



Object Oriented Python

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

DOWNLOAD PDF CODING BUGS NOTES GALLERY



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

About the Tutorial

Python has been an object-oriented language since it existed. In this tutorial we will try to get in-depth features of OOPS in Python programming.

Audience

This tutorial has been prepared for the beginners and intermediate to help them understand the Python OOPS features and concepts through programming.

Prerequisites

Understanding on basic of Python programming language will help to understand and learn quickly. If you are new to programming, it is recommended to first go through "Python for beginners" tutorials.

Table of Contents

About the Tutorial	ii
Audience.....	ii
Prerequisites.....	ii
OOP IN PYTHON – INTRODUCTION	1
Language Programming Classification Scheme	1
What is Object Oriented Programming?	2
Why to Choose Object-oriented programming?.....	2
Procedural vs. Object Oriented Programming.....	2
Principles of Object Oriented Programming.....	3
Object-Oriented Python.....	5
Modules vs. Classes and Objects.....	5
OOP IN PYTHON – ENVIRONMENT SETUP.....	8
Prerequisites and Toolkits	8
Installing Python.....	8
Choosing an IDE	10
Pycharm.....	10
Komodo IDE	11
Eric Python IDE	12
Choosing a Text Editor	13
Atom Text Editor	13
Screenshot of Atom text	14
Sublime Text Editor	14
Notepad ++.....	15
OOP IN PYTHON – DATA STRUCTURES	17
Lists	17

Accessing Items in Python List	18
Empty Objects	18
Tuples	19
Dictionary	21
Sets.....	24
OOP IN PYTHON – BUILDING BLOCKS.....	28
Class Bundles : Behavior and State	28
Creation and Instantiation	29
Instance Methods.....	30
Encapsulation	31
Init Constructor.....	33
Class Attributes.....	34
Working with Class and Instance Data	35
OOP IN PYTHON – OBJECT ORIENTED SHORTCUT	37
Python Built-in Functions.....	37
Default Arguments	42
OOP IN PYTHON – INHERITANCE AND POLYMORPHISM	44
Inheritance	44
Inheriting Attributes	44
Inheritance Examples.....	45
Polymorphism (“MANY SHAPES”).....	47
Overriding.....	48
Inheriting the Constructor	49
Multiple Inheritance and the Lookup Tree	50
Decorators, Static and Class Methods.....	54
OOP IN PYTHON –PYTHON DESIGN PATTERN.....	57

Overview	57
Why is Design Pattern Important?	57
Classification of Design Patterns.....	57
Commonly used Design Patterns	58
 OOP IN PYTHON – ADVANCED FEATURES	60
Core Syntax in our Class design.....	60
Inheriting From built-in types	61
Naming Conventions.....	63
 OOP IN PYTHON – FILES AND STRINGS.....	65
Strings.....	66
File I/O.....	71
 OOP IN PYTHON – EXCEPTION AND EXCEPTION CLASSES.....	72
Identifying Exception (Errors)	72
Catching/Trapping Exception	73
Raising Exceptions	75
Creating Custom exception class.....	76
 OOP IN PYTHON – OBJECT SERIALIZATION	80
Pickle	80
Methods	81
Unpickling.....	82
JSON	82
YAML	85
Installing YAML.....	85
PDB – The Python Debugger	89
Logging	91
Benchmarking.....	93

12. OOP IN PYTHON – PYTHON LIBRARIES	96
Requests: Python Requests Module	96
Making a GET Request	96
Making POST Requests	97
Pandas: Python Library Pandas.....	97
Indexing DataFrames	98
Pygame	99
Beautiful Soup: Web Scraping with Beautiful Soup	102

1. OOP in Python – Introduction

Programming languages are emerging constantly, and so are different methodologies. Object-oriented programming is one such methodology that has become quite popular over past few years.

This chapter talks about the features of Python programming language that makes it an object-oriented programming language.

Language Programming Classification Scheme

Python can be characterized under object-oriented programming methodologies. The following image shows the characteristics of various programming languages. Observe the features of Python that makes it object-oriented.

Language Classes	Categories	Languages
Programming Paradigm	Procedural	C, C++, C#, Objective-C, Java, Go
	Scripting	CoffeeScript, JavaScript, Python, Perl, Php, Ruby
	Functional	Closure, Erlang, Haskell, Scala
Compilation Class	Static	C, C++, C#, Objective-C, Java, Go, Haskell, Scala
	Dynamic	CoffeeScript, JavaScript, Python, Perl, Php, Ruby, Closure, Erlang
Type Class	Strong	C#, Java, Go, Python, Ruby, Closure, Erlang, Haskell, Scala
	Weak	C, C++, Objective-C, CoffeeScript, JavaScript, Perl, Php
Memory Class	Managed	Others
	Unmanaged	C, C++, Objective-C

What is Object Oriented Programming?

Object Oriented means directed towards objects. In other words, it means functionally directed towards modelling objects. This is one of the many techniques used for modelling complex systems by describing a collection of interacting objects via their data and behavior.

Python, an Object Oriented programming (OOP), is a way of programming that focuses on using objects and classes to design and build applications.. Major pillars of Object Oriented Programming (OOP) are **Inheritance**, **Polymorphism**, **Abstraction**, ad **Encapsulation**.

Object Oriented Analysis(OOA) is the process of examining a problem, system or task and identifying the objects and interactions between them.

Why to Choose Object Oriented Programming?

Python was designed with an object-oriented approach. OOP offers the following advantages:

- Provides a clear program structure, which makes it easy to map real world problems and their solutions.
- Facilitates easy maintenance and modification of existing code.
- Enhances program modularity because each object exists independently and new features can be added easily without disturbing the existing ones.
- Presents a good framework for code libraries where supplied components can be easily adapted and modified by the programmer.
- Imparts code reusability

Procedural vs. Object Oriented Programming

Procedural based programming is derived from structural programming based on the concepts of **functions/procedure/routines**. It is easy to access and change the data in procedural oriented programming. On the other hand, Object Oriented Programming (OOP) allows decomposition of a problem into a number of units called **objects** and then build the data and functions around these objects. It emphasis more on the data than procedure or functions. Also in OOP, data is hidden and cannot be accessed by external procedure.

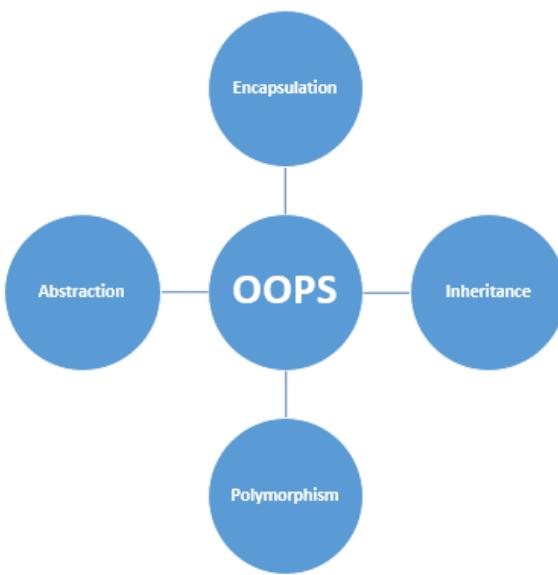
The table in the following image shows the major differences between POP and OOP approach.

Difference between Procedural Oriented Programming (POP) vs. Object oriented programming (OOP).

	Procedural Oriented Programming	Object Oriented Programming
Based On	In Pop, entire focus is on data and functions	Oops is based on a real world scenarios. Whole program is divided into small parts called object
Reusability	Limited Code reuse	Code reuse
Approach	Top down approach	Object focused Design
Access specifiers	Not any	Public, Private and Protected
Data movement	Data can move freely from functions to function in the system	In oops, objects can move and communicate with each other through member functions
Data Access	In pop, most function uses global data for sharing that can be accessed freely from function to function in the system	In oops, data cannot move freely from method to method, it can be kept in public or private so we can control the access of data.
Data Hiding	In pop, so specific way to hide data, so little bit less secure	It provides data hiding, so much more secure.
Overloading	Not possible	Function and Operator Overloading
Example-Languages	C, VB, Fortran, Pascal	C++, Python, Java, C#
Abstraction	Uses abstraction at procedure level	Uses abstraction at class and object level

Principles of Object Oriented Programming

Object Oriented Programming (OOP) is based on the concept of **objects** rather than actions, and **data** rather than logic. In order for a programming language to be object-oriented, it should have a mechanism to enable working with classes and objects as well as the implementation and usage of the fundamental object-oriented principles and concepts namely inheritance, abstraction, encapsulation and polymorphism.



Four Pillars of Object-Oriented Programming

Let us understand each of the pillars of object-oriented programming in brief:

Encapsulation

This property hides unnecessary details and makes it easier to manage the program structure. Each object's implementation and state are hidden behind well-defined boundaries and that provides a clean and simple interface for working with them. One way to accomplish this is by making the data private.

Inheritance

Inheritance, also called generalization, allows us to capture a hierachal relationship between classes and objects. For instance, a 'fruit' is a generalization of 'orange'. Inheritance is very useful from a code reuse perspective.

Abstraction

This property allows us to hide the details and expose only the essential features of a concept or object. For example, a person driving a scooter knows that on pressing a horn, sound is emitted, but he has no idea about how the sound is actually generated on pressing the horn.

Polymorphism

Poly-morphism means many forms. That is, a thing or action is present in different forms or ways. One good example of polymorphism is constructor overloading in classes.

Object-Oriented Python

The heart of Python programming is **object** and **OOP**, however you need not restrict yourself to use the OOP by organizing your code into classes. OOP adds to the whole design philosophy of Python and encourages a clean and pragmatic way to programming. OOP also enables in writing bigger and complex programs.

Modules vs. Classes and Objects

Modules are like “Dictionaries”

When working on Modules, note the following points:

- A Python module is a package to encapsulate reusable code.
- Modules reside in a folder with a **`__init__.py`** file on it.
- Modules contain functions and classes.
- Modules are imported using the **`import`** keyword.

Recall that a dictionary is a **key-value** pair. That means if you have a dictionary with a key **EmployeeID** and you want to retrieve it, then you will have to use the following lines of code:

```
employee = {"EmployeeID": "Employee Unique Identity!"}
print (employee ['EmployeeID'])
```

You will have to work on modules with the following process:

- A module is a Python file with some functions or variables in it.
- Import the file you need.
- Now, you can access the functions or variables in that module with the **'.'** (**dot**) Operator.

Consider a module named **employee.py** with a function in it called **employee**. The code of the function is given below:

```
# this goes in employee.py
def EmployeeID():
    print ("Employee Unique Identity!")
```

Now import the module and then access the function **EmployeeID**:

```
import employee
employee.EmployeeID()
```

You can insert a variable in it named **Age**, as shown:

```
def EmployeID():
    print ("Employee Unique Identity!")
# just a variable
Age = "Employee age is **"
```

Now, access that variable in the following way:

```
import employee
employee.EmployeID()
print(employee.Age)
```

Now, let's compare this to dictionary:

```
Employee[ 'EmployeID' ]      # get EmployeID from employee
Employee.employeID()         # get employeID from the module
Employee.Age                 # get access to variable
```

Notice that there is common pattern in Python:

- Take a **key = value** style container
- Get something out of it by the key's name

When comparing module with a dictionary, both are similar, except with the following:

- In the case of the **dictionary**, the key is a string and the syntax is [key].
- In the case of the **module**, the key is an identifier, and the syntax is .key.

Classes are like Modules

Module is a specialized dictionary that can store Python code so you can get to it with the '.' Operator. A class is a way to take a grouping of functions and data and place them inside a container so you can access them with the '.'operator.

If you have to create a class similar to the employee module, you can do it using the following code:

```
class employee(object):
    def __init__(self):
        self. Age = "Employee Age is ##"
    def EmployeID(self):
        print ("This is just employee unique identity")
```

Note: Classes are preferred over modules because you can reuse them as they are and without much interference. While with modules, you have only one with the entire program.

Objects are like Mini-imports

A class is like a **mini-module** and you can import in a similar way as you do for classes, using the concept called **instantiate**. Note that when you instantiate a class, you get an **object**.

You can instantiate an object, similar to calling a class like a function, as shown:

```
this_obj = employee()           # Instantiatethis_obj.EmployeID()      #
get EmployeId from the class

print(this_obj.Age)             # get variable Age
```

You can do this in any of the following three ways:

```
# dictionary style
Employee['EmployeID']

# module style
Employee.EmployeID()
Print(employee.Age)

# Class style
this_obj = employee()
this_obj.employeID()
Print(this_obj.Age)
```

2. OOP in Python – Environment Setup

This chapter will explain in detail about setting up the Python environment on your local computer.

Prerequisites and Toolkits

Before you proceed with learning further on Python, we suggest you to check whether the following prerequisites are met:

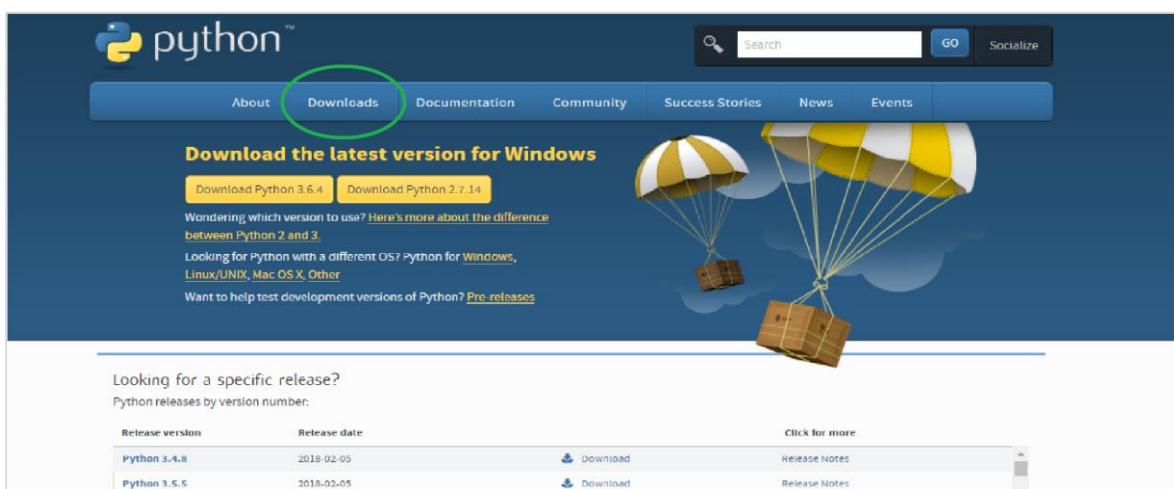
- Latest version of Python is installed on your computer
- An IDE or text editor is installed
- You have basic familiarity to write and debug in Python, that is you can do the following in Python:
 - Able to write and run Python programs.
 - Debug programs and diagnose errors.
 - Work with basic data types.
 - Write **for** loops, **while** loops, and **if** statements
 - Code **functions**

If you don't have any programming language experience, you can find lots of beginner tutorials in Python on [TutorialsPoint](#).

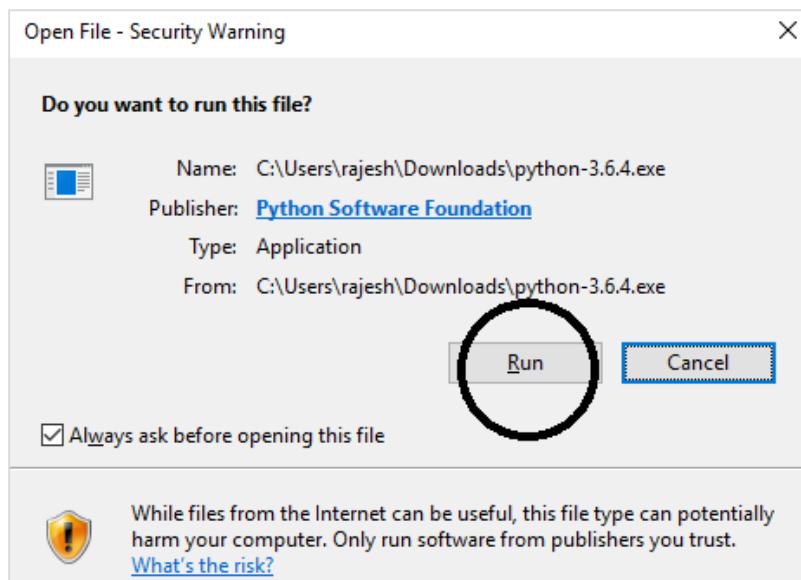
Installing Python

The following steps show you in detail how to install Python on your local computer:

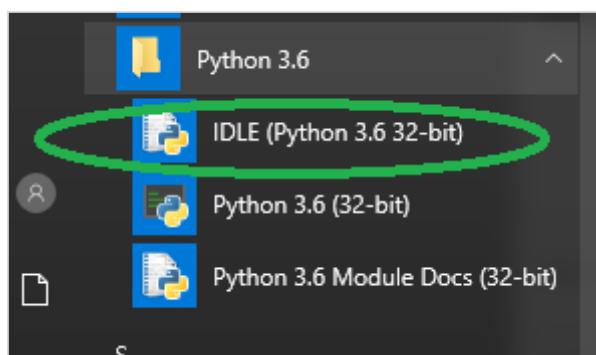
Step 1: Go to the official Python website <https://www.Python.org/>, click on the **Downloads** menu and choose the latest or any stable version of your choice.



Step 2: Save the Python installer exe file that you're downloading and once you have downloaded it, open it. Click on **Run** and choose **Next** option by default and finish the installation.



Step 3: After you have installed, you should now see the Python menu as shown in the image below. Start the program by choosing IDLE (Python GUI).



This will start the Python shell. Type in simple commands to check the installation.

```

Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)]
Type "copyright", "credits" or "license()" for more information.
>>> 2+2
4
>>> print("Hello World!")
Hello World!
>>> |

```

Choosing an IDE

An Integrated Development Environment is a text editor geared towards software development. You will have to install an IDE to control the flow of your programming and to group projects together when working on Python. Here are some of IDEs available online. You can choose one at your convenience.

- Pycharm IDE
- Komodo IDE
- Eric Python IDE

Note: Eclipse IDE is mostly used in Java, however it has a Python plugin.

Pycharm

Pycharm, the cross-platform IDE is one of the most popular IDE currently available. It provides coding assistance and analysis with code completion, project and code navigation, integrated unit testing, version control integration, debugging and much more.

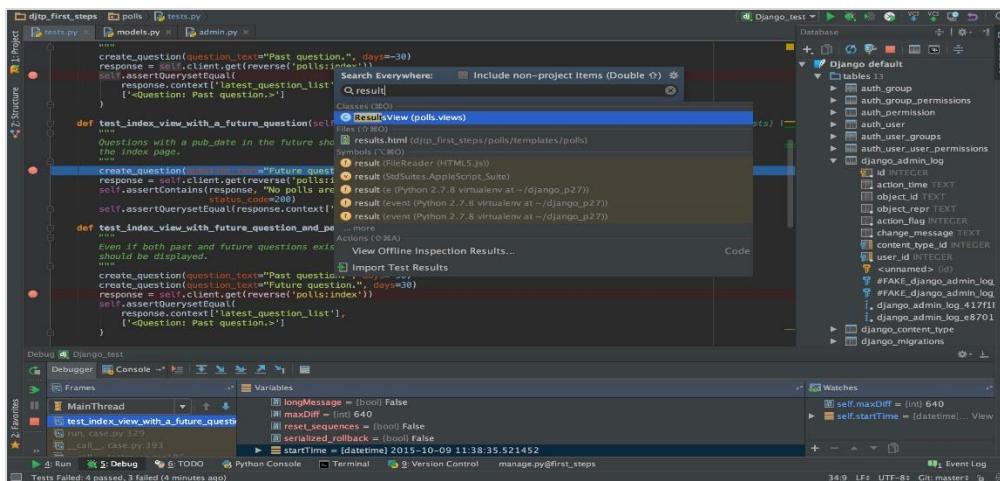


Download link

<https://www.jetbrains.com/pycharm/download/>

Languages Supported: Python, HTML, CSS, JavaScript, Coffee Script, TypeScript, Cython, AngularJS, Node.js, template languages.

Screenshot



Why to Choose?

PyCharm offers the following features and benefits for its users:

- Cross platform IDE compatible with Windows, Linux, and Mac OS
- Includes Django IDE, plus CSS and JavaScript support
- Includes thousands of plugins, integrated terminal and version control
- Integrates with Git, SVN and Mercurial

- Offers intelligent editing tools for Python
- Easy integration with Virtualenv, Docker and Vagrant
- Simple navigation and search features
- Code analysis and refactoring
- Configurable injections
- Supports tons of Python libraries
- Contains Templates and JavaScript debuggers
- Includes Python/Django debuggers
- Works with Google App Engine, additional frameworks and libraries.
- Has customizable UI, VIM emulation available

Komodo IDE

It is a polyglot IDE which supports 100+ languages and basically for dynamic languages such as Python, PHP and Ruby. It is a commercial IDE available for 21 days free trial with full functionality. ActiveState is the software company managing the development of the Komodo IDE. It also offers a trimmed version of Komodo known as Komodo Edit for simple programming tasks.



This IDE contains all kinds of features from most basic to advanced level. If you are a student or a freelancer, then you can buy it almost half of the actual price. However, it's completely free for teachers and professors from recognized institutions and universities.

It got all the features you need for web and mobile development, including support for all your languages and frameworks.

Download link

The download links for Komodo Edit(free version) and Komodo IDE(paid version) are as given here:

Komodo Edit (free)

<https://www.activestate.com/komodo-edit>

Komodo IDE (paid)

<https://www.activestate.com/komodo-ide/downloads/ide>

Screenshot

The screenshot shows the ActiveState Komodo IDE 5.0 interface. The main window has a title bar "context-menu.js (/home/troyt/.komodoide/5.0/samples/js_tutorials) - ActiveState Komodo IDE 5.0". The menu bar includes File, Edit, Code, View, Debug, Project, Toolbox, Tools, Window, and Help. The toolbar contains various icons for file operations like Open, Save, Print, and Find. The left sidebar shows a file tree with files like "context-menu.js", "demo", "onCitySelect", and "myTarget". The central code editor pane displays a JavaScript function "onCitySelect". Below the code editor are tabs for Breakpoints, Command Output, Find Results 1, Test Results, and SCC Output. The "Find Results 1" tab is active, showing 11 occurrences of "onCitySelect" across five files. The bottom status bar shows "Visual Ready" and "JavaScript".

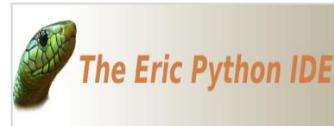
Why to Choose?

- Powerful IDE with support for Perl, PHP, Python, Ruby and many more.
- Cross-Platform IDE.

It includes basic features like integrated debugger support, auto complete, Document Object Model(DOM) viewer, code browser, interactive shells, breakpoint configuration, code profiling, integrated unit testing. In short, it is a professional IDE with a host of productivity-boosting features.

Eric Python IDE

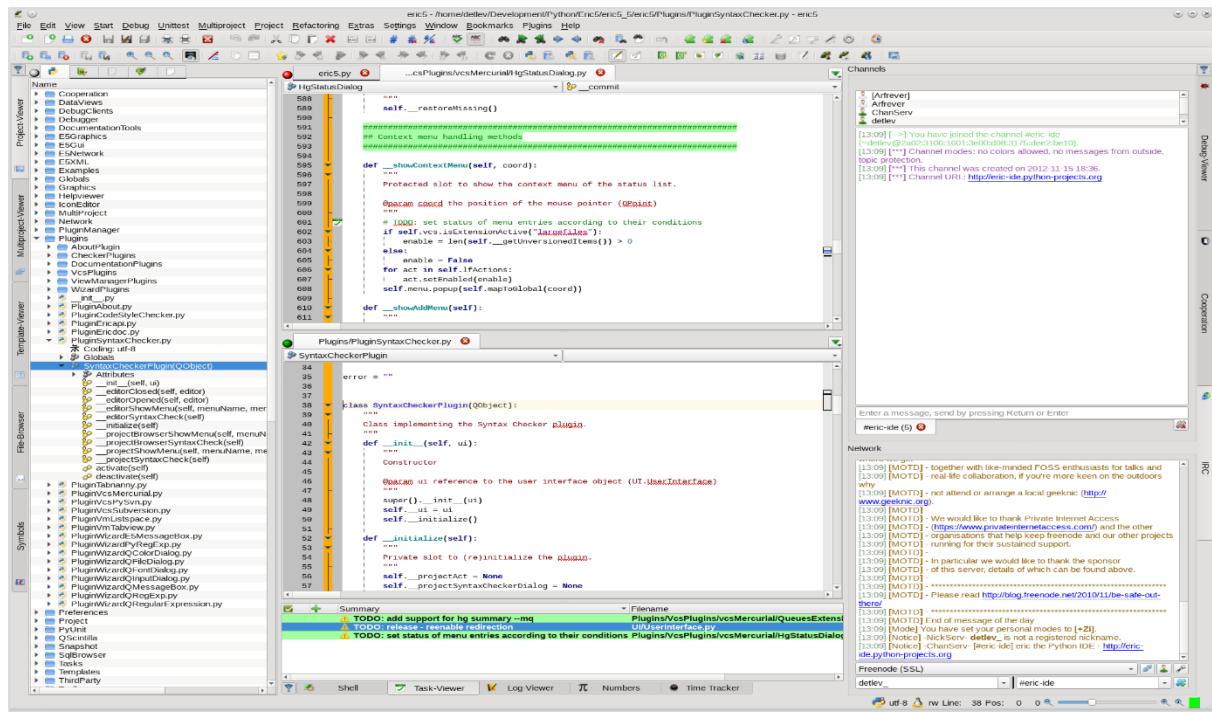
It is an open-source IDE for Python and Ruby. Eric is a full featured editor and IDE, written in Python. It is based on the cross platform Qt GUI toolkit, integrating the highly flexible Scintilla editor control. The IDE is very much configurable and one can choose what to use and what not. You can download Eric IDE from below link:
<https://eric-ide.Python-projects.org/eric-download.html>



Why to Choose

- Great indentation, error highlighting.
- Code assistance
- Code completion
- Code cleanup with PyLint
- Quick search
- Integrated Python debugger.

Screenshot



Choosing a Text Editor

You may not always need an IDE. For tasks such as learning to code with Python or Arduino, or when working on a quick script in shell script to help you automate some tasks a simple and light weight code-centric text editor will do. Also many text editors offer features such as syntax highlighting and in-program script execution, similar to IDEs. Some of the text editors are given here:

- Atom
- Sublime Text
- Notepad++

Atom Text Editor

Atom is a hackable text editor built by the team of GitHub. It is a free and open source text and code editor which means that all the code is available for you to read, modify for your own use and even contribute improvements. It is a cross-platform text editor compatible for macOS, Linux, and Microsoft Windows with support for plug-ins written in Node.js and embedded Git Control.



Download link

<https://atom.io/>

Screenshot

The screenshot shows a terminal window with a file browser on the left and a code editor on the right.

File Browser (Left):

- File: README.md, scipion, config, instal, pyworkflow, scripts, software, ignore, SConstruct
- scripts: __init__.py, change_rpath.py, clean, find_deps.py, fix_links.py, monitor.py, plotter.py, run_apache.py, run_tests.py, split_stacks.py, sync_data.py

Code Editor (Right):

```
177 add('--check-all', action='store_true',  
178 help='See if there is any remote dataset not in sync with locals.')  
179 add('-v', '--verbose', action='store_true', help='Print more details.')  
180  
181 return parser  
182  
183 def listDatasets(url):  
184     """ Print a list of local and remote datasets """  
185  
186     tdir = os.environ['SCIPION_TESTS']  
187     print("Local datasets in %s" % yellow(tdir))  
188     for folder in sorted(os.listdir(tdir)):  
189         if os.path.isdir(os.path.join(tdir, folder)):  
190             if exists(join(tdir, folder, 'MANIFEST')):  
191                 print(" * %s" % folder)  
192             else:  
193                 print(" * %s (not in dataset format)" % folder)  
194  
195     try:  
196         print("\nRemote datasets in %s" % yellow(url))  
197         for line in sorted(urllib.urlopen("%s/manifest" % url).readlines()):  
198             print(" * %s" % line.rstrip("\r\n"))  
199     except Exception as e:  
200         print("Error reading %s (%s)" % (url, e))  
201  
202  
203 def check(dataset, url, verbose=False, updateMANIFEST=False):  
204     """ See if our local copy of dataset is the same as the remote one.  
205     Return True if it is (if all the checksums are equal). False if not.  
206     """  
207  
208     def vlog(txt): sys.stdout.write(txt) if verbose else None # verbose log  
209  
210     vlog("Checking dataset %s ... %s" % dataset)  
211  
212     if updateMANIFEST:  
213         createMANIFEST(join(os.environ['SCIPION_TESTS'], dataset))  
214     else:  
215         vlog("(not updating local MANIFEST) ")  
216  
217     try:  
218         md5sRemote = dict(x.split() for x in  
219                         urllib.urlopen("%s/%s/manifest" % (url, dataset)))  
220  
221         md5sLocal = dict(x.split() for x in  
222                         open("%s/manifest" % dataset))  
223  
224         if md5sLocal == md5sRemote:  
225             vlog("Dataset %s is up-to-date" % dataset)  
226         else:  
227             vlog("Dataset %s is NOT up-to-date" % dataset)  
228  
229  
230 if __name__ == "__main__":  
231     main()  
232  
233 if __name__ == "scipion":  
234     get_parser()  
235  
236 if __name__ == "manager":  
237     manager()  
238  
239 if __name__ == "sync_data":  
240     sync_data()
```

Languages Supported

C/C++, C#, CSS, CoffeeScript, HTML, JavaScript, Java, JSON, Julia, Objective-C, PHP, Perl, Python, Ruby on Rails, Ruby, Shell script, Scala, SQL, XML, YAML and many more.

Sublime Text Editor

Sublime text is a proprietary software and it offers you a free trial version to test it before you purchase it. According to [Stackoverflow's 2018 developer survey](#), it's the fourth most popular Development Environment.



Some of the advantages it provides is its incredible speed, ease of use and community support. It also supports many programming languages and mark-up languages, and functions can be added by users with plugins, typically community-built and maintained under free-software licenses.

Screenshot

```

1 <?php
2 class Email
3 {
4     /**
5      * @static send a email
6      * @param array $data
7      *      array('subject'=>?, 'params'=>array(), 'view'=>?, 'to'=>?, 'from'=>?)
8      * @param bool $requireView if true, it will only send email if view is existed
9     */
10    public static function mail($data, $requireView = false)
11    {
12        try {
13            $message = new YiiMailMessage($data['subject']);
14            if (!isset($data['view'])) {
15                if ($requireView) {
16                    $path = YiiBase::getPathOfAlias(Yii::app() ->mail ->viewPath) . '/' . $data['view'] . '.php';
17                    if (!file_exists($path)) {
18                        Yii::log($data['view'] . ' do not exists', 'error', 'Email:mail');
19                        return;
20                    }
21                }
22                $message->view = $data['view'];
23            }
24            elseif ($requireView)
25                return;
26            $message->setBody($data['params'], 'text/html');
27
28            if (is_array($data['to'])) {
29                foreach ($data['to'] as $t)
30                {
31                    $message->addTo($t);
32                }
33            }
34            else
35                $message->addTo($data['to']);
36            $message->from = $data['from'];
37            if (YII_DEBUG)
38                Yii::app() ->mail ->send($message);
39            else
40                @Yii::app() ->mail ->send($message);
41            catch (Exception $e)
42            {
43                Yii::log($e->getMessage(), 'error', 'Email:mail');
44            }
45        }
46    }
47 }

```

Line 1, Column 1 Spaces: 4 PHP

Language supported

- Python, Ruby, JavaScript etc.

Why to Choose?

- Customize key bindings, menus, snippets, macros, completions and more.
- Auto completion feature
- Quickly Insert Text & code with sublime text snippets using snippets, field markers and place holders
- Opens Quickly
- Cross Platform support for Mac, Linux and Windows.
- Jump the cursor to where you want to go
- Select Multiple Lines, Words and Columns

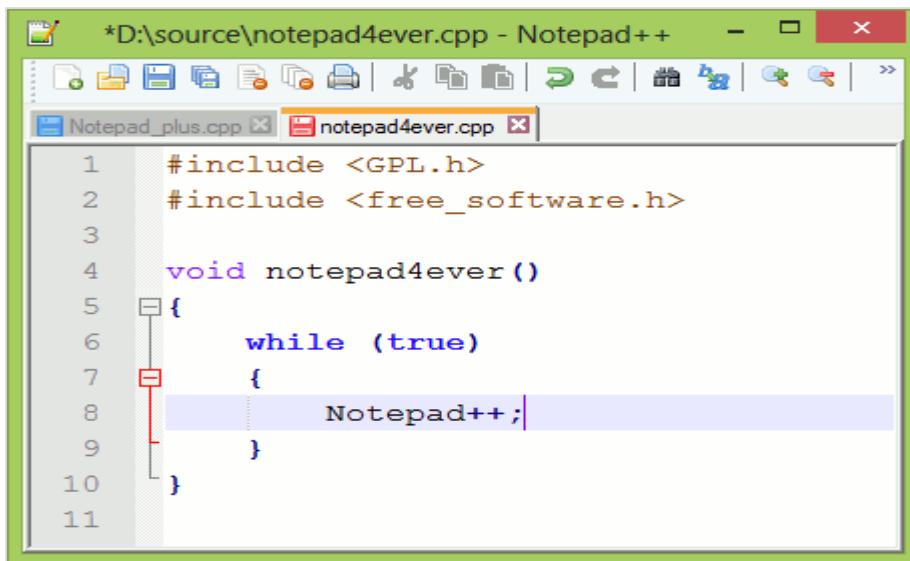
Notepad++

It's a free source code editor and Notepad replacement that supports several languages from Assembly to XML and including Python. Running in the MS windows environment, its use is governed by GPL license.

In addition to syntax highlighting, Notepad++ has some features that are particularly useful to coders.



Screenshot



The screenshot shows the Notepad++ interface with two tabs open: "Notepad_plus.cpp" and "notepad4ever.cpp". The "notepad4ever.cpp" tab is active, displaying the following C++ code:

```
1 #include <GPL.h>
2 #include <free_software.h>
3
4 void notepad4ever()
5 {
6     while (true)
7     {
8         Notepad++;
9     }
10 }
```

The code uses color-coded syntax highlighting and syntax folding. The "Notepad++" line at the end of the loop body is highlighted in blue and has a red square selection box to its left, indicating it is selected or being edited.

Key Features

- Syntax highlighting and syntax folding
- PCRE (Perl Compatible Regular Expression) Search/Replace.
- Entirely customizable GUI
- Auto completion
- Tabbed editing
- Multi-View
- Multi-Language environment
- Launchable with different arguments.

Language Supported

- Almost every language (60+ languages) like Python, C, C++, C#, Java etc.

3. OOP in Python – Data Structures

Python data structures are very intuitive from a syntax point of view and they offer a large choice of operations. You need to choose Python data structure depending on what the data involves, if it needs to be modified, or if it is a fixed data and what access type is required, such as at the beginning/end/random etc.

Lists

A List represents the most versatile type of data structure in Python. A list is a container which holds comma-separated values (items or elements) between square brackets. Lists are helpful when we want to work with multiple related values. As lists keep data together, we can perform the same methods and operations on multiple values at once. Lists indices start from zero and unlike strings, lists are mutable.

Data Structure - List

```
>>>
>>> # Any Empty List
>>> empty_list = []
>>>
>>> # A list of String
>>> str_list = ['Life', 'Is', 'Beautiful']
>>> # A list of Integers
>>> int_list = [1, 4, 5, 9, 18]
>>>
>>> #Mixed items list
>>> mixed_list = ['This', 9, 'is', 18, 45.9, 'a', 54, 'mixed', 99, 'list']
>>> # To print the list
>>>
>>> print(empty_list)
[]
>>> print(str_list)
['Life', 'Is', 'Beautiful']
>>> print(type(str_list))
<class 'list'>
>>> print(int_list)
[1, 4, 5, 9, 18]
```

```
>>> print(mixed_list)
['This', 9, 'is', 18, 45.9, 'a', 54, 'mixed', 99, 'list']
```

Accessing Items in Python List

Each item of a list is assigned a number – that is the index or position of that number. Indexing always start from zero, the second index is one and so forth. To access items in a list, we can use these index numbers within a square bracket. Observe the following code for example:

```
>>> mixed_list = ['This', 9, 'is', 18, 45.9, 'a', 54, 'mixed', 99, 'list']
>>>
>>> # To access the First Item of the list
>>> mixed_list[0]
'This'
>>> # To access the 4th item
>>> mixed_list[3]
18
>>> # To access the last item of the list
>>> mixed_list[-1]
'list'
```

Empty Objects

Empty Objects are the simplest and most basic Python built-in types. We have used them multiple times without noticing and have extended it to every class we have created. The main purpose to write an empty class is to block something for time being and later extend and add a behavior to it.

To add a behavior to a class means to replace a data structure with an object and change all references to it. So it is important to check the data, whether it is an object in disguise, before you create anything. Observe the following code for better understanding:

```
>>> #Empty objects
>>>
>>> obj = object()
>>> obj.x = 9
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    obj.x = 9
AttributeError: 'object' object has no attribute 'x'
```

So from above, we can see it's not possible to set any attributes on an object that was instantiated directly. When Python allows an object to have arbitrary attributes, it takes a certain amount of system memory to keep track of what attributes each object has, for storing both the attribute name and its value. Even if no attributes are stored, a certain amount of memory is allocated for potential new attributes.

So Python disables arbitrary properties on object and several other built-ins, by default.

```
>>> # Empty Objects
>>>
>>> class EmpObject:
    pass
>>> obj = EmpObject()
>>> obj.x = 'Hello, World!'
>>> obj.x
'Hello, World!'
```

Hence, if we want to group properties together, we could store them in an empty object as shown in the code above. However, this method is not always suggested. Remember that classes and objects should only be used when you want to specify both data and behaviors.

Tuples

Tuples are similar to lists and can store elements. However, they are immutable, so we cannot add, remove or replace objects. The primary benefit tuple provides because of its immutability is that we can use them as keys in dictionaries, or in other locations where an object requires a hash value.

Tuples are used to store data, and not behavior. In case you require behavior to manipulate a tuple, you need to pass the tuple into a function(or method on another object) that performs the action.

As tuple can act as a dictionary key, the stored values are different from each other. We can create a tuple by separating the values with a comma. Tuples are wrapped in parentheses but not mandatory. The following code shows two identical assignments .

```
>>> stock1 = 'MSFT', 95.00, 97.45, 92.45
>>> stock2 = ('MSFT', 95.00, 97.45, 92.45)
>>> type(stock1)
<class 'tuple'>
>>> type(stock2)
<class 'tuple'>
>>> stock1 == stock2
True
>>>
```

Defining a Tuple

Tuples are very similar to list except that the whole set of elements are enclosed in parentheses instead of square brackets.

Just like when you slice a list, you get a new list and when you slice a tuple, you get a new tuple.

```
>>> tupl = ('Tuple', 'is', 'an', 'IMMUTABLE', 'list')
>>> tupl
('Tuple', 'is', 'an', 'IMMUTABLE', 'list')
>>> tupl[0]
'Tuple'
>>> tupl[-1]
'list'
>>> tupl[1:3]
('is', 'an')
```

Python Tuple Methods

The following code shows the methods in Python tuples:

```
>>> tupl
('Tuple', 'is', 'an', 'IMMUTABLE', 'list')
>>> tupl.append('new')
Traceback (most recent call last):
  File "<pyshell#148>", line 1, in <module>
    tupl.append('new')
AttributeError: 'tuple' object has no attribute 'append'
>>> tupl.remove('is')
Traceback (most recent call last):
  File "<pyshell#149>", line 1, in <module>
    tupl.remove('is')
AttributeError: 'tuple' object has no attribute 'remove'
>>> tupl.index('list')
4
>>> tupl.index('new')
Traceback (most recent call last):
  File "<pyshell#151>", line 1, in <module>
    tupl.index('new')
```

```

ValueError: tuple.index(x): x not in tuple
>>> "is" in tupl
True
>>> tupl.count('is')
1

```

From the code shown above, we can understand that tuples are immutable and hence:

- You **cannot** add elements to a tuple.
- You **cannot** append or extend a method.
- You **cannot** remove elements from a tuple.
- Tuples have **no** remove or pop method.
- Count and index are the methods available in a tuple.

Dictionary

Dictionary is one of the Python's built-in data types and it defines one-to-one relationships between keys and values.

Defining Dictionaries

Observe the following code to understand about defining a dictionary:

```

>>> # empty dictionary
>>> my_dict = {}
>>>
>>> # dictionary with integer keys
>>> my_dict = { 1:'msft', 2: 'IT'}
>>>
>>> # dictionary with mixed keys
>>> my_dict = {'name': 'Aarav', 1: [ 2, 4, 10]}
>>>
>>> # using built-in function dict()
>>> my_dict = dict({1:'msft', 2:'IT'})
>>>
>>> # From sequence having each item as a pair
>>> my_dict = dict([(1,'msft'), (2,'IT')])
>>>
>>> # Accessing elements of a dictionary
>>> my_dict[1]

```

```
'msft'
>>> my_dict[2]
'IT'
>>> my_dict['IT']
Traceback (most recent call last):
  File "<pyshell#177>", line 1, in <module>
    my_dict['IT']
KeyError: 'IT'

>>>
```

From the above code we can observe that:

- First we create a dictionary with two elements and assign it to the variable **my_dict**. Each element is a key-value pair, and the whole set of elements is enclosed in curly braces.
- The number **1** is the key and **msft** is its value. Similarly, **2** is the key and **IT** is its value.
- You can get values by key, but not vice-versa. Thus when we try **my_dict['IT']** , it raises an exception, because **IT** is not a key.

Modifying Dictionaries

Observe the following code to understand about modifying a dictionary:

```
>>> # Modifying a Dictionary
>>>
>>> my_dict
{1: 'msft', 2: 'IT'}
>>> my_dict[2] = 'Software'
>>> my_dict
{1: 'msft', 2: 'Software'}
>>>
>>> my_dict[3] = 'Microsoft Technologies'
>>> my_dict
{1: 'msft', 2: 'Software', 3: 'Microsoft Technologies'}
```

From the above code we can observe that:

- You cannot have duplicate keys in a dictionary. Altering the value of an existing key will delete the old value.
- You can add new key-value pairs at any time.

- Dictionaries have no concept of order among elements. They are simple unordered collections.

Mixing Data types in a Dictionary

Observe the following code to understand about mixing data types in a dictionary:

```
>>> # Mixing Data Types in a Dictionary
>>>
>>> my_dict
{1: 'msft', 2: 'Software', 3: 'Microsoft Technologies'}
>>> my_dict[4] = 'Operating System'
>>> my_dict
{1: 'msft', 2: 'Software', 3: 'Microsoft Technologies', 4: 'Operating System'}
>>> my_dict['Bill Gates'] = 'Owner'
>>> my_dict
{1: 'msft', 2: 'Software', 3: 'Microsoft Technologies', 4: 'Operating System',
'Bill Gates': 'Owner'}
```

From the above code we can observe that:

- Not just strings but dictionary value can be of any data type including strings, integers, including the dictionary itself.
- Unlike dictionary values, dictionary keys are more restricted, but can be of any type like strings, integers or any other.

Deleting Items from Dictionaries

Observe the following code to understand about deleting items from a dictionary:

```
>>> # Deleting Items from a Dictionary
>>>
>>> my_dict
{1: 'msft', 2: 'Software', 3: 'Microsoft Technologies', 4: 'Operating System',
'Bill Gates': 'Owner'}
>>>
>>> del my_dict['Bill Gates']
>>> my_dict
{1: 'msft', 2: 'Software', 3: 'Microsoft Technologies', 4: 'Operating System'}
>>>
>>> my_dict.clear()
>>> my_dict
{}
```

From the above code we can observe that:

- **del** lets you delete individual items from a dictionary by key.
- **clear** deletes all items from a dictionary.

Sets

`Set()` is an unordered collection with no duplicate elements. Though individual items are immutable, set itself is mutable, that is we can add or remove elements/items from the set. We can perform mathematical operations like union, intersection etc. with set.

Though sets in general can be implemented using trees, set in Python can be implemented using a hash table. This allows it a highly optimized method for checking whether a specific element is contained in the set

Creating a set

A set is created by placing all the items (elements) inside curly braces `{}`, separated by comma or by using the built-in function `set()`. Observe the following lines of code:

```
>>> #set of integers
>>> my_set = {1,2,4,8}
>>> print(my_set)
{8, 1, 2, 4}
>>>
>>> #set of mixed datatypes
>>> my_set = {1.0, "Hello World!", (2, 4, 6)}
>>> print(my_set)
{1.0, (2, 4, 6), 'Hello World!'}
```

Methods for Sets

Observe the following code to understand about methods for sets:

```
>>> >>> #METHODS FOR SETS
>>>
>>> #add(x) Method
>>> topics = {'Python', 'Java', 'C#'}
>>> topics.add('C++')
>>> topics
{'C#', 'C++', 'Java', 'Python'}
>>>
>>> #union(s) Method, returns a union of two set.
```

```

>>> topics
{'C#', 'C++', 'Java', 'Python'}
>>> team = {'Developer', 'Content Writer', 'Editor', 'Tester'}
>>> group = topics.union(team)
>>> group
{'Tester', 'C#', 'Python', 'Editor', 'Developer', 'C++', 'Java', 'Content
Writer'}
>>> # intersets(s) method, returns an intersection of two sets
>>> inters = topics.intersection(team)
>>> inters
set()
>>>
>>> # difference(s) Method, returns a set containing all the elements of
invoking set but not of the second set.
>>>
>>> safe = topics.difference(team)
>>> safe
{'Python', 'C++', 'Java', 'C#'}
>>>
>>> diff = topics.difference(group)
>>> diff
set()
>>> #clear() Method, Empties the whole set.
>>> group.clear()
>>> group
set()
>>>

```

Operators for Sets

Observe the following code to understand about operators for sets:

```

>>> # PYTHON SET OPERATIONS
>>>
>>> #Creating two sets
>>> set1 = set()
>>> set2 = set()
>>>
>>> # Adding elements to set

```

```

>>> for i in range(1,5):
    set1.add(i)
>>> for j in range(4,9):
    set2.add(j)

>>> set1
{1, 2, 3, 4}
>>> set2
{4, 5, 6, 7, 8}
>>>
>>> #Union of set1 and set2
>>> set3 = set1 | set2 # same as set1.union(set2)
>>> print('Union of set1 & set2: set3 = ', set3)
Union of set1 & set2: set3 = {1, 2, 3, 4, 5, 6, 7, 8}
>>>
>>> #Intersection of set1 & set2
>>> set4 = set1 & set2 # same as set1.intersection(set2)
>>> print('Intersection of set1 and set2: set4 = ', set4)
Intersection of set1 and set2: set4 = {4}
>>>
>>> # Checking relation between set3 and set4
>>> if set3 > set4: # set3.issuperset(set4)
    print('Set3 is superset of set4')
elif set3 < set4: #set3.issubset(set4)
    print('Set3 is subset of set4')
else: #set3 == set4
    print('Set 3 is same as set4')

Set3 is superset of set4
>>>
>>> # Difference between set3 and set4
>>> set5 = set3 - set4
>>> print('Elements in set3 and not in set4: set5 = ', set5)
Elements in set3 and not in set4: set5 = {1, 2, 3, 5, 6, 7, 8}
>>>
>>> # Check if set4 and set5 are disjoint sets
>>> if set4.isdisjoint(set5):

```

```
print('Set4 and set5 have nothing in common\n')
Set4 and set5 have nothing in common

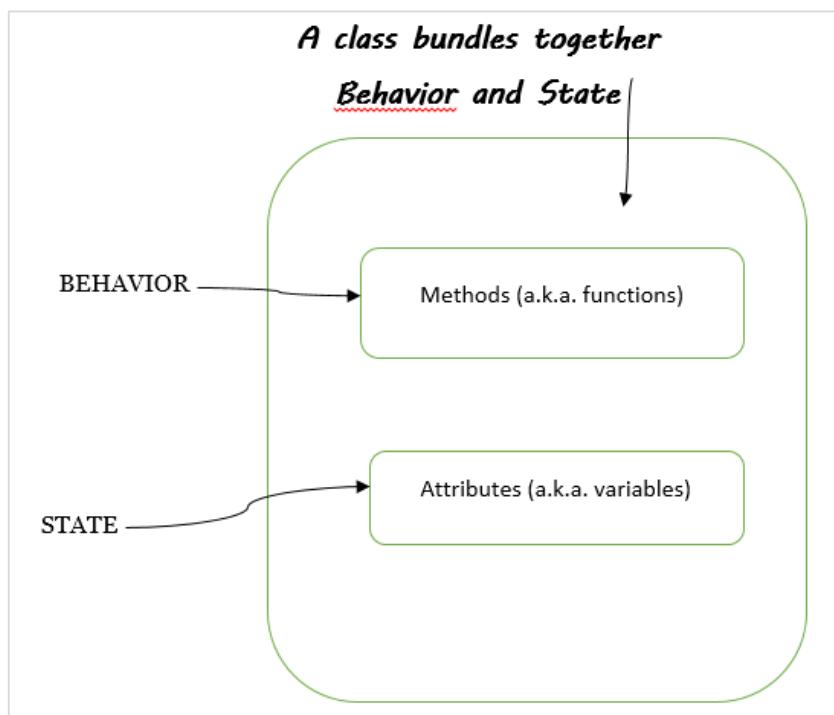
>>> # Removing all the values of set5
>>> set5.clear()
>>> set5 set()
```

4. OOP in Python – Building Blocks

In this chapter, we will discuss object oriented terms and programming concepts in detail. Class is just a factory for an instance. This factory contains the blueprint which describes how to make the instances. An instances or object are constructed from the class. In most cases, we can have more than one instances of a class. Every instance has a set of attribute and these attributes are defined in a class, so every instance of a particular class is expected to have the same attributes.

Class Bundles : Behavior and State

A class will let you bundle together the behavior and state of an object. Observe the following diagram for better understanding:



The following points are worth notable when discussing class bundles:

- The word **behavior** is identical to **function** – it is a piece of code that does something (or implements a behavior)
- The word **state** is identical to **variables** – it is a place to store values within a class.
- When we assert a class behavior and state together, it means that a class packages functions and variables.

Classes have methods and attributes

In Python, creating a method defines a class behavior. The word **method** is the OOP name given to a function that is defined within a class. To sum up:

- **Class functions** is synonym for **methods**
- **Class variables** is synonym for **name attributes**.
- **Class** – a blueprint for an instance with exact behavior.
- **Object** – one of the instances of the class, perform functionality defined in the class.
- **Type** – indicates the class the instance belongs to
- **Attribute** – Any object value: `object.attribute`
- **Method** – a “callable attribute” defined in the class

Observe the following piece of code for example:

```
var = "Hello, John"
print( type (var))      # < type 'str'> or <class 'str'>
print(var.upper())       # upper() method is called, HELLO, JOHN
```

Creation and Instantiation

The following code shows how to create our first class and then its instance.

```
class MyClass(object):
    pass

# Create first instance of MyClass
this_obj = MyClass()
print(this_obj)

# Another instance of MyClass
that_obj = MyClass()
print (that_obj)
```

Here we have created a class called **MyClass** and which does not do any task. The argument **object** in **MyClass** class involves class inheritance and will be discussed in later chapters. **pass** in the above code indicates that this block is empty, that is it is an empty class definition.

Let us create an instance **this_obj** of **MyClass()** class and print it as shown:

```
<__main__.MyClass object at 0x03B08E10>
<__main__.MyClass object at 0x0369D390>
```

Here, we have created an instance of **MyClass**. The hex code refers to the address where the object is being stored. Another instance is pointing to another address.

Now let us define one variable inside the class **MyClass()** and get the variable from the instance of that class as shown in the following code:

```
class MyClass(object):
    var = 9

    # Create first instance of MyClass
    this_obj = MyClass()
    print(this_obj.var)

    # Another instance of MyClass

    that_obj = MyClass()
    print (that_obj.var)
```

Output

You can observe the following output when you execute the code given above:

```
9
9
```

As instance knows from which class it is instantiated, so when requested for an attribute from an instance, the instance looks for the attribute and the class. This is called the **attribute lookup**.

Instance Methods

A function defined in a class is called a **method**. An instance method requires an instance in order to call it and requires no decorator. When creating an instance method, the first parameter is always **self**. Though we can call it (**self**) by any other name, it is recommended to use **self**, as it is a naming convention.

```
class MyClass(object):
    var=9

    def firstM(self):
        print("hello, World")
obj = MyClass()
print(obj.var)
obj.firstM()
```

Output

You can observe the following output when you execute the code given above:

```
9
hello, World
```

Note that in the above program, we defined a method with **self** as argument. But we cannot call the method as we have not declared any argument to it.

```
class MyClass(object):
    def firstM(self):
        print("hello, World")
        print(self)
obj = MyClass()
obj.firstM()
print(obj)
```

Output

You can observe the following output when you execute the code given above:

```
hello, World
<__main__.MyClass object at 0x036A8E10>
<__main__.MyClass object at 0x036A8E10>
```

Encapsulation

Encapsulation is one of the fundamentals of OOP. OOP enables us to hide the complexity of the internal working of the object which is advantageous to the developer in the following ways:

- Simplifies and makes it easy to understand to use an object without knowing the internals.
- Any change can be easily manageable.

Object-oriented programming relies heavily on encapsulation. The terms encapsulation and abstraction (also called data hiding) are often used as synonyms. They are nearly synonymous, as abstraction is achieved through encapsulation.

Encapsulation provides us the mechanism of restricting the access to some of the object's components, this means that the internal representation of an object can't be seen from outside of the object definition. Access to this data is typically achieved through special methods: **Getters** and **Setters**.

This data is stored in instance attributes and can be manipulated from anywhere outside the class. To secure it, that data should only be accessed using instance methods. Direct access should not be permitted.

```
class MyClass(object):
    def setAge(self, num):
        self.age = num

    def getAge(self):
        return self.age

zack = MyClass()
zack.setAge(45)
print(zack.getAge())

zack.setAge("Fourty Five")
print(zack.getAge())
```

Output

You can observe the following output when you execute the code given above:

```
45
Fourty Five
```

The data should be stored only if it is correct and valid, using Exception handling constructs. As we can see above, there is no restriction on the user input to **setAge()** method. It could be a string, a number, or a list. So we need to check onto above code to ensure correctness of being stored.

```
class MyClass(object):
    def setAge(self, num):
        self.age = num

    def getAge(self):
        return self.age
```

```
zack = MyClass()
zack.setAge(45)
```

```
print(zack.getAge())
zack.setAge("Fourty Five")
print(zack.getAge())
```

Init Constructor

The `__init__` method is implicitly called as soon as an object of a class is instantiated. This will initialize the object.

```
x = MyClass()
```

The line of code shown above will create a new instance and assigns this object to the local variable `x`.

The instantiation operation, that is **calling a class object**, creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore, a class may define a special method named '`__init__()`' as shown:

```
def __init__(self):
    self.data = []
```

Python calls `__init__` during the instantiation to define an additional attribute that should occur when a class is instantiated that may be setting up some beginning values for that object or running a routine required on instantiation. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

The `__init__()` method can have single or multiple arguments for a greater flexibility. The `init` stands for initialization, as it initializes attributes of the instance. It is called the constructor of a class.

```
class myclass(object):
    def __init__(self,aaa, bbb):
        self.a = aaa
        self.b = bbb

x = myclass(4.5, 3)
print(x.a, x.b)
```

Output

```
4.5 3
```

Class Attributes

The attribute defined in the class is called “class attributes” and the attributes defined in the function is called ‘instance attributes’. While defining, these attributes are not prefixed by `self`, as these are the property of the class and not of a particular instance.

The class attributes can be accessed by the class itself (`className.attributeName`) as well as by the instances of the class (`inst.attributeName`). So, the instances have access to both the instance attribute as well as class attributes.

```
>>> class myclass():
    age = 21
>>> myclass.age
21
>>> x = myclass()
>>> x.age
21
>>>
```

A class attribute can be overridden in an instance, even though it is not a good method to break encapsulation.

There is a lookup path for attributes in Python. The first being the method defined within the class, and then the class above it.

```
>>> class myclass(object):
    classy = 'class value'
>>> dd = myclass()
>>> print(dd.classy)      # This should return the string 'class value'
class value
>>>
>>> dd.classy = "Instance Value"
>>> print(dd.classy)      # Return the string "Instance Value"
Instance Value
>>>
>>> # This will delete the value set for 'dd.classy' in the instance.
>>> del dd.classy
>>> >>> # Since the overriding attribute was deleted, this will print 'class
value'.
>>>
>>> print(dd.classy)
class value
>>>
```

We are overriding the 'classy' class attribute in the instance **dd**. When it's overridden, the Python interpreter reads the overridden value. But once the new value is deleted with 'del', the overridden value is no longer present in the instance, and hence the lookup goes a level above and gets it from the class.

Working with Class and Instance Data

In this section, let us understand how the class data relates to the instance data. We can store data either in a class or in an instance. When we design a class, we decide which data belongs to the instance and which data should be stored into the overall class.

An instance can access the class data. If we create multiple instances, then these instances can access their individual attribute values as well the overall class data.

Thus, a class data is the data that is shared among all the instances. Observe the code given below for better understanding:

```
class InstanceCounter(object):
    count = 0                      # class attribute, will be accessible to all
    instances

    def __init__(self, val):
        self.val = val

        InstanceCounter.count +=1   # Increment the value of class attribute,
    accessible through class name

    # In above line, class ('InstanceCounter') act as an object

    def set_val(self, newval):
        self.val = newval

    def get_val(self):
        return self.val

    def get_count(self):
        return InstanceCounter.count

a = InstanceCounter(9)
b = InstanceCounter(18)
c = InstanceCounter(27)

for obj in (a, b, c):
    print ('val of obj: %s' %(obj.get_val()))      # Initialized value ( 9, 18,
27)
    print ('count: %s' %(obj.get_count()))          # always 3
```

Output

```
val of obj: 9
count: 3
val of obj: 18
count: 3
val of obj: 27
count: 3
```

In short, class attributes are same for all instances of class whereas instance attributes is particular for each instance. For two different instances, we will have two different instance attributes.

```
class myClass:
    class_attribute = 99

    def class_method(self):
        self.instance_attribute = 'I am instance attribute'

print (myClass.__dict__)
```

Output

You can observe the following output when you execute the code given above:

```
{'__module__': '__main__', 'class_attribute': 99, 'class_method': <function myClass.class_method at 0x04128D68>, '__dict__': <attribute '__dict__' of 'myClass' objects>, '__weakref__': <attribute '__weakref__' of 'myClass' objects>, '__doc__': None}
```

The instance attribute **myClass.__dict__** as shown:

```
>>> a = myClass()
>>> a.class_method()
>>> print(a.__dict__)
{'instance_attribute': 'I am instance attribute'}
```

5. OOP in Python – Object Oriented Shortcuts

This chapter talks in detail about various built-in functions in Python, file I/O operations and overloading concepts.

