Module-I: Introduction to Python

Introduction: History of Python, Features of Python Programming, Applications of Python Programming, Running Python Scripts, Comments, Typed Language, Identifiers, Variables, Keywords, Input/output, Indentation, Data types, Type Checking, range (), format (), Math Module.

Introduction to Python:

Python is a general purpose high level programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- Web development (server-side)
- Software development
- Mathematics
- System scripting.

Why Python:

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

The most recent major version of Python is **Python 3**, which we shall be using in this Course. However, Python 2, although not being updated with anything other than security updates, is still quite popular.

It is possible Python will be written in a text editor and we have save file as **filename.py**. It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

8 World-Class Software Companies That Use Python

- Industrial Light and Magic.
- Google.
- Facebook.
- Instagram.
- Spotify.
- Quora.
- Netflix.
- Dropbox.

1.1 History of python

- The implementation of Python was started in December 1989 by **Guido Van Rossum** at CWI in Netherland.
- In February 1991, **Guido Van Rossum** published the code (labeled version 0.9.0) to alt.sources.
- In 1994, Python 1.0 was released with new features like lambda, map, filter, and reduce.
- Python 2.0 added new features such as list comprehensions, garbage collection systems.
- 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 and interfacing with the Amoeba Operating System.
- The following programming languages influence Python:
    - ABC language.
    - Modula-3

Python Version List

Python programming language is being updated regularly with new features and supports. There are lots of update in Python versions, started from 1994 to current release.

A list of Python versions with its released date is given below.

Python Version	Released Date
Python 1.0	January 1994
Python 1.5	December 31, 1997
Python 1.6	September 5, 2000
Python 2.0	October 16, 2000
Python 2.1	April 17, 2001
Python 2.2	December 21, 2001
Python 2.3	July 29, 2003
Python 2.4	November 30, 2004
Python 2.5	September 19, 2006
Python 2.6	October 1, 2008
Python 2.7	July 3, 2010
Python 3.0	December 3, 2008
Python 3.1	June 27, 2009
Python 3.2	February 20, 2011
Python 3.3	September 29, 2012
Python 3.4	March 16, 2014
Python 3.5	September 13, 2015
Python 3.6	December 23, 2016
Python 3.7	June 27, 2018
Python 3.8	October 14, 2019

Why the Name Python?

- ✓ There is a fact behind choosing the name Python. Guido van Rossum was reading the script of a popular BBC comedy series "Monty Python's Flying Circus". It was late on-air 1970s.
- ✓ Van Rossum wanted to select a name which unique, sort, and little-bit mysterious. So he decided to select naming Python after the "Monty Python's Flying Circus" for their newly created programming language.
- ✓ The comedy series was creative and well random. It talks about everything. Thus it is slow and unpredictable, which made it very interesting.

1.2 Features of Python Programming

Python is a dynamic, high level, free open source and interpreted programming language. It supports object-oriented programming as well as procedural oriented programming. There are many features in Python, some of which are discussed below

1. **Easy to code:** Python is a high-level programming language. Python is very easy to learn the language as compared to other languages like C, C#, Javascript, Java, etc. It is very easy to code in python language and anybody can learn python basics in a few hours or days. It is also a developer-friendly language.
2. **Free and Open Source:** Python language is freely available at the official website and you can download it from the official website. Since it is open-source, this means that source code is also available to the public. So you can download it as, use it as well as share it.
3. **Object-Oriented Language:** One of the key features of python is Object-Oriented programming. Python supports object-oriented language and concepts of classes, objects encapsulation, etc.
4. **GUI Programming Support:** Graphical User interfaces can be made using a module such as PyQt5, PyQt4, wxPython, or Tk in python. PyQt5 is the most popular option for creating graphical apps with Python.

5. **High-Level Language:** Python is a high-level language. When we write programs in python, we do not need to remember the system architecture, nor do we need to manage the memory.
6. **Extensible feature:** Python is an Extensible language. We can write us some Python code into C or C++ language and also we can compile that code in C/C++ language.
7. **Python is Portable language:** Python language is also a portable language. For example, if we have python code for windows and if we want to run this code on other platforms such as Linux, UNIX, and Mac then we do not need to change it, we can run this code on any platform.
8. **Python is integrated language:** Python is also an integrated language because we can easily integrated python with other languages like c, c++, etc.
9. **Interpreted Language:** Python is an Interpreted Language because Python code is executed line by line at a time, like other languages C, C++, Java, etc. there is no need to compile python code this makes it easier to debug our code. The source code of python is converted into an immediate form called byte code.
10. **Large Standard Library:** Python has a large standard library which provides a rich set of module and functions so you do not have to write your own code for every single thing. There are many libraries present in python for such as regular expressions, unit-testing, web browsers, etc.
11. **Dynamically Typed Language:** Python is a dynamically-typed language. That means the type (for example- int, double, long, etc.) for a variable is decided at run time not in advance because of this feature we don't need to specify the type of variable.

1.3 Applications of Python Programming

The following are the best applications of python programming

1. **Web Development:** Python can be used to make web-applications at a rapid rate. It is because of the frameworks Python uses to create these applications. Web framework like Django, Pyramid and Flask are based on Python. They help you write server side code which helps you manage database, write backend programming logic, mapping urls etc.
2. **Game Development:** Python is also used in the development of interactive games. There are libraries such as PySoy which is a 3D game engine supporting Python

3, PyGame which provides functionality and a library for game development. Games such as Civilization-IV, Disney's Toontown Online, Vega Strike etc. have been built using Python.

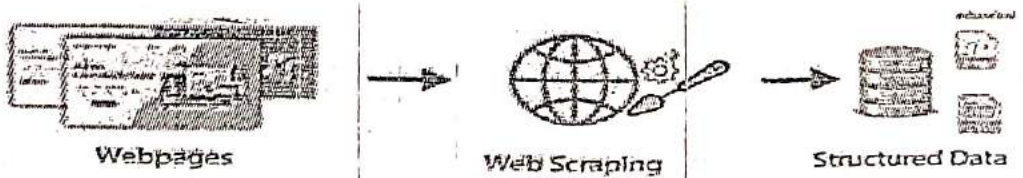
3. **Machine Learning and Artificial Intelligence:** Python along with its inbuilt libraries and tools facilitate the development of AI and ML algorithms. Further, it offers simple, concise, and readable code which makes it easier for developers to write complex algorithms and provide a versatile flow. Some of the inbuilt libraries and tools that enhance AI and ML processes are:

   Numpy for complex data analysis
   Keras for Machine learning
   SciPy for technical computing
   Seaborn for data visualization

4. **Data Science and Data Visualization:** Data science involves data collection, data sorting, data analysis, and data visualization Python programming language is perfect for data visualization because it provides many ready-to-use functions and modules for data visualization. It is also an excellent programming language for Python Data Science Stack. Libraries such as Pandas, NumPy help you in extracting information. You can even visualize the data libraries such as Matplotlib, Seaborn, which are helpful in plotting graphs and much more

5. **Desktop GUI:** Python is an interactive programming language that helps developers to create GUIs easily and efficiently. It provides the Tkinter library that can be used to develop user interfaces. It has a huge list of inbuilt tools like PyQT, kivy, wxWidgets, and many other libraries like them to build a fully functional GUI in an extremely secure and efficient manner.

6. **Web Scraping Application:** Web scraping is an automated process used to extract information from websites in an easier and faster way. The information is used by researchers, organizations, and analysts for a wide variety of tasks



7. **Business Applications:** Python offers excellent security and scalability features that make it perfect for delivering high-performance business applications. It has inbuilt libraries and tools like:
   - ✓ Odoo is an all-in-one management software that offers a range of business applications that form a complete suite of enterprise management applications.
   - ✓ Tryton is a three-tier high-level general purpose application platform.
8. **Software Development:** Python is just the perfect option for software development. Popular applications like Google, Netflix, and Reddit all use Python. This language offers amazing features like: Platform independence Inbuilt libraries and frameworks to provide ease of development. Enhanced code reusability and readability High compatibility
9. **Audio and Visual Applications:** Audio and video applications are undoubtedly the most amazing feature of Python. Video and audio applications such as TimPlayer, Cplay have been developed using Python libraries. Applications that are coded in Python include popular ones like Netflix, Spotify, and YouTube. This can be handled by libraries like
   - ➢ Dejavu
   - ➢ Pyo Mingus
   - ➢ SciPy
   - ➢ OpenCV
10. **CAD Applications:** CAD refers to computer-aided design; it is the process of creating 3D and 2D models digitally. Python is embedded with amazing applications like Blender, FreeCAD, open cascade, and a lot more to efficiently design products. These provide enhanced features like technical drawing, dynamic system development, recordings, file export, and import.
11. **Embedded Applications:** Python is based on C which means that it can be used to create Embedded C software for embedded applications. This helps us to perform higher-level applications on smaller devices which can compute Python.

1.4 Running Python Scripts

The Python script is basically a file containing code written in Python. The file containing Python script has the extension '.py' or can also have the extension '.pyw' if it is being run on

a Windows machine.To run a Python script, we need a Python interpreter that needs to be downloaded and installed.

Different ways to run Python Script

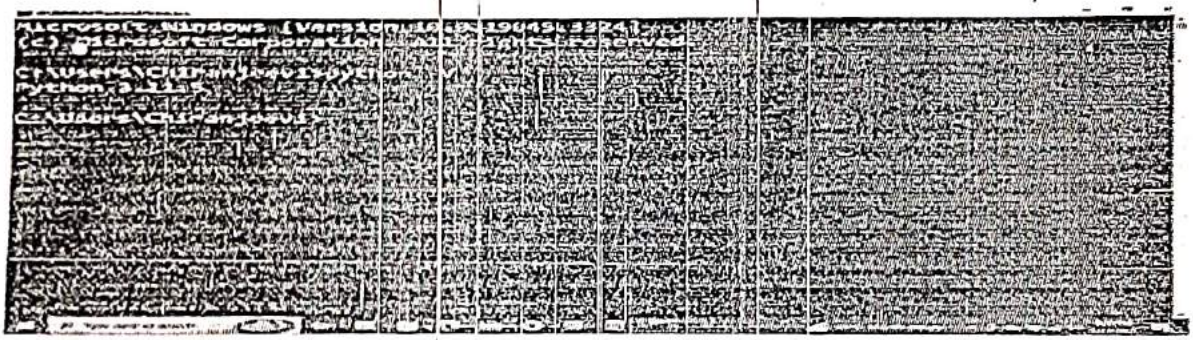
There are different ways using which we can run a Python script.

1. Interactive Mode
2. Command Line
3. Text Editor (VS Code)
4. IDE (PyCharm)

There are various ways to run a Python script but before going toward the different ways to run a Python script, we first have to check whether a Python interpreter is installed on the system or not. So in Windows, open 'cmd' (Command Prompt) and type the following command.

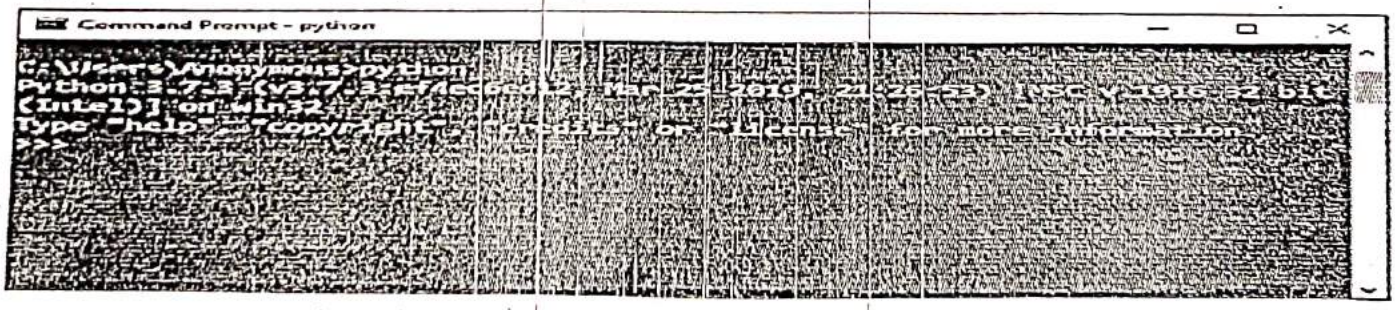
Python -V

This command will give the version number of the Python interpreter installed or will display an error if otherwise.



1. How to Run Python Script Interactively

In Python Interactive Mode, you can run your script line by line in a sequence. To enter an interactive mode, you will have to open Command Prompt on your Windows machine and type 'python' and press Enter.



Example1: Run the following line in the interactive mode:

Print ('Welcome to AIML')

Note: To exit from this mode, press 'Ctrl+Z' and then press 'Enter' or type 'exit()' and then press Enter.

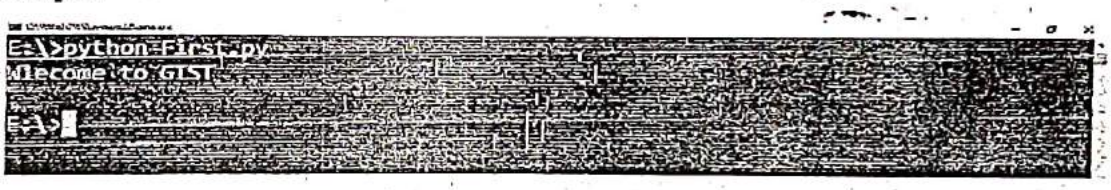
2. How to Run Python Script by the Command Line

To run a Python script store in a '.py' file in the command line, we have to write 'python' keyword before the file name in the command prompt.

python First.py

You can write your own file name in place of '**First.py**'.

Output:

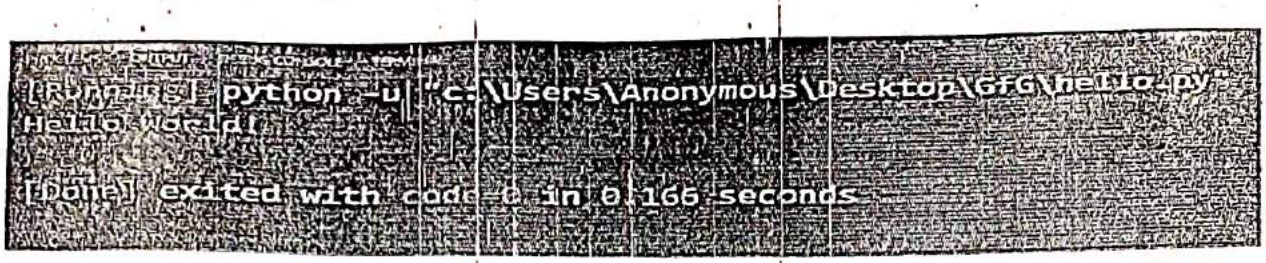


3. Running Python Scripts using a Text Editor

To run Python script on a text editor like VS Code (Visual Studio Code) then you will have to do the following:

- Go to the extension section or press **'Ctrl+Shift+X'** on Windows, then search and install the extension named 'Python' and 'Code Runner'. Restart your vs code after that.
- Now, create a new file with the name '**hello.py**' and write the below code in it:
- print('Hello World!')
- Then, right-click anywhere in the text area and select the option that says 'Run Code' or press **'Ctrl+Alt+N'** to run the code.

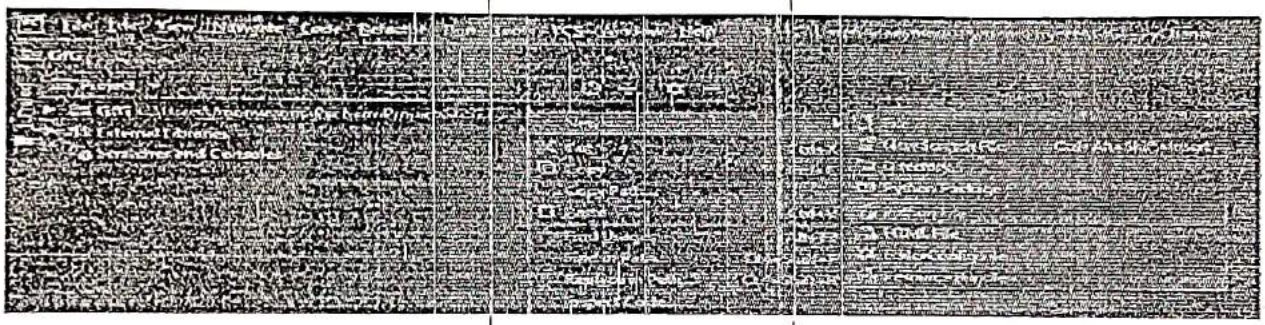
Output:



4. Running Python Scripts using an IDE

To run Python script on an IDE (Integrated Development Environment) like PyCharm, you will have to do the following:

- Create a new project.
- Give a name to that project as 'First' and click on Create.
- Select the root directory with the project name we specified in the last step. Right-click on it, go to **New**, anto, and click on the **'Python file'** option. Then give the name of the file as **'hello'** (you can specify any name as per your project requirement). This will create a **'hello.py'** file in the project root directory.



Now write the below Python script to print the message:

```
print('Hello World !')
```

To run this Python script, Right click and select the 'Run File in Python Console' option. This will open a console box at the bottom and show the output there. We can also run using the Green Play Button at the top right corner of the IDE.

```
print('Hello World')
```

Output:

```
PyDev console: starting.
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v
>>> runfile('C:/Users/Anonymous/PycharmProjects/GfG/hello.py',
Hello World
>>>
```

1.5 Comments in Python

Comments in Python are the lines in the code that are ignored by the interpreter during the execution of the program. Comments enhance the readability of the code and help the programmers to understand the code very carefully

There are three types of comments in Python:

1. Single line Comments
2. Multiline Comments
3. Docstring Comments

1. **Single-line comments** begins with a hash(#) symbol and is useful in mentioning that the whole line should be considered as a comment until the end of line.

```
>>> name = "Welcome to Gist" #String a line comment
>>> print(name)
Welcome to Gist
>>>
```

2. **A Multi line comment is** useful when we need to comment on many lines. In python, triple double quote(" " ") and single quote(' ' ')are used for multi-line commenting.

   Example: Write the following line of code in First.py file

   print("Wlecome to GIST")
   """Multi
   Line comment"""
   '''this is
   multi line comment'''

   Output:

   E:\>python First.py
   Wlecome to GIST

3. **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 with Python modules, functions, classes, and-methods. It is added right below the functions, modules, or classes to describe what they do. In Python, the docstring is then made available via the __doc__ attribute.

   Example:

```
def multiply(a, b):
  """Multiplies the value of a and b"""
  return a*b
# Print the docstring of multiply function
print(multiply.__doc__)
```

   Output:

```
Multiplies the value of a and b
```

1.6 Typed Language

Typed Language: Typed languages are the languages in which we define the type of data type and it will be known by machine at the compile-time or at runtime.

Typed languages can be classified into two categories:

1. Statically typed languages
2. Dynamically typed languages

1. **Statically typed languages:** Statically typed languages are the languages like **C, C++, Java,** etc, In this type of language the data type of a variable is known at the compile time which means the programmer has to specify the data type of a variable at the time of its declaration
2. **Dynamically typed language:** These are the languages that do not require any pre-defined data type for any variable as it is interpreted at runtime by the machine itself. In these languages, interpreters assign the data type to a variable at runtime depending on its value. We don't even need to specify the type of variable that a function is returning or accepting in these languages. **JavaScript, Python, Ruby, Perl,** etc are examples of dynamically typed languages.

1.7 Identifiers in python

Identifier is a user-defined name given to a variable, function, class, module, etc. The identifier is a combination of character digits and an underscore. They are case-sensitive i.e., 'num' and 'Num' and 'NUM' are three different identifiers in python. It is a good programming practice to give meaningful names to identifiers to make the code understandable

Rules for Naming Python Identifiers

- It cannot be a reserved python keyword.
- It should not contain white space.
- It can be a combination of A-Z, a-z, 0-9, or underscore.
- It should start with an alphabet character or an underscore (_).
- It should not contain any special character other than an underscore (_).

Valid identifiers:

Variable _variable _K_variable variable_

Invalid Identifiers

!variable 1variable 1_variable Identi#1

1.8 Variables in Python

Variables are nothing but reserved memory locations to store values. It means that when you create a variable, you reserve some space in the memory. Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory

Assigning Values to Variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

- The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

For example :

```
counter =100 # An integer assignment

miles =1000.0 # A floating point

name ="John" # A string

print(counter)

print(miles)

print(name)
```

Here, 100, 1000.0 and "John" are the values assigned to counter, miles, and name variables, respectively. This produces the following result –

Output:

```
100

1000.0

'John'
```

Multiple Assignments: Python allows you to assign a single value to several variables simultaneously.

For example: a = b = c = 1

Here, an integer object is created with the value 1, and all the three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables.

For example: a, b, c = 1, 2, "john"

Here, two integer objects with values 1 and 2 are assigned to the variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

1.9 Keywords in python

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

1.10 Input and output: In Python, we use the input() function to take input from the user. Whatever you enter as input, the input function converts it into a string. If you enter an integer value still input() function converts it into a string.

Taking Input from the user:

Sometimes a developer might want to take input from the user at some point in the program. To do this Python provides an **input() function.**

Syntax:

```
input('prompt')
```

Where, prompt is a string that is displayed on the string at the time of taking input.

Example1:

```
# Taking input from the user
>>>name = input("Enter your name: ")
>>>print("Hello, " + name)
```

output:

```
Enter your name: Chiranjeevi
Hello, Chiranjeevi
```

Note: By default input() function takes the user's input in a string.

Example 2: By default input() function takes the user's input in a string. So, to take the input in the form of int, you need to use int() along with input function.

```
# Taking input from the user as integer
num = int(input("Enter a number: "))
add = num + 1
# Output
print(add)
```

Output:

```
Enter a number: 25
26
```

Displaying Output:

Python provides the print() function to display output to the console.

Syntax: print(value(s), sep= ' ', end = '\n', file=file, flush=flush)

Parameters:

value(s) : Any value, and as many as you like. Will be converted to string before printed

sep='separator' : (Optional) Specify how to separate the objects, if there is more than one.**Default :' '**

end='end': (Optional) Specify what to print at the end.**Default : '\n'**

file : (Optional) An object with a write method. Default :sys.stdout

flush : (Optional) A Boolean, specifying if the output is flushed (True) or buffered (False). Default: False

Returns: It returns output to the screen.

```
# Python program to demonstrate print() method
#using Single quotes
print('Python is Easy')

# using Double Quotes
print("Chiranjeevi's class")

# using triple Quotes
print('''"Python" is Easy''')

# code for disabling the softspace feature
print('Python', 'is', 'Easy', sep ='@')

# using end argument
print("Python is Easy", end = '+')
print("Importtant")

#printing integers
a=9
print('The value of a is :', a)
```

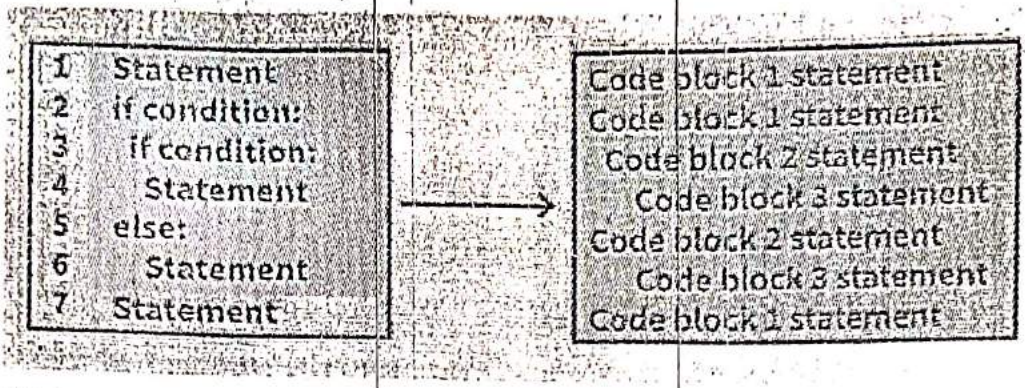
Output:

```
Python is Easy
Chiranjeevi's class
"Python" is easy
Python@is@easy
Python is easy+important
The Value of a is : 9
```

1.11 Indentation

- Indentation in Python is simply the spaces at the beginning of a code line. Indentation in other languages like C, C++, etc., is just for readability, but in Python, the indentation is an essential and mandatory concept that should be followed when writing a python code; otherwise, the python interpreter throws Indentation Error.
- Indentation is the leading spaces or/and tabs before any statement in Python. Python treats the statements with the same indentation level (statements with an equal number of whitespaces before them) as a single code block. So whereas in languages like C, C++,

etc. a block of code is represented by curly braces { }, in python a block is a group of statements that have the same Indentation level i.e same number of leading whitespaces.



- All the statements which are on the same level of Indentation(Same no of whitespaces before them) belong to a single block, so from the above diagram, statements in line 1, line 2, and line 7 belong to a single block, and the block has the zero or lowest level of indentation. Statements 3 and 5 are indented one step, forming another block at the first level of indentation. Similarly, statements 4 and 6 are indented two steps, so they together form another block at the second level of indentation.
- Below the line 2 statement, which is an if statement, statements 3 and 5 are indented one step; hence, they belong to a single block. And since line 2 is an if statement, the block indented below the first if forms the body of second if. So here, the body of the if statement at line 2 includes all the lines that are indented below it, i.e., lines 3, 4,5 and 6.
- Now that we know that statement at line numbers 3,4,5 and 6 forms the body of the if statement at line 2. Let us understand the indentation for them. Statements at 3 and 5 are uniformly indented, so they belong to a single block (block2 from the interpretation), and they will be executed one by one.
- Statement at line 4 makes up the body of the if statement at line 3, as we know any statements that are indented below an if form the body of if statement, similarity the statement at line 6 makes up the body of else statement at line 5.

1.12 Data types:

The Data type in python defines what type of data we are going to store in a variable Python has the following data types built-in by default, in these categories:

Text Type : str

Numeric Types	:	int, float, complex
Sequence Types	:	list, tuple, range
Mapping Type	:	dict
Set Types	:	set, frozenset
Boolean Type	:	bool

Text Type:

String: A string value is a collection of one or more characters put in single, double or triple quotes.

```
>>>x = "Hello World"  # DataType Output: string
>>>type(x)
<class 'str'
```

Numeric Types: The numeric data type in Python represents the data that has a numeric value. A numeric value can be an integer, a floating number, or even a complex number. These values are defined as Python int, Python float, and Python complex classes in Python. Numeric types are: int, float and complex

```
a = 5
print("Type of a: ", type(a))
b = 5.0
print("\nType of b: ", type(b))
c = 2 + 4j
print("\nType of c: ", type(c))
```

```
Output:
Type of a: <class 'int'>
Type of b: <class 'float'>
Type of c: <class 'complex'>
```

Sequence Type:

A sequence is an ordered collection of similar or different data types. Python has the following built-in sequence data types:

List: A list object is an ordered collection of one or more data items, not necessarily of the same type, put in square brackets.

```
>>> b=['apple',1,2,5.0]
>>> b
['apple', 1, 2, 5.0]
>>> type(b)
<class 'list'>
```

Tuple: A Tuple object is an ordered collection of one or more data items, not necessarily of the same type, put in parentheses.

```
>>> c=(1,2,'apple')
>>> c
(1, 2, 'apple')
>>> type(c)
<class 'tuple'>
```

Range: 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.

Syntax: **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

Examples:.

```
>>> x=range(5)
>>> x
range(0, 5)
>>> type(x)
<class 'range'>
>>> x = range(3, 6)
>>> x
range(3, 6)
>>> x = range(3, 20, 2)
>>> x
range(3, 20, 2)
```

Mapping Type:

Python Dictionary: Dictionary is an unordered collection of key-value pairs.It is generally used when we have a huge amount of data. Dictionaries are optimized for retrieving data. We must know the key to retrieve the value.In Python, dictionaries are defined within braces {} with each item being a pair in the form key:value. Key and value can be of any type.

```
>>> dict={1:'one',2:'two',3:'three'}
>>> dict[1]
'one'
>>> dict[2]
'two'
>>> dict[3]
'three'
```

Set Types: set, frozenset

Set: Sets are used to store multiple items in a single variable. Set items are unordered, unchangeable, and do not allow duplicate values. Unordered means that the items in a set do not

have a defined order. Set items can appear in a different order every time you use them, and cannot be referred to by index or key. Sets are unchangeable, meaning that we cannot change the items after the set has been created. Once a set is created, you cannot change its items, but you can add new items.

```
>>> s={"apple", "banana", "cherry"}
>>> s
{'cherry', 'banana', 'apple'}
>>> type(s)
<class 'set'>
```

Frozenset: The frozenset() function returns an unchangeable frozenset object (which is like a set object, only unchangeable).

Syntax is `frozenset(iterable)` An iterable object, like list, set, tuple etc.

```
>>> frozenset(s)
frozenset({'cherry', 'banana', 'apple'})
>>> p=frozenset(s)
>>> type(p)
<class 'frozenset'>
```

Boolean Type(bool): In programming you often need to know if an expression is True or False. You can evaluate any expression in Python, and get one of two answers, True or False. When you compare two values, the expression is evaluated and Python returns the Boolean answer:

```
>>> print(10>9)
True
>>> print(10==9)
False
>>> print(10<9)
False
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

1.13 Type Checking: In python to check the data type of any variable we use type () function: In python you can get the data type of any object by using the type () function:

```
>>>x = "Hello World"   # DataType Output: string
>>>type(x)
<class 'str'>
>>>x = 50              # DataType Output: integer
>>>type(x)
<class 'int'>
```

```
<class 'int'>
>>>x=25.43            # DataType Output: float
>>>type(x)
<class 'float'>
>>>x=2+4j             # DataType Output: complex
>>>type(x)
<class 'complex'>
>>>x = ["welcome", "for", "star"]    # DataType Output: list
>>>type(x)
<class 'list'>
>>>x = ("welcome", "for", "star")    # DataType Output: tuple
>>>type(x)
<class 'tuple'>
>>>x=range(2,3,1)            # DataType Output: range
>>>type(x)
<class 'range'>
>>>x = {"name": "Joy", "age": 20}   # DataType Output: dict
>>>type(x)
<class 'dict'>
>>>x = {"welcome", "for", "star"}   # DataType Output: set
>>>type(x)
<class 'set'>
>>>x = frozenset({"welcome", "for", "star"})#DataTypeOutput: frozenset
>>>type(x)
<class 'frozenset'>
>>> x=True                  # DataType Output: bool
>>> type(x)
<class 'bool'>
>>> x = b"chv"              # DataType Output: bytes
>>> type(x)
<class 'bytes'>
>>> x=b'chv'                # DataType Output: bytes
>>> type(x)
>>> type(x)
<class 'bytes'>
>>> x.= bytearray(2)        # DataType Output: bytearray
>>> type(x)
<class 'bytearray'>
```

1.14 range() function: range () allows the user to generate a series of numbers within a given range. Depending on how many arguments the user is passing to the function, the user can decide where that series of numbers will begin and end, as well as how big the difference will be between one number and the next. Python range () function can be initialized in 3 ways.

1. range (stop) takes one argument.
2. range (start, stop) takes two arguments.
3. range (start, stop, step) takes three arguments.

1. **range(stop) :** When the user call range() with one argument, the user will get a series of numbers that starts at 0 and includes every whole number up to, but not including, the number that the user has provided as the stop.

```
# printing first 6
# whole number
for i in range(6):
 ...print(i)
```

2. **range (start, stop):** When the user call range() with two arguments, the user gets to decide not only where the series of numbers stops but also where it starts, so the user doesn't have to start at 0 all the time. Users can use range() to generate a series of numbers from X to Y using range(X, Y).

Example:

```
for i in range(5, 10):
... print(i, end=" ")
```

Output: 5 6 7 8 9

3. **range (start, stop, step):** When the user call range() with three arguments, the user can choose not only where the series of numbers will start and stop, but also how big the difference will be between one number and the next. If the user doesn't provide a step, then range() will automatically behave as if the step is 1

Example:

```
for i in range(0, 10, 3):
...print(i, end=" ")
print()
```

Output: 0 3 6 9

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

- Python range() function doesn't support float numbers. i.e. user cannot use floating-point or non-integer numbers in any of its arguments.
- A sequence of numbers is returned by the range () function as its object that can be accessed by its index value. Both positive and negative indexing is supported by its object.

1.15 format () function:

- The format () method returns a formatted representation of the given value controlled by the format specifier. The format() is a built-in function in Python that converts a value into the required format/representation
- The format function converts numbers from decimal into binary, octal, and hexadecimal numbers.

Syntax of format () in Python: The format () function has two parameters (value and format spec).The syntax of the format () function is:

```
>>> Format (value, format spec)
```

The format () function has two parameters:

- Value: the value which has to be formatted (value can be a number or a string).
- Format spec (optional): This is a string for inputting the specification of the formatting requirements.

The second parameter (format spec) is optional; in its absence, the value will be formatted into a string by default.

Format of format specifier (format spec): The second parameter of the format () function called format spec follows a certain format

Format: [[fill]align][sign][#][0][width][,][.precision][type]

where, the options are

fill ::= any character
align ::= "<" | ">" | "=" | "^"
sign ::= "+" | "-" | " "
width ::= integer
precision ::= integer

type ::= "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x" | "X" | "%"

1. fill: (any character) :It is a character that fills up the empty spaces after formatting. It is also called a padding character.
2. align: It is an option to specify the alignment of the output string.
   "<": left-alignment specifier
   ">": right-alignment specifier
   "^": center-alignment specifier
   "=": justified specifier
3. sign: This determines the use of signs for the output. "+": Positive numbers have a "+" sign, and negative numbers have a "-" sign.
   "-": Negative numbers have a "-" sign.
4. " " (space): Positive numbers are preceded by a space, and negative numbers are preceded by a "-" sign.
5. "#": This specifies that the return value should have a type indication for numbers. For example, hexadecimal numbers should have a "0x" prefix added to them.
6. "0": This specifies that the output should be sign aware and padded with 0's for consistent output.
7. width: Specifies the full width of the return value.
8. ",": Specifies that the return value should have commas as a thousand separator.
9. .precision: Determines the number of characters after the decimal point.
10. type: This Specifies the output type. The types are three types:
    - String: Use an "s" or nothing at all to specify a string.
    - Integer: d (decimal), b (binary), o (octal), x (hexadecimal with lowercase characters), X (hexadecimal with uppercase characters), c (character)
    - Floating point: f (lowercase fixed point), F (uppercase fixed point), e (exponent using "e" as separator), E (exponent using "E" as separator), g (lowercase general format), G (uppercase general format), % (percentage)

Examples on format function:

Number Formatting with format()

Converting a decimal number into b(binary), o(octal), and x(hexadecimal).

```
val = 12  # Original value
Print ("Original Value'(Decimal):")
Print (val)
Print ()
# Binary value
Print ("Binary value:")
Print (format(12, "b"))  # Formatting in Binary
print()
# Octal value
print("Octal value:")
print(format(12, "o"))  # Formatting in Octal
print()
# Hexadecimal value
print("Hexadecimal value:")
print(format(12, "x"))  # Formatting in Hexadecimal
```

Output:

```
Original Value (Decimal):
12
Binary value:
1100
Octal value:
14
Hexadecimal value:
c
```

Number formatting with fill, align, sign, width, precision, and type

```
print(format(4453, "*>+7,d"))
```

Output:

```
*+4,453
```

In the above example, we formatted the integer "4453"; the format spec is *>+7,d.

Let's understand the significance of each symbol in *>+7,d:

* - The fill character fills up the empty spaces after formatting.

> - The right align option aligns the output string to the right.

+ - It is a sign option forcing the number to have a sign on its left.

7 - It is the width option forcing the number to have a minimum width of 7; empty spaces get filled by fill character.

, - It is the thousands operator, which puts a comma between all thousands.

d - It is the type specifier option specifying the number into an integer.

1.16 Math module:

- The math module is a standard module in Python and is always available. To use mathematical functions under this module, you have to import the module using

```
>>> import math
```

- Math module provides functions to deal with both basic operations such as addition (+), subtraction (-), multiplication (*), division (/) and advance operations like trigonometric, logarithmic, exponential functions.
- Python Math module provides various value of various constants like pi, tau. Having such constants saves the time of writing the value of each constant every time we want to use it and that too with great precision. Constants provided by the math module are Euler's Number, Pi, Tau, Infinity, Not a Number (NaN)

Euler's Number: The **math.e** constant returns the Euler's number: 2.71828182846.

Syntax: math.e

Pi : The pi is depicted as either 22/7 or 3.14. **math.pi** provides a more precise value for the pi.

Syntax: *math.pi*

Tau is defined as the ratio of the circumference to the radius of a circle. The **math.tau** constant returns the value tau: 6.283185307179586.

Syntax: math.tau

Infinity : Infinity basically means something which is never-ending or boundless from both directions i.e. negative and positive. It cannot be depicted by a number. The **math.inf** constant returns of positive infinity. For negative infinity, use **-math.inf**.

Syntax: math.inf

NaN : The **math.nan** constant returns a floating-point nan (Not a Number) value. This value is not a legal number. The nan constant is equivalent to float("nan").

Functions in Python Math Module: The list of all the functions and attributes defined in math module with a brief explanation of what they do.

List of Functions in Python Math Module

Function	Description
ceil(x)	Returns the smallest integer greater than or equal to x.
copysign(x, y)	Returns x with the sign of y
fabs(x)	Returns the absolute value of x
factorial(x)	Returns the factorial of x
floor(x)	Returns the largest integer less than or equal to x
fmod(x, y)	Returns the remainder when x is divided by y
frexp(x)	Returns the mantissa and exponent of x as the pair (m, e)
fsum(iterable)	Returns an accurate floating point sum of values in the iterable
isfinite(x)	Returns True if x is neither an infinity nor a NaN (Not a Number)

isinf(x)	Returns True if x is a positive or negative infinity
isnan(x)	Returns True if x is a NaN
ldexp(x, i)	Returns x * (2**i)
modf(x)	Returns the fractional and integer parts of x
trunc(x)	Returns the truncated integer value of x
exp(x)	Returns e**x
expm1(x)	Returns e**x - 1
log(x[, b])	Returns the logarithm of x to the base b (defaults to e)
log1p(x)	Returns the natural logarithm of 1+x
log2(x)	Returns the base-2 logarithm of x
log10(x)	Returns the base-10 logarithm of x
pow(x, y)	Returns x raised to the power y
sqrt(x)	Returns the square root of x
acos(x)	Returns the arc cosine of x
asin(x)	Returns the arc sine of x
atan(x)	Returns the arc tangent of x
atan2(y, x)	Returns atan(y / x)

cos(x)	Returns the cosine of x
hypot(x, y)	Returns the Euclidean norm, sqrt(x*x+y*y)
sin(x)	Returns the sine of x
tan(x)	Returns the tangent of x
degrees(x)	Converts angle x from radians to degrees
radians(x)	Converts angle x from degrees to radians
acosh(x)	Returns the inverse hyperbolic cosine of x
asinh(x)	Returns the inverse hyperbolic sine of x
atanh(x)	Returns the inverse hyperbolic tangent of x
cosh(x)	Returns the hyperbolic cosine of x
sinh(x)	Returns the hyperbolic cosine of x
tanh(x)	Returns the hyperbolic tangent of x
erf(x)	Returns the error function at x
erfc(x)	Returns the complementary error function at x
gamma(x)	Returns the Gamma function at x
lgamma(x)	Returns the natural logarithm of the absolute value of the Gamma function at x

Module-II: Operators Expressions and Functions
Operators and Expressions: Arithmetic, Assignment, Relational, Logical, Boolean, Bitwise, Membership, Identity, Expressions and Order of Evaluations, Control Statements. **Functions:** Introduction, Defining Functions, Calling Functions, Anonymous Function, Fruitful Functions and Void Functions, Parameters and Arguments, Passing Arguments, Types of Arguments, Scope of variables, Recursive Functions

Operators and Expressions:

Operators are used to perform operations on variables and values. Python divides the operators in the following groups

1. Arithmetic operators
2. Assignment operators
3. Comparison operators
4. Logical operators/Boolean
5. Identity operators
6. Membership operators
7. Bitwise operators

2.1 Arithmetic operators: Arithmetic operators are used with numeric values to perform common mathematical operations

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y

```
#Arithmetic operations
A,B=6,3
print('Addition of A+B=',A+B)
print('Subtraction of A-B=',A-B)
print('Multiplication of A*B=',A*B)
print('Division of A/B=',A/B)
print('Modulus A+B=',A%B)
print('Exponent of A**B=',A**B)
print('Floor Division of A//B=',A//B)
```

Mixed-Mode Arithmetic: The calculation which done both integer and floating-point number is called mixed-mode arithmetic. When each operand is of a different type.

Example: 9/2.0 ---> 4.5

2.2 Assignment operators: Assignment operators are used to assign values to variables

Operator	Description
=	x=y , y is assigned to x
+=	x+=y is equivalent to x=x+y
-=	x-=y is equivalent to x=x-y
*=	x*=y is equivalent to x=x*y
/=	x/=y is equivalent to x=x/y
**=	x**=y is equivalent to x=x**y

Assignment operators in Python

Operator	Example	Equivalent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

2.3 **Relational operators:** The relational operators are also known as comparison operators, comparison operators are used to compare two values stored in variables:

```
# Relational Operators in Python
x = 5
y = 2
print('x > y is',x>y)                    # Output: x > y is False
print('x < y is',x<y)                    # Output: x < y is True
print('x == y is',x==y)                  # Output: x == y is False
print('x != y is',x!=y)                  # Output: x != y is True
print('x >= y is',x>=y)                  # Output: x >= y is False
print('x <= y is',x<=y)                  # Output: x <= y is True

output:          x > y is True
                 x < y is False
                 x == y is False
                 x != y is True
                 x >= y is True
                 x <= y is False
```

2.4 Logical operators: Logical operators are used to combine conditional statements, the logical operators

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
Or	Returns True if one of the statements is true	x < 5 or x < 4
Not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

```
# Implement Logical operators in python
x = True
y = False
print('x and y is',x and y)
print('x or y is',x or y)
print('not x is',not x)
a,b,c=10,30,40
d=(a>b) and (a>c)
e=(a>b) or (a>c)
f= not (a)
print(d,e,f,sep="\n")

output: x and y is False
        x or y is True
        not x is False
        False
        False
        False
```

2.5 Bitwise operators: Bitwise operators are used to compare two (binary) numbers in digit level

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits f
>>	Signed right shift	ight by pushing copies of the leftmost bit in from the left, and l ne rightmost bits fall off

2.6 Membership operators: Python supports the following membership operators those are

1. In
2. not in

- **in:** "in" operator return true if a character or the entire substring is present in the specified string, otherwise false.
- **not in:** "not in" operator return true if a character or entire substring does not exist in the specified string, otherwise false.

2.7 Identity operators: Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

2.8 Expressions and Order of Evaluations:

- A combination of operands and operators is called an expression. The expression in Python produces some value or result after being interpreted by the Python interpreter. An example of expression can be: x=x+10. In this expression, the first 10 is added to the variable x. After the addition is performed, the result is assigned to the variable x.
- An expression in Python can contain identifiers, operators, and operands
- An identifier is a name that is used to define and identify a class, variable, or function in Python.
- An operand is an object that is operated on. On the other hand, an operator is a special symbol that performs the arithmetic or logical computations on the operands.
- The expression in Python can be considered as a logical line of code that is evaluated to obtain some result. If there are various operators in an expression then the operators are resolved based on their precedence.

Types of Expression in Python: Python supports various types of expression in Python

1. **Constant Expressions:** A constant expression in Python that contains only constant values is known as a constant expression. In a constant expression in Python, the operator(s) is a constant. A constant is a value that cannot be changed after its initialization.

   Example: x=20 +30 -10

2. **Arithmetic Expressions:** An expression in Python that contains a combination of operators, operands, and sometimes parenthesis is known as an arithmetic expression. The result of an arithmetic expression is also a numeric value just like the constant expression

   Example: x,y=10,20

   Z= x+ y

3. **Integral Expressions:** An integral expression in Python is used for computations and type conversion (integer to float, a string to integer, etc.). An integral expression always produces an integer value as a resultant.

   Example: x, y=10,2.5

   Total= x+int(y)

4. **Floating Expressions:** A floating expression in Python is used for computations and type conversion (integer to float, a string to integer, etc.). A floating expression always produces a floating-point number as a resultant.

   Example: x,y=10.25,20

   Total=x+float(y)

5. **Relational Expressions:** A relational expression in Python can be considered as a combination of two or more arithmetic expressions joined using relational operators. The overall expression results in either True or False (boolean result). A relational operator produces a boolean result so they are also known as Boolean Expressions.

   Example: result = (a + b) == (c - d)

6. **Logical Expressions:** A logical expression performs the logical computation, and the overall expression results in either True or False (boolean result).

   Example: result = x and y

7. **Bitwise Expressions:** The expression in which the operation or computation is performed at the bit level is known as a bitwise expression in Python. The bitwise expression contains the bitwise operators.

   Example: x = 25

   value= x << 1 # left_shift

8. **Combinational Expressions:** A combinational expression can contain a single or multiple expressions which result in an integer or boolean value depending upon the expressions involved.

   Example: result = x + (y << 1)

Expression (Operator Precedence) Evaluation: The operator precedence is used to define the operator's priority i.e. which operator will be executed first. The operator precedence is similar to the BODMAS rule that we learned in mathematics. Refer to the list specified below for operator precedence.

Precedence	Operator	Name
1	( ) [ ] { }	Parenthesis
2	**	Exponentiation
3	-value , +value , ~value	Unary plus or minus, complement
4	/ * // %	Multiply, Divide, Modulo
5	+ –	Addition & Subtraction
6	>> <<	Shift Operators
7	&	Bitwise AND
8	^	Bitwise XOR
9	|	pipe symbol
10	>= <= > <	Comparison Operators
11	== !=	Equality Operators

2.9 Control Statements

Decision making/control statements: Generally, statements in the script or program are executed sequential manner from the first to the last. But if you want to alter the sequential flow, you can use either conditional control or repetition control statements:

Conditional statements or decision making: a block of one or more statements will be executed if a certain expression is true. The decision control statements in python are if, else, elif statements.

If Condition: Python uses the if keyword to implement decision control. Python's syntax for executing a block conditionally is as below:

It is used to decide whether a certain statement or block of statements will be executed or not.

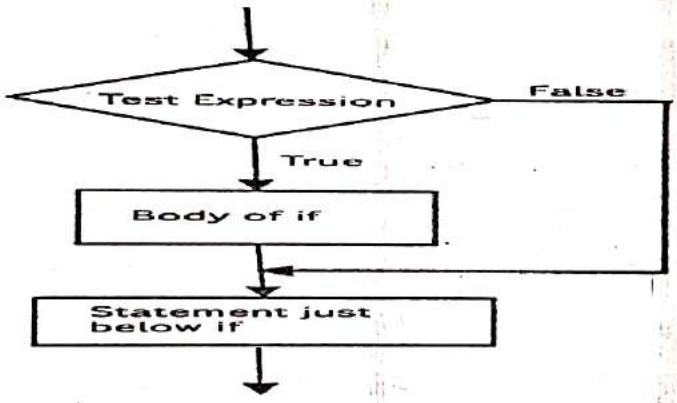
Syntax:

```
If [Boolean expression]:
   statement1
   statement2
   ...
```

Statement N

- Any Boolean expression evaluating to True or False appears after the if keyword. Use the: symbol and press Enter after the expression to start a block with increased indent. One or more statements written with the same level of indent will be executed if the Boolean expression evaluates to True.
- To end the block, decrease the indentation. Subsequent statements after the block will be executed out of the if condition.
- The if statement executing the group of statements or single statement only if the condition or expression true.

Flowchart of Python if statement:



Example:

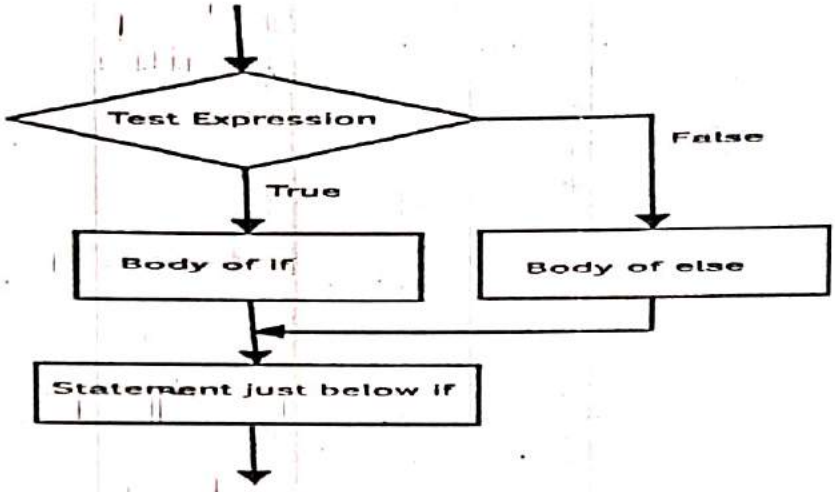
```
# Python program to illustrate If statement
i =int(input("Enter age from keyboard"))
if (i > 18):
---- print ("You are eligible for vote")
```

If-else Condition: The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But if we want to do something else if the condition is false, we can use the else statement with if statement to execute a block of code when the if condition is false.

Syntax:

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

Flowchart of Python if-else statement:



Example:

```
# Python program to illustrate If statement
i =int(input("Enter age from keyboard:"))
if (i > 18):
---- print ("You are eligible for vote")
Else:
—— print ("You are not eligible for vote")
```

Output: Enter age from keyboard: 19

You are not eligible for vote

Elif statement: elif condition is used to include multiple conditional expressions between if and else.

Syntax:

```
if [boolean expression]:
```

```
    [statements]
elif [boolean expresion]:
    [statements]
elif [boolean expresion]:
    [statements]
elif [boolean expresion]:
    [statements]
else:
    [statements]
```

Example:

```
# Python program to illustrate if-elif-else ladder
        i = 20
        if (i == 10):
                print("i is 10")
        elif (i == 15):
                print("i is 15")
        elif (i == 20):
                print("i is 20")
        else:
                print("i is not present")
```

Output: `i is 20`

Looping statements/iterative statement: Loops are important in Python or in any other programming language as they help you to execute a block of code repeatedly. You will often come face to face with situations where you would need to use a piece of code over and over but you don't want to write the same line of code multiple times.

Python programming language provides the following types of loops to handle looping requirements. Python provides three ways for executing the loops

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

syntax:

```
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.

Example:

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

Output:

```
Python
Python
Python
```

Else statement with While Loop in Python: 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.

Syntax of While Loop with else statement:

```
while condition:
        # execute these statements
else:
        # execute these statements
```

Example:

```
# Python program to illustrate
# combining else with while
count = 0
```

```
while (count <2):
        count = count + 1
        print("Hello")
else:
        print("python)
```

Output:

Hello

Hello

python

Infinite While Loop in Python: If we want a block of code to execute infinite number of time, we can use the while loop in Python to do so.

Note: It is suggested not to use this type of loop as it is a never-ending infinite loop where the condition is always true and you have to forcefully terminate the compiler.

Example:

```
# Python program to illustrate
# Single statement while block
count = 0
while (count == 0):
        print("Hello Geek")
```

2. **For Loop in Python:** For loops are used for sequential traversal. For example: traversing a list or string or array etc. In Python, there is "for in" loop which is similar to for each loop in other languages. Let us learn how to use for in loop for sequential traversals.

   Syntax:

```
for iterator_var in sequence:
        Statements(s)
```

Example with List, Tuple iteration using For Loops in Python

```
# Python program to illustrate
# Iterating over a list
print("List Iteration")
l = ["python", "for", "lab"]
for i in l:
    print(i)

# Iterating over a tuple (immutable)
print("\nTuple Iteration")
t = ("python", "for", "lab")
for i in t:
    print(i)
```

```
List Iteration
python
for
lab

Tuple Iteration
python
for
lab
```

It can be used to iterate over a range and iterators.

```
# Python program to illustrate
# Iterating over range 0 to n-1
n = 4
for i in range(0, n):
    print(i)
```

Iterating by the index of sequences: We can also use the index of elements in the sequence to iterate. The key idea is to first calculate the length of the list and in iterate over the sequence within the range of this length. See the below example:

Example:

```
# Python program to illustrate
# iterating by index
List = ["python", "for", "lab"]
For index in range (len (list)):
   Print (list [index])
```

Output:

```
Python
For
Lab
```

Else statement with for loop in Python: 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.

Example:

```
# Python program to illustrate
# combining else with for
list = ["geeks", "for", "geeks"]
for index in range(len(list)):
    print(list[index])
else:
    print("Inside Else Block")
```

Nested Loops: Python programming language allows to use one loop inside another loop.

Syntax:

```
for iterator_var in sequence:
    for iterator_var in sequence:
        statements(s)
    statements(s)
```

The syntax for a nested while loop statement in the Python programming language is as follows:

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

Note: we can use any loop inside the another loop

Loop control statements: You might face a situation in which you need to exit a loop completely when an external condition is triggered or there may also be a situation when you want to skip a part of the loop and start next execution.

Python provides break and continue statements to handle such situations and to have good control on your loop.

The following are the loop control statements in python

1. Break
2. Continue
3. Pass

1. **The Break Statement:** The break statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in C. The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The break statement can be used in both while and for loops.

Example:

```
for letter in 'Python':   # First Example
----if letter == 'h':
   ----break
print 'Current Letter :', letter
```

Output: This will produce the following result:

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

```
var = 10                  # Second Example
while var > 0:
print 'Current variable value :', var
var = var -1
if var == 5:
----break
print "Good bye!"
```

output:

```
Current variable value : 10
Current variable value : 9
Current variable value : 8
```

Current variable value : 7

Current variable value : 6

Good bye!

2. **The continue statement:** The continue statement in Python returns the control to the beginning of the while loop. The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.
   The continue statement can be used in both while and for loops.

Example:

```
for letter in 'Python': # First Example
----if letter == 'h':
--------continue
Print(letter,sep="\t")
```

Output:

P y t o n

```
var =4  # Second Example
while var > 0:
var = var -1
if var ==3 :
continue
print 'Current variable value :', var
print "Good bye!"
```

output:

Current variable value : 4

Current variable value : 2

Current variable value : 1

Good bye!

The else statement used with loops

Python supports to have an else statement associated with a loop statements.

If the else statement is used with a for loop, the else statement is executed when the loop has exhausted iterating the list. If the else statement is used with a while loop, the else statement is executed when the condition becomes false.

Example:

```
for letter in 'Python': # First Example
if letter == 'h':
continue
print 'Current Letter :', letter
```

3. **The pass statement:** The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.
   The pass statement is a null operation; nothing happens when it executes. The pass is also useful in places where your code will eventually go, but has not been written yet

Example:

```
for letter in 'Python':
----if letter == 'h':
   -----pass
   ----print 'This is pass block'
print 'Current Letter :', letter
print "Good bye!"
```

- The preceding code does not execute any statement or code if the value of letter is 'h'. The pass. Statement is helpful when you have created a code block but it is no longer required.

- You can then remove the statements inside the block but let the block remain with a pass statement so that it doesn't interfere with other parts of the code

Python Functions:

- In Python, a function is a group of related statements that performs a specific task.
- Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.
- 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. It avoids repetition and makes the code reusable.

Benefits of Using Functions

1. Increase Code Readability
2. Increase Code Reusability

Types of Python Functions: There are many types of Python Functions. And each of them is very vital in its own way. The following are the different types of Python Functions:

1. Python Built-in Functions
2. Python Lambda Functions
3. Python User-defined Functions
4. Python Recursion Functions

1. Python Built-in Functions:

The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. For example, print () function prints the given object to the standard output device (screen) or to the text stream file.

In Python 3.6, there are 68 built-in functions. But for the sake of simplicity let us consider the majorly used functions and we can build on from there.

1. **Python abs() Function:** The abs() method returns the absolute value of the given number. If the number is a complex number, abs() returns its magnitude.

   Syntax : abs(num)

2. **Python all() Function:** The all() method returns True when all elements in the given iterable are true. If not, it returns False.

   The syntax of all() method is: all(iterable)

   The all() method takes a single parameter:

   iterable – Any iterable (list, tuple, dictionary, etc.) which contains the elements

3. **Python ascii() Function:** The ascii() method returns a string containing a printable representation of an object. It escapes the non-ASCII characters in the string using x, u or U escapes.

   The syntax of ascii() method is: ascii(object)

4. Python enumerate() Function: The enumerate() method adds counter to an iterable and returns it (the enumerate object).

   The syntax of enumerate() method is: enumerate(iterable, start=0)

   Parameters: The enumerate() method takes two parameters:

   iterable – a sequence, an iterator, or objects that support iteration

   start (optional) – enumerate() starts counting from this number. If start is omitted, 0 is taken as the start.

   Example:

   ```
   grocery = ['bread', 'milk', 'butter']
   enumerateGrocery = enumerate(grocery)
   print(type(enumerateGrocery))
   ```

5. Python eval() Function: The eval() method parses the expression passed to this method and runs python expression (code) within the program.

   The syntax of eval() method is: eval(expression, globals=None, locals=None)

   Parameters: The eval() takes three parameters:

Expression – this string is parsed and evaluated as a Python expression

globals (optional) – a dictionary

locals (optional)- a mapping object. Dictionary is the standard and commonly used mapping type in Python.

Example:

```
x = 1
print(eval('x + 1'))
```

Output

sum = 11

6. Python help() Function: The help() method calls the built-in Python help system.

   The syntax of help() method is: help(object)

   Parameters: The help() method takes the maximum of one parameter.

   object (optional) – you want to generate the help of the given object

   Example : >>> help('print')

7. Python bin() Function: The bin() method converts and returns the binary equivalent string of a given integer. If the parameter isn't an integer, it has to implement __index__() method to return an integer.

   The syntax of bin() method is: bin(num)

   Parameters: The bin() method takes a single parameter:

   num – an integer number whose binary equivalent is to be calculated.

   If not an integer, should implement __index__() method to return an integer.

   Example:

```
number = 5
print('The binary equivalent of 5 is:', bin(number))
```

8. Python sum() Function: The sum() method returns the reversed iterator of the given sequence.

The syntax of sum() method is: sum(iterable, start)

Parameters:

iterable – iterable (list, tuple, dict etc) whose item's sum is to be found. Normally, items of the iterable should be numbers.

start (optional) – this value is added to the sum of items of the iterable. The default value of start is 0 (if omitted)

Example: numbers = [2.5, 3, 4, -5]

numbersSum = sum(numbers)

print(numbersSum)

output: 4.5

2. **Python Anonymous/Lambda Function:**

- In Python, an anonymous function is a function that is defined without a name.
- While normal functions are defined using the def keyword in Python, anonymous functions are defined using the lambda keyword.
- Hence, anonymous functions are also called lambda functions.
- Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

Syntax of Lambda Function: lambda arguments: expression

Example: # Program to show the use of lambda functions

double = lambda x: x * 2

print(double(5))

Output:

10

Use of Lambda Function in python:

- We use lambda functions when we require a nameless function for a short period of time.
- In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like filter (), map () etc.

Example use with filter(): The filter() function in Python takes in a function and a list as arguments.

```
# Program to filter out only the even items from a list
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list (filter (lambda x: (x%2 == 0), my_list))
Print (new_list)
```

Output:

[4, 6, 8, 12]

Example use with map (): The map () function in Python takes in a function and a list.

```
# Program to double each item in a list using map ()
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(map(lambda x: x * 2 , my_list))
print(new_list)
```

Output: [2, 10, 8, 12, 16, 22, 6, 24]

3. Python User-defined Functions

Functions that we define ourselves to do certain specific task are referred as user-defined functions.

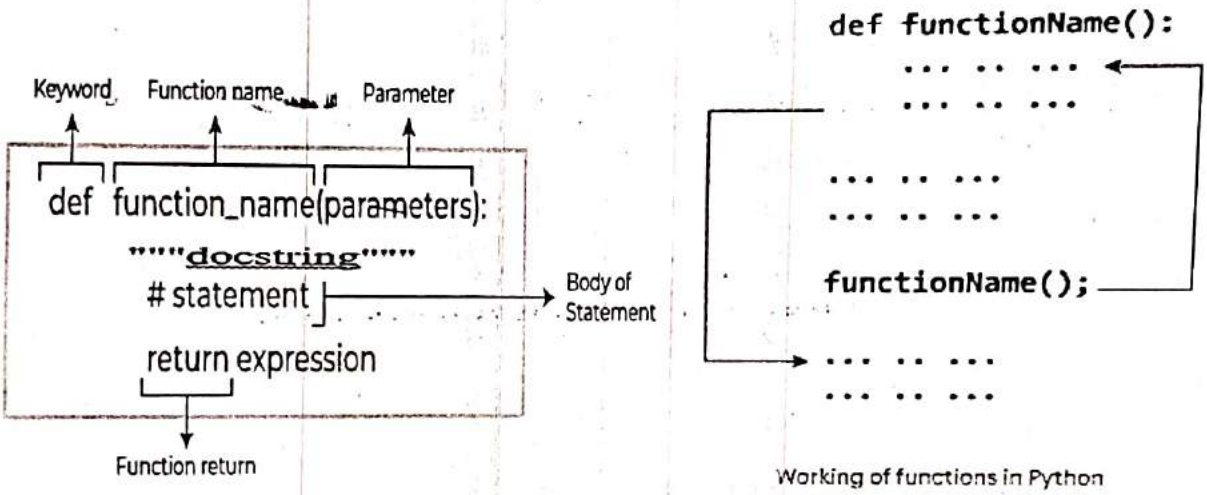
If we use functions written by others in the form of library, it can be termed as library functions. All the other functions that we write on our own fall under user-defined functions. So, our user-defined function could be a library function to someone else.

Advantages of user-defined functions:

- User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
- If repeated code occurs in a program. Function can be used to include those codes and executewhen needed by calling that function.
- Programmers' working on large project can divide the workload by making different functions.

Creating user-defined function

Syntax of function



Working of functions in Python

Above shown is a function definition that consists of the following components.

- Keyword def that marks the start of the function header.
- A function name to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python.
- Parameters (arguments) through which we pass values to a function. They are optional.
- A colon (:) to mark the end of the function header.
- Documentation string (docstring) to describe what the function does which is optional.
- One or more valid python statements that make up the function body. Statements must have the same indentation level (usually 4 spaces).

- Return statement to return a value from the function which is optional.

Example of a function:

```
def greet(name):                #function defination
"""
This function greets to
the person passed in as
a parameter
"""
        print("Hello, " + name + ". Good morning!")

greet('Tirumala')    # calling function
```

Calling a function: Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
def greet(name):        #function defination
----print("Hello, " + name + ". Good morning!")

greet('Tirumala')    # calling function
```

The return statement:

- The return statement is used to exit a function and go back to the place from where it was called.
- This statement can contain an expression that gets evaluated and the value is returned.
- If there is no expression in the statement or the return statement itself is not present inside afunction, then the function will return none object.

Syntax of return: return [expression_list]

Example:

```
def absolute_value(num):
"""This function returns the absolute value of the entered number"""
----if num >= 0:
--------return num
    else:
    ----return -num
print(absolute_value(2))
print(absolute_value(-4))
```

Docstrings:

- The first string after the function header is called the docstring and is short for documentation string. It is briefly used to explain what a function does.

- Python docstrings are the string literals that appear after the definition of a method, class, or module also.
- In the above example, we have a docstring immediately below the function header. We generally use triple quotes so that docstring can extend up to multiple lines.
- We can access these docstrings using the __doc__ attribute.

```
print(greet.__doc__)
This function greets to
the person passed in as
a parameter
```

Example-1:

```
def square(n):
'''Takes in a number n, returns the square of n'''
    return n**2
print(square.__doc__)
```

Output:

Takes in a number n, returns the square of n

Arguments:

- Information can be passed into functions as arguments.
- Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

Example:

```
def my_function(fname,lname):
    print(fname + " " + lname)
my_function('Van','Rossum')
```

output:

Van Rossum

Parameters and Arguments: The terms parameter and argument can be used for the same thing: information that are passed into a function the parameters and arguments are

From a function's perspective:

- A parameter is the variable listed inside the parentheses in the function definition.
- An argument is the value that is sent to the function when it is called.

formal and actual arguments:

- A formal parameter, i.e. a parameter, is in the function definition.
- An actual parameter, i.e. an argument, is in a function call.

```
def square(n):          # n is formal argument
----return n*n
a=2
b=3
print(square(a))        # a is actual argument
print(square(b))        # b is actual argument
```

Python Function Arguments: In Python, you can define a function that takes variable number of arguments.

Example:

```
def greet(name, msg):
"""This function greets to
the person with the provided message"""
----print("Hello", name + ', ' + msg)
greet("Titumala", "Good morning!")
```

Output:

Hello Tirumala, Good morning!

- Here, the function greet() has two parameters. Since we have called this function with two arguments, it runs smoothly and we do not get any error.
- If we call it with a different number of arguments, the interpreter will show an error message.
- If we call to this function with one and no arguments along with their respective error messages.

```
>>> greet ("Monica") # only one argument
Type Error: greet () missing 1 required positional argument: 'msg'
>>> greet () # no arguments
Type Error: greet() missing 2 required positional arguments: 'name' and 'msg'
```

Fruitful function/ Functions with Return Values

A fruitful function is a function that returns a value when it is called. Most of the built-in functions that we have used are fruitful. For example, the function abs returns a new number – namely, the absolute value of its argument:

```
>>> abs(-42)
42
```

Defining a fruitful function

If we want a function to return a result to the caller of the function, we use the return statement. For example, here we define two fruitful functions.

Example:

```
import math
def square(x):
----return x * x
def circle area (diameter ):
-----radius = diameter / 2.0
-----return math.pi * square(radius)
```

In general, a return statement consists of the return keyword followed by an expression, which is evaluated and returned to the function caller. How python evaluates a return statement Python evaluates fruitful functions pretty much the same way as non-fruitful. The only thing new is how it executes a return statement. Here's how:

1. Evaluate the expression. This produces a memory address.
2. Pass back that memory address to the caller. Leave the function immediately and return to the location where the function was called.

Python returns immediately when it reaches a return statement. The print statement in the function body below will never be executed:

Void functions/Functions without Return Values:

In python some functions do not return anything at all. A function does not have to return a value. This kind of function is commonly known as a void function in programming terminology.

The following program (Program 3) defines a function named printGrade and invokes (calls) it to print the grade based on a given score.

```
# Print grade for the score
def printGrade(score):
----if score >= 90.0:
    ----print('A')
score = int(input("Enter a score: "))
printGrade(score)
```

Types of Python Function Arguments: Python supports various types of arguments that can be passed at the time of the function call. In Python, we have the following 4 types of function arguments.

1. Default argument
2. Keyword arguments (named arguments)
3. Positional arguments
4. Arbitrary arguments (variable-length arguments *args and **kwargs)

1. Default Arguments: Function arguments can have default values. A default argument is a parameter that assumes a default value, if a value is not provided in the function call for that argument. We can provide a default value to an argument by using the assignment operator (=).

Example:

```
# Python program to demonstrate
# default arguments
def greet(name, msg="Good morning!"):
----print("Hello", name + ', ' + msg)
greet("Chiru")
greet("CSE", "Welcome to Technical Wing.")
```

Output:

Hello Chiru, Good morning!

Hello CSE, Welcome to Technical Wing.

Explanation:

- The parameter **name** does not have a default value and is required (mandatory) during a call.
- The parameter **msg** has a default value of **"Good morning!"**. So, it is optional during a call. If a value is provided, it will overwrite the default value.
- Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.
- This means to say, non-default arguments cannot follow default arguments.

Example:

```
def greet(msg = "Good morning!", name):
```

Output: Syntax Error: non-default argument follows default argument.

2. **Keyword Arguments:** Python allows functions to be called using keyword arguments. When we call functions in this way, the idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.

Example:

```
# Python program to demonstrate Keyword Arguments
def student(firstname, lastname):
----print(firstname, lastname)
 # Keyword arguments
student(firstname='Geeks', lastname='Practice')
student(lastname='Practice', firstname='Geeks')
```

3. **Positional Arguments**

We used the Position argument during the function call so that the first argument (or value) is assigned to name and the second argument (or value) is assigned to age. By changing the position, or if you forget the order of the positions, the values can be used in the wrong places, as shown in the Case-2 example below, where 27 is assigned to the name and Suraj is assigned to the age.

Example:

```
def nameAge(name, age):
----print("Hi, I am", name)
```

```
----print("My age is ", age)
# You will get correct output because
# argument is given in order
print("Case-1:")
nameAge("Rossum", 67)
# You will get incorrect output because
# argument is not in order
print("\nCase-2:")
nameAge(67, "Rossum")
```

4. Arbitrary Arguments / Variable Length Arguments:

- Sometimes, we do not know in advance the number of arguments that will be passed into a function.
- Python allows us to handle this kind of situation through function calls with an arbitrary number of arguments. There are two special symbols:
    1. *args in Python (Non-Keyword Arguments)
    2. **kwargs in Python (Keyword Arguments)

Example program for Non-Keyword Arguments

```
# Python program to illustrate
# *args for variable number of arguments
def myFun(*argv):
----for arg in argv:
    ----print(arg)
myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')
```

Example program for Keyword Arguments

```
# Python program to illustrate
# *kwargs for variable number of keyword arguments
def myFun(**kwargs):
----for key, value in kwargs.items():
    ----print("%s == %s" % (key, value))
# Driver code
myFun(first='Geeks', mid='for', last='Geeks')
```

Scope and Lifetime of variables:

- Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.
- The lifetime of a variable is the period throughout which the variable exits in the memory. The life time of variables inside a function is as long as the function executes.
- They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Example:

```
def my_fun():
----x = 10
----print("Value inside function:",x)
x = 20
my_fun()
print("Value outside function:",x)
```

Output:

```
Value inside function: 10
Value outside function: 20
```

- In the above example, we can see that the value of x is 20 initially. Even though the function my_fun() changed the value of x to 10, it did not affect the value outside the function.
- This is because the variable x inside the function is different (local to the function) from the one outside. Although they have the same names, they are two different variables with different scopes.

Python Global, Local and Nonlocal variables

Global Variables: In Python, a variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.

Example 1: Create a Global Variable

```
x = "global"
def fun():
----print("x inside:", x)
```

```
fun()
print("x outside:", x)
```

Output:

```
x inside: global
x outside: global
```

Example 2:

```
x = "global"
def fun():
----x = x * 2
----print(x)
fun()
```

Output:

UnboundLocalError: local variable 'x' referenced before assignment

- The output shows an error because Python treats x as a local variable and x is also not defined inside fun(). To make this work, we use the global keyword.

Example 3: Changing Global Variable from inside a Function using global

```
c = 0 # global variable
def add():
----global c
----c = c + 2 # increment by 2
----print("Inside add():", c)
print("Before In main:", c)
add()
print("In main:", c)
```

Output:

```
Before In main: 0
Inside add(): 2
In main: 2
```

Local Variables: A variable declared inside the function's body or in the local scope is known as a local variable.

Example 1: Accessing local variable outside the scope

```
def fun():
----y = "local"
fun()
print(y)
Output:
        NameError: name 'y' is not defined
```

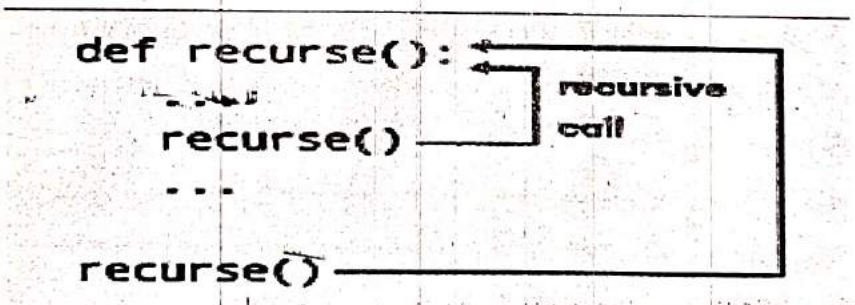
- The output shows an error because we are trying to access a local variable y in a global scope whereas the local variable only works inside fun() or local scope.

Example 2: Create a Local Variable

```
def fun():
----y = "local"
----print(y)
fun()
Output:
        Local
```

Recursive Function:

- A function that calls itself is known as Recursive Function.
- A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.
- The following image shows the working of a recursive function called recurse.



Recursive Function in Python

Example: Factorial of a number is the product of all the integers from 1 to that number.
The factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.

```
def factorial(x):
----if x == 1:
    ----return 1
```

```
    else:
    ----return (x * factorial(x-1))
num = 3
print("The factorial of", num, "is", factorial(num))
```

Output:

The factorial of 3 is 6

Advantages of Recursion:

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion:

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

Infinite Recursion: If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as infinite recursion, and it is generally not a good idea. Here is a minimal program with an infinite recursion:

```
def recurse():
----recurse()
```

In most programming environments, a program with infinite recursion does not really run forever. Python reports an error message when the maximum recursion depth is reached:

Module-III: Strings, Lists, Tuples, and Dictionaries

Strings, Lists, Tuples, and Dictionaries: Strings- Operations, Slicing, Methods, List- Operations, slicing, Methods, Tuple- Operations, Methods, Dictionaries- Operations, Methods, Mutable Vs Immutable, Arrays Vs Lists, Map, Reduce, Filter, Comprehensions.

III.1 Introduction to Strings and operations

A String is a data structure in Python that represents a sequence of characters. It is an immutable data type, meaning that once you have created a string, you cannot change it.

- Strings are used widely in many different applications, such as storing and manipulating text data, representing names, addresses, and other types of data that can be represented as text.

Creating a String in Python: Strings in Python can be created using single quotes or double quotes or even triple quotes. Create a string using single quotes (' '), double quotes (" "), and triple double quotes (""" """). The triple quotes can be used to declare multiline strings in Python.

Example:

```
# Python Program for Creation of String
# creating a String with single Quotes
String1 = 'Welcome to Python programming'
Print ("String with the use of Single Quotes: ")
print (String1)
# creating a String with double Quotes
String2 = "Welcome to Python programming"
print ("\n String with the use of Double Quotes: ")
Print (String2)
# creating a String with triple Quotes
String3 = """I'm D Samantha author of this book"""
print("\n String with the use of Triple Quotes: ")
print (String3)
```

Output:

```
String with the use of Single Quotes:
Welcome to Python programming

String with the use of Double Quotes:
Welcome to Python programming

String with the use of Triple Quotes:
Im D Samantha author of this book

Creating a multiline String:
Welcome
To
Python
```

```
# creating String with triple Quotes allows multiple lines
print("\n Creating a multiline String: ")
String3 = '''Welcome
                To
                Python'''
print (String3)
```

Accessing characters in Python String: In Python, individual characters of a String can be accessed by using the method of Indexing. Indexing allows negative address references to access characters from the back of the String, e.g. -1 refers to the last character, -2 refers to the second last character, and so on. While accessing an index out of the range will cause an Index Error. Only Integers are allowed to be passed as an index, float or other types that will cause a Type Error.

G	E	E	K	S	F	O	R	G	E	E	K	S
0	1	2	3	4	5	6	7	8	9	10	11	12
-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Python String indexing

 # Python Program to Access characters of String

```
String1 = "welcome to python"
print("\n First  and nine character of String is using positive indexing : ")
print(String1 [0],String1[8])
print("\n Last character of String using negative index: ")
print(String1[-1])
```

Output:

First and nine character of String is using positive indexing:

w t

Last character of String using negative index:

n

III.2 String Slicing:

In Python, the String Slicing method is used to access a range of characters in the String. Slicing in a String is done by using a Slicing operator, i.e., a colon (:). One thing to keep in mind while using this method is that the string returned after slicing includes the character at the start index but not the character at the last index.

Syntax: [start index :end Index]

Example:

```
# Creating a String
Name = "python programming"
print("Initial String: ")
print(Name)
# Printing 3rd to 12th character
print("\nSlicing characters from 3-12: ")
print(Name[3:12])
```

Output:

Initial String:
python programming
Slicing characters from 3-12:
hon progr

Reversing a Python String: we can also reverse strings in Python. We can reverse a string by using String slicing method. We will reverse a string by accessing the index. We did not specify the first two parts of the slice indicating that we are considering the whole string, from the start index to the last index.

Print (name [::-1]) **output:** Reverse of a string: gnimmargorp nohtyp

III.3 String Methods

- Python has a set of built-in methods that you can use on strings.
- Note: All string methods returns new values. They do not change the original string.

S.No	Method	Description
1	capitalize()	Converts the first character to upper case
2	casefold()	Converts string into lower case
3	center()	Returns a centered string
4	count()	Returns the number of times a specified value occurs in a string
5	encode()	Returns an encoded version of the string
6	endswith()	Returns true if the string ends with the specified value
7	expandtabs()	Sets the tab size of the string
8	find()	Searches the string for a specified value and returns the position of where it was found
9	index()	Searches the string for a specified value and returns the position of where it was found
10	isalnum()	Returns True if all characters in the string are alphanumeric
11	isalpha()	Returns True if all characters in the string are in the alphabet
12	isdecimal()	Returns True if all characters in the string are decimals
13	isdigit()	Returns True if all characters in the string are digits
14	isidentifier()	Returns True if the string is an identifier
15	islower()	Returns True if all characters in the string are lower case
16	isnumeric()	Returns True if all characters in the string are numeric
17	isprintable()	Returns True if all characters in the string are printable

18	isspace()	Returns True if all characters in the string are whitespaces
19	istitle()	Returns True if the string follows the rules of a title
20	isupper()	Returns True if all characters in the string are upper case
21	join()	Joins the elements of an iterable to the end of the string
22	ljust()	Returns a left justified version of the string
23	lower()	Converts a string into lower case
24	replace()	Returns a string where a specified value is replaced with a specified value
25	split()	Splits the string at the specified separator, and returns a list
26	startswith()	Returns true if the string starts with the specified value
27	strip()	Returns a trimmed version of the string
28	swapcase()	Swaps cases, lower case becomes upper case and vice versa
29	title()	Converts the first character of each word to upper case
30	upper()	Converts a string into upper case
31	rfind()	Searches the string for a specified value and returns the last position of where it was found
32	rindex()	Searches the string for a specified value and returns the last position of where it was found
33	rjust()	Returns a right justified version of the string
34	rsplit()	Splits the string at the specified separator, and returns a list
35	rstrip()	Returns a right trim version of the string
36	splitlines()	Splits the string at line breaks and returns a list
37	translate()	Returns a translated string
38	zfill()	Fills the string with a specified number of 0 values at the beginning

1. **capitalize ():** Converts the first character to upper case

Syntax: string.capitalize ()

Example: 1

```
>>> str="python is a trending programming language"
>>> str.capitalize ()
```

Output: 'Python is a trending programming language'

2. **casefold ():** Converts string into lower case. This method is similar to the lower() method, but the casefold()method is stronger, more aggressive, meaning that it will convert more characters into lower case, and will find more matches when comparing two strings and both are converted using the casefold()method.

Syntax: string.casefold ()

Example: 1

>>> str='PYTHON IS A TRENDING PROGRAMMING LANGUAGE'

>>> str.casefold ()

Output: 'python is a trending programming language'

3. **Center ():** The center () method will center align the string, using a specified character (space is default) as the fill character.

Syntax: string.center (length, character)

Length = Required. The length of the returned string.

Character = Optional. The character to fill the missing space on each side. Default is " " (space)

Example-1: Print the word "Engineering", taking up the space of 128 characters, with "Engineering" in the middle:

str1="Engineering"

>>> str1.center (128)

Output: ----------' Engineering '----------

4. **Count ():** The count () method returns the number of times a specified value appears in the string.

Syntax: string. Count (value, start, end)

Value =Required. A String. The string to value to search for

Start = Optional. An Integer. The position to start the search. Default is 0

end = Optional. An Integer. The position to end the search.

Default is the end of the string

Example-1: Return the number of times the value "is" appears in the string:

str2="my favorite subject is programming, python is a programming language"

str2.count ('is')

Output: 2

Example-2: Search from position 10 to 24:

>>>str2='my favorite subject is programming, python is a programming language'

>>>str2.count ('is', 20, 40) **Output:** 1

5. **encode ():** The encode() method encodes the string, using the specified encoding. If no encoding is specified, UTF-8 will be used.

Syntax: string.encode (encoding=encoding, errors=errors)

Encoding = Optional. A String specifying the encoding to use. Default is UTF-8

Errors = Optional. A String specifying the error method.

Legal values are:

- backslashreplace' - uses a backslash instead of the character that could not be encoded
- 'ignore'- ignores the characters that cannot be encoded
- 'name replace'- replaces the character with a text explaining the character
- 'strict' - Default, raises an error on failure
- 'Replace'- replaces the character with a question mark
- 'xmlcharrefreplace' - replaces the character with an xml character

6. **Endswith ():** The endswith () method returns True if the string ends with the specified value, otherwise False.

Syntax: string. Endswith (value, start, end)

Value Required. The value to check if the string ends with

Start Optional. An Integer specifying at which position to start the search

End Optional. An Integer specifying at which position to end the search

Example-1:

txt = "Hello, welcome to my world." x = txt.endswith ("my world.") print(x)

Output: True

7. **expandtabs ():** The expandtabs() method sets the tab size to the specified number of whitespaces.

Syntax: string.expandtabs (tabsize)

Here: tabsize Optional. A number specifying the tabsize. Default tabsize is 8

Example-1: txt = "H\te\tl\tl\to"

x = txt.expandtabs (2)

print(x)

Output: H e l l o

8. **find ():** The find () method finds the first occurrence of the specified value. The find() method returns -1 if the value is not found.

   The find() method is almost the same as the index() method, the only difference is that the index() method raises an exception if the value is not found.

Syntax: string. find(value, start, end)

Here: value required. The value to search for, start Optional. Where to start the search. Default is 0, end Optional. Where to end the search. Default is to the end of the string

Example-1:

txt = "Hello, welcome to my world."

x = txt.find("welcome")

Print(x)

Output: 7

9. Index (): The index () method finds the first occurrence of the specified value. The index () method raises an exception if the value is not found. The index() method is almost the same as the find() method, the only difference is that the find() method returns -1 if the value is not found

Syntax: string. Index (value, start, end)

Here: **value** required. The value to search for, **start** is Optional. Where to start the search. Default is 0, **end** Optional. Where to end the search. Default is to the end of the string

Example-1:

```
txt = "Hello, welcome to my world."
x = txt.index("welcome")
Print(x)
```

Output: 7

10. **Isalnum ():** The isalnum () method returns True if all the characters are alphanumeric, meaning alphabet letter (a-z) and numbers (0-9). Example of characters that are not alphanumeric: (space)!#%&? etc.

Syntax: string. Isalnum ()

Example-1:

```
txt = "Company12"
x = txt.isalnum()
print(x)
```

Output: True

11. **Isalpha ():** The isalpha () method returns True if all the characters are alphabet letters (a-z). Example of characters that are not alphabet letters: (space)!#%&? etc.

Syntax: string. Isalpha ()

Example-1:

```
txt = "CompanyX"
x = txt.isalpha()
print(x)
```

Output: True

12. **Isdecimal ():** The isdecimal () method returns True if all the characters are decimals (0-9). This method is used on Unicode objects.

Syntax: string. Isdecimal ()

Example-1:

```
txt = "\u0033" #unicode for 3
x = txt.isdecimal ()
print(x)
```

Output: True

13. **Isdigit ():** The isdigit () method returns True if all the characters are digits, otherwise False. Exponents, like ², are also considered to be a digit.

Syntax: string. Isdigit ()

Example-1:

```
txt = "50800"
x = txt.isdigit()
print(x)
```

Output: True

14. **Isidentifier ():** The isidentifier () method returns True if the string is a valid identifier, otherwise False. A string is considered a valid identifier if it only contains alphanumeric letters (a-z) and (0-9), or underscores (_). A valid identifier cannot start with a number, or contain any spaces.

Syntax: string.isidentifier ()

Example-1:

```
txt = "Demo"
x = txt.isidentifier ()
print(x)
```

Output: True

15. **Islower ():** The islower () method returns True if all the characters are in lower case, otherwise False. Numbers, symbols and spaces are not checked, only alphabet characters.

Syntax: string. Islower ()

Example-1:

txt = "hello world!"

x = txt.islower()

print(x)

Output: True

16. **Isnumeric ():** The isnumeric () method returns True if all the characters are numeric (0-9), otherwise False. Exponents, like ² and ¾ are also considered to be numeric values.

Syntax: string. Isnumeric ()

Example-1:

txt = "565543"

x = txt.isnumeric()

print(x)

Output: True

17. **Isprintable ():** The isprintable () method returns True if all the characters are printable, otherwise False. Example of none printable character can be carriage return and line feed.

Syntax: string. Isprintable ()

18. **Isspace ():** The isspace () method returns True if all the characters in a string are whitespaces, otherwise False.

Syntax: string. Isspace ()

Example-1:

txt = " "

x = txt.isspace()

print(x)

Output: True

19. **istitle():** The istitle() method returns True if all words in a text start with a upper case letter, AND the rest of the word are lower case letters, otherwise False. Symbols and numbers are ignored.

Syntax: string.istitle ()

Example-1:

```
txt = "Hello, And Welcome To My World!"
x = txt.istitle()
print(x)
```

Output: True

20. **Isupper ():** The isupper () method returns True if all the characters are in upper case, otherwise False. Numbers, symbols and spaces are not checked, only alphabet characters.

Syntax: string. Isupper ()

Example-1:

```
txt = "THIS IS NOW!"
x = txt.isupper()
print(x)
```

Output: True

21. **Join ():** The join () method takes all items in an iterable and joins them into one string.A string must be specified as the separator.

Syntax: string.join (iterable)

Example-1:

```
MyTuple = ("John", "Peter", "Vicky")
x = "#".join (MyTuple)
Print(x)
```

Output: John#Peter#Vicky

22. **Ljust ():** The ljust () method will left align the string, using a specified character (space is default) as the fill character.

Syntax: string. Ljust (length, character)

Here:: length Required. The length of the returned string, **character** Optional. A character to fill the missing space (to the right of the string). Default is " " (space).

Example-1:

```
txt = "banana"
x = txt.ljust(20)
print(x, "is my favorite fruit.")
```

Output: banana is my favorite fruit.

23. **lower ():** The lower () method returns a string where all characters are lower case. Symbols and Numbers are ignored.

Syntax: string. lower ()

Example-1:

```
txt = "Hello my FRIENDS"
x = txt.lower()
print(x)
```

Output: hello my friends

24. **replace():** The replace() method replaces a specified phrase with another specified phrase.

Note: All occurrences of the specified phrase will be replaced, if nothing else is specified.

Syntax: string.replace (oldvalue, newvalue, count)

Here: oldvalue: The string to search for (required)

Newvalue: The string to replace the old value with (required)

Count: A number specifying how many occurrences of the old value you want to replace. Default is all occurrences (optional)

Example-1:

```
txt = "I like bananas"
x = txt.replace ("bananas", "apples")
print(x)
```

Output: I like apples 17 www.Jntufastupdates.com

25. **split():** The split() method splits a string into a list. You can specify the separator, default separator is any whitespace. When maxsplit is specified, the list will contain the specified number of elements plus one.

Syntax: string.split (separator, maxsplit)

separator (Optional): Specifies the separator to use when splitting the string. By default any whitespace is a separator

maxsplit (Optional): Specifies how many splits to do. Default value is -1, which is "all occurrences"

Example-1:

```
txt = "welcome to the jungle"
x = txt.split()
print(x)
```

Output: ['welcome', 'to', 'the', 'jungle']

26. **startswith():** The startswith() method returns True if the string starts with the specified value, otherwise False.

Syntax: string.startswith (value, start, end)

Example-1:

```
txt = "Hello, welcome to my world."
x = txt.startswith("Hello")
print(x)
```

Output: True

27. **strip():** The strip() method removes any leading (spaces at the beginning) and trailing (spaces at the end) characters (space is the default leading character to remove)

Syntax: string.strip (characters)

Characters (Optional): A set of characters to remove as leading/trailing characters.

28. **swapcase ():** The swapcase () method returns a string where all the upper case letters are lower case and vice versa.

Syntax: string.swapcase ()

Example-1:

```
txt = "Hello My Name Is Purple"
x = txt.swapcase ()
Print(x)
```

Output: hELLO mY nAME iS pURPLE

29. **title ():** The title() method returns a string where the first character in every word is upper case. Like a header, or a title. If the word contains a number or a symbol, the first letter after that will be converted to uppercase.

Syntax: string.title ()

Example-1:

```
txt = "Welcome to my world"
x = txt.title()
print(x)
```

Output: Welcome To My World

30. **upper ():** The upper () method returns a string where all characters are in upper case. Symbols and Numbers are ignored.

Syntax: string.upper()

Example-1:

```
txt = "Hello my friends"
x = txt.upper()
print(x)
```

Output: HELLO MY FRIENDS

31. **rfind():** The rfind() method finds the last occurrence of the specified value. The rfind() method returns -1 if the value is not found. The rfind() method is almost the same as the rindex() method.

Syntax: string.rfind (value, start, end)

Example-1:

```
txt = "Mi casa, su casa."
x = txt.rfind("casa")
print(x)
```

Output: 12

32. **rindex():** The rindex() method finds the last occurrence of the specified value. The rindex() method raises an exception if the value is not found. The rindex() method is almost the same as the rfind() method.

Syntax: string.rindex (value, start, end)

33. **rjust():** The rjust() method will right align the string, using a specified character (space is default) as the fill character.

Syntax: string.rjust (length, character)

34. **rsplit():** The rsplit() method splits a string into a list, starting from the right. If no "max" is specified, this method will return the same as the split() method. When maxsplit is specified, the list will contain the specified number of elements plus one.

Syntax: string.rsplit (separator, maxsplit)

35. **rstrip():**The rstrip() method removes any trailing characters (characters at the end a string), space is the default trailing character to remove.

Syntax: string.rstrip(characters)

36. **splitlines():** The splitlines() method splits a string into a list. The splitting is done at line breaks.

Syntax: string.splitlines (keep line breaks)

Example-1:

```
txt = "Thank you for the music\n Welcome to the jungle"
```

x = txt.splitlines(True)

print(x)

Output: ['Thank you for the music\n', 'Welcome to the jungle']

37. **translate():** The translate() method returns a string where some specified characters are replaced with the character described in a dictionary, or in a mapping table. Use the maketrans() method to create a mapping table. If a character is not specified in the dictionary/table, the character will not be replaced. If you use a dictionary, you must use ASCII codes instead of characters.

Syntax: string.translate(table)

Example-1:

txt = "Hello Sam!";

mytable = txt.maketrans("S", "P");

print(txt.translate(mytable));

Output: Hello Pam!

38. **zfill():** The zfill() method adds zeros (0) at the beginning of the string, until it reaches the specified length. If the value of the len parameter is less than the length of the string, no filling is done.

Syntax: string.zfill (len)

Example-1:

txt = "50"

x = txt.zfill(10)

print(x)

Output: 0000000050

III.4 Lists

List operations: List is used to store the group of values & we can manipulate them, in list the values are stores in index format starts with 0. List is mutable object so we can do the manipulations. A python list is enclosed between square ([]) brackets.

- In list insertion order is preserved it means in which order we inserted element same order output is printed.
- A list can be composed by storing a sequence of different type of values separated by commas.
- The list contains forward indexing & backward indexing.

Syntax: <list_name> = [value1, value2, value3,..., value n]

Example:

