlist(map(splicer,mynames))", 'def
check_even(num):\n    return num % 2 == 0 \n\nnums = [0,1,2,3,4,5,6,7,8,9,10]\nf
ilter(check_even,nums)\nlist(filter(check_even,nums))', 'double = lambda x: x *
2\n\n# Output: 10\nprint(double(5))', 'x = 25\n\ndef printer():\n    x = 50\n
return x\n\n# print(x)\n# print(printer())', 'print(x)', 'print(printer())', '#
x is local here:\nf = lambda x:x**2', "name = 'This is a global name'\n\ndef
greet():\n    # Enclosing function\n    name = 'Sammy'\n    \n    def hello():\n
print('Hello '+name)\n    \n    hello()\n\ngreet()", 'print(name)', 'len', "x =
50\n\ndef func(x):\n    print('x is', x)\n    x = 2\n    print('Changed local x
to', x)\n\nfunc(x)\nprint('x is still', x)", "x = 50\n\ndef func():\n    global
x\n    print('This function is now using the global x!')\n    print('Because of
global x is: ', x)\n    x = 2\n    print('Ran func(), changed global x to',
x)\n\nprint('Before calling func(), x is: ', x)\nfunc()\nprint('Value of x
(outside of func()) is: ', x)", 'def greet(name, msg = "Good morning!"):\n
print("Hello",name + \', \' + msg)\n\ngreet("Kate")\ngreet("Bruce","How do you
do?")', 'def myfunc(a,b):\n    return sum((a,b))*.05\n\nmyfunc(40,60)', 'def
myfunc(a=0,b=0,c=0,d=0,e=0):\n    return
sum((a,b,c,d,e))*.05\n\nmyfunc(40,60,20)', 'def myfunc(*args):\n    return
sum(args)*.05\n\nmyfunc(40,60,20)', 'def myfunc(*spam):\n    return
sum(spam)*.05\n\nmyfunc(40,60,20)', 'def myfunc(**kwargs):\n    if \'fruit\' in
kwargs:\n        print(f"My favorite fruit is {kwargs[\'fruit\']}")  # review
String Formatting and f-strings if this syntax is unfamiliar\n    else:\n
print("I don\'t like fruit")\n        \nmyfunc(fruit=\'pineapple\')',
'myfunc()', 'def myfunc(*args, **kwargs):\n    if \'fruit\' and \'juice\' in
kwargs:\n        print(f"I like {\' and \'.join(args)} and my favorite fruit is
{kwargs[\'fruit\']}")\n        print(f"May I have some {kwargs[\'juice\']}
juice?")\n    else:\n        pass\n
\nmyfunc(\'eggs\',\'spam\',fruit=\'cherries\',juice=\'orange\')',
"myfunc(fruit='cherries',juice='orange','eggs','spam')", '# first class
function\ndef square(x):\n    return x*x\n\ndef cube(x):\n    return
x*x*x\n\ndef my_map(func, arg_list):\n    result = []\n    for i in arg_list:\n
result.append(func(i))\n    return result\n\nsquares =
my_map(cube,[1,2,3,4,5])\n\nprint(squares)', "def outer_func():\n    message =
'Hi'\n\n    def innner_func():\n        print(message)\n    return
innner_func()\n\nouter_func()\nouter_func()\nouter_func()", "def
outer_func(msg):\n    message = msg\n\n    def innner_func():\n
print(message)\n    return innner_func\n\nhi_func = outer_func('Hi')\nbye_func =
outer_func('Bye')\n\nhi_func()\nbye_func()", 'def make_counter():\n\n    count =
0\n    def inner():\n\n        nonlocal count\n        count += 1\n
return count\n\n    return inner\n\n\ncounter = make_counter()\n\nc =
counter()\nprint(c)\n\nc = counter()\nprint(c)\n\nc = counter()\nprint(c)', 'def
hello(name=\'Jose\'):\n    print(\'The hello() function has been executed\')\n
\n    def greet():\n        return \'\\t This is inside the greet() function\'\n
\n    def welcome():\n        return "\\t This is inside the welcome()
function"\n    \n    print(greet())\n    print(welcome())\n    print("Now we are
back inside the hello() function")', 'hello()', 'welcome()', "class Dog:\n
def __init__(self, name):\n        self.name = name\n\n    def speak(self):\n
return self.name+' says Woof!'\n    \nclass Cat:\n    def __init__(self,
name):\n        self.name = name\n\n    def speak(self):\n        return
self.name+' says Meow!' \n    \nniko = Dog('Niko')\nfelix =
Cat('Felix')\n\nprint(niko.speak())\nprint(felix.speak())", 'for pet in
[niko,felix]:\n    print(pet.speak())', 'def pet_speak(pet):\n
print(pet.speak())\n\npet_speak(niko)\npet_speak(felix)', 'class Animal:\n
def __init__(self, name):    # Constructor of the class\n        self.name =
name\n\n    def speak(self):              # Abstract method, defined by
convention only\n        raise NotImplementedError("Subclass must implement
abstract method")\n\n\nclass Dog(Animal):\n    \n    def speak(self):\n
return self.name+\' says Woof!\'\n    \nclass Cat(Animal):\n\n    def
speak(self):\n        return self.name+\' says Meow!\'\n    \nfido =
Dog(\'Fido\')\nisis =
Cat(\'Isis\')\n\nprint(fido.speak())\nprint(isis.speak())', 'class Computer:\n\n
def __init__(self):\n        self.__maxprice = 900\n\n    def sell(self):\n
print("Selling Price: {}".format(self.__maxprice))\n\n    def setMaxPrice(self,
price):\n        self.__maxprice = price\n\nc = Computer()\nc.sell()', '# change
the price\nc.__maxprice = 1000\nc.sell()', '# using setter
function\nc.setMaxPrice(1000)\nc.sell()', 'class Book:\n    def __init__(self,
title, author, pages):\n        print("A book is created")\n        self.title =
title\n        self.author = author\n        self.pages = pages\n\n    def
__str__(self):\n        return "Title: %s, author: %s, pages: %s" %(self.title,
self.author, self.pages)\n\n    def __len__(self):\n        return
self.pages\n\n    def __del__(self):\n        print("A book is destroyed")',
'book = Book("Python Rocks!", "Jose Portilla", 159)\n\n#Special
Methods\nprint(book)\nprint(len(book))\ndel book', 'class Employee():\n    \n
raise_amount =1.04\n    def __init__(self,first,last,pay):\n        self.first
= first\n        self.last = last\n        self.pay = pay\n        self.email =
first+\'.\'+last+\'@company.com\'\n\n    def fullname(self):\n        return
self.first+\' \'+self.last\n    \n    def apply_raise(self):\n        self.pay =
int(self.pay * self.raise_amount)\n        return self.pay\n    \n
@classmethod\n    def set_raise_amount(cls, amount):\n        cls.raise_amount =
amount\n    \n    @classmethod\n    def from_string(cls,emp_str):\n        "
class method used as constructor"\n        first,last,pay =
emp_str.split(\'-\')\n        return cls(first,last,pay)\n    \n
@staticmethod\n    def is_workday(day):\n        if day.weekday() ==5 or
day.weekday() ==6:\n            return False\n        return True', "emp_1 =
Employee('Sudhir','Kumar',2000)\nemp_2 = Employee('Latha','Shet',4000)\n\nprint(
emp_1.raise_amount)\nEmployee.set_raise_amount(1.1)\nprint(emp_1.raise_amount)",
"# class method used as constructor\nemp_str_1 = 'John-Doe-2000'\nnew_emp =
Employee.from_string(emp_str_1)\nprint(new_emp.fullname())", 'import
datetime\nmy_day = datetime.date(2018,11,21)\nEmployee.is_workday(my_day)', '#
Suppose this is foo.py.\n\nprint("before import")\nimport math\n\nprint("before
functionA")\ndef functionA():\n    print("Function A")\n\nprint("before
functionB")\ndef functionB():\n    print("Function B
{}".format(math.sqrt(100)))\n\nprint("before __name__ guard")\nif __name__ ==
\'__main__\':\n    functionA()\n    functionB()\nprint("after __name__ guard")',
"get_ipython().run_cell_magic('writefile', 'file1.py', 'def myfunc(x):\\n
return [num for num in range(x) if num%2==0]\\nlist1 = myfunc(11)\\n')",
"get_ipython().run_cell_magic('writefile', 'file2.py', 'import
file1\\nfile1.list1.append(12)\\nprint(file1.list1)\\n')",
"get_ipython().system(' python file2.py')", 'import file1\nprint(file1.list1)',
"get_ipython().run_cell_magic('writefile', 'file3.py', 'import sys\\nimport
file1\\nnum = int(sys.argv[1])\\nprint(file1.myfunc(num))\\n')",
"get_ipython().system(' python file3.py 21')", '# import the library\nimport
math', '# use it (ceiling rounding)\nmath.ceil(2.4)', 'print(dir(math))',
'help(math.ceil)', "# Just an example, this won't work\nimport foo.bar", 'def
hello(name=\'Jose\'):\n    \n    def greet():\n        return \'\\t This is
inside the greet() function\'\n    \n    def welcome():\n        return "\\t
This is inside the welcome() function"\n    \n    if name == \'Jose\':\n
return greet\n    else:\n        return welcome', 'x = hello()', 'x',
'print(x())', "def hello():\n    return 'Hi Jose!'\n\ndef other(func):\n
print('Other code would go here')\n    print(func())", 'other(hello)', 'lst =
[1,2,3]', 'lst.count(2)',
'print(type(1))\nprint(type([]))\nprint(type(()))\nprint(type({}))', '# Create a
new object type called Sample\nclass Sample:\n    pass\n\n# Instance of
Sample\nx = Sample()\n\nprint(type(x))', "class Dog:\n    def
__init__(self,breed):\n        self.breed = breed\n        \nsam =
Dog(breed='Lab')\nfrank = Dog(breed='Huskie')", 'sam.breed', 'frank.breed',
"class Dog:\n    \n    # Class Object Attribute\n    species = 'mammal'\n    \n
def __init__(self,breed,name):\n        self.breed = breed\n        self.name =
name", "sam = Dog('Lab','Sam')", 'sam.name', 'sam.species', "class Employee():\n
\n    raise_amount =1.04\n    def __init__(self,first,last,pay):\n
self.first  = first\n        self.last = last\n        self.pay = pay\n
self.email = first+'.'+last+'@company.com'\n\n    def fullname(self):\n
return self.first+' '+self.last\n    \n    def apply_raise(self):\n
self.pay = int(self.pay * self.raise_amount)\n        return self.pay", "emp_1 =
Employee('Sudhir','Kumar',200)\nemp_2 = Employee('Latha','Shet',200)\n\nprint(em
p_1.__dict__)\nprint(emp_2.fullname())\nprint(emp_1.apply_raise())",
'print(emp_1.first)\nprint(emp_1.email)', "class Circle:\n    pi = 3.14\n\n    #
Circle gets instantiated with a radius (default is 1)\n    def __init__(self,
radius=1):\n        self.radius = radius \n        self.area = radius * radius *
Circle.pi\n\n    # Method for resetting Radius\n    def setRadius(self,
new_radius):\n        self.radius = new_radius\n        self.area = new_radius *
new_radius * self.pi\n\n    # Method for getting Circumference\n    def
getCircumference(self):\n        return self.radius * self.pi * 2\n\n\nc =
Circle()\n\nprint('Radius is: ',c.radius)\nprint('Area is:
```

```
',c.area)\nprint('Circumference is: ',c.getCircumference())",
"c.setRadius(2)\n\nprint('Radius is: ',c.radius)\nprint('Area is:
',c.area)\nprint('Circumference is: ',c.getCircumference())", 'class Animal:\n
def __init__(self):\n        print("Animal created")\n\n    def whoAmI(self):\n
print("Animal")\n\n    def eat(self):\n        print("Eating")\n\n\nclass
Dog(Animal):\n    def __init__(self):\n        Animal.__init__(self)\n
print("Dog created")\n\n    def whoAmI(self):\n        print("Dog")\n\n    def
bark(self):\n        print("Woof!")', 'd = Dog()', 'd.whoAmI()', 'd.eat()',
'd.bark()', "class Dog:\n    def __init__(self, name):\n        self.name =
name\n\n    def speak(self):\n        return self.name+' says Woof!'\n
\nclass Cat:\n    def __init__(self, name):\n        self.name = name\n\n    def
speak(self):\n        return self.name+' says Meow!' \n    \nniko =
Dog('Niko')\nfelix = Cat('Felix')\n\nprint(niko.speak())\nprint(felix.speak())",
'for pet in [niko,felix]:\n    print(pet.speak())', 'def pet_speak(pet):\n
print(pet.speak())\n\npet_speak(niko)\npet_speak(felix)', 'class Animal:\n
def __init__(self, name):    # Constructor of the class\n        self.name =
name\n\n    def speak(self):              # Abstract method, defined by
convention only\n        raise NotImplementedError("Subclass must implement
abstract method")\n\n\nclass Dog(Animal):\n    \n    def speak(self):\n
return self.name+\' says Woof!\'\n    \nclass Cat(Animal):\n\n    def
speak(self):\n        return self.name+\' says Meow!\'\n    \nfido =
Dog(\'Fido\')\nisis =
Cat(\'Isis\')\n\nprint(fido.speak())\nprint(isis.speak())', 'class Computer:\n\n
def __init__(self):\n        self.__maxprice = 900\n\n    def sell(self):\n
print("Selling Price: {}".format(self.__maxprice))\n\n    def setMaxPrice(self,
price):\n        self.__maxprice = price\n\nc = Computer()\nc.sell()', '# change
the price\nc.__maxprice = 1000\nc.sell()', '# using setter
function\nc.setMaxPrice(1000)\nc.sell()', 'class Book:\n    def __init__(self,
title, author, pages):\n        print("A book is created")\n        self.title =
title\n        self.author = author\n        self.pages = pages\n\n    def
__str__(self):\n        return "Title: %s, author: %s, pages: %s" %(self.title,
self.author, self.pages)\n\n    def __len__(self):\n        return
self.pages\n\n    def __del__(self):\n        print("A book is destroyed")',
'book = Book("Python Rocks!", "Jose Portilla", 159)\n\n#Special
Methods\nprint(book)\nprint(len(book))\ndel book', 'class Employee():\n    \n
raise_amount =1.04\n    def __init__(self,first,last,pay):\n        self.first
= first\n        self.last = last\n        self.pay = pay\n        self.email =
first+\'.\'+last+\'@company.com\'\n\n    def fullname(self):\n        return
self.first+\' \'+self.last\n    \n    def apply_raise(self):\n        self.pay =
int(self.pay * self.raise_amount)\n        return self.pay\n    \n
@classmethod\n    def set_raise_amount(cls, amount):\n        cls.raise_amount =
amount\n    \n    @classmethod\n    def from_string(cls,emp_str):\n        "
class method used as constructor"\n        first,last,pay =
emp_str.split(\'-\')\n        return cls(first,last,pay)\n    \n
@staticmethod\n    def is_workday(day):\n        if day.weekday() ==5 or
day.weekday() ==6:\n            return False\n        return True', "emp_1 =
Employee('Sudhir','Kumar',2000)\nemp_2 = Employee('Latha','Shet',4000)\n\nprint(
emp_1.raise_amount)\nEmployee.set_raise_amount(1.1)\nprint(emp_1.raise_amount)",
"# class method used as constructor\nemp_str_1 = 'John-Doe-2000'\nnew_emp =
Employee.from_string(emp_str_1)\nprint(new_emp.fullname())", 'import
datetime\nmy_day = datetime.date(2018,11,21)\nEmployee.is_workday(my_day)', '#
Suppose this is foo.py.\n\nprint("before import")\nimport math\n\nprint("before
functionA")\ndef functionA():\n    print("Function A")\n\nprint("before
functionB")\ndef functionB():\n    print("Function B
{}".format(math.sqrt(100)))\n\nprint("before __name__ guard")\nif __name__ ==
\'__main__\':\n    functionA()\n    functionB()\nprint("after __name__ guard")',
"get_ipython().run_cell_magic('writefile', 'file1.py', 'def myfunc(x):\\n
return [num for num in range(x) if num%2==0]\\nlist1 = myfunc(11)\\n')",
"get_ipython().run_cell_magic('writefile', 'file2.py', 'import
file1\\nfile1.list1.append(12)\\nprint(file1.list1)\\n')",
"get_ipython().system(' python file2.py')", 'import file1\nprint(file1.list1)',
"get_ipython().run_cell_magic('writefile', 'file3.py', 'import sys\\nimport
file1\\nnum = int(sys.argv[1])\\nprint(file1.myfunc(num))\\n')",
"get_ipython().system(' python file3.py 21')", '# import the library\nimport
math', '# use it (ceiling rounding)\nmath.ceil(2.4)', 'print(dir(math))',
'help(math.ceil)', "# Just an example, this won't work\nimport foo.bar", '# OR
could do it this way\nfrom foo import bar', "get_ipython().run_line_magic('ls',
'')", '__init__.py:\n\n__all__ = ["bar"]', "print('Hello)", "print('Hello')",
'try:\n    f = open(\'testfile\',\'w\')\n    f.write(\'Test write
this\')\nexcept IOError:\n    # This will only check for an IOError exception
and then execute this print statement\n    print("Error: Could not find file or
read data")\nelse:\n    print("Content written successfully")\n    f.close()',
'try:\n    f = open(\'testfile\',\'r\')\n    f.write(\'Test write
this\')\nexcept IOError:\n    # This will only check for an IOError exception
and then execute this print statement\n    print("Error: Could not find file or
read data")\nelse:\n    print("Content written successfully")\n    f.close()',
'try:\n    f = open(\'testfile\',\'r\')\n    f.write(\'Test write
this\')\nexcept:\n    # This will check for any exception and then execute this
print statement\n    print("Error: Could not find file or read data")\nelse:\n
print("Content written successfully")\n    f.close()', 'try:\n    f =
open("testfile", "w")\n    f.write("Test write statement")\n
f.close()\nfinally:\n    print("Always execute finally code blocks")', 'def
askint():\n    try:\n        val = int(input("Please enter an integer: "))\n
except:\n        print("Looks like you did not enter an integer!")\n\n
finally:\n        print("Finally, I executed!")\n    print(val)', 'askint()',
'askint()', 'def askint():\n    try:\n        val = int(input("Please enter an
integer: "))\n    except:\n        print("Looks like you did not enter an
integer!")\n        val = int(input("Try again-Please enter an integer: "))\n
finally:\n        print("Finally, I executed!")\n    print(val)', 'askint()',
'def askint():\n    while True:\n        try:\n            val =
int(input("Please enter an integer: "))\n        except:\n
print("Looks like you did not enter an integer!")\n            continue\n
else:\n            print("Yep that\'s an integer!")\n            break\n
finally:\n            print("Finally, I executed!")\n        print(val)',
'askint()', 'def askint():\n    while True:\n        try:\n            val =
int(input("Please enter an integer: "))\n        except:\n
```

```
print("Looks like you did not enter an integer!")\n           continue\n
else:\n           print("Yep that\'s an integer!")\n           print(val)\n
break\n       finally:\n           print("Finally, I executed!")', 'askint()',
'def fahrenheit(celsius):\n    return (9/5)*celsius + 32\n    \ntemps = [0,
22.5, 40, 100]', 'F_temps = map(fahrenheit, temps)\n\n#Show\nlist(F_temps)', 'a
= [1,2,3,4]\nb = [5,6,7,8]\nc = [9,10,11,12]\n\nlist(map(lambda x,y:x+y,a,b))',
'# Now all three lists\nlist(map(lambda x,y,z:x+y+z,a,b,c))', 'from functools
import reduce\n\nlst =[47,11,42,13]\nreduce(lambda x,y: x+y,lst)', "from
IPython.display import Image\nImage('http://www.python-
course.eu/images/reduce_diagram.png')", '#Find the maximum of a sequence (This
already exists as max())\nmax_find = lambda a,b: a if (a > b) else b', '#Find
max\nreduce(max_find,lst)', "#First let's make a function\ndef
even_check(num):\n    if num%2 ==0:\n        return True", 'lst
=range(20)\n\nlist(filter(even_check,lst))', 'list(filter(lambda x:
x%2==0,lst))', 'x = [1,2,3]\ny = [4,5,6]\n\n# Zip the lists
together\nlist(zip(x,y))', "lst = ['a','b','c']\nfor count,item in
enumerate(lst):\n    if count >= 2:\n        break\n    else:\n
print(item)", "months =
['March','April','May','June']\n\nlist(enumerate(months,start=3))", 'lst =
[True,True,False,True]', 'all(lst)', 'any(lst)', '# Create 2+3j\ncomplex(2,3)',
"complex('12+2j')", 'def func():\n    return 1\nfunc()', "s = 'Global
Variable'\n\ndef check_for_locals():\n    print(locals())", 'print(globals())',
'print(globals().keys())', "globals()['s']", 'check_for_locals()', "def
hello(name='Jose'):\n    return 'Hello '+name", 'hello()', 'greet = hello',
'greet', 'greet()', 'del hello', 'hello()', 'print(globals()[10])',
'print(globals()[:10])', 'print(globals())', '#print(globals())',
'print(globals().keys())', 'print(globals())'], '_oh': {1: 3, 2: 1, 3: 4, 4:
1.5, 5: 1, 6: 2, 7: 8, 9: 10, 13: 2, 15: ['Sammy', 'Frankie'], 16: 'hello', 17:
'Hello World', 18: 11, 20: 'H', 21: 'Hello Worl', 22: 'Hello World', 23:
'HloWrd', 24: 'dlroW olleH', 26: 'Hello World concatenate me!', 27: 'HELLO
WORLD', 28: 'hello world', 29: ['Hello', 'World'], 30: ['Hello ', 'orld'], 31:
'Insert another string with curly brackets: The inserted string', 43: 4, 45:
'one', 46: ['one', 'two', 'three', 4, 5, 'new item'], 47: ['one', 'two',
'three', 4, 5, 'one', 'two', 'three', 4, 5], 48: [2, 3, 'append me!'], 49: [2,
3, 'append me!'], 50: 1, 51: [2, 3, 'append me!'], 53: [[1, 2, 3], [4, 5, 6],
[7, 8, 9]], 55: [1, 4, 7], 56: 'value2', 58: 'value', 60: dict_keys(['key1',
'key2', 'key3']), 61: dict_values([1, 2, 3]), 62: dict_items([('key1', 1),
('key2', 2), ('key3', 3)]), 63: 3, 64: ('one', 2), 65: 0, 66: 1, 71: True, 72:
True, 73: True, 74: True, 75: True, 88: range(0, 11), 89: [0, 1, 2, 3, 4, 5, 6,
7, 8, 9, 10], 90: ([1, 3, 5, 7, 9], [0, 2, 4, 6, 8]), 91: [10, 9, 8, 7, 6, 5, 4,
3, 2, 1, 0, -1, -2, -3, -4, -5, -6, -7, -8, -9], 92: [(10, 'a'), (11, 'b'), (12,
'c'), (13, 'd')], 94: [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')], 95:
<zip object at 0x7f63344fa2d0>, 97: True, 98: False, 99: 10, 100: 100, 102: [20,
30, 40, 10, 100], 103: 'Hi,10', 104: ['w', 'o', 'r', 'd'], 106: [1, 2, 3, 4, 5,
6], 111: [1, 4, 9, 16, 25], 112: ['even', 'C', 'S', 'K', 'even'], 113: [0, 2, 4,
6, 8, 10], 121: <built-in function len>, 125: 5.0, 126: 6.0, 127: 6.0, 128: 6.0,
152: True, 161: 3, 167: <function hello.<locals>.greet at 0x7f63345344d0>, 172:
1, 176: 'Lab', 177: 'Huskie', 180: 'Sam', 181: 'mammal', 204: True, 213: 3, 236:
```

```
[32.0, 72.5, 104.0, 212.0], 237: [6, 8, 10, 12], 238: [15, 18, 21, 24], 239:
113, 240: <IPython.core.display.Image object>, 242: 47, 244: [0, 2, 4, 6, 8, 10,
12, 14, 16, 18], 245: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18], 246: [(1, 4), (2, 5),
(3, 6)], 248: [(3, 'March'), (4, 'April'), (5, 'May'), (6, 'June')], 250: False,
251: True, 252: (2+3j), 253: (12+2j), 254: 1, 258: 'Global Variable', 261:
'Hello Jose', 263: <function hello at 0x7f63340550e0>, 264: 'Hello Jose'},
'_dh': ['/home/sudhir/Downloads/AI/Courses/Complete Python Bootcamp: Go from
zero to hero in Python'], 'In': ['', '# Addition\n2+1', '# Subtraction\n2-1', '#
Multiplication\n2*2', '# Division\n3/2', '# Floor Division\n7//4', '# Mod \n10 %
4', '# Powers\n2**3', '# Let\'s create an object called "a" and assign it the
number 5\na = 5', '# Adding the objects\na+a', "# Use object names to keep
better track of what's going on in your code!\nmy_income = 100\n\ntax_rate =
0.1\n\nmy_taxes = my_income*tax_rate", 'a = b = c = 1', 'my_dogs = 2',
'my_dogs', "my_dogs = ['Sammy', 'Frankie']", 'my_dogs', "# Single
word\n'hello'", "# We can simply declare a string\n'Hello World'", "len('Hello
World')", "# Assign s as a string\ns = 'Hello World'", '# Show first element (in
this case a letter)\ns[0]', '# Grab everything but the last letter\ns[:-1]', '#
Grab everything, but go in steps size of 1\ns[::1]', '# Grab everything, but go
in step sizes of 2\ns[::2]', '# We can use this to print a string
backwards\ns[::-1]', "# Let's try to change the first letter to 'x'\n#s[0] =
'x'", "# Concatenate strings!\ns + ' concatenate me!'", '# Upper Case a
string\ns.upper()', '# Lower case\ns.lower()', '# Split a string by blank space
(this is the default)\ns.split()', "# Split by a specific element (doesn't
include the element that was split on)\ns.split('W')", "'Insert another string
with curly brackets: {}'.format('The inserted string')", 'print("I\'m going to
inject %s text here, and %s text here." %(\'some\',\'more\'))', 'x, y =
\'some\', \'more\'\nprint("I\'m going to inject %s text here, and %s text
here."%(x,y))', "print('I once caught a fish %s.' %'this \\tbig')\nprint('I once
caught a fish %r.' %'this \\tbig')", "print('I wrote %s programs today.'
%3.75)\nprint('I wrote %d programs today.' %3.75)   ", "print('Floating point
numbers: %5.2f' %(13.144))", "print('Floating point numbers: %25.2f'
%(13.144))", "print('{0:8} | {1:9}'.format('Fruit', 'Quantity'))\nprint('{0:8} |
{1:9}'.format('Apples', 3.))\nprint('{0:8} | {1:9}'.format('Oranges', 10))",
'name = \'Fred\'\n\nprint(f"He said his name is {name}.")', 'print(f"He said his
name is {name!r}")', '# Assign a list to an variable named my_list\nmy_list =
[1,2,3]', "my_list = ['A string',23,100.232,'o']", 'len(my_list)', "my_list =
['one','two','three',4,5]", '# Grab element at index 0\nmy_list[0]', "my_list +
['new item']", '# Make the list double\nmy_list * 2', '# Create a new
list\nlist1 = [1,2,3]\nlist1', "# Append\nlist1.append('append me!')\nlist1", '#
Pop off the 0 indexed item\nlist1.pop(0)', '# Show\nlist1', "# Let's make three
lists\nlst_1=[1,2,3]\nlst_2=[4,5,6]\nlst_3=[7,8,9]\n\n# Make a list of lists to
form a matrix\nmatrix = [lst_1,lst_2,lst_3]", '# Show\nmatrix', '# Build a list
comprehension by deconstructing a for loop within a []\nfirst_col = [row[0] for
row in matrix]', 'first_col', "# Make a dictionary with {} and : to signify a
key and a value\nmy_dict = {'key1':'value1','key2':'value2'}\n# Call values by
their key\nmy_dict['key2']", "# Dictionary nested inside a dictionary nested
inside a dictionary\nd = {'key1':{'nestkey':{'subnestkey':'value'}}}", "# Keep
calling the keys\nd['key1']['nestkey']['subnestkey']", "# Create a typical
```

```
dictionary\nd = {'key1':1,'key2':2,'key3':3}", '# Method to return a list of all
keys \nd.keys()', '# Method to grab all values\nd.values()', "# Method to return
tuples of all items  (we'll learn about tuples soon)\nd.items()", '# Create a
tuple\nt = (1,2,3)\n# Check len just like a list\nlen(t)', "# Can also mix
object types\nt = ('one',2)\n\n# Show\nt", "# Use .index to enter a value and
return the index\nt.index('one')", "# Use .count to count the number of times a
value appears\nt.count('one')", "t[0]= 'change'",
"get_ipython().run_cell_magic('writefile', 'test.txt', 'Hello, this is a quick
test file.\\n')", "myfile = open('whoops.txt')", '# Read the
file\nmy_file.seek(0)\nmy_file.read()', '2 == 2', '2 <= 2', '2 <= 4', '1 < 2 <
3', '1<2 and 2<3', "if True:\n    print('It was true!')", "x = False\n\nif x:\n
print('x was True!')\nelse:\n    print('I will be printed in any case where x is
not true')", "# We'll learn how to automate this sort of list in the next
lecture\nlist1 = [1,2,3,4,5,6,7,8,9,10]\n\nfor num in list1:\n    print(num)",
'tup = (1,2,3,4,5)\n\nfor t in tup:\n    print(t)', 'list2 =
[(2,4),(6,8),(10,12)]', 'for tup in list2:\n    print(tup)', '# Now with
unpacking!\nfor (t1,t2) in list2:\n    print(t1)', "d = {'k1':1,'k2':2,'k3':3}",
'for item in d:\n    print(item)', "x = 0\n\nwhile x < 10:\n    print('x is
currently: ',x)\n    print(' x is still less than 10, adding 1 to x')\n
x+=1", "x = 0\n\nwhile x < 10:\n    print('x is currently: ',x)\n    print(' x
is still less than 10, adding 1 to x')\n    x+=1\n    \nelse:\n    print('All
Done!')", "x = 0\n\nwhile x < 10:\n    print('x is currently: ',x)\n    print('
x is still less than 10, adding 1 to x')\n    x+=1\n    if x==3:\n
print('Breaking because x==3')\n        break\n    else:\n
print('continuing…')\n        continue", 'range(0,11)', '# Notice how 11 is
not included, up to but not including 11, just like slice
notation!\nlist(range(0,11))', '# odd --------- even\nlist(range(1,10,2)),
list(range(0,10,2))', 'list(range(10,-10,-1))', "x = ['a','b','c','d']\na=
enumerate(x, start = 10)\nlist(a)", 'index_count = 0\n\nfor letter in
\'abcde\':\n    print("At index {} the letter is
{}".format(index_count,letter))\n    index_count += 1',
"list(enumerate('abcde'))", "mylist1 = [1,2,3,4,5]\nmylist2 =
['a','b','c','d','e']\n\n# This one is also a generator! We will explain this
later, but for now let's transform it to a list\nzip(mylist1,mylist2)", '#
Notice the tuple unpacking!\n\nfor i,letter in enumerate(\'abcde\'):\n
print("At index {} the letter is {}".format(i,letter))', "'x' in ['x','y','z']",
"'x' in [1,2,3]", 'mylist = [10,20,30,40,100]\nmin(mylist)', 'max(mylist)',
'from random import shuffle', '# This shuffles the list "in-place" meaning it
won\'t return\n# anything, instead it will effect the list
passed\nshuffle(mylist)\nmylist', "input('Enter Something into this box: ')", "#
Grab every letter in string\nlst = [x for x in 'word']\nlst", '# Create a simple
list\nlst = [1,2,3,4,5]', 'lst.append(6)\nlst', 'help(lst.count)', "def
name_of_function(arg1,arg2):\n    '''\n    This is where the function's Document
String (docstring) goes\n    '''\n    # Do stuff here\n    # Return desired
result", "def is_prime(num):\n    '''\n    Naive method of checking for primes.
\n    '''\n    for n in range(2,num):\n        if num % n == 0:\n
print(num,'is not prime')\n            break\n    else: # If never mod zero,
then prime\n        print(num,'is prime!')", 'is_prime(16)', 'def square(num):\n
```

```
return num**2\nmy_nums = [1,2,3,4,5]\nmap(square,my_nums)\n\n# To get the
results, either iterate through map() \n# or just cast to a
list\nlist(map(square,my_nums))', "def splicer(mystring):\n    if len(mystring)
% 2 == 0:\n        return 'even'\n    else:\n        return mystring[0]\n
\nmynames =
['John','Cindy','Sarah','Kelly','Mike']\nlist(map(splicer,mynames))", 'def
check_even(num):\n    return num % 2 == 0 \n\nnums = [0,1,2,3,4,5,6,7,8,9,10]\nf
ilter(check_even,nums)\nlist(filter(check_even,nums))', 'double = lambda x: x *
2\n\n# Output: 10\nprint(double(5))', 'x = 25\n\ndef printer():\n    x = 50\n
return x\n\n# print(x)\n# print(printer())', 'print(x)', 'print(printer())', '#
x is local here:\nf = lambda x:x**2', "name = 'This is a global name'\n\ndef
greet():\n    # Enclosing function\n    name = 'Sammy'\n    \n    def hello():\n
print('Hello '+name)\n    \n    hello()\n\ngreet()", 'print(name)', 'len', "x =
50\n\ndef func(x):\n    print('x is', x)\n    x = 2\n    print('Changed local x
to', x)\n\nfunc(x)\nprint('x is still', x)", "x = 50\n\ndef func():\n    global
x\n    print('This function is now using the global x!')\n    print('Because of
global x is: ', x)\n    x = 2\n    print('Ran func(), changed global x to',
x)\n\nprint('Before calling func(), x is: ', x)\nfunc()\nprint('Value of x
(outside of func()) is: ', x)", 'def greet(name, msg = "Good morning!"):\n
print("Hello",name + \', \' + msg)\n\ngreet("Kate")\ngreet("Bruce","How do you
do?")', 'def myfunc(a,b):\n    return sum((a,b))*.05\n\nmyfunc(40,60)', 'def
myfunc(a=0,b=0,c=0,d=0,e=0):\n    return
sum((a,b,c,d,e))*.05\n\nmyfunc(40,60,20)', 'def myfunc(*args):\n    return
sum(args)*.05\n\nmyfunc(40,60,20)', 'def myfunc(*spam):\n    return
sum(spam)*.05\n\nmyfunc(40,60,20)', 'def myfunc(**kwargs):\n    if \'fruit\' in
kwargs:\n        print(f"My favorite fruit is {kwargs[\'fruit\']}")  # review
String Formatting and f-strings if this syntax is unfamiliar\n    else:\n
print("I don\'t like fruit")\n        \nmyfunc(fruit=\'pineapple\')',
'myfunc()', 'def myfunc(*args, **kwargs):\n    if \'fruit\' and \'juice\' in
kwargs:\n        print(f"I like {\' and \'.join(args)} and my favorite fruit is
{kwargs[\'fruit\']}")\n        print(f"May I have some {kwargs[\'juice\']}
juice?")\n    else:\n        pass\n
\nmyfunc(\'eggs\',\'spam\',fruit=\'cherries\',juice=\'orange\')',
"myfunc(fruit='cherries',juice='orange','eggs','spam')", '# first class
function\ndef square(x):\n    return x*x\n\ndef cube(x):\n    return
x*x*x\n\ndef my_map(func, arg_list):\n    result = []\n    for i in arg_list:\n
result.append(func(i))\n    return result\n\nsquares =
my_map(cube,[1,2,3,4,5])\n\nprint(squares)', "def outer_func():\n    message =
'Hi'\n\n    def innner_func():\n        print(message)\n    return
innner_func()\n\nouter_func()\nouter_func()\nouter_func()", "def
outer_func(msg):\n    message = msg\n\n    def innner_func():\n
print(message)\n    return innner_func\n\nhi_func = outer_func('Hi')\nbye_func =
outer_func('Bye')\n\nhi_func()\nbye_func()", 'def make_counter():\n\n    count =
0\n    def inner():\n\n        nonlocal count\n        count += 1\n
return count\n\n    return inner\n\n\ncounter = make_counter()\n\nc =
counter()\nprint(c)\n\nc = counter()\nprint(c)\n\nc = counter()\nprint(c)', 'def
hello(name=\'Jose\'):\n    print(\'The hello() function has been executed\')\n
\n    def greet():\n        return \'\\t This is inside the greet() function\'\n
```

```
\n    def welcome():\n        return "\\t This is inside the welcome()
function"\n    \n    print(greet())\n    print(welcome())\n    print("Now we are
back inside the hello() function")', 'hello()', 'welcome()', "class Dog:\n
def __init__(self, name):\n        self.name = name\n\n    def speak(self):\n
return self.name+' says Woof!'\n    \nclass Cat:\n    def __init__(self,
name):\n        self.name = name\n\n    def speak(self):\n        return
self.name+' says Meow!' \n    \nniko = Dog('Niko')\nfelix =
Cat('Felix')\n\nprint(niko.speak())\nprint(felix.speak())", 'for pet in
[niko,felix]:\n    print(pet.speak())', 'def pet_speak(pet):\n
print(pet.speak())\n\npet_speak(niko)\npet_speak(felix)', 'class Animal:\n
def __init__(self, name):    # Constructor of the class\n        self.name =
name\n\n    def speak(self):              # Abstract method, defined by
convention only\n        raise NotImplementedError("Subclass must implement
abstract method")\n\n\nclass Dog(Animal):\n    \n    def speak(self):\n
return self.name+\' says Woof!\'\n    \nclass Cat(Animal):\n\n    def
speak(self):\n        return self.name+\' says Meow!\'\n    \nfido =
Dog(\'Fido\')\nisis =
Cat(\'Isis\')\n\nprint(fido.speak())\nprint(isis.speak())', 'class Computer:\n\n
def __init__(self):\n        self.__maxprice = 900\n\n    def sell(self):\n
print("Selling Price: {}".format(self.__maxprice))\n\n    def setMaxPrice(self,
price):\n        self.__maxprice = price\n\nc = Computer()\nc.sell()', '# change
the price\nc.__maxprice = 1000\nc.sell()', '# using setter
function\nc.setMaxPrice(1000)\nc.sell()', 'class Book:\n    def __init__(self,
title, author, pages):\n        print("A book is created")\n        self.title =
title\n        self.author = author\n        self.pages = pages\n\n    def
__str__(self):\n        return "Title: %s, author: %s, pages: %s" %(self.title,
self.author, self.pages)\n\n    def __len__(self):\n        return
self.pages\n\n    def __del__(self):\n        print("A book is destroyed")',
'book = Book("Python Rocks!", "Jose Portilla", 159)\n\n#Special
Methods\nprint(book)\nprint(len(book))\ndel book', 'class Employee():\n    \n
raise_amount =1.04\n    def __init__(self,first,last,pay):\n        self.first
= first\n        self.last = last\n        self.pay = pay\n        self.email =
first+\'.\'+last+\'@company.com\'\n\n    def fullname(self):\n        return
self.first+\' \'+self.last\n    \n    def apply_raise(self):\n        self.pay =
int(self.pay * self.raise_amount)\n        return self.pay\n    \n
@classmethod\n    def set_raise_amount(cls, amount):\n        cls.raise_amount =
amount\n    \n    @classmethod\n    def from_string(cls,emp_str):\n        "
class method used as constructor"\n        first,last,pay =
emp_str.split(\'-\')\n        return cls(first,last,pay)\n    \n
@staticmethod\n    def is_workday(day):\n        if day.weekday() ==5 or
day.weekday() ==6:\n            return False\n        return True', "emp_1 =
Employee('Sudhir','Kumar',2000)\nemp_2 = Employee('Latha','Shet',4000)\n\nprint(
emp_1.raise_amount)\nEmployee.set_raise_amount(1.1)\nprint(emp_1.raise_amount)",
"# class method used as constructor\nemp_str_1 = 'John-Doe-2000'\nnew_emp =
Employee.from_string(emp_str_1)\nprint(new_emp.fullname())", 'import
datetime\nmy_day = datetime.date(2018,11,21)\nEmployee.is_workday(my_day)', '#
Suppose this is foo.py.\n\nprint("before import")\nimport math\n\nprint("before
functionA")\ndef functionA():\n    print("Function A")\n\nprint("before
functionB")\ndef functionB():\n    print("Function B
{}".format(math.sqrt(100)))\n\nprint("before __name__ guard")\nif __name__ ==
\'__main__\':\n    functionA()\n    functionB()\nprint("after __name__ guard")',
"get_ipython().run_cell_magic('writefile', 'file1.py', 'def myfunc(x):\\n
return [num for num in range(x) if num%2==0]\\nlist1 = myfunc(11)\\n')",
"get_ipython().run_cell_magic('writefile', 'file2.py', 'import
file1\\nfile1.list1.append(12)\\nprint(file1.list1)\\n')",
"get_ipython().system(' python file2.py')", 'import file1\nprint(file1.list1)',
"get_ipython().run_cell_magic('writefile', 'file3.py', 'import sys\\nimport
file1\\nnum = int(sys.argv[1])\\nprint(file1.myfunc(num))\\n')",
"get_ipython().system(' python file3.py 21')", '# import the library\nimport
math', '# use it (ceiling rounding)\nmath.ceil(2.4)', 'print(dir(math))',
'help(math.ceil)', "# Just an example, this won't work\nimport foo.bar", 'def
hello(name=\'Jose\'):\n    \n    def greet():\n        return \'\\t This is
inside the greet() function\'\n    \n    def welcome():\n        return "\\t
This is inside the welcome() function"\n    \n    if name == \'Jose\':\n
return greet\n    else:\n        return welcome', 'x = hello()', 'x',
'print(x())', "def hello():\n    return 'Hi Jose!'\n\ndef other(func):\n
print('Other code would go here')\n    print(func())", 'other(hello)', 'lst =
[1,2,3]', 'lst.count(2)',
'print(type(1))\nprint(type([]))\nprint(type(()))\nprint(type({}))', '# Create a
new object type called Sample\nclass Sample:\n    pass\n\n# Instance of
Sample\nx = Sample()\n\nprint(type(x))', "class Dog:\n    def
__init__(self,breed):\n        self.breed = breed\n        \nsam =
Dog(breed='Lab')\nfrank = Dog(breed='Huskie')", 'sam.breed', 'frank.breed',
"class Dog:\n    \n    # Class Object Attribute\n    species = 'mammal'\n    \n
def __init__(self,breed,name):\n        self.breed = breed\n        self.name =
name", "sam = Dog('Lab','Sam')", 'sam.name', 'sam.species', "class Employee():\n
\n    raise_amount =1.04\n    def __init__(self,first,last,pay):\n
self.first  = first\n        self.last = last\n        self.pay = pay\n
self.email = first+'.'+last+'@company.com'\n\n    def fullname(self):\n
return self.first+' '+self.last\n    \n    def apply_raise(self):\n
self.pay = int(self.pay * self.raise_amount)\n        return self.pay", "emp_1 =
Employee('Sudhir','Kumar',200)\nemp_2 = Employee('Latha','Shet',200)\n\nprint(em
p_1.__dict__)\nprint(emp_2.fullname())\nprint(emp_1.apply_raise())",
'print(emp_1.first)\nprint(emp_1.email)', "class Circle:\n    pi = 3.14\n\n    #
Circle gets instantiated with a radius (default is 1)\n    def __init__(self,
radius=1):\n        self.radius = radius \n        self.area = radius * radius *
Circle.pi\n\n    # Method for resetting Radius\n    def setRadius(self,
new_radius):\n        self.radius = new_radius\n        self.area = new_radius *
new_radius * self.pi\n\n    # Method for getting Circumference\n    def
getCircumference(self):\n        return self.radius * self.pi * 2\n\n\nc =
Circle()\n\nprint('Radius is: ',c.radius)\nprint('Area is:
',c.area)\nprint('Circumference is: ',c.getCircumference())",
"c.setRadius(2)\n\nprint('Radius is: ',c.radius)\nprint('Area is:
',c.area)\nprint('Circumference is: ',c.getCircumference())", 'class Animal:\n
def __init__(self):\n        print("Animal created")\n\n    def whoAmI(self):\n
print("Animal")\n\n    def eat(self):\n        print("Eating")\n\n\nclass
```

```
Dog(Animal):\n    def __init__(self):\n        Animal.__init__(self)\n
print("Dog created")\n\n    def whoAmI(self):\n        print("Dog")\n\n    def
bark(self):\n        print("Woof!")', 'd = Dog()', 'd.whoAmI()', 'd.eat()',
'd.bark()', "class Dog:\n    def __init__(self, name):\n        self.name =
name\n\n    def speak(self):\n        return self.name+' says Woof!'\n
\nclass Cat:\n    def __init__(self, name):\n        self.name = name\n\n    def
speak(self):\n        return self.name+' says Meow!' \n    \nniko =
Dog('Niko')\nfelix = Cat('Felix')\n\nprint(niko.speak())\nprint(felix.speak())",
'for pet in [niko,felix]:\n    print(pet.speak())', 'def pet_speak(pet):\n
print(pet.speak())\n\npet_speak(niko)\npet_speak(felix)', 'class Animal:\n
def __init__(self, name):    # Constructor of the class\n        self.name =
name\n\n    def speak(self):              # Abstract method, defined by
convention only\n        raise NotImplementedError("Subclass must implement
abstract method")\n\n\nclass Dog(Animal):\n    \n    def speak(self):\n
return self.name+\' says Woof!\'\n    \nclass Cat(Animal):\n\n    def
speak(self):\n        return self.name+\' says Meow!\'\n    \nfido =
Dog(\'Fido\')\nisis =
Cat(\'Isis\')\n\nprint(fido.speak())\nprint(isis.speak())', 'class Computer:\n\n
def __init__(self):\n        self.__maxprice = 900\n\n    def sell(self):\n
print("Selling Price: {}".format(self.__maxprice))\n\n    def setMaxPrice(self,
price):\n        self.__maxprice = price\n\nc = Computer()\nc.sell()', '# change
the price\nc.__maxprice = 1000\nc.sell()', '# using setter
function\nc.setMaxPrice(1000)\nc.sell()', 'class Book:\n    def __init__(self,
title, author, pages):\n        print("A book is created")\n        self.title =
title\n        self.author = author\n        self.pages = pages\n\n    def
__str__(self):\n        return "Title: %s, author: %s, pages: %s" %(self.title,
self.author, self.pages)\n\n    def __len__(self):\n        return
self.pages\n\n    def __del__(self):\n        print("A book is destroyed")',
'book = Book("Python Rocks!", "Jose Portilla", 159)\n\n#Special
Methods\nprint(book)\nprint(len(book))\ndel book', 'class Employee():\n    \n
raise_amount =1.04\n    def __init__(self,first,last,pay):\n        self.first
= first\n        self.last = last\n        self.pay = pay\n        self.email =
first+\'.\'+last+\'@company.com\'\n\n    def fullname(self):\n        return
self.first+\' \'+self.last\n    \n    def apply_raise(self):\n        self.pay =
int(self.pay * self.raise_amount)\n        return self.pay\n    \n
@classmethod\n    def set_raise_amount(cls, amount):\n        cls.raise_amount =
amount\n    \n    @classmethod\n    def from_string(cls,emp_str):\n        "
class method used as constructor"\n        first,last,pay =
emp_str.split(\'-\')\n        return cls(first,last,pay)\n    \n
@staticmethod\n    def is_workday(day):\n        if day.weekday() ==5 or
day.weekday() ==6:\n            return False\n        return True', "emp_1 =
Employee('Sudhir','Kumar',2000)\nemp_2 = Employee('Latha','Shet',4000)\n\nprint(
emp_1.raise_amount)\nEmployee.set_raise_amount(1.1)\nprint(emp_1.raise_amount)",
"# class method used as constructor\nemp_str_1 = 'John-Doe-2000'\nnew_emp =
Employee.from_string(emp_str_1)\nprint(new_emp.fullname())", 'import
datetime\nmy_day = datetime.date(2018,11,21)\nEmployee.is_workday(my_day)', '#
Suppose this is foo.py.\n\nprint("before import")\nimport math\n\nprint("before
functionA")\ndef functionA():\n    print("Function A")\n\nprint("before
functionB")\ndef functionB():\n    print("Function B
{}".format(math.sqrt(100)))\n\nprint("before __name__ guard")\nif __name__ ==
\'__main__\':\n    functionA()\n    functionB()\nprint("after __name__ guard")',
"get_ipython().run_cell_magic('writefile', 'file1.py', 'def myfunc(x):\\n
return [num for num in range(x) if num%2==0]\\nlist1 = myfunc(11)\\n')",
"get_ipython().run_cell_magic('writefile', 'file2.py', 'import
file1\\nfile1.list1.append(12)\\nprint(file1.list1)\\n')",
"get_ipython().system(' python file2.py')", 'import file1\nprint(file1.list1)',
"get_ipython().run_cell_magic('writefile', 'file3.py', 'import sys\\nimport
file1\\nnum = int(sys.argv[1])\\nprint(file1.myfunc(num))\\n')",
"get_ipython().system(' python file3.py 21')", '# import the library\nimport
math', '# use it (ceiling rounding)\nmath.ceil(2.4)', 'print(dir(math))',
'help(math.ceil)', "# Just an example, this won't work\nimport foo.bar", '# OR
could do it this way\nfrom foo import bar', "get_ipython().run_line_magic('ls',
'')", '__init__.py:\n\n__all__ = ["bar"]', "print('Hello)", "print('Hello')",
'try:\n    f = open(\'testfile\',\'w\')\n    f.write(\'Test write
this\')\nexcept IOError:\n    # This will only check for an IOError exception
and then execute this print statement\n    print("Error: Could not find file or
read data")\nelse:\n    print("Content written successfully")\n    f.close()',
'try:\n    f = open(\'testfile\',\'r\')\n    f.write(\'Test write
this\')\nexcept IOError:\n    # This will only check for an IOError exception
and then execute this print statement\n    print("Error: Could not find file or
read data")\nelse:\n    print("Content written successfully")\n    f.close()',
'try:\n    f = open(\'testfile\',\'r\')\n    f.write(\'Test write
this\')\nexcept:\n    # This will check for any exception and then execute this
print statement\n    print("Error: Could not find file or read data")\nelse:\n
print("Content written successfully")\n    f.close()', 'try:\n    f =
open("testfile", "w")\n    f.write("Test write statement")\n
f.close()\nfinally:\n    print("Always execute finally code blocks")', 'def
askint():\n    try:\n        val = int(input("Please enter an integer: "))\n
except:\n        print("Looks like you did not enter an integer!")\n\n
finally:\n        print("Finally, I executed!")\n    print(val)', 'askint()',
'askint()', 'def askint():\n    try:\n        val = int(input("Please enter an
integer: "))\n    except:\n        print("Looks like you did not enter an
integer!")\n        val = int(input("Try again-Please enter an integer: "))\n
finally:\n        print("Finally, I executed!")\n    print(val)', 'askint()',
'def askint():\n    while True:\n        try:\n            val =
int(input("Please enter an integer: "))\n        except:\n
print("Looks like you did not enter an integer!")\n            continue\n
else:\n            print("Yep that\'s an integer!")\n            break\n
finally:\n            print("Finally, I executed!")\n        print(val)',
'askint()', 'def askint():\n    while True:\n        try:\n            val =
int(input("Please enter an integer: "))\n        except:\n
print("Looks like you did not enter an integer!")\n            continue\n
else:\n            print("Yep that\'s an integer!")\n            print(val)\n
break\n        finally:\n            print("Finally, I executed!")', 'askint()',
'def fahrenheit(celsius):\n    return (9/5)*celsius + 32\n    \ntemps = [0,
22.5, 40, 100]', 'F_temps = map(fahrenheit, temps)\n\n#Show\nlist(F_temps)', 'a
```

```
= [1,2,3,4]\nb = [5,6,7,8]\nc = [9,10,11,12]\n\nlist(map(lambda x,y:x+y,a,b))',
'# Now all three lists\nlist(map(lambda x,y,z:x+y+z,a,b,c))', 'from functools
import reduce\n\nlst =[47,11,42,13]\nreduce(lambda x,y: x+y,lst)', "from
IPython.display import Image\nImage('http://www.python-
course.eu/images/reduce_diagram.png')", '#Find the maximum of a sequence (This
already exists as max())\nmax_find = lambda a,b: a if (a > b) else b', '#Find
max\nreduce(max_find,lst)', "#First let's make a function\ndef
even_check(num):\n    if num%2 ==0:\n        return True", 'lst
=range(20)\n\nlist(filter(even_check,lst))', 'list(filter(lambda x:
x%2==0,lst))', 'x = [1,2,3]\ny = [4,5,6]\n\n# Zip the lists
together\nlist(zip(x,y))', "lst = ['a','b','c']\nfor count,item in
enumerate(lst):\n    if count >= 2:\n        break\n    else:\n
print(item)", "months =
['March','April','May','June']\n\nlist(enumerate(months,start=3))", 'lst =
[True,True,False,True]', 'all(lst)', 'any(lst)', '# Create 2+3j\ncomplex(2,3)',
"complex('12+2j')", 'def func():\n    return 1\nfunc()', "s = 'Global
Variable'\n\ndef check_for_locals():\n    print(locals())", 'print(globals())',
'print(globals().keys())', "globals()['s']", 'check_for_locals()', "def
hello(name='Jose'):\n    return 'Hello '+name", 'hello()', 'greet = hello',
'greet', 'greet()', 'del hello', 'hello()', 'print(globals()[10])',
'print(globals()[:10])', 'print(globals())', '#print(globals())',
'print(globals().keys())', 'print(globals())'], 'Out': {1: 3, 2: 1, 3: 4, 4:
1.5, 5: 1, 6: 2, 7: 8, 9: 10, 13: 2, 15: ['Sammy', 'Frankie'], 16: 'hello', 17:
'Hello World', 18: 11, 20: 'H', 21: 'Hello Worl', 22: 'Hello World', 23:
'HloWrd', 24: 'dlroW olleH', 26: 'Hello World concatenate me!', 27: 'HELLO
WORLD', 28: 'hello world', 29: ['Hello', 'World'], 30: ['Hello ', 'orld'], 31:
'Insert another string with curly brackets: The inserted string', 43: 4, 45:
'one', 46: ['one', 'two', 'three', 4, 5, 'new item'], 47: ['one', 'two',
'three', 4, 5, 'one', 'two', 'three', 4, 5], 48: [2, 3, 'append me!'], 49: [2,
3, 'append me!'], 50: 1, 51: [2, 3, 'append me!'], 53: [[1, 2, 3], [4, 5, 6],
[7, 8, 9]], 55: [1, 4, 7], 56: 'value2', 58: 'value', 60: dict_keys(['key1',
'key2', 'key3']), 61: dict_values([1, 2, 3]), 62: dict_items([('key1', 1),
('key2', 2), ('key3', 3)]), 63: 3, 64: ('one', 2), 65: 0, 66: 1, 71: True, 72:
True, 73: True, 74: True, 75: True, 88: range(0, 11), 89: [0, 1, 2, 3, 4, 5, 6,
7, 8, 9, 10], 90: ([1, 3, 5, 7, 9], [0, 2, 4, 6, 8]), 91: [10, 9, 8, 7, 6, 5, 4,
3, 2, 1, 0, -1, -2, -3, -4, -5, -6, -7, -8, -9], 92: [(10, 'a'), (11, 'b'), (12,
'c'), (13, 'd')], 94: [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')], 95:
<zip object at 0x7f63344fa2d0>, 97: True, 98: False, 99: 10, 100: 100, 102: [20,
30, 40, 10, 100], 103: 'Hi,10', 104: ['w', 'o', 'r', 'd'], 106: [1, 2, 3, 4, 5,
6], 111: [1, 4, 9, 16, 25], 112: ['even', 'C', 'S', 'K', 'even'], 113: [0, 2, 4,
6, 8, 10], 121: <built-in function len>, 125: 5.0, 126: 6.0, 127: 6.0, 128: 6.0,
152: True, 161: 3, 167: <function hello.<locals>.greet at 0x7f63345344d0>, 172:
1, 176: 'Lab', 177: 'Huskie', 180: 'Sam', 181: 'mammal', 204: True, 213: 3, 236:
[32.0, 72.5, 104.0, 212.0], 237: [6, 8, 10, 12], 238: [15, 18, 21, 24], 239:
113, 240: <IPython.core.display.Image object>, 242: 47, 244: [0, 2, 4, 6, 8, 10,
12, 14, 16, 18], 245: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18], 246: [(1, 4), (2, 5),
(3, 6)], 248: [(3, 'March'), (4, 'April'), (5, 'May'), (6, 'June')], 250: False,
251: True, 252: (2+3j), 253: (12+2j), 254: 1, 258: 'Global Variable', 261:
```

```
'Hello Jose', 263: <function hello at 0x7f63340550e0>, 264: 'Hello Jose'},
'get_ipython': <bound method InteractiveShell.get_ipython of
<ipykernel.zmqshell.ZMQInteractiveShell object at 0x7f633820e950>>, 'exit':
<IPython.core.autocall.ZMQExitAutocall object at 0x7f6334645f90>, 'quit':
<IPython.core.autocall.ZMQExitAutocall object at 0x7f6334645f90>, '_': 'Hello
Jose', '__': <function hello at 0x7f63340550e0>, '___': 'Hello Jose', '_i':
'print(globals().keys())', '_ii': '#print(globals())', '_iii':
'print(globals())', '_i1': '# Addition\n2+1', '_1': 3, '_i2': '#
Subtraction\n2-1', '_2': 1, '_i3': '# Multiplication\n2*2', '_3': 4, '_i4': '#
Division\n3/2', '_4': 1.5, '_i5': '# Floor Division\n7//4', '_5': 1, '_i6': '#
Mod \n10 % 4', '_6': 2, '_i7': '# Powers\n2**3', '_7': 8, '_i8': '# Let\'s
create an object called "a" and assign it the number 5\na = 5', 'a': [1, 2, 3,
4], '_i9': '# Adding the objects\na+a', '_9': 10, '_i10': "# Use object names to
keep better track of what's going on in your code!\nmy_income = 100\n\ntax_rate
= 0.1\n\nmy_taxes = my_income*tax_rate", 'my_income': 100, 'tax_rate': 0.1,
'my_taxes': 10.0, '_i11': 'a = b = c = 1', 'b': [5, 6, 7, 8], 'c': [9, 10, 11,
12], '_i12': 'my_dogs = 2', 'my_dogs': ['Sammy', 'Frankie'], '_i13': 'my_dogs',
'_13': 2, '_i14': "my_dogs = ['Sammy', 'Frankie']", '_i15': 'my_dogs', '_15':
['Sammy', 'Frankie'], '_i16': "# Single word\n'hello'", '_16': 'hello', '_i17':
"# We can simply declare a string\n'Hello World'", '_17': 'Hello World', '_i18':
"len('Hello World')", '_18': 11, '_i19': "# Assign s as a string\ns = 'Hello
World'", 's': 'Global Variable', '_i20': '# Show first element (in this case a
letter)\ns[0]', '_20': 'H', '_i21': '# Grab everything but the last
letter\ns[:-1]', '_21': 'Hello Worl', '_i22': '# Grab everything, but go in
steps size of 1\ns[::1]', '_22': 'Hello World', '_i23': '# Grab everything, but
go in step sizes of 2\ns[::2]', '_23': 'HloWrd', '_i24': '# We can use this to
print a string backwards\ns[::-1]', '_24': 'dlroW olleH', '_i25': "# Let's try
to change the first letter to 'x'\n#s[0] = 'x'", '_i26': "# Concatenate
strings!\ns + ' concatenate me!'", '_26': 'Hello World concatenate me!', '_i27':
'# Upper Case a string\ns.upper()', '_27': 'HELLO WORLD', '_i28': '# Lower
case\ns.lower()', '_28': 'hello world', '_i29': '# Split a string by blank space
(this is the default)\ns.split()', '_29': ['Hello', 'World'], '_i30': "# Split
by a specific element (doesn't include the element that was split
on)\ns.split('W')", '_30': ['Hello ', 'orld'], '_i31': "'Insert another string
with curly brackets: {}'.format('The inserted string')", '_31': 'Insert another
string with curly brackets: The inserted string', '_i32': 'print("I\'m going to
inject %s text here, and %s text here." %(\'some\',\'more\'))', '_i33': 'x, y =
\'some\', \'more\'\nprint("I\'m going to inject %s text here, and %s text
here."%(x,y))', 'x': [1, 2, 3], 'y': [4, 5, 6], '_i34': "print('I once caught a
fish %s.' %'this \\tbig')\nprint('I once caught a fish %r.' %'this \\tbig')",
'_i35': "print('I wrote %s programs today.' %3.75)\nprint('I wrote %d programs
today.' %3.75)   ", '_i36': "print('Floating point numbers: %5.2f' %(13.144))",
'_i37': "print('Floating point numbers: %25.2f' %(13.144))", '_i38':
"print('{0:8} | {1:9}'.format('Fruit', 'Quantity'))\nprint('{0:8} |
{1:9}'.format('Apples', 3.))\nprint('{0:8} | {1:9}'.format('Oranges', 10))",
'_i39': 'name = \'Fred\'\n\nprint(f"He said his name is {name}.")', 'name':
'This is a global name', '_i40': 'print(f"He said his name is {name!r}")',
'_i41': '# Assign a list to an variable named my_list\nmy_list = [1,2,3]',
'my_list': ['one', 'two', 'three', 4, 5], '_i42': "my_list = ['A
string',23,100.232,'o']", '_i43': 'len(my_list)', '_43': 4, '_i44': "my_list =
['one','two','three',4,5]", '_i45': '# Grab element at index 0\nmy_list[0]',
'_45': 'one', '_i46': "my_list + ['new item']", '_46': ['one', 'two', 'three',
4, 5, 'new item'], '_i47': '# Make the list double\nmy_list * 2', '_47': ['one',
'two', 'three', 4, 5, 'one', 'two', 'three', 4, 5], '_i48': '# Create a new
list\nlist1 = [1,2,3]\nlist1', 'list1': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], '_48':
[2, 3, 'append me!'], '_i49': "# Append\nlist1.append('append me!')\nlist1",
'_49': [2, 3, 'append me!'], '_i50': '# Pop off the 0 indexed
item\nlist1.pop(0)', '_50': 1, '_i51': '# Show\nlist1', '_51': [2, 3, 'append
me!'], '_i52': "# Let's make three
lists\nlst_1=[1,2,3]\nlst_2=[4,5,6]\nlst_3=[7,8,9]\n\n# Make a list of lists to
form a matrix\nmatrix = [lst_1,lst_2,lst_3]", 'lst_1': [1, 2, 3], 'lst_2': [4,
5, 6], 'lst_3': [7, 8, 9], 'matrix': [[1, 2, 3], [4, 5, 6], [7, 8, 9]], '_i53':
'# Show\nmatrix', '_53': [[1, 2, 3], [4, 5, 6], [7, 8, 9]], '_i54': '# Build a
list comprehension by deconstructing a for loop within a []\nfirst_col = [row[0]
for row in matrix]', 'first_col': [1, 4, 7], '_i55': 'first_col', '_55': [1, 4,
7], '_i56': "# Make a dictionary with {} and : to signify a key and a
value\nmy_dict = {'key1':'value1','key2':'value2'}\n# Call values by their
key\nmy_dict['key2']", 'my_dict': {'key1': 'value1', 'key2': 'value2'}, '_56':
'value2', '_i57': "# Dictionary nested inside a dictionary nested inside a
dictionary\nd = {'key1':{'nestkey':{'subnestkey':'value'}}}", 'd': <__main__.Dog
object at 0x7f63344d9b50>, '_i58': "# Keep calling the
keys\nd['key1']['nestkey']['subnestkey']", '_58': 'value', '_i59': "# Create a
typical dictionary\nd = {'key1':1,'key2':2,'key3':3}", '_i60': '# Method to
return a list of all keys \nd.keys()', '_60': dict_keys(['key1', 'key2',
'key3']), '_i61': '# Method to grab all values\nd.values()', '_61':
dict_values([1, 2, 3]), '_i62': "# Method to return tuples of all items  (we'll
learn about tuples soon)\nd.items()", '_62': dict_items([('key1', 1), ('key2',
2), ('key3', 3)]), '_i63': '# Create a tuple\nt = (1,2,3)\n# Check len just like
a list\nlen(t)', 't': 5, '_63': 3, '_i64': "# Can also mix object types\nt =
('one',2)\n\n# Show\nt", '_64': ('one', 2), '_i65': "# Use .index to enter a
value and return the index\nt.index('one')", '_65': 0, '_i66': "# Use .count to
count the number of times a value appears\nt.count('one')", '_66': 1, '_i67':
"t[0]= 'change'", '_i68': '%%writefile test.txt\nHello, this is a quick test
file.', '_i69': "myfile = open('whoops.txt')", '_i70': '# Read the
file\nmy_file.seek(0)\nmy_file.read()', '_i71': '2 == 2', '_71': True, '_i72':
'2 <= 2', '_72': True, '_i73': '2 <= 4', '_73': True, '_i74': '1 < 2 < 3',
'_74': True, '_i75': '1<2 and 2<3', '_75': True, '_i76': "if True:\n
print('It was true!')", '_i77': "x = False\n\nif x:\n    print('x was
True!')\nelse:\n    print('I will be printed in any case where x is not true')",
'_i78': "# We'll learn how to automate this sort of list in the next
lecture\nlist1 = [1,2,3,4,5,6,7,8,9,10]\n\nfor num in list1:\n    print(num)",
'num': 10, '_i79': 'tup = (1,2,3,4,5)\n\nfor t in tup:\n    print(t)', 'tup':
(10, 12), '_i80': 'list2 = [(2,4),(6,8),(10,12)]', 'list2': [(2, 4), (6, 8),
(10, 12)], '_i81': 'for tup in list2:\n    print(tup)', '_i82': '# Now with
unpacking!\nfor (t1,t2) in list2:\n    print(t1)', 't1': 10, 't2': 12, '_i83':
"d = {'k1':1,'k2':2,'k3':3}", '_i84': 'for item in d:\n    print(item)', 'item':
```

```
'c', '_i85': "x = 0\n\nwhile x < 10:\n    print('x is currently: ',x)\n
print(' x is still less than 10, adding 1 to x')\n    x+=1", '_i86': "x =
0\n\nwhile x < 10:\n    print('x is currently: ',x)\n    print(' x is still less
than 10, adding 1 to x')\n    x+=1\n    \nelse:\n    print('All Done!')",
'_i87': "x = 0\n\nwhile x < 10:\n    print('x is currently: ',x)\n    print(' x
is still less than 10, adding 1 to x')\n    x+=1\n    if x==3:\n
print('Breaking because x==3')\n        break\n    else:\n
print('continuing…')\n        continue", '_i88': 'range(0,11)', '_88':
range(0, 11), '_i89': '# Notice how 11 is not included, up to but not including
11, just like slice notation!\nlist(range(0,11))', '_89': [0, 1, 2, 3, 4, 5, 6,
7, 8, 9, 10], '_i90': '# odd --------- even\nlist(range(1,10,2)),
list(range(0,10,2))', '_90': ([1, 3, 5, 7, 9], [0, 2, 4, 6, 8]), '_i91':
'list(range(10,-10,-1))', '_91': [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -2, -3,
-4, -5, -6, -7, -8, -9], '_i92': "x = ['a','b','c','d']\na= enumerate(x, start =
10)\nlist(a)", '_92': [(10, 'a'), (11, 'b'), (12, 'c'), (13, 'd')], '_i93':
'index_count = 0\n\nfor letter in \'abcde\':\n    print("At index {} the letter
is {}".format(index_count,letter))\n    index_count += 1', 'index_count': 5,
'letter': 'e', '_i94': "list(enumerate('abcde'))", '_94': [(0, 'a'), (1, 'b'),
(2, 'c'), (3, 'd'), (4, 'e')], '_i95': "mylist1 = [1,2,3,4,5]\nmylist2 =
['a','b','c','d','e']\n\n# This one is also a generator! We will explain this
later, but for now let's transform it to a list\nzip(mylist1,mylist2)",
'mylist1': [1, 2, 3, 4, 5], 'mylist2': ['a', 'b', 'c', 'd', 'e'], '_95': <zip
object at 0x7f63344fa2d0>, '_i96': '# Notice the tuple unpacking!\n\nfor
i,letter in enumerate(\'abcde\'):\n    print("At index {} the letter is
{}".format(i,letter))', 'i': 4, '_i97': "'x' in ['x','y','z']", '_97': True,
'_i98': "'x' in [1,2,3]", '_98': False, '_i99': 'mylist =
[10,20,30,40,100]\nmin(mylist)', 'mylist': [20, 30, 40, 10, 100], '_99': 10,
'_i100': 'max(mylist)', '_100': 100, '_i101': 'from random import shuffle',
'shuffle': <bound method Random.shuffle of <random.Random object at
0x5592e1b4add0>>, '_i102': '# This shuffles the list "in-place" meaning it
won\'t return\n# anything, instead it will effect the list
passed\nshuffle(mylist)\nmylist', '_102': [20, 30, 40, 10, 100], '_i103':
"input('Enter Something into this box: ')", '_103': 'Hi,10', '_i104': "# Grab
every letter in string\nlst = [x for x in 'word']\nlst", 'lst': [True, True,
False, True], '_104': ['w', 'o', 'r', 'd'], '_i105': '# Create a simple
list\nlst = [1,2,3,4,5]', '_i106': 'lst.append(6)\nlst', '_106': [1, 2, 3, 4, 5,
6], '_i107': 'help(lst.count)', '_i108': "def name_of_function(arg1,arg2):\n
'''\n    This is where the function's Document String (docstring) goes\n
'''\n    # Do stuff here\n    # Return desired result", 'name_of_function':
<function name_of_function at 0x7f63345b7ef0>, '_i109': "def is_prime(num):\n
'''\n    Naive method of checking for primes. \n    '''\n    for n in
range(2,num):\n        if num % n == 0:\n            print(num,'is not prime')\n
break\n    else: # If never mod zero, then prime\n        print(num,'is
prime!')", 'is_prime': <function is_prime at 0x7f63344f7200>, '_i110':
'is_prime(16)', '_i111': 'def square(num):\n    return num**2\nmy_nums =
[1,2,3,4,5]\nmap(square,my_nums)\n\n# To get the results, either iterate through
map() \n# or just cast to a list\nlist(map(square,my_nums))', 'square':
<function square at 0x7f633450e290>, 'my_nums': [1, 2, 3, 4, 5], '_111': [1, 4,
9, 16, 25], '_i112': "def splicer(mystring):\n    if len(mystring) % 2 == 0:\n
return 'even'\n    else:\n        return mystring[0]\n    \nmynames =
['John','Cindy','Sarah','Kelly','Mike']\nlist(map(splicer,mynames))", 'splicer':
<function splicer at 0x7f63344f7320>, 'mynames': ['John', 'Cindy', 'Sarah',
'Kelly', 'Mike'], '_112': ['even', 'C', 'S', 'K', 'even'], '_i113': 'def
check_even(num):\n    return num % 2 == 0 \n\nnums = [0,1,2,3,4,5,6,7,8,9,10]\nf
ilter(check_even,nums)\nlist(filter(check_even,nums))', 'check_even': <function
check_even at 0x7f63344f7710>, 'nums': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
'_113': [0, 2, 4, 6, 8, 10], '_i114': 'double = lambda x: x * 2\n\n# Output:
10\nprint(double(5))', 'double': <function <lambda> at 0x7f63344f7c20>, '_i115':
'x = 25\n\ndef printer():\n    x = 50\n    return x\n\n# print(x)\n#
print(printer())', 'printer': <function printer at 0x7f633450d200>, '_i116':
'print(x)', '_i117': 'print(printer())', '_i118': '# x is local here:\nf =
lambda x:x**2', 'f': <_io.TextIOWrapper name='testfile' mode='w'
encoding='UTF-8'>, '_i119': "name = 'This is a global name'\n\ndef greet():\n
# Enclosing function\n    name = 'Sammy'\n    \n    def hello():\n
print('Hello '+name)\n    \n    hello()\n\ngreet()", 'greet': <function hello at
0x7f63340550e0>, '_i120': 'print(name)', '_i121': 'len', '_121': <built-in
function len>, '_i122': "x = 50\n\ndef func(x):\n    print('x is', x)\n    x =
2\n    print('Changed local x to', x)\n\nfunc(x)\nprint('x is still', x)",
'func': <function func at 0x7f6334055830>, '_i123': "x = 50\n\ndef func():\n
global x\n    print('This function is now using the global x!')\n
print('Because of global x is: ', x)\n    x = 2\n    print('Ran func(), changed
global x to', x)\n\nprint('Before calling func(), x is: ',
x)\nfunc()\nprint('Value of x (outside of func()) is: ', x)", '_i124': 'def
greet(name, msg = "Good morning!"):\n    print("Hello",name + \', \' +
msg)\n\ngreet("Kate")\ngreet("Bruce","How do you do?")', '_i125': 'def
myfunc(a,b):\n    return sum((a,b))*.05\n\nmyfunc(40,60)', 'myfunc': <function
myfunc at 0x7f633450ddd0>, '_125': 5.0, '_i126': 'def
myfunc(a=0,b=0,c=0,d=0,e=0):\n    return
sum((a,b,c,d,e))*.05\n\nmyfunc(40,60,20)', '_126': 6.0, '_i127': 'def
myfunc(*args):\n    return sum(args)*.05\n\nmyfunc(40,60,20)', '_127': 6.0,
'_i128': 'def myfunc(*spam):\n    return sum(spam)*.05\n\nmyfunc(40,60,20)',
'_128': 6.0, '_i129': 'def myfunc(**kwargs):\n    if \'fruit\' in kwargs:\n
print(f"My favorite fruit is {kwargs[\'fruit\']}")  # review String Formatting
and f-strings if this syntax is unfamiliar\n    else:\n        print("I don\'t
like fruit")\n        \nmyfunc(fruit=\'pineapple\')', '_i130': 'myfunc()',
'_i131': 'def myfunc(*args, **kwargs):\n    if \'fruit\' and \'juice\' in
kwargs:\n        print(f"I like {\' and \'.join(args)} and my favorite fruit is
{kwargs[\'fruit\']}")\n        print(f"May I have some {kwargs[\'juice\']}
juice?")\n    else:\n        pass\n
\nmyfunc(\'eggs\',\'spam\',fruit=\'cherries\',juice=\'orange\')', '_i132':
"myfunc(fruit='cherries',juice='orange','eggs','spam')", '_i133': '# first class
function\ndef square(x):\n    return x*x\n\ndef cube(x):\n    return
x*x*x\n\ndef my_map(func, arg_list):\n    result = []\n    for i in arg_list:\n
result.append(func(i))\n    return result\n\nsquares =
my_map(cube,[1,2,3,4,5])\n\nprint(squares)', 'cube': <function cube at
0x7f633450e3b0>, 'my_map': <function my_map at 0x7f633450e4d0>, 'squares': [1,
8, 27, 64, 125], '_i134': "def outer_func():\n    message = 'Hi'\n\n    def
innner_func():\n        print(message)\n    return
innner_func()\n\nouter_func()\nouter_func()\nouter_func()", 'outer_func':
<function outer_func at 0x7f633461a7a0>, '_i135': "def outer_func(msg):\n
message = msg\n\n    def innner_func():\n        print(message)\n    return
innner_func\n\nhi_func = outer_func('Hi')\nbye_func =
outer_func('Bye')\n\nhi_func()\nbye_func()", 'hi_func': <function
outer_func.<locals>.innner_func at 0x7f633461a290>, 'bye_func': <function
outer_func.<locals>.innner_func at 0x7f633461a8c0>, '_i136': 'def
make_counter():\n\n    count = 0\n    def inner():\n\n        nonlocal count\n
count += 1\n        return count\n\n    return inner\n\n\ncounter =
make_counter()\n\nc = counter()\nprint(c)\n\nc = counter()\nprint(c)\n\nc =
counter()\nprint(c)', 'make_counter': <function make_counter at 0x7f63344f7050>,
'counter': <function make_counter.<locals>.inner at 0x7f63344f7170>, '_i137':
'def hello(name=\'Jose\'):\n    print(\'The hello() function has been
executed\')\n    \n    def greet():\n        return \'\\t This is inside the
greet() function\'\n    \n    def welcome():\n        return "\\t This is inside
the welcome() function"\n    \n    print(greet())\n    print(welcome())\n
print("Now we are back inside the hello() function")', '_i138': 'hello()',
'_i139': 'welcome()', '_i140': "class Dog:\n    def __init__(self, name):\n
self.name = name\n\n    def speak(self):\n        return self.name+' says
Woof!'\n    \nclass Cat:\n    def __init__(self, name):\n        self.name =
name\n\n    def speak(self):\n        return self.name+' says Meow!' \n
\nniko = Dog('Niko')\nfelix =
Cat('Felix')\n\nprint(niko.speak())\nprint(felix.speak())", 'Dog': <class
'__main__.Dog'>, 'Cat': <class '__main__.Cat'>, 'niko': <__main__.Dog object at
0x7f633403d9d0>, 'felix': <__main__.Cat object at 0x7f633403d990>, '_i141': 'for
pet in [niko,felix]:\n    print(pet.speak())', 'pet': <__main__.Cat object at
0x7f633403d990>, '_i142': 'def pet_speak(pet):\n
print(pet.speak())\n\npet_speak(niko)\npet_speak(felix)', 'pet_speak': <function
pet_speak at 0x7f63344f6170>, '_i143': 'class Animal:\n    def __init__(self,
name):    # Constructor of the class\n        self.name = name\n\n    def
speak(self):              # Abstract method, defined by convention only\n
raise NotImplementedError("Subclass must implement abstract method")\n\n\nclass
Dog(Animal):\n    \n    def speak(self):\n        return self.name+\' says
Woof!\'\n    \nclass Cat(Animal):\n\n    def speak(self):\n        return
self.name+\' says Meow!\'\n    \nfido = Dog(\'Fido\')\nisis =
Cat(\'Isis\')\n\nprint(fido.speak())\nprint(isis.speak())', 'Animal': <class
'__main__.Animal'>, 'fido': <__main__.Dog object at 0x7f633403c090>, 'isis':
<__main__.Cat object at 0x7f633403dc50>, '_i144': 'class Computer:\n\n    def
__init__(self):\n        self.__maxprice = 900\n\n    def sell(self):\n
print("Selling Price: {}".format(self.__maxprice))\n\n    def setMaxPrice(self,
price):\n        self.__maxprice = price\n\nc = Computer()\nc.sell()',
'Computer': <class '__main__.Computer'>, '_i145': '# change the
price\nc.__maxprice = 1000\nc.sell()', '_i146': '# using setter
function\nc.setMaxPrice(1000)\nc.sell()', '_i147': 'class Book:\n    def
__init__(self, title, author, pages):\n        print("A book is created")\n
self.title = title\n        self.author = author\n        self.pages = pages\n\n
```

```
def __str__(self):\n        return "Title: %s, author: %s, pages: %s"
%(self.title, self.author, self.pages)\n\n    def __len__(self):\n        return
self.pages\n\n    def __del__(self):\n        print("A book is destroyed")',
'Book': <class '__main__.Book'>, '_i148': 'book = Book("Python Rocks!", "Jose
Portilla", 159)\n\n#Special Methods\nprint(book)\nprint(len(book))\ndel book',
'_i149': 'class Employee():\n    \n    raise_amount =1.04\n    def
__init__(self,first,last,pay):\n        self.first  = first\n        self.last =
last\n        self.pay = pay\n        self.email =
first+\'.\'+last+\'@company.com\'\n\n    def fullname(self):\n        return
self.first+\' \'+self.last\n    \n    def apply_raise(self):\n        self.pay =
int(self.pay * self.raise_amount)\n        return self.pay\n    \n
@classmethod\n    def set_raise_amount(cls, amount):\n        cls.raise_amount =
amount\n    \n    @classmethod\n    def from_string(cls,emp_str):\n        "
class method used as constructor"\n        first,last,pay =
emp_str.split(\'-\')\n        return cls(first,last,pay)\n    \n
@staticmethod\n    def is_workday(day):\n        if day.weekday() ==5 or
day.weekday() ==6:\n            return False\n        return True', 'Employee':
<class '__main__.Employee'>, '_i150': "emp_1 =
Employee('Sudhir','Kumar',2000)\nemp_2 = Employee('Latha','Shet',4000)\n\nprint(
emp_1.raise_amount)\nEmployee.set_raise_amount(1.1)\nprint(emp_1.raise_amount)",
'emp_1': <__main__.Employee object at 0x7f633403d290>, 'emp_2':
<__main__.Employee object at 0x7f633403db90>, '_i151': "# class method used as
constructor\nemp_str_1 = 'John-Doe-2000'\nnew_emp =
Employee.from_string(emp_str_1)\nprint(new_emp.fullname())", 'emp_str_1': 'John-
Doe-2000', 'new_emp': <__main__.Employee object at 0x7f63344cddd0>, '_i152':
'import datetime\nmy_day =
datetime.date(2018,11,21)\nEmployee.is_workday(my_day)', 'datetime': <module
'datetime' from '/home/sudhir/anaconda3/lib/python3.7/datetime.py'>, 'my_day':
datetime.date(2018, 11, 21), '_152': True, '_i153': '# Suppose this is
foo.py.\n\nprint("before import")\nimport math\n\nprint("before functionA")\ndef
functionA():\n    print("Function A")\n\nprint("before functionB")\ndef
functionB():\n    print("Function B {}".format(math.sqrt(100)))\n\nprint("before
__name__ guard")\nif __name__ == \'__main__\':\n    functionA()\n
functionB()\nprint("after __name__ guard")', 'math': <module 'math' from
'/home/sudhir/anaconda3/lib/python3.7/lib-dynload/math.cpython-37m-x86_64-linux-
gnu.so'>, 'functionA': <function functionA at 0x7f633451c8c0>, 'functionB':
<function functionB at 0x7f633451c200>, '_i154': '%%writefile file1.py\ndef
myfunc(x):\n    return [num for num in range(x) if num%2==0]\nlist1 =
myfunc(11)', '_i155': '%%writefile file2.py\nimport
file1\nfile1.list1.append(12)\nprint(file1.list1)', '_i156': '! python
file2.py', '_exit_code': 0, '_i157': 'import file1\nprint(file1.list1)',
'file1': <module 'file1' from '/home/sudhir/Downloads/AI/Courses/Complete Python
Bootcamp: Go from zero to hero in Python/file1.py'>, '_i158': '%%writefile
file3.py\nimport sys\nimport file1\nnum =
int(sys.argv[1])\nprint(file1.myfunc(num))', '_i159': '! python file3.py 21',
'_i160': '# import the library\nimport math', '_i161': '# use it (ceiling
rounding)\nmath.ceil(2.4)', '_161': 3, '_i162': 'print(dir(math))', '_i163':
'help(math.ceil)', '_i164': "# Just an example, this won't work\nimport
```

```
foo.bar", '_i165': 'def hello(name=\'Jose\'):\n    \n    def greet():\n
return \'\\t This is inside the greet() function\'\n    \n    def welcome():\n
return "\\t This is inside the welcome() function"\n    \n    if name ==
\'Jose\':\n        return greet\n    else:\n        return welcome', '_i166': 'x
= hello()', '_i167': 'x', '_167': <function hello.<locals>.greet at
0x7f63345344d0>, '_i168': 'print(x())', '_i169': "def hello():\n    return 'Hi
Jose!'\n\ndef other(func):\n    print('Other code would go here')\n
print(func())", 'other': <function other at 0x7f6334534440>, '_i170':
'other(hello)', '_i171': 'lst = [1,2,3]', '_i172': 'lst.count(2)', '_172': 1,
'_i173': 'print(type(1))\nprint(type([]))\nprint(type(()))\nprint(type({}))',
'_i174': '# Create a new object type called Sample\nclass Sample:\n    pass\n\n#
Instance of Sample\nx = Sample()\n\nprint(type(x))', 'Sample': <class
'__main__.Sample'>, '_i175': "class Dog:\n    def __init__(self,breed):\n
self.breed = breed\n        \nsam = Dog(breed='Lab')\nfrank =
Dog(breed='Huskie')", 'sam': <__main__.Dog object at 0x7f63344dfb10>, 'frank':
<__main__.Dog object at 0x7f63344d93d0>, '_i176': 'sam.breed', '_176': 'Lab',
'_i177': 'frank.breed', '_177': 'Huskie', '_i178': "class Dog:\n    \n    #
Class Object Attribute\n    species = 'mammal'\n    \n    def
__init__(self,breed,name):\n        self.breed = breed\n        self.name =
name", '_i179': "sam = Dog('Lab','Sam')", '_i180': 'sam.name', '_180': 'Sam',
'_i181': 'sam.species', '_181': 'mammal', '_i182': "class Employee():\n    \n
raise_amount =1.04\n    def __init__(self,first,last,pay):\n        self.first
= first\n        self.last = last\n        self.pay = pay\n        self.email =
first+'.'+last+'@company.com'\n\n    def fullname(self):\n        return
self.first+' '+self.last\n    \n    def apply_raise(self):\n        self.pay =
int(self.pay * self.raise_amount)\n        return self.pay", '_i183': "emp_1 =
Employee('Sudhir','Kumar',200)\nemp_2 = Employee('Latha','Shet',200)\n\nprint(em
p_1.__dict__)\nprint(emp_2.fullname())\nprint(emp_1.apply_raise())", '_i184':
'print(emp_1.first)\nprint(emp_1.email)', '_i185': "class Circle:\n    pi =
3.14\n\n    # Circle gets instantiated with a radius (default is 1)\n    def
__init__(self, radius=1):\n        self.radius = radius \n        self.area =
radius * radius * Circle.pi\n\n    # Method for resetting Radius\n    def
setRadius(self, new_radius):\n        self.radius = new_radius\n
self.area = new_radius * new_radius * self.pi\n\n    # Method for getting
Circumference\n    def getCircumference(self):\n        return self.radius *
self.pi * 2\n\n\nc = Circle()\n\nprint('Radius is: ',c.radius)\nprint('Area is:
',c.area)\nprint('Circumference is: ',c.getCircumference())", 'Circle': <class
'__main__.Circle'>, '_i186': "c.setRadius(2)\n\nprint('Radius is:
',c.radius)\nprint('Area is: ',c.area)\nprint('Circumference is:
',c.getCircumference())", '_i187': 'class Animal:\n    def __init__(self):\n
print("Animal created")\n\n    def whoAmI(self):\n        print("Animal")\n\n
def eat(self):\n        print("Eating")\n\n\nclass Dog(Animal):\n    def
__init__(self):\n        Animal.__init__(self)\n        print("Dog created")\n\n
def whoAmI(self):\n        print("Dog")\n\n    def bark(self):\n
print("Woof!")', '_i188': 'd = Dog()', '_i189': 'd.whoAmI()', '_i190':
'd.eat()', '_i191': 'd.bark()', '_i192': "class Dog:\n    def __init__(self,
name):\n        self.name = name\n\n    def speak(self):\n        return
self.name+' says Woof!'\n    \nclass Cat:\n    def __init__(self, name):\n
```

self.name = name\n\n   def speak(self):\n   return self.name+' says Meow!'
\n   \nniko = Dog('Niko')\nfelix =
Cat('Felix')\n\nprint(niko.speak())\nprint(felix.speak())", '_i193': 'for pet in
[niko,felix]:\n   print(pet.speak())', '_i194': 'def pet_speak(pet):\n
print(pet.speak())\n\npet_speak(niko)\npet_speak(felix)', '_i195': 'class
Animal:\n   def __init__(self, name):   # Constructor of the class\n
self.name = name\n\n   def speak(self):   # Abstract method, defined
by convention only\n   raise NotImplementedError("Subclass must implement
abstract method")\n\n\nclass Dog(Animal):\n   \n   def speak(self):\n
return self.name+\' says Woof!\'\n   \nclass Cat(Animal):\n\n   def
speak(self):\n   return self.name+\' says Meow!\'\n   \nfido =
Dog(\'Fido\')\nisis =
Cat(\'Isis\')\n\nprint(fido.speak())\nprint(isis.speak())', '_i196': 'class
Computer:\n\n   def __init__(self):\n   self.__maxprice = 900\n\n   def
sell(self):\n   print("Selling Price: {}".format(self.__maxprice))\n\n
def setMaxPrice(self, price):\n   self.__maxprice = price\n\nc =
Computer()\nc.sell()', '_i197': '# change the price\nc.__maxprice =
1000\nc.sell()', '_i198': '# using setter
function\nc.setMaxPrice(1000)\nc.sell()', '_i199': 'class Book:\n   def
__init__(self, title, author, pages):\n   print("A book is created")\n
self.title = title\n   self.author = author\n   self.pages = pages\n\n
def __str__(self):\n   return "Title: %s, author: %s, pages: %s"
%(self.title, self.author, self.pages)\n\n   def __len__(self):\n   return
self.pages\n\n   def __del__(self):\n   print("A book is destroyed")',
'_i200': 'book = Book("Python Rocks!", "Jose Portilla", 159)\n\n#Special
Methods\nprint(book)\nprint(len(book))\ndel book', '_i201': 'class Employee():\n
\n   raise_amount =1.04\n   def __init__(self,first,last,pay):\n
self.first = first\n   self.last = last\n   self.pay = pay\n
self.email = first+\'.\'+last+\'@company.com\'\n\n   def fullname(self):\n
return self.first+\' \'+self.last\n   \n   def apply_raise(self):\n
self.pay = int(self.pay * self.raise_amount)\n   return self.pay\n   \n
@classmethod\n   def set_raise_amount(cls, amount):\n   cls.raise_amount =
amount\n   \n   @classmethod\n   def from_string(cls,emp_str):\n   "
class method used as constructor"\n   first,last,pay =
emp_str.split(\'-\')\n   return cls(first,last,pay)\n   \n
@staticmethod\n   def is_workday(day):\n   if day.weekday() ==5 or
day.weekday() ==6:\n   return False\n   return True', '_i202':
"emp_1 = Employee('Sudhir','Kumar',2000)\nemp_2 = Employee('Latha','Shet',4000)\
n\nprint(emp_1.raise_amount)\nEmployee.set_raise_amount(1.1)\nprint(emp_1.raise_
amount)", '_i203': "# class method used as constructor\nemp_str_1 = 'John-
Doe-2000'\nnew_emp =
Employee.from_string(emp_str_1)\nprint(new_emp.fullname())", '_i204': 'import
datetime\nmy_day = datetime.date(2018,11,21)\nEmployee.is_workday(my_day)',
'_204': True, '_i205': '# Suppose this is foo.py.\n\nprint("before
import")\nimport math\n\nprint("before functionA")\ndef functionA():\n
print("Function A")\n\nprint("before functionB")\ndef functionB():\n
print("Function B {}".format(math.sqrt(100)))\n\nprint("before __name__
guard")\nif __name__ == \'__main__\':\n   functionA()\n

```
functionB()\nprint("after __name__ guard")', '_i206': '%%writefile file1.py\ndef
myfunc(x):\n    return [num for num in range(x) if num%2==0]\nlist1 =
myfunc(11)', '_i207': '%%writefile file2.py\nimport
file1\nfile1.list1.append(12)\nprint(file1.list1)', '_i208': '! python
file2.py', '_i209': 'import file1\nprint(file1.list1)', '_i210': '%%writefile
file3.py\nimport sys\nimport file1\nnum =
int(sys.argv[1])\nprint(file1.myfunc(num))', '_i211': '! python file3.py 21',
'_i212': '# import the library\nimport math', '_i213': '# use it (ceiling
rounding)\nmath.ceil(2.4)', '_213': 3, '_i214': 'print(dir(math))', '_i215':
'help(math.ceil)', '_i216': "# Just an example, this won't work\nimport
foo.bar", '_i217': '# OR could do it this way\nfrom foo import bar', '_i218':
'ls', '_i219': '__init__.py:\n\n__all__ = ["bar"]', '_i220': "print('Hello)",
'_i221': "print('Hello')", '_i222': 'try:\n    f = open(\'testfile\',\'w\')\n
f.write(\'Test write this\')\nexcept IOError:\n    # This will only check for an
IOError exception and then execute this print statement\n    print("Error: Could
not find file or read data")\nelse:\n    print("Content written successfully")\n
f.close()', '_i223': 'try:\n    f = open(\'testfile\',\'r\')\n    f.write(\'Test
write this\')\nexcept IOError:\n    # This will only check for an IOError
exception and then execute this print statement\n    print("Error: Could not
find file or read data")\nelse:\n    print("Content written successfully")\n
f.close()', '_i224': 'try:\n    f = open(\'testfile\',\'r\')\n    f.write(\'Test
write this\')\nexcept:\n    # This will check for any exception and then execute
this print statement\n    print("Error: Could not find file or read
data")\nelse:\n    print("Content written successfully")\n    f.close()',
'_i225': 'try:\n    f = open("testfile", "w")\n    f.write("Test write
statement")\n    f.close()\nfinally:\n    print("Always execute finally code
blocks")', '_i226': 'def askint():\n    try:\n        val = int(input("Please
enter an integer: "))\n    except:\n        print("Looks like you did not enter
an integer!")\n\n    finally:\n        print("Finally, I executed!")\n
print(val)', 'askint': <function askint at 0x7f63344fd680>, '_i227': 'askint()',
'_i228': 'askint()', '_i229': 'def askint():\n    try:\n        val =
int(input("Please enter an integer: "))\n    except:\n        print("Looks like
you did not enter an integer!")\n        val = int(input("Try again-Please enter
an integer: "))\n    finally:\n        print("Finally, I executed!")\n
print(val)', '_i230': 'askint()', '_i231': 'def askint():\n    while True:\n
try:\n            val = int(input("Please enter an integer: "))\n
except:\n            print("Looks like you did not enter an integer!")\n
continue\n        else:\n            print("Yep that\'s an integer!")\n
break\n        finally:\n            print("Finally, I executed!")\n
print(val)', '_i232': 'askint()', '_i233': 'def askint():\n    while True:\n
try:\n            val = int(input("Please enter an integer: "))\n
except:\n            print("Looks like you did not enter an integer!")\n
continue\n        else:\n            print("Yep that\'s an integer!")\n
print(val)\n            break\n        finally:\n            print("Finally, I
executed!")', '_i234': 'askint()', '_i235': 'def fahrenheit(celsius):\n
return (9/5)*celsius + 32\n    \ntemps = [0, 22.5, 40, 100]', 'fahrenheit':
<function fahrenheit at 0x7f63344fd710>, 'temps': [0, 22.5, 40, 100], '_i236':
'F_temps = map(fahrenheit, temps)\n\n#Show\nlist(F_temps)', 'F_temps': <map
```

```
object at 0x7f63344d2750>, '_236': [32.0, 72.5, 104.0, 212.0], '_i237': 'a =
[1,2,3,4]\nb = [5,6,7,8]\nc = [9,10,11,12]\n\nlist(map(lambda x,y:x+y,a,b))',
'_237': [6, 8, 10, 12], '_i238': '# Now all three lists\nlist(map(lambda
x,y,z:x+y+z,a,b,c))', '_238': [15, 18, 21, 24], '_i239': 'from functools import
reduce\n\nlst =[47,11,42,13]\nreduce(lambda x,y: x+y,lst)', 'reduce': <built-in
function reduce>, '_239': 113, '_i240': "from IPython.display import
Image\nImage('http://www.python-course.eu/images/reduce_diagram.png')", 'Image':
<class 'IPython.core.display.Image'>, '_240': <IPython.core.display.Image
object>, '_i241': '#Find the maximum of a sequence (This already exists as
max())\nmax_find = lambda a,b: a if (a > b) else b', 'max_find': <function
<lambda> at 0x7f633403eb90>, '_i242': '#Find max\nreduce(max_find,lst)', '_242':
47, '_i243': "#First let's make a function\ndef even_check(num):\n    if num%2
==0:\n        return True", 'even_check': <function even_check at
0x7f63344f1f80>, '_i244': 'lst =range(20)\n\nlist(filter(even_check,lst))',
'_244': [0, 2, 4, 6, 8, 10, 12, 14, 16, 18], '_i245': 'list(filter(lambda x:
x%2==0,lst))', '_245': [0, 2, 4, 6, 8, 10, 12, 14, 16, 18], '_i246': 'x =
[1,2,3]\ny = [4,5,6]\n\n# Zip the lists together\nlist(zip(x,y))', '_246': [(1,
4), (2, 5), (3, 6)], '_i247': "lst = ['a','b','c']\nfor count,item in
enumerate(lst):\n    if count >= 2:\n        break\n    else:\n
print(item)", 'count': 2, '_i248': "months =
['March','April','May','June']\n\nlist(enumerate(months,start=3))", 'months':
['March', 'April', 'May', 'June'], '_248': [(3, 'March'), (4, 'April'), (5,
'May'), (6, 'June')], '_i249': 'lst = [True,True,False,True]', '_i250':
'all(lst)', '_250': False, '_i251': 'any(lst)', '_251': True, '_i252': '# Create
2+3j\ncomplex(2,3)', '_252': (2+3j), '_i253': "complex('12+2j')", '_253':
(12+2j), '_i254': 'def func():\n    return 1\nfunc()', '_254': 1, '_i255': "s =
'Global Variable'\n\ndef check_for_locals():\n    print(locals())",
'check_for_locals': <function check_for_locals at 0x7f6334055cb0>, '_i256':
'print(globals())', '_i257': 'print(globals().keys())', '_i258':
"globals()['s']", '_258': 'Global Variable', '_i259': 'check_for_locals()',
'_i260': "def hello(name='Jose'):\n    return 'Hello '+name", '_i261':
'hello()', '_261': 'Hello Jose', '_i262': 'greet = hello', '_i263': 'greet',
'_263': <function hello at 0x7f63340550e0>, '_i264': 'greet()', '_264': 'Hello
Jose', '_i265': 'del hello', '_i266': 'hello()', '_i267':
'print(globals()[10])', '_i268': 'print(globals()[:10])', '_i269':
'print(globals())', '_i270': '#print(globals())', '_i271':
'print(globals().keys())', '_i272': 'print(globals())'}
```

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:

[271]:
```
print(globals().keys())
```

```
dict_keys(['__name__', '__doc__', '__package__', '__loader__', '__spec__',
'__builtin__', '__builtins__', '_ih', '_oh', '_dh', 'In', 'Out', 'get_ipython',
'exit', 'quit', '_', '__', '___', '_i', '_ii', '_iii', '_i1', '_1', '_i2', '_2',
'_i3', '_3', '_i4', '_4', '_i5', '_5', '_i6', '_6', '_i7', '_7', '_i8', 'a',
'_i9', '_9', '_i10', 'my_income', 'tax_rate', 'my_taxes', '_i11', 'b', 'c',
'_i12', 'my_dogs', '_i13', '_13', '_i14', '_i15', '_15', '_i16', '_16', '_i17',
'_17', '_i18', '_18', '_i19', 's', '_i20', '_20', '_i21', '_21', '_i22', '_22',
'_i23', '_23', '_i24', '_24', '_i25', '_i26', '_26', '_i27', '_27', '_i28',
'_28', '_i29', '_29', '_i30', '_30', '_i31', '_31', '_i32', '_i33', 'x', 'y',
'_i34', '_i35', '_i36', '_i37', '_i38', '_i39', 'name', '_i40', '_i41',
'my_list', '_i42', '_i43', '_43', '_i44', '_i45', '_45', '_i46', '_46', '_i47',
'_47', '_i48', 'list1', '_48', '_i49', '_49', '_i50', '_50', '_i51', '_51',
'_i52', 'lst_1', 'lst_2', 'lst_3', 'matrix', '_i53', '_53', '_i54', 'first_col',
'_i55', '_55', '_i56', 'my_dict', '_56', '_i57', 'd', '_i58', '_58', '_i59',
'_i60', '_60', '_i61', '_61', '_i62', '_62', '_i63', 't', '_63', '_i64', '_64',
'_i65', '_65', '_i66', '_66', '_i67', '_i68', '_i69', '_i70', '_i71', '_71',
'_i72', '_72', '_i73', '_73', '_i74', '_74', '_i75', '_75', '_i76', '_i77',
'_i78', 'num', '_i79', 'tup', '_i80', 'list2', '_i81', '_i82', 't1', 't2',
'_i83', '_i84', 'item', '_i85', '_i86', '_i87', '_i88', '_88', '_i89', '_89',
'_i90', '_90', '_i91', '_91', '_i92', '_92', '_i93', 'index_count', 'letter',
'_i94', '_94', '_i95', 'mylist1', 'mylist2', '_95', '_i96', 'i', '_i97', '_97',
'_i98', '_98', '_i99', 'mylist', '_99', '_i100', '_100', '_i101', 'shuffle',
'_i102', '_102', '_i103', '_103', '_i104', 'lst', '_104', '_i105', '_i106',
'_106', '_i107', '_i108', 'name_of_function', '_i109', 'is_prime', '_i110',
'_i111', 'square', 'my_nums', '_111', '_i112', 'splicer', 'mynames', '_112',
'_i113', 'check_even', 'nums', '_113', '_i114', 'double', '_i115', 'printer',
'_i116', '_i117', '_i118', 'f', '_i119', 'greet', '_i120', '_i121', '_121',
'_i122', 'func', '_i123', '_i124', '_i125', 'myfunc', '_125', '_i126', '_126',
'_i127', '_127', '_i128', '_128', '_i129', '_i130', '_i131', '_i132', '_i133',
'cube', 'my_map', 'squares', '_i134', 'outer_func', '_i135', 'hi_func',
'bye_func', '_i136', 'make_counter', 'counter', '_i137', '_i138', '_i139',
'_i140', 'Dog', 'Cat', 'niko', 'felix', '_i141', 'pet', '_i142', 'pet_speak',
'_i143', 'Animal', 'fido', 'isis', '_i144', 'Computer', '_i145', '_i146',
'_i147', 'Book', '_i148', '_i149', 'Employee', '_i150', 'emp_1', 'emp_2',
'_i151', 'emp_str_1', 'new_emp', '_i152', 'datetime', 'my_day', '_152', '_i153',
'math', 'functionA', 'functionB', '_i154', '_i155', '_i156', '_exit_code',
'_i157', 'file1', '_i158', '_i159', '_i160', '_i161', '_161', '_i162', '_i163',
'_i164', '_i165', '_i166', '_i167', '_167', '_i168', '_i169', 'other', '_i170',
'_i171', '_i172', '_172', '_i173', '_i174', 'Sample', '_i175', 'sam', 'frank',
'_i176', '_176', '_i177', '_177', '_i178', '_i179', '_i180', '_180', '_i181',
'_181', '_i182', '_i183', '_i184', '_i185', 'Circle', '_i186', '_i187', '_i188',
'_i189', '_i190', '_i191', '_i192', '_i193', '_i194', '_i195', '_i196', '_i197',
'_i198', '_i199', '_i200', '_i201', '_i202', '_i203', '_i204', '_204', '_i205',
'_i206', '_i207', '_i208', '_i209', '_i210', '_i211', '_i212', '_i213', '_213',
'_i214', '_i215', '_i216', '_i217', '_i218', '_i219', '_i220', '_i221', '_i222',
'_i223', '_i224', '_i225', '_i226', 'askint', '_i227', '_i228', '_i229',
'_i230', '_i231', '_i232', '_i233', '_i234', '_i235', 'fahrenheit', 'temps',
'_i236', 'F_temps', '_236', '_i237', '_237', '_i238', '_238', '_i239', 'reduce',
'_239', '_i240', 'Image', '_240', '_i241', 'max_find', '_i242', '_242', '_i243',
'even_check', '_i244', '_244', '_i245', '_245', '_i246', '_246', '_i247',
'count', '_i248', 'months', '_248', '_i249', '_i250', '_250', '_i251', '_251',
'_i252', '_252', '_i253', '_253', '_i254', '_254', '_i255', 'check_for_locals',
'_i256', '_i257', '_i258', '_258', '_i259', '_i260', '_i261', '_261', '_i262',
'_i263', '_263', '_i264', '_264', '_i265', '_i266', '_i267', '_i268', '_i269',
'_i270', '_i271'])
```

Note how **s** is there, the Global Variable we defined as a string:

```
[273]: globals()['s']
```

```
[273]: 'Global Variable'
```

Now let's run our function to check for local variables that might exist inside our function (there shouldn't be any)

```
[274]: check_for_locals()
```

```
{}
```

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:

```
[275]: def hello(name='Jose'):
           return 'Hello '+name
```

```
[261]: hello()
```

```
[261]: 'Hello Jose'
```

Assign another label to the function. Note that we are not using parentheses here because we are not calling the function **hello**, instead we are just passing a function object to the **greet** variable.

```
[262]: greet = hello
```

```
[263]: greet
```

```
[263]: <function __main__.hello(name='Jose')>
```

```
[264]: greet()
```

```
[264]: 'Hello Jose'
```

So what happens when we delete the name **hello**?

```
[265]: del hello
```

```
[266]: hello()
```

```
        ␣
↪---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call␣
↪last)

        <ipython-input-266-a75d7781aaeb> in <module>
    ----> 1 hello()


        NameError: name 'hello' is not defined
```

[276]:
```
greet()
```

[276]:
```
'Hello Jose'
```

Even though we deleted the name **hello**, the name **greet** *still points to* our original function object. It is important to know that functions are objects that can be passed to other objects!

8.1 Creating a Decorator

In the previous example we actually manually created a Decorator. Here we will modify it to make its use case clear:

[277]:
```python
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")
```

[278]:
```
func_needs_decorator()
```

```
This function is in need of a Decorator
```

[279]:
```python
# Reassign func_needs_decorator
func_needs_decorator = new_decorator(func_needs_decorator)
```

```
[280]: 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 simply wrapped the function and modified its behavior. Now let's understand how we can rewrite this code using the @ symbol, which is what Python uses for Decorators:

```
[281]: @new_decorator
       def func_needs_decorator():
           print("This function is in need of a Decorator")
```

```
[282]: 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!

```
[283]: def smart_divide(func):
           def inner(a,b):
               print("I am going to divide",a,"and",b)
               if b == 0:
                   print("Whoops! cannot divide")
                   return
               return func(a,b)
           return inner

       @smart_divide
       def divide(a,b):
           return a/b
```

```
[284]: divide(2,5)
```

```
I am going to divide 2 and 5
```

```
[284]: 0.4
```

```
[285]: divide(2,0)
```

```
I am going to divide 2 and 0
Whoops! cannot divide
```

8.2 Iterators and Generators

In this section of the course we will be learning 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 touched on this topic in the past when discussing certain built-in Python functions like **range()**, **map()** and **filter()**.

Let's explore a little deeper. 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 supports an iteration protocol. That means when they are called in your code they don't actually return a value and then exit. Instead, 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 up front, the generator computes one value and then suspends its activity awaiting the next instruction. This feature is known as *state suspension*.

To start getting a better understanding of generators, let's go ahead and see how we can create some.

[286]:
```python
# Generator function for the cube of numbers (power of 3)
def gencubes(n):
    for num in range(n):
        yield num**3
```

[287]:
```python
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.

Let's create another example generator which calculates fibonacci numbers:

[288]:
```python
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
```

[289]:
```python
for num in genfibon(10):
    print(num)
```

```
1
1
2
3
5
8
13
21
34
55
```

What if this was a normal function, what would it look like?

[290]:
```python
def fibon(n):
    a = 1
    b = 1
    output = []

    for i in range(n):
        output.append(a)
        a,b = b,a+b

    return output
```

[291]:
```python
fibon(10)
```

[291]:
```
[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!

8.2.1 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:

[292]:
```
def simple_gen():
    for x in range(3):
        yield x
```

[293]:
```
# Assign simple_gen
g = simple_gen()
```

[294]:
```
print(next(g))
```

```
0
```

[295]:
```
print(next(g))
```

```
1
```

[296]:
```
print(next(g))
```

```
2
```

[297]:
```
print(next(g))
```

```
       ␣
 ↪---------------------------------------------------------------------------

        StopIteration                             Traceback (most recent call␣
 ↪last)

        <ipython-input-297-1dfb29d6357e> 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? A for loop automatically catches this error and stops calling next().

Let's go ahead and check out how to use iter(). You remember that strings are iterables:

[298]:
```
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:

[299]:
```
next(s)
```

```
       ␣
↪---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call␣
↪last)

        <ipython-input-299-61c30b5fe1d5> 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!

[300]:
```
s_iter = iter(s)
```

[301]:
```
next(s_iter)
```

[301]:
```
'h'
```

[302]:
```
next(s_iter)
```

[302]:
```
'e'
```

Here is how a generator function differs from a normal function.

- Generator function contains one or more yield statement.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like **iter**() and **next**() are implemented automatically. So we can iterate through the items using next().
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, StopIteration is raised automatically on further calls.

8.3 Iterators

Python iterator objects are required to support two methods while following the iterator protocol.

`__iter__`

returns the iterator object itself. This is used in for and in statements.

`__next__`

method returns the next value from the iterator. If there is no more items to return then it should raise StopIteration exception.

[303]:
```
class Counter(object):
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        'Returns itself as an iterator object'
        return self

    def __next__(self):
        'Returns the next value till current is lower than high'
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1
```

[304]:
```
c = Counter(5,10)
for i in c: print(i, end=' ')
```

```
5 6 7 8 9 10
```

[305]:
```
c = Counter(5,10)
next(c)
```

[305]:
```
5
```

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

Another StackOverflow Answer