Python Built-in Functions

The Python interpreter has a number of functions called built-in functions that are readily available for use. In its latest version, Python contains 68 built-in functions as listed in the table given below:

BUILT-IN FUNCTIONS				
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

This section discusses some of the important functions in brief:

len() function

The len() function gets the length of strings, list or collections. It returns the length or number of items of an object, where object can be a string, list or a collection.

```
>>> len(['hello', 9 , 45.0, 24])  
4
```

`len()` function internally works like `list.__len__()` or `tuple.__len__()`. Thus, note that `len()` works only on objects that has a `__len__()` method.

```
>>> set1
{1, 2, 3, 4}
>>> set1.__len__()
4
```

However, in practice, we prefer `len()` instead of the `__len__()` function because of the following reasons:

- It is more efficient. And it is not necessary that a particular method is written to refuse access to special methods such as `__len__`.
- It is easy to maintain.
- It supports backward compatibility.

Reversed(seq)

It returns the reverse iterator. `seq` must be an object which has `__reversed__()` method or supports the sequence protocol (the `__len__()` method and the `__getitem__()` method). It is generally used in `for` loops when we want to loop over items from back to front.

```
>>> normal_list = [2, 4, 5, 7, 9]
>>>
>>> class CustomSequence():
    def __len__(self):
        return 5
    def __getitem__(self, index):
        return "x{}".format(index)
>>> class funkyback():
    def __reversed__(self):
        return 'backwards!'
>>> for seq in normal_list, CustomSequence(), funkyback():
    print('\n{}: {}'.format(seq.__class__.__name__, end=""))
        for item in reversed(seq):
            print(item, end=", ")
```

The for loop at the end prints the reversed list of a normal list, and instances of the two custom sequences. The output shows that `reversed()` works on all the three of them, but has a very different results when we define `__reversed__`.

Output

You can observe the following output when you execute the code given above:

```
list: 9, 7, 5, 4, 2,
CustomSequence: x4, x3, x2, x1, x0,
funkyback: b, a, c, k, w, a, r, d, s, !,
```

Enumerate

The **enumerate ()** method adds a counter to an iterable and returns the enumerate object.

The syntax of `enumerate ()` is:

```
enumerate(iterable, start=0)
```

Here the second argument **start** is optional, and by default index starts with **zero (0)**.

```
>>> # Enumerate
>>> names = ['Rajesh', 'Rahul', 'Aarav', 'Sahil', 'Trevor']
>>> enumerate(names)
<enumerate object at 0x031D9F80>
>>> list(enumerate(names))
[(0, 'Rajesh'), (1, 'Rahul'), (2, 'Aarav'), (3, 'Sahil'), (4, 'Trevor')]
>>>
```

So **enumerate()** returns an iterator which yields a tuple that keeps count of the elements in the sequence passed. Since the return value is an iterator, directly accessing it is not much useful. A better approach for `enumerate()` is keeping count within a for loop.

```
>>> for i, n in enumerate(names):
    print('Names number: ' + str(i))
    print(n)

Names number: 0
Rajesh
Names number: 1
Rahul
Names number: 2
Aarav
Names number: 3
Sahil
Names number: 4
Trevor
```

There are many other functions in the standard library, and here is another list of some more widely used functions:

- **hasattr**, **getattr**, **setattr** and **delattr**, which allows attributes of an object to be manipulated by their string names.
- **all** and **any**, which accept an iterable object and return **True** if all, or any, of the items evaluate to be true.
- **nzip**, which takes two or more sequences and returns a new sequence of tuples, where each tuple contains a single value from each sequence.

File I/O

The concept of files is associated with the term object-oriented programming. Python has wrapped the interface that operating systems provided in abstraction that allows us to work with file objects.

The **open()** built-in function is used to open a file and return a file object. It is the most commonly used function with two arguments:

```
open(filename, mode)
```

The **open()** function calls two argument, first is the filename and second is the mode. Here mode can be 'r' for read only mode, 'w' for only writing (an existing file with the same name will be erased), and 'a' opens the file for appending, any data written to the file is automatically added to the end. 'r+' opens the file for both reading and writing. The default mode is read only.

On windows, 'b' appended to the mode opens the file in binary mode, so there are also modes like 'rb', 'wb' and 'r+b'.

```
>>> text = 'This is the first line'
>>> file = open('datawork', 'w')
>>> file.write(text)
22
>>> file.close()
```

In some cases, we just want to append to the existing file rather than over-writing it, for that we could supply the value 'a' as a mode argument, to append to the end of the file, rather than completely overwriting existing file contents.

```
>>> f = open('datawork', 'a')
>>> text1 = ' This is second line'
>>> f.write(text1)
20
>>> f.close()
```

Once a file is opened for reading, we can call the read, readline, or readlines method to get the contents of the file. The read method returns the entire contents of the file as a str or bytes object, depending on whether the second argument is 'b'.

For readability, and to avoid reading a large file in one go, it is often better to use a for loop directly on a file object. For text files, it will read each line, one at a time, and we can process it inside the loop body. For binary files however it's better to read fixed-sized chunks of data using the read() method, passing a parameter for the maximum number of bytes to read.

```
>>> f = open('fileone', 'r+')
>>> f.readline()
'This is the first line. \n'
>>> f.readline()
'This is the second line. \n'
```

Writing to a file, through write method on file objects will writes a string (bytes for binary data) object to the file. The writelines method accepts a sequence of strings and write each of the iterated values to the file. The writelines method does not append a new line after each item in the sequence.

Finally the close() method should be called when we are finished reading or writing the file, to ensure any buffered writes are written to the disk, that the file has been properly cleaned up and that all resources tied with the file are released back to the operating system. It's a better approach to call the close() method but technically this will happen automatically when the script exists.

An alternative to method overloading

Method overloading refers to having multiple methods with the same name that accept different sets of arguments.

Given a single method or function, we can specify the number of parameters ourself. Depending on the function definition, it can be called with zero, one, two or more parameters.

```
class Human:
    def sayHello(self, name=None):
        if name is not None:
            print('Hello ' + name)
        else:
            print('Hello ')

#Create Instance
```

```

obj = Human()

#Call the method, else part will be executed
obj.sayHello()

#Call the method with a parameter, if part will be executed
obj.sayHello('Rahul')

```

Output

```

Hello
Hello Rahul

```

Default Arguments

Functions Are Objects Too

A callable object is an object can accept some arguments and possibly will return an object. A function is the simplest callable object in Python, but there are others too like classes or certain class instances.

Every function in a Python is an object. Objects can contain methods or functions but object is not necessary a function.

```

def my_func():
    print('My function was called')
my_func.description = 'A silly function'

def second_func():

    print('Second function was called')

second_func.description = 'One more sillier function'

def another_func(func):
    print("The description:", end=" ")
    print(func.description)
    print('The name: ', end=' ')
    print(func.__name__)
    print('The class:', end=' ')

```

```

print(func.__class__)
print("Now I'll call the function passed in")
func()

another_func(my_func)
another_func(second_func)

```

In the above code, we are able to pass two different functions as argument into our third function, and get different output for each one:

```

The description: A silly function
The name: my_func
The class: <class 'function'>
Now I'll call the function passed in
My function was called

The description: One more sillier function
The name: second_func
The class: <class 'function'>
Now I'll call the function passed in
Second function was called

```

callable objects

Just as functions are objects that can have attributes set on them, it is possible to create an object that can be called as though it were a function.

In Python any object with a `__call__()` method can be called using function-call syntax.

6. OOP in Python – Inheritance and Polymorphism

Inheritance and polymorphism – this is a very important concept in Python. You must understand it better if you want to learn.

Inheritance

One of the major advantages of Object Oriented Programming is re-use. Inheritance is one of the mechanisms to achieve the same. Inheritance allows programmer to create a general or a base class first and then later extend it to more specialized class. It allows programmer to write better code.

Using inheritance you can use or inherit all the data fields and methods available in your base class. Later you can add your own methods and data fields, thus inheritance provides a way to organize code, rather than rewriting it from scratch.

In object-oriented terminology when class X extends class Y, then Y is called super/parent/base class and X is called subclass/child/derived class. One point to note here is that only data fields and method which are not private are accessible by child classes. Private data fields and methods are accessible only inside the class.

Syntax to create a derived class is:

```
class BaseClass:  
    Body of base class  
  
class DerivedClass(BaseClass):  
    Body of derived class
```

Inheriting Attributes

Now look at the below example:

```
# Inheriting_Attributes  
  
class Date(object):          # Inherits from the 'object' Class  
    def get_date(self):  
        return '2018-02-02'  
  
class Time(Date):           # Inherits from the 'Date' Class  
    def get_time(self):  
        return '09:00:00'  
  
dt = Date()  
print("Get date from the Date class: ",dt.get_date())  
  
tm = Time()  
print("Get time from the Time class: ",tm.get_time())  
print('Get date from time class by inheriting or calling Date class method: ',tm.get_date())      # Found this method in the 'Date' class
```

Output

```
Get date from the Date class: 2018-02-02
Get time from the Time class: 09:00:00
Get date from time class by inheriting or calling Date class method: 2018-02-02
```

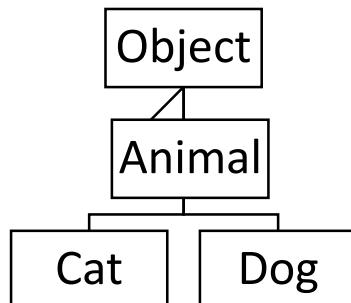
We first created a class called Date and pass the object as an argument, here-object is built-in class provided by Python. Later we created another class called time and called the Date class as an argument. Through this call we get access to all the data and attributes of Date class into the Time class. Because of that when we try to get the get_date method from the Time class object tm we created earlier possible.

Object.Attribute Lookup Hierarchy

- The instance
- The class
- Any class from which this class inherits

Inheritance Examples

Let's take a closure look into the inheritance example:



Let's create couple of classes to participate in examples:

- Animal: Class simulate an animal
- Cat: Subclass of Animal
- Dog: Subclass of Animal

In Python, constructor of class used to create an object (instance), and assign the value for the attributes.

Constructor of subclasses always called to a constructor of parent class to initialize value for the attributes in the parent class, then it start assign value for its attributes.

```
#Inheritance Example
class Animal(object):

    def __init__(self, name):
        self.name = name
    def eat(self, food):          # Common method(or property) of both subclass
        print ('%s is eating %s.' %(self.name, food))

class Dog(Animal):

    def fetch(self, thing):
        print('%s goes after the %s!' %(self.name, thing))

class Cat(Animal):

    def swatstring(self):
        print('%s shreds the string!' %(self.name))

d = Dog('Ranger')      # Created Dog object, d
c = Cat('MeOw')        # Created Cat object, c

d.fetch('ball')         # Rover goes after the paper!=
c.swatstring()          # Fluffy shreds the string!
d.eat('Dog Food')      # Rover is eating dog Food
c.eat('Cat Food')      #Fluffy is eating cat food.
d.swatstring()          #Attribute Error: 'Dog' object has no Attribute 'swatstring'
```

Output

```
Ranger goes after the ball!
MeOw shreds the string!
Ranger is eating Dog Food.
MeOw is eating Cat Food.
Traceback (most recent call last):
  File "animal.py", line 27, in <module>
    d.swatstring()          #Attribute Error: 'Dog' object has no Attribute 'swatstring'
AttributeError: 'Dog' object has no attribute 'swatstring'
```

In the above example, we see the command attributes or methods we put in the parent class so that all subclasses or child classes will inherits that property from the parent class.

If a subclass try to inherits methods or data from another subclass then it will through an error as we see when Dog class try to call swatstring() methods from that cat class, it throws an error(like AttributeError in our case).

Polymorphism (“MANY SHAPES”)

Polymorphism is an important feature of class definition in Python that is utilized when you have commonly named methods across classes or subclasses. This permits functions to use entities of different types at different times. So, it provides flexibility and loose coupling so that code can be extended and easily maintained over time.

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.

Let understand the concept of polymorphism with our previous inheritance example and add one common method called `show_affection` in both subclasses:

From the example we can see, it refers to a design in which object of dissimilar type can be treated in the same manner or more specifically two or more classes with method of the same name or common interface because same method(`show_affection` in below example) is called with either type of objects.

```
#Polymorphism Example
class Animal(object):

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

    def eat(self, food):          # Common method(or property) of both subclass
        print ('{} eats {}'.format(self.name, food))

class Dog(Animal):

    def fetch(self, thing):
        print ('{} goes after the {}'.format(self.name, thing))

    def show_affection(self):     # Same method is called in Dog class
        print ('{} wags tail'.format(self.name))

class Cat(Animal):

    def swatstring(self):
        print ('{} shreds the string!'.format(self.name))

    def show_affection(self):     # Same method is called in Cat class
        print ('{} purrs'.format(self.name))
| 

for a in (Dog('Rover'), Cat('Fluffy'), Cat('Precious'), Dog('Scout')): a.show_affection()      # Same method is called with different attribute
```

Output

```
Rover wags tail
Fluffy purrs
Precious purrs
Scout wags tail
```

So, all animals show affections (`show_affection`), but they do differently. The “`show_affection`” behaviors is thus polymorphic in the sense that it acted differently depending on the animal. So, the abstract “animal” concept does not actually “`show_affection`”, but specific animals(like dogs and cats) have a concrete implementation of the action “`show_affection`”.

Python itself have classes that are polymorphic. Example, the len() function can be used with multiple objects and all return the correct output based on the input parameter.

```
>>> len('hello') # passing a string to len function
5
>>> len([1, 2, 3]) # passing a list of 3 elements
3
>>> len(('x', 'y', 'z')) # passing a tuple of 3 elements
3
>>> len({'a':9, 'b':18}) # passing a dictionary of 2 pairs
2
>>> # illustrate it further,
>>> dir('hello')
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>>
>>> dir([1,2,3])
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> dir({'a':9})
['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
>>>
```

Overriding

In Python, when a subclass contains a method that overrides a method of the superclass, you can also call the superclass method by calling Super(Subclass, self).method instead of self.method.

Example

```
class Thought(object):
    def __init__(self):
        pass
    def message(self):
        print("Thought, always come and go")

class Advice(Thought):
    def __init__(self):
        super(Advice, self).__init__()
    def message(self):
        print('Warning: Risk is always involved when you are dealing with market!')
```

Inheriting the Constructor

If we see from our previous inheritance example, `__init__` was located in the parent class in the up 'cause the child class dog or cat didn't've `__init__` method in it. Python used the inheritance attribute lookup to find `__init__` in animal class. When we created the child class, first it will look the `__init__` method in the dog class, then it didn't find it then looked into parent class Animal and found there and called that there. So as our class design became complex we may wish to initialize a instance firstly processing it through parent class constructor and then through child class constructor.

```
import random

class Animal(object):

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

class Dog(Animal):

    def __init__(self, name):
        super(Dog, self).__init__(name)
        self.breed = random.choice(['Doberman', 'German shepherd', 'Beagle'])

    def fetch(self, thing):
        print('%s goes after the %s!' %(self.name, thing))

d = Dog('dogname')

print(d.name)
print(d.breed)
```

Output

```
dogname
German shepherd
```

In above example- all animals have a name and all dogs a particular breed. We called parent class constructor with super. So dog has its own `__init__` but the first thing that happen is we call super. Super is built in function and it is designed to relate a class to its super class or its parent class.

In this case we saying that get the super class of dog and pass the dog instance to whatever method we say here the constructor `__init__`. So in another words we are calling parent class Animal `__init__` with the dog object. You may ask why we won't just say Animal `__init__` with the dog instance, we could do this but if the name of animal class were to change, sometime in the future. What if we wanna rearrange the class hierarchy,

so the dog inherited from another class. Using super in this case allows us to keep things modular and easy to change and maintain.

So in this example we are able to combine general `__init__` functionality with more specific functionality. This gives us opportunity to separate common functionality from the specific functionality which can eliminate code duplication and relate class to one another in a way that reflects the system overall design.

Conclusion

- `__init__` is like any other method; it can be inherited
- If a class does not have a `__init__` constructor, Python will check its parent class to see if it can find one.
- As soon as it finds one, Python calls it and stops looking
- We can use the `super()` function to call methods in the parent class.
- We may want to initialize in the parent as well as our own class.

Multiple Inheritance and the Lookup Tree

As its name indicates, multiple inheritance in Python is when a class inherits from multiple classes.

For example, a child inherits personality traits from both parents (Mother and Father).

Python Multiple Inheritance Syntax

To make a class inherits from multiple parents classes, we write the names of these classes inside the parentheses to the derived class while defining it. We separate these names with comma.

Below is an example of that:

```
>>> class Mother:
    pass

>>> class Father:
    pass

>>> class Child(Mother, Father):
    pass

>>> issubclass(Child, Mother) and issubclass(Child, Father)
True
```

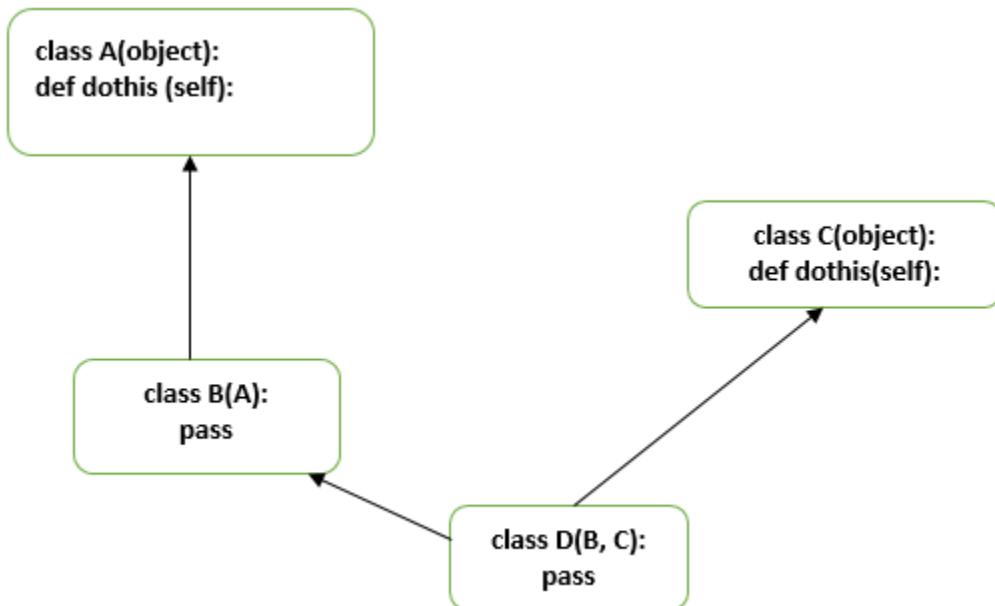
Multiple inheritance refers to the ability of inheriting from two or more than two class. The complexity arises as child inherits from parent and parents inherits from the grandparent class. Python climbs an inheriting tree looking for attributes that is being requested to be read from an object. It will check the in the instance, within class then

parent class and lastly from the grandparent class. Now the question arises in what order the classes will be searched - breath-first or depth-first. By default, Python goes with the depth-first.

That's why in the below diagram the Python searches the dothis() method first in class A. So the method resolution order in the below example will be

Mro- D->B->A->C

Look at the below multiple inheritance diagram:



Let's go through an example to understand the "mro" feature of an Python.

```
class A(object):
    def dothis(self):
        print('doing this in A')

class B(A):
    pass

class C(object):
    def dothis(self):
        print('do this in C')

class D(B,C):
    pass

d_instance = D()

d_instance.dothis()

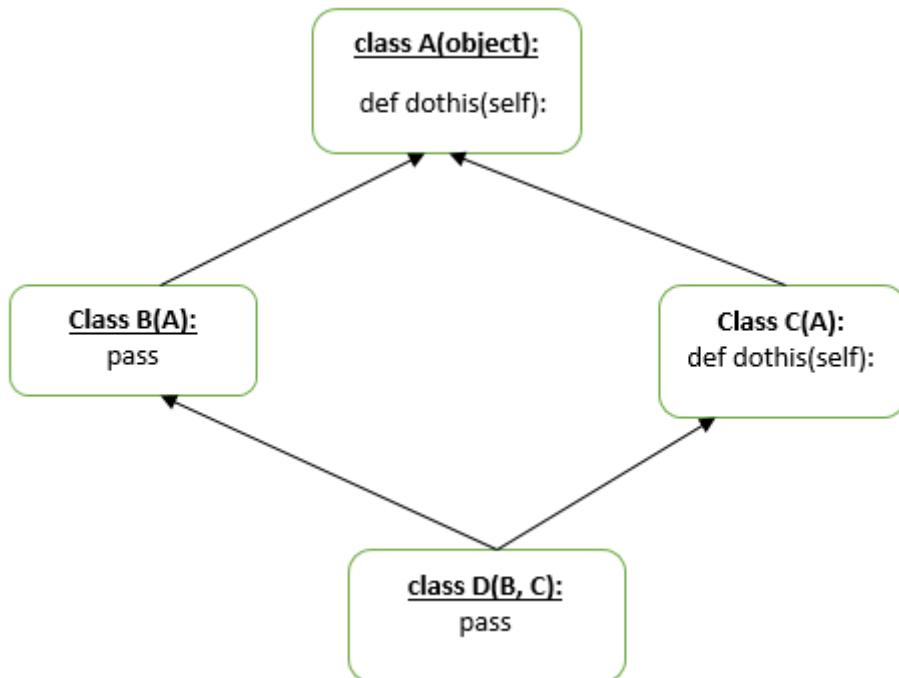
print(D.mro())
```

Output

```
doing this in A
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.A'>, <class '__main__.C'>, <class 'object'>]
```

Example 3

Let's take another example of "diamond shape" multiple inheritance.



Above diagram will be considered ambiguous. From our previous example understanding "method resolution order" .i.e. mro will be D->B->A->C->A but it's not. On getting the second A from the C, Python will ignore the previous A. so the mro will be in this case will be D->B->C->A.

Let's create an example based on above diagram:

```
class A(object):
    def dothis(self):
        print('doing this in A')

class B(A):
    pass

class C(A):
    def dothis(self):
        print('do this in C')

class D(B,C):
    pass

d_instance = D()
d_instance.dothis()

print(D.mro())
```

Output

```
do this in C
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

Simple rule to understand the above output is- if the same class appear in the method resolution order, the earlier appearances of this class will be remove from the method resolution order.

In conclusion:

- Any class can inherit from multiple classes
- Python normally uses a "depth-first" order when searching inheriting classes.
- But when two classes inherit from the same class, Python eliminates the first appearances of that class from the mro.

Decorators, Static and Class Methods

Functions(or methods) are created by def statement.



Though methods works in exactly the same way as a function except one point where method first argument is instance object.

We can classify methods based on how they behave, like

- **Simple method:** defined outside of a class. This function can access class attributes by feeding instance argument:

```
def outside_func():
```

- **Instance method:**

```
def func(self,)
```

- **Class method:** if we need to use class attributes

```
@classmethod  
def cfunc(cls,)
```

- **Static method:** do not have any info about the class

```
@staticmethod  
def sfoo()
```

Till now we have seen the instance method, now is the time to get some insight into the other two methods,

Class Method

The `@classmethod` decorator, is a builtin function decorator that gets passed the class it was called on or the class of the instance it was called on as first argument. The result of that evaluation shadows your function definition.

Syntax

```
class C(object):  
    @classmethod  
    def fun(cls, arg1, arg2, ...):  
        ....  
    fun: function that needs to be converted into a class method  
    returns: a class method for function
```

They have the access to this `cls` argument, it can't modify object instance state. That would require access to `self`.

- It is bound to the class and not the object of the class.
- Class methods can still modify class state that applies across all instances of the class.

Static Method

A static method takes neither a `self` nor a `cls(class)` parameter but it's free to accept an arbitrary number of other parameters.

Syntax

```
class C(object):
    @staticmethod
    def fun(arg1, arg2, ...):
        ...

returns: a static method for function funself.
```

- A static method can neither modify object state nor class state.
- They are restricted in what data they can access.

When to use what

- We generally use class method to create factory methods. Factory methods return class object (similar to a constructor) for different use cases.
- We generally use static methods to create utility functions.

7. OOP in Python –Python Design Pattern

Overview

Modern software development needs to address complex business requirements. It also needs to take into account factors such as future extensibility and maintainability. A good design of a software system is vital to accomplish these goals. Design patterns play an important role in such systems.

To understand design pattern, let's consider below example-

- Every car's design follows a basic design pattern, four wheels, steering wheel, the core drive system like accelerator-break-clutch, etc.

So, all things repeatedly built/ produced, shall inevitably follow a pattern in its design.. it cars, bicycle, pizza, atm machines, whatever...even your sofa bed.

Designs that have almost become standard way of coding some logic/mechanism/technique in software, hence come to be known as or studied as, Software Design Patterns.

Why is Design Pattern Important?

Benefits of using Design Patterns are:

- Helps you to solve common design problems through a proven approach
- No ambiguity in the understanding as they are well documented.
- Reduce the overall development time.
- Helps you deal with future extensions and modifications with more ease than otherwise.
- May reduce errors in the system since they are proven solutions to common problems.

Classification of Design Patterns

The GoF (Gang of Four) design patterns are classified into three categories namely creational, structural and behavioral.

Creational Patterns

Creational design patterns separate the object creation logic from the rest of the system. Instead of you creating objects, creational patterns creates them for you. The creational patterns include Abstract Factory, Builder, Factory Method, Prototype and Singleton.

Creational Patterns are not commonly used in Python because of the dynamic nature of the language. Also language itself provide us with all the flexibility we need to create in a sufficient elegant fashion, we rarely need to implement anything on top, like singleton or Factory.

Also these patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using a new operator.

Structural Patterns

Sometimes instead of starting from scratch, you need to build larger structures by using an existing set of classes. That's where structural class patterns use inheritance to build a new structure. Structural object patterns use composition/ aggregation to obtain a new functionality. Adapter, Bridge, Composite, Decorator, Façade, Flyweight and Proxy are Structural Patterns. They offer best ways to organize class hierarchy.

Behavioral Patterns

Behavioral patterns offer best ways of handling communication between objects. Patterns come under this categories are: Visitor, Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy and Template method are Behavioral Patterns.

Because they represent the behavior of a system, they are used generally to describe the functionality of software systems.

Commonly used Design Patterns

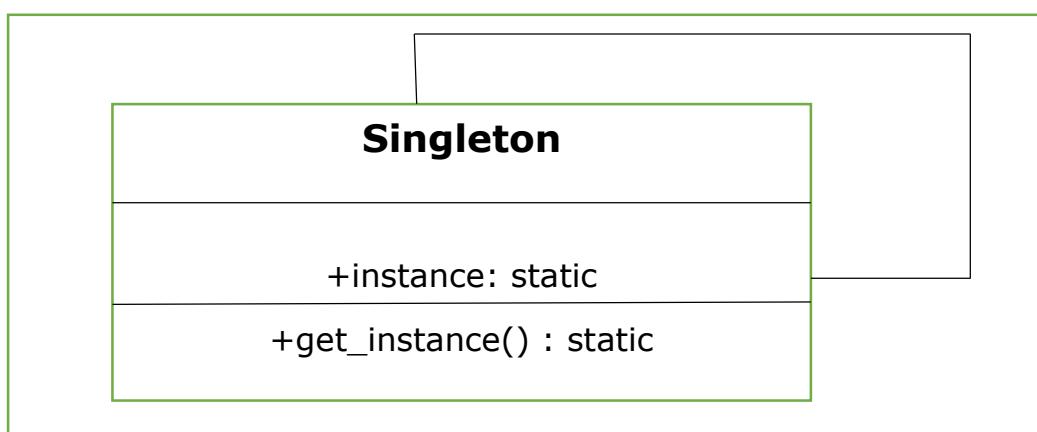
Singleton

It is one of the most controversial and famous of all design patterns. It is used in overly object-oriented languages, and is a vital part of traditional object-oriented programming.

The Singleton pattern is used for,

- When logging needs to be implemented. The logger instance is shared by all the components of the system.
- The configuration files use this because cache of information needs to be maintained and shared by all the various components in the system.
- Managing a connection to a database.

Here is the UML diagram,



```

class Logger(object):
    def __new__(cls, *args, **kwargs):
        if not hasattr(cls, '_logger'):
            ...
  
```

```
cls._logger = super(Logger, cls).__new__(cls, *args, **kwargs)
return cls._logger
```

In this example, Logger is a Singleton.

When `__new__` is called, it normally constructs a new instance of that class. When we override it, we first check if our singleton instance has been created or not. If not, we create it using a super call. Thus, whenever we call the constructor on Logger, we always get the exact same instance.

```
>>>
>>> obj1 = Logger()
>>> obj2 = Logger()
>>> obj1 == obj2
True
>>>
>>> obj1
<__main__.Logger object at 0x03224090>
>>> obj2
<__main__.Logger object at 0x03224090>
```

8. OOP in Python – Advanced Features

In this we will look into some of the advanced features which Python provide

Core Syntax in our Class design

In this we will look onto, how Python allows us to take advantage of operators in our classes. Python is largely objects and methods call on objects and this even goes on even when its hidden by some convenient syntax.

```
>>> var1 = 'Hello'  
>>> var2 = ' World!'  
>>> var1 + var2  
'Hello World!'  
>>>  
>>> var1.__add__(var2)  
'Hello World!'  
>>> num1 = 45  
>>> num2 = 60  
>>> num1.__add__(num2)  
105  
>>> var3 = ['a', 'b']  
>>> var4 = ['hello', ' John']  
>>> var3.__add__(var4)  
['a', 'b', 'hello', ' John']
```

So if we have to add magic method `__add__` to our own classes, could we do that too. Let's try to do that.

We have a class called Sumlist which has a contructor `__init__` which takes list as an argument called `my_list`.

```
class SumList(object):  
    def __init__(self, my_list):  
        self.mylist = my_list  
    def __add__(self, other):
```

```

new_list = [ x + y for x, y in zip(self.mylist, other.mylist) ]

return SumList(new_list)

def __repr__(self):
    return str(self.mylist)

aa = SumList([3,6, 9, 12, 15])

bb = SumList([100, 200, 300, 400, 500])
cc = aa + bb      # aa.__add__(bb)
print(cc)          # should gives us a list ([103, 206, 309, 412, 515])

```

Output

```
[103, 206, 309, 412, 515]
```

But there are many methods which are internally managed by others magic methods. Below are some of them,

```

'abc' in var      # var.__contains__('abc')

var == 'abc'      # var.__eq__('abc')

var[1]            # var.__getitem__(1)

var[1:3]          # var.__getslice__(1, 3)

len(var)          # var.__len__()

print(var)         # var.__repr__()

```

Inheriting From built-in types

Classes can also inherit from built-in types this means inherits from any built-in and take advantage of all the functionality found there.

In below example we are inheriting from dictionary but then we are implementing one of its method `__setitem__`. This (`setitem`) is invoked when we set key and value in the dictionary. As this is a magic method, this will be called implicitly.

```

class MyDict(dict):

    def __setitem__(self, key, val):
        print('setting a key and value!')
        dict.__setitem__(self, key, val)

dd = MyDict()
dd[ 'a' ] = 10
dd[ 'b' ] = 20

for key in dd.keys():
    print('{0}={1}'.format(key, dd[key]))

```

Output:

```

setting a key and value!
setting a key and value!
a=10
b=20

```

Let's extend our previous example, below we have called two magic methods called `__getitem__` and `__setitem__` better invoked when we deal with list index.

```

# Mylist inherits from 'list' object but indexes from 1 instead for 0!
class Mylist(list):          # inherits from list

    def __getitem__(self, index):
        if index == 0:
            raise IndexError
        if index > 0:
            index = index - 1
            return list.__getitem__(self, index)      # this method is called
when

# we access a value with subscript like x[1]
def __setitem__(self, index, value):
    if index == 0:
        raise IndexError
    if index > 0:
        index = index - 1

```

```

list.__setitem__(self, index, value)

x = Mylist(['a', 'b', 'c'])      # __init__() inherited from builtin list

print(x)                      # __repr__() inherited from builtin list

x.append('HELLO');  # append() inherited from builtin list

print(x[1])                  # 'a' (Mylist.__getitem__ customizes list superclass
                            # method. index is 1, but reflects 0!

print (x[4])                 # 'HELLO' (index is 4 but reflects 3!

```

Output

```

['a', 'b', 'c']
a
HELLO

```

In above example, we set a three item list in Mylist and implicitly `__init__` method is called and when we print the element `x`, we get the three item list (`['a','b','c']`). Then we append another element to this list. Later we ask for index 1 and index 4. But if you see the output, we are getting element from the (index-1) what we have asked for. As we know list indexing start from 0 but here the indexing start from 1 (that's why we are getting the first item of the list).

Naming Conventions

In this we will look into names we'll used for variables especially private variables and conventions used by Python programmers worldwide. Although variables are designated as private but there is not privacy in Python and this by design. Like any other well documented languages, Python has naming and style conventions that it promote although it doesn't enforce them. There is a style guide written by **“Guido van Rossum” the originator of Python, that describe the best practices and use of name and is called PEP8. Here is the link for this,**

<https://www.Python.org/dev/peps/pep-0008/>

PEP stands for Python enhancement proposal and is a series of documentation that distributed among the Python community to discuss proposed changes. For example it is recommended all,

- Module names : all_lower_case
- Class names and exception names: CamelCase
- Global and local names: all_lower_case
- Functions and method names: all_lower_case
- Constants: ALL_UPPER_CASE

These are just the recommendation, you can vary if you like. But as most of the developers follows these recommendation so might me your code is less readable.

Why conform to convention?

We can follow the PEP recommendation we it allows us to get,

- More familiar to the vast majority of developers
- Clearer to most readers of your code.
- Will match style of other contributers who work on same code base.
- Mark of a professional software developers
- Everyone will accept you.

Variable Naming: ‘Public’ and ‘Private’

In Python, when we are dealing with modules and classes, we designate some variables or attribute as private. In Python, there is no existence of “Private” instance variable which cannot be accessed except inside an object. Private simply means they are simply not intended to be used by the users of the code instead they are intended to be used internally. In general, a convention is being followed by most Python developers i.e. a name prefixed with an underscore for example. _attrval (example below) should be treated as a non-public part of the API or any Python code, whether it is a function, a method or a data member. Below is the naming convention we follow,

- Public attributes or variables (intended to be used by the importer of this module or user of this class): **regular_lower_case**
- Private attributes or variables (internal use by the module or class): **_single_leading_underscore**
- Private attributes that shouldn’t be subclassed: **__double_leading_underscore**
- Magic attributes: **__double_underscores__** (use them, don’t create them)

```
class GetSet(object):

    instance_count = 0      # public

    __mangled_name = 'no privacy!' # special variable
```

9. OOP in Python – Files and Strings

```
def __init__(self, value):
    self._attrval = value      # _attrval is for internal use only
    GetSet.instance_count += 1

@property
def var(self):
    print('Getting the "var" attribute')
    return self._attrval

@var.setter
def var(self, value):
    print('setting the "var" attribute')
    self._attrval = value

@var.deleter
def var(self):
    print('deleting the "var" attribute')
    self._attrval = None

cc = GetSet(5)
cc.var = 10      # public name
print(cc._attrval)
print(cc._GetSet__mangled_name)
```

Output

```
setting the "var" attribute
10
no privacy!
```

Strings

Strings are the most popular data types used in every programming language. Why? Because we, understand text better than numbers, so in writing and talking we use text and words, similarly in programming too we use strings. In string we parse text, analyse text semantics, and do data mining – and all this data is human consumed text. The string in Python is immutable.

String Manipulation

In Python, string can be marked in multiple ways, using single quote (`), double quote(") or even triple quote (` ` `) in case of multiline strings.

```
>>> # String Examples
>>> a = "hello"
>>> b = ''' A Multi line string,
Simple!'''
>>> e = ('Multiple' 'strings' 'togethers')
```

String manipulation is very useful and very widely used in every language. Often, programmers are required to break down strings and examine them closely.

Strings can be iterated over (character by character), sliced, or concatenated. The syntax is the same as for lists.

The str class has numerous methods on it to make manipulating strings easier. The dir and help commands provides guidance in the Python interpreter how to use them.

Below are some of the commonly used string methods we use.

METHODS	DESCRIPTION
isalpha()	Checks if all characters are Alphabets
isdigit()	Checks Digit Characters
isdecimal()	Checks decimal Characters
isnumeric()	checks Numeric Characters
find()	Returns the Highest Index of substrings.
istitle()	Checks for Titlecased strings
join()	Returns a concatenated string.
lower()	returns lower cased string
upper()	returns upper cased string
partition()	Returns a tuple
bytearray()	Returns array of given byte size.

enumerate()	Returns an enumerate object.
isprintable()	Checks printable character.

Let's try to run couple of string methods,

```
>>> str1 = 'Hello World!'
>>> str1.startswith('h')
False
>>> str1.startswith('H')
True
>>> str1.endswith('d')
False
>>> str1.endswith('d!')
True
>>> str1.find('o')
4
>>> #Above returns the index of the first occurrence of the character/substring.
>>> str1.find('lo')
3
>>> str1.upper()
'HELLO WORLD!'
>>> str1.lower()
'hello world!'
>>> str1.index('b')
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    str1.index('b')
ValueError: substring not found
>>> s = ('hello How Are You')
>>> s.split(' ')
['hello', 'How', 'Are', 'You']
>>> s1=s.split(' ')
>>> '*'.join(s1)
'hello*How*Are*You'
>>> s.partition(' ')
('hello', ' ', 'How Are You')
```

```
>>>
```

String Formatting

In Python 3.x formatting of strings has changed, now it more logical and is more flexible. Formatting can be done using the `format()` method or the `%` sign(old style) in format string.

The string can contain literal text or replacement fields delimited by braces `{}` and each replacement field may contains either the numeric index of a positional argument or the name of a keyword argument.

Syntax

```
str.format(*args, **kwargs)
```

Basic Formatting

```
>>> '{} {}'.format('Example', 'One')
'Example One'
>>> '{} {}'.format('pie', '3.1415926')
'pie 3.1415926'
```

Below example allows re-arrange the order of display without changing the arguments.

```
>>> '{1} {0}'.format('pie', '3.1415926')
'3.1415926 pie'
```

Padding and aligning strings

A value can be padded to a specific length.

```
>>> #Padding Character, can be space or special character
>>> '{:12}'.format('PYTHON')
'PYTHON      '
>>> '{:>12}'.format('PYTHON')
'      PYTHON'
>>> '{:<{}s}'.format('PYTHON',12)
'PYTHON      '
>>> '{:*<12}'.format('PYTHON')
'PYTHON*****'
```

```

>>> '{:.*^12}'.format('PYTHON')
'***PYTHON***'

>>> '{:.15}'.format('PYTHON OBJECT ORIENTED PROGRAMMING')
'PYTHON OBJECT O'

>>> #Above, truncated 15 characters from the left side of a specified string
>>> '{:.{}{}'.format('PYTHON OBJECT ORIENTED',15)
'PYTHON OBJECT O'

>>> #Named Placeholders
>>> data = {'Name':'Raghu', 'Place':'Bangalore'}
>>> '{Name} {Place}'.format(**data)
'Raghu Bangalore'

>>> #Datetime
>>> from datetime import datetime
>>> '{:%Y/%m/%d.%H:%M}'.format(datetime(2018,3,26,9,57))
'2018/03/26.09:57'

```

Strings are Unicode

Strings as collections of immutable Unicode characters. Unicode strings provide an opportunity to create software or programs that works everywhere because the Unicode strings can represent any possible character not just the ASCII characters.

Many IO operations only know how to deal with bytes, even if the bytes object refers to textual data. It is therefore very important to know how to interchange between bytes and Unicode.

Converting text to bytes

Converting a strings to byte object is termed as encoding. There are numerous forms of encoding, most common ones are: PNG; JPEG, MP3, WAV, ASCII, UTF-8 etc. Also this(encoding) is a format to represent audio, images, text, etc. in bytes.

This conversion is possible through encode(). It take encoding technique as argument. By default, we use ‘UTF-8’ technique.

```

>>> # Python Code to demonstrate string encoding
>>>
>>> # Initialising a String
>>> x = 'TutorialsPoint'
>>>
>>> #Initialising a byte object
>>> y = b'TutorialsPoint'

```

```

>>>
>>> # Using encode() to encode the String
>>> # encoded version of x is stored in z using ASCII mapping
>>> z = x.encode('ASCII')
>>>
>>> # Check if x is converted to bytes or not
>>>
>>> if(z==y):
    print('Encoding Successful!')
else:
    print('Encoding Unsuccessful!')

```

Encoding Successful!

Converting bytes to text

Converting bytes to text is called the decoding. This is implemented through decode(). We can convert a byte string to a character string if we know which encoding is used to encode it.

So Encoding and decoding are inverse processes.

```

>>>
>>> # Python code to demonstrate Byte Decoding
>>>
>>> #Initialise a String
>>> x = 'TutorialsPoint'
>>>
>>> #Initialising a byte object
>>> y = b'TutorialsPoint'
>>>
>>> #using decode() to decode the Byte object
>>> # decoded version of y is stored in z using ASCII mapping
>>> z = y.decode('ASCII')

```

>>>

```
>>> #Check if y is converted to String or not
>>> if(z == x):
    print('Decoding Successful!')
else:
    print('Decoding Unsuccessful!')

Decoding Successful!
>>>
```

File I/O

Operating systems represents files as a sequence of bytes, not text.

A file is a named location on disk to store related information. It is used to permanently store data in your disk.

In Python, a file operation takes place in the following order.

- Open a file
- Read or write onto a file (operation).
- Close the file.

Python wraps the incoming (or outgoing) stream of bytes with appropriate decode (or encode) calls so we can deal directly with str objects.

Opening a file

Python has a built-in function `open()` to open a file. This will generate a file object, also called a handle as it is used to read or modify the file accordingly.

```
>>> f = open(r'c:\\users\\rajesh\\Desktop\\index.webm','rb')
>>> f
<_io.BufferedReader name='c:\\users\\rajesh\\Desktop\\index.webm'>
>>> f.mode
'rb'
>>> f.name
'c:\\users\\rajesh\\Desktop\\index.webm'
```

For reading text from a file, we only need to pass the filename into the function. The file will be opened for reading, and the bytes will be converted to text using the platform default encoding.

10. OOP in Python – Exception and Exception Classes

In general, an exception is any unusual condition. Exception usually indicates errors but sometimes they intentionally puts in the program, in cases like terminating a procedure early or recovering from a resource shortage. There are number of built-in exceptions, which indicate conditions like reading past the end of a file, or dividing by zero. We can define our own exceptions called custom exception.

Exception handling enables you handle errors gracefully and do something meaningful about it. Exception handling has two components: "throwing" and 'catching'.

Identifying Exception (Errors)

Every error occurs in Python result an exception which will an error condition identified by its error type.

```
>>> #Exception
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    1/0
ZeroDivisionError: division by zero

>>>
>>> var=20
>>> print(ver)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    print(ver)
NameError: name 'ver' is not defined

>>> #Above as we have misspelled a variable name so we get an NameError.
>>>
>>> print('hello')

SyntaxError: EOL while scanning string literal
>>> #Above we have not closed the quote in a string, so we get SyntaxError.

>>>
>>> #Below we are asking for a key, that doen't exists.
```

```

>>> mydict = {}
>>> mydict['x']
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    mydict['x']
KeyError: 'x'

>>> #Above KeyError

>>>

>>> #Below asking for a index that didn't exist in a list.

>>> mylist = [1,2,3,4]
>>> mylist[5]
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    mylist[5]
IndexError: list index out of range
>>> #Above, index out of range, raised IndexError.

```

Catching/Trapping Exception

When something unusual occurs in your program and you wish to handle it using the exception mechanism, you 'throw an exception'. The keywords try and except are used to catch exceptions. Whenever an error occurs within a try block, Python looks for a matching except block to handle it. If there is one, execution jumps there.

Syntax:

```

try:
    #write some code
    #that might throw some exception
except <ExceptionType>:
    # Exception handler, alert the user

```

The code within the try clause will be executed statement by statement.

If an exception occurs, the rest of the try block will be skipped and the except clause will be executed.

```

try:
    some statement here
except:
    exception handling

```

Let's write some code to see what happens when you not use any error handling mechanism in your program.

```
number = int(input('Please enter the number between 1 & 10: '))
print('You have entered number',number)
```

Above programme will work correctly as long as the user enters a number, but what happens if the users try to puts some other data type(like a string or a list).

```
Please enter the number between 1 & 10: 'Hi'
Traceback (most recent call last):
  File "C:/Python/Python361/exception2.py", line 1, in <module>
    number = int(input('Please enter the number between 1 & 10: '))
ValueError: invalid literal for int() with base 10: "'Hi'"
```

Now ValueError is an exception type. Let's try to rewrite the above code with exception handling.

```
import sys

print('Previous code with exception handling')

try:
    number = int(input('Enter number between 1 & 10: '))

except(ValueError):
    print('Error..numbers only')
    sys.exit()

print('You have entered number: ',number)
```

If we run the program, and enter a string (instead of a number), we can see that we get a different result.

```
Previous code with exception handling
Enter number between 1 & 10: 'Hi'
Error..numbers only
```

Raising Exceptions

To raise your exceptions from your own methods you need to use raise keyword like this

```
raise ExceptionClass('Some Text Here')
```

Let's take an example

```
def enterAge(age):
    if age<0:
        raise ValueError('Only positive integers are allowed')
    if age % 2 ==0:
        print('Entered Age is even')
    else:
        print('Entered Age is odd')
try:
    num = int(input('Enter your age: '))
    enterAge(num)
except ValueError:
    print('Only positive integers are allowed')
```

Run the program and enter positive integer.

Expected Output

```
Enter your age: 12
Entered Age is even
```

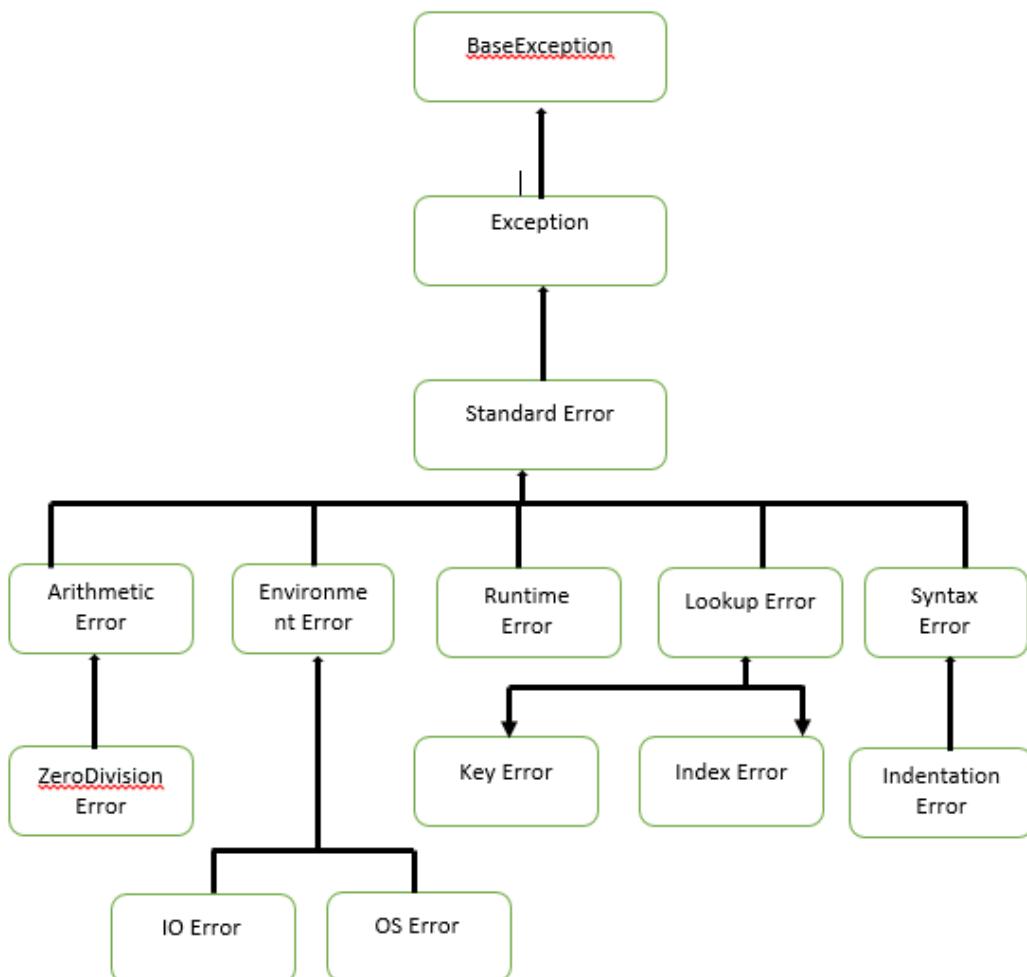
But when we try to enter a negative number we get,

Expected Output

```
Enter your age: -2
Only positive integers are allowed
```

Creating Custom exception class

You can create a custom exception class by Extending BaseException class or subclass of BaseException.



From above diagram we can see most of the exception classes in Python extends from the BaseException class. You can derive your own exception class from BaseException class or from its subclass.

Create a new file called NegativeNumberException.py and write the following code.

```

class NegativeNumberException(RuntimeError):
    def __init__(self, age):
        super().__init__()
        self.age = age
  
```

Above code creates a new exception class named NegativeNumberException, which consists of only constructor which call parent class constructor using super().__init__() and sets the age.

Now to create your own custom exception class, will write some code and import the new exception class.

```
from NegativeNumberException import NegativeNumberException

def enterage(age):
    if age < 0:
        raise NegativeNumberException('Only positive integers are allowed')

    if age % 2 == 0:
        print('Age is Even')

    else:
        print('Age is Odd')

try:
    num = int(input('Enter your age: '))
    enterage(num)
except NegativeNumberException:
    print('Only positive integers are allowed')
except:
    print('Something is wrong')
```

Output:

```
Enter your age: -2
Only positive integers are allowed
```

Another way to create a custom Exception class.

```
class customException(Exception):
    def __init__(self, value):
        self.parameter = value

    def __str__(self):
        return repr(self.parameter)

try:
    raise customException('My Useful Error Message!')
except customException as instance:
    print('Caught: ' + instance.parameter)
```

Output

```
Caught: My Useful Error Message!
```

Exception hierarchy

The class hierarchy for built-in exceptions is:

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
+-- StopIteration
+-- StopAsyncIteration
+-- ArithmeticError
|   +-- FloatingPointError
|   +-- OverflowError
|   +-- ZeroDivisionError
+-- AssertionError
+-- AttributeError
+-- BufferError
+-- EOFError
+-- ImportError
+-- LookupError
|   +-- IndexError
|   +-- KeyError
+-- MemoryError
+-- NameError
|   +-- UnboundLocalError
+-- OSError
|   +-- BlockingIOError
|   +-- ChildProcessError
|   +-- ConnectionError
|       +-- BrokenPipeError
|       +-- ConnectionAbortedError
|       +-- ConnectionRefusedError
|       +-- ConnectionResetError
+-- FileNotFoundError
+-- InterruptedError
+-- IsADirectoryError
+-- NotADirectoryError
+-- PermissionError
+-- ProcessLookupError
|   +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
|   +-- NotImplementedError
|   +-- RecursionError
+-- SyntaxError
|   +-- IndentationError
```

```
| +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
| +-- UnicodeError
| +-- UnicodeDecodeError
| +-- UnicodeEncodeError
| +-- UnicodeTranslateError
+-- Warning
+-- DeprecationWarning
+-- PendingDeprecationWarning
+-- RuntimeWarning
+-- SyntaxWarning
+-- UserWarning
+-- FutureWarning
+-- ImportWarning
+-- UnicodeWarning
+-- BytesWarning
+-- ResourceWarning
```

11. OOP in Python – Object Serialization

In the context of data storage, serialization is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer) or transmitted and reconstructed later.

In serialization, an object is transformed into a format that can be stored, so as to be able to deserialize it later and recreate the original object from the serialized format.

Pickle

Pickling is the process whereby a Python object hierarchy is converted into a byte stream (usually not human readable) to be written to a file, this is also known as **Serialization**. Unpickling is the reverse operation, whereby a byte stream is converted back into a working Python object hierarchy.

Pickle is operationally simplest way to store the object. The Python Pickle module is an object-oriented way to store objects directly in a special storage format.

What can it do?

- Pickle can store and reproduce dictionaries and lists very easily.
- Stores object attributes and restores them back to the same State.

What pickle can't do?

- It does not save an objects code. Only its attributes values.
- It cannot store file handles or connection sockets.

In short we can say, pickling is a way to store and retrieve data variables into and out from files where variables can be lists, classes, etc.

To Pickle something you must:

- import pickle
- Write a variable to file, something like

```
pickle.dump(mystring, outfile, protocol),
```

where 3rd argument protocol is optional

To unpickling something you must:

Import pickle

Write a variable to a file, something like

```
myString = pickle.load(inputfile)
```

Methods

The pickle interface provides four different methods.

- `dump()` : The `dump()` method serializes to an open file (file-like object).
- `dumps()`: Serializes to a string
- `load()`: Deserializes from an open-like object.
- `loads()` : Deserializes from a string.

Based on above procedure, below is an example of “pickling”.

```
import pickle
class Animal:
    def __init__(self, number_of_legs, color):
        self.number_of_legs = number_of_legs
        self.color = color
class Cat(Animal):
    def __init__(self, color):
        Animal.__init__(self, 4, color)

pussy = Cat("White")
print (str.format("My Cat pussy is {0} and has {1} legs", pussy.color, pussy.number_of_legs))
pickled_pussy = pickle.dumps(pussy)
print ("Would you like to see her pickled? Here she is!")
print (pickled_pussy)
```

Output

```
My Cat pussy is White and has 4 legs
Would you like to see her pickled? Here she is!
b'\x80\x03c__main__\nCat\nq\x00)\x81q\x01}q\x02(X\x0e\x00\x00\x00number_of_legs
q\x03K\x04X\x05\x00\x00\x00colorq\x04X\x05\x00\x00\x00Whiteq\x05ub.'
```

So, in the example above, we have created an instance of a Cat class and then we've pickled it, transforming our "Cat" instance into a simple array of bytes.

This way we can easily store the bytes array on a binary file or in a database field and restore it back to its original form from our storage support in a later time.

Also if you want to create a file with a pickled object, you can use the `dump()` method (instead of the `dumps()` one) passing also an opened binary file and the pickling result will be stored in the file automatically.

```
[....]
binary_file = open(my_pickled_Pussy.bin', mode='wb')
my_pickled_Pussy = pickle.dump(Pussy, binary_file)
binary_file.close()
```

Unpickling

The process that takes a binary array and converts it to an object hierarchy is called unpickling.

The unpickling process is done by using the `load()` function of the `pickle` module and returns a complete object hierarchy from a simple bytes array.

Let's use the `load` function in our previous example.

```
import pickle
class Animal:
    def __init__(self, number_of_legs, color):
        self.number_of_legs = number_of_legs
        self.color = color
class Cat(Animal):
    def __init__(self, color):
        Animal.__init__(self, 4, color)

# Step 1: Let's create the Cat Pussy
Pussy = Cat("white")
# Step 2: Let's pickle Pussy
my_pickled_Pussy = pickle.dumps(Pussy)
# Step 3: Now, let's unpickle our Cat Pussy creating another instance, another Cat... MeOw!
MeOw = pickle.loads(my_pickled_Pussy)
# MeOw and Pussy are two different objects, in fact if we specify another color for MeOw
# there are no consequences for Pussy
MeOw.color = "black"
print(str.format("MeOw is {0} ", MeOw.color))
print(str.format("Pussy is {0} ", Pussy.color))
```

Output

```
MeOw is black
Pussy is white
```

JSON

JSON(JavaScript Object Notation) has been part of the Python standard library is a lightweight data-interchange format. It is easy for humans to read and write. It is easy to parse and generate.

Because of its simplicity, JSON is a way by which we store and exchange data, which is accomplished through its JSON syntax, and is used in many web applications. As it is in human readable format, and this may be one of the reasons for using it in data transmission, in addition to its effectiveness when working with APIs.

An example of JSON-formatted data is as follow:

```
{"EmployID": 40203, "Name": "Zack", "Age":54, "isEmployed": True}
```

Python makes it simple to work with Json files. The module used for this purpose is the `JSON` module. This module should be included (built-in) within your Python installation.

So let's see how can we convert Python dictionary to JSON and write it to a text file.

JSON to Python

Reading JSON means converting JSON into a Python value (object). The json library parses JSON into a dictionary or list in Python. In order to do that, we use the loads() function (load from a string), as follow:

```
import json

jsonData = '{"EmployId":402040, "EmployeeName":"Zack", "Department":"Financial Services"}'

jsonToPython = json.loads(jsonData)

print(jsonToPython)
```

Output

```
{'EmployId': 402040, 'EmployeeName': 'Zack', 'Department': 'Financial Services'}
```

Below is one sample json file,

```
data1.json
{
  "menu": {
    "id": "file",
    "value": "File",
    "popup": {
      "menuitem": [
        {"value": "New", "onclick": "CreateNewDoc()"},
        {"value": "Open", "onclick": "OpenDoc()"},
        {"value": "Close", "onclick": "CloseDoc()"}
      ]
    }
  }
}
```

Above content (Data1.json) looks like a conventional dictionary. We can use pickle to store this file but the output of it is not human readable form.

JSON(Java Script Object Notification) is a very simple format and that's one of the reason for its popularity. Now let's look into json output through below program.

```

import json

with open('data1.json') as fh:
    conf = json.load(fh)

print(conf)
print("++++++")
print(type(conf))
print("++++++")
conf['neykey'] = 9.04050

with open('data1.json', 'w') as fh:
    json.dump(conf, fh)

print(conf)

```

Output

```

{'menu': {'id': 'file', 'value': 'File', 'popup': {'menuitem': [{'value': 'New', 'onclick': 'CreateNewDoc()'}, {'value': 'Open', 'onclick': 'OpenDoc()'}, {'value': 'Close', 'onclick': 'CloseDoc()'}]]}}
+++++
<class 'dict'>
+++++
{'menu': {'id': 'file', 'value': 'File', 'popup': {'menuitem': [{'value': 'New', 'onclick': 'CreateNewDoc()'}, {'value': 'Open', 'onclick': 'OpenDoc()'}, {'value': 'Close', 'onclick': 'CloseDoc()'}]}}, 'neykey': 9.0405}
>>>

```

Above we open the json file (data1.json) for reading, obtain the file handler and pass on to json.load and getting back the object. When we try to print the output of the object, its same as the json file. Although the type of the object is dictionary, it comes out as a Python object. Writing to the json is simple as we saw this pickle. Above we load the json file, add another key value pair and writing it back to the same json file. Now if we see our data1.json, it looks different i.e. not in the same format as we see previously.

To make our output looks same (human readable format), add the couple of arguments into our last line of the program,

```
json.dump(conf, fh, indent=4, separators = (',', ': '))
```

Similarly like pickle, we can print the string with dumps and load with loads. Below is an example of that,

```

>>> import json
>>>
>>> x = json.dumps({'x1': [ 2, 4, 6], 'y1':[3, 6, 9], 'z1': [4, 8, 12]})
>>>
>>> print(x)
{"x1": [2, 4, 6], "y1": [3, 6, 9], "z1": [4, 8, 12]}
>>>
>>> mystruct = json.loads(x)
>>> for key in mystruct:
    print(key)
    for x in mystruct[key]:
        print (" {0}".format(x))

x1
2
4
6
y1
3
6
9
z1
4
8
12

```

YAML

YAML may be the most human friendly data serialization standard for all programming languages.

Python yaml module is called pyyaml

YAML is an alternative to JSON:

- Human readable code: YAML is the most human readable format so much so that even its front-page content is displayed in YAML to make this point.
- Compact code: In YAML we use whitespace indentation to denote structure not brackets.
- Syntax for relational data: For internal references we use anchors (&) and aliases (*)
- One of the area where it is used widely is for viewing/editing of data structures: for example configuration files, dumping during debugging and document headers.

Installing YAML

As yaml is not a built-in module, we need to install it manually. Best way to install yaml on windows machine is through pip. Run below command on your windows terminal to install yaml,

```

pip install pyyaml (Windows machine)
sudo pip install pyyaml (*nix and Mac)

```

On running above command, screen will display something like below based on what's the current latest version.

```
Collecting pyyaml
```

```
Using cached pyaml-17.12.1-py2.py3-none-any.whl
Collecting PyYAML (from pyaml)
Using cached PyYAML-3.12.tar.gz
Installing collected packages: PyYAML, pyaml
Running setup.py install for PyYAML ... done
Successfully installed PyYAML-3.12 pyaml-17.12.1
```

To test it, go to the Python shell and import the yaml module,
import yaml, if no error is found, then we can say installation is successful.

After installing pyaml, let's look at below code,

script_yaml1.py

```
import yaml

mydict = {'a':2, 'b':4, 'c': 6}
mylist = [1, 2, 3, 4, 5]
mytuple = ('x', 'y', 'z')

print(yaml.dump(mydict, default_flow_style=False))
#print(loader_yaml)

print(yaml.dump(mylist, default_flow_style = False))

print(yaml.dump(mytuple, default_flow_style = False))
```

Above we created three different data structure, dictionary, list and tuple. On each of the structure, we do yaml.dump. Important point is how the output is displayed on the screen.

Output

```
a: 2
b: 4
c: 6

- 1
- 2
- 3
- 4
- 5

!!python/tuple
- x
- y
- z
```



Dictionary output looks clean .ie. key: value.

White space to separate different objects.

List is notated with dash (-)

Tuple is indicated first with !!Python/tuple and then in the same format as lists.

Loading a yaml file

So let's say I have one yaml file, which contains,

```
---
# An employee record
name: Raagvendra Joshi
job: Developer
skill: Oracle
employed: True
foods:
  - Apple
  - Orange
  - Strawberry
  - Mango
languages:
  Oracle: Elite
  power_builder: Elite
  Full Stack Developer: Lame
education:
  4 GCSEs
  3 A-Levels
  MCA in something called com
```

Now let's write a code to load this yaml file through yaml.load function. Below is code for the same.

```

import yaml

with open('eRecord.yaml') as fh:
    struct = yaml.load(fh)

print(struct)

print("====")
print('To make the output in more readable format, add the json module!')

import json
print(json.dumps(struct, indent = 4, separators = (' ', ',' , ': ')))

```

As the output doesn't looks that much readable, I prettify it by using json in the end. Compare the output we got and the actual yaml file we have.

Output

```

{'name': 'Raagvendra Joshi', 'job': 'Developer', 'skill': 'Oracle', 'employed': True, 'foods': ['Apple', 'Orange', 'Strawberry', 'Mango'], 'languages': {'Oracle': 'Elite', 'power_builder': 'Elite', 'Full Stack Developer': 'Lame'}, 'education': '4 GCSEs 3 A-Levels MCA in something called computer'}
=====
To make the output in more readable format, add the json module!
{
    "name": "Raagvendra Joshi",
    "job": "Developer",
    "skill": "Oracle",
    "employed": true,
    "foods": [
        "Apple",
        "Orange",
        "Strawberry",
        "Mango"
    ],
    "languages": {
        "Oracle": "Elite",
        "power_builder": "Elite",
        "Full Stack Developer": "Lame"
    },
    "education": "4 GCSEs 3 A-Levels MCA in something called computer"
}

```

One of the most important aspect of software development is debugging. In this section we'll see different ways of Python debugging either with built-in debugger or third party debuggers.

PDB – The Python Debugger

The module PDB supports setting breakpoints. A breakpoint is an intentional pause of the program, where you can get more information about the programs state.

To set a breakpoint, insert the line

```
pdb.set_trace()
```

Example

```
pdb_example1.py
import pdb
x=9
y=7
pdb.set_trace()
total = x + y
pdb.set_trace()
```

We have inserted a few breakpoints in this program. The program will pause at each breakpoint (`pdb.set_trace()`). To view a variables contents simply type the variable name.

```
c:\Python\Python361>Python pdb_example1.py
> c:\Python\Python361\pdb_example1.py(8)<module>()
-> total = x + y
(Pdb) x
9
(Pdb) y
7
(Pdb) total
*** NameError: name 'total' is not defined
(Pdb)
```

Press `c` or continue to go on with the programs execution until the next breakpoint.

```
(Pdb) c
--Return--
> c:\Python\Python361\pdb_example1.py(8)<module>()->None
-> total = x + y
(Pdb) total
```

16

Eventually, you will need to debug much bigger programs – programs that use subroutines. And sometimes, the problem that you're trying to find will lie inside a subroutine. Consider the following program.

```
import pdb

def squar(x, y):
    out_squared = x^2 + y^2
    return out_squared

if __name__ == "__main__":
    #pdb.set_trace()
    print (squar(4, 5))
```

Now on running the above program,

```
c:\Python\Python361>Python pdb_example2.py
> c:\Python\Python361\pdb_example2.py(10)<module>()
-> print (squar(4, 5))
(Pdb)
```

We can use **?** to get help, but the arrow indicates the line that's about to be executed. At this point it's helpful to hit **s** to **s** to step into that line.

```
(Pdb) s
--Call--
> c:\Python\Python361\pdb_example2.py(3)square()
-> def square(x, y):
```

This is a call to a function. If you want an overview of where you are in your code, try **l**:

```
(Pdb) l
1     import pdb
2
3     def square(x, y):
4 ->         out_squared = x^2 + y^2
5
6         return out_squared
7
8     if __name__ == "__main__":
9         pdb.set_trace()
10        print (square(4, 5))
[EOF]
```

```
(Pdb)
```

You can hit n to advance to the next line. At this point you are inside the out_squared method and you have access to the variable declared inside the function .i.e. x and y.

```
(Pdb) x
4
(Pdb) y
5
(Pdb) x^2
6
(Pdb) y^2
7
(Pdb) x**2
16
(Pdb) y**2
25
(Pdb)
```

So we can see the ^ operator is not what we wanted instead we need to use ** operator to do squares.

This way we can debug our program inside the functions/methods.

Logging

The logging module has been a part of Python's Standard Library since Python version 2.3. As it's a built-in module all Python module can participate in logging, so that our application log can include your own message integrated with messages from third party module. It provides a lot of flexibility and functionality.

Benefits of Logging

- Diagnostic logging: It records events related to the application's operation.
- Audit logging: It records events for business analysis.

Messages are written and logged at levels of "severity":

- DEBUG (debug()): diagnostic messages for development.
- INFO (info()): standard "progress" messages.
- WARNING (warning()): detected a non-serious issue.
- ERROR (error()): encountered an error, possibly serious
- CRITICAL (critical()): usually a fatal error (program stops)

Let's looks into below simple program,

logging1.py



```

import logging

logging.basicConfig(level=logging.INFO)

logging.debug('this message will be ignored')      # This will not print
logging.info('This should be logged')               # It'll print
logging.warning('And this, too')                    # It'll print

```

Above we are logging messages on severity level. First we import the module, call basicConfig and set the logging level. Level we set above is INFO. Then we have three different statement: debug statement, info statement and a warning statement.

Output of logging1.py

```

INFO:root:This should be logged
WARNING:root:And this, too

```

As the info statement is below debug statement, we are not able to see the debug message. To get the debug statement too in the output terminal, all we need to change is the basicConfig level.

```
logging.basicConfig(level=logging.DEBUG)
```

And in the output we can see,

```

DEBUG:root:this message will be ignored
INFO:root:This should be logged
WARNING:root:And this, too

```

Also the default behavior means if we don't set any logging level is warning. Just comment out the second line from the above program and run the code.

```
#logging.basicConfig(level=logging.DEBUG)
```

Output

```

WARNING:root:And this, too

```

Python built in logging level are actually integers.

```

>>> import logging
>>>
>>> logging.DEBUG
10
>>> logging.CRITICAL

```

```

50
>>> logging.WARNING
30
>>> logging.INFO
20
>>> logging.ERROR
40
>>>

```

We can also save the log messages into the file.

```
logging.basicConfig(level=logging.DEBUG, filename = 'logging.log')
```

Now all log messages will go the file (logging.log) in your current working directory instead of the screen. This is a much better approach as it lets us to do post analysis of the messages we got.

We can also set the date stamp with our log message.

```
logging.basicConfig(level=logging.DEBUG, format = '%(asctime)s
%(levelname)s:%(message)s')
```

Output will get something like,

```

2018-03-08 19:30:00,066 DEBUG:this message will be ignored
2018-03-08 19:30:00,176 INFO:This should be logged
2018-03-08 19:30:00,201 WARNING:And this, too

```

Benchmarking

Benchmarking or profiling is basically to test how fast is your code executes and where the bottlenecks are? The main reason to do this is for optimization.

timeit

Python comes with a in-built module called timeit. You can use it to time small code snippets. The timeit module uses platform-specific time functions so that you will get the most accurate timings possible.

So, it allows us to compare two shipment of code taken by each and then optimize the scripts to given better performance.

The timeit module has a command line interface, but it can also be imported.

There are two ways to call a script. Let's use the script first, for that run the below code and see the output.

```
import timeit
```

```

print ( 'by index: ', timeit.timeit(stmt = "mydict['c']", setup="mydict = 
{'a':5, 'b':10, 'c':15}", number = 1000000))

print ( 'by get: ', timeit.timeit(stmt = 'mydict.get("c")', setup = 'mydict = 
{"a":5, "b":10, "c":15}', number = 1000000))

```

Output

```

by index:  0.1809192126703489
by get:   0.6088525265034692

```

Above we use two different method i.e. by subscript and get to access the dictionary key value. We execute statement 1 million times as it executes too fast for a very small data. Now we can see the index access much faster as compared to the get. We can run the code multiply times and there will be slight variation in the time execution to get the better understanding.

Another way is to run the above test in the command line. Let's do it,

```

c:\Python\Python361>Python -m timeit -n 1000000 -s "mydict = {'a': 5, 'b':10,
'c':15}" "mydict['c']"
1000000 loops, best of 3: 0.187 usec per loop

c:\Python\Python361>Python -m timeit -n 1000000 -s "mydict = {'a': 5, 'b':10,
'c':15}" "mydict.get('c')"
1000000 loops, best of 3: 0.659 usec per loop

```

Above output may vary based on your system hardware and what all applications are running currently in your system.

Below we can use the timeit module, if we want to call to a function. As we can add multiple statement inside the function to test.

```

import timeit

def testme(this_dict, key):
    return this_dict[key]

print (timeit.timeit("testme(mydict, key)", setup ="from __main__ import
testme; mydict = {'a':9, 'b':18, 'c':27}; key='c'", number=1000000))

```

Output



0.7713474590139164

12.12. OOP in Python – Python Libraries

Requests: Python Requests Module

Requests is a Python module which is an elegant and simple HTTP library for Python. With this you can send all kinds of HTTP requests. With this library we can add headers, form data, multipart files and parameters and access the response data.

As Requests is not a built-in module, so we need to install it first.

You can install it by running the following command in the terminal:

```
pip install requests
```

Once you have installed the module, you can verify if the installation is successful by typing below command in the Python shell.

```
import requests
```

If the installation has been successful, you won't see any error message.

Making a GET Request

As a means of example we'll be using the "pokeapi"

```
import requests
import json

def main():
    req = requests.get('http://pokeapi.co/api/v2/pokemon/1/')
    print('HTTP Status Code: ' + str(req.status_code))
    print(req.headers)
    json_response = json.loads(req.content)
    print('Pokemon Name: ' + json_response['name'])

if __name__ == '__main__':
    main()
```

Output:

```
HTTP Status Code: 200
{'Date': 'Thu, 01 Mar 2018 05:49:39 GMT', 'Content-Type': 'application/json', 'Transfer-Encoding': 'chunked', 'Connection': 'keep-alive', 'Set-Cookie': '__cfduid=dfdc3b64c65430b43d1c24d1951e66d2b1519883377; expires=Fri, 01-Mar-19 05:49:37 GMT; path=/; domain=.pokeapi.co; HttpOnly; Secure', 'Vary': 'Accept-Encoding, Cookie', 'X-Frame-Options': 'SAMEORIGIN', 'Allow': 'GET, HEAD, OPTIONS', 'X-XSS-Protection': '1; mode=block', 'Content-Encoding': 'gzip', 'Expect-CT': 'max-age=604800, report-uri="https://report-uri.cloudflare.com/cdn-cgi/beacon/expect-ct"', 'Server': 'cloudflare', 'CF-RAY': '3f495767490b0805-SIN'}
Pokemon Name: bulbasaur
```

Making POST Requests

The requests library methods for all of the HTTP verbs currently in use. If you wanted to make a simple POST request to an API endpoint then you can do that like so:

```
req = requests.post('http://api/user', data=None, json=None)
```

This would work in exactly the same fashion as our previous GET request, however it features two additional keyword parameters:

- data which can be populated with say a dictionary, a file or bytes that will be passed in the HTTP body of our POST request.
- json which can be populated with a json object that will be passed in the body of our HTTP request also.

Pandas: Python Library Pandas

Pandas is an open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures. Pandas is one of the most widely used Python libraries in data science. It is mainly used for data munging, and with good reason: Powerful and flexible group of functionality.

Built on Numpy package and the key data structure is called the DataFrame. These dataframes allows us to store and manipulate tabular data in rows of observations and columns of variables.

There are several ways to create a DataFrame. One way is to use a dictionary. For example:

```
import pandas as pd

dict = {"Brics_country": ["Brazil", "Russia", "India", "China", "South Africa"],
        "Brics_capital": ["Brasilia", "Moscow", "New Dehli", "Beijing", "Pretoria"],
        "area": [8.516, 17.10, 3.286, 9.597, 1.221],
        "population": [200.4, 143.5, 1252, 1357, 52.98] }

brics = pd.DataFrame(dict)
print(brics)
```

Output:

	Bric_country	Brics_capital	area	population
0	Brazil	Brasília	8.516	200.40
1	Russia	Moscow	17.100	143.50
2	India	New Dehli	3.286	1252.00
3	China	Beijing	9.597	1357.00
4	South Africa	Pretoria	1.221	52.98

From the output we can see new brics DataFrame, Pandas has assigned a key for each country as the numerical values 0 through 4.

If instead of giving indexing values from 0 to 4, we would like to have different index values, say the two letter country code, you can do that easily as well:

Adding below one lines in the above code, gives

```
brics.index = ['BR', 'RU', 'IN', 'CH', 'SA']
```

Output

	Bric_country	Brics_capital	area	population
BR	Brazil	Brasília	8.516	200.40
RU	Russia	Moscow	17.100	143.50
IN	India	New Dehli	3.286	1252.00
CH	China	Beijing	9.597	1357.00
SA	South Africa	Pretoria	1.221	52.98

Indexing DataFrames

```
import pandas as pd

stocks_list = pd.read_csv('stocks_list.csv', index_col = 0)

# print the stock_list.csv file
print(stocks_list)

print(" ")
# print the ISIN column as pandas series
print(stocks_list['ISIN'])

# print the ISIN as pandas DataFrame
print(stocks_list[['ISIN', 'TOTALTRADES']])
```

Output

```
>>>
===== RESTART: C:/Python/Python361/pandas_script2.py =====
      SERIES   OPEN     HIGH     LOW    CLOSE    LAST  PREVCLOSE \
SYMBOL
NTPC      EQ  164.00  169.20  163.75  168.75  168.45    167.95
RELIANCE   EQ 1575.00 1597.45 1574.00 1594.50 1595.40   1564.10
HDFC      EQ 1762.00 1780.95 1746.00 1776.90 1779.80   1759.05
INFY      EQ  928.80  928.95  912.90  914.95  915.00   926.65
HINDALCO   EQ  237.80  239.45  235.10  238.00  238.25   236.35
IOC        EQ  454.70  462.95  448.05  454.70  455.50   452.85
RELCAPITAL EQ  781.30  812.00  775.00  805.30  811.00   779.90
VEDL       EQ  305.35  310.00  300.90  308.90  308.80   305.55
YESBANK    EQ 1750.00 1774.40 1745.05 1753.05 1754.00   1752.10

      TOTTRDQTY  TOTTRDVAL  TIMESTAMP  TOTALTRADES          ISIN
SYMBOL
NTPC      83350065 13902129064 31-Aug-17      157814  INE733E01010
RELIANCE  5738495  9108681150 31-Aug-17      153607  INE002A01018
HDFC      4646322  8220571377 31-Aug-17      179595  INE001A01036
INFY      7511226  6886139481 31-Aug-17      174045  INE009A01021
HINDALCO  23200769 5516181928 31-Aug-17      79864   INE038A01020
IOC        11525950  5260716070 31-Aug-17      124043  INE242A01010
RELCAPITAL 6647282  5252965990 31-Aug-17      78603   INE013A01015
VEDL       14987864  4595411062 31-Aug-17      89794   INE205A01025
YESBANK    2566761  4515288596 31-Aug-17      80122   INE528G01019

          ISIN  TOTALTRADES
SYMBOL
NTPC      INE733E01010      157814
RELIANCE  INE002A01018      153607
HDFC      INE001A01036      179595
INFY      INE009A01021      174045
HINDALCO  INE038A01020      79864
IOC        INE242A01010      124043
RELCAPITAL INE013A01015      78603
VEDL       INE205A01025      89794
YESBANK    INE528G01019      80122
>>> |
```

Pygame

Pygame is the open source and cross-platform library that is for making multimedia applications including games. It includes computer graphics and sound libraries designed to be used with the Python programming language. You can develop many cool games with Pygame.'

Overview

Pygame is composed of various modules, each dealing with a specific set of tasks. For example, the display module deals with the display window and screen, the draw module provides functions to draw shapes and the key module works with the keyboard. These are just some of the modules of the library.

The home of the Pygame library is at <http://pygame.org>.

To make a Pygame application, you follow these steps:

Import the Pygame library

```
import pygame
```

Initialize the Pygame library

```
pygame.init()
```

Create a window.

```
screen = Pygame.display.set_mode((560,480))  
Pygame.display.set_caption('First Pygame Game')
```

Initialize game objects

In this step we load images, load sounds, do object positioning, set up some state variables, etc.

Start the game loop.

It is just a loop where we continuously handle events, checks for input, move objects, and draw them. Each iteration of the loop is called a frame.

Let's put all the above logic into one below program,

Pygame_script.py

```
#Import the pygame and the sys module for exiting the window we create
import sys, pygame

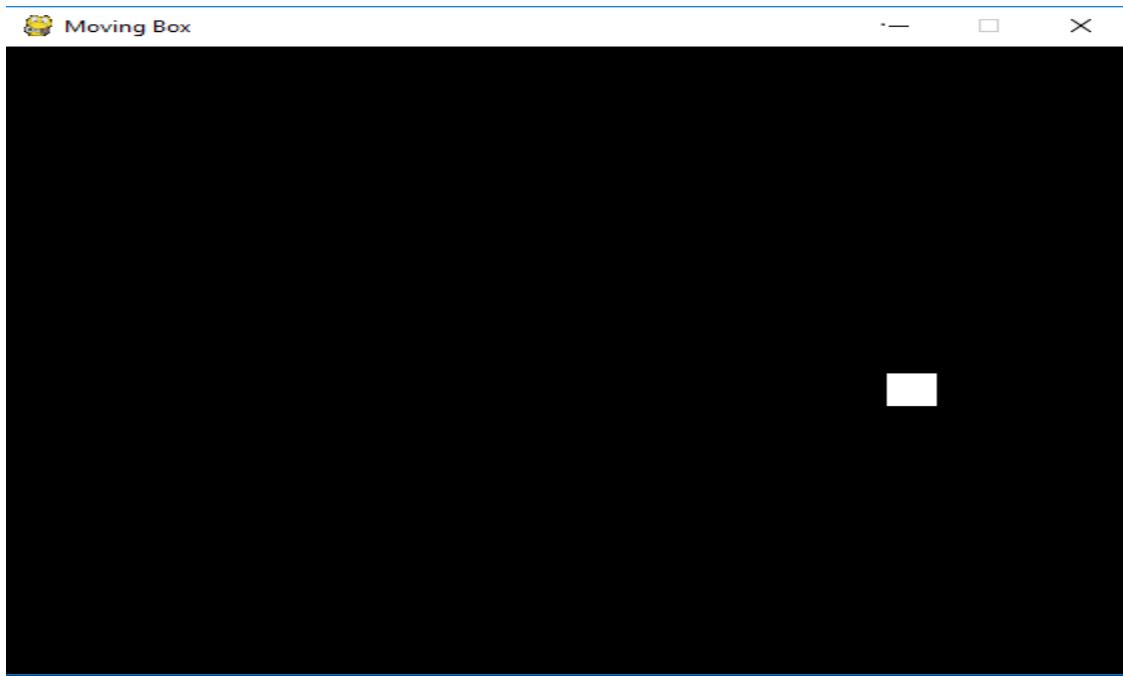
BLACK = 0, 0, 0
WHITE = 255, 255, 255
# Initialise the pygame module
pygame.init()
#Create a new window, width=560, height=480
screen = pygame.display.set_mode((560,480))
#Give the window a caption
pygame.display.set_caption("Moving Box")
clock = pygame.time.Clock()
box_x = 300
box_dir = 3
#Loop (repeat) forever
while 1:
    clock.tick(55)
#Get all the users events
    for event in pygame.event.get():
        #if the user wants to quit
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    screen.fill(BLACK)

    box_x += box_dir
    if box_x >= 620:
        box_x = 620
        box_dir = -3
    elif box_x <= 0:
        box_x = 0
        box_dir = 3

    pygame.draw.rect(screen, WHITE, (box_x, 250, 25, 25))
    pygame.display.flip()
```

Output



Beautiful Soup: Web Scraping with Beautiful Soup

The general idea behind web scraping is to get the data that exists on a website, and convert it into some format that is usable for analysis.

It's a Python library for pulling data out of HTML or XML files. With your favourite parser it provides idiomatic ways of navigating, searching and modifying the parse tree.

As BeautifulSoup is not a built-in library, we need to install it before we try to use it. To install BeautifulSoup, run the below command

```
$ apt-get install Python-bs4 # For Linux and Python2  
$ apt-get install Python3-bs4      # for Linux based system and Python3.  
  
$ easy_install beautifulsoup4      # For windows machine,  
Or  
$ pip instal beatifulsoup4       # For window machine
```

Once the installation is done, we are ready to run few examples and explore BeautifulSoup in details,

```
#Import BeautifulSoup
from bs4 import BeautifulSoup
from urllib.request import urlopen

r = urlopen('https://en.wikipedia.org/wiki/List_of_countries_by_foreign-exchange_reserves_(excluding_gold)').read()
soup = BeautifulSoup(r, 'html.parser')

print(type(soup))

print(soup.prettify()[0:1000])
```

Output

```
<class 'bs4.BeautifulSoup'>
<!DOCTYPE html>
<html class="client-nojs" dir="ltr" lang="en">
<head>
<meta charset="utf-8"/>
<title>
List of countries by foreign-exchange reserves (excluding gold) - Wikipedia
</title>
<script>
document.documentElement.className = document.documentElement.className.replace( /(^|\s)client-nojs(\s|$)/, "$1client-js$2" );
</script>
<script>
(window.RLQ=window.RLQ||[]).push(function() {mw.config.set({"wgCanonicalNamespace": "", "wgCanonicalSpecialPageName": false, "wgNamespaceNumber": 0, "wgPageName": "List_of_countries_by_foreign-exchange_reserves_(excluding_gold)", "wgTitle": "List of countries by foreign-exchange reserves (excluding gold)", "wgCurRevisionId": 828715850, "wgRevisionId": 828715850, "wgArticleId": 39057473, "wgIsArticle": true, "wgIsRedirect": false, "wgAction": "view", "wgUserName": null, "wgUserGroups": ["*"], "wgCategories": ["Use dmy dates from September 2016", "Lists of countries by economic indicator", "Foreign exchange reserves"], "wgBreakFrames": false, "wgPageContentLangua
```

Below are some simple ways to navigate that data structure:

```
>>> # Simple tricks to navigate different information we can get from data structure.
>>>
>>> soup.title
<title>List of countries by foreign-exchange reserves (excluding gold) - Wikipedia</title>

>>> soup.title.name
'title'
>>>
>>> soup.title.string
'List of countries by foreign-exchange reserves (excluding gold) - Wikipedia'
>>>
>>> soup.p
<p>This is a list of the top 33 <a href="/wiki/List_of_sovereign_states" title="List of sovereign states">sovereign states</a> of the <a href="/wiki/World" title="World">world</a> sorted by their <a href="/wiki/Foreign-exchange_reserves" title="Foreign-exchange reserves">foreign-exchange reserves</a> <b>excluding</b> <a href="/wiki/Gold_reserve" title="Gold reserve">gold reserves</a>, <b>but including</b> <a href="/wiki/Special_drawing_rights" title="Special drawing rights">special drawing rights</a> (SDRs) and <a href="/wiki/International_Monetary_Fund" title="International Monetary Fund">International Monetary Fund</a> (IMF) reserve positions.</p>
>>>
```

One common task is extracting all the URLs found within a page's `<a>` tags:

```
>>> # Extracting all the URLs found within a page's <a> tags
>>> for link in soup.find_all('a'):
    print(link.get('href'))

None
#mw-head
#p-search
/wiki/List_of_countries_by_foreign-exchange_reserves
/wiki/List_of_sovereign_states
/wiki/World
/wiki/Foreign-exchange_reserves
/wiki/Gold_reserve
/wiki/Special_drawing_rights
/wiki/International_Monetary_Fund
#cite_note-imf-1
/wiki/Sovereign_state
/wiki/Hong_Kong
/wiki/Macau
/wiki/Eurozone
/wiki/United_States_dollar
/wiki/China
#cite_note-2
#cite_note-imf-1
/wiki/Japan
#cite_note-3
#cite_note-imf-1
/wiki/Switzerland
#cite_note-4
/wiki/Saudi_Arabia
#cite_note-imf-1
/wiki/Taiwan
#cite_note-5
```

Another common task is extracting all the text from a page:

```
>>> print(soup.get_text())

List of countries by foreign-exchange reserves (excluding gold) - Wikipedia
document.documentElement.className = document.documentElement.className.replace( /(^\s|client-nojs|s|$)/, '$1client-js$2' );
(window.RLQ||[]).push(function() mw.config.set({ "wgCanonicalNamespace":0, "wgPageName": "List of countries by foreign-exchange reserves (excluding gold)", "wgTitle": "List of countries by foreign-exchange reserves (excluding gold)", "wgIsArticle":true, "wgAction": "view", "wgUserName":null, "wgUserGroups": ["*"], "wgCategories": ["Use dmy dates from September 2016"], "wgArticleId":39057473, "wgIsRedirect":false, "wgIsTransformTable": false, "wgDigitTransformTable": ["", ""], "wgBreakFrames": false, "wgPageContentLanguage": "en", "wgPageContentModel": "wikitext", "wgSeparatorTransformTable": ["", ""], "wgDefaultDateFormat": "dmy", "wgMonthNames": ["", "January", "February", "March", "April", "May", "June", "July", "August", "September", "November", "December"], "wgMonthNamesShort": ["", "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "De c"], "wgRelevantPageName": "List of countries by foreign-exchange reserves (excluding gold)", "wgRelevantPageId":39057473, "wgIsProbablyEditable":true, "wgRelevantPageIsProbablyEditable":true, "wgRestrictionEdit": [], "wgFlaggedRevsParams": {"tag": []}, "wgStableRevisionId": null, "wgWikiEditorEnabledModules": [], "wgBetaFeaturesFeatures": [], "wgMediaViewerOnClick": true, "wgPopupsShouldSendModuleToUser": true, "wgPopupsConflictsWithNavPopupsGadget": false, "wgVisualEditor": {"pageLanguageCode": "en", "pageLanguageDir": "ltr", "pageVariantFallbacks": "", "pageImages": true, "usePageDescriptions": true}, "wgPreferredVariant": "en", "wgMREnableFontChanger": true, "wgRelatedArticles": null, "wgRelatedArticlesUseCirrusSearch": true, "wgRelatedArticlesOnlyUseCirrusSearch": false, "wgULSCurrentAutonym": "English", "wgNoticeProject": "Wikipedia", "wgCentralNoticeCookiesToDelete": [], "wgCentralNoticeCategoriesUsingLegacy": ["Fundraising", "fundraising"], "wgCategoryTreePageCategoryOptions": {"model": 0, "hidePrefix": true, "showCount": true, "namespaces": false}, "wgWikibaseItemID": "Q17107491", "wgScoreNoteLanguages": ["arabi c", "العربية", "catalan", "català", "deutsch", "Deutsch", "english", "English", "espanol", "español", "italiano", "italiano", "nederlands", "Nederlands", "norsk", "norsk", "portugues", "portuguese", "suomi", "suomi", "svenska", "svenska", "West-Vlaams", "wgScoreDefaultNoteLanguage": "nederlands", "wgCentralAuthMobileDomain": false, "wgCodeMirrorEnabled": false, "wgVisualEditorToolbarScrollOffset": 0, "wgVisualEditorUnsupportedEditParams": ["undo", "undobar", "viewswitched"], "wgEditSubmitButtonLabelPublish": true}); mw.loader.state("ext.gadget.charinsert-styles": "ready", "ext.globalCSSJS.site.styles": "ready", "ext.globalCSSJS.user.styles": "ready", "user.styles": "ready", "user": "ready", "user.options": "ready", "user.tokens": "loading", "ext.cite.styles": "ready", "wgWikibaseClientInit": "ready", "ext.visualEditor.desktopOpArticleTarget.noscript": "ready", "ext.uis.interlanguage": "ready", "ext.wikimediabades": "ready", "mediawiki.legacy.shared": "ready", "mediawiki.legacy.commonPrint": "read y", "mediawiki.sectionAnchor": "ready", "mediawiki.skinning.interface": "ready", "skins.vector.styles": "ready", "ext.globalCSSJS.user": "ready", "ext.globalCSSJS.site": "ready"); mw.loader.implement("user.tokens@lqfd71", function($, jQuery, require, module){ /*nomini*/ mw.user.tokens.set({"editToken": "\\", "patrolToken": "\\", "watchToken": "\\", "certToken": "\\"}); }) mw.loader.load(["ext.cite.ally", "site", "mediawiki.page.startup", "mediawiki.user", "mediawiki.hipip", "mediawiki.page.ready", "jquery.tablesorter", "mediawiki.searchSug gest", "ext.gadget.teahouse", "ext.gadget.ReferenceToolips", "ext.gadget.watchlist-notice", "ext.gadget.DRM-wizard", "ext.gadget.charinsert", "ext.gadget.refToolbar", "ext.gadget.extra-toolbar-buttons", "ext.gadget.switcher", "ext.centralauth.centralautologin", "mmv.head", "mmv.bootstrap.autostart", "ext.popups", "ext.visualEditor.desktopArti
```



Python Basics

[DOWNLOAD PDF](#) [CODING BUGS](#) [NOTES GALLERY](#)

With Illustrations from the Financial Markets

by QuantInsti

Python Basics

With Illustrations from the Financial Markets

QuantInsti Quantitative Learning Pvt. Ltd.
- India -

[DOWNLOAD PDF](#)  [CODING BUGS](#)  [NOTES GALLERY](#)

Contents

1	Introduction	1
1.1	What is Python?	1
1.2	Where is Python used?	2
1.3	Why Python?	2
1.4	History of Python	6
1.5	Python 3 versus Python 2	7
1.6	Key Takeaways	10
2	Getting Started with Python	11
2.1	Python as a Calculator	11
2.1.1	Floating Point Expressions	14
2.2	Python Basics	17
2.2.1	Literal Constants	17
2.2.2	Numbers	18
2.2.3	Strings	18
2.2.4	Comments	19
2.2.5	<code>print()</code> function	20
2.2.6	<code>format()</code> function	22
2.2.7	Escape Sequence	23
2.2.8	Indentation	24
2.3	Key Takeaways	25
3	Variables and Data Types in Python	27
3.1	Variables	27
3.1.1	Variable Declaration and Assignment	27
3.1.2	Variable Naming Conventions	28
3.2	Data Types	31
3.2.1	Integer	31

3.2.2	Float	32
3.2.3	Boolean	34
3.2.4	String	35
3.2.5	Operations on String	38
3.2.6	type() function	41
3.3	Type Conversion	42
3.4	Key Takeaways	45
4	Modules, Packages and Libraries	47
4.1	Standard Modules	50
4.2	Packages	52
4.3	Installation of External Libraries	53
4.3.1	Installing pip	54
4.3.2	Installing Libraries	54
4.4	Importing modules	56
4.4.1	import statement	56
4.4.2	Selective imports	57
4.4.3	The Module Search Path	59
4.5	dir()function	61
4.6	Key Takeaways	63
5	Data Structures	65
5.1	Indexing and Slicing	65
5.2	Array	67
5.2.1	Visualizing an Array	67
5.2.2	Accessing Array Element	68
5.2.3	Manipulating Arrays	68
5.3	Tuples	70
5.3.1	Accessing tuple elements	71
5.3.2	Immutability	72
5.3.3	Concatenating Tuples	72
5.3.4	Unpacking Tuples	73
5.3.5	Tuple methods	73
5.4	Lists	74
5.4.1	Accessing List Items	75
5.4.2	Updating Lists	75
5.4.3	List Manipulation	77
5.4.4	Stacks and Queues	80

5.5	Dictionaries	82
5.5.1	Creating and accessing dictionaries	82
5.5.2	Altering dictionaries	85
5.5.3	Dictionary Methods	86
5.6	Sets	88
5.7	Key Takeaways	92
6	Keywords & Operators	95
6.1	Python Keywords	95
6.2	Operators	106
6.2.1	Arithmetic operators	106
6.2.2	Comparison operators	107
6.2.3	Logical operators	109
6.2.4	Bitwise operator	110
6.2.5	Assignment operators	114
6.2.6	Membership operators	118
6.2.7	Identity operators	118
6.2.8	Operator Precedence	119
6.3	Key Takeaways	121
7	Control Flow Statements	123
7.1	Conditional Statements	123
7.1.1	The <code>if</code> statement	123
7.1.2	The <code>elif</code> clause	125
7.1.3	The <code>else</code> clause	125
7.2	Loops	126
7.2.1	The <code>while</code> statement	126
7.2.2	The <code>for</code> statement	128
7.2.3	The <code>range()</code> function	128
7.2.4	Looping through lists	130
7.2.5	Looping through strings	131
7.2.6	Looping through dictionaries	131
7.2.7	Nested loops	133
7.3	Loop control statements	134
7.3.1	The <code>break</code> keyword	134
7.3.2	The <code>continue</code> keyword	135
7.3.3	The <code>pass</code> keyword	136
7.4	List comprehensions	137

7.5	Key Takeaways	140
8	Iterators & Generators	143
8.1	Iterators	143
8.1.1	Iterables	143
8.1.2	enumerate() function	145
8.1.3	The zip()function	146
8.1.4	Creating a custom iterator	147
8.2	Generators	149
8.3	Key Takeaways	151
9	Functions in Python	153
9.1	Recapping built-in functions	154
9.2	User defined functions	155
9.2.1	Functions with a single argument	156
9.2.2	Functions with multiple arguments and a return statement	157
9.2.3	Functions with default arguments	159
9.2.4	Functions with variable length arguments	160
9.2.5	DocStrings	162
9.2.6	Nested functions and non-local variable	164
9.3	Variable Namespace and Scope	166
9.3.1	Names in the Python world	167
9.3.2	Namespace	168
9.3.3	Scopes	169
9.4	Lambda functions	174
9.4.1	map() Function	175
9.4.2	filter() Function	176
9.4.3	zip() Function	177
9.5	Key Takeaways	179
10	NumPy Module	181
10.1	NumPy Arrays	182
10.1.1	N-dimensional arrays	185
10.2	Array creation using built-in functions	186
10.3	Random Sampling in NumPy	188
10.4	Array Attributes and Methods	192
10.5	Array Manipulation	198
10.6	Array Indexing and Iterating	203

10.6.1	Indexing and Subsetting	203
10.6.2	Boolean Indexing	205
10.6.3	Iterating Over Arrays	210
10.7	Key Takeaways	212
11	Pandas Module	215
11.1	Pandas Installation	215
11.1.1	Installing with pip	216
11.1.2	Installing with Conda environments	216
11.1.3	Testing Pandas installation	216
11.2	What problem does Pandas solve?	216
11.3	Pandas Series	217
11.3.1	Simple operations with Pandas Series	219
11.4	Pandas DataFrame	223
11.5	Importing data in Pandas	228
11.5.1	Importing data from CSV file	228
11.5.2	Customizing pandas import	228
11.5.3	Importing data from Excel files	229
11.6	Indexing and Subsetting	229
11.6.1	Selecting a single column	230
11.6.2	Selecting multiple columns	230
11.6.3	Selecting rows via []	231
11.6.4	Selecting via .loc[] (By label)	232
11.6.5	Selecting via .iloc[] (By position)	233
11.6.6	Boolean indexing	234
11.7	Manipulating a DataFrame	235
11.7.1	Transpose using .T	235
11.7.2	The .sort_index() method	236
11.7.3	The .sort_values() method	236
11.7.4	The .reindex() function	237
11.7.5	Adding a new column	238
11.7.6	Delete an existing column	239
11.7.7	The .at[] (By label)	241
11.7.8	The .iat[] (By position)	242
11.7.9	Conditional updating of values	243
11.7.10	The .dropna() method	244
11.7.11	The .fillna() method	246
11.7.12	The .apply() method	247

11.7.13 The <code>.shift()</code> function	248
11.8 Statistical Exploratory data analysis	250
11.8.1 The <code>info()</code> function	250
11.8.2 The <code>describe()</code> function	251
11.8.3 The <code>value_counts()</code> function	252
11.8.4 The <code>mean()</code> function	252
11.8.5 The <code>std()</code> function	253
11.9 Filtering Pandas DataFrame	253
11.10 Iterating Pandas DataFrame	255
11.11 Merge, Append and Concat Pandas DataFrame	256
11.12 TimeSeries in Pandas	259
11.12.1 Indexing Pandas TimeSeries	259
11.12.2 Resampling Pandas TimeSeries	262
11.12.3 Manipulating TimeSeries	263
11.13 Key Takeaways	265
12 Data Visualization with Matplotlib	267
12.1 Basic Concepts	268
12.1.1 Axes	269
12.1.2 Axes method v/s pyplot	272
12.1.3 Multiple Axes	273
12.2 Plotting	275
12.2.1 Line Plot	276
12.2.2 Scatter Plot	289
12.2.3 Histogram Plots	294
12.3 Customization	300
12.4 Key Takeaways	313

Preface

"If I have seen further, it is by standing upon the shoulders of giants."

- Sir Isaac Newton (1643 - 1727)

The universe we inhabit today is swimming in data. About 90% of the data created until 2016 was in just the previous couple of years!¹ There have also been simultaneous advances in affordable data storage (on both local workstations and cloud-based) and computing power. Case in point: The smartphones that we carry with us all day are cheap, pocket-sized supercomputers.

Closer home, our world of quantitative finance and algorithmic trading is awash with facts and figures, on financial markets, the macroeconomy, market sentiments, etc. We need a suitable tool to harness and profitably exploit the power of all this information. In a perfect world, this tool would be both easy to use and efficient. Reality, however, is rarely perfect. These qualities are often at odds with each other, and we have to choose our tool wisely.

As an algorithmic/quantitative trader, what are some of the critical issues we deal with every day?

- Downloading and managing massive datasets in multiple file formats from varied web sources
- Data cleaning/munging
- Backtesting and automating trading strategies
- Managing order executions with little manual intervention

¹<https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=WRL12345USEN>

We find that based on the domain-specific needs of our community, Python is a near-perfect fit. It's a high-level programming language that is relatively easy to learn. When used in conjunction with some of its libraries, it's incredibly powerful.

Why was this written?

This book grew out of the EPAT lectures that we have been conducting in Python for a number of years. In the EPAT Python lectures, we attempt to cover topics as widely as possible and therefore do not dig as deep as we'd like to, given the time limitations. These notes give us the freedom to marinate on some concepts a little longer and fill in on gaps that arise in the lecture format. We hope that our writing is compact and adequate for you, the reader, to obtain a greater understanding of the language and some of its essential libraries. We expect that for the interested reader, this book would be among the first of many books or blogs that you would read on Python. There is also a list of references towards the end of the book which we trust you may find useful.

Who should read this?

When this writing project began, our intended objective was to prepare a book geared towards the needs of our EPAT participants. In retrospect though, we think it should also be useful to

- anyone who wants a brief introduction to Python and the key components of its data science stack, and
- Python programmers who want a quick refresher on using Python for data analysis.

We do not expect any of our readers to have a formal background in computer science, although some familiarity with programming would be nice to have. The concepts and ideas here are covered with several examples to help connect theory to practice.

What's in this book?

The material presented here is a condensed introduction to Python and its data science related libraries such as NumPy, Pandas, and Matplotlib. The

illustrative examples we use are associated with the financial markets. We like to think we have done a satisfactory job of meeting the goals we set out to achieve.

How to read this?

We can suggest three ways to learn from this book depending on your conversance with Python and the time you have.

1. Read the book sequentially at your own pace from beginning to end. Ideally, you should read the chapters before or soon after you attend/watch the relevant EPAT lectures. It will certainly help in developing intuitions on new concepts that you pick up.
2. Blaze through the book linearly to get a big picture view of all the areas covered. You can then concentrate on the different parts based on what you find harder or what is more important for your work.
3. If you're already familiar with Python programming, you can pretty much hop on and hop off chapters as you like it.

We believe there is value to be had with any of these approaches, and each of us needs to assess what works best for us based on our learning style.

Where else can you find all this?

The short answer: Lots and lots of places (way too many to enlist here). We direct you to a non-exhaustive set of resources that we really like at in the references towards end of the book for further reading.

Python has been around for about three decades now. There are several excellent books, videos, online courses, and blogs covering it from various angles and directed at different kinds of users. However, the core set of ideas and concepts are well-understood and covered by most of them.

Copyright License

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License².

²<http://creativecommons.org/licenses/by-sa/4.0/>



That is why you see this image here. In essence, it means that you can use, share, or improve upon this work (even commercially) as long as you provide attribution to us. To put things in perspective, Wikipedia³ also uses the same license.

Acknowledgments

Jay Parmar, Mario Pisa Pena, and Vivek Krishnamoorthy are the authors of this book. Jay's done the lion's share of the writing and formatting. Mario's written some sections and reviewed most of the others. Vivek was the principal conspirator in hatching the book-writing plan to ease a student's learning journey as far as possible. He was also involved in the writing, the editing, the review, and in overseeing this venture.

Most of the material we present here is but an incremental change or modification to some of the fine works we have read and admired. We are in complete alignment with Isaac Newton, who viewed any knowledge as building upon itself.

Our debts in the writing of this book are many, and we spell them out now. Bear with us.

We acknowledge many in the 'References' section of each chapter in the book. We have tried our best but almost certainly failed to recognize some of the others by not making proper notes. To any such creators that we may have overlooked, our apologies.

We have learned a great deal from the writings of experts in the investor/trader community and the Python community on online Q&A forums like stackoverflow.com, quora.com, and others and are indebted to them. A special shout-out to Dave Bergstrom (Twitter handle @Dburgh), Matt Harrison (Twitter handle @_mharrison_) and PlanB (Twitter handle @100trillionUSD) for their comments on our work.

³https://en.wikipedia.org/wiki/Main_Page

We are also grateful to the helpful and supportive team members of QuantInsti. Many of them worked uncomplainingly on tight timelines and despite our badgering (or perhaps because :)), gave us insightful suggestions.

Finally, we would like to thank all the students we have taught in the past several years. A special thanks to those of you who endured our first few iterations of the lectures before we learned how best to teach it. Dear students, we exist because you exist. You have inspired us, challenged us, and pushed us never to stop learning just to keep up with you. We hope you enjoy reading this as much as we enjoyed writing it for you.

Suggestions and Errors

Any suggestions from you are welcome. We would be even more eager to receive any comments from you about errors in our work. Please write to us about any or all of these at contact@quantinsti.com.

Chapter 1

Introduction

Welcome to our first module on *programming*. In this module, we will be discussing the nuts and bolts of the Python programming language ranging from the very basic to more advanced topics.

Python is a general-purpose programming language that is becoming more and more popular for

- performing data analysis,
- automating tasks,
- learning data science,
- machine learning, etc.

1.1 What is Python?

Python is a dynamic, interpreted (bytecode-compiled) language that is used in a wide range of domains and technical fields. It was developed by Guido van Rossum in 1991. It was mainly developed for code readability and its syntax is such that it allows programmers to code/express concepts in fewer lines of code. Compared to compiled languages like C, Java, or Fortran, we do not need to declare the type of variables, functions, etc. when we write code in Python. This makes our code short and flexible. Python tracks the types of all values at runtime and flags code that does not make sense as it runs. On the Python website¹, we find the following executive summary.

¹<https://www.python.org/doc/essays/blurb/>

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms and can be freely distributed.

1.2 Where is Python used?

Python is used by the novice programmer as well as by the highly skilled professional developer. It is being used in academia, at web companies, in large corporations and financial institutions. It is used for

- *Web and Internet development:* Python is used on the server side to create web applications.
- *Software development:* Python is used to create GUI applications, connecting databases, etc.
- *Scientific and Numeric applications:* Python is used to handle big data and perform complex mathematics.
- *Education:* Python is a great language for teaching programming, both at the introductory level and in more advanced courses.
- *Desktop GUIs:* The Tk GUI library² included with most binary distributions of Python is used extensively to build desktop applications.
- *Business Applications:* Python is also used to build ERP and e-commerce systems.

1.3 Why Python?

Python is characterized by many features. Let's examine a few of them here:

²<https://wiki.python.org/moin/TkInter>

- **Simple**
 - Compared to many other programming languages, coding in Python is like writing simple strict English sentences. In fact, one of its oft-touted strengths is how Python code appears like pseudo-code. It allows us to concentrate on the solution to the problem rather than the language itself.
- **Easy to Learn**
 - As we will see, Python has a gentler learning curve (compared to languages like C, Java, etc.) due to its simple syntax.
- **Free and Open Source**
 - Python and the majority of supporting libraries available are open source and generally come with flexible and open licenses. It is an example of a FLOSS(Free/Libré and Open Source Software). In layman terms, we can freely distribute copies of open source software, access its source code, make changes to it, and use it in new free programs.
- **High-level**
 - Python is a programming language with strong abstraction from the details of the underlying platform or the machine. In contrast to low-level programming languages, it uses natural language elements, is easier to use, automates significant areas of computing systems such as resource allocation. This simplifies the development process when compared to a lower-level language. When we write programs in Python, we never need to bother about the lower-level details such as managing the memory used by programs we write, etc.
- **Dynamically Typed**
 - Types of variables, objects, etc. in Python are generally inferred during runtime and not statically assigned/declared as in most of the other compiled languages such as C or Fortran.
- **Portable/Platform Independent/Cross Platform**
 - Being open source and also with support across multiple platforms, Python can be ported to Windows, Linux and Mac

OS. All Python programs can work on any of these platforms without requiring any changes at all if we are careful in avoiding any platform-specific dependency. It is used in the running of powerful servers and also small devices like the Raspberry Pi³.

In addition to the above-mentioned platforms, following are some of the other platforms where Python can be used

- * FreeBSD OS
- * Oracle Solaris OS
- * AROS Research OS
- * QNX OS
- * BeOS
- * z/OS
- * VxWorks OS
- * RISC OS

- **Interpreted**

- A programming language can be broadly classified into two types viz. compiled or interpreted.
- A program written in a compiled language like C or C++ requires the code to be converted from the original language (C, C++, etc.) to a machine-readable language (like binary code i.e. 0 and 1) that is understood by a computer using a compiler with various flags and options. This compiled program is then fed to a computer memory to run it.
- Python, on other hand, does not require compilation to machine language. We directly *run* the program from the source code. Internally, Python converts the source code into an intermediate form known as byte code and then translates this into the native language of the underlying machine. We need not worry about proper linking and the loading into memory. This also enables Python to be much more portable, since we can run the same program onto another platform and it works just fine!
- The 'CPython' implementation is an interpreter of the language that translates Python code at runtime to executable byte code.

- **Multiparadigm**

³<https://www.raspberrypi.org/>

- Python supports various programming and implementation paradigms, such as *Object Oriented*, *Functional*, or *Procedural* programming.
- **Extensible**
 - If we need some piece of code to run fast, we can write that part of the code in C or C++ and then use it via our Python program. Conversely, we can embed Python code in a C/C++ program to give it *scripting* capabilities.
- **Extensive Libraries**
 - The Python Standard Library⁴ is huge and, it offers a wide range of facilities. It contains built-in modules written in C that provides access to system functionality such as I/O operations as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are listed below
 - * Text Processing Modules
 - * Data Types
 - * Numeric and Mathematical Modules
 - * Files and Directory Modules
 - * Cryptographic Modules
 - * Generic Operating System Modules
 - * Networking Modules
 - * Internet Protocols and Support Modules
 - * Multimedia Services
 - * Graphical User Interfaces with Tk
 - * Debugging and Profiling
 - * Software Development, Packaging and Distribution
 - In addition to the Python Standard Library, we have various other third-party libraries which can be accessed from Python Package Index⁵.

• **Garbage Collection**

- Python takes care of memory allocation and deallocation on its own. In other words, a programmer does not have to manage

⁴<https://docs.python.org/3/library/>

⁵<https://pypi.org/>

memory allocation and need not have to preallocate and deallocate memory before constructing variables and objects. Additionally, Python provides Garbage Collector⁶ interface to handle garbage collection.

1.4 History of Python

Python is the brainchild of Guido van Rossum who started its developmental efforts in the 1980s. Its name has nothing to do with anything serpentine, it's in fact inspired by the British comedy Monty Python! The first Python implementation was in December 1989 in the Netherlands. Since then, Python has gone through major turnarounds periodically. The following can be considered milestones in the development of Python:

- **Python 0.9.0** released in February 1991
- **Python 1.0** released in January 1994
- **Python 2.0** released in October 2000
- **Python 2.6** released in October 2008
- **Python 2.7** released in July 2010
- **Python 3.0** released in December 2008
- **Python 3.4** released in March 2014
- **Python 3.6** released in December 2016
- **Python 3.7** released in June 2018

Often times it is quite confusing for newcomers that there are **two major versions 2.x and 3.x available**, still being developed and in parallel use since 2008. This will likely persist for a while since both versions are quite popular and used extensively in the scientific and software development community. One point to note is that they are not entirely code compatible between the versions. We can develop programs and write code in either version but there will be syntactical and other differences. This handbook is based on the 3.x version, but we believe most of the code examples should work with version 2.x as well with some minor tweaks.

⁶<https://docs.python.org/3/library/gc.html>

1.5 Python 3 versus Python 2

The first version of the Python 3.x was released at the end of 2008. It made changes that made some of the old Python 2.x code incompatible. In this section, we will discuss the difference between the two versions. However, before moving further one might wonder why Python 3 and not Python 2. The most compelling reason for porting to Python 3 is, Python 2.x will not be developed after 2020. So it's no longer a good idea to start new projects in Python 2.x. There won't ever be a Python 2.8. Also, Python 2.7 will only get security updates from the Python 3 development branch. That being said, most of the code we write will work on either version with some small caveats.

A non-exhaustive list of features only available in 3.x releases are shown below.

- strings are Unicode by default
- clean Unicode/bytes separation
- exception chaining
- function annotations
- the syntax for keyword-only arguments
- extended tuple unpacking
- non-local variable declarations

We now discuss some of the significant changes between the two versions.

- **Unicode and Strings**

- There are two types of strings which can be broadly classified as *byte sequences* and *Unicode strings*.
- Byte sequences have to be literal characters from the ASCII alphabet. Unicode strings can hold onto pretty much any character we put in there. In Unicode, we can include various languages and, with the right encoding, emoji as well.
- In Python 2, we have to mark every single Unicode string with a `u` at the beginning, like `u'Hello Python!'`. As we use Unicode string every now and then, it becomes cumbersome to type a `u` for every Unicode string. If we forget to prefix the `u`, we would have a byte sequence instead.

- With the introduction to Python 3, we need not write a `u` every time. All strings are now Unicode by default and we have to mark byte sequences with a `b`. As Unicode is a much more common scenario, Python 3 has reduced development time for everyone this way.

- **Division with Integers**

- One of the core values of Python is to never do anything implicitly. For example, never turn a number into string unless a programmer codes for it. Unfortunately, Python 2 took this a bit too far. Consider the following operation

```
5 / 2
```

- The answer we expect here is 2.5, but instead Python 2 will return only 2. Following the core value mentioned above, Python will return the output of the same type as the input type. Here, the input is integer and Python returned the output as the integer.
- Again, this has been fixed in Python 3. It will now output 2.5 as the output to the above problem. In fact, it gives a float output to every division operation.

- **Print Function**

- The most significant change brought in Python 3 is with regard to the `print` keyword.
- In Python 2, if we are to output any text, we use `print` followed by output text, which would internally work as a function to render output.
- In Python 3, `print` has parentheses and hence, if we are to output any text, we use `print()` with the output text inside parentheses.

- **Input Function**

- There has been an important change to the `input()` function.
- In Python 2, we have `raw_input()` and `input()` functions for capturing user input from a terminal and standard input devices. `raw_input()` would capture input and treat everything as a string. Whereas, `input()` function helps a user in a way that

if an integer is inputted such as 123, it would be treated as an integer without being converted to a string. If a string is inputted for `input()`, Python 2 will throw an error.

- In Python 3, `raw_input()` is gone and `input()` no longer evaluates the data it receives. We always get back a string whatever the input may be.

- **Error Handling**

- There is a small change in the way each version handles errors.
- In Python 3, we need to use `as` keyword in `except` clause while checking error, whereas `as` keyword is not required in Python 2. Consider the following example

```
# Python 2
try:
    trying_to_check_error
except NameError, err: # 'as' keyword is NOT needed
    print (err, 'Error Occurred!')

# Python 3
try:
    trying_to_check_error
except NameError as err: # 'as' keyword is needed
    print (err, 'Error Occurred!')
```

- **`__future__` module**

- `__future__` module is introduced in Python 3 to allow backward compatibility, i.e. to use the features of Python 3 in code developed in Python 2.
- For example, if we are to use the division feature with float output or `print()` in Python 2, we can do so by using this module.

```
# Python 2 Code
from __future__ import division
from __future__ import print_function

print 5/2 # Output will be 2.5

print('Hello Python using Future module!')
```

Although we have discussed most of the key differences between two versions, there are many other changes being introduced or changed in Python 3 such as `next()` for generators and iterators, how `xrange()` became `range()`, etc.

1.6 Key Takeaways

1. Python is a high level and cross-platform language developed by Guido van Rossum in 1991.
2. It is used in many fields ranging from simple web development to scientific applications.
3. It is characterized by features such as ease of learning and extensibility to other languages.
4. In addition to built-in or standard libraries known as Python Standard Libraries, Python also offers support for third-party libraries.
5. It supports multiple programming styles like Object Oriented, Procedural and Functional.
6. There are two major versions of Python: 2.x and 3.x. The code developed in either version are to some extent compatible with each other.
7. The latest version in the Python 2.x franchise is Python 2.7. There won't be any new update in Python 2.x after 2020.

Chapter 2

Getting Started with Python

As we have seen in the previous section, Python offers different versions, comes in a variety of distributions and is suited for a myriad combinations of platform and devices. Thanks to its versatility, we can use Python to code nearly any task that we can think of logically. One of the most important tasks that a computer performs is mathematical computation. Python provides a direct interface to this fundamental functionality of modern computers. In fact an introduction of Python could be started by showing how it can be used as a tool for simple mathematical calculations.

