Python_basics

March 11, 2021

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 andm 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]:
```
# Mod
10 % 4
```

[6]:
```
2
```

[7]:
```
# 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]:
```
# 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]:
```
# 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]:
```
# 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

      my_dogs
```

```
[12]: 2
```

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

      my_dogs
```

```
[13]: ['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:

[14]:
```python
# Single word
'hello'
```

[14]:
```
'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.

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

```
[15]: 'Hello World'
```

2.3.3 String Basics

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

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

```
[16]: 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.

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

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

```
[18]: 'H'
```

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

```
[19]: '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:

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

```
[20]: 'Hello World'
```

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

```
[21]: 'HloWrd'
```

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

```
[22]: '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:

```
[23]: # Let's try to change the first letter to 'x'
      try:
          s[0] = 'x'
      except:
          print(' not  supported')
```

```
 not  supported
```

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

Something we *can* do is concatenate strings!

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

```
[24]: '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:

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

[25]: 'HELLO WORLD'

[26]:
```
# Lower case
s.lower()
```

[26]: 'hello world'

[27]:
```
# Split a string by blank space (this is the default)
s.split()
```

[27]: ['Hello', 'World']

[28]:
```
# Split by a specific element (doesn't include the element that was split on)
s.split('W')
```

[28]: ['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:

[29]:
```
'Insert another string with curly brackets: {}'.format('The inserted string')
```

[29]: '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".

[30]:
```
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:

[31]:
```
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.

[32]:
```
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:

[33]:
```
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:

[34]:
```
print('Floating point numbers: %5.2f' %(13.144))
```

```
Floating point numbers: 13.14
```

[35]:
```
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

[36]:
```
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)`.

[37]:
```
name = 'Fred'

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

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

Pass `!r` to get the string representation:

[38]:
```
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!

[39]:
```
# 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:

[40]:
```
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.

[41]:
```
len(my_list)
```

[41]:
```
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:

[42]:
```
my_list = ['one','two','three',4,5]
```

[43]:
```
# Grab element at index 0
my_list[0]
```

[43]:
```
'one'
```

[44]:
```
my_list + ['new item']
```

[44]:
```
['one', 'two', 'three', 4, 5, 'new item']
```

[45]:
```
# Make the list double
my_list * 2
```

[45]:
```
['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:

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

```
[46]: [1, 2, 3]
```

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

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

```
[47]: [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:

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

```
[48]: 1
```

```
[49]: # Show
      list1
```

```
[49]: [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!

```
[50]: # 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]
```

```
[51]: # Show
      matrix
```

```
[51]: [[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!

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

```
[53]: first_col
```

```
[53]: [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!

```
[54]: # 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']
```

```
[54]: '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:

```
[55]: # 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:

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

```
[56]: '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:

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

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

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

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

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

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

```
[60]: 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:

[61]:
```
# Create a tuple
t = (1,2,3)
# Check len just like a list
len(t)
```

[61]:
```
3
```

[62]:
```
# Can also mix object types
t = ('one',2)

# Show
t
```

[62]:
```
('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:

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

```
[63]: 0
```

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

```
[64]: 1
```

2.5.3 Immutability

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

```
[65]: try:
          t[0]= 'change'
      except:
          print('error')
```

```
error
```

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

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

```
[66]: {1}
```

```
[67]: # 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)
```

```
[67]: {1, 2, 3, 4, 5, 6}
```

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).

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

```
[68]: True
```

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

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

```
[69]: False
```

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.

[70]:
```
%%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:

`myfile = open('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**:

[71]:
```
pwd
```

[71]:
```
'/home/sudhir/Downloads/git/data-science-courses/Python_Bootcamp/notebook'
```

[72]:
```
# Open the text.txt we made earlier
my_file = open('test.txt')
```

[73]:
```
# We can now read the file
my_file.read()
```

[73]:
```
'Hello, this is a quick test file.\n'
```

[74]:
```
# But what happens if we try to read it again?
my_file.read()
```

[74]:
```
''
```

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:

[75]:
```
# Readlines returns a list of the lines in the file
my_file.seek(0)
my_file.readlines()
```

[75]:
```
['Hello, this is a quick test file.\n']
```

[76]:
```
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:

[77]:
```
# 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**!

[78]:
```
# Write to the file
my_file.write('This is a new line')
```

[78]:
```
18
```

[79]:
```
# Read the file
my_file.seek(0)
my_file.read()
```

[79]:
```
'This is a new line'
```

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.

