

STUDENT ADVISORY

Dear Students,

Please be informed that the notes provided by the institute offer a concise presentation of the syllabus. While these notes are helpful for an overview and quick revision, We would strongly suggest that you refer to the prescribed textbooks / Reference book for a comprehensive understanding and thorough preparation of all exams and writing in the examination.

Best regards,

LJ Polytechnic.

પ્રિય વિદ્યાર્થીઓ,

તમને જાણ કરવામા આવે છે કે સંસ્થા દ્વારા પ્રદાન કરવામાં આવેલી નોંધો અભ્યાસક્રમની સંક્ષિપ્ત પ્રસ્તુતિ આપે છે. આ નોંધો વિહંગાવલોકન અને ઝડપી પુનરાવર્તન માટે મદદરૂપ હોઈ શકે છે તેમ છતાં, અમે ભારપૂર્વક સૂચન કરીએ છીએ કે વિદ્યાર્થી તમામ પરીક્ષાઓ અને પરીક્ષામાં લેખનની વ્યાપક સમજણ અને સંપૂર્ણ તૈયારી માટે માત્ર સૂચવેલા પાઠ્યપુસ્તકો/સંદર્ભ પુસ્તકનો સંદર્ભ લો.

એલજે પોલિટેકનિક.

CHAPTER 1- Python Introduction

What is Python?

- **Python** is an open source, dynamic, high-level and interpreted programming language. It supports Object Oriented programming approach to develop applications. It is simple and easy to learn and provides lots of high-level data structures.
- It is a general-purpose language; it means that it can be used to create a variety of different programs and isn't specialized for any specific problems.

History of Python

- Python laid its foundation in 1980s.
- The implementation of Python was started in December 1989 by **Guido Van Rossum** at CWI (Centrum Wiskunde & Informatica) in Netherland.
- In February 1991, **Guido Van Rossum** published the labeled version 0.9.0 to alt.sources.
- In 1994, Python 1.0 was released with new features like lambda, map, filter, and reduce.
- In 2000, Python 2.0 was released with new features such as list comprehensions, garbage collection, cycle detecting and Unicode support.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify the fundamental flaw of the language.
- *ABC programming language* is said to be the predecessor of Python language, which was capable of Exception Handling.

Python Features and Advantages:

- **Easy to Code and Easy to Read**

Python is easy to learn as compared to other programming languages. Its syntax is straight forward and much the same as the English language. There is no use of the semicolon or curly-bracket.

- **Free and Open-Source**

Python is developed under an OSI-approved open-source license. Hence, it is completely free to use, even for commercial purposes. It doesn't cost anything to download Python or to include it in your application.

- **Robust Standard Library**

Python has an extensive standard library available for anyone to use. This means that programmers don't have to write their code for every single thing unlike other programming languages. There are libraries for image manipulation, databases, unit-testing, expressions and a lot of other functionalities.

- **Interpreted**

When a programming language is interpreted, it means that the source code is executed line by line, and not all at once. Programming languages such as C++ or Java are not

interpreted, and hence need to be compiled first to run them. There is no need to compile Python because it is processed at runtime by the interpreter.

- **Portable**

Python is portable in the sense that the same code can be used on different machines. Suppose you write a Python code on a Mac. If you want to run it on Windows or Linux later, you don't have to make any changes to it.

- **Object-Oriented and Procedure-Oriented**

A programming language is object-oriented if it focuses design around data and objects, rather than functions and logic. On the contrary, a programming language is procedure-oriented if it focuses more on functions (code that can be reused). One of the critical Python features is that it supports both object-oriented and procedure-oriented programming.

- **Extensible**

A programming language is said to be extensible if it can be extended to other languages. Python code can also be written in other languages like C++, making it a highly extensible language.

- **Expressive Language**

Python can perform complex tasks using a few lines of code. A simple example, the hello world program you simply type `print ("Hello World")`. It will take only one line to execute, while Java or C takes multiple lines.

- **Support for GUI**

One of the key aspects of any programming language is support for GUI or Graphical User Interface. A user can easily interact with the software using a GUI. Python offers various toolkits, such as Tkinter, wxPython and JPython, which allows for GUI's easy and fast development.

- **Dynamically Typed**

Many programming languages need to declare the type of the variable before runtime. With Python, the type of the variable can be decided during runtime. This makes Python a dynamically typed language.

- **High-level Language**

Python is a high-level programming language because programmers don't need to remember the system architecture, nor do they have to manage the memory. This makes it super programmer-friendly and is one of the key features of Python.

What is use of Python?

Python is commonly used for developing

- websites and software
- task automation
- data analysis
- data visualization etc.

What can we do with Python?

1) Data analysis and machine learning

Python has become a staple in data science, allowing data analysts and other professionals to use the language to conduct complex statistical calculations, create data visualizations, build machine learning algorithms, manipulate and analyze data, and complete other data-related tasks.

Python can build a wide range of different data visualizations, like line and bar graphs, pie charts, histograms, and 3D plots. Python also has a number of libraries that enable coders to write programs for data analysis and machine learning more quickly and efficiently, like TensorFlow and Keras.

2) Web development

Python is often used to develop the back end of a website or application—the parts that a user doesn't see. Python's role in web development can include sending data to and from servers, processing data and communicating with databases, URL routing, and ensuring security. Python offers several frameworks for web development. Commonly used ones include Django and Flask.

3) Automation or scripting

If you find yourself performing a task over and over again, you could work more efficiently by automating it with Python. Writing code used to build these automated processes is called scripting. In the coding world, automation can be used to check for errors across multiple files, convert files, execute simple math, and remove duplicates in data.

4) Software testing and prototyping

In software development, Python can aid in tasks like build control, bug tracking, and testing. With Python, software developers can automate testing for new products or features. Some Python tools used for software testing include Green and Requestium.

5) Everyday tasks

Python isn't only for programmers and data scientists. Learning Python can open new possibilities for those in less data-heavy professions, like journalists, small business owners, or social media marketers. Python can also enable non-programmer to simplify certain tasks in their lives.

Here are just a few of the tasks you could automate with Python:

- Keep track of stock market or crypto prices
- Send yourself a text reminder to carry an umbrella anytime it's raining
- Update your grocery shopping list
- Renaming large batches of files
- Converting text files to spreadsheets
- Randomly assign chores to family members
- Fill out online forms automatically

Python Interpreter, Extension & Implementations

Computers cannot understand code in the way humans write it and hence, you need an interpreter between the computer and the human written code. The job of the interpreter is to convert the code into a format that computers can then understand and process.

The interpreter processes the code in the following ways:

- Processes the Python script in a sequence.
- Compiles the code into a byte code format which is a lower-level language understood by the computers.
- The Python Virtual Machine (PVM) perform over the instructions of low-level byte code to run them one by one.

The Python script is saved with a **.py extension** which informs the computer that it is a Python program script.

Python Installation

Unlike Windows, the Unix based operating systems such as Linux and Mac come with pre-installed Python. Also, the way Python scripts are run in Windows and Unix operating systems differ.

Installing Python on Windows takes a series of few easy steps:

Step 1 – Select Version of Python to Install

Python has various versions available with differences between the syntax and working of different versions of the language. We need to choose the version which we want to use or need.

Step 2 – Download Python Executable Installer

On the web browser, in the official site of python www.python.org, move to the Download for Windows section.

All the available versions of Python will be listed. Select the version required by you and click on Download.

Step 3 – Run Executable Installer

Run the installer. The installation process will take few minutes to complete.

Step 4 – Verify Python is installed on Windows

To ensure if python is successfully installed on your system.

Follow the given steps –

- Open the command prompt.
- Type 'python' and press enter.
- The version of the python which you have installed will be displayed if the python is successfully installed on your windows.

Step 5 – Verify Pip was installed

Pip is a powerful package management system for Python software packages. Thus, make sure that you have it installed.

To verify if pip was installed, follow the given steps –

- Open the command prompt.
- Enter pip -V to check if pip was installed.
- The following output appears if pip is installed successfully.

Note: For all users, especially Windows OS users, it is highly recommended that you install Anaconda, which can be downloaded from <https://www.anaconda.com/>

Python Program Execution

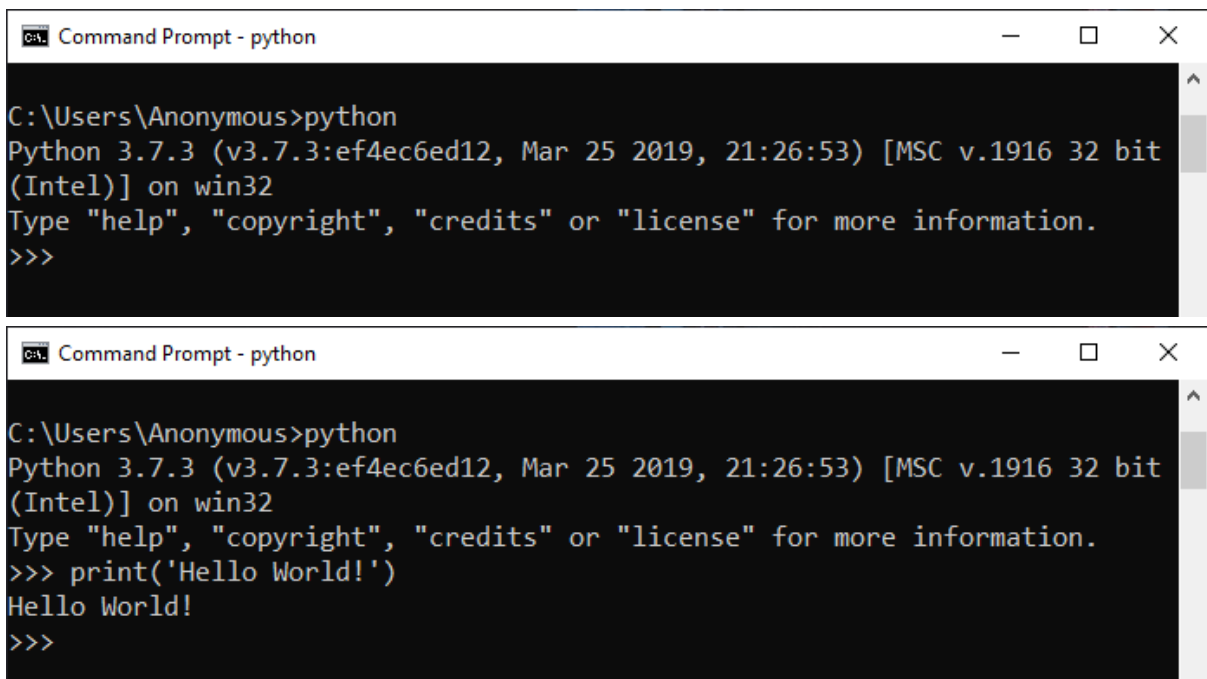
List out Different ways to run Python Script

Here are the ways with which we can run a Python script.

1. Interactive Mode –

We have to write code in command prompt line by line.

To enter in an interactive mode, you will have to open Command Prompt on your windows machine and type 'python' and press Enter.

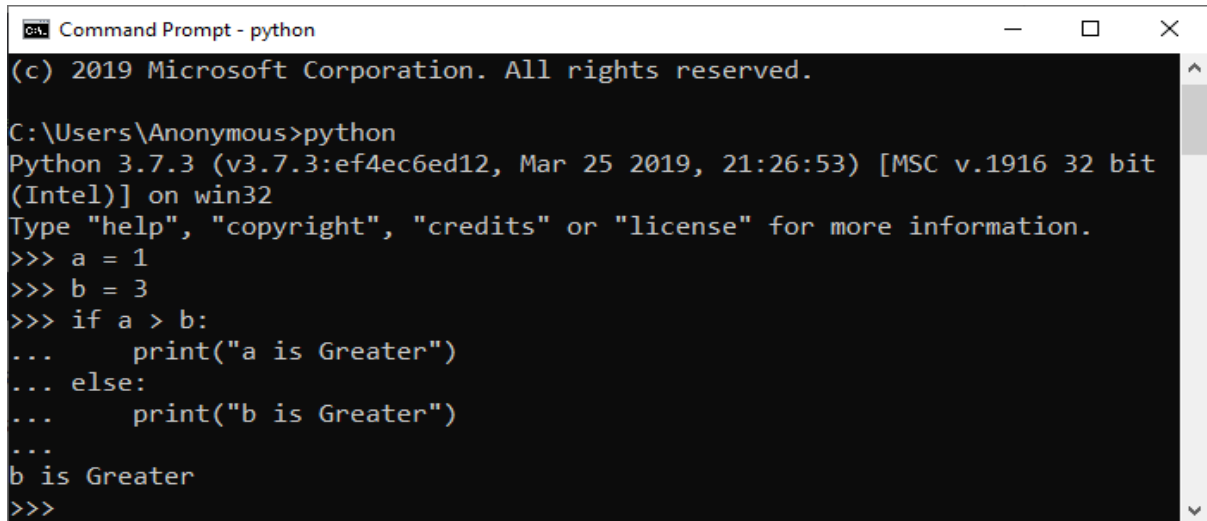


```
C:\Users\Anonymous>python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>

C:\Users\Anonymous>python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello World!')
Hello World!
>>>
```

Run the following line one by one in the interactive mode:

```
a = 1
b = 3
if a > b:
    print ("a is Greater")
else:
    print ("b is Greater")
```



```
Command Prompt - python
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Anonymous>python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 1
>>> b = 3
>>> if a > b:
...     print("a is Greater")
... else:
...     print("b is Greater")
...
b is Greater
>>>
```

On a Mac system, it is very straight-forward. All you need to do is open **Launchpad** and search for **Terminal**, and in the terminal, type **Python** and boom, it will give you an output with the Python version.

Like the Mac system, accessing terminal on a Linux system is also very easy. Right click on the **desktop** and click **Terminal** and in terminal type **Python** and that's all!

2. Command Line –

We have to make python file first and save by .py extension. This file you can run in command prompt by write python first and then your filename.

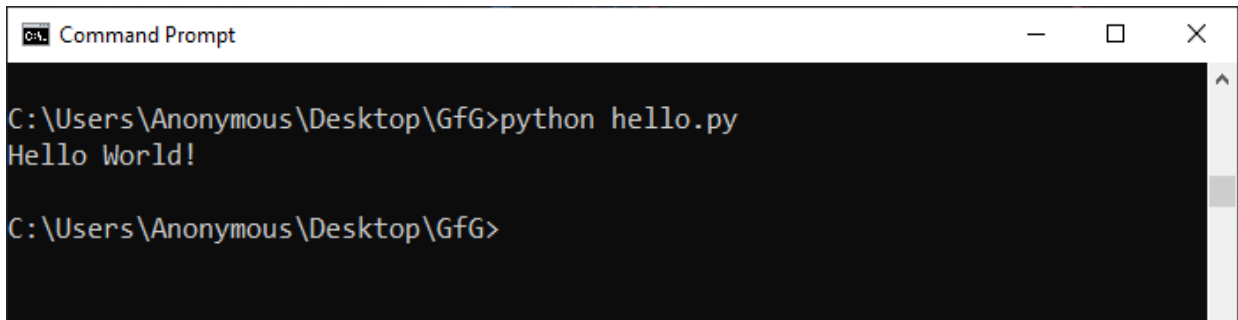
- Create a file having extension .py
- Write Python script in created file
- To run a Python script store in a '**.py**' file in command line, write 'python' keyword before the file name in the command prompt.

```
python hello.py
```

Note: You can write your own file name in place of '**hello.py**'.

```
# File Name: - hello.py
# File Path: - C:\Users\Anonymous\Desktop\GfG\hello.py

print ("Hello World!")
```



```
Command Prompt

C:\Users\Anonymous\Desktop\GfG>python hello.py
Hello World!

C:\Users\Anonymous\Desktop\GfG>
```

3. **Text Editor** (Example: Notepad++, VS Code, etc.)
4. **IDE** (Examples: IDLE, Spyder, Atom, PyCharm, etc.)

Values and Types:

- A **value** is one of the basic things a program works with, like a letter or a number.
- A value may be characters i.e. 'Hello, World!' or a number like 1, 2.2, 0.9 etc.

To print the value *print()* function is used.

Example:

```
>>> print ('Hello, World!')
Hello, World!
>>> print (1)
1
>>> print (2.2)
2.2
>>> print ('Abc@123')
Abc@123
```

To find the type of given value *type()* function is used.

In python we don't have to write int, float, char it will automatically understand the type of given value. So, if we want to find particular type of given value, we can use the type function.

Example:

```
>>> type ('Hello, World!')
<class 'str'>
>>> type (1)
<class 'int'>
>>> type (2.2)
<class 'float'>
>>> print ("Abc@123")
<class 'str'>
```


Variables

Variables are containers for storing data values.

Creating Variables

1. Python has no command for declaring a variable. We just have to write variable name and assign the value in it.
2. A variable is created the moment you first assign a value to it.

Example:

```
x = 5
y = "LJ Polytechnic"
print(x)
print(y)
```

Output:

```
5
LJ Polytechnic
```

3. Variables do not need to be declared with any particular *type*, and can even change type after they have been set. If we use the same name, the variable starts referring to a new value and type.

Example:

```
x = 5
x = "LJ Polytechnic"
print(x)
```

Output:

```
LJ Polytechnic
```

4. Python allows assigning a single value to several variables simultaneously with “=” operators.

Example:

```
x = y = z = 100
print(y)
```

Output:

```
100
```

5. Assigning different values to multiple variables:

Python allows adding different values in a single line with “,” operators.

Example:

```
a, b, c = 1, 2.2, 'ABC'
print(a)
print(b)
print(c)
```

Output:

```
1
2.2
ABC
```

6. How does + operator work with variables?

Example:

```
a = 10
b = 20
print (a + b)
x = 'LJ'
y = 'KU'
print (x + y)
```

Output:

```
30
LJKU
```

7. Can + Operator work with different type variable?

Example:

```
a = 10
b = 'LJ'
print(a + b)
```

Output:

```
TypeError: unsupported
operand type(s) for +: 'int'
and 'str'
```

Rules for creating variables in Python:

1. A variable name must start with a letter or the underscore character.
2. A variable name cannot start with a number.
3. A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
4. Variable names are case-sensitive (name, Name and NAME are three different variables).
5. The reserved words (keywords) cannot be used to naming the variable.

Keywords:

The interpreter uses keywords to recognize the structure of the program, and they cannot be used as variable names. Python reserves **35 keywords**:

and	del	from	none	true	continue	lambda
as	elif	global	nonlocal	try	def	is
assert	else	if	not	while	finally	return
break	except	in	or	with	for	async
class	false	import	pass	yield	raise	wait

String:

- Strings are arrays of bytes representing Unicode characters.
- String data type is most important data type in python.
- Python does not have a character datatype, a single character is simply a string with a length of 1.
- String is not a small. It can also big as million characters.
- Python has also built in functions and algorithms for string.
- String is the combination of multiple characters.
- String index is starting from 0.

Example:

```
s = 'P'
print(s)
type(s)

language = 'Python'
print(language)
type(language)
```

Output:

```
P
<class 'str'>
Python
<class 'str'>
```

String Indexing:

[0]	[1]	[2]	[3]	[4]	[5]
P	Y	T	H	O	N
[-6]	[-5]	[-4]	[-3]	[-2]	[-1]

- Square brackets can be used to access elements of the string.

Example:

```
language = 'Python'
letter = language[1]
print(letter)

print(language[0])
print(language[-1])
print(language[1.5])
```

Output:

```
y
P
n
TypeError: string
indices must be integers
```

Length of a String:

- To find the length of the string *len()* function is used.
- *len()* is a built-in function that returns the number of characters in a string:

Example:

```
language = 'Python'
print(len(language))
```

Output:

```
6
```

- To get the last letter of a string, you might be tempted to try something like this:
Because if you write simply length, then length will start from 0 so it can't display last digit.

Example:

```
x = "Python"
length = len(x)
y=x[length-1]
print(y)
```

Output:

```
n
```

String Slices:

A segment of a string is called a *slice*. Selecting a slice is similar to selecting a character:

We can specify *start*, *stop* and *step (optional)* within the square brackets as:
string[start: stop: step]

- **start:** It is the index from where the slice starts. The default value is **0**.
- **stop:** It is the index at which the slice stops. The character at this index is not included in the slice. The default value is the *length of the string*.
- **step:** It specifies the number of jumps to take while going from start to stop. It takes the default value of **1**.

Example:

```
s = 'Welcome to LJKU'
print(s[0:8])
print(s[11:16])

# By default starting index is "0" and middle index in "length of string"
print(s[:])
print(s[0:])

# If we can write same digit at both place then it will not give us any output
print(s[3:3])

# Replace any latter with any index of given string.
r = 'J' + s[1:]
print(r)

# If we write -1 at step place it will give output as reverse of string
print(s[: :-1])

# For step jumping
print(s[0: :2]) # it will give output as skip 1 character from starting of string

print(s[: :-2]) # it will first reverse the string and skip 1 character after reverse.
```

Output:

```
Welcome
LJKU
Welcome to LJKU
Welcome to LJKU

Jelcome to LJKU
UKJL ot emocleW
Wloet JU
UJ teolW
```

String Methods:

Everything in Python is an *object*. A *string* is an object too. Python provides us with various methods to *call on* the string object.

Note: Note that none of these methods *alters* the actual string. They instead *return a copy* of the string. This copy is the *manipulated version* of the string.

capitalize()

This method is used to *capitalize* a string.

```
>>> s = 'heLlo pYthOn'
>>> s2 = s.capitalize()
>>> print(s2)
Hello python
```

lower()

This method converts all alphabetic characters to *lowercase*.

```
>>> s = 'heLlo pYthOn'
>>> s2 = s.lower()
>>> print(s2)
hello python
```

upper()

This method converts all alphabetic characters to *uppercase*.

```
>>> s = 'heLlo pYthOn'
>>> s2 = s.upper()
>>> print(s2)
HELLO PYTHON
```

title()

This method converts the *first letter* of each word to *uppercase* and remaining letters to *lowercase*.

```
>>> s = 'heLlo pYthOn'
>>> s2 = s.title()
>>> print(s2)
Hello Python
```

swapcase([<chars>])

This method *swaps case* of all alphabetic characters.

```
>>> s = 'HeLlo PyThOn'
>>> s2 = s.swapcase ()
>>> print(s2)
hElLo pYtHoN
```

count(<sub>, <start>, <end>)

This method returns the *number of time* <sub> occurs in string.

```
>>> s = 'Python Programming'
>>> s.count('P')
2
>>> s.count('mm')
1
>>> s.count('P',3,10)
1
```

find(<sub>, <start>, <end>)

This method returns the index of the *first occurrence* of <sub> in s. Returns -1 if the substring is not present in the string.

```
>>> s = 'Python Programming'
>>> s.find('Pro')
7
>>> s.find('on',3,10)
```

lstrip()

This method removes *leading characters* from a string. Removes *leading whitespaces* if you don't provide a <chars> *argument*.

```
>>> s = '  Python  '
>>> s.lstrip()
'Python  '
```

rstrip()

This method removes *trailing characters* from a string. Removes *trailing whitespaces* if you don't provide a <chars> *argument*.

```
>>> s = '  Python  '
>>> s.rstrip()
'  Python'
```

strip()

This method removes *leading and trailing characters* from a string. Removes *leading and trailing whitespaces* if you don't provide a *<chars> argument*

```
>>> s = '  Python  '
>>> s.strip()
'Python'
```

join(<iterable>)

This method returns the string concatenated with the elements of iterable.

```
>>> s = ('We', 'are', 'coders')
>>> s1 = '_.join(s)
print(s1)
We are coders
```

split(sep=None, maxsplit=-1)

This method *splits* a string into a list of *substrings* based on sep. If you don't pass a value to sep, it splits based on *whitespaces*.

```
>>> s = 'Python Programming'
>>> s.split()
['Python', 'Programming']
>>> s = 'Python Programming'
>>> s.split('o')
['Pyth', 'n Pr', 'gramming']
```

String Concatenation

Strings can be *concatenated* (glued together) with the + operator, and repeated with *:

```
>>> 3*'un'
'ununun'
>>> 'Py'+'thon'
'Python'
>>> 'Py'+'thon'
'Python'
```

Escape Characters

To insert characters that are illegal in a string, use an escape character. An escape character is a *backslash* (\) followed by the character you want to insert.

```
>>>s = "we are the so-called "Vikings" from the north"
SyntaxError: invalid syntax
```

To fix this problem, use the escape character \"

```
>>>s = "we are the so-called \"Vikings\" from the north"
'we are the so-called "Vikings" from the north'
```

List of escape sequences available in Python 3:

Code	Result	Code	Result
\'	Single Quote	\t	Tab
\\	Backslash	\b	Backspace
\n	New Line	\f	Form Feed
\r	Carriage Return	\ooo	Octal value
\xhh	Hex value		

Numbers

There are three numeric types in Python:

- int
- float
- complex

Variables of numeric types are created when you assign a value to them:

```
>>>x = 1 #int
>>> y = 2.8 #float
>>> z = 1j #complex
>>>print(type(x))
<class 'int'>
>>>print(type(y))
<class 'float'>
>>>print(type(z))
<class 'complex'>
```


Int

Integer is a whole number, positive or negative, without decimals, of unlimited length.

```
>>>x = 1
>>>y = 35656222554887711
>>>z = -3255522
>>>print(type(x))
<class 'int'>
>>>print(type(y))
<class 'int'>
>>>print(type(z))
<class 'int'>
```

Float

Float or "floating point number" is a number, positive or negative, containing one or more decimals. Float can also be scientific numbers with an "e" to indicate the power of 10.

```
>>>x = 1.10
>>>y = 1.0
>>>z = -35.59
>>>print(type(x))
<class 'float'>
>>>print(type(y))
<class 'float'>
>>>print(type(z))
<class 'float'>
```

```
>>>x = 35e3
>>>y = 12E4
>>>z = -87.7e100
>>>print(type(x))
<class 'float'>
>>>print(type(y))
<class 'float'>
>>>print(type(z))
<class 'float'>
```

Complex

Complex numbers are written with a "j" as the imaginary part:

```
x = 3+5j
y = 5j
z = -5j
print(type(x))
print(type(y))
print(type(z))
```

```
<class 'complex'>
<class 'complex'>
<class 'complex'>
```

Type Conversion

You can convert from one type to another with the `int()`, `float()`, and `complex()` methods.

Example:

```
x = 1 # int
y = 2.8 # float
z = 1j # complex
a = float(x) # convert from int to float
b = int(y) #convert from float to int
c = complex(x) #convert from int to complex
print(a)
print(type(a))
print(b)
print(type(b))
print(c)
print(type(c))
```

Output:

```
1.0
<class 'float'>
2
<class 'int'>
(1+0j)
<class 'complex'>
```

Operators

Operators in general are used to perform operations on values and variables. These are standard symbols used for the purpose of logical and arithmetic operations.

Arithmetic Operators

Arithmetic operators are used to performing mathematical operations like addition, subtraction, multiplication, and division.

<i>Operator</i>	<i>Description</i>	<i>Syntax</i>
+	Addition: adds two operands	$x + y$
-	Subtraction: subtracts two operands	$x - y$
*	Multiplication: multiplies two operands	$x * y$
/	Division (float): divides the first operand by the second	x / y
//	Division (floor): divides the first operand by the second	$x // y$
%	Modulus: returns the remainder when the first operand is $x \% y$ divided by the second	
**	Power: Returns first raised to power second	$x ** y$

Example:**Output:**

```

a = 21
b = 10
c = 0
c = a + b
print("Addition is:", c)
c = a - b
print('Subtraction is:', c)
c = a * b
print('Multiplication is:', c)
c = a / b
print('Division is:', c)
c = a % b
print("Reminder of a/c is:", c)
p = 2
q = 3
r = p**q
print("result of p^q is:", r)
x = 11
y = 5
z = a/b
print("result of a divide b is:", z)

```

```

Addition is: 31
Subtraction is: 11
Multiplication is: 210
Division is: 2.1
Reminder of a/c is: 1
result of p^q is: 8
result of a divide b is: 2

```

Comparison Operators

Comparison of Relational operators compares the values. It either returns True or False according to the condition.

<i>Operator</i>	<i>Description</i>	<i>Syntax</i>
>	Greater than: True if the left operand is greater than the right	<code>x > y</code>
<	Less than: True if the left operand is less than the right	<code>x < y</code>
==	Equal to: True if both operands are equal	<code>x == y</code>
!=	Not equal to – True if operands are not equal	<code>x != y</code>
>=	Greater than or equal to True if the left operand is greater than or equal to the right	<code>x >= y</code>
<=	Less than or equal to True if the left operand is less than or equal to the right	<code>x <= y</code>

Example:

```
>>> # Examples of Relational Operators
>>> a, b = 13, 33

>>> # a > b is False
>>> print(a > b)
False

>>> # a < b is True
>>> print(a < b)
True

>>> # a == b is False
>>> print(a == b)
False

>>> # a != b is True
>>> print(a != b)
True
```

Logical Operators

Logical operators perform Logical AND, Logical OR, and Logical NOT operations. It is used to combine conditional statements.

<i>Operator</i>	<i>Description</i>	<i>Syntax</i>
and	Logical AND: True if both the operands are true	x and y
or	Logical OR: True if either of the operands is true	x or y
not	Logical NOT: True if the operand is false	Not x

Example:

```
x = 5
print(x>3 and x>10) #and
print (x>3 or x>10) #or

print(not(x > 3 and x > 10))
# returns False because not is used to reverse the
result
```

Output:

```
False
True

True
```

Bitwise Operators

Bitwise operators act on bits and perform the bit-by-bit operations. These are used to operate on binary numbers.

<i>Operator</i>	<i>Description</i>	<i>Syntax</i>
& Bitwise AND	Operator copies a bit to the result if it exists in both operands	$x \& y$ (means 0000 1100)
 Bitwise OR	It copies a bit if it exists in either operand.	$x y = 61$ (means 0011 1101)
~ Bitwise NOT	It copies the bit if it is set in one operand but not both.	$\sim x$
^ Bitwise XOR	It is unary and has the effect of 'flipping' bits.	$x \wedge y$
>> Bitwise right shift	The left operands value is moved left by the number of bits specified by the right operand.	$x \gg$
<< Bitwise left shift	The left operands value is moved right by the number of bits specified by the right operand.	$x \ll$

Example:

```
>>># Examples of Bitwise operators
a = 60      # 60 = 0011 1100
b = 13      # 13 = 0000 1101
c = 0

c = a & b    # 12 = 0000 1100
print('Answer of a & b is:', c)

c = a | b    # 61 = 0011 1101
print('Answer of a | b is:', c)

c = a ^ b    # 49 = 0011 0001
print('Answer of a ^ b is:', c)

c = ~a      # -61 = 1100 0011
print('Answer of ~60 is:', c)

c = a << 2   # 240 = 1111 0000
print('Answer of a<<2 is:', c)
```

Output:

```
Answer of a & b is: 12
Answer of a | b is: 61
Answer of a ^ b is: 49
Answer of ~60 is: -61
Answer of a<<2 is: 240
Answer of a>>2 is: 15
```

Assignment Operators

Assignment operators are used to assigning values to the variables.

<i>Operator</i>	<i>Description</i>	<i>Syntax</i>
=	Assign value of right side of expression to left side operand	x = y + z
+=	Add AND: Add right-side operand with left side operand and then assign to left operand	a +=b a=a+b
-=	Subtract AND: Subtract right operand from left operand and then assign to left operand	a -=b a=a - b
*=	Multiply AND: Multiply right operand with left operand and then assign to left operand	a *=b a=a * b
/=	Divide AND: Divide left operand with right operand and then assign to left operand	a /=b a=a / b
%=	Modulus AND: Takes modulus using left and right operands and assign the result to left operand	a %=b a=a % b
//=	Divide(floor) AND: Divide left operand with right operand and then assign the value(floor) to left operand	a //=b a=a // b
**=	Exponent AND: Calculate exponent(raise power) value using operands and assign value to left operand	a **=b a=a ** b
&=	Performs Bitwise AND on operands and assign value to left operand	a &=b a=a&b
=	Performs Bitwise OR on operands and assign value to left operand	a =b a=a b
^=	Performs Bitwise XOR on operands and assign value to left operand	a ^=b a=a ^ b
>>=	Performs Bitwise right shift on operands and assign value to left operand	a >>=b a=a>>b
<<=	Performs Bitwise left shift on operands and assign value to left operand	a <<=b a=a<<b

Identity Operators

is and *is not* are the identity operators both are used to check if two values are located on the same part of the memory. Two variables that are equal do not imply that they are identical.

<i>Operator</i>	<i>Description</i>
is	True if the operands are identical
is not	True if the operands are not identical

```
>>> a = 10
>>> b = 20
>>> c = a
>>> print (a is not b)
True
>>> print (a is c)
True
```

Membership Operators

in and *not in* are the membership operators; used to test whether a value or variable is in a sequence.

<i>Operator</i>	<i>Description</i>
in	True if value is found in the sequence
in not	True if value is not found in the sequence

Example:

```
#membership_ex.py
x = 24
y = 20
list = [10, 20, 30, 40, 50]

if (x not in list):
    print("x is NOT present in given list")
else:
    print("x is present in given list")

if (y in list):
    print("y is present in given list")
else:
    print("y is NOT present in given list")
```

Output:

```
x is NOT present in given
list
y is present in given list
```

Precedence and Associativity of Operators

Operator precedence and associativity determine the priorities of the operator.

Operator Precedence

This is used in an expression with more than one operator with different precedence to determine which operation to perform first.

```
>>> # Precedence of '+' & '*'
>>> expr = 10 + 20 * 30
>>> print(expr)
610

>>> # Precedence of 'or' & 'and'
>>> name = "Alex"
>>> age = 0

>>> if name == "Alex" or name == "John" and age >= 2:
...     print ("Hello! Welcome.")
... else:
...     print ("Good Bye!!")
Hello! Welcome.
```

Operator Associativity

If an expression contains two or more operators with the same precedence then Operator Associativity is used to determine. It can either be Left to Right or from Right to Left.

CHAPTER 2- Control flow & Function in Python

Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behaviour of the program accordingly. *Conditional statements* give us this ability.

if statement

The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block. The condition of if statement can be any valid logical expression which can be either evaluated to true or false.

The syntax of the if-statement is given below.

```
if (condition):  
    #statement1  
    #extra statements....  
#statement2  
  
# Here if the condition is true, if block  
# will consider only statement1 and extra statements to be inside its block.  
# statement 2 is outside of if block.
```

Example:-

```
# Python program to illustrate If statement  
  
i = 10  
if (i>15):  
    print("10 is less than 15")  
print("I am not in if block")
```

I am not in if

As the condition present in the if statement is false. So, the block below the if statement is not executed.

if-else

The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.

If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.

```
if (condition):  
    # Executes this block if  
    # condition is true  
else:  
    # Executes this block if  
    # condition is false
```

Python program to illustrate If else statement

```
I = 20  
if (I < 15):  
    print("I is smaller than 15")  
    print("I'm in if Block")  
else:  
    print("I is greater than 15")  
    print("I'm in else Block")  
print("I'm not in if and not in else Block")
```

```
I is greater than 15  
I'm in else Block  
I'm not in if and not in else Block
```

The block of code following the else statement is executed as the condition present in the if statement is false after calling the statement which is not in block (without spaces).

Nested-if

A nested if is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement. Python allows us to nest if statements within if statements. i.e., we can place an if statement inside another if statement.

```
if (condition1):  
    # Executes when condition1 is true  
    if (condition2):  
        # Executes when condition2 is true  
        # if Block is end here  
    # if Block is end here
```

```
# Python program to illustrate nested If statement
i = 10
if (i == 10):
    # First if statement
    if (i < 15):
        print("i is smaller than 15")

    # Nested - if statement
    # Will only be executed if statement above
    # it is true
    if (i < 12):
        print("i is smaller than 12 too")
    else:
        print("i is greater than 15")
```

```
i is smaller than 15
i is smaller than 12 too
```

If-elif-else ladder

The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

```
if (condition):
    statement
elif (condition):
    statement
.
.
else:
    statement
```

```
# Python program to illustrate if-elif-else ladder
```

```
i = 15
if (i == 10):
    print("i is 10")
elif (i == 15):
    print("i is 15")
else:
    print("i is not present")
```

```
i is 15
```

Looping statements:

In general, statements are executed sequentially: The first statement in a function 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.

Programming languages provide various control structures that allow for more complicated execution paths.

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

while loop

In python, while loop is used to execute a block of statements repeatedly until a given a condition is satisfied. And when the condition becomes false, the line immediately after the loop in program is executed.

```
while (expression) :  
    statement(s)
```

All the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

```
# Python program to illustrate while loop  
count = 0  
while count < 3 :  
    count = count + 1  
    print("Hello World")
```

```
Hello World  
Hello World  
Hello World
```

As discussed above, while loop executes the block until a condition is satisfied. When the condition becomes false, the statement immediately after the loop is executed.

The else clause is only executed when your while condition becomes false. If you break out of the loop, or if an exception is raised, it won't be executed.

```
#Python program to illustrate combining else with while  
count = 0  
while (count < 1):  
    count = count + 1  
    print("Hello World")  
else:  
    print("In else block")
```

```
Hello World  
In else block
```

for loop

For loops are used for sequential traversal. For example: traversing a list or string or array etc. In Python, there is no C style for loop, i.e., for (i=0; i<n; i++). It can be used to iterate over a range and iterators.

```
for variable_name in range (start, stop, step):  
    statement(s)
```

range() function

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

```
range (start, stop, step)
```

Parameter Description

Start	Optional. An integer number specifying at which position to start. Default is 0
--------------	--

Stop	Required. An integer number specifying at which position to stop (not included).
-------------	--

Step	Optional. An integer number specifying the incrementation. Default is 1
-------------	--

```
# Python program to illustrate Iterating over range 0 to n-1
```

```
n = 4  
for i in range(0, n):  
    print(i)
```

```
0  
1  
2  
3
```

```
# Using range to print number divisible by 3  
for i in range(0, 30, 3):  
    print(i, end=" ")
```

```
0 3 6 9 12 15 18 21 24 27
```

If a user wants to decrement, then the user needs steps to be a negative number. For example:

```
# Python program to decrement with range()
# incremented by -2

for i in range(25, 2, -2):
    print(i, end=" ")
```

25 23 21 19 17 15 13 11 9 7 5 3

Using else statement with for loops

We can also combine else statement with for loop like in while loop. But as there is no condition in for loop based on which the execution will terminate so the else block will be executed immediately after for block finishes execution.

```
# Python program to illustrate combining else with for

list = ["ABC", "BCD", "CDE"]
for index in range(len(list)):
    print(list[index])
else:
    print("Inside Else Block")
```

ABC
BCD
CDE
Inside Else Block

Nested loops

Python programming language allows to use one loop inside another loop.

```
# Python program to illustrate nested for loops

for i in range(1, 5):
    for j in range(i):
        print(i, end=' ')
    print()
```

1
2 2
3 3 3
4 4 4 4

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.

Continue Statement: It returns the control to the beginning of the loop

```
# Prints all letters except 'e' and 'o'

for letter in 'Python':
    if letter == 'e' or letter == 'o':
        continue
    print('Current Letter :', letter)
```

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : n
```

Break Statement: It brings control out of the loop for given condition.

```
# Break the loop as soon it sees 't'

for letter in 'Python':
    if letter == 't':
        break
    print('Current Letter :', letter)
```

```
Current Letter : P
Current Letter : y
```

Pass Statement: We use pass statement to write empty loops. Pass is also used for empty control statement, function and classes.

```
# An empty loop

for letter in 'Python':
    pass
print('Last Letter :', letter)
```

```
Last Letter : n
```

Functions

Python Functions is a block of related statements designed to perform a computational, logical, or evaluative task. The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again. Functions can be either built-in or user-defined. It helps the program to be concise, non-repetitive, and organized.

The syntax of the user-defined function is given below.

```
def function_name(parameters):  
    statement(s)  
    return expression
```

Creating a function

We can create a Python function using the **def** keyword.

```
# A simple Python function  
  
def fun():  
    print("Python Programming")
```

Calling a function

After creating a function we can call it by using the name of the function followed by parenthesis containing parameters of that particular function.

```
# A simple Python function  
  
def fun():  
    print("Python Programming")  
  
# Driver code to call a function  
fun()
```

```
Python Programming
```

Local and Global variables in Python

Local variables in Python are initialized inside a function and belong only to that particular function. It cannot be accessed anywhere outside the function.


```
def local():  
    s="Python local variable"  
    print(s)  
local()
```

Python local variable

```
def local():  
    s="Python local variable"  
    print("Inside: ", s)  
local()  
print("Outside: ", s)
```

Inside: Python local variable

Traceback (most recent call last):

File "D:\LJ\Python\Code\LJ\Chapter 2\local global.py", line 12, in <module>

print("Outside: ", s)

NameError: name 's' is not defined

Global variables in Python are outside any function and accessible throughout the program.

i.e., inside and outside of every function.

```
def glob():  
    print("Inside: ", s)  
s="Python global variable"  
glob()  
print("Outside: ", s)
```

Inside: Python global variable
Outside: Python global variable

Types of Arguments

Python supports various types of arguments that can be passed at the time of the function call. Let's discuss each type in detail.

Default arguments

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument. The following example illustrates Default arguments.

```
# Python program to demonstrate default arguments

def myFun(x, y=50):
    print("x: ", x)
    print("y: ", y)

# Driver code (We call myFun() with only argument)
myFun(10)
```

```
x: 10
y: 50
```

Keyword arguments

The idea is to allow the caller to specify the argument name with values so that caller does not need to remember the order of parameters.

```
# Python program to demonstrate Keyword Arguments

def student(firstArg, lastArg):
    print(firstArg, lastArg)

# Keyword arguments
student(firstArg='Python', lastArg='Practice')
student(lastArg='Practice', firstArg='Python')
```

```
Python Practice
Python Practice
```

Built-in function:

<i>Function</i>	<i>Description</i>
<i>len()</i>	Returns the number of items in an object
<i>min()</i>	Returns a smallest item
<i>max()</i>	Returns a largest item
<i>pow()</i>	Returns the value of x to the power of y. [x ^y]
<i>lambda()</i>	It can take any number of arguments, but can only have one expression.

```
#len()
```

```
x = (3,12,9,6,15)
print ("length of x: ", len(x))
y = ("Hello", "How", "Happy")
print ("length of y: ", len(y))
z = ("Hello")
print ("length of z: ", len(z))
```

```
length of x: 5
length of y: 3
length of z: 5
```

```
#min()
```

```
x = min (3,12,9,6,15)
print ("minimum value: ", x)
y = min ("Ujjain", "Ahmedabad", "Mumbai", "Surat")
print ("minimum value: ", y)
```

```
minimum value: 3
minimum value: Ahmedabad
```

```
#max()
```

```
x = max (3,12,9,6,15)
print ("maximum value: ", x)
y = max ("Hello", "How", "Happy")
print ("maximum value: ", y)
```

```
maximum value: 15
maximum value: How
```

pow() function:

It returns the value of x to the power of y. [x^y]. If a third parameter is present, it returns x to the power of y, modulus z.

```
pow (x, y, z)
```

Return the value of x to the power of y, modulus z (same as $(x ** y) \% z$)

Parameters Description

<i>x</i>	A number, the base
<i>y</i>	A number, the exponent
<i>z</i>	Optional. A number, the modulus

```
# pow()
```

```
x = pow(2, 3)
print ("power of 2 raise to 3: ", x)
x = pow(2, 3, 3)
print ("power of 2 raise to 3, modulus 3: ", x)
```

```
power of 2 raise to 3: 8
power of 2 raise to 3, modulus 3: 2
```

lambda() function:

A lambda function is a small anonymous function. It can take any number of arguments, but can only have one expression. Anonymous function are not defined using def keyword rather they are defined using lambda keyword.

```
lambda arguments : expression
```

```
# lambda()
```

```
x = lambda a : a + 10
print("addition:", x(5))

y = lambda m, t : print("multiplication:", m*10)
(y(2, 2))

z=lambda p, q : p * q + p / p
print("operation:", z(2, 2))
```

```
addition: 15
multiplication: 20
operation: 5.0
```

CHAPTER 3- Data Structures in Python

Iterators

An iterator is an object that contains a countable number of values. It is an object that can be iterated upon, meaning that you can traverse through all the values.

Python program to illustrate iterating over a list

```
print("List Iteration")
l = ["ABC", "BCD", "CDE"]
for i in l:
    print(i)
```

List Iteration

ABC
BCD
CDE

Iterating over a tuple (immutable)

```
print("Tuple Iteration")
t = ("ABC", "BCD", "CDE")
for i in t:
    print(i)
```

Tuple Iteration

ABC
BCD
CDE

Iterating over a String

```
print("String Iteration")
s = "ABCDE"
for i in s:
    print(i)
```

String Iteration

A
B
C
D
E

Python has four basic inbuilt data structures namely *Lists*, *Tuple*, *Dictionary* and *Set*.

List

Like a string, a *list* is a sequence of values. In a string, the values are characters; in a list, they can be *any type*. The values in list are called *elements* or sometimes *items*.

There are several ways to create a new list; the simplest is to enclose the elements in square brackets (“[” and “]”)

```
# Creating a List
```

```
List = [ ]  
print("Blank List: ", List)
```

```
Blank List:[ ]
```

```
# Creating a List of numbers
```

```
List = [10, 20, 30]  
print("List of numbers: ")  
print(List)
```

```
List of numbers:  
[10, 20, 30]
```

```
# Creating a List of strings and accessing using index
```

```
List = ["Programming", "in", "Python"]  
print("List Items: ")  
print(List[0])  
print(List[2])
```

```
List Items:  
Programming  
Python
```

```
# Creating a Multi-Dimensional List (By Nesting a list inside a List)
```

```
List = [['Programming', 'in'], ['Python']]  
print("Multi-Dimensional List: ")  
print(List)
```

```
Multi-Dimensional List:  
[['Programming', 'in'], ['Python']]
```

A list may contain duplicate values with their distinct positions and hence, multiple distinct or duplicate values can be passed as a sequence at the time of list creation.

#Creating a List with the use of Numbers/(Having duplicate values)

```
List = [1, 2, 4, 4, 3, 3, 3, 6, 5]
print("List with the use of Numbers: ")
print(List)
```

List with the use of Numbers:
[1, 2, 4, 4, 3, 3, 3, 6, 5]

Creating a List with mixed type of values (Having numbers and strings)

```
List = [1, 2, 'Programming', 4, 'in', 6, 'Python']
print("List with the use of Mixed Values: ")
print(List)
```

List with the use of Mixed Values:
[1, 2, 'Programming', 4, 'in', 6, 'Python']

Unlike strings, lists are mutable because you can change the order of items in a list or reassign an item in a list. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

Updating a List elements (List are mutable)

```
Numbers=[17,123]
Print("Before:", Numbers)
Numbers[1]=5
Print ("After:", Numbers)
```

Before: [17, 123]
After: [17,5]

Using **len()** function we can find the length (no. of elements in list) of list.

Creating a List of numbers and finding the length

```
List = [10, 20, 14]
print(len(List))
```

3

List operation

The + operator concatenates lists

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print(c)
```

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

The * operator concatenates lists

```
a = [1]
a = a * 3
print(a)
```

```
[1, 1, 1]
```

List slices

The slice operator

```
List = ['a', 'b', 'c', 'd', 'e', 'f']
print(List[1:3])
print(List[:4])
print(List[3:])
```

```
['b', 'c']
['a', 'b', 'c', 'd']
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So, if you omit both, the slice is a copy of the whole list.

```
List = ['a', 'b', 'c', 'd', 'e', 'f']
print(List[:])
```

```
['a', 'b', 'c', 'd', 'e', 'f']
```


A slice operator on the left side of an assignment can update multiple elements.

```
List = ['a', 'b', 'c', 'd', 'e', 'f']  
List[1:3] = ['x', 'y']  
print(List)
```

```
['a', 'x', 'y', 'd', 'e', 'f']
```

List methods

Python has a set of built-in methods that you can use on lists.

<i>Method</i>	<i>Description</i>
---------------	--------------------

<i>append()</i>	Adds an element at the end of the list
-----------------	--

<i>clear()</i>	Removes all the elements from the list
----------------	--

<i>copy()</i>	Returns a copy of the list
---------------	----------------------------

<i>count()</i>	Returns the number of elements with the specified value
----------------	---

<i>extend()</i>	Add the elements of a list (or any iterable), to the end of the current list
-----------------	--

<i>index()</i>	Returns the index of the first element with the specified value
----------------	---

<i>insert()</i>	Adds an element at the specified position
-----------------	---

<i>pop()</i>	Removes the element at the specified position
--------------	---

<i>remove()</i>	Removes the first item with the specified value
-----------------	---

<i>reverse()</i>	Reverses the order of the list
------------------	--------------------------------

<i>sort()</i>	Sorts the list
---------------	----------------

```
# append()
```

```
fruits = ['apple', 'banana', 'cherry']  
fruits.append("orange")  
print(fruits)
```

```
['apple', 'banana', 'cherry', 'orange']
```

```
# clear()
```

```
fruits = ['apple', 'banana', 'cherry', 'orange']  
fruits.clear()  
print(fruits)
```

```
[]
```

```
# copy()
```

```
fruits = ['apple', 'banana', 'cherry', 'orange']  
x = fruits.copy()  
print(x)
```

```
['apple', 'banana', 'cherry', 'orange']
```

```
# count()
```

```
fruits = ['apple', 'banana', 'cherry']  
x = fruits.count("cherry")  
print(x)
```

```
1
```

```
#extend()
```

```
fruits = ['apple', 'banana', 'cherry']  
cars = ['Ford', 'BMW', 'Volvo']  
fruits.extend(cars)  
print(fruits)
```

```
['apple', 'banana', 'cherry', 'Ford', 'BMW', 'Volvo']
```

```
#index()
```

```
fruits = ['apple', 'banana', 'cherry']  
x = fruits.index("cherry")  
print(x)
```

```
2
```

```
#insert()
```

```
fruits = ['apple', 'banana', 'cherry']  
fruits.insert(1, "orange")  
print(fruits)
```

```
['apple', 'orange', 'banana', 'cherry']
```

```
#pop()
```

```
fruits = ['apple', 'banana', 'cherry']  
fruits.pop(1)  
print(fruits)
```

```
['apple', 'cherry']
```

```
#remove()
```

```
fruits = ['apple', 'banana', 'cherry']  
fruits.remove("banana")  
print(fruits)
```

```
['apple', 'cherry']
```

```
#reverse()
```

```
fruits = ['apple', 'banana', 'cherry']  
fruits.reverse()  
print(fruits)
```

```
['cherry', 'banana', 'apple']
```

```
#sort()
```

```
cars = ['Ford', 'BMW', 'Volvo']  
cars.sort()  
print(cars)
```

```
['BMW', 'Ford', 'Volvo']
```

Aliasing and Cloning:

Aliasing happens whenever the value of one variable is assigned to another variable. It makes python memory efficient. Aliasing occurs in Python when two variables reference the same object in memory. Therefore, changes made through one reference will be visible through the other.

```
a = [1, 2, 3]
b = a
print (a is b)

b[1] = 4
print("list of a:", a)
print("list of b:", b)
```

```
True
list of a: [1, 4, 3]
list of b: [1, 4, 3]
```

Cloning a list, creates a copy of the original list. The new list can be changed without changing the original.

slicing Technique

```
a = [1, 2, 3]
b = a[:]
print ("original list of a:", a)
print ("original list of b:", b)

a[0] = 10
print ("updated list of a:", a)
print ("updated list of b:", b)
```

```
original list of a: [1, 2, 3]
original list of b: [1, 2, 3]
updated list of a: [10, 2, 3]
updated list of b: [1, 2, 3]
```

copy() method

```
a = [1, 2, 3]
b = a.copy()
print ("original list of a:", a)
print ("original list of b:", b)

b[0] = 10
print ("updated list of a:", a)
print ("updated list of b:", b)
```

```
original list of a: [1, 2, 3]
original list of b: [1, 2, 3]
updated list of a: [1, 2, 3]
updated list of b: [10, 2, 3]
```

Tuples

A tuple is a sequence of *immutable* (A tuple is a collection which is ordered and unchangeable) Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example

```
# creating a tuple
```

```
tup1 = ('ABC', 'pqr', 1000, 2000)
tup2 = (1, 2, 3, 4, 5)
tup3 = "a", "b", "c", "d"
print(tup1)
print(tup2)
print(tup3)
```

```
('ABC', 'pqr', 1000,
2000)
(1, 2, 3, 4,
```

The empty tuple is written as two parentheses containing nothing

```
tup1 = ()
```

To write a tuple containing a single value you have to include a comma, even though there is only one value

```
tup1 = (50,)
```

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

Nested Tuples

A tuple written inside another tuple is nested tuple.

```
#nested tuple
```

```
Tup = ((1, "abc", 2000), (2, "xyz", 2500))
print (Tup[0][0])
print (Tup[1][2])
print (Tup)
```

```
1
2500
((1, 'abc', 2000), (2, 'xyz', 2500))
```

Accessing Values in Tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example

```
# accessing a tuple
```

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5, 6, 7)
print("tup1[0]: ", tup1[0])
print("tup2[1:5]: ", tup2[1:5])
```

```
tup1[0]: physics
tup2[1:5]: (2, 3, 4, 5)
```

Updating Tuples

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates

```
# updating a tuple
```

```
tup1 = (12, 34.56)
tup2 = ('abc', 'xyz')
# following action is not valid for tuples
# tup1[0] = 100
# so let's create a new tuple as follows
tup3 = tup1 + tup2
print(tup3)
```

```
(12, 34.56, 'abc',
```

Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the ***del*** statement.

```
# deleting a tuple

tup = ('ABC', 'pqr', 1997, 2000)
print(tup)
del tup
print("After deleting tup ")
print(tup)
```

This produces the following result. Note an exception raised, this is because after ***del tup*** tuple does not exist any more

```
('ABC', 'pqr', 1997, 2000)
After deleting tup :
-----
NameError                                Traceback (most recent call)
<ipython-input-54-692273a13f26> in
    <module> 3 del tup;
          4 print("After deleting tup : ")
----> 5 print(tup)
NameError: name 'tup' is not defined
```

Tuple Operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition heretoo, except that the result is a new tuple, not a string

<i>Python Expression</i>	<i>Results</i>	<i>Description</i>
<i>T=(1,2,3)</i> <i>print(len(T))</i>	3	Length
<i>T=(1, 2, 3)+(4, 5, 6)</i> <i>print(T)</i>	(1, 2, 3, 4, 5, 6)	Concatenation
<i>T=('Hi!')* 4</i> <i>print(T)</i>	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
<i>T=(1,2,3)</i> <i>print(3 in (T))</i>	True	Membership
<i>for x in (1, 2, 3):</i> <i>print(x)</i>	1 2 3	Iteration

Tuple methods

Python has a set of built-in methods that you can use on tuples.

<i>Method</i>	<i>Description</i>
---------------	--------------------

<i>count()</i>	Returns the number of elements with the specified value
----------------	---

<i>index()</i>	Returns the index of the first element with the specified value
----------------	---

```
# count() and index()
```

```
num = (1, 2, 2, 3, 1, 2, 5)
num1 = num.count(2)
num2 = num.index(2)
print("count of 2:", num1)
print("index of 2:", num2)
```

```
count of 2: 3
```

```
index of 2: 1
```

Indexing, Slicing, Matrices

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input

```
L = ('spam', 'Spam', 'SPAM!')
```

<i>Python Expression</i>	<i>Results</i>	<i>Description</i>
<code>print(L[2])</code>	'SPAM!'	Offsets start at zero
<code>print(L[-2])</code>	'Spam'	Negative: count from the right
<code>print(L[1:])</code>	['Spam', 'SPAM!']	Slicing fetches sections

Dictionary

A dictionary is *mutable* and is another container type that can store any number of Python objects, including other container types. Dictionaries consist of *pairs* (called items) of *keys* and their corresponding *values*.

Python dictionaries are also known as *associative arrays* or *hash tables*. The general syntax of a dictionary is as follows

```
d1 = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}
```

You can create dictionary in the following way as well:

```
d1 = {'abc': 456}
d2 = {'abc': 123, 98.6: 37}
```

Each *key* is separated from its *value* by a *colon* (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are *unique* within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an *immutable* data type such as string s, numbers, or tuples.

Accessing Values in Dictionary

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example

```
# accessing a dictionary
```

```
d1 = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};
print("Name:", d1['Name']);
print("Age:", d1['Age']);
print("Class:", d1['Class']);
```

```
Name: Zara
Age: 7
Class: First
```

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows

```
d1 = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};
print("d1['Alice']: ", d1['Alice'])
```

```
Traceback (most recent call last):
  File "D:\LJ\Python\Code\LJ\Chapter
3\testing.py", line 33, in <module>
    print("d1['Alice']: ", d1['Alice'])
KeyError: 'Alice'
```

Updating Dictionary

You can update a dictionary by adding a new entry or item(i.e., a key-value pair), modifying an existing entry, or deleting an existing entry as shown below in the example

```
# updating dictionary
d1 = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print (d1)
d1['Age'] = 8 # update existing entry
d1['School'] = "LJP" # Add new entry
print ("d1['Age']: ", d1['Age'])
print ("d1['School']: ", d1['School'])
print (d1)
```

```
{'Name': 'Zara', 'Age': 7, 'Class': 'First'}
d1['Age']: 8
d1['School']: LJP
{'Name': 'Zara', 'Age': 8, 'Class': 'First', 'School': 'LJP'}
```

Deleting Dictionary Elements

To explicitly remove an entire dictionary, just use the **del** statement.

```
# deleting dictionary
d1 = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print("Name:", d1['Name'])
del d1['Name'] # remove entry with key 'Name'
print(d1)
d1.clear() # remove all entries in d1
print(d1)
del d1 # delete entire dictionary
print(d1)
```

```
Name: Zara
{'Age': 7, 'Class': 'First'}
{}
Traceback (most recent call last):
  File "D:\LJ\Python\Code\LJ\Chapter 3\testing.py", line 9, in
<module>
    print(d1)
NameError: name 'd1' is not defined
```

Build-in Dictionary Functions & Methods

<i>Method</i>	<i>Description</i>
<code>d1.clear()</code>	Removes all elements of dictionary d1
<code>d1.pop()</code>	Removes mentioned key and its values
<code>d1.copy()</code>	Returns a shallow copy of dictionary d1
<code>d1.get(key, default=None)</code>	Returns the value of the item with the specified key
<code>d1.setdefault(key, default=None)</code>	Similar to get(), but it will set d1[key]=default if key is not indicted
<code>d1.items()</code>	Returns a list of d1's (key, value) tuple pairs
<code>d1.keys()</code>	Returns list of dictionary d1's keys
<code>d1.values()</code>	Returns list of dictionary d1's values
<code>d1.update(d2)</code>	Adds dictionary d2's key-values pairs to d1
<code>d1.fromkeys()</code>	Creates a new dictionary with specified keys and specified values

clear()

```
d1 = {'Name': 'Zara', 'Age': 7}
print("Start Len:", len(d1))
d1.clear()
print("End Len:", len(d1))
```

Start Len: 2
End Len: 0

pop()

```
d1 = {'Name': 'Zara', 'Age': 7}
print("Before: ", d1)
print(a.pop('Name'))
print("After: ", d1)
```

Before: {'Name': 'Zara', 'Age': 7}
Zara
After: {'Age': 7}

copy()

```
d1 = {'Name': 'Zara', 'Age': 7}
d2 = d1.copy()
print("New Dictionary:", d2)
```

New Dictionary: {'Name': 'Zara', 'Age': 7}

```
# get()
```

```
d1 = {'Name': 'Zara', 'Age': 7}
print("Value:", d1.get('Age'))
print("Value:", d1.get('Education', "Never"))
print(d1.get('c'))
print(d1) #keys and values will not be updated
```

```
Value: 7
Value: Never
None
{'Name': 'Zara', 'Age': 7}
```

```
# setdefault()
```

```
d1 = {'Name': 'Zara', 'Age': 7}
print("old:", d1)
print("Value1:", d1.setdefault("Name", "None"))
print("Value2:", d1.setdefault("City"))
print("new:", d1) #keys and values will be updated
```

```
old: {'Name': 'Zara', 'Age': 7}
Value1: Zara
Value2: None
new: {'Name': 'Zara', 'Age': 7, 'City': None}
```

```
# items()
```

```
d1 = {'Name': 'Zara', 'Age': 7}
print("Value:", d1.items())
```

```
Value: dict_items([('Name', 'Zara'), ('Age', 7)])
```

```
# keys()
```

```
d1 = {'Name': 'Zara', 'Age': 7}
print("Keys:", d1.keys())
```

```
Keys: dict_keys(['Name', 'Age'])
```

```
#values()
```

```
d1 = {'Name': 'Zara', 'Age': 7}  
print("Values:", d1.values())
```

```
Values: dict_values(['Zara', 7])
```

```
# update()
```

```
d1 = {'Name': 'Zara', 'Age': 7}  
d2 = {'Gender': 'female'}  
print("Old value:", d1)  
d1.update(d2)  
print("Updated value:", d1)
```

```
Old value: {'Name': 'Zara', 'Age': 7}
```

```
Updated value: {'Name': 'Zara', 'Age': 7, 'Gender': 'female'}
```

```
#fromkeys
```

```
seq = 1, 2, 3  
value = "new"  
d1 = dict.fromkeys(seq)  
d2 = dict.fromkeys(seq, value)  
print("Without 2nd parameter:", d1)  
print("With 2nd parameter:", d2)
```

```
Without 2nd parameter: {1: None, 2: None, 3: None}
```

```
With 2nd parameter: {1: 'new', 2: 'new', 3: 'new'}
```

Set

Sets are used to store multiple items in a single variable. A set is a collection which is **unordered** and **unindexed**. Sets are written with curly brackets.

```
# creating set
```

```
thisset = {"apple", "banana", "cherry"}  
print(thisset)
```

```
{'banana', 'apple', 'cherry'}
```

Note: Sets are unordered, so you cannot be sure in which order the items will appear.

Duplicates Not Allowed

Sets cannot have two items with the same value.

```
# creating set
```

```
thisset = {"apple", "banana", "cherry", "cherry"}  
print(thisset)
```

```
{'banana', 'apple', 'cherry'}
```

Set Items - Data Types

Set items can be of any data type.

```
# string, int and boolean data types
```

```
set1 = {"apple", "banana", "cherry"}  
set2 = {1, 5, 7, 9, 3} #in numeric, it will print in order.  
set3 = {True, False, False}  
set4= {"abc", 34, True, 40, "male"} # set with strings, integers and boolean values  
print(set1)  
print(set2)  
print(set3)  
print(set4)
```

```
{'banana', 'cherry', 'apple'}  
{1, 3, 5, 7, 9}  
{False, True}  
{True, 34, 'male', 40, 'abc'}
```

Access Items

You cannot access items in a set by referring to an *index* or a *key*. But you can loop through the set items using a *for loop*, or ask if a specified value is present in a set, by using the *in* keyword.

loop through the set, and print the values

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
    print(x, end=",")
```

cherry,banana,apple,

check if "apple" is present in the set:

```
thisset= {"apple", "banana", "cherry"}
print("apple" in thisset)
```

True

Build-in Set Functions & Methods

<i>Method</i>	<i>Description</i>
<i>s1.add(item)</i>	To add one item to a set
<i>s1.remove(item)</i>	To remove an item in a set
<i>s1.discard(item)</i>	Removes the specified item from the set
<i>s1.pop()</i>	Removes a random item from the set
<i>s1.clear()</i>	Removes all elements in a set
<i>s1.update(s2)</i>	Updates the current set, by adding items from another set
<i>s1.union(s2)</i>	Returns a new set containing all items from both sets
<i>s1.intersection(s2)</i>	Return a set that contains the items that exist in both s1 and s2
<i>s1.intersection_update(s2)</i>	Remove the items that is not present in both s1 and s2
<i>s1.difference(s2)</i>	Return a set that contains the items that only exist in s1, and not in s2
<i>s1.difference_update(s2)</i>	Remove the items that exists in both sets and store unique one.

*# add an item to a set, using the add() method
(Once a set is created, you cannot change its items, but you can add new items.)*

```
s1 = {"apple", "banana", "cherry"}  
s1.add("orange")  
print(s1)
```

```
{'cherry', 'orange', 'banana', 'apple'}
```

remove "apple" by using the remove() method

```
s1 = {"apple", "banana", "cherry"}  
s1.remove("apple")  
print(s1)
```

```
{'cherry', 'banana'}
```

remove "apple" by using the discard() method

```
s1 = {"apple", "banana", "cherry"}  
s1.discard("apple")  
print(s1)
```

```
{'cherry', 'banana'}
```

*# remove the last item by using the pop() method
(sets are unordered, so you will not know which item that gets removed. The return value of the pop() method is the removed item.)*

```
s1 = {"apple", "banana", "cherry"}  
x = s1.pop()  
print(x)  
print(s1)
```

```
apple  
{'banana', 'cherry'}
```


#clear() method empties the set

```
s1 = {"apple", "banana", "cherry"}  
s1.clear()  
print(s1)
```

```
set()
```

add elements from s2 into s1

(The object in the update() method does not have to be a set, it can be any iterable object (tuples, lists, dictionaries etc.).

```
s1 = {"apple", "banana", "cherry"}  
s2 = ["kiwi", "orange"]  
s1.update(s2)  
print(s1)
```

```
{'cherry', 'banana', 'orange', 'apple', 'kiwi'}
```

#union() method returns a new set with all items from both sets

```
s1 = {"a", "b", "c"}  
s2 = {1, 2, 3}  
s3 = s1.union(s2)  
print(s3)
```

```
{1, 'c', 2, 3, 'a', 'b'}
```

Note: Both *union()* and *update()* will exclude any duplicate items.

#intersection() method returns set that contains the items that exist in both s1 and s2

```
s1 = {"a", "b", "c"}  
s2 = {"a", "d", "e"}  
s3 = s1.intersection(s2)  
print(s3)
```

```
{'a'}
```

#intersection_update() remove the items that is not present in both s1 and s2

```
s1 = {"p", "q", "r"}
s2 = {"p", "w", "r"}
print("old s1:", s1)
s1.intersection_update(s2)
print("new s1:", s1)
print("s2:", s2)
```

```
old s1: {'q', 'r', 'p'}
new s1: {'p', 'r'}
s2: {'r', 'p', 'w'}
```

#difference() method return a set that contains the items that only exist in s1, and not in s2

```
s1 = {"a", "b", "e"}
s2 = {"a", "d", "e"}
s3 = s1.difference(s2)
print(s3)
```

```
{'b'}
```

#difference_update() method remove the items that exists in both sets and store unique one.

```
s1 = {"p", "q", "r"}
s2 = {"p", "w", "r"}
print("old s1:", s1)
s1.difference_update(s2)
print("new s1:", s1)
print("s2:", s2)
```

```
old s1: {'p', 'r', 'q'}
new s1: {'q'}
s2: {'w', 'p', 'r'}
```

del keyword will delete the set completely.

#del keyword will delete the set completely

```
thisset = {"apple", "banana", "cherry"}  
del thisset  
print(thisset)
```

```
NameError          Traceback (most recent call last)  
<ipython-input-24-2ec9feb8cb8a> in <module> 1 thisset =  
    {"apple", "banana", "cherry"}  
    2 del thisset  
----> 3 print(thisset)  
NameError: name 'thisset' is not defined
```

CHAPTER 4 - Classes & Modules in Python

Python Classes/Objects

Python is an object-oriented programming language. Almost everything in Python is an object, with its properties and methods. A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword **class**.

```
# create a class named MyClass, with a property named x  
  
class MyClass:  
    x = 5
```

Create Object

Now we can use the class named MyClass to create objects

```
# create an object named p1, and print the value of x  
  
p1 = MyClass()  
print(p1.x)
```

```
# example of class and object  
  
class MyClass:  
    x = 5  
  
p1 = MyClass()  
print(p1.x)
```

5

The `__init__()` Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in `__init__()` function.

All classes have a function called `__init__()`, which is always executed when the class is being initiated. The `__init__()` function is used to assign values to object properties and other operations that are necessary to do when the object is being created.

```
# create a class named Person
# use the __init__() function to assign values for name and age
```

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

```
p1 = Person("John", 36)
```

```
print(p1.name)
print(p1.age)
```

```
John
36
```

The self Parameter

The **self** parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named self, you can call it whatever you like, but it has to be the first parameter of any function in the class.

```
#use the words myobject and arg instead of self
```

```
class Person:

    def __init__(myobject, name, age):
        myobject.name = name
        myobject.age = age

    def myfunc(arg):
        print("Hello my name is " + arg.name)

p1 = Person("John", 36)
p1.myfunc()
```

```
Hello my name is John
```

Modify Object Properties

You can modify properties on objects like this

```
p1.age = 40
```

```
# Set the age of p1 to 40
```

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

    def myfunc(self):
        print("Hello my name is " + self.name)

p1 = Person("John", 36)

print(p1.age)

p1.age = 40

print(p1.age)

p1.myfunc()
```

```
36
40
Hello my name is John
```

Delete Object Properties or Object

You can delete properties of objects or object by using the del keyword

```
del p1.age    #delete the age property from the p1 object

del p1        # delete the p1 object
```

```
# delete the p1 object

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):
        print("Hello my name is " + self.name)

p1 = Person("John", 36)
print(p1.age)

del p1.age
print(p1.age)

del p1
print (p1)
```

36

```
AttributeError                                Traceback (most recent call last)
<ipython-input-16-46454e0e6825> in <module>
    12 del p1.age
---> 13 print(p1.age)
AttributeError: 'Person' object has no attribute 'age'
```

```
NameError                                    Traceback (most recent call last)
    15 del p1
---> 16 print(p1)
NameError: name 'p1' is not defined
```

Inheritance

Inheritance is a way of creating a new class for using details of an existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

Inheritance is used as given below in Python:

```
# parent class
class Bird:

    def __init__(self):
        print("Bird is ready")

    def who(self):
        print("Bird")

    def swim(self):
        print("Swim faster")

# child class
class Penguin(Bird):

    def __init__(self):
        print("Penguin is ready")
        # call super() function
        super().__init__()

    def who(self):
        print("Penguin")

    def run(self):
        print("Run faster")

peggy = Penguin()
peggy.who()
peggy.swim()
peggy.run()
```

```
Penguin is ready
Bird is ready
Penguin
Swim faster
Run faster
```


In the above program, we created two classes i.e. *Bird* (parent class) and *Penguin* (child class). The child class inherits the functions of parent class. We can see this from the `swim()` method.

Again, the child class modified the behavior of the parent class. We can see this from the `who()` method. Furthermore, we extend the functions of the parent class, by creating a new `run()` method.

Additionally, we use the `super()` function inside the `__init__()` method. This allows us to run the `__init__()` method of the parent class inside the child class.

Polymorphism

Polymorphism is an ability (in OOP) to use a common interface for multiple forms (data types).

Suppose, we need to color a shape, there are multiple shape options (rectangle, square, circle). However, we could use the same method to color any shape. This concept is called Polymorphism.

Following is the example of Polymorphism in Python:

```
class Parrot:

    def fly(self):
        print("Parrot can fly")

    def swim(self):
        print("Parrot can't swim")

class Penguin:

    def fly(self):
        print("Penguin can't fly")

    def swim(self):
        print("Penguin can swim")

# common interface
def flying_test(bird):
    bird.fly()

#instantiate objects
blu = Parrot()
peggy = Penguin()

# passing the object
flying_test(blu)
flying_test(peggy)
```

Following is the output:

```
Parrot can fly
Penguin can't fly
```

In the above program, we defined two classes *Parrot* and *Penguin*. Each of them have a common `fly()` method. However, their functions are different.

To use polymorphism, we created a common interface i.e `flying_test()` function that takes any object and calls the object's `fly()` method. Thus, when we passed the *blu* and *peggy* objects in the `flying_test()` function, it ran effectively.

Modules

In Python, a module is a self-contained Python file that contains Python statements and definitions, like a file named `GFG.py`, can be considered as a module named `GFG` which can be imported with the help of `import` statement. However, one might get confused about the difference between modules and packages. A package is a collection of modules in directories that give structure and hierarchy to the modules.

Creating Modules (User-defined module)

A module is simply a Python file with a `.py` extension that can be imported inside another Python program. The name of the Python file becomes the module name. The module contains definitions and implementation of classes, variables, and functions that can be used inside another program.

```
# Create a new (module) file
# Factorial_module.py

def fact(n):
    if n==0 or n==1:
        return (1)
    else:
        return (n * fact(n-1))

# In another file, import the module.
#test_module.py

import Factorial_module
n = int(input("Enter the value: "))
print(Factorial_module.fact(n))
```

Now enter the Python interpreter and import this module with the following command:

```
>>> Enter the value: 4
>>>24
```

pip and PyPI

Python pip (**Preferred Installer Program**) is the package manager for Python packages. We can use pip to install packages that do not come with Python. The basic syntax of pip commands in command prompt is:

pip 'arguments'

How to install pip?

Python pip comes pre-installed on 3.4 or older versions of Python. To check whether pip is installed or not type the below command in the terminal.

```
pip --version
```

This command will tell the version of the pip if pip is already installed in the system.

We can install additional packages by using the Python pip install command. Let's suppose we want to install the numpy using pip. We can do it using the below command.

```
pip install numpy
```

The Python pip list command displays a list of packages installed in the system.

```
pip list
```

```
C:\> Command Prompt
C:\> pip list
Package                                Version
-----
abs1-py                                0.9.0
alabaster                               0.7.12
anaconda-client                         1.7.2
anaconda-navigator                      1.9.7
anaconda-project                       0.8.3
asn1crypto                             1.0.1
astor                                   0.7.1
astroid                                 2.3.1
astropy                                 3.2.1
atomicwrites                           1.3.0
attrs                                  19.2.0
Babel                                   2.7.0
backcall                                0.1.0
backports.functools-lru-cache          1.5
```

The Python pip uninstall command uninstalls a particular existing package.

```
pip uninstall numpy
```

The pip uninstall command does not uninstall the package dependencies. If you want to remove the dependencies as well then you can see the dependencies using the pip show command and remove each package manually.

PyPI

The **Python Package Index**, abbreviated as PyPI, is the official repository of software for the Python programming language. By default, pip — which is the most popular Python package manager — uses PyPI as the source for retrieving package dependencies.

PyPI lets you find, install and even publish your Python packages so that they are widely available to the public. More than 300,000 different packages are currently published in the index with more than 2,500,000 releases being distributed to users.

CHAPTER 5 - Python Standard Libraries

Working with path, files and directories

Python supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. Python treats file differently as text or binary and this is important. Each line of code includes a sequence of characters and they form text file. Each line of a file is terminated with a special character, called the EOL or End of Line characters like comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun.

Working with paths

Pathlib module in Python provides various classes representing file system paths with semantics appropriate for different operating systems. This module comes under Python's standard utility modules.

Path class is a basic building block to work with files and directories.

We can import the Path class from pathlib module as below:

```
from pathlib import Path
```

We can create a path class object like:

```
Path("C:\\Program Files\\Microsoft")
```

We can also create a Path object that represents a current directory like:

```
Path()
```

We can also get the home directory of current user using *home* method.

```
Path.home()
```

Let us create a path object to get the use of some methods.

```
path = Path("D://testmodule/test.txt")
```

We can check if the path exists or not by *exists* method.

```
path.exists()
```

We can also check if the path represents a file or not.

```
path.is_file()
```

Same thing we can check for directory.

```
path.is_dir()
```

We can get the file name in the path by *name* property.

```
print(path.name)
```

We can get the file name without extension by *stem* property.

```
print(path.stem)
```

Same way, we can get the extension only by *suffix* property.

```
print(path.suffix)
```

will display .py

We can get the parent directory by using *parent* property.

```
print(path.parent)
```

We can also display the absolute path by using *absolute* method.

```
print(path.absolute())
```

Working with Directories

As we have already covered, Path object can be created using:

```
path = Path("test")
```

We can create directories as follows.

```
import os
```

```
os.mkdir('name')
```

We can rename it by:

```
os.rename("name", "rename")
```

Also, we can remove directories as follows.

```
os.rmdir('name')
```

Now, to iterate through all the files and directories we can use the following code.

```
# Using os.listdir()
```

```
print("Using os.listdir():")
```

```
for item in os.listdir():
```

```
    print(item)
```

```
# Using pathlib.Path.iterdir()
```

```
current_dir = Path('.')
```

```
for item in current_dir.iterdir():
```

```
    print(item)
```

Above code shows all the directories and files on the mentioned path. So, if we want to see just the directories, we can use:

```
for p in path.iterdir():
```

```
    if p.is_dir():
```

```
        print(p)
```

This method has two limitations. It cannot search by patterns and it cannot search recursively. So, to overcome this limitation, we can use glob method.

Working with Files

To see how we can work with files, let us refer to the file we want to work with using Path class object.

```
path = Path("D://testmodule/test.txt")
```

We can check if the file exists or not using:

```
path.exists()
```

We can rename the file using rename method.

```
path.rename("init.txt")
```

We can check the details of the file using stat method.

```
print(path.stat())
```

We can write into the file using write_text method.

```
path.write_text("Hello All")
```

Reading from a file can be done using read_text method.

```
print(path.read_text())
```

Also, we can delete the file using unlink method.

```
path.unlink()
```

Working with CSV

CSV stands for Comma Separated Values which is a file that stores the values separated with comma. They serve as simple means to store and transfer data.

We can use csv module to work with csv files.

```
import csv
```

We can open a csv file and write into it by using built in open function.

```
file = open("data.csv", "w")
```

Now, this csv module has writer method to write content into csv file.

```
content = csv.writer(file)
```

As mentioned, we need to pass file object to this method.

Now, we can use writer to write tabular data into csv file. For this, we need to use writerow method to which we need to pass values in form of array.

```
content.writerow(["transaction_id", "product_id", "price"])
```

```
content.writerow([1000, 1, 5])
```

```
content.writerow([1001, 2, 15])
```

```
file.close()
```

Here, we have created three rows with three columns in a csv file.

Now, we can read the csv file using reader method. First, we need to open the file in read mode. For that we do not require to pass the mode in second parameter. After opening the file, we can use reader method to read the file. Once file has been read, we convert it into list using list method. And then we can iterate through that list to read the content of csv file.

```
file = open("data.csv")
```

```
read1 = csv.reader(file)
```



```
read2 = list(read)
```

```
for row in read2:
```

```
    print(row)
```

Working with time and datetime

There are two modules which we can use to work with date and time. time and datetime. time refers to the current timestamp, which represents the number of seconds passed after the time started, which is usually 1st January, 1970.

Let us see first how we can use time module.

```
import time
```

```
time1 = time.time()
```

```
curr = time.ctime(time1)
```

```
print("Current time: ",curr)
```

Output:

Current time: Wed Nov 23 14:40:52 2022

This statement will print the number of seconds passed after the date and time mentioned above.

This method can be used to perform time calculations such as duration to perform some task.

To work with date and time, we can use datetime class of datetime module.

```
from datetime import datetime
```

Now, we can create a datetime object by mentioning year, month and day. Optionally we can mention hour, minutes and seconds too.

```
dt = datetime(2022, 12, 25)
```

```
print("Create date: ", dt)
```

Output:

Create date: 2022-12-25 00:00:00

We can also create a datetime object which represents the current date and time.

```
dt = datetime.now()

print("Today & now: ", dt)
```

Output:

```
Today & now: 2022-11-23 09:32:02.429447
```

While taking input from user or reading from a file we deal with strings. So, when we read date or time from such sources, we need to convert this input to datetime object. We can use `strptime` method for this purpose.

Let us assume that we received a date 2022/12/25 as a string input. To convert it to datetime object, we need to specify which part of full date represents what. So, we can do that by writing:

```
dt = datetime.strptime('2022/12/25', "%Y/%m/%d")

print("str to date: ", dt)
```

Output:

```
str to date: 2022-12-25 00:00:00
```

Where, %Y, %m and %d are directives to represent four-digit year, two-digit month and two-digit date respectively.

We can use `strftime` method to do exact opposite, i.e. convert datetime object to string.

```
dt = datetime.datetime(2022,12,25)

print("date to str: ", dt.strftime("%Y/%m/%d"))
```

Output:

```
date to str: 2022/12/25
```

Similarly, we can convert timestamp into datetime object using `fromtimestamp()` method.

```
import time

dt = datetime.fromtimestamp(time.time())
```

datetime object has properties like year and month which we can use to print year and month.

```
print("timestamp to year: ", dt.year)
```

```
print("timestamp to month: ", dt.month)
```

```
print("timestamp to day: ", dt.day)
```

Output:

```
timestamp to year: 2022
```

```
timestamp to month: 11
```

```
timestamp to day:20
```

We can also compare two datetime objects to know which date is greater.

```
dt1 = datetime.datetime(2022,1,1)
```

```
dt2 = datetime.datetime(2022,12,25)
```

```
print(dt2 > dt1)
```

Above code will print TRUE as date in dt2 is greater.