2.1 Python as a Calculator

The easiest way to perform mathematical calculations using Python is to use the *Console*, which can be used as a fancy calculator. To begin with the simplest mathematical operations, such as addition, subtraction, multiplication and division, we can start using the Python *Console* using the following expressions.

```
# Addition
In []: 5 + 3
Out[]: 8

# Subtraction
In []: 5 - 3
Out[]: 2
```

```
# Multiplication
In []: 5 * 3
Out[]: 15

# Division
In []: 5 / 3
Out[]: 1.6666666666666667

# Modulo
In []: 5 % 2
Out[]: 1
```

NOTE: The content after the # symbols are comments and can be ignored when typing the examples. We will examine comments in more detail in the later sections. Here, In refers to an input provided to the Python interpreter and Out represents the output returned by the interpreter. Here we use the IPython console to perform the above mathematical operations. They can also be performed in the Python IDLE (Integrated Development and Learning Environment) (aka The Shell), the Python console, or Jupyter notebook in a similar fashion. Basically, we have a host of interfaces to choose from and programmers choose what they find most comfortable. We will stick to the Python Console interface to write and run our Python code in this handbook. To be clear, each of the above-mentioned interfaces connects us to the Python interpreter (which does the computational heavy lifting behind the scenes).

Let us dissect the above examples. A simple Python expression is similar to a mathematical expression. It consists of some numbers, connected by a mathematical *operator*. In programming terminology, numbers (this includes integers as well as numbers with fractional parts ex. 5, 3.89) are called *numeric literals*. An operator (e.g. +, -, /) indicates mathematical operation between its *operands*, and hence the *value* of the expression. The process of deriving the value of an expression is called the *evaluation* of the expression. When a mathematical expression is entered, the Python interpreter automatically evaluates and displays its value in the next line.

Similar to the / division operator, we also have the // integer division operator. The key difference is that the former outputs the decimal value known as a *float* which can be seen in the above example and the latter outputs an *integer* value i.e. without any fractional parts. We will discuss about the *float* and *integer* datatype in more detail in the upcoming sections. Below is an example of an integer division where Python returns the output value without any decimals.

```
In []: 5 // 3  
Out[]: 1
```

We can also use expressions as operands in longer *composite expressions*. For example,

```
# Composite expression  
In []: 5 + 3 - 3 + 4  
Out[]: 9
```

In the example above, the order of evaluation is from *left to right*, resulting in the expression $5 + 3$ evaluating first. Its value 8 is then combined with the next operand 3 by the $-$ operator, evaluating to the value 5 of the composite expression $5 + 3 - 3$. This value is in turn combined with the last literal 4 by the $+$ operator, ending up with the value 9 for the whole expression.

In the example, operators are applied from left to right, because $-$ and $+$ have the same priority. For an expression where we have more than one operators, it is not necessary all the operators have the same priority. Consider the following example,

```
In []: 5 + 3 * 3 - 4  
Out[]: 10
```

Here, the expression above evaluated to 10, because the $*$ operator has a higher priority compared to $-$ and $+$ operators. The expression $3 * 3$ is evaluated first resulting in the value of 9 which will be combined with the operand 5 by the operator $+$ producing the value of 14. This value is in turn combined with the next operand 4 by the operator $-$ which results in the final value of 10. The order in which operators are applied is called

operator precedence. In Python, mathematical operators follow the natural precedence observed in mathematics.

Similar to mathematical functions, Python allows the use of brackets (and) to manually specify the order of evaluation, like the one illustrated below:

```
# Brackets
In []: (5 + 3) * (3 - 4)
Out []: -8
```

The expression above evaluated to the value -8 as we explicitly defined the precedence for the expression $5 + 3$ to be evaluated first resulting in the value 8, followed by $3 - 4$ producing the value -1 and then finally we combined both the values 8 and -1 with operator * resulting in the final value to be -8.

In the examples above, an operator connects two operands, and hence they are called *binary operators*. In contrast, operators can also be *unary* which take only one operand. Such an operator is - known as negation.

```
# Negation
In []: - (5 + 3)
Out []: -8
```

First, We compute the expression $5 + 3$ resulting in the value 8 and secondly, we negate it with - operator producing the final value of -8.

2.1.1 Floating Point Expressions

Whatever examples we have seen so far were performed on integers also to yield integers. Notice that for the expression $5 / 3$ we would get a real number as an output even though the operands are integers. In computer science, real numbers are typically called *floating point numbers*. For example:

```
# Floating Point Addition
In []: 5.0 + 3
Out []: 8.0
```

```
# Floating Point Multiplication
In []: 5.0 * 3
Out[]: 15.0
```

```
# Floating Point Exponential
In []: 5.0 ** 3
Out[]: 125.0
```

```
# Floating Point Exponential
In []: 36 ** 0.5
Out[]: 6.0
```

For the above example, the last part calculates the positive square root of 36.

Python provides a very convenient option to check the type of number, be it an output of an expression or the operand itself.

```
In []: type(5)
Out[]: int
```

```
In []: type(5.0)
Out[]: float
```

```
In []: type(5.0 ** 3)
Out[]: float
```

The command `type(x)` returns the type of `x`. This command is a *function* in Python where `type()` is a built-in function in Python. We can call a function with a specific *argument* in order to obtain a return value. In the above example, calling a function `type` with the argument 5 returns the value `int` which means 5 is an Integer. The function calls can also be considered as an expression similar to mathematical expressions, with a function name followed by a comma-separated list of arguments enclosed within parentheses. The value of a function call expression is the return value of the function. Following the same example discussed above, `type` is the function name, which takes a single argument and returns the type of argument. As a result, the call to function `type(5.0)` returns the value as a `float`.

We can also convert the type of an argument using the following built-in functions. For example,

```
In []: float(5)
Out[]: 5.0
```

```
In []: type(float(5))
Out[]: float
```

```
In []: int(5.9)
Out[]: 5
```

```
In []: type(int(5.9))
Out[]: int
```

As can be seen in the above example, using the `float` function call, which takes a single argument, we can convert an integer input to a float value. Also, we cross verify it by using the `type` function. Likewise, we have an `int` function using which we can change a float input to the integer value. During the conversion process `int` function just ignores the fractional part of the input value. In the last example, the return value of `int(5.9)` is 5, even though 5.9 is numerically closer to the integer 6. For floating point conversion by rounding up an integer, we can use the `round` function.

```
In []: round(5.9)
Out[]: 6
```

```
In []: round(5.2)
Out[]: 5
```

A call to the `round` function will return a value numerically closer to the argument. It can also round up to a specific number of digits after the decimal point in the argument. We then need to specify two arguments to the function call, with the second argument specifying the number of digits to keep after the decimal point. The following examples illustrate the same. A *comma* is used to separate the arguments in the function call.

```
In []: round(5.98765, 2)
Out[]: 5.99
```

```
In []: round(5.98765, 1)
Out[]: 6.0
```

Another useful function is `abs`, which takes one numerical argument and returns its absolute value.

```
In []: abs(-5)
Out[]: 5
```

```
In []: abs(5)
Out[]: 5
```

```
In []: abs(5.0)
Out[]: 5.0
```

We can also express a floating point expression using *scientific notation* in the following manner.

```
In []: 5e1
Out[]: 50.0
```

```
In []: 5e-1
Out[]: 0.5
```

```
In []: 5E2
Out[]: 500.0
```

2.2 Python Basics

We have just completed a brief overview of some functionalities in Python. Let's now get ourselves acquainted with the basics of Python.

2.2.1 Literal Constants

We have seen literals and their use in the above examples. Here we define what they are. Some more concrete examples of numeric literals are 5, 2.85, or string literals are I am a string or Welcome to EPAT!.

It is called a *literal* because we use its value literally. The number 5 always represents itself and nothing else -- it is a *constant* because its value cannot be changed. Similarly, value 2.85 represents itself. Hence, all these are said to be a literal constant.

2.2.2 Numbers

We have already covered numbers in detail in the above section. Here we will discuss it in brief. Numbers can be broadly classified into two types - integer and float.

Examples of an integer number (*int* for short) are - 5, 3, 2, etc. It is just a whole number.

Examples of a floating point number (*floats* for short) are - 2.98745, 5.5, 5e-1, etc. Here, e refers to the power of 10. We can write either e or E, both work just fine.

NOTE: As compared to other programming languages, we do not have separate `long` or `double`. In Python, `int` can be of any length.

2.2.3 Strings

Simply put, a string is a sequence of characters. We use strings almost everywhere in Python code. Python supports both ASCII and Unicode strings. Let us explore strings in more detail.

Single Quote - We can specify a string using single quotes such as 'Python is an easy programming language!'. All spaces and tabs within the quotes are preserved as-is.

Double Quotes - We can also specify string using double quotes such as "Yes! Indeed, Python is easy.". Double quotes work the same way single quotes works. Either can be used.

Triple Quotes - This is used as a delimiter to mark the start and end of a comment. We explain it in greater detail in the next topic.

Strings are immutable - This means once we have created a string we cannot change it. Consider the following example.

```
In []: month_name = 'Fanuary'  
  
# We will be presented with an error at this line.  
In []: month_name[0] = 'J'  
Traceback (most recent call last):  
  
File "<ipython-input-24>", line 1, in <module>  
    month_name[0] = 'J'  
  
TypeError: 'str' object does not support item assignment
```

Here, `month_name` is a variable that is used to hold the value `Fanuary`. Variables can be thought of as a container with a name that is used to hold the value. We will discuss variables in detail in the upcoming sections.

In the above example, we initialize variable `month_name` with an incorrect month name `Fanuary`. Later, we try to correct it by replacing the letter `F` with `J`, where we have been presented with the `TypeError` telling us that strings do not support change operation.

2.2.4 Comments

We have already seen comments before. Comments are used to annotate codes, and they are not interpreted by Python. Comments in Python start with the hash character `#` and end at the end of the physical line in the code. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. This type of comment is also known as a single-line comment.

The other way we can annotate code is by using a multi-line comment that serves as a reference or documentation for others to understand the code.

Let us have a look at the single-line comment in the following example.

```
# Following line adds two integer numbers
In []: 5 + 3
Out[]: 8
```

We can write a comment after code line to annotate what particular line does as depicted in the following example.

```
In []: 5 + 3 # Add two literals
Out[]: 8
```

```
In []: # This is also a comment!
```

Python does not support multi-line/block comments. However, we can span comments across multiple lines using the same notation that we use for single line comments. Block comments are intended to the same level as that of code. Each line of a block comment starts with a # and a single space.

```
# This is an example of a multi-line comment
# in Python that spans several lines and
# describes the code. It can be used to
# annotate anything like the author name, revisions,
# the purpose of a script, etc. It uses a delimiter
# to mark its start and end.
```

It is always a good programming practice to generously intersperse our code with comments. Remember: The code tells you how, the comment tells you why.

2.2.5 print() function

The `print()` function is a very versatile tool that is used to print anything in Python.

NOTE: In Python 2, `print` is just a statement without parenthesis and NOT a function, whereas in Python 3, `print()` is a function and the content that we need to be outputted goes inside a parenthesis. We have covered this in detail in Chapter 1 under the topic 'Python 2 versus Python 3'.

Let us visit a few examples to understand how the `print()` works.

```
# Simple print function
In []: print("Hello World!")
Out[]: Hello World!

# Concatenating string and integer literal
In []: print("January", 2010)
Out[]: January 2010

# Concatenating two strings
# Here, stock_name is a variable containing stock name.
In []: print("AAPL", "is the stock ticker of Apple Inc.")
Out[]: AAPL is the stock ticker of Apple Inc.

# Concatenating a variable and string.
# Here, stock_name is a variable containing stock name.
In []: print(stock_name + " is the stock name of
Microsoft Corporation.")
Out[]: MSFT is the stock name of Microsoft Corporation.
```

As we can see in the above examples, `print()` can be used in a variety of ways. We can use it to print a simple statement, concatenate with a literal, concatenate two strings, or combine a string with a variable. A common way to use the `print()` function is *f-strings* or *formatted string literal*.

```
# f-strings
In []: print(f'The stock ticker for Apple Inc
           is {stock_name}.')
Out[]: The stock ticker for Apple Inc is AAPL.
```

The above string is called formatted string literal. Such strings are preceded by the letter `f` indicating that it be formatted when we use variable names between curly brackets `{}`. `stock_name` here is a variable name containing the symbol for a stock.

One more way to print a string is by using *%-formatting* style.

```
# %-formatting strings
In []: print("%s is currently trading at %.2f." % ("AAPL", 100))
```

```
%s  
Out []: AAPL is currently trading at 226.41.
```

Here we print the current trading price of AAPL stock. A stock name is stored in the variable `stock_name`, and its price is stored in the variable `price`. `%s` is used for specifying a string literal and `%f` is used to specify float literal. We use `%.2f` to limit two digits after the decimal point.

2.2.6 `format()` function

Another useful function for printing and constructing string for output is `format()` function. Using this function, we can construct a string from information like variables. Consider the following code snippet.

```
In []: stock_ticker = 'AAPL'  
In []: price = 226.41  
In []: print('We are interested in {x} which is currently  
trading at {y}'.format(x=stock_ticker, y=price))
```

Upon running the above code, we will be presented with the following output.

```
# Output  
Out []: We are interested in AAPL which is currently  
trading at 226.41
```

Above code will first prepare a string internally by substituting the `x` and `y` placeholders with variables `stock_ticker` and `price` respectively, and then prints the final output as a single string. Instead of using placeholders, we can also construct a string in the following manner:

```
In []: print('We are interested in {0} which is currently  
trading at {1}'.format(stock_ticker, price))
```

Here, the output will be similar to the above illustration. A string can be constructed using certain specifications, and the `format` function can be called to substitute those specifications with corresponding arguments of the `format` function. In the above example, `{0}` will be substituted by variable `stock_ticker` and similarly, `{1}` will get a value of `price`. Numbers provided inside the specification are optional, and hence we can also write the same statement as follows

```
print('We are interested in {} which is currently trading  
at {}'.format(stock_ticker, price))
```

which will provide the same exact output as shown above.

2.2.7 Escape Sequence

Escape characters are generally used to perform certain tasks and their usage in code directs the compiler to take a suitable action mapped to that character.

Suppose we want to write a string That's really easy.. If we are to write this within a double quote ", we could write it as "That's really easy.", but what if we are to write the same string within single quote like 'That's really easy.', we cannot write it because we have three ' and the Python interpreter will get confused as to where the string starts and ends. Hence, we need to specify that the string does not end at s in the string, instead, it is a part of the string. We can achieve this by using the *escape sequence*. We can specify it by using a \ (backslash character). For example,

```
In []: 'That\'s really easy.'  
Out[]: "That's really easy."
```

Here, we preceded ' with a \ character to indicate that it is a part of the string. Similarly, we need to use an escape sequence for double quotes if we are to write a string within double quotes. Also, we can include the backslash character in a string using \\\. We can break a single line into multiple lines using the \n escape sequence.

```
In []: print('That is really easy.\nYes, it really is.')  
Out[]: That is really easy.  
      Yes, it really is.
```

Another useful escape character is \t tab escape sequence. It is used to leave tab spaces between strings.

```
In []: print('AAPL.\tNIFTY50.\tDJIA.\tNIKKEI225.')  
Out[]: AAPL.      NIFTY50.          DJIA.          NIKKEI225.
```

In a string, if we are to mention a single \ at the end of the line, it indicates that the string is continued in the next line and no new line is added. Consider below example:

```
In []: print('AAPL is the ticker for Apple Inc. \
...:      It is a technology company.')
Out[]: AAPL is the ticker for Apple Inc. It is a
technology company.
```

Likewise, there are many more escape sequences which can be found on the official Python documentation¹.

2.2.8 Indentation

Whitespaces are important in Python. Whitespace at the start of a line is called *indentation*. It is used to mark the start of a new code block. A block or code block is a group of statements in a program or a script. Leading spaces at the beginning of a line are used to determine the indentation level, which in turn is used to determine the grouping of statements. Also, statements which go together *must* have same indentation level.

A wrong indentation raises the error. For example,

```
stock_name = 'AAPL'

# Incorrect indentation. Note a whitespace at
# the beginning of line.
print('Stock name is', stock_name)

# Correct indentation.
print('Stock name is', stock_name)
```

Upon running the following code, we will be presented with the following error

```
File "indentation_error.py", line 2
    print('Stock name is', stock_name)
^
IndentationError: unexpected indent
```

¹https://docs.python.org/3.6/reference/lexical_analysis.html#literals

The error indicates to us that the syntax of the program is invalid. That is, the program is not properly written. We *cannot* indent new blocks of statements arbitrarily. Indentation is used widely for defining new block while defining functions, control flow statement, etc. which we will be discussing in detail in the upcoming chapters.

NOTE: We either use four spaces or tab space for indentation.
Most of the modern editors do this automatically for us.

2.3 Key Takeaways

1. Python provides a direct interface known as a Python Console to interact and execute the code directly. The advanced version of the Python console is the IPython (Interactive Python) console.
2. A whole number is called an integer and a fractional number is called a float. They are represented by `int` and `float` data types respectively.
3. Expressions are evaluated from left to right in Python. An expression with a float value as input will return the float output.
4. Strings are immutable in Python. They are written using a single quote or double quote and are represented by `str` data type. Any characters enclosed within quotes is considered a string.
5. Comments are used to annotate code. In Python, `#` character marks the beginning of a single line comment. Python discards anything written after the `#`.
6. Multiline comments are within triple single or double quotes. They start and end with either `"""` or `'''`.
7. Use the `type()` function to determine the type of data, `print()` to print on the standard output device and `format()` to format the output.
8. Use the escape character to escape certain characters within a string. They can be used to mark tab within a line, a new line within a string and so on.
9. Blocks of code are separated using indentation in Python. Code statements which go together within a single block must have the same indentation level. Otherwise, Python will generate an `IndentationError`.

Chapter 3

Variables and Data Types in Python

We have previously seen that a variable can take data in various formats such as a string, an integer, a number with fractional parts (float), etc. It is now time to look at each of these concepts in greater detail. We start by defining a variable.

3.1 Variables

A variable can be thought of as a container having a *name* which is used to store a *value*. In programming parlance, it is a reserved memory location to store values. In other words, a variable in a Python program gives necessary data to a computer for processing.

In this section, we will learn about variables and their types. Let start by creating a variable.

3.1.1 Variable Declaration and Assignment

In Python, variables need NOT be declared or defined in advance, as is the case in many other programming languages. In fact, Python has no command for declaring a variable. To create a variable, we assign a value to it and start using it. An assignment is performed using a single equal

sign = a.k.a. Assignment operator. A variable is created the moment we assign the first value to it.

```
# Creating a variable  
In []: price = 226
```

The above statement can be interpreted as a variable `price` is assigned a value 226. It is also known as *initializing* the variable. Once this statement is executed, we can start using the `price` in other statements or expressions, and its value will be substituted. For example,

```
In []: print(price)  
Out[]: 226 # Output
```

Later, if we change the value of `price` and run the `print` statement again, the new value will appear as output. This is known as *re-declaration* of the variable.

```
In []: price = 230 # Assigning new value  
In []: print(price) # Printing price  
Out[]: 230 # Output
```

We can also chain assignment operation to variables in Python, which makes it possible to assign the same value to multiple variables simultaneously.

```
In []: x = y = z = 200 # Chaining assignment operation  
In []: print(x, y, z) # Printing all variables  
Out[]: 200 200 200 # Output
```

The chained assignment shown in the above example assigns the value 200 to variables `x`, `y`, and `z` simultaneously.

3.1.2 Variable Naming Conventions

We use variables everywhere in Python. A variable can have a short name or more descriptive name. The following list of rules should be followed for naming a variable.

- A variable name must start with a letter or the underscore character.

```
stock = 'AAPL' # Valid name  
_name = 'AAPL' # Valid name
```

- A variable name cannot start with a number.

```
1stock = 'AAPL' # Invalid name  
1_stock = 'AAPL' # Invalid name
```

- A variable name can only contain alpha-numeric characters(A-Z, a-z, 0-9) and underscores(_).

```
# Valid name. It starts with a capital letter.  
Stock = 'AAPL'
```

```
# Valid name. It is a combination of alphabets  
# and the underscore.  
stock_price = 226.41
```

```
# Valid name. It is a combination of alphabets  
# and a number.  
stock_1 = 'AAPL'
```

```
# Valid name. It is a combination of a capital  
# letter, alphabets and a number.  
Stock_name_2 = 'MSFT'
```

- A variable name cannot contain whitespace and signs such as +, -, etc.

```
# Invalid name. It cannot contain the whitespace.  
stock name = 'AAPL'
```

```
# Invalid name. It cannot contain characters  
# other than the underscore(_).  
stock-name = 'AAPL'
```

- Variable names are case-sensitive.

```
# STOCK, stock and Stock all three are different  
# variable names.
```

```
STOCK = 'AAPL'  
stock = 'MSFT'  
Stock = 'GOOG'
```

Remember that *variable names are case-sensitive* in Python.

- Python keywords cannot be used as a variable name.

```
# 'str', 'is', and 'for' CANNOT be used as the  
# variable name as they are reserved keywords  
# in Python. Below given names are invalid.  
str = 'AAPL'  
is = 'A Variable'  
for = 'Dummy Variable'
```

The following points are *de facto* practices followed by professional programmers.

- Use a name that describes the purpose, instead of using dummy names. In other words, it should be meaningful.

```
# Valid name but the variable does not  
# describe the purpose.  
a = 18  
  
# Valid name which describes it suitably  
age = 18
```

- Use an underscore character _ to separate two words.

```
# Valid name.  
stockname = 'AAPL'  
  
# Valid name. And it also provides concise  
# readability.  
stock_name = 'AAPL'
```

- Start a variable name with a small alphabet letter.

```
# Valid name.  
Stock_name = 'AAPL'  
  
# Valid name. Additionally, it refers to uniformity  
# with other statements.  
stock_name = 'AAPL'
```

NOTE: Adhering to these rules increases readability of code. Remember these are good coding practices (and recommended but by no means necessary to follow) which you can carry with you to any programming language, not just Python.

3.2 Data Types

Having understood what variables are and how they are used to store values, its time to learn data types of values that variables hold. We will learn about primitive data types such as numeric, string and boolean that are built into Python. Python has four basic data types:

- Integer
- Float
- String
- Boolean

Though we have already had a brief overview of integer, float and string in the previous section, we will cover these data types in greater detail in this section.

3.2.1 Integer

An integer can be thought of as a numeric value without any decimal. In fact, it is used to describe any *whole number* in Python such as 7, 256, 1024, etc. We use an integer value to represent a numeric data from negative infinity to infinity. Such numeric numbers are assigned to variables using an assignment operator.

```
In []: total_output_of_dice_roll = 6  
In []: days_elapsed = 30
```

```
In []: total_months = 12
In []: year = 2019
```

We assign a whole number 6 to a variable `total_output_of_dice_roll` as there can be no fractional output for a dice roll. Similarly, we have a variable `days_elapsed` with value 30, `total_months` having a value 12, and `year` as 2019.

3.2.2 Float

A float stands for *floating point number* which essentially means a number with fractional parts. It can also be used for rational numbers, usually ending with a decimal such as 6.5, 100.1, 123.45, etc. Below are some examples where a float value is more appropriate rather than an integer.

```
In []: stock_price = 224.61
In []: height = 6.2
In []: weight = 60.4
```

NOTE: From the statistics perspective, a float value can be thought of as a continuous value, whereas an integer value can correspondingly be a discrete value.

By doing so, we get a fairly good idea how data types and variable names go hand in hand. This, in turn, can be used in expressions to perform any mathematical calculation.

Let's revisit the topic *Python as a Calculator* very briefly but this time using variables.

```
# Assign an integer value
In []: x = 2

# Assign a float value
In []: y = 10.0

# Addition
In []: print(x + y)
Out[]: 12.0
```

```
# Subtraction
In []: print(x - y)
Out[]: -8.0

# Multiplication
In []: print(x * y)
Out[]: 20.0

# Division
In []: print(x / y)
Out[]: 0.2

# Modulo
In []: print(x % y)
Out[]: 2.0

# Exponential / Power
In []: print(x ** y)
Out[]: 1024.0
```

NOTE: Please note the precise use of *comments* used in the code snippet to describe the functionality. Also, note that *output* of all expressions to be *float* number as one of the literals used in the input is a float value.

Look at the above-mentioned examples and try to understand the code snippet. If you are able to get a sense of what's happening, that's great. You are well on track on this Pythonic journey. Nevertheless, let's try to understand the code just to get more clarity. Here, we assign an integer value of 2 to x and a float value of 10.0 to y. Then, we try to attempt various mathematical operations on these defined variables instead of using direct values. The obvious benefit is the flexibility that we get by using these variables. For example, think of a situation where we want to perform the said operation on different values such as 3 and 15.0, we just need to re-declare variables x and y with new values respectively, and the rest of the code remains as it is.

3.2.3 Boolean

This built-in data type can have one of two values, True or False. We use an assignment operator = to assign a boolean value to variables in a manner similar to what we have seen for integer and float values. For example:

```
In []: buy = True  
  
In []: print(buy)  
Out[]: True  
  
In []: sell = False  
  
In []: print(sell)  
Out[]: False
```

As we will see in upcoming sections, expressions in Python are often evaluated in the boolean context, meaning they are interpreted to represent their truth value. Boolean expressions are extensively used in logical conditions and control flow statements. Consider the following examples

```
# Checking for equality between 1 and itself using  
# comparison operator '=='.  
In []: 1 == 1  
Out[]: True  
  
# Checking for equality between values 1 and -1  
In []: 1 == -1  
Out[]: False  
  
# Comparing value 1 with -1  
In []: 1 > -1  
Out[]: True
```

The above examples are some of the simplest boolean expressions that evaluate to either True or False.

NOTE: We do NOT write True and False within quotes. It needs to be written without quotes. Also, the first letter needs to be

upper case followed by lower case letters. The following list will not be evaluated to a boolean value - 'TRUE' - TRUE - true - 'FALSE' - FALSE - false

3.2.4 String

A string is a collection of alphabets, numbers, and other characters written within a single quote ' or double quotes ". In other words, it is a sequence of characters within quotes. Let us understand how a string works with the help of some examples.

```
# Variable assignment with a string
In []: sample_string = '1% can also be expressed as 0.01'

# Print the variable sample_string
In []: sample_string
Out[]: '1% can also be expressed as 0.01'
```

In the above examples we have defined a string variable with name `sample_string` assigned a value '1% can also be expressed as 0.01'. It is interesting to note here that we have used a combination of alphabets, numbers and special characters for defining the variable. In Python, anything that goes within quotes is a string. Consider the following example,

```
In []: stock_price = '224.61'

In []: stock_price
Out[]: '224.61'
```

We define the variable `stock_price` assigning the string value '224.61'. The interesting thing to notice is the output of the variable is also a string. Python will not convert the data types implicitly whenever numeric values are given as a string.

We can concatenate two or more string using the + operator.

```
In []: 'Price of AAPL is ' + stock_price
Out[]: 'Price of AAPL is 224.61'
```

Concatenation operation using the + operator works only on a string. It does not work with different data types. If we try to perform the operation, we will be presented with an error.

```
# Re-declaring the variable with an integer value
In []: stock_price = 224.61

In []: 'Price of AAPL is ' + stock_price # Error line
Traceback (most recent call last):

File "<ipython-input-28>", line 1, in <module>
    'Price of AAPL is ' + stock_price

TypeError: must be str, not float
```

As expected, Python spat out a `TypeError` when we tried concatenating a string and float literal. Similar to the `+` operator, we can use the `*` operator with a string literal to produce the same string multiple times.

```
In []: string = 'Python! '
In []: string * 3
Out[]: 'Python! Python! Python! '
```

We can select a substring or part of a string using the slice operation. Slicing is performed using the square brackets `[]`. The syntax for slicing a single element from the string is `[index]` which will return an element at `index`. The index refers to the position of each element in a string and it begins with 0, which keeps on increasing in chronological order for every next element.

```
In []: string = 'EPAT Handbook!'
In []: string[0] # 0 refers to an element E
Out[]: 'E'

In []: string[1] # 1 refers to an element P
Out[]: 'P'
```

In the above example, an element `E` being the first character belongs to an index 0, `P` being next to `E` belongs to an index 1, and so on. Similarly, the index for element `b` will be 9. Can you guess the index for element `k` in the above example?

To slice a substring from a string, the syntax used is [start index:end index] which will return the substring starting from an element at start index up to but not including an element at end index. Consider the following example, where we substring the string from an index 0 up to 4 which yields the output 'EPAT'. Notice how the element ' ' at an index 4 is not included in the output. Similarly, we slice a substring as seen in the below example.

```
In []: string[0:4]
Out[]: 'EPAT'
```

```
In []: string[4]
Out[]: ''
```

```
In []: string[5:13]
Out[]: 'Handbook'
```

```
In []: string[13]
Out[]: '!!'
```

In Python, we cannot perform the slicing operation with an index not present in the string. Python will throw `IndexError` whenever it encounters slicing operation with incorrect index.

```
In []: string[14]
Traceback (most recent call last):

File "<ipython-input-36>", line 1, in <module>
    string[14]

IndexError: string index out of range
```

In the above example, the last index is 13. The slicing operation performed with an index 14 will result in an error `IndexError` stating that index we are looking for is not present.

NOTE: We list out some of the important points for string literals below:

- In Python 3.x all strings are *Unicode* by default.
- A string can be written within either ' ' or " ". Both work fine.

Strings are immutable. (although you can modify the variable)
- An escape sequence is used within a string to mark a new line, provide tab space, writing \ character, etc.

3.2.5 Operations on String

Here we discuss some of the most common string methods. A method is like a function, but it runs *on* an object. If the variable `sample_string` is a string, then the code `sample_string.upper()` runs the `upper()` method on that string object and returns the result (this idea of running a method on an object is one of the basic ideas that make up Object Oriented Programming, OOP). Some methods also take an argument as a parameter. We provide a parameter to a method as an argument within parentheses.

- `upper()` method: This method returns the upper case version of the string.

```
In []: sample_string.upper()  
Out[]: 'EPAT HANDBOOK!'
```

- `lower()` method: This method returns the lower case version of the string.

```
In []: sample_string.lower()  
Out[]: 'epat handbook!'
```

- `strip()` method: This method returns a string with whitespace removed from the start and end.

```
In []: ' A string with whitespace at both \  
the ends. '.strip()  
Out[]: 'A string with whitespace at both the ends.'
```

- `isalpha()` method: This method returns the boolean value `True` if all characters in a string are letters, `False` otherwise.

```
In []: 'Alphabets'.isalpha()  
Out[]: True
```

```
# The string under evaluation contains whitespace.  
In []: 'This string contains only alphabets'.isalpha()  
Out[]: False
```

- `isdigit()` method: This method returns the boolean value `True` if all characters in a string are digits, `False` otherwise.

```
In []: '12345'.isdigit()  
Out[]: True
```

- `startswith(argument)` method: This method returns the boolean value `True` if the first character of a string starts with the character provided as an argument, `False` otherwise.

```
In []: 'EPAT Handbook!'.startswith('E')  
Out[]: True
```

- `endswith(argument)` method: This method returns the boolean value `True` if the last character of a string ends with the character provided as an argument, `False` otherwise.

```
In []: 'EPAT Handbook!'.startswith('k')  
Out[]: False # String ends with the '!' character.
```

- `find(sub, start, end)` method: This method returns the lowest index in a string where substring `sub` is found within the slice `[start:end]`. Here, arguments `start` and `end` are optional. It returns `-1` if `sub` is not found.

```
In []: 'EPAT Handbook!'.find('EPAT')  
Out[]: 0
```

```
In []: 'EPAT Handbook!'.find('A')  
Out[]: 2 # First occurrence of 'A' is at index 2.
```

```
In []: 'EPAT Handbook!'.find('Z')  
Out[]: -1 # We do not have 'Z' in the string.
```

- `replace(old, new)` method: This method returns a copy of the string with all occurrences of `old` replace by `new`.

```
Out[]: '00 01 10 11'.replace('0', '1')
Out[]: '11 11 11 11' # Replace 0 with 1
```

```
In []: '00 01 10 11'.replace('1', '0')
Out[]: '00 00 00 00' # Replace 1 with 0
```

- `split(delim)` method: This method is used to split a string into multiple strings based on the `delim` argument.

```
In []: 'AAPL MSFT GOOG'.split(' ')
Out[]: ['AAPL', 'MSFT', 'GOOG']
```

Here, the Python outputs three strings in a single data structure called *List*. We will learn `list` in more detail in the upcoming section.

- `index(character)` method: This method returns the index of the first occurrence of the character.

```
In []: 'EPAT Handbook!'.index('P')
Out[]: 1
```

Python will provide an error if the character provided as an argument is not found within the string.

```
In []: 'EPAT Handbook!'.index('Z')
Traceback (most recent call last):

File "<ipython-input-52>", line 1, in <module>
    'EPAT Handbook!'.index('Z')

ValueError: substring not found
```

- `capitalize()` method: This method returns a capitalized version of the string.

```
In []: 'python is amazing!'.capitalize()
Out[]: 'Python is amazing!'
```

- `count(character)` method: This method returns a count of an argument provided by `character`.

```
In []: 'EPAT Handbook'.count('o')
Out[]: 2
```

```
In []: 'EPAT Handbook'.count('a')
Out[]: 1
```

3.2.6 type() function

The inbuilt `type(argument)` function is used to evaluate the data type and returns the class type of the argument passed as a parameter. This function is mainly used for debugging.

```
# A string is represented by the class 'str'.
In []: type('EPAT Handbook')
Out[]: str
```

```
# A float literal is represented by the class 'float'.
In []: type(224.61)
Out[]: float
```

```
# An integer literal is represented by the class 'int'.
In []: type(224)
Out[]: int
```

```
# An argument provided is within quotation marks.
In []: type('0')
Out[]: str
```

```
# A boolean value is represented by the class 'bool'.
In []: type(True)
Out[]: bool
```

```
In []: type(False)
Out[]: bool
```

```
# An argument is provided within a quotation mark.
In []: type('False')
Out[]: str
```

```

# An object passed as an argument belongs to the
# class 'list'.
In []: type([1, 2, 3])
Out[]: list

# An object passed as an argument belongs to the
# class 'dict'.
In []: type({'key':'value'})
Out[]: dict

# An object passed as an argument belongs to the
# class 'tuple'.
In []: type((1, 2, 3))
Out[]: tuple

# An object passed as an argument belongs to the
# class 'set'.
In []: type({1, 2, 3})
Out[]: set

```

A `list`, `dict`, `tuple`, `set` are native data structures within Python. We will learn these data structures in the upcoming section.

3.3 Type Conversion

We often encounter situations where it becomes necessary to change the data type of the underlying data. Or maybe we find out that we have been using an integer when what we really need is a float. In such cases, we can convert the data types of variables. We can check the data type of a variable using `type()` function as seen above.

There can be two types of conversion possible: *implicit* termed as coercion, and *explicit* often referred to as casting. When we change the type of a variable from one to another, this is called *typecasting*.

Implicit Conversion: This is an automatic type conversion and the Python interpreter handles this on the fly for us. We need not to specify any command or function for same. Take a look at the following example:

```
In []: 8 / 2
Out[]: 4.0
```

The division operation performed between two integers 8 being a dividend and 2 being a divisor. Mathematically, we expect the output to be 4 - an integer value, but instead, Python returned the output as 4.0 - a float value. That is, Python internally converted an integer 4 to float 4.0.

Explicit Conversion : This type of conversion is user-defined. We need to explicitly change the data type for certain literals to make it compatible for data operations. Let us try to concatenate a string and an integer using the + operator.

```
In []: 'This is the year ' + 2019
Traceback (most recent call last):

File "<ipython-input-68>", line 1, in <module>
  'This is the year ' + 2019

TypeError: must be str, not int
```

Here we attempted to join a string 'This is the year ' and an integer 2019. Doing so, Python threw an error `TypeError` stating incompatible data types. One way to perform the concatenation between the two is to convert the data type of 2019 to string explicitly and then perform the operation. We use `str()` to convert an integer to string.

```
In []: 'This is the year ' + str(2019)
Out[]: 'This is the year 2019'
```

Similarly, we can explicitly change the data type of literals in the following manner.

```
# Integer to float conversion
In []: float(4)
Out[]: 4.0

# String to float conversion
In []: float('4.2')
```

```
Out[]: 4.2

In []: float('4.0')
Out[]: 4.0

# Float to integer conversion
In []: int(4.0)
Out[]: 4 # Python will drop the fractional part.

In []: int(4.2)
Out[]: 4

# String to integer conversion
In []: int('4')
Out[]: 4

# Python does not convert a string literal with a
# fractional part, and instead, it will throw an error.
In []: int('4.0')
Traceback (most recent call last):

File "<ipython-input-75>", line 1, in <module>
    int('4.0')

ValueError: invalid literal for int() with base 10: '4.0'

# Float to string conversion
In []: str(4.2)
Out[]: '4.2'

# Integer to string conversion
In []: str(4)
Out[]: '4'
```

In the above example, we have seen how we can change the data type of literals from one to another. Similarly, the boolean data type represented by `bool` is no different. We can typecast `bool` to `int` as we do for the rest. In fact, Python internally treats the boolean value `False` as 0 and `True` as 1.

```
# Boolean to integer conversion
In []: int(False)
Out[]: 0

In []: int(True)
Out[]: 1
```

It is also possible to convert an integer value to boolean value. Python converts 0 to False and rest all integers gets converted to True.

```
# Integer to boolean conversion
In []: bool(0)
Out[]: False

In []: bool(1)
Out[]: True

In []: bool(-1)
Out[]: True

In []: bool(125)
Out[]: True
```

In this section, we started with familiarizing ourselves with variables, and then went on to define them, understanding data types, their internal workings, and type conversions.

3.4 Key Takeaways

1. A variable is used to store a value that can be used repetitively based on the requirement within a program.
2. Python is a loosely typed language. It is not required to specify the type of a variable while declaring it. Python determines the type of the variable based on the value assigned to it.
3. Assignment operator = is used for assigning a value to a variable.
4. Variable names should start with either a letter or an underscore character. They can contain alpha-numeric characters only. It is a good programming practice to have descriptive variable names.

5. Variable names are case sensitive and cannot start with a number.
6. There are four primitive data types in Python:
 - (a) Integer represented by `int`
 - (b) Float represented by `float`
 - (c) String represented by `str`
 - (d) Boolean (`True` or `False`) represented by `bool`
7. Internally, `True` is treated as 1 and `False` is treated as 0 in Python.
8. A substring or a part of a string is selected using the square brackets `[]` also known as slice operation.
9. Type conversion happens either implicitly or explicitly.
 - (a) Implicit type conversion happens when an operation with compatible data types is executed. For example, `4/2` (integer division) will return `2.0` (float output).
 - (b) When an operation involves incompatible data types, they need to be converted to compatible or similar data type. For example: To print a string and an integer together, the integer value needs to be converted to a string before printing.

Chapter 4

Modules, Packages and Libraries

To start with, a module allows us to organize Python code in a systematic manner. It can be considered as a file consisting of Python code. A module can define functions, classes and variables. It can also include runnable code. Consider writing code directly on the Python or IPython console. The definitions that we create(functions and variables) will be lost if we quit the console and enter it again. Therefore, in order to write a longer program, we might consider switching to a text editor to prepare an input for the interpreter and running it with that file as an input instead. This is known as writing a *script*. As a program gets longer, we may want it to split it into several small files for easier maintenance. Also, we may want to use a handy function that we have written in several programs without copying its definition into each program.

To support this, Python has a way to put a code definition in a file and use them in another script or directly in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or in the program that we code.

As we discussed above, a module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. For instance, we create a file called arithmetic.py in the current directory with the following contents:

```

# -*- coding: utf-8 -*-
"""
Created on Fri Sep 21 09:29:05 2018
@filename: arithmetic.py
@author: Jay Parmar
"""

def addition(a, b):
    """Returns the sum of of two numbers"""
    return a + b

def multiply(a, b):
    """Returns the product of two numbers"""
    return a * b

def division(dividend, divisor):
    """
    Performs the division operation between the dividend
    and divisor
    """
    return dividend / divisor

def factorial(n):
    """Returns the factorial of n"""
    i = 0
    result = 1
    while(i != n):
        i = i + 1
        result = result * i
    return result

```

We are now ready to import this file in other scripts or directly into the Python interpreter. We can do so with the following command:

In []: `import arithmetic`

Once we have imported the module, we can start using its definition in the script without re-writing the same code in the script. We can access func-

tions within the imported module using its name. Consider an example below:

```
In []: result = arithmetic.addition(2, 3)

In []: print(result)
Out[]: 5

In []: arithmetic.multiply(3, 5)
Out[]: 15

In []: arithmetic.division(10, 4)
Out[]: 2.5

In []: arithmetic.factorial(5)
Out[]: 120
```

A module name is available as a string within a script or the interpreter as the value of the global variable `__name__`.

```
In []: arithmetic.__name__
Out[]: 'arithmetic'
```

When we import the module named `arithmetic`, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named `arithmetic.py` in a list of directories given by the variables `sys.path`.

This variable is initialized from the following locations:

- The directory containing the input script (or the current directory).
- `PYTHONPATH` (An environment variable)
- The installation-dependent path.

Here, the module named `arithmetic` has been created that can be imported into the other modules as well. Apart from this, Python has a large set of built-in modules known as the *Python Standard Library*, which we will discuss next.

4.1 Standard Modules

Python comes with a library of standard modules also referred to as the Python Standard Library¹. Some modules are built into the interpreter; these modules provide access to operations that are not part of the core of the language but are either for efficiency or to provide access to tasks pertaining to the operating system. The set of such modules available also depends on the underlying platform. For example, `winreg`² module is available only on the Windows platform.

Python's standard library is very extensive and offers a wide range of facilities. The library contains built-in modules that provide access to system functionality such as file I/O operations as well as modules that provide standardized solutions for many problems that occur in everyday programming.

The Python installers for the Windows platform usually include the entire standard library and often also include many additional components. For Unix-like operating systems, Python is normally provided as a collection of packages, so it may be necessary to use the packaging tools provided with the operating system to obtain some or all of the optional components.

One particular module that deserves attention is `sys`, which is built into every Python interpreter. This module provides access to variables used or maintained by the interpreter and to functions that interact with the interpreter. It is always available and used as follows:

```
In []: import sys

# Returns a string containing the copyright pertaining to
# the Python interpreter
In []: sys.copyright
Out[]: 'Copyright (c) 2001-2018 Python Software Foundation.
        \nAll Rights Reserved.\n\nCopyright (c) 2000
        BeOpen.com.\nAll Rights Reserved.\n\n
        Copyright (c) 1995-2001 Corporation for National
```

¹<https://docs.python.org/3/library/>

²<https://docs.python.org/3/library/winreg.html#module-winreg>

```

Research Initiatives.\nAll Rights Reserved.\n\n
Copyright (c) 1991-1995 Stichting Mathematisch
Centrum, Amsterdam.\nAll Rights Reserved.'
```

*# Returns the name of the encoding used to convert between
unicode filenames and bytes filenames.*

```
In []: sys.getfilesystemencoding()
Out[]: 'utf-8'
```

Returns information regarding the Python interpreter

```
In []: sys.implementation
Out[]: namespace(cache_tag='cpython-36',
                 hexversion=50726384, name='cpython',
                 version=sys.version_info(major=3, minor=6,
                                          micro=5, releaselevel='final', serial=0))
```

Returns a string containing a platform identifier

```
In []: sys.platform
Out[]: 'win32'
```

*# Returns a string containing the version number of the
Python interpreter plus additional information on the
compiler*

```
In []: sys.version
Out[]: '3.6.5 |Anaconda, Inc.| (default, Mar 29 2018,
13:32:41) [MSC v.1900 64 bit (AMD64)]'
```

In the above examples, we discussed a handful of functionalities provided by the `sys` module. As can be seen, we can use it to access system level functionality through Python code. In addition to this, there are various other built-in modules in Python. We list some of them below based on their functionality.

- Text Processing : `string`, `readline`, `re`, `unicodedata`, etc.
- Data Types : `datetime`, `calendar`, `array`, `copy`, `pprint`, `enum`, etc.
- Mathematical : `numbers`, `math`, `random`, `decimal`, `statistics`, etc.
- Files and Directories : `pathlib`, `stat`, `glob`, `shutil`, `filinput`, etc.
- Data Persistence: `pickle`, `dbm`, `sqlite3`, etc.
- Compression and Archiving: `gzip`, `bz2`, `zipfile`, `tarfile`, etc.

- Concurrent Execution: `threading`, `multiprocessing`, `sched`, `queue`, etc.
- Networking: `socket`, `ssl`, `asyncio`, `signal`, etc.
- Internet Data Handling: `email`, `json`, `mailbox`, `mimetypes`, `binascii`, etc.
- Internet Protocols: `urllib`, `http`, `ftplib`, `smtplib`, `telnetlib`, `xmlrpc`, etc.

In addition to the standard library, there is a growing collection of several thousand modules ranging from individual modules to packages and entire application development frameworks, available from the *Python Package Index*.

4.2 Packages

Packages can be considered as a collection of modules. It is a way of structuring Python's module namespace by using "dotted module names". For example, the module name `matplotlib.pyplot` designates a submodule named `pyplot` in a package named `matplotlib`. Packaging modules in such a way saves the author of different modules from having to worry about each other's global variable names and the use of dotted module names saves the author of multi-module packages from having to worry about each other's module names.

Suppose we want to design a package (a collection of modules) for the uniform handling of various trading strategies and their data. There are many different data files based on data frequencies, so we may need to create and maintain a growing collection of modules for the conversion between the various data frequencies. Also, there are many different strategies and operations that we might need to perform. All of this put together means we would have to write a never-ending stream of modules to handle the combinatorics of data, strategies, and operations. Here's a possible package structure to make our lives easier.

<code>strats/</code>	<i>Top-level package</i>
<code>__init__.py</code>	<i>Initialize strats package</i>
<code> data/</code>	<i>Sub-package for data</i>
<code>__init__.py</code>	

```

equity.py           Equity module
currency.py
options.py
...
strategies/        Sub-package for strategies
    __init__.py
    rsi.py          RSI module
    macd.py
    smalma.py
    peratio.py
    fundamentalindex.py
    statisticalarbitrage.py
    turtle.py
...
operations/        Sub-package for operations
    __init__.py
    performanceanalytics.py
    dataconversion.py
...

```

When importing the package, Python searches through the directories in `sys.path` looking for the package subdirectory. The `__init__.py` file is required to make Python treat the directories as containing packages. If we are to use this package, we can do so in the following manner:

```

import strats.data.equity
import strats.strategies.statisticalarbitrage

```

Above statements loads the `equity` and `statisticalarbitrage` modules from the `data` and `strategies` sub-packages respectively under the `strats` package.

4.3 Installation of External Libraries

One of the great things about using Python is the number of fantastic code libraries (apart from the Python Standard Library) which are readily available for a wide variety of domains that can save much coding or make a particular task much easier to accomplish. Before we can use such

external libraries, we need to install them.

The goal here is to install software that can automatically download and install Python modules/libraries for us. Two commonly used installation managers are `conda`³ and `pip`⁴. We choose to go with `pip` for our installations.

`pip` comes pre-installed for Python ≥ 2.7 or Python ≥ 3.4 downloaded from Python official site⁵. If the Anaconda distribution has been installed, both `pip` and `conda` are available to manage package installations.

4.3.1 Installing pip

We can install a `pip` via the command line by using the *curl command*, which downloads the `pip` installation *perl* script.

```
curl -O https://bootstrap.pypa.io/get-pip.py
```

Once it is downloaded, we need to execute it in the command prompt with the Python interpreter.

```
python get-pip.py
```

If the above command fails on a Mac and Linux distribution due to permission issues (most likely because Python does not have permission to update certain directories on the file system. These directories are read-only by default to ensure that random scripts cannot mess with important files and infect the system with viruses), we may need to run following command.

```
sudo python get-pip.py
```

4.3.2 Installing Libraries

Now that we have installed `pip`, it is easy to install python modules since it does all the work for us. When we find a module that we want to use,

³<https://conda.io/docs/>

⁴<https://pip.pypa.io/en/stable/installing/>

⁵<https://www.python.org>

usually the documentation or installation instructions will include the necessary pip command.

The Python Package Index⁶ is the main repository for third-party Python packages. The advantage of a library being available on PyPI is the ease of installation using `pip install <package_name>` such as

```
pip install pandas  
pip install numpy  
pip install nsepy
```

Remember, again if the above command fails on a Mac and Linux distribution due to permission issue, we can run the following command:

```
sudo pip install pandas  
sudo pip install nsepy
```

The above examples will install the latest version of the libraries. To install a specific version, we execute the following command:

```
pip install SomeLibrary==1.1
```

To install greater than or equal to one version and less than another:

```
pip install SomeLibrary>=1, < 2
```

Listed below are some of the most popular libraries used in different domains:

- Data Science : NumPy, pandas, SciPy, etc
- Graphics : matplotlib, plotly, seaborn, bokeh, etc.
- Statistics : statsmodels
- Machine learning : SciKit-Learn, Keras, TensorFlow, Theano, etc.
- Web scraping : Scrapy, BeautifulSoup,
- GUI Toolkit : pyGtk, pyQT, wxPython, etc.
- Web Development : Django, web2py, Flask, Pyramid, etc.

We can upgrade already installed libraries to the latest version from PyPI using the following command:

⁶<https://pypi.org/>

```
pip install --upgrade SomeLibrary
```

Remember, pip commands are run directly within the command prompt or shell without the python interpreter. If we are to run pip commands from Jupyter Notebook we need to prefix it with the ! letter like !pip install SomeLibrary.

4.4 Importing modules

Now that we have installed the module that we are interested in, we can start using it right away. First, we need to import the installed module in our code. We do so with the *import* statement. The import statement is the most common way of invoking the module machinery.

4.4.1 import statement

We can import any module, either internal or external into our code using the import statement. Take a look at below example:

```
# Importing an internal module
In []: import math

# Importing an external library
In []: import pandas
```

The above example will import all definitions within the imported library. We can use these definitions using . (dot operator). For example,

```
# Accessing the 'pi' attribute of the 'math' module
In []: math.pi
Out[]: 3.141592653589793

# Accessing the 'floor' function from the 'math' module
In []: math.floor
Out[]: <function math.floor>

# Accessing the 'floor' method from the 'math'
In []: math.floor(10.8)
Out[]: 10
```

```
# Accessing the 'DataFrame' module from the 'pandas'  
# library  
In []: pandas.DataFrame  
Out[]: pandas.core.frame.DataFrame
```

As seen in the above example, we can access attributes and methods of the imported library using the dot operator along with the library name. In fact, the library we import acts as an object and hence, we can call its attributes using the dot notation. We can also alias a library name while importing it with the help of the `as` keyword.

```
# Aliasing math as 'm'  
In []: import math as m  
  
# Aliasing pandas as 'pd'  
In []: import pandas as pd  
  
# Aliasing numpy as 'np'  
In []: import numpy as np
```

An alias can be used in the same way as the module name.

```
In []: m.pi  
Out[]: 3.141592653589793  
  
In []: m.e  
Out[]: 2.718281828459045  
  
In []: m.gamma  
Out[]: <function math.gamma>
```

4.4.2 Selective imports

The other way to import a definition/module is to import all *definitions* in that particular module or all modules in the particular package. We can do so by using `from` keyword.

```
# Import all definitions of math module  
In []: from math import *
```

```
# Import all definitions from pyplot module of matplotlib
# library
In []: from matplotlib.pyplot import *

# Accessing definitions directly without module name
In []: pi
Out[]: 3.141592653589793

In []: e
Out[]: 2.718281828459045

In []: floor(10.8)
Out[]: 10
```

Remember, when definitions are directly imported from any module, we need not use the module name along with the dot operator to access them. Instead, we can directly use the definition name itself. Similarly, it is also possible to import the specific definition that we intend to use, instead of importing all definitions. Consider a scenario where we need to floor down a value, we can import the `floor()` definition from the `math` module, rather than importing every other unnecessary definition. Such import is called *selective import*.

```
# Selective import
# Import only floor from math module
In []: from math import floor

In []: floor(10.8)
Out[]: 10

# Error line as the ceil is not imported from math module
In []: ceil(10.2)
Traceback (most recent call last):

File "<ipython-input-33>", line 1, in <module>
    ceil(10.2)

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

```
# Error line as the math is not imported
# Only the floor from math is imported
In []: math.ceil(10.2)
Traceback (most recent call last):

File "<ipython-input-34>", line 1, in <module>
    math.ceil(10.2)

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

In the above example, we selectively import the `floor` from the `math` module. If we try to access any other definition from the `math` module, Python will return an error stating *definition* not defined as the interpreter won't be able to find any such definition in the code.

4.4.3 The Module Search Path

Let's say we have a module called `vwap_module.py` which is inside the folder `strategy`. We also have a script called `backtesting.py` in a directory called `backtest`.

We want to be able to import the code in `vwap_module.py` to use in `backtesting.py`. We do so by writing `import vwap_module` in `backtesting.py`. The content might look like this:

```
# Content of strategy/vwap_module.py
def run_strategy():
    print('Running strategy logic')

# Content of backtest/backtesting.py
import vwap_module

vwap_module.run_strategy()
```

The moment the Python interpreter encounters line `import vwap_module`, it will generate the following error:

```
Traceback (most recent call last):
```

```
File "backtest/backtesting.py", line 1, in <module>
    import vwap_module

ModuleNotFoundError: No module named 'vwap_module'
```

When Python hits the line `import vwap_module`, it tries to find a package or a module called `vwap_module`. A module is a file with a matching extension, such as `.py`. Here, Python is looking for a file `vwap_module.py` in the same directory where `backtesting.py` exists, and not finding it.

Python has a simple algorithm for finding a module with a given name, such as `vwap_module`. It looks for a file called `vwap_module.py` in the directories listed in the variable `sys.path`.

```
In []: import sys

In []: type(sys.path)
Out[]: list

In []: for path in sys.path:
...:     print(path)
...:

C:\Users\...\Continuum\anaconda3\python36.zip
C:\Users\...\Continuum\anaconda3\DLLs
C:\Users\...\Continuum\anaconda3\lib
C:\Users\...\Continuum\anaconda3
C:\Users\...\Continuum\anaconda3\lib\site-packages
C:\Users\...\Continuum\anaconda3\lib\site-packages\win32
C:\Users\...\Continuum\anaconda3\lib\site-packages\win32\lib
C:\Users\...\Continuum\anaconda3\lib\site-packages\Pythonwin
C:\Users\...\ipython
```

In the above code snippet, we print paths present in the `sys.path`. The `vwap_strategy.py` file is in the `strategy` directory, and this directory is not in the `sys.path` list.

Because `sys.path` is just a Python list, we can make the `import` statement work by appending the `strategy` directory to the list.

```
In []: import sys  
In []: sys.path.append('strategy')  
  
# Now the import statement will work  
In []: import vwap_strategy
```

There are various ways of making sure a directory is always on the `sys.path` list when you run Python. Some of them are

- Keep the directory into the contents of the `PYTHONPATH` environment variable.
- Make the module part of an installable package, and install it.

As a crude hack, we can keep the module in the same directory as the code file.

4.5 `dir()` function

We can use the built-in function `dir()` to find which names a module defines. It returns a sorted list of strings.

```
In []: import arithmetic  
  
In []: dir(arithmetic)  
Out[]:  
['__builtins__',  
 '__cached__',  
 '__doc__',  
 '__file__',  
 '__loader__',  
 '__name__',  
 '__package__',  
 '__spec__',  
 'addition',  
 'division',  
 'factorial',  
 'multiply']
```

Here, we can see a sorted list of names within the module `arithmetic`. All other names that begin with an underscore are default Python attributes associated with the module (we did not define them.)

Without arguments, `dir()` lists the names we have defined currently:

```
In []: a = 1

In []: b = 'string'

In []: import arithmetic

In []: dir()
Out[]:
['__builtins__',
 'a',
 'arithmetic',
 'b',
 'exit',
 'quit']
```

Note that it lists all types of names: variables, modules, functions, etc. The `dir()` does not list the names of built-in functions and variables. They are defined in the standard module `builtins`. We can list them by passing `builtins` as an argument in the `dir()`.

```
In []: import builtins

In []: dir(builtins)
Out[]: ['ArithmetError', 'AssertionError',
        'AttributeError', 'BaseException',
        'BlockingIOError', 'BrokenPipeError',
        'BufferError', 'BytesWarning', 'ChildProcessError',
        'ConnectionAbortedError', 'ConnectionError',
        'ConnectionRefusedError', 'ConnectionResetError',
        'DeprecationWarning', 'EOFError', 'Ellipsis',
        'EnvironmentError', 'Exception', 'False',
        'SyntaxError', ... ]
```

4.6 Key Takeaways

1. A module is a Python file which can be referenced and used in other Python code.
2. A single module can also have multiple Python files grouped together.
3. A collection of modules are known as packages or libraries. The words library and package are used interchangeably.
4. Python comes with a large set of built-in libraries known as the Python Standard Library.
5. Modules in Python Standard Library provides access to core system functionality and solutions for many problems that occur in everyday programming.
6. The `sys` library is present in every Python installation irrespective of the distribution and underlying architecture and it acts as an intermediary between the system and Python.
7. In addition to built-in libraries, additional third-party/external libraries can be installed using either the `pip` or `conda` package managers.
8. The `pip` command comes pre-installed for Python version ≥ 2.7 or Python ≥ 3.4 .
9. A library (either built-in or external) needs to be imported into Python code before it can be used. It can be achieved using `import library_name` keyword.
10. It is a good idea to alias the library name that we import using an `as` keyword.
11. It is always a good programming practice to selectively import only those modules which are required, instead of importing the whole library.
12. Python will look out for the library being imported in the module search path. If the library is not available in any of the paths listed by module search path, Python will throw an error.
13. The `dir()` function is used to list all attributes and methods of an object. If a library name is passed as an argument to the `dir()`, it returns sub-modules and functions of the library.

Chapter 5

Data Structures

In this section we will learn about various built-in data structures such as tuples, lists, dictionaries, and sets. Like a variable, data structures are also used to store a value. Unlike a variable, they don't just store a value, rather a collection of values in various formats. Broadly data structures are divided into *array*, *list* and *file*. Arrays can be considered a basic form of data structure while files are more advanced to store complex data.

5.1 Indexing and Slicing

Before we dive into the world of data structures, let us have a look at the concept of *indexing* and *slicing* which is applicable to all data structures in Python. A string can be thought of as a sequence of characters. Similarly, data structures store sequences of objects (floats, integers, strings, etc.).

Consider a sequence of 10 characters ranging from A to J where we assign a unique position to each literal in a sequence. The position assigned to each character is a sequence of integers beginning with 0 up to the last character. These increase successively by 1 as can be seen below.

Index	0	1	2	3	4	5	6	7	8	9
Sequence	A	B	C	D	E	F	G	H	I	J

In the above sequence, the character A is at index 0, B at 1, C at 2, and so on. Notice how the index increases in chronological order by one unit at

each step. Whenever a new character is appended to this sequence, it will be appended at the end, and will be assigned the next index value (in the above example, the new index will be 10 for the new character). Almost all data structures in Python have an index to position and locate the element.

Elements within the sequence can be accessed using the square brackets []. It takes `index` of an element and returns the element itself. The syntax for accessing a single element is as follows:

```
sequence[i]
```

The above statement will return the element from sequence at index `i`. We can access multiple elements from the sequence using the syntax `[start index : end index]` in the following manner:

```
sequence[si : ei]
```

The above statement will return values starting at index `si` up to but NOT including the element at index `ei`. This operation is referred to as *slicing*. For example:

`sequence[0:4]` will return elements from 'A' to 'D' and not up to 'E'. Element at the last index in the provided range will not be returned.

Python also supports negative indexing to access elements from the sequence end and it starts with -1 as follows:

Index	0	1	2	3	4	5	6	7	8	9
Sequence	A	B	C	D	E	F	G	H	I	J
Negative Index	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

A sequence can also be sliced using the negative indexing. In order to access the last element, we write

```
sequence[-1]
```

and it will return the element J. Similarly, a range can be provided to access multiple elements.

```
sequence[-5:-1] will return elements from 'F' to 'I'
```

5.2 Array

An array can be thought of as a container that can hold a fixed number of data values of the same type. Though the use of array is less popular in Python as compared to other languages such as C and Java, most other data structures internally make use of arrays to implement their algorithms. An array consists of two components, viz *Element* and *Index*.

- Element: These are the actual data values to be stored in an array.
- Index: Each element in array is positioned at the particular location depicted by an index. Python follows *zero based indexing* which means an index will always start with 0.

We can create an array by using the built-in array module. It can be created as follows:

```
In []: from array import *
In []: arr = array('i', [2, 4, 6, 8])
In []:
Out[]: array('i', [2, 4, 6, 8])
In []:
Out[]: type(arr)
Out[]: array.array
```

In the above example, we import the array method from the array module and then initialize the variable arr with values 2, 4, 6, and 8 within the square brackets. The i represents the data type of values. In this case, it represents integer. Python array documentation¹ provides more information about the various type codes available in the Python.

5.2.1 Visualizing an Array

An array declared above can be represented in the following manner:

¹<https://docs.python.org/3.4/library/array.html>

<i>Index</i>	0	1	2	3
<i>Element</i>	2	4	6	8

From the above illustration, following are the points to be considered.

- Index starts with 0.
- Array length is 4 which means it can store 4 values.
- Array can hold values with single data type only.
- Each element can be accessed via its index.

5.2.2 Accessing Array Element

We use slicing operation to access array elements. Slicing operation is performed using the square brackets `[]`. It takes an index of an element we are interested in. It can be noticed that the index of the first element in the above array is 0. So, in order to access an element at the position 3, we use the notation `arr[2]` to access it.

```
# Here, 2 represents the index of element 6
In []: arr[2]
Out []: 6

In []: arr[0]
Out []: 2
```

5.2.3 Manipulating Arrays

The `array` module provides a wide variety of operations that can be performed based on the requirement. We will learn some of the most frequently used operations.

We use insertion operation to insert one or more data elements into an array. Based on the requirement, an element can be inserted at the beginning, end or any given index using the `insert()` method.

```
# Inserting an element at the beginning
In []: arr.insert(0, 20)
```

```
In []: arr
Out[]: array('i', [20, 2, 4, 6, 8])

# Inserting an element at the index 3
In []: arr.insert(3, 60)

In []: arr
Out[]: array('i', [20, 2, 4, 60, 6, 8])
```

An element can be deleted from an array using the built-in `remove()` method.

```
In []: arr.remove(20)

In []: arr
Out[]: array('i', [2, 4, 60, 6, 8])

In []: arr.remove(60)

In []: arr
Out[]: array('i', [2, 4, 6, 8])
```

We can update an element at the specific index using the assignment operator `=` in the following manner:

```
# Update an element at index 1
In []: arr[0] = 1

In []: arr
Out[]: array('i', [1, 4, 6, 8])

# Update an element at index 3
In []: arr[3] = 7

In []: arr
Out[]: array('i', [1, 4, 6, 7])
```

In addition to the above mentioned operation, the `array` module provides a bunch of other operations that can be carried out on an array such as reverse, pop, append, search, conversion to other types, etc.

Though Python allows us to perform a wide variety of operations on arrays, the built-in array module is rarely used. Instead, in real world programming most programmers prefers to use NumPy arrays provided by the NumPy library.

5.3 Tuples

In Python, tuples are part of the standard library. Like arrays, tuples also hold multiple values within them separated by commas. In addition, it also allows storing values of different types together. Tuples are immutable, and usually, contain a heterogeneous sequence of elements that are accessed via unpacking or indexing.

To create a tuple, we place all elements within brackets (). Unlike arrays, we need not import any module for using tuples.

```
# Creating a tuple 'tup' with elements of the same
# data type
In []: tup = (1, 2, 3)

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

# Verifying the type
In []: type(tup)
Out[]: tuple

# Creating a tuple 'tupl' with elements of different data
# types
In []: tupl = (1, 'a', 2.5)

In []: type(tupl)
Out[]: tuple
```

The tuple `tupl` created above can be visualized in the following manner:

Index	0	1	2
Element	1	'a'	2.5

A tuple can also be created without using the brackets.

```
# Creating a tuple without brackets
In []: tup = 1, 2, 3

In []: type(tup)
Out[]: tuple
```

We can repeat a value multiple times within a tuple as follows:

```
In []: tupl = (1,) * 5 # Note trailing comma

In []: tupl
Out[]: (1, 1, 1, 1, 1)
```

5.3.1 Accessing tuple elements

A slice operation performed using the square brackets [] is used to access tuple elements. We pass the index value within the square brackets to get an element of our interest. Like arrays, tuples also have an index and all elements are associated with the particular index number. Again, the index starts with '0'.

```
# Access an element at index 0
In []: tup[0]
Out[]: 1

# Access an element at index 2
In []: tup[2]
Out[]: 1
```

Python throws an error if we try to access an element that does not exist. In other words, if we use the slice operation with a non-existent index, we will get an error.

```
In []: tup[3]
Traceback (most recent call last):

File "<ipython-input-30>", line 1, in <module>
```

```
tup[3]
```

```
IndexError: tuple index out of range
```

In the above example, we try to access an element with index 3 which does not exist. Hence, Python threw an error stating `index out of range`. The built-in `len()` function is used to check the length of a tuple.

```
In []: len(tup)  
Out[]: 3
```

```
In []: len(tupl)  
Out[]: 5
```

5.3.2 Immutability

In Python, tuple objects are immutable. That is, once they are created, it cannot be modified. If we try to modify a tuple, Python will throw an error.

```
In []: tup[1] = 10  
Traceback (most recent call last):  
  
File "<ipython-input-33>", line 1, in <module>  
    tup[1] = 10  
  
TypeError: 'tuple' object does not support item assignment
```

As expected, the interpreter threw an error depicting the tuple object to be immutable.

5.3.3 Concatenating Tuples

Python allows us to combine two or more tuples or directly concatenate new values to an existing tuple. The concatenation is performed in the following manner:

```
In []: t1 = (1, 2, 3)
```

```
In []: t2 = (4, 5)
```

```
In []: t1 + t2  
Out[]: (1, 2, 3, 4, 5)
```

Tuples can be concatenated using operators *= and +=.

```
In []: t1 = (1, 2, 3)  
  
In []: t1 += 4, 5  
  
In []: t1  
Out[]: (1, 2, 3, 4, 5)
```

5.3.4 Unpacking Tuples

In one of the above example, we encountered the statement `tup = 1, 2, 3` which is in turn an example of *tuple packing*. That is we pack various values together into a single variable `tup`. The reverse operation is also possible:

```
In []: tup  
Out[]: (1, 2, 3)  
  
In []: x, y, z = tup
```

The above statement performs the *unpacking* operation. It will assign the value 1 to the variable `x`, 2 to `y`, and 3 to `z`. This operation requires that there are as many variables on the left hand side of the equal sign as there are elements in the tuple.

5.3.5 Tuple methods

Tuple being one of the simple objects in Python, it is easier to maintain. There are only two methods available for tuple objects:

- `index()` : This method returns the index of the element.

```
In []: tup  
Out[]: (1, 2, 3)
```

```
# Returns the index of value '3'.
In []: tup.index(3)
Out[]: 2
```

- `count()` : This method counts the number of occurrences of a value.

```
In []: tup = (1, 1, 1, 1, 1)
```

```
In []: tup.count(1)
Out[]: 5
```

Some of the reasons why tuples are useful are given below:

- They are faster than lists.
- They protect the data as they are immutable.
- They can be used as keys on dictionaries.

5.4 Lists

A list is a data structure that holds an ordered collection of items i.e. we can store a *sequence* of items in a list. In Python, lists are created by placing all items within square brackets [] separated by comma.

It can have any number of items and they may be of different data types and can be created in the following manner:

```
# Empty list
In []: list_a = []

In []: list_a
Out[]: []

# List with integers
In []: list_b = [1, 2, 3]

In []: list_b
Out[]: [1, 2, 3]
```

```
# List with mixed data types
In []: list_c =[1, 2.5, 'hello']

In []:
Out[]: [1, 2.5, 'hello']
```

A list can also have another list as an item. This is called *nested list*.

```
In []: a_list = [1, 2, 3, ['hello', 'stock'], 4.5]
```

5.4.1 Accessing List Items

Like with any other data structure, slice operator is used to access list items or a range of list items. It can be used in the following manner.

```
In []: stock_list = ['HP', 'GOOG', 'TSLA', 'MSFT', 'AAPL',
                     'AMZN', 'NFLX']
```

```
# Accessing an element at index 2
In []: stock_list[2]
Out[]: 'TSLA'
```

```
# Accessing multiple elements using slicing
In []: stock_list[1:4]
Out[]: ['GOOG', 'TSLA', 'MSFT']
```

```
# Accessing last element using negative index
In []: stock_list[-1]
Out[]: 'NFLX'
```

5.4.2 Updating Lists

Unlike tuples, lists are mutable. That is, we can change the content even after it is created. Again, the slicing operation helps us here

```
In []: stock_list
Out[]: ['HP', 'GOOG', 'TSLA', 'MSFT', 'AAPL', 'AMZN',
        'NFLX']
```

```
# Updating the first element
In []: stock_list[0] = 'NVDA'

# Updating the last three elements
In []: stock_list[-3:] = ['AMD', 'GE', 'BAC']

In []:
Out[]: ['NVDA', 'GOOG', 'TSLA', 'AMD', 'GE', 'BAC']
```

It is also possible to add new elements to an existing list. Essentially a list is an object in Python. Hence, the list class provides various methods to be used upon the list object. There are two methods `append()` and `extend()` which are used to update an existing list.

- `append(element)` method adds a single element to the end of the list. It does not return the new list, just modifies the original list.
- `extend(list2)` method adds the elements in `list2` to the end of the list.

```
In []:
Out[]: ['HP', 'GOOG', 'MSFT']

In []:
Out[]: stock_list.append('AMZN')

In []:
Out[]: ['HP', 'GOOG', 'MSFT', 'AMZN']
```

In the above example, we add new element using the `append()` method. Let's add multiple elements to the list. In Python, whenever we are to add multiple literal to any object, we enclose it within list i.e. using `[]` the square brackets. The output that we expect is the appended list will all the new elements.

```
In []:
Out[]: stock_list.append(['TSLA', 'GE', 'NFLX'])

In []:
Out[]: stock_list
Out[]: ['HP', 'GOOG', 'MSFT', 'AMZN', ['TSLA', 'GE', 'NFLX']]
```

The output we got is not as per our expectation. Python amended the new element as a single element to the `stock_list` instead of appending three different elements. Python provides the `extend()` method to achieve this.

```
In []: stock_list
Out[]: ['HP', 'GOOG', 'MSFT', 'AMZN']

In []: stock_list.extend(['TSLA', 'GE', 'NFLX'])

In []: stock_list
Out[]: ['HP', 'GOOG', 'MSFT', 'AMZN', 'TSLA', 'GE',
        'NFLX']
```

To simplify, the `append()` method is used to add a single element to existing list and it takes a single element as an argument, whereas the `extend()` method is used to add multiple elements to existing list and it takes a list as an argument.

5.4.3 List Manipulation

Lists are one of the most versatile and used data structures in Python. In addition to the above discussed methods, we also have other useful methods at our disposal. Some of them are listed below:

- `insert(index, element)` : Inserts an item at a given position. The first argument is the index of the element before which to insert, so `list.insert(0, element)` inserts at the beginning of the list.

```
# Inserting an element at index position 1.
In []: stock_list.insert(1, 'AAPL')

In []: stock_list
Out[]: ['HP', 'AAPL', 'GOOG', 'MSFT', 'AMZN', 'TSLA', 'GE',
        'NFLX']
```

- `remove(element)` : Removes the first item whose value is `element` provided in an argument. Python will throw an error if there is no such item.

```

# Removing the element 'AAPL'
In []: stock_list.remove('AAPL')

In []: stock_list
Out[]: ['HP', 'GOOG', 'MSFT', 'AMZN', 'TSLA', 'GE',
        'NFLX']

# Again removing the element 'AAPL'.
# This line will throw an error as there is no element
# 'AAPL' the list.
In []: stock_list.remove('AAPL')
Traceback (most recent call last):

File "<ipython-input-73>", line 1, in <module>
    stock_list.remove('AAPL')

ValueError: list.remove(x): x not in list

```

- `pop()` : This function removes and returns the last item in the list. If we provide the `index` as an argument, it removes the item at the given position in the list and returns it. It is optional to provide an argument here.

```

# Without providing index position as an argument. Returns
# and removes the last element in the list.
In []: stock_list.pop()
Out[]: 'NFLX'

In []: stock_list
Out[]: ['HP', 'GOOG', 'MSFT', 'AMZN', 'TSLA', 'GE']

# Providing an index position as an argument. Returns and
# removes the element from the specific location.
In []: stock_list.pop(2)
Out[]: 'MSFT'

In []: stock_list
Out[]: ['HP', 'GOOG', 'AMZN', 'TSLA', 'GE']

```

- `index(element)` : Returns the index of the first item whose value is element provided in an argument. Python will throw an error if there is no such item.

```
In []: stock_list.index('GOOG')
Out[]: 1
```

```
In []: stock_list.index('GE')
Out[]: 4
```

- `count(element)` : Returns the number of times element appears in the list.

```
# Count the element 'GOOG'
In []: stock_list.count('GOOG')
Out[]: 1
```

```
# Appending the same list with 'GOOG'
In []: stock_list.append('GOOG')
```

```
In []: stock_list
Out[]: ['HP', 'GOOG', 'AMZN', 'TSLA', 'GE', 'GOOG']
```

```
# Again, counting the element 'GOOG'
In []: stock_list.count('GOOG')
Out[]: 2
```

- `sort()` : When called, this method returns the sorted list. The sort operation will be in place.

```
# Sorting the list. The same list will be updated.
In []: stock_list.sort()
```

```
In []: stock_list
Out[]: ['AMZN', 'GE', 'GOOG', 'GOOG', 'HP', 'TSLA']
```

- `reverse()` : This method reverses the elements of the list and the operation performed will be in place.

```
# Reversing the elements within the list.  
In []: stock_list.reverse()  
  
In []: stock_list  
Out[]: ['TSLA', 'HP', 'GOOG', 'GOOG', 'GE', 'AMZN']
```

5.4.4 Stacks and Queues

The list methods make it very easy to use a list as a stack or queue. A *stack* is a data structure (though not available directly in Python) where the last element added is the first element retrieved, also known as *Last In, First Out (LIFO)*. A list can be used as a stack using the `append()` and `pop()` method. To add an item to the top of the stack, we use the `append()` and to retrieve an item from the top of the stack, we use the `pop()` without an explicit index. For example:

```
# (Bottom) 1 -> 5 -> 6 (Top)  
In []: stack = [1, 5, 6]  
  
In []: stack.append(4)      # 4 is added on top of 6 (Top)  
  
In []: stack.append(5)      # 5 is added on top of 4 (Top)  
  
In []: stack  
Out[]: [1, 5, 6, 4, 5]  
  
In []: stack.pop()         # 5 is removed from the top  
Out[]: 5  
  
In []: stack.pop()         # 4 is removed from the top  
Out[]: 4  
  
In []: stack.pop()         # 6 is removed from the top  
Out[]: 6  
  
In []: stack      # Remaining elements in the stack  
Out[]: [1, 5]
```

Another data structure that can be built using list methods is *queue*, where

the first element added is the first element retrieved, also known as *First In, First Out (FIFO)*. Consider a queue at a ticket counter where people are catered according to their arrival sequence and hence the first person to arrive is also the first to leave.

In order to implement a queue, we need to use the `collections.deque` module; however, lists are not efficient for this purpose as it involves heavy memory usage to change the position of every element with each insertion and deletion operation.

It can be created using the `append()` and `popleft()` methods. For example,

```
# Import 'deque' module from the 'collections' package
In []: from collections import deque

# Define initial queue
In []: queue = deque(['Perl', 'PHP', 'Go'])

# 'R' arrives and joins the queue
In []: queue.append('R')

# 'Python' arrives and joins the queue
In []: queue.append('Python')

# The first to arrive leaves the queue
In []: queue.popleft()
Out[]: 'Perl'

# The second to arrive leaves the queue
In []: queue.popleft()
Out[]: 'PHP'

# The remaining queue in order of arrival
In []: queue
Out[]: deque(['Go', 'R', 'Python'])
```

5.5 Dictionaries

A Python dictionary is an unordered collection of items. It stores data in *key-value* pairs. A dictionary is like a phone-book where we can find the phone numbers or contact details of a person by knowing only his/her name i.e. we associate names (*keys*) with corresponding details (*values*). Note that the keys must be unique just like the way it is in a phone book i.e. we cannot have two persons with the exact same name.

In a dictionary, pairs of keys and values are specified within curly brackets {} using the following notation:

```
dictionary = {key1 : value1, key2 : value2, key3 : value3}
```

Notice that the key-value pairs are separated by the colon : and pairs themselves are separated by ,. Also, we can use only immutable objects like strings and tuples for the keys of a dictionary. Values of a dictionary can be either mutable or immutable objects. Dictionaries that we create are instances of the dict class and they are unordered, so the order that keys are added doesn't necessarily reflect the same order when they are retrieved back.

5.5.1 Creating and accessing dictionaries

A dictionary can be created either using the curly brackets {} or the method dict(). For example:

```
# Creating an empty dictionary using {}
In []: tickers = {}

In []: type(tickers)
Out[]: dict

# Creating an empty dictionary using the dict() method
In []: tickers = dict()

In []: type(tickers)
Out[]: dict
```

Let us create a dictionary with values of the same data type.

```
In []: tickers = {'GOOG' : 'Alphabet Inc.',
...:             'AAPL' : 'Apple Inc.',
...:             'MSFT' : 'Microsoft Corporation'}
```



```
In []: tickers
Out[]:
{'GOOG': 'Alphabet Inc.',
 'AAPL': 'Apple Inc.',
 'MSFT': 'Microsoft Corporation'}
```

Next, we will create a dictionary with multiple data types.

```
In []: ticker = {'symbol' : 'AAPL',
...:             'price' : 224.95,
...:             'company' : 'Apple Inc',
...:             'founded' : 1976,
...:             'products' : ['Machintosh', 'iPod',
...:                         'iPhone', 'iPad']
...: }
```

We can also provide a dictionary as a value to another dictionary key. Such a dictionary is called *nested dictionary*. Take a look at below example:

```
In []: tickers = {'AAPL' : {'name' : 'Apple Inc.',
...:                         'price' : 224.95
...: },
...:             'GOOG' : {'name' : 'Alphabet Inc.',
...:                         'price' : 1194.64
...: }
...: }
```

Keys in a dictionary should be unique. If we supply the same key for multiple pairs, Python will ignore the previous value associated with the key and only the recent value will be stored. Consider the following example:

```
In []: same_keys = {'symbol' : 'AAPL',
...:                   'symbol' : 'GOOG'}
```



```
In []: same_keys
Out[]: {'symbol': 'GOOG'}
```

In the above example, Python discarded the value AAPL and retained the latest value assigned to the same key. Once we have created dictionaries, we can access them with the help of the respective keys. We use the slice operator [] to access the values; however, we supply a key to obtain its value. With the dictionaries created above, we can access values in the following manner:

```
In []: ticker
Out[]:
{'symbol': 'AAPL',
 'price': 224.95,
 'company': 'Apple Inc',
 'founded': 1976,
 'products': ['Machintosh', 'iPod', 'iPhone', 'iPad']}

In []: tickers
Out[]:
{'AAPL': {'name': 'Apple Inc.', 'price': 224.95},
 'GOOG': {'name': 'Alphabet Inc.', 'price': 1194.64}}

# Accessing the symbol name
In []: ticker['symbol']
Out[]: 'AAPL'

# Accessing the ticker price
In []: ticker['price']
Out[]: 224.95

# Accessing the product list
In []: ticker['products']
Out[]: ['Machintosh', 'iPod', 'iPhone', 'iPad']

# Accessing the item at position 2 in the product list.
In []: ticker['products'][2]
Out[]: 'iPhone'

# Accessing the first nested dictionary from the
# 'tickers' dictionary
In []: tickers['AAPL']
```

```
Out[]: {'name': 'Apple Inc.', 'price': 224.95}

# Accessing the price of 'GOOG' ticker using chaining
# operation
In []: tickers['GOOG']['price']
Out[]: 1194.64
```

5.5.2 Altering dictionaries

A value in a dictionary can be updated by assigning a new value to its corresponding key using the assignment operator =.

```
In []: ticker['price']
Out[]: 224.95

In []: ticker['price'] = 226

In []: ticker['price']
Out[]: 226
```

A new key-value pair can also be added in a similar fashion. To add a new element, we write the new key inside the square brackets [] and assign a new value. For example:

```
In []: ticker['founders'] = ['Steve Jobs', 'Steve Wozniak',
                             'Ronald Wayne']

In []: ticker
Out[]:
{'symbol': 'AAPL',
 'price': 226,
 'company': 'Apple Inc',
 'founded': 1976,
 'products': ['Machintosh', 'iPod', 'iPhone', 'iPad'],
 'founders': ['Steve Jobs', 'Steve Wozniak',
             'Ronald Wayne']}
```

In the above example, we add the key founders and assign the list ['Steve Jobs', 'Steve Wozniak', 'Ronald Wayne'] as value. If we are to delete

any key-value pair in the dictionary, we use the built-in `del()` function as follows:

```
In []: del(ticker['founders'])

In []: ticker
Out[]:
{'symbol': 'AAPL',
 'price': 226,
 'company': 'Apple Inc',
 'founded': 1976,
 'products': ['Machintosh', 'iPod', 'iPhone', 'iPad']}
```

5.5.3 Dictionary Methods

The `dict` class provides various methods using which we can perform a variety of operations. In addition to these methods, we can use built-in `len()` functions to get the length of a dictionary.

```
In []: len(ticker)
Out[]: 5

In []: len(tickers)
Out[]: 2
```

Now we discuss some of the popular methods provided by the `dict` class.

- `items()` : This method returns a object containing all times in the calling object.

```
In []: ticker.items()
Out[]: dict_items([('symbol', 'AAPL'), ('price', 226),
                  ('company', 'Apple Inc'),
                  ('founded', 1976),
                  ('products', ['Machintosh', 'iPod',
                               'iPhone', 'iPad'])])
```

- `keys()` : This method returns all keys in the calling dictionary.

```
In []: ticker.keys()  
Out[]: dict_keys(['symbol', 'price', 'company', 'founded',  
                  'products'])
```

- `values()` : This method returns all values in the calling object.

```
In []: ticker.values()  
Out[]: dict_values(['AAPL', 224.95, 'Apple Inc', 1976,  
                     ['Machintosh', 'iPod', 'iPhone',  
                      'iPad']])
```

- `pop()` : This method pops the item whose key is given as an argument.

```
In []: tickers  
Out[]:  
{'GOOG': 'Alphabet Inc.',  
 'AAPL': 'Apple Inc.',  
 'MSFT': 'Microsoft Corporation'}
```

```
In []: tickers.pop('GOOG')  
Out[]: 'Alphabet Inc.'
```

```
In []: tickers  
Out[]: {'AAPL': 'Apple Inc.',  
        'MSFT': 'Microsoft Corporation'}
```

- `copy()` : As the name suggests, this method copies the calling dictionary to another dictionary.

```
In []: aapl = ticker.copy()
```

```
In []: aapl  
Out[]:  
{'symbol': 'AAPL',  
 'price': 224.95,  
 'company': 'Apple Inc',  
 'founded': 1976,  
 'products': ['Machintosh', 'iPod', 'iPhone', 'iPad']}
```

- `clear()` : This method empties the calling dictionary.

```
In []: ticker.clear()
```

```
In []: ticker  
Out[]: {}
```

- `update()` : This method allows to add new key-pair value from another dictionary.

```
In []: ticker1 = {'NFLX' : 'Netflix'}
```

```
In []: ticker2 = {'AMZN' : 'Amazon'}
```

```
In []: new_tickers = {}
```

```
In []: new_tickers.update(ticker1)
```

```
In []: new_tickers.update(ticker2)
```

```
In []: new_tickers  
Out[]: {'NFLX': 'Netflix', 'AMZN': 'Amazon'}
```

5.6 Sets

A set is an unordered and unindexed collection of items. It is a collection data type which is mutable, iterable and contains no duplicate values. A set in Python represents the mathematical notion of a set.

In Python sets are written using the curly brackets in the following way:

```
In []: universe ={'GOOG', 'AAPL', 'NFLX', 'GE'}
```

```
In []: universe  
Out[]: {'AAPL', 'GE', 'GOOG', 'NFLX'}
```

We cannot access items in a set by referring to an index (slicing operation), since sets are unordered the item has no index. But we can loop through all items using the `for` loop which will be discussed in the upcoming section. Once a set is created, we cannot change its items, but we can add new items using the `add()` method.

```
In []: universe.add('AMZN')

In []: universe
Out[]: {'AAPL', 'AMZN', 'GE', 'GOOG', 'NFLX'}
```

Python won't add the same item again nor will it throw any error.

```
In []: universe.add('AMZN')

In []: universe.add('GOOG')

In []: universe
Out[]: {'AAPL', 'AMZN', 'GE', 'GOOG', 'NFLX'}
```

In order to add multiple items, we use the `update()` method with new items to be added within a list.

```
In []: universe.update(['FB', 'TSLA'])

In []: universe
Out[]: {'AAPL', 'AMZN', 'FB', 'GE', 'GOOG', 'NFLX', 'TSLA'}
```

We can use the inbuilt `len()` function to determine the length of a set.

```
In []: len(universe)
Out[]: 7
```

To remove or delete an item, we can use the `remove()` or `discard()` methods. For example,

```
In []: universe.remove('FB')

In []: universe.discard('TSLA')

In []: universe
Out[]: {'AAPL', 'AMZN', 'GE', 'GOOG', 'NFLX'}
```

If we try to remove an item using the `remove()` which is not present in the set, Python will throw an error.

```
In []: universe.remove('FB')
Traceback (most recent call last):

File "<ipython-input-159>", line 1, in <module>
    universe.remove('FB')

KeyError: 'FB'
```

The `discard()` method will not throw any error if we try to discard an item which is not present in the set.

```
In []: universe
Out[]: {'AAPL', 'AMZN', 'GE', 'GOOG', 'NFLX'}
```



```
In []: universe.discard('FB')
```

We use the `clear()` method to empty the set.

```
In []: universe.clear()
```



```
In []: universe
Out[]: set()
```

Following the mathematical notation, we can perform set operations such as union, intersection, difference, etc. in Python using the set. Consider the following examples:

- We define two sets `tech_stocks` and `fin_stocks` as follows:

```
In []: tech_stocks = {'AMD', 'GOOG', 'AAPL', 'WDC'}
```



```
In []: fin_stocks = {'BAC', 'BMO', 'JPLS'}
```

- `union()` method: This method allows performing a union between sets. This operation returns all elements within both sets.

```
# Performs the 'union` operation
In []: universe = tech_stocks.union(fin_stocks)
```



```
# 'universe' contains all elements of both sets
```

```
In []: universe
Out[]: {'AAPL', 'AMD', 'BAC', 'BMO', 'GOOG', 'JPLS', 'WDC'}
```

- `intersection()` method: This method performs the intersection between sets. It returns only elements which are available in both sets.

```
# Only elements present in the 'universe' set and
# and 'fin_stocks' are returned
In []: universe.intersection(fin_stocks)
Out[]: {'BAC', 'BMO', 'JPLS'}
```

- `difference()` method: This method performs the difference operation and returns a set containing all elements of the calling object but not including elements of the second set.

```
# All elements of the 'universe' set is returned except
# elements of the 'fin_stock'
In []: universe.difference(fin_stocks)
Out[]: {'AAPL', 'AMD', 'GOOG', 'WDC'}
```

- `issubset()` method: This method checks whether all elements of calling set is present within a second set or not. It returns true if the calling set is subset of the second set, false otherwise.

```
# True, as the 'universe' contains all elements of
# the 'fin_stocks'
In []: fin_stocks.issubset(universe)
Out[]: True
```

```
# Can you guess why it resulted in to False?
In []: universe.issubset(tech_stocks)
Out[]: False
```

- `isdisjoint()` method: This method checks for the intersection between two sets. It returns true if the calling set is disjoint and not intersected with the second set, false otherwise.

```
# True, none of the set contains any element of each other
In []: fin_stocks.isdisjoint(tech_stocks)
```

```
Out[]: True

# False, the 'universe' set contains elements of
# the 'fin_stocks' set
In []: fin_stocks.isdisjoint(universe)
Out[]: False
```

- `issuperset()` method: This method checks whether the calling set contains all elements of the second set. It returns true, if the calling set contains all elements of the second set, false otherwise.

```
# True, the 'universe' set contains all elements of
# the 'fin_stocks'
In []: universe.issuperset(fin_stocks)
Out[]: True
```

```
# True, the 'universe' set contains all elements of
# the 'tech_stocks'
In []: universe.issuperset(tech_stocks)
Out[]: True
```

```
# False, the 'fin_stocks' set does not contain all
# elements of the 'universe' set
In []: fin_stocks.issuperset(universe)
Out[]: False
```

5.7 Key Takeaways

1. Data structures are used to store a collection of values.
2. Arrays, tuples, lists, sets and dictionaries are primitive data structures in Python. Except for dictionaries, all data structures are sequential in nature.
3. In Python, indexing starts with 0 and negative indexing starts with -1.
4. Elements within data structures are accessed using the slicing operation `[start_index:end_index]` where `start_index` is inclusive and `end_index` is exclusive.
5. An array can hold a fixed number of data values of the same type. Arrays are not built-in data structures. We can use an array library to

perform basic array operations.

6. A tuple can hold multiple values of different types within it separated by commas. Tuples are enclosed within parentheses () and they are immutable.
7. A list holds an ordered collection of items. Lists are created by placing all items within square brackets [] separated by a comma. They can also be used to implement other data structures like stacks and queues.
8. A list can also have another list as an item. This is called a nested list.
9. Dictionary stores data in the form of a key-value pair. It can be created using the curly brackets {}. Element/Pair within the dictionary is accessed using the corresponding keys instead of an index.
10. Sets are an unordered data structure created using the curly brackets {}. It cannot contain duplicate elements.

Chapter 6

Keywords & Operators

Python is a high-level language that allows us to nearly write programs in a natural language like English. However, there are certain words and symbols used internally by Python which carry a definite unambiguous meaning. They can be used only in certain pre-defined ways when we program. We will explore such words known as keywords and various operators in this section.

6.1 Python Keywords

Keywords are reserved (plainspeak for 'set aside' or 'on hold') words in Python. They are built into the core language. They cannot be used as a variable name, class name, function name or any other identifier. These keywords are used to define the syntax and semantics of the Python.

Since Python is case-sensitive, so are the keywords. All keywords are in lowercase except `True`, `False` and `None`. In this section, we will learn various keywords.

- The `and` keyword is a logical operator used to combine conditional statements and it returns `True` if both statements are `True`, `False` otherwise.

```
In []: (5 > 3) and (4 > 2)
Out[]: True
```

- The `as` keyword is used to create an alias. Consider the following example where we create an alias for the `calendar` module as `c` while importing it. Once aliased we can refer to the imported module with its alias.

```
In []: import calendar as c
```

```
In []: c.isleap(2019)
```

```
Out[]: False
```

- The `assert` keyword is used while debugging code. It allows us to check if a condition in code returns `True`, if not Python will raise an `AssertionError`. If condition returns `True`, no output is displayed.

```
In []: stock = 'GOOG'
```

```
In []: assert stock == 'GOOG'
```

```
In []: assert stock == 'AAPL'
```

```
Traceback (most recent call last):
```

```
File "<ipython-input-6>", line 1, in <module>
    assert stock == 'AAPL'
```

```
AssertionError
```

- The `break` keyword is used to break a `for` loop and `while` loop.

```
# A for loop that will print from 0 to 9
# Loop will break when 'i' will be greater than 3
```

```
In []: for i in range(10):
....:     if i > 3:
....:         break
....:     print(i)
```

```
# Output
```

```
0
```

```
1
```

```
2
```

```
3
```

- The `class` keyword is used to create a class.

```
In []: class stock():
....      name = 'AAPL'
....      price = 224.61
....
```

```
In []: s1 = stock()
```

```
In []: s1.name
Out[]: 'AAPL'
```

```
In []: s1.price
Out[]: 224.61
```

- The `continue` keyword is used to end the current iteration in a `for` loop or `while` loop, and continues to the next iteration.

```
# A for loop to print 0 to 9
# When `i` will be 4, iteration will continue without
# checking further
```

```
In []: for i in range(9):
....      if i == 4:
....          continue
....      else:
....          print(i)
```

```
# Output: It does not contain 4 as loop continued with
# the next iteration
```

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

- The `def` keyword is used to create/define a function within Python.

```
In []: def python_function():
...:     print('Hello! This is a Python Function.')
```

```
In []: python_function()
Out[]: Hello! This is a Python Function.
```

- The `del` keyword is used to delete objects. It can also be used to delete variables, lists, or elements from data structures, etc.

```
# Define the variable 'a'
```

```
In []: a = 'Hello'
```

```
# Delete the variable 'a'
```

```
In []: del(a)
```

```
# Print the variable 'a'. Python will throw an error as the
# variable 'a' does not exist.
```

```
In []: a
```

```
Traceback (most recent call last):
```

```
File "<ipython-input-17>", line 1, in <module>
      a
```

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

- The `if` keyword is used as a logical test between boolean expression(s). If the boolean expression evaluates to `True`, the code following the 'if' condition will be executed.
- The `elif` keyword is used to define multiple if conditions. It is also referred to as 'else if'. If the expression evaluates to `True`, the code following the condition will be executed.
- The `else` keyword is used to define the code block that will be executed when all if conditions above it fail. It does not check for any condition, it just executes the code if all the conditions above it fail.

```
In []: number = 5
```

```
...:
```

```
...: if number < 5:
```

```
....:     print('Number is less than 5')
....: elif number == 5:
....:     print('Number is equal to 5')
....: else:
....:     print('Number is greater than 5')
....:

# Output
Number is equal to 5
```

- The `try` keyword is used to define the `try...except` code block which will be followed by the code block defined by `except` keyword. Python tries to execute the code within `try` block and if it executes successfully, it ignores subsequent blocks.
- The `except` keyword is used in `try...except` blocks for handling any error raised by the `try` block. It is used to define a code block that will be executed if the `try` block fails to execute and raises any error.

```
# Python will throw an error here because the variable 'y'
# is not defined
In []: try:
....:     y > 5
....: except:
....:     print('Something went wrong.')
....:

# Output
Something went wrong.
```

- The `finally` keyword is used to `try...except` block to define a block of code that will run no matter if the `try` block raises an error or not. This can be useful to close objects and clean up resources.

```
# Code in the 'finally' block will execute whether or not
# the code in the 'try' block raises an error
In []: try:
....:     y > 5
....: except:
```

```
....:     print('Something went wrong.')
....: finally:
....:     print('The try...except code is finished')
....:
```

Output

```
Something went wrong.
The try...except code is finished
```

- The `False` keyword is used to represent the boolean result false. It evaluates to 0 when it is cast to an integer value.

```
In []: buy_flag = False
```

```
In []: int(False)
Out[]: 0
```

- The `True` keyword is used to represent the boolean result true. It evaluates to 1 when cast to an integer value.

```
In []: buy_flag = True
```

```
In []: int(True)
Out[]: 1
```

- The `for` keyword is used to define/create a for loop.

```
In []: for i in range(5):
....:     print(i)
....:
....:
```

Output

```
0
1
2
3
4
```

- The `import` keyword is used to import external libraries and modules in to current program code.

```
In []: import pandas
```

```
In []: import numpy
```

- The `from` keyword is used while importing modules and libraries in Python. It is used to import specific modules from the libraries.

```
# Importing DataFrame module from the pandas library
```

```
In []: from pandas import DataFrame
```

```
# Importing time module from the datetime package
```

```
In []: from datetime import time
```

```
# Importing floor function from the math module
```

```
In []: from math import floor
```

- The `global` keyword is used to declare a global variables to be used from non-global scope.

```
In []: ticker = 'GOOG'  
....:  
....: def stocks():  
....:     global ticker  
....:     # Redeclaring or assigning new value 'MSFT' to  
....:     # global variable 'ticker'  
....:     ticker = 'MSFT'  
....:  
....:  
....: stocks()  
....:  
....: print(ticker)
```

```
# Output
```

```
MSFT
```

- The `in` keyword is used to check if a value is present in a sequence. It is also used to iterate through a sequence in a `for` loop.

```
In []: stock_list = ['GOOG', 'MSFT', 'NFLX', 'TSLA']
```

```
In []: 'GOOG' in stock_list
Out[]: True
```

```
In []: 'AMZN' in stock_list
Out[]: False
```

- The `is` keyword is used to test if two variables refers to the same object in Python. It returns true if two variables are same objects, false otherwise.

```
In []: stock_list = ['GOOG', 'MSFT', 'NFLX', 'TSLA']
```

```
In []: y = stock_list
```

```
# Checks whether the 'stock_list' and 'y' are same or not
In []: stock_list is y
Out[]: True
```

```
In []: y is stock_list
Out[]: True
```

```
# Reversing elements in the 'stock_list' also reverses
# elements in the 'y' as both are same
In []: stock_list.reverse()
```

```
In []: y
Out[]: ['TSLA', 'NFLX', 'MSFT', 'GOOG']
```

- The `lambda` keyword is used to create a small anonymous function in Python. It can take multiple arguments but accepts only a single expression.

```
# Creates an anonymous function that adds two values
# provided by 'x' and 'y'
In []: addition = lambda x, y : x + y
```

```
In []: addition(5, 2)
Out[]: 7
```

- The `None` keyword is used to define a null value, or no value at all. It is not same as 0, `False`, or an empty string. `None` is represented by a datatype of `NoneType`.

```
In []: x = None
```

```
In []: type(x)
```

```
Out[]: NoneType
```

- The `nonlocal` keyword is used to declare a variable that is not local. It is used to work with variables inside nested functions, where the variable should not belong to the inner function.

```
# Instead of creating a new variable 'x' within the
# 'nested_function' block, it will use the variable 'x'
# defined in the 'main function'
```

```
In []: def main_function():
....:     x = "MSFT"
....:     def nested_function():
....:         nonlocal x
....:         x = "GOOG"
....:         nested_function()
....:     return x
....:
....:
....: print(main_function())
```

```
# Output
```

```
GOOG
```

- The `not` keyword is a logical operator similar to the `and` operator. It returns the boolean value `True` if an expression is not true, `False` otherwise.

```
In []: buy = False
```

```
In []: not buy
```

```
Out[]: True
```

- The `or` keyword is a logical operator used to check multiple conditional statements and it returns `True` if at least one statement is `True`, `False` otherwise.

```
In []: (5 > 3) or (4 < 2)
Out[]: True
```

- The `pass` keyword is used as a placeholder for a null statement. It does nothing when used. If we have empty function definition, Python will return an error. Hence, it can be used as a placeholder in an empty function.

```
In []: def empty_function():
    ...
    ...
    ...
In []: empty_function()

File "<ipython-input-49>", line 5
    empty_function()
    ^
IndentationError: expected an indented block
```

```
In []: def empty_function():
    ...:     pass
    ...:
    ...:
    ...: empty_function()
```

- The `raise` keyword is used to raise an error explicitly in a code.

```
In []: x = 'Python'

In []: if not type(x) is int:
    ...:     raise TypeError('Only integers are allowed')
    ...:
Traceback (most recent call last):

File "<ipython-input-52>", line 2, in <module>
```

```
    raise TypeError('Only integers are allowed')
```

```
TypeError: Only integers are allowed
```

- The `return` keyword is used to return a value from a function or method.

```
In []: def addition(a, b):  
....:     return a + b  
....:
```

```
In []: addition(2, 3)  
Out[]: 5
```

- The `with` keyword is used to wrap the execution of a block with methods defined by a context manager¹. It simplifies exception handling by encapsulating common preparation and cleanup tasks. For example, the `open()` function is a context manager in itself, which allows opening a file, keeping it open as long as the execution is in context of the `with`, and closing it as soon as we leave the context. So simply put, some resources are acquired by the `with` statement and released when we leave the `with` context.

```
# Open the file 'abc.txt' using the 'with' keyword in the  
# append mode
```

```
In []: with open('abc.txt', 'a') as file:  
....:     # Append the file  
....:     file.write('Hello Python')  
....:  
....:
```

```
# We do not need to close the file as it will be called  
# automatically as soon as we leave the 'with' block
```

```
# Open the file in the read mode
```

```
In []: with open('abc.txt', 'r') as file:  
....:     print(file.readline())  
....:
```

¹<https://docs.python.org/3/reference/datamodel.html#context-managers>

```
....  
Out []: Hello Python
```

6.2 Operators

Operators are constructs or special symbols which can manipulate or compute the values of operands in an expression. In other words, they are used to perform operations on variables and values. Python provides a bunch of different operators to perform a variety of operations. They are broadly categorized into the following:

6.2.1 Arithmetic operators

Arithmetic operators are used with numerical values to perform the common mathematical operations.

- + : This is an addition operator used to perform the addition between values.

```
In []: 5 + 3  
Out []: 8
```

- - : This is a subtraction operator used to perform the subtraction between operands.

```
In []: 5 - 2  
Out []: 3
```

- * : This is a multiplication operator used to multiply the operands.

```
In []: 5 * 2  
Out []: 10
```

- / : This is a division operator which performs the division operation and returns a float output.

```
In []: 10 / 2  
Out []: 5.0
```

- % : This is a modulus operator. It returns the remainder of the division operation.

```
In []: 16 % 5  
Out[]: 1
```

- ** : This operator is used to perform the exponentiation operation, sometimes referred to as the *raised to power* operation. This essentially performs the operation of raising one quantity to the power of another quantity.

```
In []: 2 ** 3  
Out[]: 8
```

```
In []: 3 ** 2  
Out[]: 9
```

- // : This operator is used to perform the floor division operation and it returns the integer output.

```
# Floor division operation  
In []: 10 // 4  
Out[]: 2
```

```
# Normal division operation  
In []: 10 / 4  
Out[]: 2.5
```

6.2.2 Comparison operators

Comparison operators are used to compare two or more values. It works with almost all data types in Python and returns either True or False. We define the below variables to understand these operators better.

```
In []: a = 5
```

```
In []: b = 3
```

```
In []: x = 5
```

```
In []: y = 8
```

- `==` : This is an *equal to* operator used to check whether two values are equal or not. It returns true if values are equal, false otherwise.

```
In []: a == x
Out[]: True
```

```
In []: a == b
Out[]: False
```

- `!=` : This is a *not equal to* operator and works exactly opposite to the above discussed *equal to* operator. It returns True if values are not equal, and false otherwise.

```
In []: a != x
Out[]: False
```

```
In []: a != b
Out[]: True
```

- `>` : This is a *greater than* operator used to check whether one value is greater than another value. It returns true if the first value is greater compared to the latter, false otherwise.

```
In []: y > x
Out[]: True
```

```
In []: b > y
Out[]: False
```

- `<` : This is a *less than* operator used to check whether one value is less than another value. It returns true if the first value is less compared to the latter, false otherwise.

```
In []: y < x
Out[]: False
```

```
In []: b < y
Out[]: True
```

- `>=` : This is a *greater than or equal to* operator used to check whether one value is greater than or equal to another value or not. It returns true if the first value is either greater than or equal to the latter value, false otherwise.

```
In []: a >= x  
Out[]: True
```

```
In []: y >= a  
Out[]: True
```

```
In []: b >= x  
Out[]: False
```

- `<=` : This is a *less than or equal to* operator used to check whether one value is less than or equal to another value or not. It returns true if the first value is either less than or equal to the latter value, false otherwise.

```
In []: a <= x  
Out[]: True
```

```
In []: y <= a  
Out[]: False
```

```
In []: b <= x  
Out[]: True
```

6.2.3 Logical operators

Logical operators are used to compare two or more conditional statements or expressions, and returns boolean result.

- `and` : This operator compares multiple conditional statements and returns true if all statements results in true, and false if any statement is false.

```
In []: 5 == 5 and 3 < 5  
Out[]: True
```

```
In []: 8 >= 8 and 5 < 5
Out[]: False
```

```
In []: 5 > 3 and 8 == 8 and 3 <= 5
Out[]: True
```

- **or** : This operator compares multiple conditional statements and returns true if at least one of the statements is true, and false if all statements are false.

```
In []: 5 == 5 or 3 > 5
Out[]: True
```

```
In []: 3 <= 3 or 5 < 3 or 8 < 5
Out[]: True
```

```
In []: 3 < 3 or 5 < 3 or 8 < 5
Out[]: False
```

- **not** : This operator reverses the result. It returns true if the result is false, and vice versa.

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

```
In []: not 3 == 3
Out[]: False
```

```
In []: 3 != 3
Out[]: False
```

```
In []: not 3 != 3
Out[]: True
```

6.2.4 Bitwise operator

Bitwise operators are used to compare and perform logical operations on binary numbers. Essentially operations are performed on each bit of a binary

number instead of a number. Binary numbers are represented by a combination of 0 and 1. For better understanding, we define following numbers (integers) and their corresponding binary numbers.

Number	Binary
201	1100 1001
15	0000 1111

In the above example, both 201 and 15 are represented by 8 bits. Bitwise operators work on multi-bit values, but conceptually one bit at a time. In other words, these operator works on 0 and 1 representation of underlying numbers.

- `&` : This is a bitwise *AND* operator that returns 1 only if *both* of its inputs are 1, 0 otherwise. Below is the truth table for the `&` operator with four bits.

Bits	1	2	3	4
<i>Input 1</i>	0	0	1	1
<i>Input 2</i>	0	1	0	1
<i>& Output</i>	0	0	0	1

We can compute the bitwise `&` operation between 201 and 15 as follows:

```
In [] : 201 & 15
Out [] : 9
```

Let us understand with the help of a truth table, how Python returned the value 9.

Binary Numbers		
<i>Input 1</i>	201	1100 1001
<i>Input 2</i>	15	0000 1111
<i>& Output</i>	9	0000 1001

Python evaluated `&` operation based on each bit of inputs and re-

turned an integer equivalent of the binary output. In the above example, decimal equivalent of 0000 1001 is 9.

- | : This is a bitwise *OR* operator that returns 1 if *any* of its inputs are 1, 0 otherwise. Below is the truth table for the | operator with four bits.

Bits	1	2	3	4
<i>Input 1</i>	0	0	1	1
<i>Input 2</i>	0	1	0	1
<i>Output</i>	0	1	1	1

The bitwise | operation between 201 and 15 can be performed in the following way:

```
In [] : 201 | 15  
Out [] : 207
```

The above operation can be verified via the truth table as shown below:

Binary Numbers		
<i>Input 1</i>	201	1100 1001
<i>Input 2</i>	15	0000 1111
<i>Output</i>	207	1100 1111

In the above example, Python evaluated | operation bitwise and returned the output 1100 1111 which is the binary equivalent of 207.

- ^ : This is a bitwise *XOR* operator that returns 1 only if *any one* of its input is 1, 0 otherwise. Below is the truth table for the XOR operation.

Bits	1	2	3	4
<i>Input 1</i>	0	0	1	1
<i>Input 2</i>	0	1	0	1
<i>Output</i>	0	1	1	0

Notice that it does not return 1 if all inputs are 1. The bitwise ^ can be

performed as follows:

```
In [] : 201 ^ 15
Out [] : 198
```

The output returned by the Python can be verified via its corresponding truth table as shown below.

Binary Numbers		
<i>Input 1</i>	201	1100 1001
<i>Input 2</i>	15	0000 1111
<i>^ Output</i>	207	1100 0110

In the above example, Python performed the XOR operation between its input and returns the result as 1100 0110 which is the decimal equivalent of 207.

- `~` : This is a bitwise NOT operator. It is an unary operator that takes only one input and inverts all the bits of an input, and returns the inverted bits. Consider the following truth table

Bits	1 2
<i>Input</i>	0 1
<i>Output</i>	1 0

- `<<` : This is a bitwise left shift operator. It takes two inputs: *number to operate on* and *number of bits to shift*. It shifts bits to the left by pushing zeros in from the right and let the leftmost bits fall off. Consider the following example:

```
In [] : 15 << 2
Out [] : 60
```

In the above example, we are shifting the number 15 left by 2 bits. The first input refers to the number to operate on and the second input refers to the number of bits of shift. We compute the truth table for the above operation as below:

Binary		
<i>Input</i>	15	0000 1111
<< <i>Output</i>	60	0011 1100

- `>>` : Similar to the left shift operator, we have a shift right operator that shifts bits right and fills zero on the left. While shifting bits to right, it let the rightmost bits fall off and add new zeros to the left.

```
In [] : 201 >> 2
Out [] : 50
```

In the above example, the number 201 gets shifted right by 2 bits and we get 50 as an output. Its integrity can be verified by the following truth table.

Binary Numbers		
<i>Input</i>	201	1100 1001
>> <i>Output</i>	50	0011 0010

Bitwise operators find great significance in the quantitative trading domain. They are used to determine the trading signals based on different conditions. Generally, we assign 1 to the *buy signal*, -1 to the *sell signal* and 0 to the *no signal*. If we have multiple buy conditions and need to check if all conditions are satisfied, we use bitwise & operator to determine if all buy conditions are 1 and buy the asset under consideration. We will look at these things in more detail when we discuss *pandas* and *numpy* libraries.

6.2.5 Assignment operators

As the name suggests, assignment operators are used to assign values to variables.

- `=` : This operator assigns the value on its right side to the operand on its left.

```
In [] : a = 5
In [] : b = 3
```

We can also use this operator to assign multiple values to multiple operands on the left side. Number of values and operands must be same on both sides, else Python will throw an error.

```
In []: a, b = 5, 3

# Error line. Number of operands on both sides should
# be same.
In []: a, b = 5, 3, 8
Traceback (most recent call last):

File "<ipython-input-1-file>", line 1, in <module>
      a, b = 5, 3, 8

ValueError: too many values to unpack (expected 2)
```

- **+=**: This operator adds the operand on the right side with the operand on the left side and assigns the result back to the same operand on the left side.

```
In []: a += 2

In []: print(a)
Out[]: 7

# The above operation is same as the one mentioned below
In []: a = a + 2
```

- **-=**: This operator subtracts the operand on the right side with the operand on the left side and assigns the result back to the same operand on the left side.

```
In []: a -= 2

In []: print(a)
Out[]: 5
```

- ***=**: This operator multiplies the operand on the right side with the operand on the left side and assigns the result back to the same operand on the left side.

```
In []: a *= 2
```

```
In []: print(a)  
Out[]: 10
```

- `/=`: This operator divides the operand on the left side by the operand on the right side and assigns the result back to the same operand on the left side.

```
In []: a /= 3
```

```
In []: print(a)  
Out[]: 3.333333333333335
```

- `%=`: This operator performs the division operation between operands and assigns the remainder to the operand on the left.

```
In []: a = 10
```

```
In []: a %= 3
```

```
In []: print(a)  
Out[]: 1
```

- `**=`: This operator performs the exponential operation between operands and assigns the result to the operand on the left.

```
In []: a **= 3
```

```
In []: print(a)  
Out[]: 8
```

- `//=`: This operator divides the left operand with the right operand and then assigns the result (floored to immediate integer) to the operand on the left.

```
In []: a = 10
```

```
In []: a //= 4
```

```
In []: print(a)  
Out[]: 2
```

- `&=` : This operator performs the bitwise 'AND' operation between the operands and then assigns the result to the operand on the left side.

```
In []: a = 0
In []: a &= 1

# & operation results into 0 as one operand is 0 and
# the other is 1.
In []: print(a)
Out[]: 0
```

- `|=` : This operator performs the bitwise 'OR' operation between the operands and then assigns the result to the operand on the left side.

```
In []: a = 0
In []: a |= 1

# | operation results into 1
In []: print(a)
Out[]: 1
```

- `^=` : This operator performs the bitwise 'XOR' operation between the operands and then assigns the result to the operand on the left side.

```
In []: a = 1
In []: a ^= 1

# ^ operation results into 0 as both operands will be 1
In []: print(a)
Out[]: 0
```

- `>>=` : This operator shifts bits of the left operand to the right specified by the right operand and then assigns the new value to the operand on the left side.

```
In []: a = 32
In []: a >>= 2

In []: print(a)
Out[]: 8
```

- `<=>` : This operator shifts bits of the left operand to the left specified by the left operand and then assigns the new value to the operand on the left side.

```
In []: a = 8
In []: a <=> 2

In []: print(a)
Out[]: 32
```

6.2.6 Membership operators

These operators are used check whether the value/variable exist in a sequence or not.

- `in` : This operator returns True if a value exists in a sequence, False otherwise.

```
In []: stock_list = ['GOOG', 'MSFT', 'AMZN', 'NFLX']
```

```
In []: 'GOOG' in stock_list
Out[]: True
```

```
In []: 'AAPL' in stock_list
Out[]: False
```

- `not in` : This operator returns True if a value *does not* exists in a sequence, False otherwise.

```
In []: 'AAPL' not in stock_list
Out[]: True
```

```
In []: 'GOOG' not in stock_list
Out[]: False
```

6.2.7 Identity operators

These operators are used to check if two values or objects belong to same memory location or refer to same instance in Python. They can be used in the following way:

- `is` : This operator returns True if both operands are identical, False otherwise.

```
In []: a = 3
In []: b = a

# True as both variables refers to same value in
# the memory
In []: a is b
Out[]: True

In []: x = 3

# True as Python will create new reference of variable 'x'
# to value 3 on the same memory location
In []: x is a
Out[]: True
```

- `is not` : This operator returns True if both operands are *not* on same memory location, False otherwise.

```
In []: stock_list = ['AMZN', 'NFLX']
In []: my_list = ['AMZN', 'NFLX']

In []: stock_list is my_list
Out[]: False

# Though both lists are identical, they will not be stored
# at the same memory location as lists are mutable.
In []: stock_list is not my_list
Out[]: True
```

6.2.8 Operator Precedence

In Python, expressions are evaluated from left to right order. That is, if there are multiple operators within an expression, Python will evaluate its value starting from left most operand and ultimately up to the right most operator. Consider the following example:

```
In []: 2 + 5 - 3 + 1  
Out []: 5
```

In the above example, Python will first evaluate `2 + 5` resulting into 7, then subtracts 3 from it to get 4, and finally adding 1 to obtain the final result of 5. But this is not the case always. If we include more operators, Python will behave in the different manner. For example,

```
In []: 2 + 5 * 3  
Out []: 17
```

Based on the principle discussed above (left to right evaluation), Python should evaluate the above expression to 21, but instead, it returned 17. Here, the Python first evaluated `5 * 3` resulting into 15 and then added 2 to obtain the final value of 17, because the operator `*` has higher precedence over the `+`.

If there are multiple operators in an expression, Python will execute operators at same precedence from the left to right order starting from operators having the highest precedence. Following table lists the operators from highest precedence to lowest.

Operators	Precedence
<code>()</code>	Parentheses
<code>**</code>	Exponential
<code>+, -, ~</code>	Positive, Negative, Bitwise NOT
<code>*, /, //, %</code>	Multiplication, Division, Floor Division, Modulus
<code>+, -</code>	Addition, Subtraction
<code><<, >></code>	Bitwise Left, Bitwise Right
<code>&</code>	Bitwise AND
<code>^</code>	Bitwise XOR
<code> </code>	Bitwise OR
<code>==, !=, >, >=, <, <=, is, is not, in, not in</code>	Comparison, Identity, Membership Operators
<code>not</code>	Logical NOT
<code>and</code>	Logical AND
<code>or</code>	Logical OR

As the above table lists the () with the highest precedence, it can be used to change the precedence of any operator to be highest. Any expression written inside the parentheses () gets highest precedence and evaluated first.

```
In []: (5 / 2) * (2 + 5)
Out[]: 17.5
```

In the above example, the order of evaluation will be (5 / 2) resulting into 2.5 followed by (2 + 5) evaluating to 7 and finally 2.5 multiplied with 7 with the multiplication * operator providing the output as 17.5.

6.3 Key Takeaways

1. Keywords are reserved words in Python. They can be used only in certain predefined ways when we code. They are always available and cannot be used as a variable name, class name, function name or any other identifier.
2. Keywords are case-sensitive. All keywords are in lowercase except True, False and None.
3. In Python, logical expressions are evaluated from left to right.
4. Operators are special constructs or symbols which are used to perform operations on variables and literals.
5. Arithmetic operators are used for performing various mathematical operations.
6. Comparison operators are used for comparing two or more values. It works with almost all data types and returns either True or False as an output.
7. Logical operators are used for comparing two or more conditional statements or expressions. They return boolean True or False as an output.
8. Bitwise operators act on bits and perform bit by bit operations. These operators perform operations on a binary (0 and 1) equivalent of a decimal number.
9. Assignment operators are used for assigning values to variables.
10. Membership operators check whether the given value exists in a data structure or not.