[80]:
```
my_file = open('test.txt','a+')
my_file.write('\nThis is text being appended to test.txt')
my_file.write('\nAnd another line here.')
```

[80]:
```
23
```

[81]:
```
my_file.seek(0)
print(my_file.read())
```

```
This is a new line
This is text being appended to test.txt
And another line here.
```

[82]:
```
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:

[83]:
```
%%writefile test.txt
First Line
Second Line
```

```
Overwriting test.txt
```

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

[84]:
```
for line in open('test.txt'):
    print(line)
```

```
First Line

Second Line

```

Don't worry about fully understanding this yet, for loops are coming up soon. But we'll break down what we did above. We said that for every line in this text file, go ahead and print that line. 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()
```

[87]:
```
ls
```

```
file1.py   image/          Python_basics.ipynb  testfile
file2.py   newmyfile.txt   Python_oops.ipynb    test.txt
file3.py   __pycache__/    Python_oops.pdf
```

[88]:
```
%%writefile newmyfile.txt
this is first line
this is the second line
```

```
Overwriting newmyfile.txt
```

[90]:
```
with open('newmyfile.txt',mode = 'a') as f:
    f.write('Four on forth')
```

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

```
[92]: 2 == 2
```

```
[92]: True
```

- Less than or Equal to *

```
[93]: 2 <= 2
```

```
[93]: True
```

```
[94]: 2 <= 4
```

```
[94]: 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:

```
[95]: 1 < 2 < 3
```

```
[95]: 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:

```
[96]: 1<2 and 2<3
```

```
[96]: 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).

[97]:
```
if True:
    print('It was true!')
```

```
It was true!
```

Let's add in some else logic:

[98]:
```python
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.

[99]:
```python
# 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
```

[100]:
```python
tup = (1,2,3,4,5)

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

```
1
2
3
4
5
```

[101]:
```python
list2 = [(2,4),(6,8),(10,12)]
```

[102]:
```python
for tup in list2:
    print(tup)
```

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

[103]:
```python
# Now with unpacking!
for (t1,t2) in list2:
    print(t1)
```

```
2
6
10
```

[104]:
```python
d = {'k1':1,'k2':2,'k3':3}
```

[105]:
```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.

[106]:
```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:

[107]:
```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!

[108]:
```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!m

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:

[109]:
```
range(0,11)
```

[109]:
```
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!

[110]:
```
# Notice how 11 is not included, up to but not including 11, just like slice␣
↪notation!
list(range(0,11))
```

[110]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

[111]:
```
# odd --------- even
list(range(1,10,2)), list(range(0,10,2))
```

[111]:
```
([1, 3, 5, 7, 9], [0, 2, 4, 6, 8])
```

[112]:
```
list(range(10,-10,-1))
```

[112]:
```
[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:

[113]:
```
x = ['a','b','c','d']
a= enumerate(x, start = 10)
list(a)
```

[113]:
```
[(10, 'a'), (11, 'b'), (12, 'c'), (13, 'd')]
```

[114]:
```
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()

[115]:
```
list(enumerate('abcde'))
```

[115]:
```
[(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.

[116]:
```
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)
```

[116]:
```
<zip at 0x7f79dc0cffc0>
```

[117]:
```
# 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

[118]:
```
'x' in ['x','y','z']
```

[118]:
```
True
```

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

```
[119]: False
```

4.7.5 min and max

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

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

```
[120]: 10
```

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

```
[121]: 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.

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

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

```
[123]: [30, 10, 20, 100, 40]
```

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

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

```
Enter Something into this box:  o
```

```
[124]: 'o'
```

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:

[125]:
```
# Grab every letter in string
lst = [x for x in 'word']
lst
```

[125]:
```
['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**

```
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:m

[126]:
```
# 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:

[127]:
```
lst.append(6)
lst
```

[127]:
```
[1, 2, 3, 4, 5, 6]
```

[128]:
```
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:

[129]:
```python
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.

[130]:
```python
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!')
```

[131]:
```
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:

[132]:
```
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))
```

[132]:
```
[1, 4, 9, 16, 25]
```

[133]:
```
def splicer(mystring):
    if len(mystring) % 2 == 0:
        return 'even'
    else:
        return mystring[0]

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

[133]:
```
['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.

[134]:
```
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))
```

[134]:
```
[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.

[135]:
```
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:

[136]:
```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?

[137]:
```python
print(x)
```

```
25
```

[138]:
```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

[139]:
```
# 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)

[140]:
```
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!

[141]:
```
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!)

[142]:
```
len
```

[142]:
```
<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:

[143]:
```
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:

[144]:
```
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.

[145]:
```
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:

[146]:
```
def myfunc(a,b):
    return sum((a,b))*.05

myfunc(40,60)
```

[146]:
```
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.

[147]:
```
def myfunc(a=0,b=0,c=0,d=0,e=0):
    return sum((a,b,c,d,e))*.05

myfunc(40,60,20)
```

[147]:
```
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:

[148]:
```
def myfunc(*args):
    return sum(args)*.05

myfunc(40,60,20)
```

[148]:
```
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:

[149]:
```
def myfunc(*spam):
    return sum(spam)*.05
```

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

[149]:
```
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:

[150]:
```
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
```

[151]:
```
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`

[152]:
```
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:

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

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".

[153]:
```
# 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.

[154]:
```
def outer_func():
    message = 'Hi'

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

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

```
Hi
Hi
Hi
```

[155]:
```
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
```

[156]:
```
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:

[157]:
```
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")
```

[158]:
```
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
```

[159]:
```
try:
    welceme()
except:
    print('Hi error')
```

```
Hi error
```

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:

[160]:
```
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.

[161]:
```
x = hello()

x
```

[161]:
```
<function __main__.hello.<locals>.greet()>
```

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

[162]:
```
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:

[163]:
```python
def hello():
    return 'Hi Jose!'

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

[164]:
```python
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: