VOL-1

PYTHON 101

FUNDAMENTALS

Sam

VOL-1

PYTHON 101

FUNDAMENTALS

Sam

INDEX

INTRODUCTION

This book is the first part of a series that is called the ***Python 101***. In this series, we are going to focus on learning the Python Programming Language in an effective and easy way. The goal is to learn it smartly, practically and fast without needing to read thousands of pages. We will keep it simple and precise. In this first book, we will introduce you to the language and learn the basics.

This book is for complete beginners and no *Programming, IT* or *Math* skills are required to understand it. At the end, you will be able to write various simple and interesting programs and you will have the necessary basis to continue with volume two.

What is Python?

Actually, there is a big misconception about Python. Most of the programmer says that Python is an Interpreted Language and other says it is a compiled language. But if you are finding an appropriate answer then it will be both. Python is an Interpreted Language as well as Compiled Language.

Because though we are not compiling our source code, but there is a background process that Compiled source the code into *Byte Code*, and then the Python Interpreter convert that byte code into Machine Language. And these all process done by Python Virtual Machine *(which we will download and install in the First Chapter of this book)*. That's why we can call Python a Compiled as well as Interpreted Language.

As we learned before Python is an interpreted and compiled Language. But Python is also Object-oriented, High-level programming language with dynamic semantics.

WHY PYTHON?

One question you might be asking yourself right now is- ‘Why learn Python? Why not Java, C, C++ or others? Why?’

First of all, a good Programmer is fluent in all programming language. So, learning Python doesn’t mean that you can’t learn others programming language. But secondly Python is probably the best language to start coding journey.

Python Syntax is so easy that even a child can easily grasp it. In other language there are a lot of things that you need to do manually. In this case Python makes our life easier. Because the hues library gives you ability to create something new with the help of things that already made. Meaning you do not need to make everything from Zero, you can use others program inside your program to create something new and obesely to save your valuable time & hard work.

Besides that, Python’s popularity is skyrocketing. According to TIOBE Index Python is the most popular language with upward trend *(August 2022).*

Source - www.tiobe.com/tiobe-index

CHAPTER – 1

Getting Start with Python

So now, before dive into the coding we need to install Python and development environment. As we learn before *(What is Python)* Python is a so-called Interpreted Language. It means that the source code needs the Interpreter or a certain Software to execute.

On the other hand, other languages like C, C++ or Java needs compiler to compile or convert source code into machine language *(Combination of 0 & 1)* instead of interpreter to execute the source code. But Python cannot do this. That why we need to install Python before get into coding.

1.1 - Python Installation

For downloading, visit the official website of Python & get the latest version from there. The link is given bellow.

Download Python: www.python.org

Download the installer and follow the instruction. Once you are completed installation process, you should have Python Interpreter as well as IDLE on your PC.

1.2 - Integrated Development Environment *(IDE)*

An IDE, or Integrated Development Environment, enables programmers to consolidate the different aspects of writing a computer program.

IDEs increase programmer productivity by combining common activities of writing software into a single application: editing source code, execute code and debugging.

You can use any IDE you want. But here we will discuss about 4 commonly used IDEs for Python Program.

1.2.1 - Editor & CLI

This is not mandatory to code Python in an IDE. You can write your code on any text editor like, Notepad, Vim, Atom, Sublime whatever you want. To do so at first you need to write and save code with right file name & extension (<filename>.py). Then you can run the code or script on CMD (Windows) or Terminal (Linux/Mac). In order to run the code, you need to use the following syntax:

python <filename>.py

This option is less convenient, but if you want you can go through it.

1.2.2 - Python IDLE

Python IDLE comes with Python installation. You've no need to install IDLE externally. One can call it as default IDE of Python. It is a great tool to write code and it has an Integrated Interpreter. Also, it's User Interface is very straight forward. For Beginner it is definitely more than enough. In this book will using the IDLE to run our Code.

1.2.3 - PyCharm

PyCharm is one of the most popular Python IDEs. There is a multitude of reasons for this. PyCharm is a cross-platform application, it is compatible with Linux, macOS, and Windows platforms. PyCharm comes with a plethora of modules, packages, and also professional syntax highlighting and a great user interface. I would definitely recommend it to every Python developer. If you are interested, you can get the community edition for free.

1.2.3 - Visual Studio Code

Last but not the least and my personal favourite very professional IDEs is Visual Studio Code. It is developed by Microsoft. You can also choose it as your Development Environment. There are a lot of reason to liking it. Mainly it's extensions, code snippet and advanced syntax highlighting, which helps a developer a lot. Also, it has ability to run any kind of code in one place and it is completely free. So now, Let's dive into code.

CHAPTER – 2

Our First Program

2 - Our First Program

Every Programming Language has some of its own syntax. Likes other Python also has its own syntax. But don’t worry, Python syntax is as simple as normal English Language. We are going to start our coding journey with a very simple program.

2.1 - Hello World!

It is a tradition in programming to start with a Hello World application, when you are learning a new language. So, we are going to do that in this chapter. A Hello World application is just a script that gives the text “Hello World!” as output onto the screen. In Python this is very easy task.

```
print("Hello World!")
```

As you can see, here we are using just a single line of code to print a text onto the screen. In other languages, we would have to define a basic structure with functions, classes and more, just to print one text.

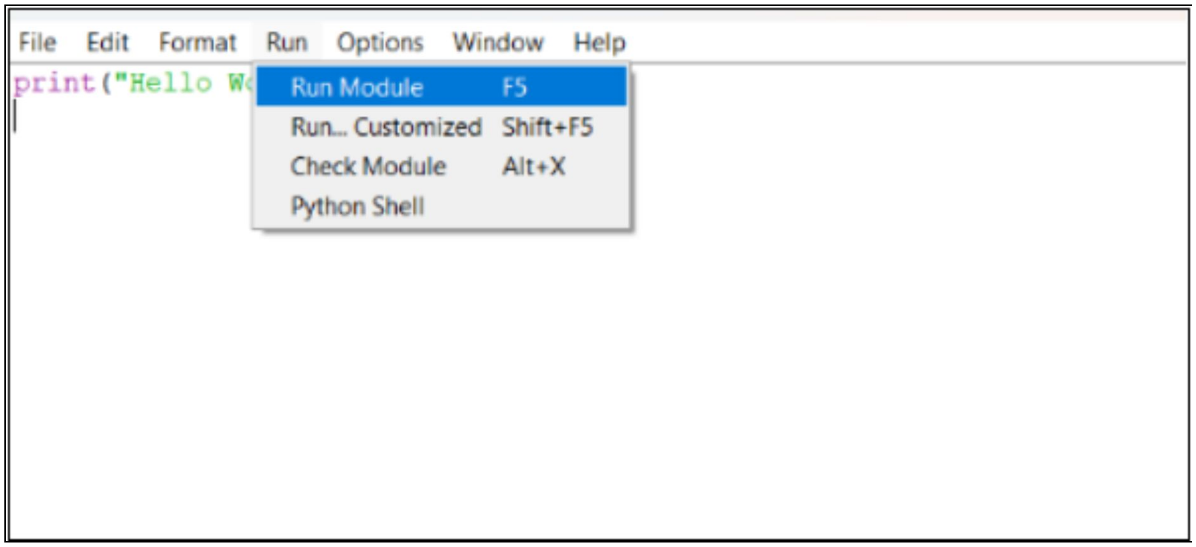
Let’s see what is happening here. The first thing we can notice is a so-called *function* with the name of *print*. The function *print* is used to print a certain text onto the screen. We should put the text between the parentheses which we want to print.

But there is a very important thing you have to keep in mind. The text we want to print that should be a string. String is a Datatype of Python. Don’t thing enough about it we’ll learn it briefly later in this book, just remember that whatever text we use in Python Programming that is a string. And we denote a string with the quotation mark. If we don’t use the quotation mark the Python Interpreter will think that *Hello World!* is a Variable name not a text and also give an *Error Message* as the Variable name is not defined. Again, don’t think more about Variable, because we’ll learn more about it in the next chapter.

2.1 – Running the Program

Now, time to run the script we just wrote. To do this you need to save the script into a Python file first.

- Go to **File** > **Save** or press **Ctrl** + **S** in order to save the script.
-

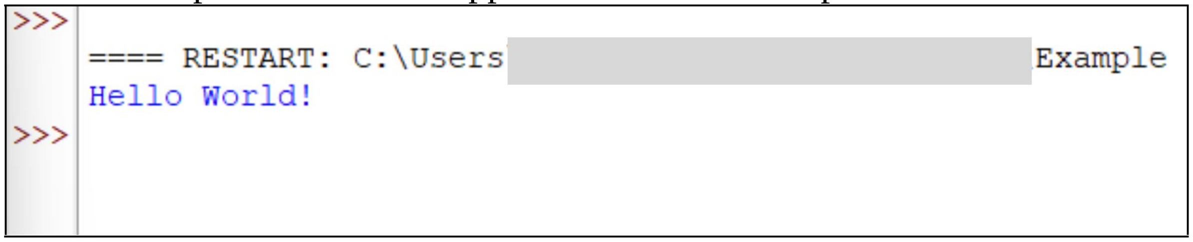


```
print("Hello Wo
```

Then click on **Run** > **Run Module** or press **F5** to run the script.

Running Python Program in IDLE

Then an Output window will appear and show the output.



```
>>>
    ==== RESTART: C:\Users                                    Example
    Hello World!
>>>
```

Output of Hello World! Program

And now, say congratulations to yourself, as you successfully run your *First Program.*

CHAPTER – 3

Data Types, Variables & Comments

3.1 – What is Data Types?

Data types are the classification or categorization of data. It represents the kind of value that tells what operations can be performed on a particular data. In Python Programming data types are actually classes and variables are instances or objects of that particular class.

3.2 – Types of Data Types

There are different types of data type in Python Programming. Here we'll learn all about them.

3.2.1 – Strings

A *String* is nothing but a sequence of character or a text. You should remember we print a string in our first program. In Python Programming whatever text or character, we write inside of quotation mark ("" or '') is called *String*. If a text isn't surrounded by quotation mark, then Python Interpreter will never realize the text as *String*. The keyword of String is ***str***.

3.2.2 – Integers

Integers are also called Numerical Data Types. An *Integer* is just a regular whole number, which we can use for basic calculations. The keyword of Integer is ***int***.

Example: 1, 54, 69, 23, 0, 32, 1947 etc.

3.2.3 – Floats

Floats are same as Integers. These are Numerical Data Types too. But in *Integer* we can't use decimal places, and the *Floats* are allowed us to use decimal places. Because it is a floating-point number. The keyword of *Float* is ***float***.

Example: 32.42, 5.01, 0.122 etc.

3.2.4 – Booleans

Boolean is basically a binary data type, which contains only two values, namely *True* or *False*. We will use it a lot when we get into the Conditions or Loops. The Keyword of Boolean data type is ***bool***.

3.2.5 – Sequences

We will learn about Sequences in a later chapter. But since *Sequences* are also data types, we should at least mention them here.

Data Types	Keyword	Description
List	list	Collection of values
Tuple	tuple	Immutable List
Dictionary	dict	List of key-value pairs

3.3 – What is Variable

A Variable is nothing but a container, which contain some values. Every variable has a specific data type. But the good thing of python programming is we don't need to declare variable or their data type before using. A variable is created at that moment whenever we assign a value to it. Because Python is not "Statically Typed".

3.4 – Creating Variable

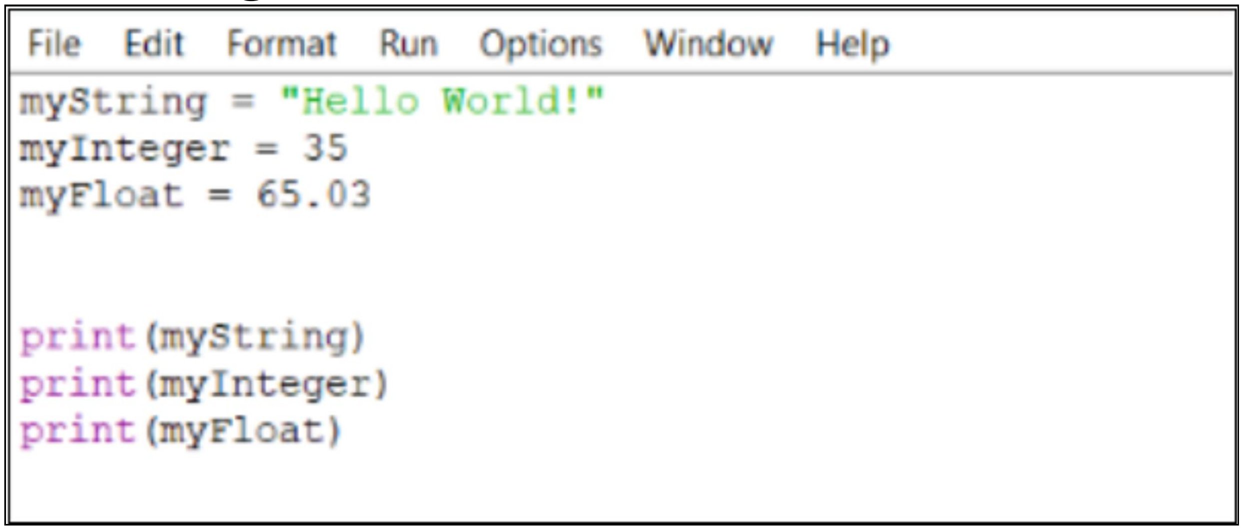
Creating variable in Python is very simple. You just choose a suitable name and assign a value to it with the help of *Assignment Operator* (=). For example

```
myString = "Hello World!"
myInteger = 35
```

```
myFloat = 65.03
```

Here we defined three variables. First one is a *String* type, second one is an *Integer* type & the last one is a *Float* type. Basically, you choose whatever name you want for variable, but there are some limitations. You are not allowed to use any reserved Keywords like, *str, int, dict, if, for, while etc.* Also, variable name can't be start with integers or special characters other than underscore (_).

3.5 – Using Variable

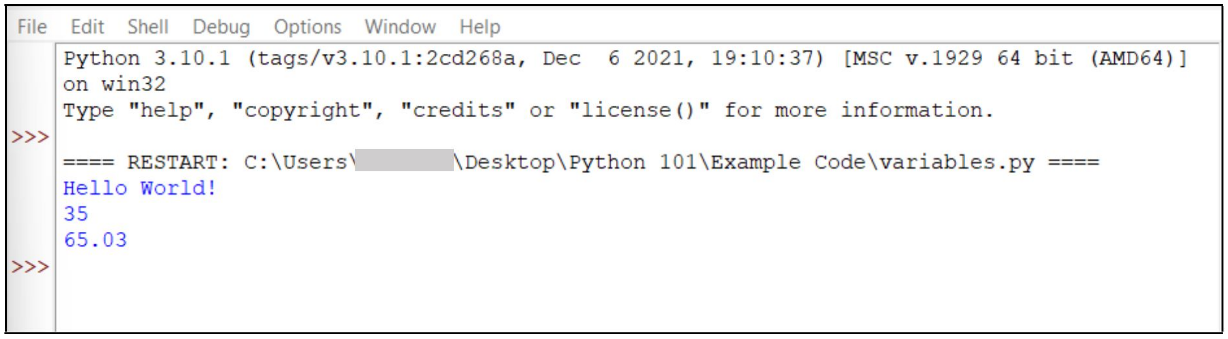


```
myString = "Hello World!"
myInteger = 35
myFloat = 65.03


print(myString)
print(myInteger)
print(myFloat)
```

Now we can start using our variables that we have defined before. For example, we are just printing our variables onto the screen. But you can use your variables anywhere & any purpose inside the script.

Here you can see we are not using quotation mark, for this the text inside parenthesis is treated as variable name. And Python Interpreter is printing the value of variables not the actual text written inside the parenthesis. The output of above code is given bellow.



```
Python 3.10.1 (tags/v3.10.1:2cd268a, Dec  6 2021, 19:10:37) [MSC v.1929 64 bit (AMD64)]
on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==== RESTART: C:\Users\[redacted]\Desktop\Python 101\Example Code\variables.py ====
Hello World!
35
65.03
>>>
```

3.6 – Typecasting

Sometime we will get values in such a tata type that we can't work with properly. For this we convert those values into our desired data type, and this process is called *Typecasting*.

For example, we might get a string as an input but the string number as its value. In this case "11" is not equal to 11. We can't do calculation with string, even though the string represents a number. For this reason, we need to typecast.

Syntax:

```
<required_datatype>(expression)
```

Example:

```
value = "11"
number = int(value)
```

Typecasting done by using specific data type function. In this case we are converting a *String* to an *Integer*. So, we need to use ***int()*** Function. You can also reverse this process using ***str()*** Function. In Python it is very important thing, we will use this quite often during coding.

3.6 – Comments

Comments in Python are nothing but one or more lines of text in the code, which are ignored by the interpreter during executing the script. It helps us to take short note about a code block and enhance readability of code. There are three types of comments in Python Programming.

Single Line Comments: If we put a Hash Symbol (#) at the beginning of any line in script the interpreter considers that line as a single line comment.

```
# This is a Single Line Comment
print("Python 101")
```

We can also add single line comment after any statement. Like,

```
# This is a Single Line Comment
print("Python 101")     # We can add comment here also
```

Multi-line Comment: Python doesn't offer a separate way to write multiline comments. However, there are other ways to get around this issue.

We can use Hash Symbol (#) at the beginning of each line of comment on multiple lines.

```
# This is a
# Multi-line
# Comment
print("Python 101")
```

Docstring Comment: Python docstring is the string literals with triple quotes that are appeared right after the function. It is used to associate documentation that has been written inside Python functions. It is added right below the functions to describe what they do.

```
def sum(a, b):
    """This Function takes two parameters
    and returns sum of them"""
    result = a + b
    return result
```

We will use Docstring Comment in later chapter *Function*. Also, no need to think more about above syntax of function, we will learn it there.

CHAPTER – 4

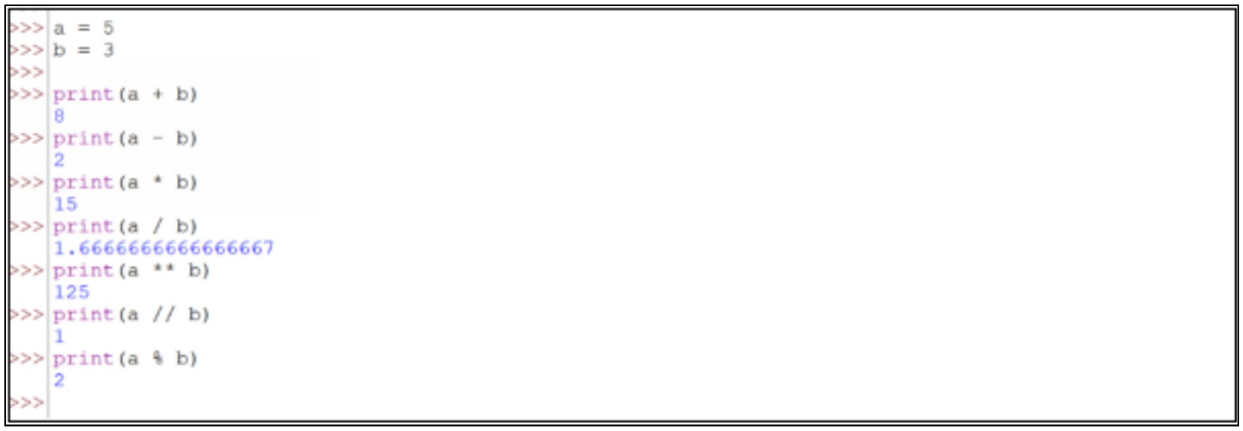
Operators

Python Operators are in general used to perform operations on variables and values. These are standard some symbols used for the purpose of logical, comparison & arithmetic operations. In this chapter, we will look into different types of Python operators.

4.1 – Arithmetic Operators

If I'm not wrong you already know about this type to operators. Because *Arithmetic Operators* are used to performing Arithmetic Operations like, *Addition, Subtraction, Multiplication & Division.*

Operator	Name	Description	Syntax
+	Addition	Adds two operands.	a + b
-	Subtraction	Subtract second operand from first operand.	a - b
*	Multiplication	Multiply two operands.	a * b
/	Division	Divide first operand with second one & returns ***Float*** Value.	a / b
**	Exponent	Second operand is the ***power*** of first one.	a ** b
//	Floor Division	Divide first operand with second one & returns ***Integer*** Value.	a // b
%	Modulus	Returns ***Reminder*** of a Division.	a % b



```
>>> a = 5
>>> b = 3
>>>
>>> print(a + b)
8
>>> print(a - b)
2
>>> print(a * b)
15
>>> print(a / b)
1.6666666666666667
>>> print(a ** b)
125
>>> print(a // b)
1
>>> print(a % b)
2
>>>
```

***Example of Arithmetic Operators**:*

4.2 – Assignment Operators

Basically, *Assignment Operators* are used for assigning value to a variable. In *Creating Variable* section, we used '=' to assign value to the variables. Here is the list of more *Assignment Operators*.

Operator	Name	Description	Syntax
=	Equal to	Assign Value of right-side expression to left side's variable.	x = a + b
+=	Addition AND	Add right-side operand with left-side operand and assign the value to left side operand.	a += b or a = a + b
-=	Subtraction AND	Subtract right-side operand from left-side operand and assign the value to left side operand.	a -= b or a = a - b
*=	Multiplication AND	Multiply right-side operand with left-side operand and assign the value to left side operand.	a *= b or a = a * b
/=	Division AND	Divide left-side operand with right -side operand and assign the value to left side operand.	a /= b or a = a / b
**=	Exponent	Calculate Exponent value &	a**=b or

	AND	assign to the left side operand.	a = a ** b
//=	Floor Division AND	Divide left-side operand with right -side operand and assign the value *(Floor)* to the left side operand.	A //= b or a = a // b
%=	Modulus AND	Takes modulus using left & right operands and assign value to the left-side operand.	a %= b or a = a % b

Basically, we use these operators to directly assign values to variables. The two-statement given bellow have the same effect. It's just simpler way to write it.

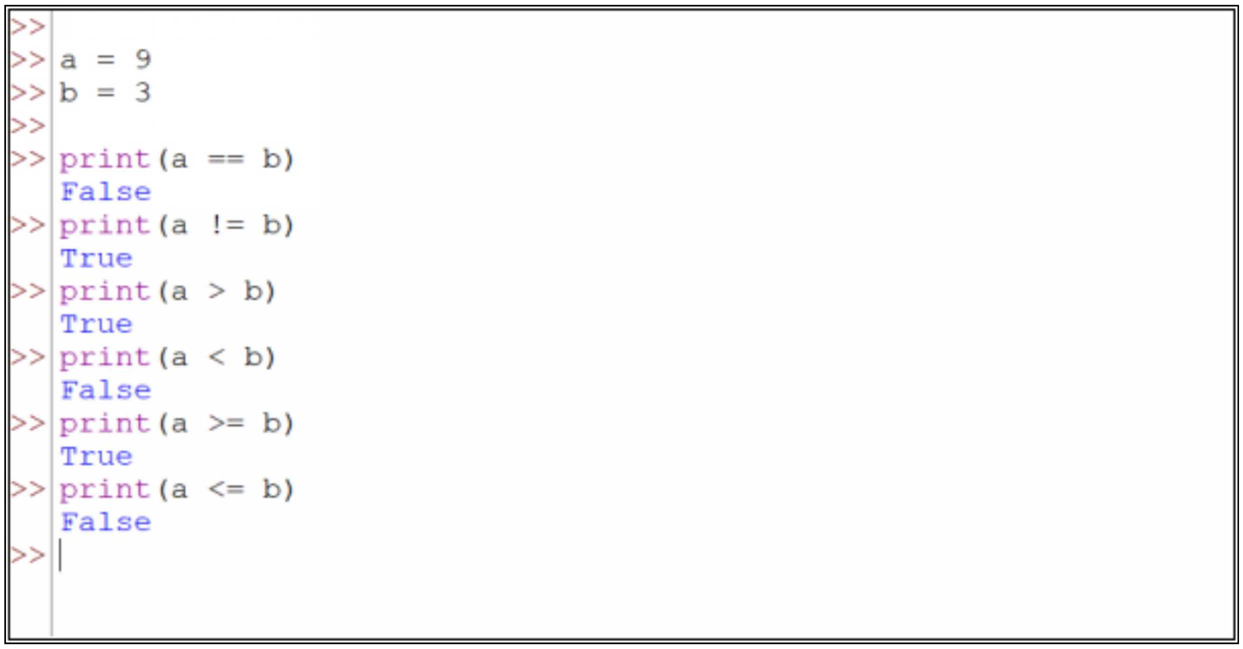
```
a = a + b
a += b
```

4.3 – Comparison Operators

This type of operators basically used to compare two or more values. When we use *Comparison Operators* in order to compare two or more values it returns a *Boolean*, means *True* or *False*.

Operator	Name	Description	Syntax
==	Equal	Values of both sides are ***same***	a == b
!=	Not Equal	Values of both sides are ***not same***	a != b
>	Greater Than	Value of left-side is ***greater*** than right-side	a > b
<	Less Than	Value of left-side is ***less*** than right-side	a < b
>=	Greater or Equal	Value of left-side is ***greater*** or ***equal*** to the right-side	a >= b
<=	Less or Equal	Value of left-side is ***less*** or ***equal*** to the right-side	a <= b

In Python Programming we use *Comparison Operators* to deal with *Conditions* & *Loops*. These two topics we will cover in later chapter.



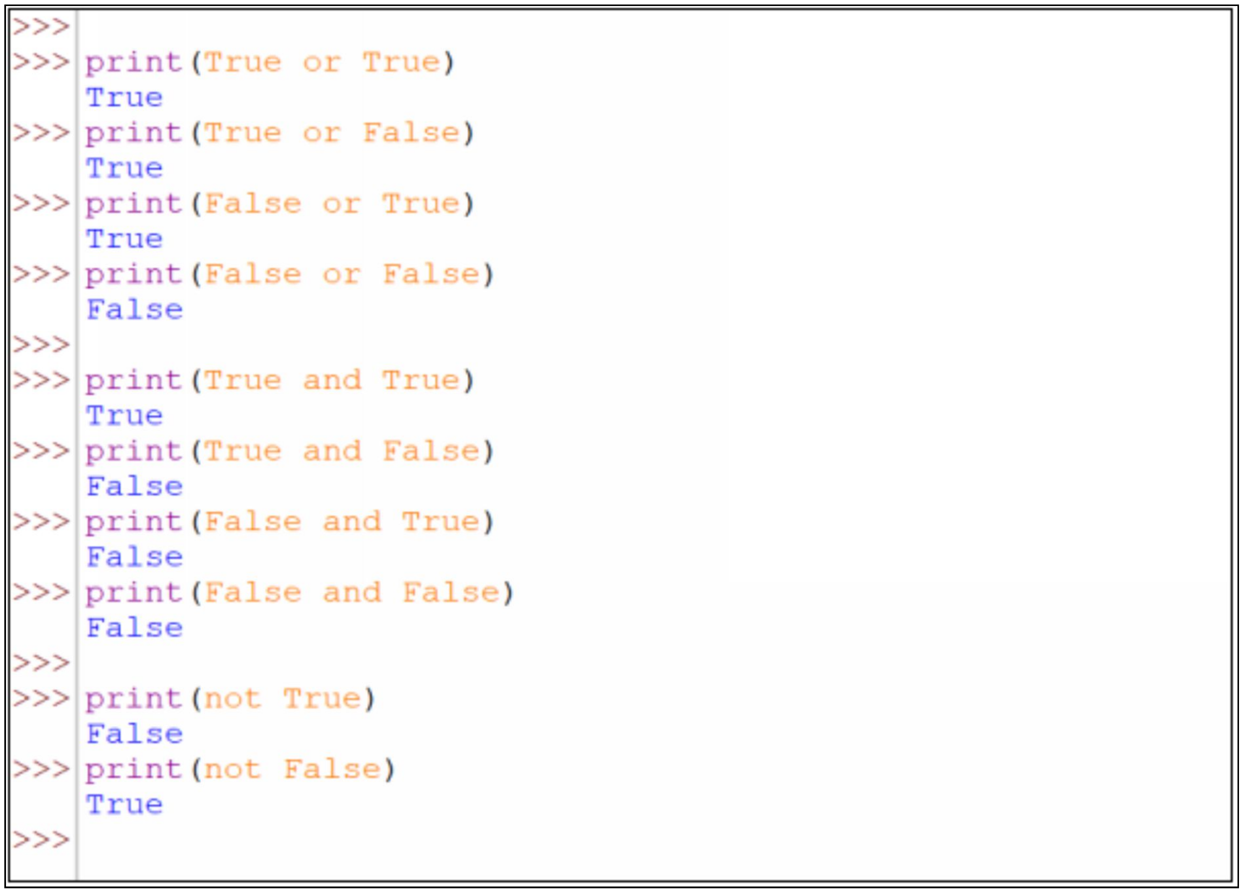
```
>>
>> a = 9
>> b = 3
>>
>> print(a == b)
   False
>> print(a != b)
   True
>> print(a > b)
   True
>> print(a < b)
   False
>> print(a >= b)
   True
>> print(a <= b)
   False
>> |
```

Example of Comparison Operators:

4.4 – Logical Operators

Logical Operators are also used in *Loops* or checking *Conditions*. In Python there are three Logical *Operators*, **or**, **and** & **not.**

Operator	Name	Description	Syntax
or	Logical Or	Condition will True if at least one operand is True.	a or b
and	Logical And	Condition will True if all operands are True.	a and b
not	Logical Not	Condition will True if operand is False.	not b



```
>>>
>>> print(True or True)
    True
>>> print(True or False)
    True
>>> print(False or True)
    True
>>> print(False or False)
    False
>>>
>>> print(True and True)
    True
>>> print(True and False)
    False
>>> print(False and True)
    False
>>> print(False and False)
    False
>>>
>>> print(not True)
    False
>>> print(not False)
    True
>>>
```

If you have basic knowledge of *Boolean Algebra*, it will help you to understand it. If you have no idea don't worry, the example given bellow will help you to understand.

4.5 – Other Operators

There are some more Operators in Python like *Membership operators* or *Bitwise Operators*. But some them we don't need or we need bit more programming knowledge to understand the. That's why we are not discussing about them in this chapter. We will learn it when we read *Sequences (Chapter- 8)*.

CHAPTER – 5

User Input

It is one of the most important things to get some data from users to give some sort of result in programming. Python has a inbuilt function called *input()*, which allows us to get data from users. In this chapter we are going to take a look at user input and how it works.

5.1 – Input Function

Input function is nothing but a inbuilt function of Python like print function, which gives us ability to get input from users.

```
# Taking input & storing into variable 'name'
name = input("Enter your name: ")

# Printing 'name' onto the screen
print(name)
```

When you execute the above script then you will able to enter your name on terminal and your input will be stored in variable *name*. Then the print function will print the value of *name* onto the terminal.

Output:

```
Enter your name: Sam
Sam
```

Let’s have a look at another example,

```
# Taking two numbers from user
number1 = input("Enter Number-1: ")
number2 = input("Enter Number-2: ")

# Adding number1 & number2 and storing into 'sum'
sum = number1 + number2

# Printing Result
print("Result is: ", sum)
```

Here we are taking two numbers from user then adding them & printing the result. But if you run this script, you will get a wired output. You will see the numbers you are entering are not adding, they just appending one to another. So why is this happening?

The problem is that the function *input()* returns only *string*. Means when you enter 15 & 43, it stores into variables as “15” & “43”, which are *string*.

So, what happen when we add two strings? We just append one to other. This means that the sum of “15” & “43” would be “1543” not 58.

To overcome this problem, means if you want to do any mathematical operation with input data, you need to typecast the variables into ***int*** or ***float*** first.

```
# Typecasting variables
number1 = int(number1)
number2 = int(number2)
```

You need to add these two lines of code before adding number1 & number2. Now if you enter 15 & 43 you will get 58 as output.

Output:

```
Enter Number-1: 15
Enter Number-2: 43
Result is: 58
```

Now our script is working well! Always remember that the input function returns a string and you need to typecast it, if you want to do calculations with it.

CHAPTER – 6

Conditional Statements

Decision-making is as important in Programming as it in our real life. Conditional Statements in Python perform different computations depending on whether a condition evaluates to *True* or *False*. These conditions are handled by if-else statement in Python.

6.1 – If-Elif-Else Conditions

Basically, a condition needs to be *True*, so that our script continues with the code written in its block. The three important keyword here are ***if***, ***elif*** and ***else***.

Syntax:

```
if Condition:
    <Statement>

elif Condition:
    <Statement>

else:
    <Statement >
```

Example:

```
number = input("Enter a Number: ")
number = int(number)

if number < 20:
    print("Number is less than 20.")
```

```
elif number > 30:
    print("Number is greater than 30.")

else:
    print("The Number is in 20 - 30")
```

Here you can see, we are taking ‘number’ input from user and typecasting the number into integer. Then our first *if-statement* check whether given number is less than 20 or not. Remember that the compression (number < 20) always returns True or False. So, if the condition returns True then only the ***if block*** of code will be executed. Interpreter will never check rest of the conditions.

```
Output:
Enter a Number: 1
Number is less than 20.
```

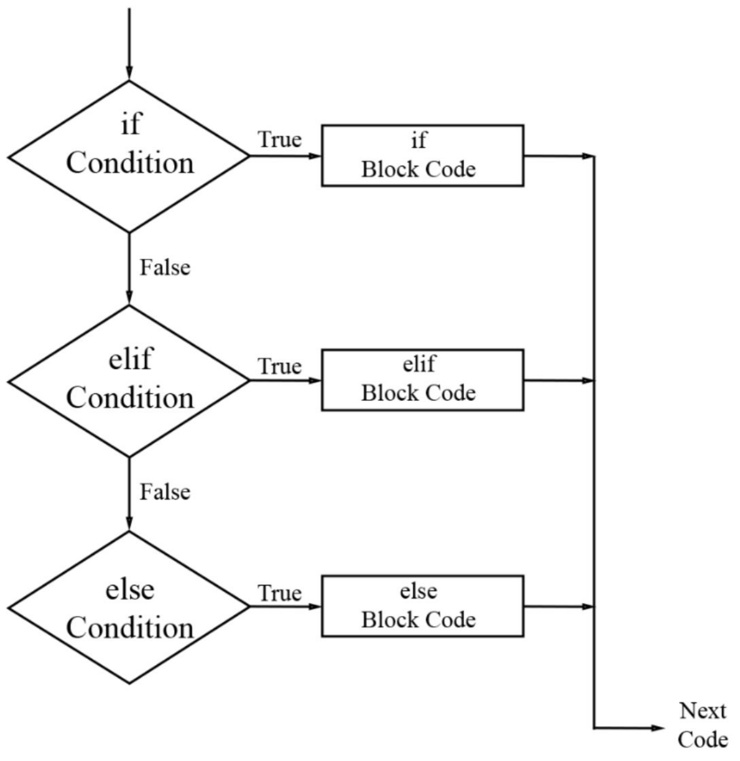
Now suppose, given number is 39. So, what will happen?

Our first condition will be False, because 39 is not less than 20. So, Interpreter will forget ***if block*** and will go forward to check next conditions. When a condition returns True, interpreter will execute that particular code block only. In this case next ***elif block*** will be executed. Because 39 if greater than 30. If no conditions return True the interpreter will execute ***else block*** only.

```
Output:
Enter a Number: 39
Number is greater than 30.
```

Remember, you can use unlimited ***elif*** conditions, but cannot use ***if*** or ***else*** more than one in an ***if-else*** ladder. The whole set of ***if-else*** condition is also called ***if-else*** ladder.

And of course, you don't need an ***elif*** or ***else*** block. You can just write an ***if*** statement and if the condition is not met, it skips the code and continues with the rest of the script.



6.2 – Flowchart

In this flowchart you can see how these basic if, elif and else trees work.

6.2 – Nested If-Else Statements

You can also put ***if-blocks*** into ***if-blocks***. These are called nested ***if-statements***.

```
number = input("Enter a Number: ")
number = int(number)

if number % 2 == 0:
    if number == 0:
        print("Your number is even but zero")
    else:
        print("Your number is even")
else:
    print ("Your number is odd")
```

So, here we have the first condition, which checks if the number is even. When it’s even it then checks if it’s a zero or not. That’s a trivial example but you get the concept.

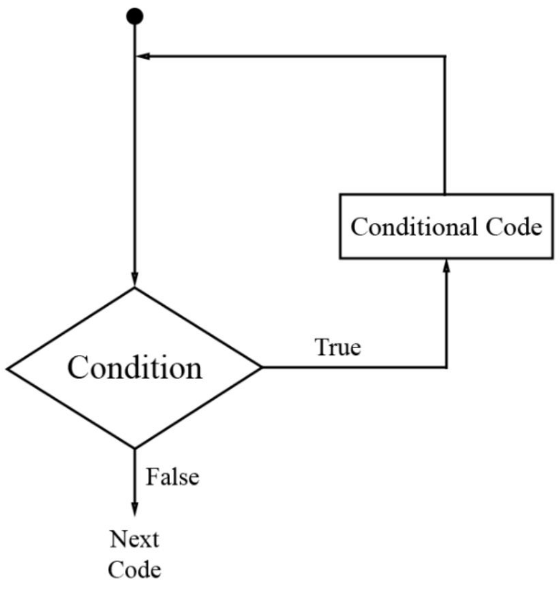
CHAPTER – 7

Loops

In general, statements are executed sequentially; The first statement is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

A loop statement allows us to execute a statement or group of statements multiple times.

7.1 – While Loop



There are two types of loops in Python Programming, While Loop & For Loop. A While Loop executes the code block only when *while* condition is met. The following diagram illustrates a *while loop* statement:

Here you can see, it goes in circles until the *while* condition returns *false*. Let's understand the concept with an example.

Suppose we need to print numbers from 1 to 10. So, you can easily use print() statement 10 times. But *while* loop allows as to do this task very easy way.

Let's have at the code,

```
number = 1

while number <= 10:
    print(number)
    number += 1
```

We define a while loop with the keyword *while*. Then we write our condition, like if-else condition and again code block is indented after colon ':'. As you know we are trying to print 1 to 10 in this example. So, first we initialized variable (number) with the value of 1. In every iteration, we are printing value of *number* and incrementing it with 1. This iteration will be done as long as the number is less than or equal to 10.

7.1.1 –Infinite Loop

In the section we'll learn about ***Infinite Loop*** or ***Endless Loop*** using while loop. This might seem less useful, but there are some applications of it.

```
while True:
    print("This will run forever")
```

We define an *Endless Loop* with the condition ***True*** since the condition is always ***True*** the loop will never break. The above loop will run & execute the print statement forever, unless you terminate the script.

Warning*: This might overload your computer, if you are using a low-end system.*

7.2 – For Loop

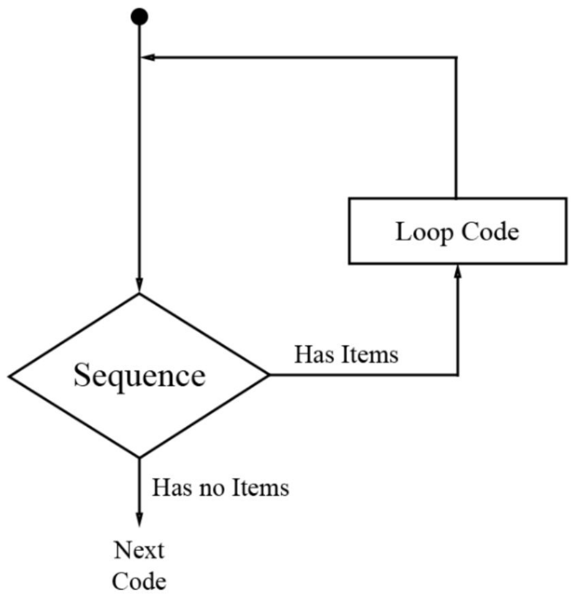
In python programming ***For Loop*** works on a different way. Usually, we use For Loop to iterate all or some of items of any ***Sequence*** which we'll learn in next chapter, we won't get deep into it here.

For now, we won't care about syntax of *Sequence*. Just notice we have a variable called ***numberList*** which is a list of 5 numbers.

```
numberList = [2, 4, 5, 9, 1]

for number in numberList:
    print(number)
```

Here we have stored a list of 5 numbers in variable ***numberList***. Then we use ***for*** keyword to define Loop and also define a control variable ***number***. For every iteration it will assign the value of next item of list. Here for every iteration the ***number*** will assign every item of ***numberList*** one by one. After that we can use the ***number*** for any purpose. For example, we are just printing all the numbers one by one.



The following flowchart will illustrate a ***For Loop*** statement:

7.2.1 – Range Function

Range is an inbuilt function of Python. Again, don't worry about what function is, because we'll learn it later chapter in this book. Just remember that when we pass a number to the ***Rang Function*** it returns a list of number from 0 to that number. And we can use that list in for loop. Like,

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

When we run this code, it will print 0 to 9. Because range function starts from 0 and ends on nth number what we'll pass to it. He the 10th number is 9. So, it will print 0 to 9.

But what if we want to start with a non-zero number? Then we have to pass two arguments to the range function.

Suppose, we need a list of numbers from 20 to 30. So, we have to pass these two arguments to the function. Like,

```
for i in range(20, 30):
    print (i)
```

Again, this loop will print numbers from 20 to 29. If you want to print 30 also, you need to pass 31. Do it yourself with different arguments, so that you can understand it better,

7.3 – Loop Control Statements

In order to manage loops, there are some so-called Control Statements in Python. They allow us to manipulate the flow of a loop at a specific point.

7.3.1 – Continue Statement

Continue Statement allow us to skip a specific iteration of a loop. Suppose we want to printing 1 to 10 except 7. So, we can do this,

```
for i in range(10):
    if i+1 == 7:
        continue
    print (i+1)
```

You can see we are printing (i+1) because as we learn earlier that range function returns index from 0. So, if we print (i+1) then we'll get numbers from 1 to 10. But you can also see that we are cheeking number if the number is equal to 7 or not with the *if statement*. And there is a ***Continue*** statement inside *if* statement. Means when the number will be 7 then *if* condition will be *True* & it will continue or skip that iteration.

Here we'll get an output like this,

```
Output:
```

```
1
2
3
4
5
6                        # The 7 is
8                        # Missing Here
9
10
```

7.3.2 – Break Statement

In *Continue Statement* we've seen that we can skip a specific iteration, but what if we want to stop any farther iteration when it met a specific condition.

We are using the same example to understand this, but we will use ***break statement*** instead of ***continue***.

```
for i in range(10):
    if i+1 == 7:
        break
    print (i+1)
```

And we will get an output like,

```
Output:
1
2
3
```

```
4
5
6                    # Loop Will Break Here
```

In this case when *if condition* will be *True* then it will break the loop. It will never iterate the loop any more.

7.3.3 – Pass Statement

The pass statement is a very special statement, since it does absolutely nothing. Actually, it is not really a loop control statement, but a placeholder for code.

```
if number == 5:
    pass
else:
    pass

while number < 5:
    pass
```

Sometimes you want to write your basic code structure, without implementing the logic yet. In this case, we can use the pass statement, in order to fill the code blocks. Otherwise, we can’t run the script, it will through an error like,

```
IndentationError: expected an indented block after 'if' statement on line 1
```

CHAPTER – 8

Sequences

Sequence is the most basic Data structure of Python. It contains multiple elements and they are indexed with a specific number. In this chapter, we are going to talk about the different types of sequences and their functions.

8.1 – List

We learn about *List* earlier a little bit. Here we'll learn in details. It is what the name says - just a list of items.

Numbers = [2, 5, 6, 7, 1, 6]

In Python, we define a ***List*** by using *Square Bracket*. Then we put all items in between of those and separate them by comma. And the most interesting thing is there is no obligation on Data Types in Python List. You can use any Data Type & also mix them. Like,

```
names = ["Sam", "Alex", "Virat", "jack"]
ages = [22, 19, 25, 24]
mixed = ["India", 9, True, 98.3]
```

8.1.1 – Accessing Values

In order to access the values of list, you need to know about indices first. Basically, the indices are the positions of every element of a list. In Python indexing starting with Zero. Means the index of first element is Zero, the second element has index One and so on. Now, let's try to access values by using their index.

```
print(names[0])
print(ages[1])
print(mixed[2])
```

Here we are accessing first item of names (Sam), second item of ages (19) and third item of mixed (True).

In order to access values, we need to put index in between square bracket just after the variable. This is the basic syntax of accessing values in Python.

Output:

```
Sam
19
True
```

In the above example we are accessing only one value of every list. But what if we want to access multiple values of a specific range or the whole list. Let’s have a look at the following examples.

Example - 1:

```
print(names[0:3])
print(names[:3])
print(names[1:])
```

Output:

```
['Sam', 'Alex', 'Virat']
['Sam', 'Alex', 'Virat']
['Alex', 'Virat', 'jack']
```

Here you can see we are passing a range of index to access values. In first print statement we are using *[0:3]* to access Zeroth to Third item. May a question can bother you that, we are passing end index 3, so why we are not getting the fourth item. Because the index 3 is for 4th element.

The reason is, what we are passing after Colene (here, 3) is not end index. It is nth element (here, 3rd element is “Virat”).

Syntax: ***Variable[starting index : nth element]***

There is one more thing to remember, if we don’t pass starting index, it will take 0 by default (second statement). And also, you can pass nothing as length to access rest of the list.

Hope you understand the concept of index. Let’s have a look at some more example to better understanding.

```
# Returns whole list
print(ages)
>>> [22, 19, 25, 24]

# Returns 3rd item to rest of the list
print(mixed[2:])
>>> [True, 98.3]

# Returns 2nd to 3rd item
print(names[1:3])
>>> ['Alex', 'Virat']
```

8.1.2 – Modifying Values

In Python list we can not only access but also modify the values by using indices. The process of modifying values is very simple.

```
names = ["Sam", "Alex", "Virat", "jack"]
ages = [22, 19, 25, 24]

names[3] = "Rahul"
ages[3] = 20

print(names)
print(ages)

Output:
['Sam', 'Alex', 'Virat', 'Rahul']
[22, 19, 25, 20]
```

Here we modified 4th element of both lists. Jack replaced by Rahul of names & 24 replaced by 20 of ages list.

8.1.3 – List Operations

In chapter Operators we saw different types of operators. We can apply those operators to our lists.

```

Adding 2 Lists:

list_1 = [22, 19, 25, 24]
list_2 = [54, 9, 11, 94]

list_3 = list_1 + list_2

print(list_3)


Output:
>>> [22, 19, 25, 24, 54, 9, 11, 94]
```

We can also multiple a list multiple time.

```

Multiplying a List 3 times:

list_1 = ["Sam", 22]
list_2 = list_1 * 3
```

```
print(list_2)


Output:
>>> ['Sam', 22, 'Sam', 22, 'Sam', 22]
```

8.1.4 – List Functions

In Python there are a lot of inbuilt function and methods of list to make our life easier. But here we are not going through them all. We will look at the most important once only.

List Functions:

Function	Description
len(list)	Returns the length of List
max(list)	Returns the item with Max value
min(list)	Returns the item with Min value
list(element)	Typecast any element into list

List Methods:

Method	Description
list.append(x)	Appends element (*x*) to the list
list.count(x)	Count how many elements (*x*) present in the list
list.index(x)	Returns the index of element (*x*)
list.pop(x)	Remove the element of list
list.reverse(x)	Reverse the order of elements in list
list.sort(x)	Sort the elements of list

8.2 – Tuple

The next sequence type we are going to look at is very similar to the list. It's the ***Tuple***. The only difference between a list and a tuple is that a tuple is *Immutable*. We can't manipulate it.

Numbers = (93, 43, 87, 54, 35)

Notice that a tuple is defined by using parentheses rather than square brackets.

Basically, all the reading and accessing functions like *len*, min and max stay the same and can be used with tuples. But of course, it is not possible to use any modifying or appending functions as tuple is immutable.

8.3 – Dictionary

Python Dictionary is a collection of *keys* & *values*. Every entry in this sequence has a key and a respective value. In other programming languages this structure is called *hash map*.

```
dict = {
    "name" : "Sam",
    "age" : 22,
    "dept" : "Programmer"
}
```

We define dictionaries by using *curly brackets* and the key-value pairs are separated by *commas*. The key and the value themselves are separated by *colons*. On the left side there is the *key* and on the right side the according *value*.

Since the keys are like index of list or tuple, it should be unique. But it is not mandatory to be unique for values. We can have many keys with the same value but when we address a certain key, it has to be the only one with that particular name. Also, keys can’t be changed.

And an interesting thing is we can store a list or tuple as value.

```
dict = {
    "name" : ["Sam", "Alex", "Virat", "jack"],
    "age" : [22, 19, 25, 24],
    "dept" : ("Programmer", "Writer", "Photographer", "Editor")
}
```

8.3.1 – Accessing Values

As we learn earlier that in dictionary keys are as unique as index of list, so we can access values by using keys.

```
print(dict["name"])
print(dict["age"])
```

Notice that if there were multiple keys with the same name, we couldn’t get a result because it wouldn’t know which value we are talking about.

8.3.2 – Dictionary Functions

Like lists & tuple Python Dictionary also has some *Functions* & *Methods*. But they are not same, because dictionary do not have indices.

Dictionary Functions:

Function	Description
len(dict)	Returns the length of Dictionary
str(dict)	Typecast Dictionary into a String

Dictionary Methods:

Method	Description
dict.clear()	Removes all elements of Dictionary
dict.fromkeys(dict)	Creates a new Dictionary with same key, but empty value
dict.copy()	Makes a copy of original Dictionary
dict.get(key)	Returns the value of given key
dict.items()	Returns all the items in a list of tuples
dict.keys()	Returns a list of all the keys
dict.values()	Returns a list of all the values
dict.update(dict2)	Add the content of a Dictionary (dict2) to another Dictionary (dict)

8.4 – Membership Operators

If you remember we skipped the *Membership Operators* in Operators Chapter *(Chapter- 4)*. Now time to learn about them. Usually, we use *Membership Operators* to check an item is a member of a sequence or not, and also iterate over them.

```
list = [23, "Sam", 32.4, True, "India"]

print("Sam" in list)            >>> True

print(32 in list)         >>> False

print("London" not in list)               >>> True
```

The two *membership operators* are ***in*** & ***not in***. Using them we can check a sequence contains the certain element or not. If contain it returns ***True***, otherwise returns ***False***.

Also, we can iterate all elements of a sequence by using membership operators.

```
list = [23, "Sam", 32.4, True, "India"]

for i in list:
    print(i)
```

For every iteration *i* becomes the value of next element and gets printed. We already talked about this in *Chapter-7 (Loop)*.