11. Identity operators check if two objects belong to the same memory location or refer to the same instances or not.
12. Each operator has specific precedence in Python. The precedence of any expression can be changed using the parenthesis () .

Chapter 7

Control Flow Statements

The code we write gets executed in the order they are written. In other words, a program's *control flow* is the order in which the program's code executes. Using conditional statements such as `if` statements, and loops, we can define or alter the execution order of the code. This section covers a conditional `if` statement and `for` and `while` loops; functions are covered in the upcoming section. Raising and handling exceptions also affects the control flow which will be discussed in subsequent sections.

7.1 Conditional Statements

Often times it is required that a code should execute only if a condition holds true, or depending on several mutually exclusive conditions. Python allows us to implement such a scenario using an `if` statement.

7.1.1 The `if` statement

The `if` statement is used when we want a code to be executed under certain conditions only. It can be either a single condition or multiple conditions. The code within the `if` block will be executed if and only if the logical conditions are held true. We use a comparison operator to check the truthfulness of the condition. The `if` statement evaluates the output of any logical condition to be either `True` or `False`, and the codes within the `if` block gets executed if the condition is evaluated true.

Let us consider a scenario where we want to go long on a stock if `buy_condition` is True.

```
# Buy a stock when the buy condition is true
if buy_condition == True:
    position = 'Buy'
```

In Python, we use the colon : to mark the end of the statement and start of the block. This is true for any statement such as a class or function definition, conditional statements, loops, etc. Notice the : at the end of the if statement which marks the start of the if block. The code is indented inside the block to depict that it belongs to a particular block. To end the block, we just write the code without any indentation.

In the above statement, the code statement `position = 'Buy'` is said to be inside the if block. If the variable or logical condition itself tends to be boolean i.e. True or False, we can directly write the condition without comparing. We can re-write the above example as shown below:

```
# Here, buy_condition itself is a boolean variable
if buy_condition:
    position = 'Buy'
```

In another scenario we want to buy a stock when an indicator is below 20, we can use the if statement as depicted in the following example:

```
# Buy a stock when an indicator (RSI value) is less than
# or equal to 20
if rsi_indicator <= 20:
    position = 'Buy'
```

Additionally, two or more conditions can be combined using any logical operator such as and, or, etc. In a more refined scenario, we might want to go long on a stock only if two conditions are true. We can do so in the following way:

```
# Input
if buy_condition_1 == True and rsi_indicator <= 20:
    position = 'Buy'
```

Similar to the above scenario, we can compound the if condition to be as complex as we want it to be, using different combinations of logical operators.

7.1.2 The elif clause

The elif clause checks for new conditions and executes the code if they are held true after the conditions evaluated by the previous if statement weren't true. In a scenario with mutually exclusive conditions, we might want the code to execute when one set of condition/s fails and another holds true. Consider the following example:

```
# Input
if buy_condition_1 == True and rsi_indicator <= 20:
    position = 'Buy'
elif sell_condition_1 and rsi_indicator >= 80:
    position = 'Sell'
```

During the execution, the interpreter will first check whether the conditions listed by the if statement holds true or not. If they are true, the code within the if block will be executed. Otherwise, the interpreter will try to check the conditions listed by the elif statement and if they are true, the code within the elif block will be executed. And if they are false, the interpreter will execute the code following the elif block. It is also possible to have multiple elif blocks, and the interpreter will keep on checking the conditions listed by each elif clause and executes the code block wherever conditions will be held true.

7.1.3 The else clause

The else clause can be thought of as the last part of conditional statements. It does not evaluate any conditions. When defined it just executes the code within its block if all conditions defined by the if and elif statements are false. We can use the else clause in the following way.

```
# Input
if buy_condition_1 == True and rsi_indicator <= 20:
    position = 'Buy'
elif sell_condition_1 and rsi_indicator >= 80:
```

```
    position = 'Sell'  
else:  
    position = 'None'
```

In the above example, if the conditions listed by the `if` and `elif` clauses are false, the code within the `else` block gets executed and the variable `position` will be assigned a value '`None`'.

7.2 Loops

Let us consider a scenario where we want to compare the value of the variable `rsi_indicator` multiple times. To address this situation, we need to update the variable each time manually and check it with the `if` statement. We keep repeating this until we check all the values that we are interested in. Another approach we can use is to write multiple `if` conditions for checking multiple values. The first approach is botched and cumbersome, whereas the latter is practically non-feasible.

The approach we are left with is to have a range of values that need to be logically compared, check each value and keep iterating over them. Python allows us to implement such approach using loops or more precisely the `while` and `for` statements.

7.2.1 The `while` statement

The `while` statement in Python is used to repeat execution of code or block of code that is controlled by a conditional expression. The syntax for a `while` loop is given below:

```
while conditional expression:  
    code statement 1  
    code statement 2  
    ...  
    code statement n
```

A `while` statement allows us to repeat code execution until a conditional expression becomes true. Consider the following `while` loop:

```
# Input
data_points = 6
count = 0

while count != data_points:
    print(count)
    count += 1
```

The `while` statement is an example of what is called a *looping* statement. The above loop will print 6 digits starting from 0 up to 5 and the output will be the following:

```
# Output
0
1
2
3
4
5
```

When the above code is run, the interpreter will first check the conditional expression laid by the `while` loop. If the expression is `false` and the condition is *not* met, it will enter the loop and executes the code statements within the loop. The interpreter will keep executing the code within the loop until the condition becomes true. Once the condition is true, the interpreter will stop executing the code within the loop and move to the next code statement. A `while` statement can have an optional `else` clause. Continuing the above example, we can add the `else` clause as shown in the below example:

```
# Input
data_points = 6
count = 0

while count != data_points:
    print(count)
    count += 1
else:
    print('The while loop is over.')
```

In the above example, the interpreter will execute the while loop as we discussed above. Additionally, when the condition becomes true, the interpreter will execute the `else` clause also and the output will be as follows:

```
# Output
0
1
2
3
4
5
The while loop is over.
```

7.2.2 The `for` statement

The `for` statement in Python is another looping technique which *iterates* over a sequence of objects. That is, it will go through each item in a sequence. A sequence may be either a list, tuple, dictionary, set or string. The syntax of a `for` loop is as follows:

```
for item in sequence:
    code statement 1
    code statement 2
    ...
    code statement n
```

The `for` statement is also known as `for..in` loop in Python. The `item` in the above syntax is the placeholder for each item in the sequence.

The `for` loop in Python is different as compared to other programming languages. We shall now see some of its avatars below.

7.2.3 The `range()` function

Python provides the built-in function `range()` that is used to generate sequences or arithmetic progressions which in turn can be combined with the `for` loop to repeat the code.

The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number. The syntax of `range()` is as follows:

```
range([start,] stop [, step])
```

Parameter Values:-

`start` : Optional. An integer specifying at which number to start. The default is 0.
`stop` : Required. An integer specifying at which number to end.
`step` : Optional. An integer specifying the incrementation. The default is 1.

The `range()` function can be used along with the `for` loop as follows:

```
# Input
for i in range(5):
    print(i)
```

Here, we have provided only `stop` parameter value as 5. Hence, the `range()` function will start with 0 and end at 5 providing us with a sequence of 5 numbers. The output of the above `for` loop will be the following:

```
# Output
0
1
2
3
4
```

In the above `for` loop, the variable `i` will take the value of 0 generated by the `range()` function for the first iteration and execute the code block following it. For the second iteration, the variable `i` will take the value of 1 and again execute the code block following it and such repetition will continue until the last value is yielded by the `range()` function.

It is also possible to use various combinations of the `start`, `stop` and `step` parameters in a `range()` function to generate any sort of sequence. Consider the following example:

```
# Input
for i in range(1, 10, 2):
    print(i)
```

The above `range()` function will generate the sequence starting from 1 up to 10 with an increment of 2 and the output will be the following:

```
# Output
1
3
5
7
9
```

7.2.4 Looping through lists

With the `for` loop we can execute a set of code, once for each item in a list. The `for` loop will execute the code for all elements in a list.

```
# Input
top_gainers = ['BHARTIARTL', 'EICHERMOT', 'HCLTECH',
                'BAJFINANCE', 'RELIANCE']

for gainer in top_gainers:
    print(str(top_gainers.index(gainer)) + ' : ' + gainer)
```

Here the `for` loop will iterate over the list `top_gainers` and it will print each item within it along with their corresponding index number. The output of the above `for` loop is shown below:

```
# Output
0 : BHARTIARTL
1 : EICHERMOT
2 : HCLTECH
3 : BAJFINANCE
4 : RELIANCE
```

7.2.5 Looping through strings

Strings in Python are iterable objects. In other words, strings are a sequence of characters. Hence, we can use a string as a sequence object in the `for` loop.

```
volume_name = 'Python'

for character in volume_name:
    print(character)
```

We initialize the string `volume_name` with the value 'Python' and provide it as an iterable object to the `for` loop. The `for` loop yields each character from the it and prints the respective character using the `print` statement. The output is shown below:

```
# Output
P
y
t
h
o
n
```

7.2.6 Looping through dictionaries

Another sequential data structure available at our disposal is *dictionary*. We learnt about dictionaries in detail in the previous section. Looping through dictionaries involves a different approach as compared to lists and strings. As dictionaries are not index based, we need to use its built-in `items()` method as depicted below:

```
dict = {'AAPL':193.53,
        'HP':24.16,
        'MSFT':108.29,
        'GOOG':1061.49}

for key, value in dict.items():
    print(f'Price of {key} is {value}')
```

If we execute the command `dict.items()` directly, Python will return us a collection of a dictionary items (in form of tuples). as shown below:

```
# Input
dict.items()

# Output
dict_items([('AAPL', 193.53), ('HP', 24.16),
            ('MSFT', 108.29), ('GOOG', 1061.49)])
```

As we are iterating over tuples, we need to fetch a key and value for each item in the `for` loop. We fetch the key and value of each item yielded by the `dict.items()` method in the `key` and `value` variables and the output is shown below:

```
# Output
Price of AAPL is 193.53
Price of HP is 24.16
Price of MSFT is 108.29
Price of GOOG is 1061.49
```

In Python version 2.x, we need to use the method `iteritems()` of the dictionary object to iterate over its items.

A `for` loop can also have an optional `else` statement which gets executed once the `for` loop completes iterating over all items in a sequence. Sample `for` loop with an optional `else` statement is shown below:

```
for item in range(1, 6):
    print(f'This is {item}.')
else:
    print('For loop is over!')
```

The above `for` loop prints five statements and once it completes iterating over the `range()` function, it will execute the `else` clause and the output will be the following:

```
# Output
This is 1.
```

```
This is 2.  
This is 3.  
This is 4.  
This is 5.  
For loop is over!
```

7.2.7 Nested loops

Often times it is required that we need to loop through multiple sequences simultaneously. Python allows the usage of one loop inside another loop. Consider a scenario where we want to generate multiplication tables of 1 up to 9 simultaneously. We can do so by using nested `for` loops as given below:

```
for table_value in range(1, 10):  
    for multiplier in range(1, 11):  
        answer = table_value * multiplier  
        print(answer, end=' ')  
    print()  
else:  
    print('For loop is over!')
```

The first `for` loop defines the range for table from 1 to 9. Similarly, the second or the inner `for` loop defines the multiplier value from 1 to 10. The `print()` in the inner loop has parameter `end=' '` which appends a space instead of default new line. Hence, answers for a particular table will appear in a single row. The output for the above nested loops is shown below:

```
# Output  
1 2 3 4 5 6 7 8 9 10  
2 4 6 8 10 12 14 16 18 20  
3 6 9 12 15 18 21 24 27 30  
4 8 12 16 20 24 28 32 36 40  
5 10 15 20 25 30 35 40 45 50  
6 12 18 24 30 36 42 48 54 60  
7 14 21 28 35 42 49 56 63 70  
8 16 24 32 40 48 56 64 72 80  
9 18 27 36 45 54 63 72 81 90  
For loop is over!
```

The same scenario can also be implemented using the `while` nested loops as given below and we will get the same output shown above.

```
# Input
table_value = 1

while table_value != 10:
    multiplier = 1
    while multiplier != 11:
        answer = table_value * multiplier
        print(answer, end=' ')
        multiplier += 1
    table_value += 1
    print()
else:
    print('While loop is over!')
```

In Python it is also possible to nest different loops together. That is, we can nest a `for` loop inside a `while` loop and vice versa.

7.3 Loop control statements

Python provides various ways to alter the code execution flow during the execution of loops. Three keywords to do so are `break`, `pass` and `continue`. Though we already got a glimpse of these keywords in the previous section, we will learn its usage in this section. These keywords can be used with any looping technique in Python. Here, we will learn its implementation using a `for` loop.

7.3.1 The `break` keyword

The `break` keyword is used to break the execution flow of a loop. When used inside a loop, this keyword stops executing the loop and the execution control shifts to the first statement outside the loop. For example, we can use this keyword to break the execution flow upon certain condition.

```
# Input
for item in range(1,10):
```

```
print(f'This is {item}.')
if item == 6:
    print('Exiting FOR loop.')
    break
print('Not in FOR loop.')
```

We define a `for` loop that iterates over a range of 1 to 9 in the above example. Python will try to execute the code block following the loop definition, where it will check if the item under consideration is 6. If true, the interpreter will break and exit the loop as soon as it encounters the `break` statement and starts executing the statement following the loop. The output of the above loop will be the following:

```
# Output
This is 1.
This is 2.
This is 3.
This is 4.
This is 5.
This is 6.
Exiting FOR loop.
Not in FOR loop.
```

7.3.2 The `continue` keyword

Similar to the `break` keyword discussed above, we have the `continue` keyword which will skip the current iteration and continue with the next iteration. Consider the below example:

```
# Input
for item in range(1,10):
    if item == 6:
        continue
    print('This statement will not be executed.')
    print(f'This is {item}.')
print('Not in FOR loop.')
```

Again, we define a `for` loop to iterate over a range of 1 to 9 and check whether the item under consideration is 6 or not? If it is, we skip that iteration and continues the loop. Python interpreter will not attempt to execute

any statement once it encounters the `continue` keyword. The output of the above `for` loop is shown below:

```
# Output
This is 1.
This is 2.
This is 3.
This is 4.
This is 5.
This is 6.
This is 7.
This is 8.
This is 9.
Not in FOR loop.
```

As seen in the output above, the interpreter didn't print anything once it encountered the `continue` keyword thereby skipping the iteration.

7.3.3 The pass keyword

Essentially the `pass` keyword is not used to alter the execution flow, but rather it is used merely as a placeholder. It is a null statement. The only difference between a comment and a `pass` statement in Python is that the interpreter will entirely ignore the comment whereas a `pass` statement is not ignored. However, nothing happens when `pass` is executed.

In Python, loops cannot have an empty body. Suppose we have a loop that is not implemented yet, but we want to implement it in the future, we can use the `pass` statement to construct a body that does nothing.

```
# Input
stocks = ['AAPL', 'HP', 'MSFT', 'GOOG']

for stock in stocks:
    pass
else:
    print('For loop is over!')
```

In the loop defined above, Python will just iterate over each item without producing any output and finally execute the `else` clause. The output will be as shown below:

```
# Output  
For loop is over!
```

7.4 List comprehensions

List comprehension is an elegant way to define and create a list in Python. It is used to create a new list from another sequence, just like a mathematical set notation in a single line. Consider the following set notation:

```
{i^3: i is a natural number less than 10}
```

The output of the above set notation will be cubes of all natural numbers less than 10. Now let's look at the corresponding Python code implementing list comprehension.

```
[i**3 for i in range(0,10)]
```

As we see in the Python code above, list comprehension starts and ends with square brackets to help us remember that the output will be a list. If we look closely, it is a `for` loop embedded in the square bracket. In a general sense, a `for` loop works as follows:

```
for item in sequence:  
    if condition:  
        output expression
```

The same gets implemented in a simple list comprehension construct in a single line as:

```
[output expression for item in sequence if condition]
```

As shown above, the syntax for *list comprehension* starts with the opening square bracket [followed by `output expression`, `for` loop, and optional `if` condition. It has to be ended with the closing square bracket].

The set defined above can also be implemented using the `for` loop in the following way:

```
# Input
cube_list = []

for i in range(0,10):
    cube_list.append(i**3)
```

The corresponding list comprehension is constructed in the following way:

```
# Input
[i**3 for i in range(0,10)]
```

The output for the `for` loop and the list comprehension defined above will be the same shown below:

```
# Output
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

We can filter the output produced by a list comprehension using the condition part in its construct. Consider the revised set notation given below:

```
{i^3: i is a whole number less than 20, i is even}
```

The set defined above contains cubes of all whole numbers which are less than 20 and even. It can be implemented using the `for` loop as given below:

```
# Input
cube_list = []

for i in range(1,20):
    if i%2==0:
        cube_list.append(i**3)

print(cube_list)

# Output
[8, 64, 216, 512, 1000, 1728, 2744, 4096, 5832]
```

The output we got is in line with the set defined above and the `for` loop defined above can be implemented in a single line using the LC construct.

```
# Input  
[i**3 for i in range(1,20) if i%2==0]
```

With a list comprehension, it does not have to be a single condition. We can have multiple conditions to filter the output produced by it. Suppose, we want to have a list of all positive numbers less than 20 which are divisible by 2 and 3 both. Such a list can be generated as follows:

```
# Input  
[i for i in range(0,20) if i%2==0 if i%3==0]  
  
#Output  
[0, 6, 12, 18]
```

Python provides a flexible way to integrate `if` conditions within a list comprehension. It also allows us to embed the `if...else` condition. Let us segregate a list of positive numbers into Odd and Even using a comprehension construct.

```
# Input  
[str(i)+': Even' if i%2==0 else str(i)+': Odd' for i in  
range(0,6)]
```

In such a scenario, we need to put the `if` and `else` part of a condition before the `for` loop in the comprehension. The output of the above construct is as below:

```
# Output  
['0: Even', '1: Odd', '2: Even', '3: Odd', '4: Even',  
'5: Odd']
```

Finally, we can use a list comprehension to write a nested `for` loop. We resort to normal `for` loop and implement the multiplication table of 7 using the nested `for` loops in the following example:

```
# Input  
for i in range(7,8):  
    for j in range(1,11):  
        print(f'{i} * {j} = {i * j}')
```

```
# Output  
7 * 1 = 7  
7 * 2 = 14  
7 * 3 = 21  
7 * 4 = 28  
7 * 5 = 35  
7 * 6 = 42  
7 * 7 = 49  
7 * 8 = 56  
7 * 9 = 63  
7 * 10 = 70
```

Such nested `for` loops can be implemented using a comprehension in the following way:

```
# Input  
[i * j for j in range(1,11) for i in range(7,8)]
```

Here, the output will only be the result of multiplication as shown below:

```
# Output  
[7, 14, 21, 28, 35, 42, 49, 56, 63, 70]
```

Is it possible to produce the exact output using comprehension as generated by nested `for` loops above? The answer is *yes*. Go ahead and give it a try. This brings us to an end of this section. Here we took a deep dive into a conditional statement and various looping techniques in Python. We learned about ways to implement `if` conditions within a `for` and `while` loop and explored list comprehensions and their applications.

7.5 Key Takeaways

1. Control flow statements are Python constructs that alter the flow of the execution.
2. The conditional `if` statement is used when code needs to be executed based on some condition.
3. The `if` statement can evaluate multiple conditions.

4. The `elif` statement can be used in case of multiple mutually exclusive conditions.
5. The `else` statement can be used when code needs to be executed if all previous conditions fail.
6. Loops are used to perform an iterative process. In other words, loops are used to execute the same code more than one time.
7. A loop can be implemented using: a `while` statement and a `for` statement.
8. A counter needs to be coded explicitly for a `while` loop, else it might run infinitely.
9. The `range()` function is used to generate sequences in Python.
10. A loop within a loop is known as a nested loop.
11. A `for` loop is used to iterate over data structures such as lists, tuples, dictionaries and string as well.
12. The `break` keyword is used to break the execution of a loop and directs the execution flow outside the loop.
13. The `continue` keyword is used to skip the current iteration of a loop and moves the execution flow to the next iteration.
14. In Python, loops cannot have an empty body.
15. The `pass` keyword is used as a placeholder in an empty loop.
16. A list comprehension returns list. It consists of square brackets containing an expression that gets executed for each element in the iteration over a loop.

Chapter 8

Iterators & Generators

In this section we will explore the natural world of iterators, objects that we have already encountered in the context of `for` loops without necessarily knowing it, followed by its easier implementation via a handy concept of generators. Let's begin.

8.1 Iterators

Iterators are everywhere in Python. They are elegantly implemented in `for` loop, comprehensions, etc. but they are simply hidden in plain sight. An iterator is an object that can be iterated upon and which will return data, one element at a time. It allows us to traverse through all elements of a collection, regardless of its specific implementation.

Technically, in Python, an iterator is an object which implements the iterator protocol, which in turn consists of the methods `__next__()` and `__iter__()`.

8.1.1 Iterables

An iterable is an object, not necessarily a data structure that can return an iterator. Its primary purpose is to return all of its elements. An object is known as iterable if we can get an iterator from it. Directly or indirectly it will define two methods:

- `__iter__()` method which returns the iterator object itself and is used while using the `for` and `in` keywords.
- `__next__()` method returns the next value. It also returns `StopIteration` error once all the objects have been traversed.

The Python Standard Library contains many iterables: lists, tuples, strings, dictionaries and even files and we can run a loop over them. It essentially means we have indirectly used the iterator in the previous section while implementing looping techniques.

All these objects have an `iter()` method which is used to get an iterator. Below code snippet returns an iterator from a tuple, and prints each value:

```
In []: stocks = ('AAPL', 'MSFT', 'AMZN')
In []: iterator = iter(stocks)

In []: next(iterator)
Out[]: 'AAPL'

In []: next(iterator)
Out[]: 'MSFT'

In []: iterator.__next__()
Out[]: 'AMZN'

In []: next(iterator)
Traceback (most recent call last):

File "<ipython-input-6>", line 1, in <module>
    next(iterator)

StopIteration
```

We use the `next()` function to iterate manually through all the items of an iterator. Also, the `next()` function will implicitly call the `__next__()` method of an iterator as seen in the above example. It will raise `StopIteration` error once we reach the end and there is no more data to be returned.

We can iterate manually through other iterables like strings and list, in the manner similar to one we used to iterate over the tuple int the above example. The more elegant and automated way is to use a `for` loop. The `for` loop actually creates an iterator object and executes the `next()` method for each loop.

We are now going to dive a bit deeper into the world of iterators and iterables by looking at some handy functions viz. the `enumerate()`, `zip()` and `unzip()` functions.

8.1.2 `enumerate()` function

The `enumerate()` function takes any iterable such as a list as an argument and returns a special `enumerate` object which consists of pairs containing an element of an original iterable along with their index within the iterable. We can use the `list()` function to convert the `enumerate` object into a list of tuples. Let's see this in practice.

```
In []: stocks = ['AAPL', 'MSFT', 'TSLA']

In []: en_object = enumerate(stocks)

In []: en_object
Out[]: <enumerate at 0x7833948>

In []: list(en_object)
Out[]: [(0, 'AAPL'), (1, 'MSFT'), (2, 'TSLA')]
```

The `enumerate` object itself is also iterable, and we can loop over while unpacking its elements using the following clause.

```
In []: for index, value in enumerate(stocks):
....:     print(index, value)

0 AAPL
1 MSFT
2 TSLA
```

It is the default behaviour to start an index with 0. We can alter this behaviour using the `start` parameter within the `enumerate()` function.

```
In []: for index, value in enumerate(stocks, start=10):
...:     print(index, value)

10 AAPL
11 MSFT
12 TSLA
```

Next, we have the `zip()` function.

8.1.3 The `zip()` function

The `zip()` function accepts an arbitrary number of iterables and returns a `zip` object which is an iterator of tuples. Consider the following example:

```
In []: company_names = ['Apple', 'Microsoft', 'Tesla']
In []: tickers = ['AAPL', 'MSFT', 'TSLA']

In []: z = zip(company_names, tickers)

In []: print(type(z))
<class 'zip'>
```

Here, we have two lists `company_names` and `tickers`. Zipping them together creates a `zip` object which can be then converted to list and looped over.

```
In []: z_list = list(z)

In []: z_list
Out[]: [('Apple', 'AAPL'), ('Microsoft', 'MSFT'),
         ('Tesla', 'TSLA')]
```

The first element of the `z_list` is a tuple which contains the first element of each list that was zipped. The second element in each tuple contains the corresponding element of each list that was zipped and so on. Alternatively, we could use a `for()` loop to iterate over a `zip` object print the tuples.

```
In []: for company, ticker in z_list:
...:     print(f'{ticker} = {company}')
```

```
AAPL = Apple
MSFT = Microsoft
TSLA = Tesla
```

We could also have used the splat operator(*) to print all the elements.

```
In []: print(*z)
('Apple', 'AAPL') ('Microsoft', 'MSFT') ('Tesla', 'TSLA')
```

8.1.4 Creating a custom iterator

Let's see how an iterator works internally to produce the next element in a sequence when asked for. Python iterator objects are required to support two methods while following the iterator protocol. They are `__iter__()` and `__next__()`. The custom iterator coded below returns a series of numbers:

```
class Counter(object):
    def __init__(self, start, end):
        """Initialize the object"""
        self.current = start
        self.end = end

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

    def __next__(self):
        """Returns the next element in the series"""
        if self.current > self.end:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1
```

We created a Counter class which takes two arguments start (depicts the start of a counter) and end (the end of the counter). The `__init__()` method is a constructor method which initializes the object with the start and end

parameters received. The `__iter__()` method returns the iterator object and the `__next__()` method computes the next element within the series and returns it. Now we can use the above-defined iterator in our code as shown below:

```
# Creates a new instance of the class 'Counter' and
# initializes it with start and end values
counter = Counter(1, 5)

# Run a loop over the newly created object and print its
# values
for element in counter:
    print(element)
```

The output of the above `for` loop will be as follows:

```
# Output
1
2
3
4
5
```

Remember that an iterator object can be used only once. It means once we have traversed through all elements of an iterator, and it has raised `StopIteration`, it will keep raising the same exception. So, if we run the above `for` loop again, Python will not provide us with any output. Internally it will keep raising the `StopIteration` error. This can be verified using the `next()` method.

```
In []: next(counter)
Traceback (most recent call last):

File "<ipython-input-18>", line 21, in <module>
    next(counter)

File "<ipython-input-12>", line 11, in __next__
    raise StopIteration

StopIteration
```

8.2 Generators

Python generator gives us an easier way to create iterators. But before we make an attempt to learn what *generators* in Python are, let us recall the list comprehension we learned in the previous section. To create a list of the first 10 even digits, we can use the comprehension as shown below:

```
In []: [number*2 for number in range(10)]  
Out[]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Now, if we replace the square brackets [] in the above list comprehension with the round parenthesis (), Python returns something called generator objects.

```
In []: (number*2 for number in range(10))  
Out[]: <generator object <genexpr> at 0x00000000CE4E780>
```

But what are actually generator objects? Well, a generator object is like list comprehension except it does not store the list in memory; it does not construct the list but is an object we can iterate over to produce elements of the list as required. For example:

```
In []: numbers = (number for number in range(10))
```

```
In []: type(numbers)  
Out[]: generator
```

```
In []: for nums in numbers:  
...:     print(nums)  
...:  
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

Here we can see that looping over a generator object produces the elements of the analogous list. We can also pass the generator to the function `list()` to print the list.

```
In []: numbers = (number for number in range(10))

In []: list(numbers)
Out[]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Moreover, like any other iterator, we can pass a generator to the function `next()` to iterate through its elements.

```
In []: numbers = (number for number in range(5))

In []: next(numbers)
Out[]: 0

In []: next(numbers)
Out[]: 1
```

This is known as *lazy evaluation*, whereby the evaluation of the expression is delayed until its value is needed. This can help a great deal when we are working with extremely large sequences as we don't want to store the entire list in memory, which is what comprehensions do; we want to generate elements of the sequences on the fly.

We also have generator functions that produce generator objects when called. They are written with the syntax of any other user-defined function, however, instead of returning values using the keyword `return`, they yield sequences of values using the keyword `yield`. Let us see it in practice.

```
def counter(start, end):
    """Generate values from start to end."""
    while start <= end:
        yield start
        start += 1
```

In the above function, the `while` loop is true until `start` is less than or equal to `end` and then the generator ceases to yield values. Calling the above function will return a generator object.

```
In []: c = counter(1, 5)
```

```
In []: type(c)
Out[]: generator
```

And again, as seen above, we can call the `list()` function or run a loop over generator object to traverse through its elements. Here, we pass the object `c` to the `list()` function.

```
In []: list(c)
Out[]: [1, 2, 3, 4, 5]
```

This brings us to an end of this section. Iterators are a powerful and useful tool in Python and generators are a good approach to work with lots of data. If we don't want to load all the data in the memory, we can use a generator which will pass us each piece of data at a time. Using the generator implementation saves memory.

8.3 Key Takeaways

1. An iterator is an object which can be iterated upon and will return data, one element at a time. It implements the iterator protocol, that is, `__next__()` and `__iter__()` methods.
2. An iterable is an object that can return an iterator.
3. Lists, Tuples, Strings, Dictionaries, etc. are iterables in Python. Directly or indirectly they implement the above-mentioned two methods.
4. Iterables have an `iter()` method which returns an iterator.
5. The `next()` method is used to iterate manually through all the items of an iterator.
6. The `enumerate()` function takes an iterable as an input and returns the enumerate object containing a pair of index and elements.
7. The `zip()` function accepts an arbitrary number of iterables and returns zip object which can be iterated upon.
8. Generators provides an easy way to create and implement iterators.
9. The syntax for generators is very similar to list comprehension, except that it uses a parentheses `()`.
10. Generators do not store elements in the memory and often creates the elements on the fly as required.

11. The `list()` method is used to convert generators to lists.

Chapter 9

Functions in Python

Let's now explore this remarkably handy feature seen in almost all programming languages: functions. There are lots of fantastic in-built functions in Python and its ecosystem. However, often, we as a Python programmer need to write custom functions to solve problems that are unique to our needs. Here is the definition of a function.

A function is a block of code(that performs a specific task) which runs only when it is called.

From the definition, it can be inferred that writing such block of codes, i.e. functions, provides benefits such as

- *Reusability:* Code written within a function can be called as and when needed. Hence, the same code can be reused thereby reducing the overall number of lines of code.
- *Modular Approach:* Writing a function implicitly follows a modular approach. We can break down the entire problem that we are trying to solve into smaller chunks, and each chunk, in turn, is implemented via a function.

Functions can be thought of as building blocks while writing a program, and as our program keeps growing larger and more intricate, functions help make it organized and more manageable. They allow us to give a name to a block of code, allowing us to run that block using the given name anywhere in a program any number of times. This is referred to as

calling a function. For example, if we want to compute the length of a list, we call a built-in `len` function. Using any function means we are calling it to perform the task for which it is designed.

We need to provide an input to the `len` function while calling it. The input we provide to the function is called an *argument*. It can be a data structure, string, value or a variable referring to them. Depending upon the functionality, a function can take single or multiple arguments.

There are three types of functions in Python:

- Built-in functions such as `print` to print on the standard output device, `type` to check data type of an object, etc. These are the functions that Python provides to accomplish common tasks.
- User-Defined functions: As the name suggests these are custom functions to help/resolve/achieve a particular task.
- Anonymous functions, also known as `lambda` functions are custom-made without having any name identifier.

9.1 Recapping built-in functions

Built-in functions are the ones provided by Python. The very first built-in function we had learned was the `print` function to print a string given to it as an argument on the standard output device. They can be directly used within our code without importing any module and are always available to use.

In addition to `print` function, We have learned the following built-in functions until now in the previous sections.

- `type(object)` is used to check the data type of an *object*.
- `float([value])` returns a floating point number constructed from a number or string *value*.
- `int([value])` returns an integer object constructed from a float or string *value*, or return 0 if no arguments are given.
- `round(number[, ndigits])` is used to round a float *number* up to digits specified by *ndigits*.

- `abs(value)` returns the absolute value of a *value* provided as an argument.
- `format(value[, format_spec])` converts a *value* to a ‘formatted’ representation, as controlled by *format_spec*.
- `str([object])` returns a string version of *object*. If the object is not provided, returns the empty string.
- `bool([value])` return a Boolean value, i.e. one of `True` or `False`. *value* is converted using the standard truth testing procedure¹. If the *value* is false or omitted, this returns `False`; otherwise, it returns `True`.
- `dir([object])` returns the list of names in the current local scope when an argument is not provided. With an argument, it attempts to return a list of valid attributes for that object.
- `len(object)` returns the length (the number of items) of an *object*. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

It is worth noting that almost all built-in functions take one or more arguments, perform the specific operation on it and return the output. We will keep learning about many more built-in functions as we progress through our Python learning journey. More information about various built-in functions can be obtained from Python official documentation².

9.2 User defined functions

Although Python provides a wide array of built-in functions, it does not suffice in tackling issues we would face while we develop programs and applications. As Python programmers, we might need to break down the programming challenge into smaller chunks and implement them in the form of custom or *user defined* functions. The concept of writing functions is probably an essential feature of any programming language.

Functions are defined using the `def` keyword, followed by an *identifier* name along with the parentheses, and by the final colon that ends the line. The

¹<https://docs.python.org/3/library/stdtypes.html#truth>

²<https://docs.python.org/3/library/functions.html>

block of statements which forms the *body* of the function follows the *function definition*. Here's a simple example.

```
def greet():
    """Block of statement.
    or Body of function.
    """
    print(' Hello from inside the function!')
```

The above defined `greet` function can be called using its name as shown here.

```
# Calling the function
greet()
```

And the output will be

```
Hello from inside the function.
```

9.2.1 Functions with a single argument

A function can be called as many times as we want, and Python will execute the statements within its body. This function neither takes any input nor does it return any output. It just prints the statement written within it. If the function has to take any input, it goes within the parentheses as a *parameter* during the function definition. Parameters are the values we supply to the function so that the function can do something utilizing those values.

Note the terminology used here:

- *Parameters*: They are specified within parentheses in the function definition, separated by commas.
- *Arguments*: When we call a function, values that parameters take are to be given as arguments in a comma separated format.

The modified version of the above simple function explains these two terms:

```

# Here 'person_name' is a parameter.
def greet(person_name):
    """Prints greetings along with the value received
    via the parameter."""
    print('Hello ' + person_name + '!')

```

The above function definition defines `person_name` as a parameter to the function `greet`, and it can be called as shown below:

```

# Calling the function
greet('Amigo')

```

The above call to the function `greet` takes a string `Amigo` as an argument and the output will be as follows:

Hello Amigo!

9.2.2 Functions with multiple arguments and a return statement

Both versions of the `greet` functions defined above were actually straightforward in terms of functionality that they perform. One more functionality that functions are capable of performing is to return a value to the calling statement using the keyword `return`. Consider a function that takes several parameters, performs some mathematical calculation on it and returns the output. For example:

```

# Function with two parameters 'a' and 'b'
def add(a, b):
    """Computes the addition and returns the result.

    It does not implement the print statement.
    """
    result = a + b # Computes addition
    return result # Returns the result variable

```

This user defined function `add` takes two parameters `a` and `b`, sums them together and assigns its output to a variable `result` and ultimately returns the variable to calling statement as shown below:

```
# Calling the add function
x = 5
y = 6
print(f'The addition of {x} and {y} is {add(x, y)}.')
```

We call the function `add` with two arguments `x` and `y` (as the function definition has two parameters) initialized with 5 and 6 respectively, and the addition returned by the function gets printed via the `print` statement as shown below:

```
The addition of 5 and 6 is 11.
```

Similarly, functions can also return multiple values based on the implementation. The following function demonstrates the same.

```
# Function definition
def upper_lower(x):
    """
    Returns the upper and lower version of the string.

    The value must be a string, else it will result in
    an error.
    This function does not implement any error handling
    mechanism.
    """
    upper = x.upper() # Convert x to upper string
    lower = x.lower() # Convert x to lower string
    return upper, lower # Return both variables
```

The above `upper_lower` function takes one argument `x` (a string) and converts it to their upper and lower versions. Let us call it and see the output.

NOTE: The function `upper_lower` implicitly assumes to have a string as a parameter. Providing an integer or float value as an argument while calling will result in an error.

```
# Calling the function
upper, lower = upper_lower('Python')
```

```
# Printing output
print(upper)
PYTHON

print(lower)
python
```

Here, the call to `upper_lower` function has been assigned to two variables `upper` and `lower` as the function returns two values which will be unpacked to each variable respectively and the same can be verified in the output shown above.

9.2.3 Functions with default arguments

Let us say, we are writing a function that takes multiple parameters. Often, there are common values for some of these parameters. In such cases, we would like to be able to call the function without explicitly specifying every parameter. In other words, we would like some parameters to have default values that will be used when they are not specified in the function call.

To define a function with a default argument value, we need to assign a value to the parameter of interest while defining a function.

```
def power(number, pow=2):
    """Returns the value of number to the power of pow."""
    return number**pow
```

Notice that the above function computes the first argument to the power of the second argument. The default value of the latter is 2. So now when we call the function `power` only with a single argument, it will be assigned to the `number` parameter and the return value will be obtained by squaring `number`.

```
# Calling the power function only with required argument
print(power(2))

# Output
4
```

In other words, the argument value to the second parameter `pow` became optional. If we want to calculate the number for a different power, we can obviously provide a value for it and the function will return the corresponding value.

```
# Calling the power function with both arguments
print(power(2, 5))

# Output
32
```

We can have any number of default value parameters in a function. Note however that they must follow non-default value parameters in the definition. Otherwise, Python will throw an error as shown below:

```
# Calling the power function that will throw an error
def power(pow=2, number):
    """Returns the raised number to the power of pow."""
    return number**pow

File "<ipython-input-57>", line 1
    def power(pow=2, number):
    ^
SyntaxError: non-default argument follows default argument
```

9.2.4 Functions with variable length arguments

Let's consider a scenario where we as developers aren't sure about how many arguments a user will want to pass while calling a function. For example, a function that takes floats or integers (irrespective of how many they are) as arguments and returns the sum of all of them. We can implement this scenario as shown below:

```
def sum_all(*args):
    """Sum all values in the *args."""
    # Initialize result to 0
    result = 0
```

```

# Sum all values
for i in args:
    result += i

# Return the result
return result

```

The flexible argument is written as * followed by the parameter name in the function definition. The parameter args preceded by * denotes that this parameter is of variable length. Python then unpacks it to a *tuple* of the same name args which will be available to use within the function. In the above example, we initialize the variable result to 0 which will hold the sum of all arguments. We then loop over the args to compute a sum and update the result with each iteration. Finally, we return the sum to the calling statement. The sum_all function can be called with any number of arguments and it will add them all up as follows:

```

# Calling the sum_all function with arbitrary number of
# arguments.
print(sum_all(1, 2, 3, 4, 5))

# Output
15

# Calling with different numbers of arguments.
print(sum_all(15, 20, 6))

# Output
41

```

Here, *args is used as the parameter name (the shorthand for *arguments*), but we can use any valid identifier as the parameter name. It just needs to be preceded by * to make it flexible in length. On the same lines, Python provides another flavor of flexible arguments which are preceded by double asterisk marks. When used ,they are unpacked to *dictionaries* (with the same name) by the interpreter and are available to use within the function. For example:

```

def info(**kwargs):
    """Print out key-value pairs in **kwargs."""

```

```
# Run for loop to prints dictionary items
for key, value in kwargs.items():
    print(key + ': ' + value)
```

Here, the parameter `**kwargs` are known as *keywords arguments* which will be converted into a dictionary of the same name. We then loop over it and print all keys and values. Again, it is totally valid to use an identifier other than `kwargs` as the parameter name. The `info` function can be called as follows:

```
# Calling the function
print(info(ticker='AAPL', price='146.83',
           name='Apple Inc.', country='US'))

# Output
ticker: AAPL
price: 146.83
name: Apple Inc.
country: US
```

That is all about the default and flexible arguments. We now attempt to head towards the documentation part of functions.

9.2.5 DocStrings

Python has a nifty feature called *documentation string*, usually referred to by its shorter name *docstrings*. This is an important but not required tool that should be used every time we write a program since it helps to document the program better and makes it easier to understand.

Docstrings are written within triple single/double quotes just after definition header. They are written on the first logical line of a function. Docstrings are not limited to functions only; they also apply to modules and classes. The convention followed for a docstring is a multi-line string where the first line starts with a capital letter and ends with a dot. The second line is blank followed by any detailed explanation starting from the third line. It is strongly advised to follow this convention for all docstrings. Let's see this in practice with the help of an example:

```

def power(x, y):
    """
    Equivalent to x**y or built-in pow() with two
    arguments.

    x and y should be numerical values else an appropriate
    error will be thrown for incompatible types.

    Parameters:
        x (int or float): Base value for the power operation.
        y (int or float): Power to which base value should be
                           raised.

    Returns:
        int or float: It returns x raised to the power of y.
    """

try:
    return x ** y
except Exception as e:
    print(e)

```

The function `power` defined above returns the raised value of the argument `x` powered to `y`. The thing of our interest is the docstring written within `'''` which documents the function. We can access a docstring of any function using the `__doc__` attribute (notice the *double underscores*) of that function. The docstring for the `power` function can be accessed with the following code:

```
print(power.__doc__)
```

And the output is shown below:

Equivalent to `x**y` or built-in `pow()` with two arguments.

`x` and `y` should be numerical values else an appropriate error will be thrown for incompatible types.

Parameters:

```
x (int or float): Base value for the power operation.  
y (int or float): Power to which base value should be  
raised.
```

Returns:

```
int or float: It returns x raised to the power of y.
```

We have already seen the indirect usage of docstrings in previous sections. When we use a function help in Python, it will show up the docstring. What it does is fetch the `__doc__` attribute of that function and displays it in a neat manner. If we ask for the help on the user defined power using the `print(help(power))`, Python will return the same output as shown above that we got using the `print(power.__doc__)`.

9.2.6 Nested functions and non-local variable

A nested function is a function that is defined inside another function. The syntax for the nested function is the same as that of any other function. Though the applications of nested functions are complex in nature and limited at times, even in the quant domain, it is worth mentioning it, as we might encounter this out there in the wild. Below is an example which demonstrates the nested functions.

```
# Defining nested function  
def outer():  
    """This is an enclosing function"""  
    def inner():  
        """This is a nested function"""  
        print('Got printed from the nested function.')  
  
    print('Got printed from the outer function.')  
    inner()
```

We define the function `outer` which nests another function `inner` within it. The `outer` function is referred to as an *enclosing* function and `inner` is known as *nested* function. They are also referred to as *inner* functions sometimes. Upon calling the `outer` function, Python will, in turn, call the `inner` function nested inside it and execute it. The output for the same is shown below:

```
# Calling the 'outer' function
outer()

# Output
Got printed from the outer function.
Got printed from the nested function.
```

The output we got here is intuitive. First, the print statement within the outer function got executed, followed by the print statement in the inner function. Additionally, nested functions can access variables of the enclosing functions. i.e. variables defined in the outer function can be accessed by the inner function. However, the inner or the nested function cannot modify the variables defined in the outer or enclosing function.

```
def outer(n):
    number = n

    def inner():
        print('Number =', number)

    inner()
```

A call to outer function will print the following

```
outer(5)

# Output
Number = 5
```

Though the variable `number` is not defined within `inner` function, it is able to access and print the `number`. This is possible because of scope mechanism that Python provided. We discuss more on this in the following section. Now consider, what if we want the nested function to modify the variable that is declared in the enclosing function. The default behavior of Python does not allow this. If we try to modify it, we will be presented with an error. To handle such a situation, the keyword `nonlocal` comes to the rescue.

In the nested function, we use the keyword `nonlocal` to create and change the variables defined in the enclosing function. In the example that follows, we alter the value of the variable `number`.

```
def outer(n):

    number = n
    def inner():
        nonlocal number
        number = number ** 2
        print('Square of number =', number)

    print('Number =', number)
    inner()
    print('Number =', number)
```

A call to the `outer` function will now print the number passed as an argument to it, the square of it and the newly updated number (which is nothing but the squared number only).

```
outer(3)

# Output
Number = 3
Square of number = 9
Number = 9
```

Remember, assigning a value to a variable will only create or change the variable within a particular function (or a scope) unless they are declared using the `nonlocal` statement.

9.3 Variable Namespace and Scope

If we read the *The Zen of Python* (try `import this` in Python console), the last line states *Namespaces are one honking great idea -- let's do more of those!* Let's try to understand what these mysterious namespaces are. However, before that, it will be worth spending some time understanding *names* in the context of Python.

9.3.1 Names in the Python world

A *name* (also known as an identifier) is simply a name given to an object. From Python basics, we know that everything in Python are objects. And a name is a way to access the underlying object. Let us create a new variable with a name `price` having a value 144, and check the *memory location identifier* accessible by the function `id`.

```
# Creating new variable
price = 144

# Case 1: Print memory id of the variable price
print(id(price))

# Case 1: Output
1948155424

# Case 2: Print memory id of the absolute value 144
print(id(144))

# Case 2: Output
1948155424
```

Interestingly we see that the memory location of both cases (the variable and its assigned value) is the same. In other words, both refer to the same integer object. If you would execute the above code on your workstation, memory location would almost certainly be different, but it would be the same for both the variable and value. Let's add more fun to it. Consider the following code:

```
# Assign price to old_price
old_price = price

# Assign new value to price
price = price + 1

# Print price
print(price)
```

```

# Output
145

# Print memory location of price and 145
print('Memory location of price:', id(price))
print('Memory location of 145:', id(145))

# Output
Memory location of price: 1948155456
Memory location of 145: 1948155456

# Print memory location of old_price and 144
print('Memory location of old_price:', id(old_price))
print('Memory location of 144:', id(144))

# Output
Memory location of old_price: 1948155424
Memory location of 144: 1948155424

```

We increased the value of a variable `price` by 1 unit and see that the memory location of it got changed. As you may have guessed, the memory location of an integer object `145` would also be the same as that of `price`. However, if we check the memory location of a variable `old_price`, it would point to the memory location of integer object `144`. This is efficient as Python does not need to create duplicate objects. This also makes Python powerful in a sense that a name could refer to any object, even functions. Note that functions are also objects in Python. Now that we are aware of the nitty-gritty of names in Python, we are ready to examine namespaces closely.

9.3.2 Namespace

Name conflicts happen all the time in real life. For example, we often see that there are multiple students with the same name X in a classroom. If someone has to call the student X, there would be a conflicting situation for determining which student X is actually being called. While calling, one might use the last name along with the student's first name to ensure that the call is made to the correct student X.

Similarly, such conflicts also arise in programming. It is easy and manageable to have unique names when programs are small without any external dependencies. Things start becoming complex when programs become larger and external modules are incorporated. It becomes difficult and wearisome to have unique names for all objects in the program when it spans hundreds of lines.

A namespace can be thought of a naming system to avoid ambiguity between names and ensures that all the names in a program are unique and can be used without any conflict. Most namespaces are implemented as a dictionary in Python. There is a name to object mapping, with names as keys and objects as values. Multiple namespaces can use the same name and map it to a different object. Namespaces are created at different moments and have different lifetimes. Examples of namespaces are:

- The set of built-in names: It includes built-in functions and built-in exception names.
- The global names in a module: It includes names from various modules imported in a program.
- The local names in a function: It includes names inside a function. It is created when a function is called and lasts until the function returns.

The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; that is, two different modules can contain a function `sum` without any conflict or confusion. However, they must be prefixed with the module name when used.

9.3.3 Scopes

Until now we've been using objects anywhere in a program. However, an important thing to note is not all objects are always accessible everywhere in a program. This is where the concept of scope comes into the picture. A *scope* is a region of a Python program where a namespace is directly accessible. That is when a reference to a name (lists, tuples, variables, etc.) is made, Python attempts to find the name in the namespace. The different types of scopes are:

Local scope: Names that are defined within a local scope means they are defined inside a function. They are accessible only within a function. Names

defined within a function cannot be accessed outside of it. Once the execution of a function is over, names within the local scope cease to exist. This is illustrated below:

```
# Defining a function
def print_number():
    # This is local scope
    n = 10
    # Printing number
    print('Within function: Number is', n)

print_number()

# This statement will cause error when executed
print('Outside function: Number is', n)

# Output
Within function: Number is 10

Traceback (most recent call last):
File "<ipython-input-2>", line 8, in <module>
    print('Outside function: Number is', n)
NameError: name 'n' is not defined
```

Enclosing scope: Names in the enclosing scope refer to the names defined within enclosing functions. When there is a reference to a name that is not available within the local scope, it will be searched within the enclosing scope. This is known as scope resolution. The following example helps us understand this better:

```
# This is enclosing / outer function
def outer():

    number = 10

    # This is nested / inner function
    def inner():
```

```

    print('Number is', number)

inner()

outer()

# Output
Number is 10

```

We try to print the variable `number` from within the `inner` function where it is not defined. Hence, Python tries to find the variable in the `outer` function which works as an enclosing function. What if the variable is not found within the enclosing scope as well? Python will try to find it in the *global* scope which we discuss next.

Global scope: Names in the global scope means they are defined within the main script of a program. They are accessible almost everywhere within the program. Consider the following example where we define a variable `n` before a function definition (that is, within global scope) and define another variable with the same name `n` within the function.

```

# Global variable
n = 3

def relu(val):
    # Local variable
    n = max(0, val)
    return n

print('First statement: ', relu(-3))
print('Second statement:', n)

# Output
First statement: 0
Second statement: 3

```

Here, the first print statement calls the `relu` function with a value of -3 which evaluates the maximum number to 0 and assigns the maximum

number to the variable n which in turn gets returned thereby printing 0. Next, we attempt to print the n and Python prints 3. This is because Python now refers to the variable n defined outside the function (within the global scope). Hence, we got two different values of n as they reside in different scopes. This brings us to one obvious question, what if the variable is not defined within the local scope, but available in the globals scope and we try to access that global variable? The answer is intuitive, we will be able to access it within the function. However, it would be a read-only variable and hence we won't be able to modify it. An attempt to modify a global variable result in the error as shown below:

```
# Global variable
number = 5

# Function that updates the global variable
def update_number():
    number = number + 2
    print('Within function: Number is', number)

# Calling the function
update_number()

print('Outside function: Number is', number)

# Output
Traceback (most recent call last):

File "<ipython-input-8>", line 8, in <module>
    update_number()

File "<ipython-input-8>", line 4, in update_number
    number = number + 2

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

To handle such a situation which demands modification of a global name, we define the global name within the function followed by the global key-

word. The `global` keywords allow us to access the global name within the local scope. Let us run the above code, but with the `global` keyword.

```
# Global variable
number = 5

# Function that updates the global variable
def update_number():
    global number
    number = number + 2
    print('Within function: Number is', number)

# Calling the function
update_number()

print('Outside function: Number is', number)

# Output
Within function: Number is 7
Outside function: Number is 7
```

The `global` keyword allowed us to modify the global variable from the local scope without any issues. This is very similar to the keyword `non-local` which allows us to modify variables defined in the enclosing scope.

Built-in scope: This scope consists of names predefined within built-ins module in Python such as `sum`, `print`, `type`, etc. Though we neither define these functions anywhere in our program nor we import them from any external module they are always available to use.

To summarize, when executing a Python code, names are searched in various scopes in the following order:

1. Local
2. Enclosing
3. Global
4. Built-in

If they are not found in any scope, Python will throw an error.

9.4 Lambda functions

We have written functions above using the `def` keyword, function headers, DocStrings and function bodies. There's a quicker way to write on-the-fly functions in Python and they are known as lambda functions. They are also referred to as anonymous functions sometimes. We use the keyword `lambda` to write such functions. The syntax for lambda functions is as follows:

```
lambda arguments: expression
```

Firstly, the syntax shows that there is no function name. Secondly, *arguments* refers to parameters, and finally, *expression* depicts the function body. Let us create a function `square` which squares the argument provided to it and returns the result. We create this function using the `def` keyword.

```
# Function definition
def square(arg):
    """
    Computes the square of an argument and returns the
    result.

    It does not implement the print statement."""
    result = arg * arg
    return result

# Calling the function and printing its output
print(square(3))

# Output
9
```

The function `square` defined above can be re-written in a single line using the `lambda` keyword as shown below:

```
# Creating a lambda function and assigning it to square
square = lambda arg: arg * arg
```

```
# Calling the lambda function using the name 'square'
print(square(3))

# Outpuut
9
```

In the above lambda function, it takes one argument denoted by *arg* and returns its square. Lambda functions can have as many number of arguments as we want after the `lambda` keyword during its definition. We will restrict our discussion up to two arguments to understand how multiple arguments work. We create another lambda function to raise the first argument to the power of the second argument.

```
# Creating a lambda function to mimic 'raise to power'
# operation
power = lambda a, b: a ** b

# Calling the lambda function using the name 'power'
print(power(2, 3))

# Output
8
```

Lambda functions are extensively used along with built-in `map` and `filter` functions.

9.4.1 `map()` Function

The `map` function takes two arguments: a function and a sequence such as a list. This function makes an iterator that applies the function to each element of a sequence. We can pass lambda function to this `map` function without even naming it. In this case, we refer to lambda functions as an anonymous function. In the following example, we create a list `nums` consisting of numbers and pass it to a `map` function along with the lambda function which will square each element of the list.

```
# Creating a list of all numbers
nums = [1, 2, 3, 4, 5]
```

```
# Defining a lambda function to square each number and
# passing it as an argument to map function
squares = map(lambda num: num ** 2, nums)
```

The lambda function in the above example will square each element of the list `nums` and the `map` function will map each output to the corresponding elements in the original list. We then store the result into a variable called `squares`. If we print the `square` variable, Python will reveal us that it is a `map` object.

```
# Printing squares
print(squares)

# Output
<map object at 0x00000000074EAD68>
```

To see what this object contains, we need to cast it to list using the `list` function as shown below:

```
# Casting map object squares to a list and printing it
print(list(squares))

# Output
[1, 4, 9, 16, 25]
```

9.4.2 filter() Function

The `filter` function takes two arguments: a function or `None` and a sequence. This function offers a way to filter out elements from a list that don't satisfy certain criteria. Before we embed a lambda function with it, let's understand how it works.

```
# Creating a list of booleans
booleans = [False, True, True, False, True]

# Filtering 'booleans', casting it to a list, and finally
# printing it
print(list(filter(None, booleans)))
```

```
# Output  
[True, True, True]
```

In the above example, we first create a list of random boolean values. Next, we pass it to the `filter` function along with the `None` which specifies to return the items that are true. Lastly, we cast the output of the `filter` function to a list as it outputs a filter object. In a more advanced scenario, we can embed a lambda function in the `filter` function. Consider that we have been given a scenario where we need to filter all strings whose length is greater than 3 from a given set of strings. We can use `filter` and lambda functions together to achieve this. This is illustrated below:

```
# Creating a pool of random strings  
strings = ['one', 'two', 'three', 'four', 'five', 'six']  
  
# Filtering strings using a lambda and filter functions  
filtered_strings = filter(lambda string: len(string) > 3,  
                           strings)  
  
# Casting 'filtered_strings' to a list and printing it  
print(list(filtered_strings))  
  
# Output  
['three', 'four', 'five']
```

In the above example, a lambda function is used within the `filter` function which checks for the length of each string in the `strings` list. And the `filter` function will then filter out the strings which match the criteria defined by the lambda function.

Apart from the `map` and `filter` functions discussed above, now we will learn another handy function `zip` which can be used for iterating through multiple sequences simultaneously.

9.4.3 `zip()` Function

As regular computer users, we often come across a file with `.zip` extension aka zip files. Basically, these files are the files which have zipped other files

within them. In other words, zip files work as a container to hold other files.

In the Python world, the `zip` function works more or less as a container for iterables instead of real files. The syntax for the `zip` is shown below:

```
zip(*iterables)
```

It takes an iterable as an input and returns the iterator that aggregates elements from each of the iterable. The output contains the iterator of a tuple. The *i-th* element in the iterator is the tuple consisting the *i-th* element from each input. If the iterables in the input are of unequal sizes, the output iterator stops when the shortest input iterable is exhausted. With no input, it returns an empty iterator. Let us understand the working of `zip` with the help of an example.

```
# Defining iterables for the input
tickers = ['AAPL', 'MSFT', 'GOOG']
companies = ['Apple Inc', 'Microsoft Corporation',
             'Alphabet Inc']

# Zipping the above defined iterables using the 'zip'
zipped = zip(tickers, companies)
```

We define two lists `tickers` and `companies` which are used as an input to the `zip`. The `zipped` object is the iterator of type `zip` and hence we can iterate either over it using a looping technique to print its content:

```
# Iterating over a zipped object
for ticker, company in zipped:
    print('Ticker name of {} is {}'.format(ticker,
                                             company))

# Output
Ticker name of AAPL is Apple Inc.
Ticker name of MSFT is Microsoft Corporation.
Ticker name of GOOG is Alphabet Inc.
```

or cast it to sequential data structures such as list or tuple easily.

```
# Casting the zip object to a list and printing it
print(list(zipped))

# Output
[('AAPL', 'Apple Inc.'),
 ('MSFT', 'Microsoft Corporation'),
 ('GOOG', 'Alphabet Inc.')]
```

As we should expect, the zipped object contains a sequence of tuples where elements are from the corresponding inputs. A `zip` object in conjunction with `*` unzips the elements as they were before. For example:

```
# Unzipping the zipped object
new_tickers, new_companies = zip(*zipped)

# Printing new unzipped sequences
print(new_tickers)
('AAPL', 'MSFT', 'GOOG')

print(new_companies)
('Apple Inc.', 'Microsoft Corporation', 'Alphabet Inc.')
```

We unzip the `zip` object zipped to two sequences `new_tickers` and `new_companies`. By printing these sequences, we can see that the operation got successful and elements got unzipped successfully into respective tuples.

9.5 Key Takeaways

1. A function is a block of statements that can be reused as and when required.
2. There are three types of functions available in Python: Built-in functions, User-defined functions and anonymous functions using the `lambda` keyword.
3. Python provides various built-in functions to perform common programming tasks such as `print()`, `len()`, `dir()`, `type()`, etc
4. User-defined functions are defined using the keyword `def`.
5. Functions may take zero or more arguments. Arguments are specified while defining a function within parentheses `()`.

6. It is possible that arguments take some default value in the function definition. Such arguments are called default arguments.
7. Functions can return a value using the keyword `return`.
8. Functions can have variable-length arguments. There are two types of such arguments:
 - (a) An argument that is preceded by `*` in the function definition can have a flexible number of values within it. And gets unpacked to a tuple inside a function.
 - (b) An argument that is preceded by `**` in the function definition can have a flexible number of key-value pairs and gets unpacked to a dictionary inside a function.
9. A docstring is used to document a function and is written within triple single/double quotes. It can be accessed by the `__doc__` attribute on the function name.
10. Docstrings can be used to document modules and classes as well.
11. A namespace is a naming system in Python to avoid ambiguity between names (variable names, object names, module names, class names, etc.).
12. A scope is a region of a Python program where a namespace is directly accessible.
13. The `global` keyword is used when a variable that is defined outside a function and needs to be accessed from within a function.
14. The `lambda` keyword is used to create anonymous functions. Such functions are created during the run time and do not have any name associated with them.
15. `map()`, `filter()` and `zip()` functions are often used with anonymous functions.

Chapter 10

NumPy Module

NumPy, an acronym for *Numerical Python*, is a package to perform scientific computing in Python efficiently. It includes random number generation capabilities, functions for basic linear algebra, Fourier transforms as well as a tool for integrating Fortran and C/C++ code along with a bunch of other functionalities.

NumPy is an open-source project and a successor to two earlier scientific Python libraries: *Numeric* and *Numarray*.

It can be used as an efficient multi-dimensional container of generic data. This allows NumPy to integrate with a wide variety of databases seamlessly. It also features a collection of routines for processing single and multidimensional vectors known as arrays in programming parlance.

NumPy is not a part of the Python Standard Library and hence, as with any other such library or module, it needs to be installed on a workstation before it can be used. Based on the Python distribution one uses, it can be installed via a command prompt, conda prompt, or terminal using the following command. *One point to note is that if we use the Anaconda distribution to install Python, most of the libraries (like NumPy, pandas, scikit-learn, matplotlib, etc.) used in the scientific Python ecosystem come pre-installed.*

```
pip install numpy
```

NOTE: If we use the Python or iPython console to install the

NumPy library, the command to install it would be preceded by the character !.

Once installed we can use it by importing into our program by using the import statement. The de facto way of importing is shown below:

```
import numpy as np
```

Here, the NumPy library is imported with an alias of np so that any functionality within it can be used with convenience. We will be using this form of alias for all examples in this section.

10.1 NumPy Arrays

A Python list is a pretty powerful sequential data structure with some nifty features. For example, it can hold elements of various data types which can be added, changed or removed as required. Also, it allows index subsetting and traversal. But lists lack an important feature that is needed while performing data analysis tasks. We often want to carry out operations over an entire collection of elements, and we expect Python to perform this fast. With lists executing such operations over all elements efficiently is a problem. For example, let's consider a case where we calculate PCR (Put Call Ratio) for the previous 5 days. Say, we have put and call options volume (in Lacs) stored in lists call_vol and put_vol respectively. We then compute the PCR by dividing put volume by call volume as illustrated in the below script:

```
# Put volume in lacs
In []: put_vol = [52.89, 45.14, 63.84, 77.1, 74.6]

# Call volume in lacs
In []: call_vol = [49.51, 50.45, 59.11, 80.49, 65.11]

# Computing Put Call Ratio (PCR)
In []: put_vol / call_vol
Traceback (most recent call last):

File "<ipython-input-12>", line 1, in <module>
```

```
put_vol / call_vol

TypeError: unsupported operand type(s) for /: 'list' and
'list'
```

Unfortunately, Python threw an error while calculating PCR values as it has no idea on how to do calculations on lists. We can do this by iterating over each item in lists and calculating the PCR for each day separately. However, doing so is inefficient and tiresome too. A way more elegant solution is to use NumPy arrays, an alternative to the regular Python list.

The NumPy array is pretty similar to the list, but has one useful feature: we can perform operations over entire arrays(all elements in arrays). It's easy as well as super fast. Let us start by creating a NumPy array. To do this, we use `array()` function from the NumPy package and create the NumPy version of `put_vol` and `call_vol` lists.

```
# Importing NumPy library
In []: import numpy as np

# Creating arrays
In []: n_put_vol = np.array(put_vol)

In []: n_call_vol = np.array(call_vol)

In []: n_put_vol
Out[]: array([52.89, 45.14, 63.84, 77.1 , 74.6 ])

In []: n_call_vol
Out[]: array([49.51, 50.45, 59.11, 80.49, 65.11])
```

Here, we have two arrays `n_put_vol` and `n_call_vol` which holds put and call volume respectively. Now, we can calculate PCR in one line:

```
# Computing Put Call Ratio (PCR)
In []: pcr = n_put_vol / n_call_vol

In []: pcr
Out[]: array([1.06826904, 0.89474727, 1.0800203,
0.95788297, 1.14575334])
```

This time it worked, and calculations were performed element-wise. The first observation in pcr array was calculated by dividing the first element in n_put_vol by the first element in n_call_vol array. The second element in pcr was computed using the second element in the respective arrays and so on.

First, when we tried to compute PCR with regular lists, we got an error, because Python cannot do calculations with lists like we want it to. Then we converted these regular lists to NumPy arrays and the same operation worked without any problem. NumPy work with arrays as if they are scalars. But we need to pay attention here. NumPy can do this easily because it assumes that array can only contain values of a single type. It's either an array of integers, floats or booleans and so on. If we try to create an array of different types like the one mentioned below, the resulting NumPy array will contain a single type only. String in the below case:

```
In []: np.array([1, 'Python', True])
Out[]: array(['1', 'Python', 'True'], dtype='<U11')
```

NOTE: NumPy arrays are made to be created as homogeneous arrays, considering the mathematical operations that can be performed on them. It would not be possible with heterogeneous data sets.

In the example given above, an integer and a boolean were both converted to strings. NumPy array is a new type of data structure type like the Python list type that we have seen before. This also means that it comes with its own methods, which will behave differently from other types. Let us implement the + operation on the Python list and NumPy arrays and see how they differ.

```
# Creating lists
In []: list_1 = [1, 2, 3]

In []: list_2 = [5, 6, 4]

# Adding two lists
In []: list_1 + list_2
Out[]: [1, 2, 3, 5, 6, 4]
```

```
# Creating arrays
In []: arr_1 = np.array([1, 2, 3])

In []: arr_2 = np.array([5, 6, 4])

# Adding two arrays
In []: arr_1 + arr_2
Out[]: array([6, 8, 7])
```

As can be seen in the above example, performing the `+` operation with `list_1` and `list_2`, the list elements are pasted together, generating a list with 6 elements. On the other hand, if we do this with NumPy arrays, Python will do an element-wise sum of the arrays.

10.1.1 N-dimensional arrays

Until now we have worked with two arrays: `n_put_vol` and `n_call_vol`. If we are to check its type using `type()`, Python tells us that they are of type `numpy.ndarray` as shown below:

```
# Checking array type
In []: type(n_put_vol)
Out[]: numpy.ndarray
```

Based on the output we got, it can be inferred that they are of data type `ndarray` which stands for *n-dimensional array* within NumPy. These arrays are one-dimensional arrays, but NumPy also allows us to create two dimensional, three dimensional and so on. We will stick to two dimensional for our learning purpose in this module. We can create a 2D (two dimensional) NumPy array from a regular Python list of lists. Let us create one array for all put and call volumes.

```
# Recalling put and call volumes lists
In []: put_vol
Out[]: [52.89, 45.14, 63.84, 77.1, 74.6]

In []: call_vol
Out[]: [49.51, 50.45, 59.11, 80.49, 65.11]
```

```
# Creating a two-dimensional array
In []: n_2d = np.array([put_vol, call_vol])

In []: n_2d
Out[]:
array([[52.89, 45.14, 63.84, 77.1 , 74.6 ],
       [49.51, 50.45, 59.11, 80.49, 65.11]])
```

We see that `n_2d` array is a rectangular data structure. Each list provided in the `np.array` creation function corresponds to a row in the two-dimensional NumPy array. Also for 2D arrays, the NumPy rule applies: an array can only contain a single type. If we change one float value in the above array definition, all the array elements will be coerced to strings, to end up with a homogeneous array. We can think of a 2D array as an advanced version of lists of a list. We can perform element-wise operation with 2D as we had seen for a single dimensional array.

10.2 Array creation using built-in functions

An explicit input has been provided while creating `n_call_vol` and `n_put_vol` arrays. In contrast, NumPy provides various built-in functions to create arrays and input to them will be produced by NumPy. Below we discuss a handful of such functions:

- `zeros(shape, dtype=float)` returns an array of a given shape and type, filled with zeros. If the `dtype` is not provided as an input, the default type for the array would be float.

```
# Creating a one-dimensional array
In []: np.zeros(5)
Out[]: array([0., 0., 0., 0., 0.])
```

```
# Creating a two-dimensional array
In []: np.zeros((3, 5))
Out[]:
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
```

```
[0., 0., 0., 0., 0.]])
```

```
# Creating a one-dimensional array of integer type
```

```
In []: np.zeros(5, dtype=int)
```

```
Out[]: array([0, 0, 0, 0, 0])
```

- `ones(shape, dtype=float)` returns an array of a given shape and type, filled with ones. If the *dtype* is not provided as an input, the default type for the array would be float.

```
# Creating a one-dimensional array
```

```
In []: np.ones(5)
```

```
Out[]: array([1., 1., 1., 1., 1.])
```

```
# Creating a one-dimensional array of integer type
```

```
In []: np.ones(5, dtype=int)
```

```
Out[]: array([1, 1, 1, 1, 1])
```

- `full(shape, fill_value, dtype=None)` returns an array of a given shape and type, fill with *fill_value* given in input parameters.

```
# Creating a one-dimensional array with value as 12
```

```
In []: np.full(5, 12)
```

```
Out[]: array([12, 12, 12, 12, 12])
```

```
# Creating a two-dimensional array with value as 9
```

```
In []: np.full((2, 3), 9)
```

```
Out[]:
```

```
array([[9, 9, 9],  
       [9, 9, 9]])
```

- `arange([start,]stop, [step])` returns an array with evenly spaced values within a given interval. Here the *start* and *step* parameters are optional. If they are provided NumPy will consider them while computing the output. Otherwise, range computation starts from 0. For all cases, stop value will be excluded in the output.

```
# Creating an array with only stop argument
```

```
In []: np.arange(5)
```

```

Out[]: array([0, 1, 2, 3, 4])

# Creating an array with start and stop arguments
In []: np.arange(3, 8)
Out[]: array([3, 4, 5, 6, 7])

# Creating an array with given interval and step value
# as 0.5
In []: np.arange(3, 8, 0.5)
Out[]: array([3. , 3.5, 4. , 4.5, 5. , 5.5, 6. , 6.5, 7. , 7.5])

```

- `linspace(start, stop, num=50, endpoint=True)` returns evenly spaced numbers over a specified interval. The number of samples to be returned is specified by the `num` parameter. The endpoint of the interval can optionally be excluded.

```

# Creating an evenly spaced array with five numbers within
# interval 2 to 3
In []: np.linspace(2.0, 3.0, num=5)
Out[]: array([2. , 2.25, 2.5 , 2.75, 3. ])

```

```

# Creating an array excluding end value
In []: np.linspace(2.0, 3.0, num=5, endpoint=False)
Out[]: array([2. , 2.2, 2.4, 2.6, 2.8])

```

```

# Creating an array with ten values within the specified
# interval
In []: np.linspace(11, 20, num=10)
Out[]: array([11., 12., 13., 14., 15., 16., 17., 18., 19.,
20.])

```

10.3 Random Sampling in NumPy

In addition to built-in functions discussed above, we have a `random` submodule within the NumPy that provides handy functions to generate data randomly and draw samples from various distributions. Some of the widely used such functions are discussed here.

- `rand([d0, d1, ..., dn])` is used to create an array of a given shape and populate it with random samples from a *uniform distribution* over [0, 1). It takes only positive arguments. If no argument is provided, a single float value is returned.

```
# Generating single random number
In []: np.random.rand()
Out[]: 0.1380210268817208

# Generating a one-dimensional array with four random
# values
In []: np.random.rand(4)
Out[]: array([0.24694323, 0.83698849, 0.0578015,
0.42668907])

# Generating a two-dimensional array
In []: np.random.rand(2, 3)
Out[]:
array([[0.79364317, 0.15883039, 0.75798628],
[0.82658529, 0.12216677, 0.78431111]])
```

- `randn([d0, d1, ..., dn])` is used to create an array of the given shape and populate it with random samples from a *standard normal* distributions. It takes only positive arguments and generates an array of shape (d0, d1, ..., dn) filled with random floats sampled from a univariate normal distribution of mean 0 and variance 1. If no argument is provided, a single float randomly sampled from the distribution is returned.

```
# Generating a random sample
In []: np.random.randn()
Out[]: 0.5569441449249491

# Generating a two-dimensional array over N(0, 1)
In []: np.random.randn(2, 3)
Out[]:
array([[ 0.43363995, -1.04734652, -0.29569917],
[ 0.31077962, -0.49519421,  0.29426536]])
```

```
# Generating a two-dimensional array over N(3, 2.25)
In []: 1.5 * np.random.randn(2, 3) + 3
Out[]:
array([[1.75071139, 2.81267831, 1.08075029],
       [3.35670489, 3.96981281, 1.7714606 ]])
```

- `randint(low, high=None, size=None)` returns a random integer from a discrete uniform distribution with limits of *low* (inclusive) and *high* (exclusive). If *high* is None (the default), then results are from 0 to *low*. If the *size* is specified, it returns an array of the specified size.

```
# Generating a random integer between 0 and 6
In []: np.random.randint(6)
Out[]: 2
```

```
# Generating a random integer between 6 and 9
In []: np.random.randint(6, 9)
Out[]: 7
```

```
# Generating a one-dimensional array with values between 3
# and 9
In []: np.random.randint(3, 9, size=5)
Out[]: array([6, 7, 8, 8, 5])
```

```
# Generating a two-dimensional array with values between 3
# and 9
In []: np.random.randint(3, 9, size=(2, 5))
Out[]:
array([[5, 7, 4, 6, 4],
       [6, 8, 8, 5, 3]])
```

- `random(size=None)` returns a random float value between 0 and 1 which is drawn from the *continuous uniform* distribution.

```
# Generating a random float
In []: np.random.random()
Out[]: 0.6013749764953444
```

```
# Generating a one-dimensional array
```

```
In []: np.random.random(3)
Out[]: array([0.69929315, 0.61152299, 0.91313813])

# Generating a two-dimensional array
In []: np.random.random((3, 2))
Out[]:
array([[0.55779547, 0.6822698 ],
       [0.75476145, 0.224952 ],
       [0.99264158, 0.02755453]])
```

- `binomial(n, p, size=None)` returns samples drawn from a binomial distribution with n trials and p probability of success where n is greater than 0 and p is in the interval of 0 and 1.

```
# Number of trials, probability of each trial
In []: n, p = 1, .5

# Flipping a coin 1 time for 50 times
In []: samples = np.random.binomial(n, p, 50)

In []: samples
Out[]:
array([1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0,
       0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1,
       0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0])
```

- `normal(mu=0.0, sigma=1.0, size=None)` draws random samples from a normal (Gaussian) distribution. If no arguments provided, a sample will be drawn from $N(0, 1)$.

```
# Initialize mu and sigma
In []: mu, sigma = 0, 0.1

# Drawing 5 samples in a one-dimensional array
In []: np.random.normal(mu, sigma, 5)
Out[]: array([ 0.06790522, 0.0092956, 0.07063545,
              0.28022021, -0.13597963])
```

```
# Drawing 10 samples in a two-dimensional array of
```

```
# shape (2, 5)
In []: np.random.normal(mu, sigma, (2, 5))
Out[]:
array([[-0.10696306, -0.0147926, -0.07027478, 0.04399432,
       -0.03861839],
       [-0.02004485, 0.08760261, 0.18348247, -0.09351321,
       -0.19487115]])
```

- `uniform(low=0.0, high=1.0, size=None)` draws samples from a uniform distribution over the interval 0 (including) and 1 (excluding), if no arguments are provided. In other words, any value drawn is equally likely within the interval.

```
# Creating a one-dimensional array with samples drawn
# within [-1, 0)
In []: np.random.uniform(-1, 0, 10)
Out[]:
array([-0.7910379, -0.64144624, -0.64691011, -0.03817127,
       -0.24480339, -0.82549031, -0.37500955, -0.88304322,
       -0.35196588, -0.51377252])
```



```
# Creating a two-dimensional array with samples drawn
# within [0, 1)
In []: np.random.uniform(size=(5, 2))
Out[]:
array([[0.43155784, 0.41360889],
       [0.81813931, 0.70115211],
       [0.40003811, 0.2114227],
       [0.95487774, 0.92251769],
       [0.91042434, 0.98697917]])
```

In addition to functions shown above, we can draw samples from various other distributions such as Poisson, Gamma, Exponential, etc. using NumPy.

10.4 Array Attributes and Methods

We now have some idea about the working of NumPy arrays. Let us now explore the functionalities provided by them. As with any Python object,

NumPy arrays also have a rich set of attributes and methods which simplifies the data analysis process to a great extent. Following are the most useful array attributes. For illustration purpose, we will be using previously defined arrays.

- `ndim` attribute displays the number of dimensions of an array. Using this attribute on `n_call_vol` and `pcr`, we expect dimensions to be 1 and 2 respectively. Let's check.

```
# Checking dimensions for n_call_vol array
```

```
In []: np_call_vol.ndim  
Out[]: 1
```

```
In []: n_2d.ndim  
Out[]: 2
```

- `shape` returns a tuple with the dimensions of the array. It may also be used to reshape the array in-place by assigning a tuple of array dimensions to it.

```
# Checking the shape of the one-dimensional array
```

```
In []: n_put_vol.shape  
Out[]: (5,)
```

```
# Checking shape of the two-dimensional array
```

```
In []: n_2d.shape  
Out[]: (2, 5) # It shows 2 rows and 5 columns
```

```
# Printing n_2d with 2 rows and 5 columns
```

```
In []: n_2d  
Out[]:  
array([[52.89, 45.14, 63.84, 77.1 , 74.6 ],  
       [49.51, 50.45, 59.11, 80.49, 65.11]])
```

```
# Reshaping n_2d using the shape attribute
```

```
In []: n_2d.shape = (5, 2)
```

```
# Printing reshaped array
```

```
In []: n_2d
```

```
Out[]:  
array([[52.89, 45.14],  
       [63.84, 77.1 ],  
       [74.6 , 49.51],  
       [50.45, 59.11],  
       [80.49, 65.11]])
```

- `size` returns the number of elements in the array.

```
In []: n_call_vol.size  
Out[]: 5
```

```
In []: n_2d.size  
Out[]: 10
```

- `dtype` returns the data-type of the array's elements. As we learned above, NumPy comes with its own data type just like regular built-in data types such as `int`, `float`, `str`, etc.

```
In []: n_put_vol.dtype  
Out[]: dtype('float64')
```

A typical first step in analyzing a data is getting to the data in the first place. In an ideal data analysis process, we generally have thousands of numbers which need to be analyzed. Simply staring at these numbers won't provide us with any insights. Instead, what we can do is generate summary statistics of the data. Among many useful features, NumPy also provides various statistical functions which are good to perform such statistics on arrays.

Let us create a `samples` array and populate it with samples drawn from a normal distribution with a mean of 5 and standard deviation of 1.5 and compute various statistics on it.

```
# Creating a one-dimensional array with 1000 samples drawn  
# from a normal distribution  
In []: samples = np.random.normal(5, 1.5, 1000)  
  
# Creating a two-dimensional array with 25 samples
```

```
# drawn from a normal distribution
In []: samples_2d = np.random.normal(5, 1.5, size=(5, 5))

In []: samples_2d
Out[]:
array([[5.30338102, 6.29371936, 2.74075451, 3.45505812,
       7.24391809],
       [5.20554917, 5.33264245, 6.08886915, 5.06753721,
       6.36235494],
       [5.86023616, 5.54254211, 5.38921487, 6.77609903,
       7.79595902],
       [5.81532883, 0.76402556, 5.01475416, 5.20297957,
       7.57517601],
       [5.76591337, 1.79107751, 5.03874984, 5.05631362,
       2.16099478]])
```

- `mean(a, axis=None)` returns the average of the array elements. The average is computed over the flattened array by default, otherwise over the specified axis.
- `average(a, axis=None)` returns the average of the array elements and works similar to that of `mean()`.

```
# Computing mean
In []: np.mean(samples)
Out[]: 5.009649198007546

In []: np.average(samples)
Out[]: 5.009649198007546

# Computing mean with axis=1 (over each row)
In []: np.mean(samples_2d, axis=1)
Out[]: array([5.00736622, 5.61139058, 6.27281024,
       4.87445283, 3.96260983])

In []: np.average(samples_2d, axis=1)
Out[]: array([5.00736622, 5.61139058, 6.27281024,
       4.87445283, 3.96260983])
```

- `max(a, axis=None)` returns the maximum of an array or maximum along an axis.

```
In []: np.max(samples)
Out[]: 9.626572532562523
```

```
In []: np.max(samples_2d, axis=1)
Out[]: array([7.24391809, 6.36235494, 7.79595902,
              7.57517601, 5.76591337])
```

- `median(a, axis=None)` returns the median along the specified axis.

```
In []: np.median(samples)
Out[]: 5.0074934668143865
```

```
In []: np.median(samples_2d)
Out[]: 5.332642448141249
```

- `min(a, axis=None)` returns the minimum of an array or minimum along an axis.

```
In []: np.min(samples)
Out[]: 0.1551821703754115
```

```
In []: np.min(samples_2d, axis=1)
Out[]: array([2.74075451, 5.06753721, 5.38921487,
              0.76402556, 1.79107751])
```

- `var(a, axis=None)` returns the variance of an array or along the specified axis.

```
In []: np.var(samples)
Out[]: 2.2967299389550466
```

```
In []: np.var(samples_2d)
Out[]: 2.93390175942658
```

```
# The variance is computed over each column of numbers
In []: np.var(samples_2d, axis=0)
Out[]: array([0.07693981, 4.95043105, 1.26742732,
              1.10560727, 4.37281009])
```

- `std(a, axis=None)` returns the standard deviation of an array or along the specified axis.

```
In []: np.std(samples)
Out[]: 1.5154965981337756
```

```
In []: np.std(samples_2d)
Out[]: 1.7128636137844075
```

- `sum(a, axis=None)` returns the sum of array elements.

```
# Recalling the array n_put_vol
In []: n_put_vol
Out[]: array([52.89, 45.14, 63.84, 77.1 , 74.6 ])
```

```
# Computing sum of all elements within n_put_vol
In []: np.sum(n_put_vol)
Out[]: 313.57
```

```
# Computing sum of all array over each row
In []: np.sum(samples_2d, axis=1)
Out[]: array([25.03683109, 28.05695291, 31.36405118,
              24.37226413, 19.81304913])
```

- `cumsum(a, axis=None)` returns the cumulative sum of the elements along a given axis.

```
In []: np.cumsum(n_put_vol)
Out[]: array([ 52.89,  98.03, 161.87, 238.97, 313.57])
```

The methods discussed above can also be directly called upon NumPy objects such as `samples`, `n_put_vol`, `samples_2d`, etc. instead of using the `np.` format as shown below. The output will be the same in both cases.

```
# Using np. format to compute the sum
In []: np.sum(samples)
Out[]: 5009.649198007546
```

```
# Calling sum() directly on a NumPy object
In []: samples.sum()
Out[]: 5009.649198007546
```

10.5 Array Manipulation

NumPy defines a new data type called `ndarray` for the array object it creates. This also means that various operators such as arithmetic operators, logical operator, boolean operators, etc. work in ways unique to it as we've seen so far. There's a flexible and useful array manipulation technique that NumPy provides to use on its data structure using *broadcasting*.

The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations (with certain constraints). The smaller array is '*broadcast*' across the larger array so that they have compatible shapes. It also provides a mean of vectorizing array operations.

NumPy operations are usually done on pairs of arrays on an element-by-element basis. In the simplest case, the two arrays must have exactly the same shape as in the following example.

```
In []: a = np.array([1, 2, 3])
In []: b = np.array([3, 3, 3])
In []: a * b
Out[]: array([3, 6, 9])
```

NumPy's broadcasting rule relaxes this constraint when the array's shapes meet certain constraints. The simplest broadcasting example occurs when an array and a scalar value are combined in operation as depicted below:

```
In []: a = np.array([1, 2, 3])
In []: b = 3
In []: a * b
Out[]: array([3, 6, 9])
```

The result is equivalent to the previous example where `b` was an array. We can think of the scalar `b` in the above example being *stretched* during the arithmetic operation into an array with the same shape as `a`. The new elements in `b` are simply copies of the original scalar. Here, the

stretching analogy is only conceptual. NumPy is smart enough to use the original scalar value without actually making copies so that broadcasting operations are as memory and computationally efficient as possible.

The code in the last example is more efficient because broadcasting moves less memory around during the multiplication than that of its counterpart defined above it. Along with efficient number processing capabilities, NumPy also provides various methods for array manipulation thereby proving versatility. We discuss some of them here.

- `exp(*args)` returns the exponential of all elements in the input array. The numbers will be raised to e also known as Euler's number.

```
# Computing exponentials for the array 'a'  
In []: np.exp(a)  
Out[]: array([ 2.71828183, 7.3890561, 20.08553692])
```

- `sqrt(*args)` returns the positive square-root of an array, element-wise.

```
# Computing square roots of a given array  
In []: np.sqrt([1, 4, 9, 16, 25])  
Out[]: array([1., 2., 3., 4., 5.])
```

- `reshape(new_shape)` gives a new shape to an array without changing its data.

```
# Creating a one-dimensional array with 12 elements  
In []: res = np.arange(12)  
  
In []: res  
Out[]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
```

```
# Reshaping the 'res' array to 2-dimensional array  
In []: np.reshape(res, (3, 4))  
Out[]:  
array([[ 0, 1, 2, 3],  
       [ 4, 5, 6, 7],  
       [ 8, 9, 10, 11]])
```

```
# Reshaping the dimensions from (3, 4) to (2, 6)
In []: np.reshape(res, (2, 6))
Out[]:
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

- `resize(a, new_shape)` return a new array with the specified shape. If the new array is larger than the original array, then the new array is filled with repeated copies of *a*.

```
# Creating a one-dimensional array
In []: demo = np.arange(4)
```

```
In []: demo
Out[]: array([0, 1, 2, 3])
```

```
# Resizing a 'demo' array to (2, 2)
In []: np.resize(demo, (2, 2))
Out[]:
array([[0, 1],
       [2, 3]])
```

```
# Resizing a 'demo' greater than its size.
In []: np.resize(demo, (4, 2))
Out[]:
array([[0, 1],
       [2, 3],
       [0, 1],
       [2, 3]])
```

- `round(a, decimals=0)` round an array to the given number of decimals. If decimals are not given, elements will be rounded to the whole number.

```
# Creating a one-dimensional array
In []: a = np.random.rand(5)
```

```
# Printing array
```

```

In []: a
Out[]: array([0.71056952, 0.58306487, 0.13270092,
              0.38583513, 0.7912277])

# Rounding to 0 decimals
In []: a.round()
Out[]: array([1., 1., 0., 0., 1.])

# Rounding to 0 decimals using the np.round syntax
In []: np.round(a)
Out[]: array([1., 1., 0., 0., 1.])

# Rounding to 2 decimals
In []: a.round(2)
Out[]: array([0.71, 0.58, 0.13, 0.39, 0.79])

# Rounding to 3 decimals using the np.round syntax
In []: np.round(a, 3)
Out[]: array([0.711, 0.583, 0.133, 0.386, 0.791])

```

- `sort(a, kind='quicksort')` returns a sorted copy of an array. The default sorting algorithm used is *quicksort*. Other available options are *mergesort* and *heapsort*.

```

In []: np.sort(n_put_vol)
Out[]: array([45.14, 52.89, 63.84, 74.6 , 77.1 ])

In []: np.sort(samples_2d)
Out[]:
array([[2.74075451, 3.45505812, 5.30338102, 6.29371936,
       7.24391809],
       [5.06753721, 5.20554917, 5.33264245, 6.08886915,
       6.36235494],
       [5.38921487, 5.54254211, 5.86023616, 6.77609903,
       7.79595902],
       [0.76402556, 5.01475416, 5.20297957, 5.81532883,
       7.57517601],
       [1.79107751, 2.16099478, 5.03874984, 5.05631362,
       5.76591337]])

```

- `vstack(tup)` stacks arrays provided via *tup* in sequence vertically (row wise).
- `hstack(tup)` stacks arrays provided via *tup* in sequence horizontally (column wise).
- `column_stack(tup)` stacks 1-dimensional arrays as column into a 2-dimensional array. It takes a sequence of 1-D arrays and stacks them as columns to make a single 2-D array.

```
# Creating sample arrays
In []: a = np.array([1, 2, 3])

In []: b = np.array([4, 5, 6])

In []: c = np.array([7, 8, 9])

# Stacking arrays vertically
In []: np.vstack((a, b, c))
Out[]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

# Stacking arrays horizontally
In []: np.hstack((a, b, c))
Out[]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])

# Stacking two arrays together
In []: np.column_stack((a, b))
Out[]:
array([[1, 4],
       [2, 5],
       [3, 6]])

# Stacking three arrays together
In []: np.column_stack((a, b, c))
Out[]:
array([[1, 4, 7],
```

```
[2, 5, 8],  
[3, 6, 9]])
```

- `transpose()` permutes the dimensions of an array.

```
# Creating a two-dimensional array with shape (2, 4)  
In []: a = np.arange(8).reshape(2, 4)  
  
# Printing it  
In []: a  
Out[]:  
array([[0, 1, 2, 3],  
       [4, 5, 6, 7]])  
  
# Transposing the array  
In []: a.transpose()  
Out[]:  
array([[0, 4],  
       [1, 5],  
       [2, 6],  
       [3, 7]])
```

10.6 Array Indexing and Iterating

NumPy is an excellent library for efficient number crunching along with ease of use. It seamlessly integrates with Python and its syntax. Following this attribute, NumPy provides subsetting and iterating techniques very similar to lists. We can use square brackets to subset NumPy arrays, Python built-in constructs to iterate, and other built-in methods to slice them.

10.6.1 Indexing and Subsetting

NumPy arrays follow indexing structure similar to Python lists. Index starts with 0 and each element in an array is associated with a unique index. Below table shows NumPy indexing for a one-dimensional array.

Index	0	1	2	3	4
np.array	52	88	41	63	94

The index structure for a two-dimensional array with a shape of (3, 3) is shown below.

Index	0	1	2
0	a	b	c
1	d	e	f
2	g	h	i

The two arrays `arr_1d` and `arr_2d` which depicts the above-shown structure have been created below:

```
# Creating one-dimensional array
In []: arr_1d = np.array([52, 88, 41, 63, 94])

# Creating two-dimensional array
In []: arr_2d = np.array([[ 'a', 'b', 'c'],
...:                      [ 'd', 'e', 'f'],
...:                      [ 'g', 'h', 'i']])
```

We use square brackets `[]` to subset each element from NumPy arrays. Let us subset arrays created above using indexing.

```
# Slicing the element at index 0
In []: arr_1d[0]
Out[]: 52

# Slicing the last element using negative index
In []: arr_1d[-1]
Out[]: 94

# Slicing elements from position 2 (inclusive) to
# 5 (exclusive)
In []: arr_1d[2:5]
Out[]: array([41, 63, 94])
```

In the above examples, we sliced a one-dimensional array. Similarly, square brackets also allow slicing two-dimensional using the syntax `[r, c]` where `r` is a row and `c` is a column.

```

# Slicing the element at position (0, 1)
In []: arr_2d[0, 1]
Out[]: 'b'

# Slicing the element at position (1, 2)
In []: arr_2d[1, 2]
Out[]: 'f'

# Slicing the element at position (2, 0)
In []: arr_2d[2, 0]
Out[]: 'g'

# Slicing the first row
In []: arr_2d[0, :]
Out[]: array(['a', 'b', 'c'], dtype='<U1')

# Slicing the last column
In []: arr_2d[:, 2]
Out[]: array(['c', 'f', 'i'], dtype='<U1')

```

Notice the syntax in the last example where we slice the last column. The `:` has been provided as an input which denotes all elements and then filtering the last column. Using only `:` would return us all elements in the array.

```

In []: arr_2d[:, :]
Out[]:
array([['a', 'b', 'c'],
       ['d', 'e', 'f'],
       ['g', 'h', 'i']], dtype='<U1')

```

10.6.2 Boolean Indexing

NumPy arrays can be indexed with other arrays (or lists). The arrays used for indexing other arrays are known as *index arrays*. Mostly, it is a simple array which is used to subset other arrays. The use of index arrays ranges from simple, straightforward cases to complex and hard to understand cases. When an array is indexed using another array, a copy of the original data is returned, not a view as one gets for slices. To illustrate:

```

# Creating an array
In []: arr = np.arange(1, 10)

# Printing the array
In []: arr
Out[]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])

# Subsetting the array `arr` using an anonymous array
In []: arr[np.array([2, 5, 5, 1])]
Out[]: array([3, 6, 6, 2])

```

We create an array `arr` with ten elements in the above example. Then we try to subset it using an anonymous index array. The index array consisting of the values 2, 5, 5 and 1 correspondingly create an array of length 4, i.e. same as the length index array. Values in the index array work as an index to subset (in the above-given operation) and it simply returns the corresponding values from the `arr`.

Extending this concept, an array can be indexed with itself. Using logical operators, NumPy arrays can be filtered as desired. Consider a scenario, where we need to filter array values which are greater than a certain threshold. This is shown below:

```

# Creating an random array with 20 values
In []: rand_arr = np.random.randint(1, 50, 20)

# Printing the array
In []: rand_arr
Out[]:
array([14, 25, 39, 18, 40, 10, 33, 36, 29, 25, 27, 4, 28,
       43, 43, 19, 30, 29, 47, 41])

# Filtering the array values which are greater than 30
In []: rand_arr[rand_arr > 30]
Out[]: array([39, 40, 33, 36, 43, 43, 47, 41])

```

Here, we create an array with the name `rand_arr` with 20 random values. We then try to subset it with values which are greater than 30 using the logical operator `>`. When an array is being sliced using the logical operator,

NumPy generates an anonymous array of True and False values which is then used to subset the array. To illustrate this, let us execute the code used to subset the `rand_arr`, i.e. code written within the square brackets.

```
In []: filter_ = rand_arr > 30

In []: filter_
Out[]:
array([False, False,  True, False,  True, False,  True,
       True, False, False, False, False, False,  True,
       True, False, False, False,  True,  True])
```

It returned a boolean array with only True and False values. Here, True appears wherever the logical condition holds true. NumPy uses this outputted array to subset the original array and returns only those values where it is True.

Apart from this approach, NumPy provides a `where` method using which we can achieve the same filtered output. We pass a logical condition within `where` condition, and it will return an array with all values for which conditions stands true. We filter out all values greater than 30 from the `rand_arr` using the `where` condition in the following example:

```
# Filtering an array using np.where method
In []: rand_arr[np.where(rand_arr > 30)]
Out[]: array([39, 40, 33, 36, 43, 43, 47, 41])
```

We got the same result by executing `rand_arr[rand_arr > 30]` and `rand_arr[np.where(rand_arr > 30)]`. However, the `where` method provided by NumPy just do not filter values. Instead, it can be used for more versatile scenarios. Below given is the official syntax:

```
np.where[condition[, x, y]]
```

It returns the elements, either from `x` or `y`, depending on `condition`. As these parameters, `x` and `y` are optional, `condition` when true yields `x` if given or boolean `True`, otherwise `y` or boolean `False`.

Below we create an array `heights` that contains 20 elements with height ranging from 150 to 160 and look at various uses of `where` method.

```
# Creating an array heights
In []: heights = np.random.uniform(150, 160, size=20)

In []:
Out[]:
array([153.69911134, 154.12173942, 150.35772942,
       151.53160722, 153.27900307, 154.42448961,
       153.25276742, 151.08520803, 154.13922276,
       159.71336708, 151.45302507, 155.01280829,
       156.9504274 , 154.40626961, 155.46637317,
       156.36825413, 151.5096344 , 156.75707004,
       151.14597394, 153.03848597])
```

Usage 1: Without x and y parameters. Using the where method without the optional parameter as illustrated in the following example would return the index values of the original array where the condition is true.

```
In []: np.where(heights > 153)
Out[]:
(array([ 0,  1,  4,  5,  6,  8,  9, 11, 12, 13, 14, 15, 17, 19],
      dtype=int64),)
```

The above codes returned index values of the heights array where values are greater than 153. This scenario is very similar to the one we have seen above with the random array rand_arr where we tried to filter values above 30. Here, the output is merely the index values. If we want the original values, we need to subset the heights array using the output that we obtained.

Usage 2: With x as True and y as False. Having these optional parameters in place would return either of the parameters based on the condition. This is shown in the below example:

```
In []: np.where(heights > 153, True, False)
Out[]:
array([True, True, False, False, True, True, True, False,
       True, True, False, True, True, True, True, True,
       False, True, False, True])
```

The output in the *Usage 2* provides either True or False for all the elements in the heights array in contrast to the *Usage 1* where it returned index values of only those elements where the condition was true. The optional parameters can also be array like elements instead of scalars or static value such as True or False.

Usage 3: With x and y being arrays. Now that we have quite a good understanding of how the where method works, it is fairly easy to guess the output. The output will contain values from either x array or y array based on the condition in the first argument. For example:

```
# Creating an array 'x_array'  
In []: x_array = np.arange(11, 31, 1)  
  
In []: x_array  
Out[]:  
array([11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,  
     24, 25, 26, 27, 28, 29, 30])  
  
# Creating an array 'y_array'  
In []: y_array = np.arange(111, 131, 1)  
  
In []: y_array  
Out[]:  
array([111, 112, 113, 114, 115, 116, 117, 118, 119, 120,  
     121, 122, 123, 124, 125, 126, 127, 128, 129, 130])  
  
In []: np.where(heights > 153, x_array, y_array)  
Out[]:  
array([ 11,  12, 113, 114,  15,  16,  17, 118,  19,  20,  
     121,  22,  23,  24,  25,  26, 127,  28, 129,  30])
```

As expected, the output of the above code snippet contains values from the array x_array when the value in the heights array is greater than 153, otherwise, the value from the y_array will be outputted.

Having understood the working of where method provided by the NumPy library, let us now see how it is useful in back-testing strategies. Consider a scenario where we have all the required data for generating trading signals.

Data that we have for this hypothetical example is the close price of a stock for 20 periods and its average price.

```
# Hypothetical close prices for 20 periods
In []: close_price = np.random.randint(132, 140, 20)

# Printing close_price
In []: close_price
Out[]:
array([137, 138, 133, 132, 134, 139, 132, 138, 137, 135,
       136, 134, 134, 139, 135, 133, 136, 139, 132, 134])
```

We are to generate trading signals based on the buy condition given to us. i.e. we go long or buy the stock when the closing price is greater than the average price of 135.45. It can be easily computed using the `where` method as shown below:

```
# Average close price
In []: avg_price = 135.45

# Computing trading signals with 1 being 'buy' and 0
# represents 'no signal'
In []: signals = np.where(close_price > avg_price, 1, 0)

# Printing signals
In []: signals
Out[]: array([1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0,
       0, 1, 1, 0, 0])
```

The `signals` array contains the trading signals where 1 represents the buy and 0 represents no trading signal.

10.6.3 Iterating Over Arrays

NumPy arrays are iterable objects in Python which means that we can directly iterate over them using the `iter()` and `next()` methods as with any other iterable. This also implies that we can use built-in looping constructs to iterate over them. The following examples show iterating NumPy arrays using a `for` loop.

```
# Looping over a one-dimensional array
In []: for element in arr_1d:
....:     print(element)

# Output
52
88
41
63
94
```

Looping over a one-dimensional array is easy and straight forward. But, if we are to execute the `for` loop with `arr_2d`, it will be traverse all rows and provide that as the output. It is demonstrated in the following example.

```
In []: for element in arr_2d:
....:     print(element)

# Output
['a' 'b' 'c']
['d' 'e' 'f']
['g' 'h' 'i']
```

To iterate over two-dimensional arrays, we need to take care of both axes. A separate `for` loop can be used in a nested format to traverse through each element as shown below:

```
In []: for element in arr_2d:
....:     for e in element:
....:         print(e)

# Output
a
b
c
d
e
f
g
```

```
h  
i
```

The output that we got can also be achieved using `nditer()` method of NumPy, and it works for irrespective of dimensions.

```
In []: for element in np.nditer(arr_2d):  
....:     print(element)  
  
# Output  
a  
b  
c  
d  
e  
f  
g  
h  
i
```

This brings us to the end of a journey with the NumPy module. The examples provided above depicts only a minimal set of NumPy functionalities. Though not comprehensive, it should give us a pretty good feel about what is NumPy and why we should be using it.

10.7 Key Takeaways

1. NumPy library is used to perform scientific computing in Python.
2. It is not a part of the Python Standard Library and needs to be installed explicitly before it can be used in a program.
3. It allows creating n-dimensional arrays of the type `ndarray`.
4. NumPy arrays can hold elements of single data type only.
5. They can be created using any sequential data structures such as lists or tuples, or using built-in NumPy functions.
6. The random module of the NumPy library allows generating samples from various data distributions.
7. NumPy supports for element-wise operation using broadcast functionality.

8. Similar to lists, NumPy arrays can also be sliced using square brackets `[]` and starts indexing with 0.
9. It is also possible to slice NumPy arrays based on logical conditions. The resultant array would be an array of boolean True or False based on which other arrays are sliced or filtered. This is known as boolean indexing.

Chapter 11

Pandas Module

Pandas is a Python library to deal with sequential and tabular data. It includes many tools to manage, analyze and manipulate data in a convenient and efficient manner. We can think of its data structures as akin to database tables or spreadsheets.

Pandas is built on top of the Numpy library and has two primary data structures viz. Series (1-dimensional) and DataFrame (2-dimensional). It can handle both homogeneous and heterogeneous data, and some of its many capabilities are:

- ETL tools (Extraction, Transformation and Load tools)
- Dealing with missing data (NaN)
- Dealing with data files (csv, xls, db, hdf5, etc.)
- Time-series manipulation tools

In the Python ecosystem, Pandas is the best choice to retrieve, manipulate, analyze and transform financial data.

11.1 Pandas Installation

The official documentation¹ has a detailed explanation that spans over several pages on installing Pandas. We summarize it below.

¹<https://pandas.pydata.org/pandas-docs/stable/install.html>

11.1.1 Installing with pip

The simplest way to install Pandas is from PyPI.
In a terminal window, run the following command.

```
pip install pandas
```

In your code, you can use the escape character ‘!’ to install pandas directly from your Python console.

```
!pip install pandas
```

Pip is a useful tool to manage Python’s packages and it is worth investing some time in knowing it better.

```
pip help
```

11.1.2 Installing with Conda environments

For advanced users, who like to work with Python environments for each project, you can create a new environment and install pandas as shown below.

```
conda create -n EPAT python
source activate EPAT
conda install pandas
```

11.1.3 Testing Pandas installation

To check the installation, Pandas comes with a test suite to test almost all of the codebase and verify that everything is working.

```
import pandas as pd
pd.test()
```

11.2 What problem does Pandas solve?

Pandas works with homogeneous data series (1-Dimension) and heterogeneous tabular data series (2-Dimensions). It includes a multitude of tools to work with these data types, such as:

- Indexes and labels.
- Searching of elements.
- Insertion, deletion and modification of elements.
- Apply set techniques, such as grouping, joining, selecting, etc.
- Data processing and cleaning.
- Work with time series.
- Make statistical calculations
- Draw graphics
- Connectors for multiple data file formats, such as, csv, xlsx, hdf5, etc.

11.3 Pandas Series

The first data structure in Pandas that we are going to see is the Series. They are homogeneous one-dimensional objects, that is, all data are of the same type and are implicitly labeled with an index.

For example, we can have a Series of integers, real numbers, characters, strings, dictionaries, etc. We can conveniently manipulate these series performing operations like adding, deleting, ordering, joining, filtering, vectorized operations, statistical analysis, plotting, etc.

Let's see some examples of how to create and manipulate a Pandas Series:

- We will start by creating an empty Pandas Series:

```
import pandas as pd
s = pd.Series()
print(s)

Out[]: Series([], dtype: float64)
```

- Let's create a Pandas Series of integers and print it:

```
import pandas as pd
s = pd.Series([1, 2, 3, 4, 5, 6, 7])
print(s)

Out[]: 0    1
```

```
1    2  
2    3  
3    4  
4    5  
5    6  
6    7  
dtype: int64
```

- Let's create a Pandas Series of characters:

```
import pandas as pd  
s = pd.Series(['a', 'b', 'c', 'd', 'e'])  
print(s)
```

```
Out[]: 0    1  
       1    2  
       2    3  
       3    4  
       4    5  
       5    6  
       6    7  
dtype: int64
```

- Let's create a random Pandas Series of float numbers:

```
import pandas as pd  
import numpy as np  
s = pd.Series(np.random.randn(5))  
print(s)
```

```
Out[]: 0    0.383567  
       1    0.869761  
       2    1.100957  
       3   -0.259689  
       4    0.704537  
dtype: float64
```

In all these examples, we have allowed the index label to appear by default (without explicitly programming it). It starts at 0, and we can check the index as:

```
In []: s.index
```

```
Out[]: RangeIndex(start=0, stop=5, step=1)
```

But we can also specify the index we need, for example:

```
In []: s = pd.Series(np.random.randn(5),
                     index=['a', 'b', 'c', 'd', 'e'])
```

```
Out[]: a    1.392051
       b    0.515690
       c   -0.432243
       d   -0.803225
       e    0.832119
       dtype: float64
```

- Let's create a Pandas Series from a dictionary:

```
import pandas as pd
dictionary = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5}
s = pd.Series(dictionary)
print(s)
```

```
Out[]: a    1
       b    2
       c    3
       d    4
       e    5
       dtype: int64
```

In this case, the Pandas Series is created with the dictionary keys as index unless we specify any other index.

11.3.1 Simple operations with Pandas Series

When we have a Pandas Series, we can perform several simple operations on it. For example, let's create two Series. One from a dictionary and the other from an array of integers:

```

In []: import pandas as pd
        dictionary = {'a' : 1, 'b' : 2, 'c' : 3, 'd': 4,
                        'e': 5}
        s1 = pd.Series(dictionary)

        array = [1, 2, 3, 4, 5]
        s2 = pd.Series(array)

Out[]: a    1
       b    2
       c    3
       d    4
       e    5
      dtype: int64

       0    1
       1    2
       2    3
       3    4
       4    5
      dtype: int64

```

We can perform operations similar to Numpy arrays:

- Selecting one item from the Pandas Series by means of its index:

```

In []: s1[0] # Select the first element
Out[]: 1

```

```

In []: s1['a']
Out[]: 1

```

```

In []: s2[0]
Out[]: 1

```

- Selecting several items from the Pandas Series by means of its index:

```

In []: s1[[1, 4]]
Out[]: b    2

```

```
e      5  
dtype: int64
```

```
In []: s1[['b', 'e']]  
Out[]: b      2  
       e      5  
       dtype: int64
```

```
In []: s2[[1, 4]]  
Out[]: b      2  
       e      5  
       dtype: int64
```

- Get the series starting from an element:

```
In []: s1[2:]  
Out[]: c      3  
       d      4  
       e      5  
       dtype: int64
```

```
In []: s2[2:]  
Out[]: 2      3  
       3      4  
       4      5  
       dtype: int64
```

- Get the series up to one element:

```
In []: s1[:2]  
Out[]: c      3  
       d      4  
       e      5  
       dtype: int64
```

```
In []: s2[:2]  
Out[]: 2      3  
       3      4  
       4      5  
       dtype: int64
```

We can perform operations like a dictionary:

- Assign a value:

```
In []: s1[1] = 99  
       s1['a'] = 99
```

```
Out[]: a      1  
        b     99  
        c      3  
        d      4  
        e      5  
       dtype: int64
```

```
In []: s2[1] = 99  
       print(s2)
```

```
Out[]: 0      1  
        1     99  
        2      3  
        3      4  
        4      5  
       dtype: int64
```

- Get a value by index (like dictionary key):

```
In []: s.get('b')  
Out[]: 2
```

Here are some powerful vectorized operations that let us perform quickly calculations, for example:

- Add, subtract, multiply, divide, power, and almost any NumPy function that accepts NumPy arrays.

```
s1 + 2  
s1 - 2  
s1 * 2  
s1 / 2  
s1 ** 2  
np.exp(s1)
```

- We can perform the same operations over two Pandas Series although these must be aligned, that is, to have the same index, in other case, perform a Union operation.

```
In []: s1 + s1 # The indices are aligned
Out[]: a    2
        b    4
        c    6
        d    8
        e   10
       dtype: int64
```

```
In []: s1 + s2 # The indices are unaligned
Out[]: a    NaN
        b    NaN
        c    NaN
        d    NaN
        e    NaN
        0    NaN
        1    NaN
        2    NaN
        3    NaN
        4    NaN
       dtype: float64
```

11.4 Pandas DataFrame

The second data structure in Pandas that we are going to see is the DataFrame.

Pandas DataFrame is a heterogeneous two-dimensional object, that is, the data are of the same type within each column but it could be a different data type for each column and are implicitly or explicitly labeled with an index.

We can think of a DataFrame as a database table, in which we store heterogeneous data. For example, a DataFrame with one column for the first name, another for the last name and a third column for the phone

number, or a dataframe with columns to store the opening price, close price, high, low, volume, and so on.

The index can be implicit, starting with zero or we can specify it ourselves, even working with dates and times as indexes as well. Let's see some examples of how to create and manipulate a Pandas DataFrame.

- Creating an empty DataFrame:

```
In []: import pandas as pd  
       s = pd.DataFrame()  
       print(s)
```

```
Out[]: Empty DataFrame  
       Columns: []  
       Index: []
```

- Creating an empty structure DataFrame:

```
In []: import pandas as pd  
       s = pd.DataFrame(columns=['A', 'B', 'C', 'D', 'E'])  
       print(s)
```

```
Out[]: Empty DataFrame  
       Columns: [A, B, C, D, E]  
       Index: []
```

```
In []: import pandas as pd  
       s = pd.DataFrame(columns=['A', 'B', 'C', 'D', 'E'],  
                         index=range(1, 6))  
       print(s)
```

```
Out[]:      A      B      C      D      E  
1    NaN    NaN    NaN    NaN    NaN  
2    NaN    NaN    NaN    NaN    NaN  
3    NaN    NaN    NaN    NaN    NaN  
4    NaN    NaN    NaN    NaN    NaN  
5    NaN    NaN    NaN    NaN    NaN
```

- Creating a DataFrame passing a NumPy array:

```
In []: array = {'A' : [1, 2, 3, 4],  
               'B' : [4, 3, 2, 1]}
```

```
pd.DataFrame(array)
```

```
Out[]:   A    B  
0 1    4  
1 2    3  
2 3    2  
3 4    1
```

- Creating a DataFrame passing a NumPy array, with datetime index:

```
In []: import pandas as pd  
array = {'A': [1, 2, 3, 4], 'B': [4, 3, 2, 1]}  
index = pd.DatetimeIndex(['2018-12-01',  
                           '2018-12-02',  
                           '2018-12-03',  
                           '2018-12-04'])  
pd.DataFrame(array, index=index)
```

```
Out[]:          A    B  
2018-12-01  1    4  
2018-12-02  2    3  
2018-12-03  3    2  
2018-12-04  4    1
```

- Creating a DataFrame passing a Dictionary:

```
In []: import pandas as pd  
dictionary = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}  
pd.DataFrame([dictionary])
```

```
Out[]:   a    b    c    d    e  
0 1    2    3    4    5
```

- Viewing a DataFrame: We can use some methods to explore the Pandas DataFrame:

First, we go to create a Pandas DataFrame to work with it.

```
In []: import pandas as pd  
pd.DataFrame({'A':np.random.randn(10),  
              'B':np.random.randn(10),  
              'C':np.random.randn(10)})
```

```
Out[]:      A          B          C  
0   0.164358  1.689183  1.745963  
1  -1.830385  0.035618  0.047832  
2   1.304339  2.236809  0.920484  
3   0.365616  1.877610 -0.287531  
4  -0.741372 -1.443922 -1.566839  
5  -0.119836 -1.249112 -0.134560  
6  -0.848425 -0.569149 -1.222911  
7  -1.172688  0.515443  1.492492  
8   0.765836  0.307303  0.788815  
9   0.761520 -0.409206  1.298350
```

- Get the first three rows:

```
In []: import pandas as pd  
df=pd.DataFrame({'A':np.random.randn(10),  
                  'B':np.random.randn(10),  
                  'C':np.random.randn(10)})  
df.head(3)
```

```
Out[]:      A          B          C  
0   0.164358  1.689183  1.745963  
1  -1.830385  0.035618  0.047832  
2   1.304339  2.236809  0.920484
```

- Get the last three rows:

```
In []: import pandas as pd  
df=pd.DataFrame({'A':np.random.randn(10),  
                  'B':np.random.randn(10),  
                  'C':np.random.randn(10)})  
df.tail(3)
```

```
Out[]:      A          B          C
    7 -1.172688  0.515443  1.492492
    8  0.765836  0.307303  0.788815
    9  0.761520 -0.409206  1.298350
```

- Get the DataFrame's index:

```
In []: import pandas as pd
        df=pd.DataFrame({'A':np.random.randn(10),
                          'B':np.random.randn(10),
                          'C':np.random.randn(10)})
        df.index
```

```
Out[]: RangeIndex(start=0, stop=10, step=1)
```

- Get the DataFrame's columns:

```
In []: import pandas as pd
        df=pd.DataFrame({'A':np.random.randn(10),
                          'B':np.random.randn(10),
                          'C':np.random.randn(10)})
        df.columns
```

```
Out[]: Index(['A', 'B', 'C'], dtype='object')
```

- Get the DataFrame's values:

```
In []: import pandas as pd
        df=pd.DataFrame({'A':np.random.randn(10),
                          'B':np.random.randn(10),
                          'C':np.random.randn(10)})
        df.values
```

```
Out[]: array([[ 0.6612966 , -0.60985049,  1.11955054],
               [-0.74105636,  1.42532491, -0.74883362],
               [ 0.10406892,  0.5511436 ,  2.63730671],
               [-0.73027121, -0.11088373, -0.19143175],
               [ 0.11676573,  0.27582786, -0.38271609],
               [ 0.51073858, -0.3313141 ,  0.20516165],
```

```
[ 0.23917755,  0.55362   , -0.62717194] ,  
[ 0.25565784, -1.4960713 ,  0.58886377] ,  
[ 1.20284041,  0.21173483,  2.0331718 ] ,  
[ 0.62247283,  2.18407105,  0.02431867]])
```

11.5 Importing data in Pandas

Pandas DataFrame is able to read several data formats, some of the most used are: CSV, JSON, Excel, HDF5, SQL, etc.

11.5.1 Importing data from CSV file

One of the most useful functions is `read_csv` that allows us to read csv files with almost any format and load it into our DataFrame to work with it. Let's see how to work with csv files:

```
import pandas as pd  
df=pd.read_csv('Filename.csv')  
type(df)  
  
Out[]: pandas.core.frame.DataFrame
```

This simple operation, loads the csv file into the Pandas DataFrame after which we can explore it as we have seen before.

11.5.2 Customizing pandas import

Sometimes the format of the csv file come with a particular separator or we need specific columns or rows. We will now see some ways to deal with this.

In this example, we want to load a csv file with blank space as separator:

```
import pandas as pd  
df=pd.read_csv('Filename.csv', sep=' ')
```

In this example, we want to load columns from 0 and 5 and the first 100 rows:

```
import pandas as pd
df=pd.read_csv('Filename.csv', usecols=[0, 1, 2, 3, 4, 5],
               nrows=100)
```

It's possible to customize the headers, convert the columns or rows names and carry out a good number of other operations.

11.5.3 Importing data from Excel files

In the same way that we have worked with csv files, we can work with Excel file with the `read_excel` function, let's see some examples:

In this example, we want to load the sheet 1 from an Excel file:

```
import pandas as pd
df=pd.read_excel('Filename.xls', sheet_name='Sheet1')
```

This simple operation, loads the Sheet 1 from the Excel file into the Pandas DataFrame.

11.6 Indexing and Subsetting

Once we have the Pandas DataFrame prepared, independent of the source of our data (csv, Excel, hdf5, etc.) we can work with it, as if it were a database table, selecting the elements that interest us. We will work with some examples on how to index and extract subsets of data.

Let's begin with loading a csv file having details of a market instrument.

```
In []: import pandas as pd
        df=pd.read_csv('MSFT.csv',
                        usecols=[0, 1, 2, 3, 4])
        df.head()
        df.shape
```

```
Out []:    Date      Open      High      Low      Close
          0   2008-12-29  19.15    19.21   18.64   18.96
          1   2008-12-30  19.01    19.49   19.00   19.34
          2   2008-12-31  19.31    19.68   19.27   19.44
```

```
3    2009-01-02  19.53   20.40   19.37   20.33
4    2009-01-05  20.20   20.67   20.06   20.52
(1000, 5)
```

Here, we have read a csv file, of which we only need the columns of date, opening, closing, high and low (the first 5 columns) and we check the form of the DataFrame that has 1000 rows and 5 columns.

11.6.1 Selecting a single column

In the previous code, we have read directly the first 5 columns from the csv file. This is a filter that we applied, because we were only interested in those columns.

We can apply selection filters to the DataFrame itself, to select one column to work with. For example, we could need the Close column:

```
In []: close=df['Close']
       close.head()

Out[]:    Close
0    18.96
1    19.34
2    19.44
3    20.33
4    20.52
```

11.6.2 Selecting multiple columns

We can select multiple columns too:

```
In []: closevol=df[['Close', 'Volume']]
       closevol.head()

Out[]:    Close    Volume
0    18.96  58512800.0
1    19.34  43224100.0
2    19.44  46419000.0
```

```
3    20.33    50084000.0
4    20.52    61475200.0
```

11.6.3 Selecting rows via []

We can select a set of rows by index:

```
In []: import pandas as pd
        df=pd.read_csv('TSLA.csv' )
        df[100:110]
Out[] :
```

	Date	Open	...	AdjVolume	Name
100	2017-10-30	319.18	...	4236029.0	TSLA
101	2017-10-27	319.75	...	6942493.0	TSLA
102	2017-10-26	327.78	...	4980316.0	TSLA
103	2017-10-25	336.70	...	8547764.0	TSLA
104	2017-10-24	338.80	...	4463807.0	TSLA
105	2017-10-23	349.88	...	5715817.0	TSLA
106	2017-10-20	352.69	...	4888221.0	TSLA
107	2017-10-19	355.56	...	5032884.0	TSLA
108	2017-10-18	355.97	...	4898808.0	TSLA
109	2017-10-17	350.91	...	3280670.0	TSLA

Or we can select a set of rows and columns:

```
In []: df[100:110][['Close', 'Volume']]
```

```
Out[] :
      Close    Volume
100 320.08  4236029.0
101 320.87  6942493.0
102 326.17  4980316.0
103 325.84  8547764.0
104 337.34  4463807.0
105 337.02  5715817.0
106 345.10  4888221.0
```

```
107 351.81 5032884.0  
108 359.65 4898808.0  
109 355.75 3280670.0
```

11.6.4 Selecting via .loc[] (By label)

With df.loc we can do the same selections using labels:

To select a set of rows, we can code the following using the index number as label:

```
In []: df.loc[100:110]  
Out []:
```

	Date	Open	...	AdjVolume	Name
100	2017-10-30	319.18	...	4236029.0	TSLA
101	2017-10-27	319.75	...	6942493.0	TSLA
102	2017-10-26	327.78	...	4980316.0	TSLA
103	2017-10-25	336.70	...	8547764.0	TSLA
104	2017-10-24	338.80	...	4463807.0	TSLA
105	2017-10-23	349.88	...	5715817.0	TSLA
106	2017-10-20	352.69	...	4888221.0	TSLA
107	2017-10-19	355.56	...	5032884.0	TSLA
108	2017-10-18	355.97	...	4898808.0	TSLA
109	2017-10-17	350.91	...	3280670.0	TSLA

Or we can select a set of rows and columns like before:

```
In []: df.loc[100:110, ['Close', 'Volume']]  
Out []:  
      Close    Volume  
100 320.08 4236029.0  
101 320.87 6942493.0  
102 326.17 4980316.0  
103 325.84 8547764.0  
104 337.34 4463807.0  
105 337.02 5715817.0  
106 345.10 4888221.0
```

```
107 351.81 5032884.0  
108 359.65 4898808.0  
109 355.75 3280670.0  
110 350.60 5353262.0
```

11.6.5 Selecting via `.iloc[]` (By position)

With `df.iloc` we can do the same selections using integer position:

```
In []: df.iloc[100:110]  
Out[]:
```

	Date	Open	...	AdjVolume	Name
100	2017-10-30	319.18	...	4236029.0	TSLA
101	2017-10-27	319.75	...	6942493.0	TSLA
102	2017-10-26	327.78	...	4980316.0	TSLA
103	2017-10-25	336.70	...	8547764.0	TSLA
104	2017-10-24	338.80	...	4463807.0	TSLA
105	2017-10-23	349.88	...	5715817.0	TSLA
106	2017-10-20	352.69	...	4888221.0	TSLA
107	2017-10-19	355.56	...	5032884.0	TSLA
108	2017-10-18	355.97	...	4898808.0	TSLA
109	2017-10-17	350.91	...	3280670.0	TSLA

In the last example, we used the index as an integer position rather than by label.

We can select a set of rows and columns like before:

```
In []: df.iloc[100:110, [3, 4]]  
Out[]:  
      Low     Close  
100 317.25 320.08  
101 316.66 320.87  
102 323.20 326.17  
103 323.56 325.84  
104 336.16 337.34  
105 336.25 337.02
```

```
106 344.34 345.10
107 348.20 351.81
108 354.13 359.65
109 350.07 355.75
```

11.6.6 Boolean indexing

So far, we have sliced subsets of data by label or by position. Now let's see how to select data that meet some criteria. We do this with Boolean indexing. We can use the same criteria similar to what we have seen with Numpy arrays. We show you just two illustrative examples here. This is by no means enough to get comfortable with it and so would encourage you to check the documentation and further readings at the end of this chapter to learn more.

- We can filter data that is greater (less) than a number.

```
In []: df[df.Close > 110]
Out []:
```

	Date	Open	...	Close	...	Name
0	2017-03-27	304.00	...	279.18	...	TSLA
1	2017-03-26	307.34	...	304.18	...	TSLA
2	2017-03-23	311.25	...	301.54	...	TSLA

1080	2017-12-09	137.00	...	141.60	...	TSLA
1081	2017-12-06	141.51	...	137.36	...	TSLA
1082	2013-12-05	140.51	...	140.48	...	TSLA

1083 rows × 14 columns

```
In []: df[(df['Close'] > 110) | (df['Close'] < 120)]
Out []:
```

	Date	Open	...	Close	...	Name
0	2017-03-27	304.00	...	279.18	...	TSLA

	Date	Open	...	Close	...	Name
1	2017-03-26	307.34	...	304.18	...	TSLA
2	2017-03-23	311.25	...	301.54	...	TSLA
...
1080	2017-12-09	137.00	...	141.60	...	TSLA
1081	2017-12-06	141.51	...	137.36	...	TSLA
1082	2013-12-05	140.51	...	140.48	...	TSLA

1083 rows × 14 columns

11.7 Manipulating a DataFrame

When we are working with data, the most common structure is the DataFrame. Until now we have seen how to create them, make selections and find data. We are now going to see how to manipulate the DataFrame to transform it into another DataFrame that has the form that our problem requires.

We'll see how to sort it, re-index it, eliminate unwanted (or spurious) data, add or remove columns and update values.

11.7.1 Transpose using .T

The Pandas DataFrame transpose function T allows us to transpose the rows as columns, and logically the columns as rows:

```
In []: import pandas as pd
df=pd.read_csv('TSLA.csv')
df2=df[100:110][['Close', 'Volume']]

df2.T

Out[] :
```

	100	101	102	...	109
Close	320.08	320.87	326.17	...	355.75

	100	101	102	...	109
Volume	4236029.00	6942493.00	4980316.00	...	3280670.00

11.7.2 The `.sort_index()` method

When we are working with Pandas Dataframe it is usual to add or remove rows, order by columns, etc. That's why it's important to have a function that allows us to easily and comfortably sort the DataFrame by its index. We do this with the `sort_index` function of Pandas DataFrame.

```
In []: df.sort_index()
Out[]:
```

	Date	Open	High	Low	Close	...	Name
0	2017-03-27	304.00	304.27	277.18	279.18	...	TSLA
1	2017-03-26	307.34	307.59	291.36	304.18	...	TSLA
2	2017-03-23	311.25	311.61	300.45	301.54	...	TSLA
...

11.7.3 The `.sort_values()` method

Sometimes, we may be interested in sorting the DataFrame by some column or even with several columns as criteria. For example, sort the column by first names and the second criterion by last names. We do this with the `sort_values` function of Pandas DataFrame.

```
In []: df.sort_values(by='Close')
Out[]:
```

	Date	Open	High	Low	Close	...	Name
1081	2013-12-06	141.51	142.49	136.30	137.36	...	TSLA
1057	2014-01-13	145.78	147.00	137.82	139.34	...	TSLA
1078	2013-12-11	141.88	143.05	139.49	139.65	...	TSLA
...

```
In []: df.sort_values(by=['Open', 'Close'])
Out []:
```

	Date	Open	High	Low	Close	...	Name
1080	2013-12-09	137.00	141.70	134.21	141.60	...	TSLA
1077	2014-12-12	139.70	148.24	138.53	147.47	...	TSLA
1079	2013-12-10	140.05	145.87	139.86	142.19	...	TSLA

11.7.4 The .reindex() function

The Pandas' reindex funtion let us to realign the index of the Series or DataFrame, it's useful when we need to reorganize the index to meet some criteria. For example, we can play with the Series or DataFrame that we create before to alter the original index. For example, when the index is a label, we can reorganize as we need:

```
In []: import pandas as pd
        import numpy as np
        df=pd.DataFrame(np.random.randn(5),
                        index=['a', 'b', 'c', 'd', 'e'])
        df
Out []:
```

	0
a	-0.134133
b	-0.586051
c	1.179358
d	0.433142
e	-0.365686

Now, we can reorganize the index as follows:

```
In []: df.reindex(['b', 'a', 'd', 'c', 'e'])
Out []:
```

	0
b	-0.586051
a	-0.134133
d	0.433142
c	1.179358
e	-0.365686

When the index is numeric we can use the same function to order by hand the index:

```
In []: import pandas as pd
        import numpy as np
        df = pd.DataFrame(np.random.randn(5))
        df.reindex([4,3,2,1,0])
Out[] :
```

	0
4	1.058589
3	1.194400
2	-0.645806
1	0.836606
0	1.288102

Later in this section, we'll see how to work and reorganize date and time indices.

11.7.5 Adding a new column

Another interesting feature of DataFrames is the possibility of adding new columns to an existing DataFrame.

For example, we can add a new column to the random DataFrame that we have created before:

```
In []: import pandas as pd
        import numpy as np
        df = pd.DataFrame(np.random.randn(5))
Out[] :
```

	0
0	0.238304
1	2.068558
2	1.015650
3	0.506208
4	0.214760

To add a new column, we only need to include the new column name in the DataFrame and assign a initialization value, or assign to the new column a Pandas Series or another column from other DataFrame.

```
In []: df['new']=1  
       df  
Out[]:
```

	0	new
0	0.238304	1
1	2.068558	1
2	1.015650	1
3	0.506208	1
4	0.214760	1

11.7.6 Delete an existing column

Likewise, we can remove one or more columns from the DataFrame. Let's create a DataFrame with 5 rows and 4 columns with random values to delete one column.

```
In []: import pandas as pd  
       import numpy as np  
       df = pd.DataFrame(np.random.randn(5, 4))  
       df  
Out[]:
```

	0	1	2	3
0	-1.171562	-0.086348	-1.971855	1.168017
1	-0.408317	-0.061397	-0.542212	-1.412755
2	-0.365539	-0.587147	1.494690	1.756105
3	0.642882	0.924202	0.517975	-0.914366
4	0.777869	-0.431151	-0.401093	0.145646

Now, we can delete the column that we specify by index or by label if any:

```
In []: del df[0]
Out[]:
```

	1	2	3
0	-0.086348	-1.971855	1.168017
1	-0.061397	-0.542212	-1.412755
2	-0.587147	1.494690	1.756105
3	0.924202	0.517975	-0.914366
4	-0.431151	-0.401093	0.145646

```
In []: df['new']=1
df
Out[]:
```

	0	new
0	0.238304	1
1	2.068558	1
2	1.015650	1
3	0.506208	1

```
In []: del df['new']
Out[]:
```

	0
0	0.238304
1	2.068558
2	1.015650
3	0.506208

11.7.7 The .at[] (By label)

With at we can to locate a specific value by row and column labels as follows:

```
In []: import pandas as pd
        import numpy as np
        df=pd.DataFrame(np.random.randn(5,4),
                        index=['a', 'b', 'c', 'd', 'e'],
                        columns=['A', 'B', 'C', 'D'])
        print(df)
Out[]:
```

	A	B	C	D
a	0.996496	-0.165002	0.727912	0.564858
b	-0.388169	1.171039	-0.231934	-1.124595
c	-1.385129	0.299195	0.573570	-1.736860
d	1.222447	-0.312667	0.957139	-0.054156
e	1.188335	0.679921	1.508681	-0.677776

```
In []: df.at['a', 'A']
Out[]: 0.9964957014209125
```

It is possible to assign a new value with the same funcion too:

```
In []: df.at['a', 'A'] = 0
Out[]:
```

	A	B	C	D
a	0.000000	-0.165002	0.727912	0.564858
b	-0.388169	1.171039	-0.231934	-1.124595
c	-1.385129	0.299195	0.573570	-1.736860
d	1.222447	-0.312667	0.957139	-0.054156
e	1.188335	0.679921	1.508681	-0.677776

11.7.8 The .iat[] (By position)

With iat we can to locate a specific value by row and column index as follow:

```
In []: import pandas as pd
        import numpy as np
df=pd.DataFrame(np.random.randn(5, 4),
                index=['a', 'b', 'c', 'd', 'e'],
                columns=['A', 'B', 'C', 'D'])
print(df)
df.iat[0, 0]
```

Out []: 0.996496

It is possible to assign a new value with the same funcion too:

```
In []: df.iat[0, 0] = 0
Out []:
```

	A	B	C	D
a	0.000000	-0.165002	0.727912	0.564858
b	-0.388169	1.171039	-0.231934	-1.124595
c	-1.385129	0.299195	0.573570	-1.736860
d	1.222447	-0.312667	0.957139	-0.054156
e	1.188335	0.679921	1.508681	-0.677776

11.7.9 Conditional updating of values

Another useful function is to update values that meet some criteria, for example, update values whose values are greater than 0:

```
In []: import pandas as pd
        import numpy as np
        df=pd.DataFrame(np.random.randn(5, 4),
                        index=['a', 'b', 'c', 'd', 'e'],
                        columns=['A', 'B', 'C', 'D'])
        print(df)

        df[df > 0] = 1
        df
```

Out [] :

	A	B	C	D
a	1.000000	-0.082466	1.000000	-0.728372
b	-0.784404	-0.663096	-0.595112	1.000000
c	-1.460702	-1.072931	-0.761314	1.000000
d	1.000000	1.000000	1.000000	-0.302310
e	-0.488556	1.000000	-0.798716	-0.590920

We can also update the values of a specific column that meet some criteria, or even work with several columns as criteria and update a specific column.

```
In []: df['A'][df['A'] < 0] = 1
        print(df)
```

Out [] :

	A	B	C	D
a	1.0	-0.082466	1.000000	-0.728372
b	1.0	-0.663096	-0.595112	1.000000
c	1.0	-1.072931	-0.761314	1.000000

	A	B	C	D
d	1.0	1.000000	1.000000	-0.302310
e	1.0	1.000000	-0.798716	-0.590920

```
In []: df['A'][(df['B'] < 0) & (df['C'] < 0)] = 9
print(df)
```

Out[]:

	A	B	C	D
a	1.0	-0.082466	1.000000	-0.728372
b	9.0	-0.663096	-0.595112	1.000000
c	9.0	-1.072931	-0.761314	1.000000
d	1.0	1.000000	1.000000	-0.302310
e	1.0	1.000000	-0.798716	-0.590920

11.7.10 The .dropna() method

Occasionally, we may have a DataFrame that, for whatever reason, includes NA values. This type of values is usually problematic when we are making calculations or operations and must be treated properly before proceeding with them. The easiest way to eliminate NA values is to remove the row that contains it.

```
In []: import pandas as pd
import numpy as np
df=pd.DataFrame(np.random.randn(5, 4),
                 index=['a', 'b', 'c', 'd', 'e'],
                 columns=['A', 'B', 'C', 'D'])
print(df)
```

Out[]:

	A	B	C	D
a	1.272361	1.799535	-0.593619	1.152889
b	-0.318368	-0.190419	0.129420	1.551332

	A	B	C	D
c	0.166951	1.669034	-1.653618	0.656313
d	0.219999	0.951074	0.442325	-0.170177
e	0.312319	-0.765930	-1.641234	-1.388924

```
In []: df['A'][(df['B'] < 0) & (df['C'] < 0)] = np.nan
print(df)
```

Out [] :

	A	B	C	D
a	1.272361	1.799535	-0.593619	1.152889
b	-0.318368	-0.190419	0.129420	1.551332
c	0.166951	1.669034	-1.653618	0.656313
d	0.219999	0.951074	0.442325	-0.170177
e	NaN	-0.765930	-1.641234	-1.388924

```
In []: df=df.dropna()
print(df)
```

Out [] :

	A	B	C	D
a	1.272361	1.799535	-0.593619	1.152889
b	-0.318368	-0.190419	0.129420	1.551332
c	0.166951	1.669034	-1.653618	0.656313
d	0.219999	0.951074	0.442325	-0.170177

Here we are deleting the whole row that has, in any of its columns, a NaN value, but we can also specify that it deletes the column that any of its values is NaN:

```
df=df.dropna(axis=1)
print(df)
```

We can specify if a single NaN value is enough to delete the row or column, or if the whole row or column must have NaN to delete it.

```
python python df=df.dropna(how='all') print(df) "
```

11.7.11 The .fillna() method

With the previous function we have seen how to eliminate a complete row or column that contains one or all the values to NaN, this operation can be a little drastic if we have valid values in the row or column.

For this, it is interesting to use the `fillna` function that substitutes the NaN values with some fixed value.

```
In []: import pandas as pd
        import numpy as np
        df=pd.DataFrame(np.random.randn(5, 4),
                        index=['a', 'b', 'c', 'd', 'e'],
                        columns=['A', 'B', 'C', 'D'])
        print(df)
Out[]:
```

	A	B	C	D
a	1.272361	1.799535	-0.593619	1.152889
b	-0.318368	-0.190419	0.129420	1.551332
c	0.166951	1.669034	-1.653618	0.656313
d	0.219999	0.951074	0.442325	-0.170177
e	0.312319	-0.765930	-1.641234	-1.388924

```
In []: df['A'][ (df['B'] < 0) & (df['C'] < 0)] = np.nan
        print(df)
Out[]:
```

	A	B	C	D
a	1.272361	1.799535	-0.593619	1.152889
b	-0.318368	-0.190419	0.129420	1.551332
c	0.166951	1.669034	-1.653618	0.656313
d	0.219999	0.951074	0.442325	-0.170177

	A	B	C	D
e	NaN	-0.765930	-1.641234	-1.388924

```
In []: df=df.fillna(999)
```

```
    print(df)
```

```
Out[] :
```

	A	B	C	D
a	1.272361	1.799535	-0.593619	1.152889
b	-0.318368	-0.190419	0.129420	1.551332
c	0.166951	1.669034	-1.653618	0.656313
d	0.219999	0.951074	0.442325	-0.170177
e	999	-0.765930	-1.641234	-1.388924

11.7.12 The .apply() method

The apply is a very useful way to use functions or methods in a DataFrame without having to loop through it. We can apply the apply method to a Series or DataFrame to apply a function to all rows or columns of the DataFrame. Let's see some examples.

Suppose we are working with the randomly generated DataFrame and need to apply a function. In this example, for simplicity's sake, we're going to create a custom function to square a number.

```
In []: import pandas as pd
        import numpy as np
        df=pd.DataFrame(np.random.randn(5, 4),
                        index=['a', 'b', 'c', 'd', 'e'],
                        columns=['A', 'B', 'C', 'D'])
        print(df)
```

```
Out[] :
```

	A	B	C	D
a	-0.633249	-2.699088	0.574052	0.652742

	A	B	C	D
b	0.060295	-0.150527	0.149123	-0.701216
c	-0.052515	0.469481	0.899180	-0.608409
d	-1.352912	0.103302	0.457878	-1.897170
e	0.088279	0.418317	-1.102989	0.582455

```
def square_number(number):
    return number**2

# Test the function
In []: square_number(2)
Out []: 4
```

Now, let's use the custom function through Apply:

```
In []: df.apply(square_number, axis=1)
Out []:
```

	A	B	C	D
a	0.401005	7.285074	0.329536	0.426073
b	0.003636	0.022658	0.022238	0.491704
c	0.002758	0.220412	0.808524	0.370161
d	1.830372	0.010671	0.209652	3.599253
e	0.007793	0.174989	1.216586	0.339254

This method apply the funcion square_number to all rows of the DataFrame.

11.7.13 The .shift() function

The shift function allows us to move a row to the right or left and/or to move a column up or down. Let's look at some examples.

First, we are going to move the values of a column downwards:

```
In []: import pandas as pd
        import numpy as np
```

```

df=pd.DataFrame(np.random.randn(5, 4),
                index=['a', 'b', 'c', 'd', 'e'],
                columns=['A', 'B', 'C', 'D'])

print(df)
Out[]:

```

	A	B	C	D
a	-0.633249	-2.699088	0.574052	0.652742
b	0.060295	-0.150527	0.149123	-0.701216
c	-0.052515	0.469481	0.899180	-0.608409
d	-1.352912	0.103302	0.457878	-1.897170
e	0.088279	0.418317	-1.102989	0.582455

```

In []: df['D'].shift(1)
Out[]:

```

	A	B	C	D
a	-0.633249	-2.699088	0.574052	NaN
b	0.060295	-0.150527	0.149123	0.652742
c	-0.052515	0.469481	0.899180	-0.701216
d	-1.352912	0.103302	0.457878	-0.608409
e	0.088279	0.418317	-1.102989	-1.897170

We are going to move the values of a column upwards

```

In []: df['shift'] = df['D'].shift(-1)
Out[]:

```

	A	B	C	D
a	-0.633249	-2.699088	0.574052	-0.701216
b	0.060295	-0.150527	0.149123	-0.608409
c	-0.052515	0.469481	0.899180	-1.897170
d	-1.352912	0.103302	0.457878	0.582455
e	0.088279	0.418317	-1.102989	NaN

This is very useful for comparing the current value with the previous value.

11.8 Statistical Exploratory data analysis

Pandas DataFrame allows us to make some descriptive statistics calculations, which are very useful to make a first analysis of the data we are handling. Let's see some useful functions.

11.8.1 The info() function

It is a good practice to know the structure and format of our DataFrame, the Info function offers us just that:

```
In []: import pandas as pd
        import numpy as np
        df=pd.DataFrame(np.random.randn(5, 4),
                        index=['a', 'b', 'c', 'd', 'e'],
                        columns=['A', 'B', 'C', 'D'])
        print(df)
Out[]:
```

	A	B	C	D
a	-0.633249	-2.699088	0.574052	0.652742
b	0.060295	-0.150527	0.149123	-0.701216
c	-0.052515	0.469481	0.899180	-0.608409
d	-1.352912	0.103302	0.457878	-1.897170
e	0.088279	0.418317	-1.102989	0.582455

```
In []: df.info()
Out[]: <class 'pandas.core.frame.DataFrame'>
Index: 5 entries, a to e
Data columns (total 5 columns):
 A      5 non-null float64
 B      5 non-null float64
 C      5 non-null float64
```

```
D      5 non-null float64
shift  4 non-null float64
dtypes: float64(5)
memory usage: 240.0+ bytes
```

11.8.2 The describe() function

We can obtain a statistical overview of the DataFrame with the ‘describe’ function, which gives us the mean, median, standard deviation, maximum, minimum, quartiles, etc. of each DataFrame column.

```
In []: import pandas as pd
        import numpy as np
        df=pd.DataFrame(np.random.randn(5, 4),
                        index=['a', 'b', 'c', 'd', 'e'],
                        columns=['A', 'B', 'C', 'D'])
        print(df)
Out[]:
```

	A	B	C	D
a	-0.633249	-2.699088	0.574052	0.652742
b	0.060295	-0.150527	0.149123	-0.701216
c	-0.052515	0.469481	0.899180	-0.608409
d	-1.352912	0.103302	0.457878	-1.897170
e	0.088279	0.418317	-1.102989	0.582455

```
In []: df.describe()
```

```
Out[]:
```

	A	B	C	D
count	5.000000	5.000000	5.000000	5.000000
mean	-0.378020	-0.371703	0.195449	-0.394319
std	0.618681	1.325046	0.773876	1.054633
min	-1.352912	-2.699088	-1.102989	-1.897170
25%	-0.633249	-0.150527	0.149123	-0.701216
50%	-0.052515	0.103302	0.457878	-0.608409

	A	B	C	D
75%	0.060295	0.418317	0.574052	0.582455
max	0.088279	0.469481	0.899180	0.652742

11.8.3 The value_counts() function

The function `value_counts` counts the repeated values of the specified column:

```
In []: df['A'].value_counts()
Out[]: 0.088279      1
       -0.052515     1
       0.060295      1
       -0.633249     1
       -1.352912     1
Name: A, dtype: int64
```

11.8.4 The mean() function

We can obtain the mean of a specific column or row by means of the `mean` function.

```
In []: df['A'].mean() # Specifying a column
Out[]: -0.3780203497252693

In []: df.mean() # By column
       df.mean(axis=0) # By column

Out[]: A      -0.378020
       B      -0.371703
       C      0.195449
       D      -0.394319
       shift -0.638513
Name: float64

In []: df.mean(axis=1) # By row
Out[]: a      -0.526386
```

```
b      0.002084  
c      0.001304  
d     -0.659462  
e     -0.382222  
dtype: float64
```

11.8.5 The std() function

We can obtain the standard deviation of a specific column or row by means of the `std` function.

```
In []: df['A'].std() # Specifying a column  
Out[]: 0.6186812554819784  
  
In []: df.std() # By column  
       df.std(axis=0) # By column  
  
Out[]: A      0.618681  
       B      1.325046  
       C      0.773876  
       D      1.054633  
       shift  1.041857  
       dtype: float64  
  
In []: df.std(axis=1) # By row  
  
Out[]: a      1.563475  
       b      0.491499  
       c      0.688032  
       d      0.980517  
       e      1.073244  
       dtype: float64
```

11.9 Filtering Pandas DataFrame

We have already seen how to filter data in a DataFrame, including logical statements to filter rows or columns with some logical criteria. For example, we will filter rows whose column 'A' is greater than zero:

```
In []: import pandas as pd
        import numpy as np
        df=pd.DataFrame(np.random.randn(5, 4),
                        index=['a', 'b', 'c', 'd', 'e'],
                        columns=['A', 'B', 'C', 'D'])
        print(df)
Out[] :
```

	A	B	C	D
a	-0.633249	-2.699088	0.574052	0.652742
b	0.060295	-0.150527	0.149123	-0.701216
c	-0.052515	0.469481	0.899180	-0.608409
d	-1.352912	0.103302	0.457878	-1.897170
e	0.088279	0.418317	-1.102989	0.582455

```
In []: df_filtered = df[df['A'] > 0]
        print(df_filtered)
Out[] :
```

	A	B	C	D
b	0.060295	-0.150527	0.149123	-0.701216
e	0.088279	0.418317	-1.102989	0.582455

We can also combine logical statements, we will filter all rows whose column 'A' and 'B' have their values greater than zero.

```
In []: df_filtered = df[(df['A'] > 0) & (df['B'] > 0)]
        print(df_filtered)
Out[] :
```

	A	B	C	D
e	0.088279	0.418317	-1.102989	0.582455

11.10 Iterating Pandas DataFrame

We can go through the DataFrame row by row to do operations in each iteration, let's see some examples.

```
In []: for item in df.iterrows():
         print(item)
Out[]:

('a', A      -0.633249
 B      -2.699088
 C      0.574052
 D      0.652742
shift      NaN
Name: a, dtype: float64)
('b', A      0.060295
 B      -0.150527
 C      0.149123
D      -0.701216
shift      0.652742
Name: b, dtype: float64)
('c', A      -0.052515
 B      0.469481
 C      0.899180
 D      -0.608409
shift      -0.701216
Name: c, dtype: float64)
('d', A      -1.352912
 B      0.103302
 C      0.457878
 D      -1.897170
shift      -0.608409
Name: d, dtype: float64)
('e', A      0.088279
 B      0.418317
 C      -1.102989
 D      0.582455
shift      -1.897170
Name: e, dtype: float64)
```

11.11 Merge, Append and Concat Pandas DataFrame

Another interesting feature of DataFrames is that we can merge, concatenate them and add new values, let's see how to do each of these operations.

- `merge` function allows us to merge two DataFrame by rows:

```
In []: import pandas as pd
        import numpy as np
        df=pd.DataFrame(np.random.randn(5, 4),
                        index=['a', 'b', 'c', 'd', 'e'],
                        columns=['A', 'B', 'C', 'D'])
        print(df)
Out[]:
```

	A	B	C	D
a	1.179924	-1.512124	0.767557	0.019265
b	0.019969	-1.351649	0.665298	-0.989025
c	0.351921	-0.792914	0.455174	0.170751
d	-0.150499	0.151942	-0.628074	-0.347300
e	-1.307590	0.185759	0.175967	-0.170334

```
In []: import pandas as pd
        import numpy as np
        df=pd.DataFrame(np.random.randn(5, 4),
                        index=['a', 'b', 'c', 'd', 'e'],
                        columns=['A', 'B', 'C', 'D'])
        print(df)
Out[]:
```

	A	B	C	D
a	2.030462	-0.337738	-0.894440	-0.757323
b	0.475807	1.350088	-0.514070	-0.843963
c	0.948164	-0.155052	-0.618893	1.319999
d	1.433736	-0.455008	1.445698	-1.051454
e	0.565345	1.802485	-0.167189	-0.227519

```
In []: df3 = pd.merge(df1, df2)
       print(df3)
```

```
Out[]: Empty DataFrame
        Columns: [A, B, C, D]
        Index: []
```

- append function allows us to append rows from one DataFrame to another DataFrame by rows:

```
In []: import pandas as pd
       import numpy as np
       df1=pd.DataFrame(np.random.randn(5, 4),
                        index=['a', 'b', 'c', 'd', 'e'],
                        columns=['A', 'B', 'C', 'D'])
```

```
In []: df2=pd.DataFrame(np.random.randn(5, 4),
                        index=['a', 'b', 'c', 'd', 'e'],
                        columns=['A', 'B', 'C', 'D'])
```

```
In []: df3 = df1.append(df2)
       print(df3)
```

```
Out[]:
```

	A	B	C	D
a	1.179924	-1.512124	0.767557	0.019265
b	0.019969	-1.351649	0.665298	-0.989025
c	0.351921	-0.792914	0.455174	0.170751
d	-0.150499	0.151942	-0.628074	-0.347300
e	-1.307590	0.185759	0.175967	-0.170334
a	2.030462	-0.337738	-0.894440	-0.757323
b	0.475807	1.350088	-0.514070	-0.843963
c	0.948164	-0.155052	-0.618893	1.319999
d	1.433736	-0.455008	1.445698	-1.051454
e	0.565345	1.802485	-0.167189	-0.227519

- concat function allows us to merge two DataFrame by rows or

columns:

```
In []: import pandas as pd
        import numpy as np
        df1=pd.DataFrame(np.random.randn(5, 4),
                         index=['a', 'b', 'c', 'd', 'e'],
                         columns=['A', 'B', 'C', 'D'])
```

```
In []: df2=pd.DataFrame(np.random.randn(5, 4),
                         index=['a', 'b', 'c', 'd', 'e'],
                         columns=['A', 'B', 'C', 'D'])
```

```
In []: df3 = pd.concat([df1, df2]) # Concat by row
       print(df3)
```

Out[] :

	A	B	C	D
a	1.179924	-1.512124	0.767557	0.019265
b	0.019969	-1.351649	0.665298	-0.989025
c	0.351921	-0.792914	0.455174	0.170751
d	-0.150499	0.151942	-0.628074	-0.347300
e	-1.307590	0.185759	0.175967	-0.170334
a	2.030462	-0.337738	-0.894440	-0.757323
b	0.475807	1.350088	-0.514070	-0.843963
c	0.948164	-0.155052	-0.618893	1.319999
d	1.433736	-0.455008	1.445698	-1.051454
e	0.565345	1.802485	-0.167189	-0.227519

```
In []: df3 = pd.concat([df1, df2], axis=0) # Concat by row
       print(df3)
```

Out[] :

	A	B	C	D
a	1.179924	-1.512124	0.767557	0.019265
b	0.019969	-1.351649	0.665298	-0.989025
c	0.351921	-0.792914	0.455174	0.170751

	A	B	C	D
d	-0.150499	0.151942	-0.628074	-0.347300
e	-1.307590	0.185759	0.175967	-0.170334
a	2.030462	-0.337738	-0.894440	-0.757323
b	0.475807	1.350088	-0.514070	-0.843963
c	0.948164	-0.155052	-0.618893	1.319999
d	1.433736	-0.455008	1.445698	-1.051454
e	0.565345	1.802485	-0.167189	-0.227519

```
# Concat by column
In []: df3 = pd.concat([df1, df2], axis=1)
        print(df3)

Out[]:
```

	A	B	...	D	A	...	D
a	1.179924	-1.512124	...	0.019265	2.030462	...	-0.757323
b	0.019969	-1.351649	...	-0.989025	0.475807	...	-0.843963
c	0.351921	-0.792914	...	0.170751	0.948164	...	1.319999
d	-0.150499	0.151942	...	-0.347300	1.433736	...	-1.051454
e	-1.307590	0.185759	...	-0.170334	0.565345	...	-0.227519

11.12 TimeSeries in Pandas

Pandas TimeSeries includes a set of tools to work with Series or DataFrames indexed in time. Usually, the series of financial data are of this type and therefore, knowing these tools will make our work much more comfortable. We are going to start creating time series from scratch and then we will see how to manipulate them and convert them to different frequencies.

11.12.1 Indexing Pandas TimeSeries

With `date_range` Panda's method, we can create a time range with a certain frequency. For example, create a range starting in December 1st, 2018, with 30 occurrences with an hourly frequency.

```
In []: rng = pd.date_range('12/1/2018', periods=30,
                           freq='H')
        print(rng)

Out[]: DatetimeIndex([
    '2018-12-01 00:00:00', '2018-12-01 01:00:00',
    '2018-12-01 02:00:00', '2018-12-01 03:00:00',
    '2018-12-01 04:00:00', '2018-12-01 05:00:00',
    '2018-12-01 06:00:00', '2018-12-01 07:00:00',
    '2018-12-01 08:00:00', '2018-12-01 09:00:00',
    '2018-12-01 10:00:00', '2018-12-01 11:00:00',
    '2018-12-01 12:00:00', '2018-12-01 13:00:00',
    '2018-12-01 14:00:00', '2018-12-01 15:00:00',
    '2018-12-01 16:00:00', '2018-12-01 17:00:00',
    '2018-12-01 18:00:00', '2018-12-01 19:00:00',
    '2018-12-01 20:00:00', '2018-12-01 21:00:00',
    '2018-12-01 22:00:00', '2018-12-01 23:00:00',
    '2018-12-02 00:00:00', '2018-12-02 01:00:00',
    '2018-12-02 02:00:00', '2018-12-02 03:00:00',
    '2018-12-02 04:00:00', '2018-12-02 05:00:00'],
   dtype='datetime64[ns]', freq='H')
```

We can do the same to get a daily frequency² (or any other, as per our requirement). We can use the `freq` parameter to adjust this.

```
In []: rng = pd.date_range('12/1/2018', periods=10,
                           freq='D')
        print(rng)

Out[]: DatetimeIndex(['2018-12-01', '2018-12-02',
    '2018-12-03', '2018-12-04', '2018-12-05',
    '2018-12-06', '2018-12-07', '2018-12-08',
    '2018-12-09', '2018-12-10'],
   dtype='datetime64[ns]', freq='D')
```

Now, we have a `DatetimeIndex` in the `rng` object and we can use it to create a Series or DataFrame:

²<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandasrange.html>

```
In []: ts = pd.DataFrame(np.random.randn(len(rng), 4),
                        index=rng, columns=['A', 'B', 'C', 'D'])
                    print(ts)
Out[]:
```

	A	B	C	D
2018-12-01	0.048603	0.968522	0.408213	0.921774
2018-12-02	-2.301373	-2.310408	-0.559381	-0.652291
2018-12-03	-2.337844	0.329954	0.289221	0.259132
2018-12-04	1.357521	0.969808	1.341875	0.767797
2018-12-05	-1.212355	-0.077457	-0.529564	0.375572
2018-12-06	-0.673065	0.527754	0.006344	-0.533316
2018-12-07	0.226145	0.235027	0.945678	-1.766167
2018-12-08	1.735185	-0.604229	0.274809	0.841128

```
In []: ts = pd.Series(np.random.randn(len(rng)), index=rng)
                    print(ts)
Out[]: 2018-12-01    0.349234
        2018-12-02   -1.807753
        2018-12-03    0.112777
        2018-12-04    0.421516
        2018-12-05   -0.992449
        2018-12-06    1.254999
        2018-12-07   -0.311152
        2018-12-08    0.331584
        2018-12-09    0.196904
        2018-12-10   -1.619186
        2018-12-11    0.478510
        2018-12-12   -1.036074
```

Sometimes, we read the data from internet sources or from csv files and we need to convert the date column into the index to work properly with the Series or DataFrame.

```
In []: import pandas as pd
        df=pd.read_csv('TSLA.csv')
```

```
df.tail()  
Out[]:
```

	Date	Open	High	Low	Close	...	Name
1078	2013-12-11	141.88	143.05	139.49	139.65	...	TSLA
1079	2013-12-10	140.05	145.87	139.86	142.19	...	TSLA
1080	2013-12-09	137.00	141.70	134.21	141.60	...	TSLA
1081	2013-12-06	141.51	142.49	136.30	137.36	...	TSLA
1082	2013-12-05	140.15	143.35	139.50	140.48	...	TSLA

Here, we can see the index as numeric and a Date column, let's convert this column into the index to indexing our DataFrame, read from a csv file, in time. For this, we are going to use the Pandas set_index method

```
In []: df = df.set_index('Date')  
       df.tail()  
Out[]:
```

Date	Open	High	Low	Close	...	Name
2013-12-11	141.88	143.05	139.49	139.65	...	TSLA
2013-12-10	140.05	145.87	139.86	142.19	...	TSLA
2013-12-09	137.00	141.70	134.21	141.60	...	TSLA
2013-12-06	141.51	142.49	136.30	137.36	...	TSLA
2013-12-05	140.15	143.35	139.50	140.48	...	TSLA

Now, we have Pandas TimeSeries ready to work.

11.12.2 Resampling Pandas TimeSeries

A very useful feature of Pandas TimeSeries is the resample capacity, this allows us to pass the current frequency to another higher frequency (we can't pass to lower frequencies, because we don't know the data).

As it can be supposed, when we pass from one frequency to another data could be lost, for this, we must use some function that treat the values of

each frequency interval, for example, if we pass from an hourly frequency to daily, we must specify what we want to do with the group of data that fall inside each frequency, we can do a mean, a sum, we can get the maximum or the minimum, etc.

```
In []: rng = pd.date_range('12/1/2018', periods=30,
                           freq='H')
        ts = pd.DataFrame(np.random.randn(len(rng), 4),
                           index=rng, columns=['A', 'B', 'C', 'D'])
        print(ts)
Out[]:
```

	A	B	C	D
2018-12-01 00:00:00	0.048603	0.968522	0.408213	0.921774
2018-12-01 01:00:00	-2.301373	-2.310408	-0.559381	-0.652291
2018-12-01 02:00:00	-2.337844	0.329954	0.289221	0.259132
2018-12-01 03:00:00	1.357521	0.969808	1.341875	0.767797
2018-12-01 04:00:00	-1.212355	-0.077457	-0.529564	0.375572
2018-12-01 05:00:00	-0.673065	0.527754	0.006344	-0.533316

```
In []: ts = ts.resample("1D").mean()
        print(ts)
Out[]:
```

	A	B	C	D
2018-12-01	0.449050	0.127412	-0.154179	-0.358324
2018-12-02	-0.539007	-0.855894	0.000010	0.454623

11.12.3 Manipulating TimeSeries

We can manipulate the Pandas TimeSeries in the same way that we have done until now, since they offer us the same capacity that the Pandas Series and the Pandas DataFrames. Additionally, we can work comfortably with all jobs related to handling dates. For example, to obtain all the data from a date, to obtain the data in a range of dates, etc.

```
In []: rng = pd.date_range('12/1/2018', periods=30,
                           freq='D')
        ts = pd.DataFrame(np.random.randn(len(rng), 4),
                           index=rng, columns=['A', 'B', 'C', 'D'])
        print(ts)
Out[]:
```

	A	B	C	D
2018-12-01	0.048603	0.968522	0.408213	0.921774
2018-12-02	-2.301373	-2.310408	-0.559381	-0.652291
2018-12-03	-2.337844	0.329954	0.289221	0.259132
2018-12-04	1.357521	0.969808	1.341875	0.767797
2018-12-05	-1.212355	-0.077457	-0.529564	0.375572
2018-12-06	-0.673065	0.527754	0.006344	-0.533316

Getting all values from a specific date:

```
In []: ts['2018-12-15']
Out[]:
```

	A	B	C	D
2018-12-02	0.324689	-0.413723	0.019163	0.385233
2018-12-03	-2.198937	0.536600	-0.540934	-0.603858
2018-12-04	-1.195148	2.191311	-0.981604	-0.942440
2018-12-05	0.621298	-1.435266	-0.761886	-1.787730
2018-12-06	0.635679	0.683265	0.351140	-1.451903

Getting all values inside a date range:

```
In []: ts['2018-12-15':'2018-12-20']
Out[]:
```

	A	B	C	D
2018-12-15	0.605576	0.584369	-1.520749	-0.242630

	A	B	C	D
2018-12-16	-0.105561	-0.092124	0.385085	0.918222
2018-12-17	0.337416	-1.367549	0.738320	2.413522
2018-12-18	-0.011610	-0.339228	-0.218382	-0.070349
2018-12-19	0.027808	-0.422975	-0.622777	0.730926
2018-12-20	0.188822	-1.016637	0.470874	0.674052

11.13 Key Takeaways

1. Pandas DataFrame and Pandas Series are some of the most important data structures. It is a must to acquire fluency in its handling because we will find them in practically all the problems that we handle.
2. A DataFrame is a data structure formed by rows and columns and has an index.
3. We must think of them as if they were data tables (for the Array with a single column) with which we can select, filter, sort, add and delete elements, either by rows or columns.
4. Help in ETL processes (Extraction, Transformation and Loading)
5. We can select, insert, delete and update elements with simple functions.
6. We can perform computations by rows or columns.
7. Has the ability to run vectorized computations.
8. We can work with several DataFrames at the same time.
9. Indexing and subsetting are the most important features from Pandas.
10. Facilitates the statistical exploration of the data.
11. It offers us a variety of options for handling NaN data.
12. Another additional advantage is the ability to read & write multiple data formats (CSV, Excel, HDF5, etc.).
13. Retrieve data from external sources (Yahoo, Google, Quandl, etc.)
14. Finally, it has the ability to work with date and time indexes and offers us a set of functions to work with dates.

Chapter 12

Data Visualization with Matplotlib

Matplotlib is a popular Python library that can be used to create data visualizations quite easily. It is probably the single most used Python package for 2D-graphics along with limited support for 3D-graphics. It provides both, a very quick way to visualize data from Python and publication-quality figures in many formats. Also, It was designed from the beginning to serve two purposes:

1. Allow for interactive, cross-platform control of figures and plots
2. Make it easy to produce static vector graphics files without the need for any GUIs.

Much like Python itself, Matplotlib gives the developer complete control over the appearance of their plots. It tries to make easy things easy and hard things possible. We can generate plots, histograms, power spectra, bar charts, error charts, scatter plots, etc. with just a few lines of code. For simple plotting, the `pyplot` module within `matplotlib` package provides a MATLAB-like interface to the underlying object-oriented plotting library. It implicitly and automatically creates figures and axes to achieve the desired plot.

To get started with Matplotlib, we first *import* the package. It is a common practice to import `matplotlib.pyplot` using the alias as `plt`. The `pyplot` being the sub-package within Matplotlib provides the common charting

functionality. Also, if we are working in a Jupyter Notebook, the line `%matplotlib inline` becomes important, as it makes sure that the plots are embedded inside the notebook. This is demonstrated in the example below:

```
import matplotlib.pyplot as plt

%matplotlib inline
```

NOTE: Matplotlib does not fall under the Python Standard Library and hence, like any other third party library, it needs to be installed before it can be used. It can be installed using the command `pip install matplotlib`.

12.1 Basic Concepts

Matplotlib allows creating a wide variety of plots and graphs. It is a humongous project and can seem daunting at first. However, we will break it down into bite-sized components and so learning it should be easier.

Different sources use 'plot' to mean different things. So let us begin by defining specific terminology used across the domain.

- **Figure** is the top-level container in the hierarchy. It is the overall window where everything is drawn. We can have multiple independent figures, and each figure can have multiple **Axes**. It can be created using the `figure` method of `pyplot` module.
- **Axes** is where the plotting occurs. The axes are effectively the area that we plot data on. Each **Axes** has an **X-Axis** and a **Y-Axis**.

The below mentioned example illustrates the use of the above-mentioned terms:

```
fig = plt.figure()
<Figure size 432x288 with 0 Axes>
```

Upon running the above example, nothing happens really. It only creates a figure of size 432 x 288 with 0 Axes. Also, Matplotlib will not show anything until told to do so. Python will wait for a call to `show` method to display the

plot. This is because we might want to add some extra features to the plot before displaying it, such as title and label customization. Hence, we need to call `plt.show()` method to show the figure as shown below:

```
plt.show()
```

As there is nothing to plot, there will be no output. While we are on the topic, we can control the size of the figure through the `figsize` argument, which expects a tuple of (width, height) in inches.

```
fig = plt.figure(figsize=(8, 4))
<Figure size 576x288 with 0 Axes>
plt.show()
```

12.1.1 Axes

All plotting happens with respect to an `Axes`. An `Axes` is made up of `Axis` objects and many other things. An `Axes` object must belong to a `Figure`. Most commands that we will ever issue will be with respect to this `Axes` object. Typically, we will set up a `Figure`, and then add `Axes` on to it. We can use `fig.add_axes` but in most cases, we find that adding a subplot fits our need perfectly. A subplot is an axes on a grid system.

- `add_subplot` method adds an `Axes` to the figure as part of a subplot arrangement.

```
# -Example 1-
# Creating figure
fig = plt.figure()

# Creating subplot
# Sub plot with 1 row and 1 column at the index 1
ax = fig.add_subplot(111)
plt.show()
```

The above code adds a single plot to the figure `fig` with the help of `add_subplot()` method. The output we get is a blank plot with axes ranging from 0 to 1 as shown in *figure 1*.

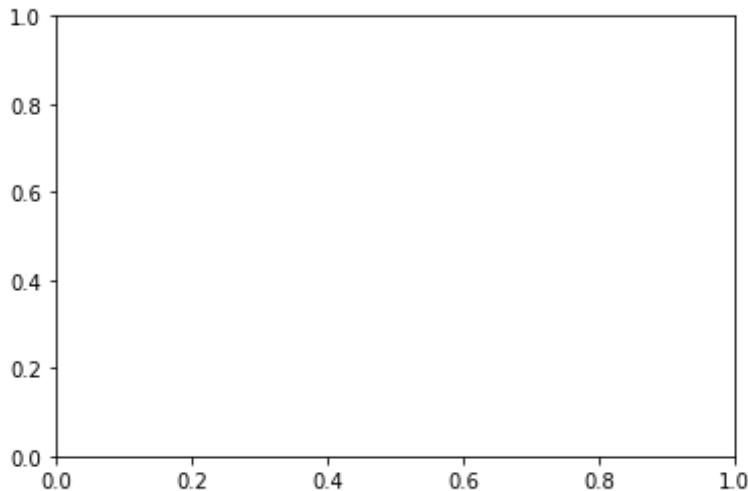


Figure 1: Empty plot added on axes

We can customize the plot using a few more built-in methods. Let us add the title, X-axis label, Y-axis label, and set limit range on both axes. This is illustrated in the below code snippet.

```
# -Example 2-
fig = plt.figure()

# Creating subplot/axes
ax = fig.add_subplot(111)

# Setting axes/plot title
ax.set_title('An Axes Title')

# Setting X-axis and Y-axis limits
ax.set_xlim([0.5, 4.5])
ax.set_ylim([-3, 7])

# Setting X-axis and Y-axis labels
ax.set_ylabel('Y-Axis Label')
ax.set_xlabel('X-Axis Label')
```

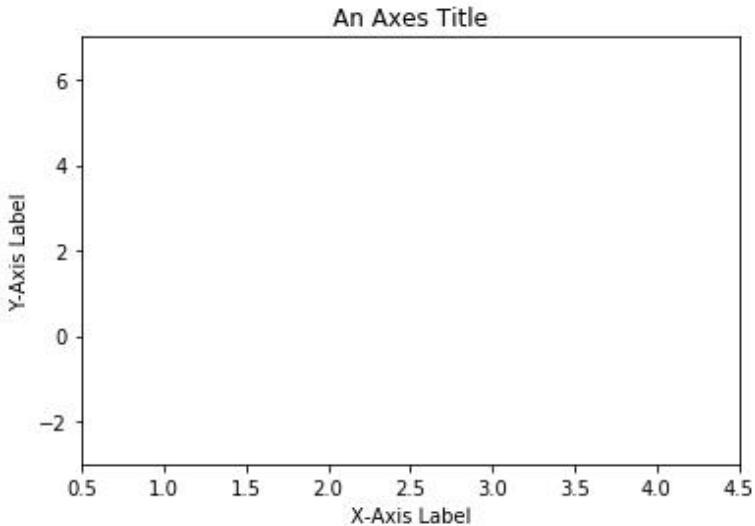


Figure 2: An empty plot with title, labels and custom axis limits

```
# Showing the plot
plt.show()
```

The output of the above code is shown in *figure 2*. Matplotlib's objects typically have lots of *explicit setters*, i.e. methods that start with `set_<something>` and control a particular option. Setting each option using explicit setters becomes repetitive, and hence we can set all required parameters directly on the axes using the `set` method as illustrated below:

```
# -Example 2 using the set method-
fig = plt.figure()

# Creating subplot/axes
ax = fig.add_subplot(111)

# Setting title and axes properties
ax.set(title='An Axes Title', xlim=[0.5, 4.5],
      ylim=[-3, 7], ylabel='Y-Axis Label',
      xlabel='X-Axis Label')

plt.show()
```

NOTE: The `set` method does not just apply to `Axes`; it applies to more-or-less all `matplotlib` objects.

The above code snippet gives the same output as *figure 2*. Using the `set` method when all required parameters are passed as arguments.

12.1.2 Axes method v/s pyplot

Interestingly, almost all methods of `axes` objects exist as a method in the `pyplot` module. For example, we can call `plt.xlabel('X-Axis Label')` to set label of X-axis (`plt` being an alias for `pyplot`), which in turn calls `ax.set_xlabel('X-Axis Label')` on whichever axes is *current*.

```
# -Example 3-
# Creating subplots, setting title and axes labels
# using `pyplot`
plt.subplots()
plt.title('Plot using pyplot')
plt.xlabel('X-Axis Label')
plt.ylabel('Y-Axis Label')
plt.show()
```

The code above is more intuitive and has fewer variables to construct a plot. The output for the same is shown in *figure 3*. It uses implicit calls to `axes` method for plotting. However, if we take a look at "*The Zen of Python*" (try `import this`), it says:

"Explicit is better than implicit."

While very simple plots, with short scripts, would benefit from the conciseness of the `pyplot` implicit approach, when doing more complicated plots, or working within larger scripts, we will want to explicitly pass around the `axes` and/or `figure` object to operate upon. We will be using both approaches here wherever it deems appropriate.

Anytime we see something like below:

```
fig = plt.figure()
ax = fig.add_subplot(111)
```

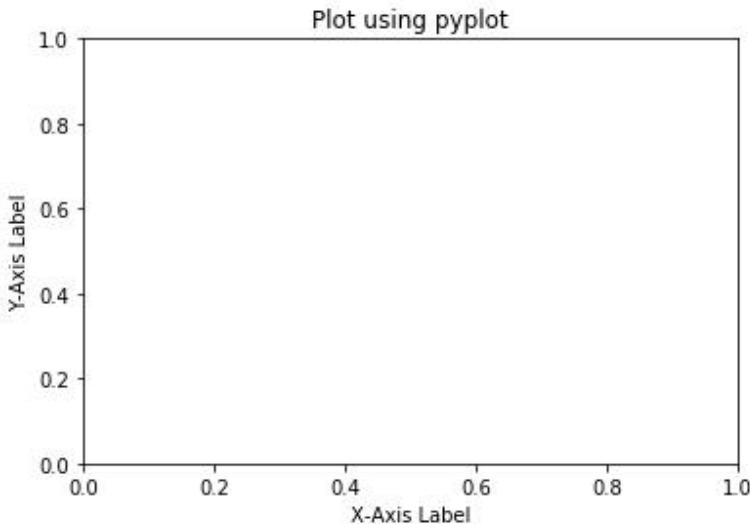


Figure 3: An empty plot using `pypplot`

can almost always be replaced with the following code:

```
fig, ax = plt.subplots()
```

Both versions of code produce the same output. However, the latter version is cleaner.

12.1.3 Multiple Axes

A figure can have more than one `Axes` on it. The easiest way is to use `plt.subplots()` call to create a figure and add the axes to it automatically. `Axes` will be on a regular grid system. For example,

```
# -Example 4-
# Creating subplots with 2 rows and 2 columns
fig, axes = plt.subplots(nrows=2, ncols=2)
plt.show()
```

Upon running the above code, Matplotlib would generate a figure with four subplots arranged with two rows and two columns as shown in *figure 4*.

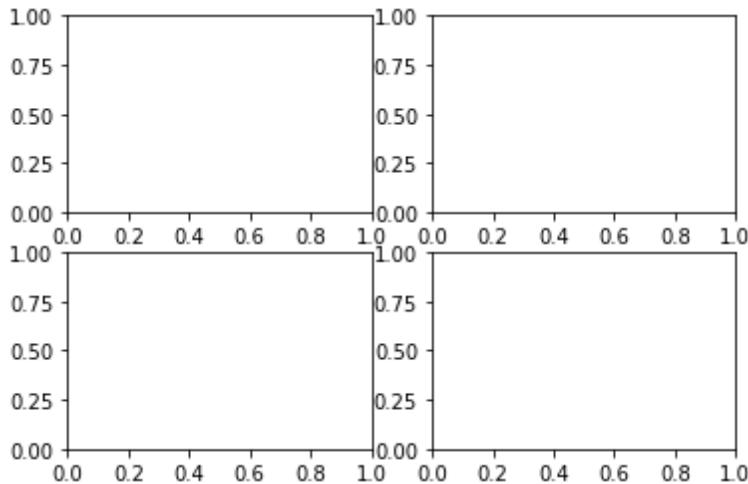


Figure 4: Figure with multiple axes

The `axes` object that was returned here would be a 2D-NumPy array, and each item in the array is one of the subplots. Therefore, when we want to work with one of these axes, we can index it and use that item's methods. Let us add the title to each subplot using the `axes` methods.

```
# -Example 5-
# Create a figure with four subplots and shared axes
fig, axes = plt.subplots(nrows=2, ncols=2, sharex=True,
sharey=True)
axes[0, 0].set(title='Upper Left')
axes[0, 1].set(title='Upper Right')
axes[1, 0].set(title='Lower Left')
axes[1, 1].set(title='Lower Right')
plt.show()
```

The above code generates a figure with four subplots and shared X and Y axes. Axes are shared among subplots in row wise and column-wise manner. We then set a title to each subplot using the `set` method for each subplot. Subplots are arranged in a clockwise fashion with each subplot having a unique index. The output is shown in *figure 5*.

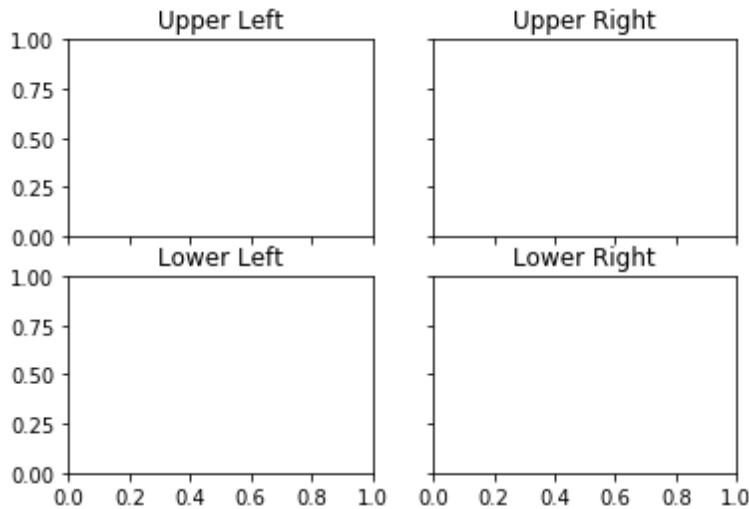


Figure 5: Subplots with the share axes

12.2 Plotting

We have discussed a lot about laying things out, but we haven't really discussed anything about plotting data yet. Matplotlib has various plotting functions. Many more than we will discuss and cover here. However, a full list or gallery¹ can be a bit overwhelming at first. Hence, we will condense it down and attempt to start with simpler plotting and then move towards more complex plotting. The `plot` method of `pyplot` is one of the most widely used methods in Matplotlib to plot the data. The syntax to call the `plot` method is shown below:

```
plot([x], y, [fmt], data=None, **kwargs)
```

The coordinates of the points or line nodes are given by x and y . The optional parameter fmt is a convenient way of defining basic formatting like color, market, and style. The `plot` method is used to plot almost any kind of data in Python. It tells Python what to plot and how to plot it, and also allows customization of the plot being generated such as color, type, etc.

¹<https://matplotlib.org/gallery/index.html>

12.2.1 Line Plot

A line plot can be plotted using the `plot` method. It plots Y versus X as lines and/or markers. Below we discuss a few scenarios for plotting line. To plot a line, we provide coordinates to be plotted along X and Y axes separately as shown in the below code snippet.

```
# -Example 6-
# Defining coordinates to be plotted on X and Y axes
# respectively
x = [1.3, 2.9, 3.1, 4.7, 5.6, 6.5, 7.4, 8.8, 9.2, 10]
y = [95, 42, 69, 11, 49, 32, 74, 62, 25, 32]

# Plot lists 'x' and 'y'
plt.plot(x, y)

# Plot axes labels and show the plot
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.show()
```

The above code plots values in the list `x` along the X-axis and values in the list `y` along the Y-axis as shown in *figure 6*.

The call to `plot` takes minimal arguments possible, i.e. values for Y-axis only. In such a case, Matplotlib will implicitly consider the index of elements in list `y` as the input to the X-axis as demonstrated in the below example:

```
# -Example 7-
# Defining 'y' coordinates
y = [95, 42, 69, 11, 49, 32, 74, 62, 25, 32]

# Plot list 'y'
plt.plot(y)

# Plot axes labels and show the plot
plt.xlabel('Index Values')
plt.ylabel('Elements in List Y')
plt.show()
```