CHAPTER – 9

Functions

Function is nothing but a block of statements. Sometime we need to use some common code in different places for different values. And it reduces the readability & beauty of code. That’s why we use *Function*.

A *Function* allows us to write that code once and use multiple time. Functions can be seen as blocks of organized code that we reuse at different places in our scripts. They make our code more modular and increase the reusability.

9.1 – Defining Function

We use the keyword ***def***, followed by ***Function Name*** & the ***Parentheses*** in order to define a *Function*. The code needs to be indented after the colon.

```
# Defining Function
def greet():
    print("Welcome to Python 101")
```

This is a very basic example of a function. Here we define a function called ***greet***, then inside the function we are just printing a text. But if you run this code now nothing will happen. Because we have to call the function in order to execute the code block inside function.

```
# Calling Function
greet()
```

We can call the function anywhere in the script. And also, we can call the function multiple time in the same script. For this function it will print *Welcome to Python 101* every time we call the function.

9.1.1 – Parameters

In above example we saw that the function *greet* was only printing the same value. But we can make it dynamic by using parameters. Means we can get different values by passing different parameters.

```
# Defining Function with Parameters
def sum(num1, num2):
    print(num1 + num2)

# Calling Function
sum(33, 16)

Output:
>>> 49
```

As you can see, we have two parameters in between the parentheses ***num1*** & ***num2***. And the function ***sum()*** prints the summation of those two numbers.

When we call the function with two arguments 33 & 16, it gives us 49 as output. Now you can get different output by using different arguments.

But remember one thing, the number of arguments must be same of the number of parameters. Otherwise, it will through an error.

9.1.2 – Return Values

Till now we saw that the function is just executing the statements. But it can also return a certain value. In case of *sum()* function, it can also return the summation value instead of printing itself. This value can be saved or in can be processed. In order to return values, we use the keyword ***return***.

```
# Defining Function with Return Values
def sum(num1, num2):
    return(num1 + num2)
```

Here we are returning the summation of *num1* & *num2* instead of printing it.

```
# Calling Function - 1
result = sum(33, 16)
print(result)

# Calling Function - 2
print(sum(33, 16))

Output:
>>> 49
>>> 49
```

As the function returns the summation value, we can either store it into a variable or use directly. In every case the output will be same.

9.2 – Default Parameters

Some time we need to pass some default values to the parameters. In order to do this, we need to assign the value to the parameter in the function definition.

```
def hello(text="Python 101"):
    print(text)
```

As you can see, we have assigned a value to the parameter *text*. Then printing the value of *text*. When we call the function, it will print that text assigned to the parameter.

```
hello()

Output:
>>> Python 101
```

The interesting thing is we can change the default value during calling the function if it needed.

```
hello(text="This is changed value")

Output:
>>> This is changed value
```

9.3 – Args & KWArgs

Till now we saw that the number of *Parameters* & *Argument* should be the same. But what it we don't know the actual number of arguments the user will pass. In this case *Args* & *KWArgs* allows us to pass multiple arguments by defining only one *Parameters*.

Let's understand with a real-life example.

Suppose we are taking employee's info. So obesely name is mandatory, but one can skip some of info. So, we can do this,

```
# Defining Function with Args
def employee(name, *info):
   print(name, info)

# Calling Function with Multiple Arguments
employee("Sam", 22, "Kolkata", "Programmer")
employee("Rio", 20, "Designer")
```

Output:

```
Sam (22, 'Kolkata', 'Programmer')
Rio (20, 'Designer')
```

Here you can see, we are defining function with two parameters *name* & *info*. But We are using a asters (*) sign before *info*. Because info is a variable parameter. It can store multiple values in form of tuple.

And the interesting thing is there is no limit bound of number of elements. Actually, if you look at the output carefully you will see that our first argument *name* is a normal string, but rest of all arguments are showing as a tuple.

If we want to access values of *Args* we need to use asters (*) sign also while printing. Like,

```python
def employee(name, *info):
    print(name, *info)

employee("Sam", 22, "Kolkata", "Programmer")
```

Output:

```
Sam 22 Kolkata Programmer
```

But here all the information is not clear. Because when we call function for different employee it returns different result. So, it become very hard to decide which one is age or address or designation. In this case *KWArgs (Keyword Variable Parameter or Argument)* helps us to overcome this problem.

```
def employee(name, **info):
   print(name, info)


employee("Sam", age = 22, address = "Kolkata", desig = "Programmer")
```

Output:

```
Sam {'age': 22, 'address': 'Kolkata', 'desig': 'Programmer'}
```

We define *kwargs* with double asters (*) sign. And if you notice the above example, it returns a dictionary of key-value pairs. So, now we can apply dictionary functions or process the data.

9.4 – Scope

The last topic of this chapter is *Scope*. Scope is not only important for Functions, but also Loops, Conditions & other structure like these. Basically, scope means the concept of deference between *Global* & *Local* variables.

```
name = "Sam"

def function():
    name = "Jhon"
    print(name)


function()
print(name)
```

Here you can see there are two variables called *name*. One is out side the function & another one is inside the function.

Now, when we run this script first it will print ***Jhon*** then ***Sam***. Because first *print* statement is inside the function & second one is outside of the function.

Here the thing you should remember that, whatever variable we define inside of any function, loop, condition or other structure like these is called *Local Variable* of that particular statement and rest of all variables are called *Global Variable*.

We can't directly access any local variable outside of the statement. Also, can't modify global variable inside any statement. If we want to modify global variable, we have to use the keyword ***global***. But we can access global variable directly if there is no local variable of same name of global variable.

```
name = "Sam"

def function():
    global name
    name = "Jhon"
    print(name)

function()
print(name)
```

Output:

```
Jhon
Jhon
```

Here you can see, we can get full access of a global variable by using keyword *global*. Even we can manipulate the variable by using it.

CHAPTER – 10

Exception Handling

A program is full of errors & exceptions. I'm sure you have come across some, if not you haven't been trying hard enough! Errors or exceptions are an everyday part of programming. Take a look at this piece of code here.

```
print(10/0)
```

This piece of code will through a ***ZeroDivisionError,*** if we want to run it. That's because a division by zero is not defined and our script doesn't know how to handle it. So, it crashes.

Let's look at another example.

```
name = "Sam"
name = int(name)
```

Output:

```
Traceback (most recent call last):
  File "c:\...\test.py", line 2, in <module>
    name = int(name)
ValueError: invalid literal for int() with base 10: 'Sam'
```

Alternatively, when we try to typecast a *string* into *integer* it will through a ***ValueError*** and script will crush again.

10.1 – Try-Except

By using try & except block we can easily handle these types of errors. The structure of try-except is as same as if-else condition. But there is no condition. We write those code which have some confusions or network issue or something like this inside the *try* block. And in except block we write exceptions.

```
try:
    name = "Sam"
    name = int(name)
except ValueError:
    print("Code block for Value-Error")
```

The statement must have at least one *except* block. It may have more than one *except* block though.

10.1.1 – Else Statement

We can also use *else* statement. It will be executed when our code has no error.

```
try:
    name = "Sam"
    # name = int(name)
except ValueError:
    print("Code block for Value-Error")
else:
    print("All ok")
```

At first it will execute try block's code. Since there is no error, it will execute else block.

10.1.1 – Finally Statement

```
try:
    name = "Sam"
    # name = int(name)
except ValueError:
    print("Code block for Value-Error")
finally:
    print("It will execute always")
```

Finally statement is same as *else* statement. The basic & only difference between them is *else* executes only if the code has no error, but *finally* executes every time, no matter if code has error or not.

CHAPTER – 11

String Manipulation

Even though *string* is just a simple text or sequence of characters, we can apply a lot of functions and operations on them. Since this book is for beginner, we will not dive into advanced level, but it's important to you to know how to deal with a string properly.

11.1 – String Slicing

As we already learned, strings are sequence of characters and they can also treat like that. Means we can slice them by using indices like list or tuple.

```
text = "My name is Sam"

print(text[:5])
>>> My na

print(text[5:10])
>>> me is

print(text[10:])
>>>  Sam
```

We can also access characters of a string by using loop. It will print every individual character one after another.

```
text = "My name is Sam"

for x in text:
    print(x)
```

11.2 – String Concatenation

String Concatenation is nothing but a technique to join two or more strings into one string. There several ways to do this. Suppose we have two strings & we want to join them.

```
str_1 = "Hey guys"
str_2 = "Welcome to Python 101"
```

Comma: We can simply put a comma between two string to join them.

```
print(str_1, str_2)

>>> Hey guys Welcome to Python 101
```

+ Operator: + *Operator* is same as *comma*, but there is a little problem.

```
print(str_1 + str_2)

>>> Hey guysWelcome to Python 101
```

When we join two strings with + *Operator*, it will remove space between them. To avoid this problem, we need to put an extra space at the end of first string or at the beginning of second string.

f-string: *f-string* is mostly used string concatenation technique in python. Also, it's my personal favourite. By using *f-string* code become readable & easy to understand.

```
print(f"{str_1} {str_2}")

>>> Hey guys Welcome to Python 101
```

We define *f-string* with keyword ***f*** out-side of quotation mark (f""). Then in-side the quotation mark we put a string variable between *curly brackets*, where we want to join the string. Also, we can use normal text in-side of the quotation mark.

```
print(f"Hey guys {str_2}. Let's start Coding.")

>>> Hey guys Welcome to Python 101. Let's start coding.
```

11.3 – String Formatting

When we have a text & we want to insert value of variables in between the text then string formatting helps us to achieve that.

```
name, age = "Sam", 22

print("My name is %s and I'm %d years old." % (name, age))

>>> My name is Sam and I'm 22 years old.
```

If you noticed, here we used **%s** and **%d** where we want to insert our variables. It's called placeholder. There are different types of placeholders for different data types. The following table will teach you better.

Placeholder	Data Type
%c	Character
%s	String

%d or %i	Integer
%f	Float
%e	Exponential Notation

If you are lazy like me and don’t want to specify data types then you can use *format()* function. Simply use a curly bracket where you want to put the variable and pass all variables as arguments of format() function.

```
name, age = "Sam", 22

print("My name is {} and I'm {} years old.".format(name, age))

>>> My name is Sam and I'm 22 years old.
```

11.4 – Escape Characters

In string we use a lot of escape characters in order to manage string properly. Actually, they are non-printable characters. Like tab, new line, backspace etc. They all are initiated by backslash (\), that’s why we use double backslash for file path (Chapter-12).

The following table will summarize the most important escape characters. If want to know more escape characters search on google. But you won’t need them for now.

Escape Characters	**Description**
\b	Backspace
\n	New Line
\t	Tab

```
print("My name is\b Sam")
>>> My name i Sam
```

```
print("My name is\n Sam")
>>> My name is
    Sam

print("My name is\t Sam")
>>> My name is      Sam
```

11.5 – String Functions

There are a lot of string functions in Python and it would be unnecessary and time wasting to talk about all of them in this book. If you want an overview, just go online and look for them.

In this chapter however, we will focus on the most essential, most interesting and most important of these functions. The ones that you might need in the near future.

11.5.1 – Case Manipulating

We have five different case manipulating string functions in Python. Let's have a look at them.

Function	Description
string.upper()	Convert all character to upper case
string.lower()	Convert all character to lower case
string.title()	Convert all character to title case
string.capitalize()	Convert first letter to upper case
string.swapcase()	Swap the case of all letters

11.5.2 – Count Function

Suppose you have a string and you want to know a specific letter or word how many times occurs in that string. In this *count* function allows us to do this.

```
str = "Hey you, listen to me; you are not allowed to go there"
```

```
print(str.count("you"))
>>> 2

print(str.count("e"))
>>> 7
```

11.5.3 – Find Function

This function is little similar to count function. It returns index of first occurrence of any word or letter in a string.

```
str = "Hey you, listen to me; you are not allowed to go there"

print(str.find("you"))
>>> 4
```

11.5.4 – Replace Function

Replace function allows us to replace a specific part of a string by other string. In the following example we are replacing name *Sam* by the name *jack*.

```
text = "My name is Sam"
text = text.replace("Sam", "Jack")
print(text)

>>> My name is Jack
```

11.5.5 – Join Function

With the join function we can join a sequence to a string and separate each element by this particular string.

```
names = ["B", "Sam"]
separator = "-"
print(separator.join(names))

>>> B-Sam
```

11.5.6 – Split Function

If we want to split a string by a specific character, then split function helps us to do this. And interesting thing is it returns a list of those parts of string.

```
name = "Sam,Jack,Axle,Mickel"
name_list = name.split(",")
print(name_list)

>>> ['Sam', 'Jack', 'Axle', ' Mickel']
```

CHAPTER – 12

File Handling

This is the last chapter of this book & in this chapter we will learn how to read or write data from external files. You can ask a question, why are we doing this? The answer is Let see.

12.1 – Opening & Closing Files

If we want to drink water, we need to open the bottle first, right. Like that it need to open file stream first in order to access data of the particular file.

Suppose we have a text file called ‘*myfile.txt*’ and it has some data. Here we will work with that file.

```
Myfile.txt:

This book is the first part of a series that is called the Python 101. In this series, we are going to focus on learning the Python Programming Language in an effective and easy way.
```

Opening File:

```
file = open('myfile.txt', 'r')
```

Here you can see we are simply using *open()* function to open the file stream. As a parameter we need to define the *file name* and the *access mode* (we will talk about that in a second). The function returns the stream and we can save it into our variable & apply different operations on it.

12.1.1 – Access Modes

Whenever we open a file in Python, we use a certain access mode. The access mode defines for which purpose we are opening the file, like reading or writing. There are various access modes in python. Following table will be summarize it batter.

Access Mode	Description
r	Reading Files
r+	Reading & Writing
rb	Reading Binary Files
rb+	Reading & Writing Binary Files
w	Writing into Files
w+	Reading & Writing
wb	Writing into Binary Files
wb+	Reading & Writing Binary Files
a	Appending Files
a+	Reading & Appending
ab	Appending Binary Files
ab+	Reading & Appending Binary Files

Hope you got the idea of access modes. Now time to close the file. When we all done with our file, we need to close the file, otherwise it may through some unwanted errors. To this simply we will use the *close()* function.

```
file = open('myfile.txt', 'r')

# All Necessary Codes

file.close()
```

12.2 – With Statement

Alternatively, we can open and close streams more effectively by using with statements. A with statement opens a stream, executes the indented code and closes the stream afterwards.

```
with open('myfile.txt', 'r') as file:
    # All Necessary Codes
```

If you are lazy like me with statement can help you. Because we don’t need to close the file stream, it will do it itself. Also, using with statement shorten the code and make it more readable.

12.3 – Reading from Files

Once we open the file stream in read mode, we can start reading data from the stream. To do this we need to simply use *read()* method.

```
# Reading in Ordinary Way
file = open('myfile.txt', 'r')
print(file.read())
file.close()

# Reading using with Statement
with open('myfile.txt', 'r') as file:
    print(file.read())
```

Here we are opening file in read mode and print all data of the file using read method.

What if we don’t need the whole data, we need first 50 or 100 characters of file? We can achieve this by passing a parameter to the read method.

```
with open('myfile.txt', 'r') as file:
    print(file.read(50))
```

This code will print first 50 characters of file.

12.3 – Writing into Files

Writing something into a file is almost similar to reading file. Just we need to change two thing, access mode & method. We need to use *write()* method instead of *read()* method in order to write into a file.

Here we are doing this using with statement, you can do both.

```
with open('myfile.txt', 'w') as file:
    file.write("Welcome to Python-101")
```

Here you can see we are passing a string to the write method & that string will be written into our file *(myfile.txt)*. Also, you can pass a variable to the write method instead of passing string directly.

Till now everything seems alright. But there is a huge problem. Every time you run the code it will override the file if the file name is same. So before writing you have to decide what you want. If you want to override every time then go for it. But if you don't want to loss past data then you should use append access mode which id '***a***'.

```
with open('myfile.txt', 'a') as file:
    file.write("Welcome to Python-101")
```

Now it appends the file with data passed as argument, even though the data is same. For every iteration data will be appended into the file if the file name is same.

WHAT NEXT?

Ha-ha you did it! In this book we covered all the fundamentals of Python Programming. Hope now you understand how this language structured and also the basic principle of *Functions, Loops, Conditions, Sequences* etc. Now you can build some of your own program like calculator, Rock-Paper-Scissors Game or other simple applications.

This book is for absolute beginners, as it is just the beginning of ***Python 101*** Series. In the next part we will dive more into Python and learn about *Modules, Classes, OOPs* and many more. Later in this series we'll also learn about Data Science, Machine Learning & other advanced topics.

Last but not the least a little reminder, this book is written for you, so that you can learn the fundamentals of Python and get as much value as possible in easy way. So, if you like the book or think you learnt something new & interesting from this book, please write a quick review on ***Amazon***. It won't take more than one minute and also it is completely free. It will help me to write more high-quality books, which can be beneficial for you.

Thank You…