```
data1= [1, 2, 3, 4]              # list of integers
data2= ['x','y','z']             # list of String
data3= [12.5, 11.6]              # list of floats
data4= []                        # empty list
data5= ['TEC', 10, 56.4,'a']     # list with mixed data types
```

Accessing List data:

- The list items are accessed by referring to the index number.
- Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc.
- A range of indexes can be specified by specifying where to start and where to end the range. When specifying a range, the return value will be a new list with the specified items.
- Note: The search will start at index 2 (included) and end at index 5 (not included).
- By leaving out the start value, the range will start at the first item.
- By leaving out the end value, the range will go on to the end of the list

Example:

```
x = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
Print(x [1])
Print(x [-1])
```

Output:

Banana

mango

Range of Negative Indexes: Specify negative indexes if you want to start the search from the end of the list.

Example:

```
x = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(x [-4:-1])
```

Output: ['orange', 'kiwi', 'melon']

Check if Item Exists: To determine if a specified item is present in a list use the in keyword.

Example: Check if "apple" is present in the list:

```
fruits= ["apple", "banana", "cherry"]
if "apple" in fruits:
print("Yes, 'apple' is in the fruits list")
```

Output: Yes, 'apple' is in the fruits list

Change Item Value: To change the value of a specific item, refer to the index number.

Example: Change the second item:

```
x = ["apple", "banana", "cherry"]
x[1] = "blackcurrant"
print(x)
```

Output: ['apple', 'blackcurrant', 'cherry']

To insert more than one item, create a list with the new values, and specify the index number where you want the new values to be inserted.

Example: Change the second value by replacing it with two new values:

```
x = ["apple", "banana", "cherry"]
x[1] = ["blackcurrant", "watermelon"]
print(x)
```

Output: ['apple', ['blackcurrant', 'watermelon'], 'cherry']

Change a Range of Item Values: To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values.

Example: Change the values "banana" and "cherry" with the values "blackcurrant" and "watermelon":

```
x = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
x [1:3] = ["blackcurrant", "watermelon"]
print(x)
```

Output: ['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']

III.5 List methods

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

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Note: Python does not have built-in support for Arrays, but Python Lists can be used instead.

1. **List append ():** The append () method appends an element to the end of the list.

Syntax: list.append (element)

Here element (Required): An element of any type (string, number, object etc.)

Example: 1 Add an element to the fruits list.

fruits = ['apple', 'banana', 'cherry']

fruits.append ("orange")

Output: ['apple', 'banana', 'cherry', 'orange']

2. **List clear ():** The clear () method removes all the elements from a list.

Syntax: list.clear ()

Example: Remove all elements from the fruits list.

fruits = ['apple', 'banana', 'cherry', 'orange']

fruits. Clear()

Output: [ ]

3. **List copy ():** The copy () method returns a copy of the specified list.

Syntax: list. Copy()

Example: Copy the fruits list.

fruits = ['apple', 'banana', 'cherry', 'orange']

x = fruits.copy ()

Output: ['apple', 'banana', 'cherry']

4. **List count ():** The count () method returns the number of elements with the specified value.

Syntax: list.count (value)

Here, value is required and it any type (string, number, list, tuple, etc.). The value to search for.

Example-2: Return the number of times the value 9 appears in the list.

points = [1, 4, 2, 9, 7, 8, 9, 3, 1]

x = points. Count(9)

Output: 2

5. **List extend ():** The extend () method adds the specified list elements (or any iterable) to the end of the current list.

Syntax: list.extend (iterable)

Example – 1: Add the elements of cars to the fruits list.

fruits = ['apple', 'banana', 'cherry']

cars = ['Ford', 'BMW', 'Volvo']

fruits.extend (cars)

Output: ['apple', 'banana', 'cherry', 'Ford', 'BMW', 'Volvo']

6. **List index ():** The index () method returns the position at the first occurrence of the specified value.

   Note: The index () method only returns the first occurrence of the value.

Syntax: list. Index (element)

Example – 1: What is the position of the value "cherry".

fruits = ['apple', 'banana', 'cherry']

x = fruits.index ("cherry")

Output: 2

7. **List insert ():** To insert a new list item, without replacing any of the existing values, we can use the insert () method. The insert () method inserts an item at the specified index:

Syntax: list. Insert (position, element)

Here,

Position = A number specifying in which position to insert the value

Element = an element of any type (string, number, object etc.)

Example: Insert "watermelon" as the third item.

x = ["apple", "banana", "cherry"]

x.insert (2, "watermelon")

print(x)

Output: ['apple', 'banana', 'watermelon', 'cherry']

8. **List pop ():** The pop () method removes the element at the specified position. The pop () method returns removed value. Position is a number specifying the position of the element you want to remove, default value is -1, which returns the last item

   Syntax: list.pop (position)

Example - 1: Remove the second element of the fruit list:

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

Output: ['apple', 'cherry']

9. **List remove ():** The remove () method removes the first occurrence of the element with the specified value.

Syntax: list.remove (element)

Example: Remove the "banana" element of the fruit list.

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

Output: ['apple', 'cherry']

10. **List reverse ():** The reverse () method reverses the sorting order of the elements.

Syntax: list.reverse ()

Example: Reverse the order of the list1.

```
List1= ["banana", "Orange", "Kiwi", "cherry"]
List1.reverse()
print(List1)
```

Output: ['cherry', 'Kiwi', 'Orange', 'banana']

Loop Through a List:

- We can loop through the list items by using a for loop.
- Print all items in the list, one by one.
- Use the range() and len() functions to print all items by referring to their index number:

Example – 1:

```
list = ["apple", "banana"]
for x in list:
    print(x)
```

Output:

apple

banana

Using a While Loop: Use the len() function to determine the length of the list, then start at 0and loop your way through the list items by referring to their indexes. Remember to increase the index by 1 after each iteration.

Example – 1:

```
list = ["apple", "banana", "cherry"]
i = 0
while i < len(list):
        print(list[i])
        i = i + 1
```

Output:

apple

banana

cherry

Looping Using List Comprehensive: It offers the shortest syntax for looping through lists.

Example – 1:

```
thislist = ["apple", "banana", "cherry"]
[print x for x in thislist]
```

Output:

apple

banana

cherry

Sort Lists: Sort List Alphanumerically. List objects have a sort () method that will sort the list alphanumerically, ascending, by default. To sort descending, use the keyword argument reverse = True.

Example -1:

```
List1= ["orange", "mango", "kiwi", "pineapple", "banana"]
List1.sort()
print (List1)
```

Output: ['banana', 'kiwi', 'mango', 'orange', 'pineapple']

III.6 Introduction to Tuples in python

Tuple is a collection of objects separated by commas. In some ways, a tuple is similar to a Python list in terms of indexing, nested objects, and repetition but the main difference between both is Python tuple is immutable, unlike the Python list which is mutable.

Tuples in Python are similar to Python lists but not entirely. Tuples are immutable and ordered and allow duplicate values. Some Characteristics of Tuples in Python.

- We can find items in a tuple since finding any item does not make changes in the tuple.
- One cannot add items to a tuple once it is created.
- Tuples cannot be appended or extended.
- We cannot remove items from a tuple once it is created.

Creating Python Tuples: There are various ways by which you can create a tuple in Python. They are as follows:

1. Using round brackets
2. With one item
3. Tuple Constructor

1. **Create Tuples using Round Brackets ():** To create a tuple we will use () operators and a sequence of data enclosed in a parenthesis

   ```
   T = ("one", "Two", "Three")
   print(T)
   ```

   Output: ('one', 'Two', 'Three')

2. Create a Tuple with One Item: Python 3.11 provides us with another way to create a Tuple.

   Values: tuple [int | float, ...] = (1,2,4,1.2,2.3)

   print(values)

   Output: (1, 2, 4, 1.2, 2.3)

3. Tuple Constructor in Python: To create a tuple with a Tuple constructor, we will pass the elements as its parameters.

   Tuple_constructor = tuple(("dsa", "26evelopment", "deep learning"))

   print(tuple_constructor)

   Output: ('dsa', 'developement', 'deep learning')

Accessing Values in Python Tuples: Tuples in Python provide two ways by which we can access the elements of a tuple.

1. Using a positive index
2. Using a negative index

Python Access Tuple using a Positive Index: Using square brackets we can get the values from tuples in Python. The positive index values starts from 0 to N-1

Example:

```
T = ("star", "for", "hero")
print("Value in T[0] = ", T[0])
print("Value in T[2] = ", T[2])
```

Output: Value in T[0] = star

Value in T[2] = hero

Access Tuple using Negative Index: The negative index is used to access all elements of a tuple in reverse order (from last to first)

Example:

```
T = ("star", "for", "hero")
print("Value in T[-1] = ", T[-1])
print("Value in T[-2] = ", T[-2])
```

Output: Value in T[-1] = hero

Value in T[-2] = for

III.7 Operations on Tuples

The different operations related to tuples in Python:

1. Concatenation
2. Nesting
3. Repetition
4. Slicing
5. Deleting
6. Finding the length

1. **Concatenation:** To Concatenation of Python Tuples, we will use plus operators (+).

   Example:

```
# concatenating 2 tuples
tuple1 = (0, 1, 2, 3)
tuple2 = ('python', 'programming')
# Concatenating above two
print(tuple1 + tuple2)
```

   Output: (0, 1, 2, 3, 'python', 'programming')

2. **Nesting:** A nested tuple in Python means a tuple inside another tuple.

   Example:

```
# concatenating 2 tuples
tuple1 = (0, 1, 2, 3)
tuple2 = ('python', 'programming')
# Nesting two tuples
t=(tuple1 , tuple2)
print(t)
```

Output: ((0, 1, 2, 3), ('python', 'programming'))

3. **Repetition:** We can create a tuple of multiple same elements from a single element in that tuple.

   Example:
   ```
   # create a tuples with a repetation
   t = ( 'programming',)*3
   print(t)
   ```

   Output: ('programming', 'programming', 'programming')

4. **Slicing:** Slicing a Python tuple means dividing a tuple into small tuples using the indexing method.

   Syntax: [start-index: end-index]

   Example:
   ```
   # slicing
   tuple1 = (0 ,1, 2, 3)
   print(tuple1[1:])
   print(tuple1[::-1])
   print(tuple1[2:4])
   ```

   Output:
   ```
   (1, 2, 3)
   (3, 2, 1, 0)
   (2, 3)
   ```

5. **Deletion:** In python, we are deleting a tuple using 'del' keyword. The output will be in the form of error because after deleting the tuple, it will give a NameError.

   Note: Remove individual tuple elements is not possible, but we can delete the whole Tuple using Del keyword.

   Syntax: del <name of tuple>

   Example:
   ```
   # Deletion of a tuple
   tuple1 = (0 ,1, 2, 3)
   del tuple1
   ```

6. **Length of Tuple:** To find the length of a tuple, we can use Python's len() function and pass the tuple as the parameter.

Example:
```
# Finding a length of a tuple
tuple1 = (0 ,1, 2, 3)
print(len(tuple1))
```

Output: 4

III.8 Methods in tuples

Python has two built-in methods that you can use on tuples.

1. Count()
2. Index()

1. Count() Method: The count () method of Tuple returns the number of times the given element appears in the tuple.

Syntax: tuple.count (element)

Example:
```
Tuple1 = (0, 1, 2, 3, 2, 3, 1, 3, 2)
res = Tuple1.count(3)
print('Count of 3 in Tuple1 is:', res)
```

Output: Count of 3 in Tuple1 is: 3

2. **Index() Method:** The Index () method returns the first occurrence of the given element from the tuple.

Syntax: tuple.index (element, start, end)

Parameters:

Element: The element to be searched.

Start (Optional): The starting index from where the searching is started

End (Optional): The ending index till where the searching is done

Example:
```
Tuple1 = (0, 1, 2, 3, 2, 3, 1, 3, 2)
res = Tuple1.index(3)
print('First occurrence of 3 is', res)
```

Output: First occurrence of 3 is 3

III.9 Dictionaries

Dictionaries

- In Python, a dictionary can be created by placing a sequence of elements within curly {} braces, separated by 'comma'.
- Dictionaries are used to store data values in key:value pairs.
- A dictionary is a collection which is unordered, it means that the items does not have a defined order, you cannot refer to an item by using an index.
- It is changeable, means that we can change, add or remove items after the dictionary has been created.
- It does not allow duplicates means cannot have two items with the same key.
- Dictionaries are written with curly brackets, and have keys and values
- It can be referred by using the key name.
- The values in dictionary items can be of any data type.

Syntax: dictionary name = {key1 : value1, key2 : value2, ...}

Example-1: Create and print a dictionary.

```
dict1 = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
print(dict1)
```

Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

Different ways to create Python Dictionary: In python dictionaries are created with {} and with a dict() constructor with key-value pairs specified within curly braces and as a list of tuples.

The dict is a predefined constructor in python it accept key-value pairs as list or tuples

Example:

```
Dict = {}
print("Empty Dictionary: ")
print(Dict)
Dict = dict({1: 'hai', 2: 'hello', 3: 'how'})
print("\nDictionary with the use of dict(): ")
print(Dict)

Dict = dict([(1, 'star'), (2, 'For')])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

III.10 Dictionary methods

Python Dictionary Methods: Python has a set of built-in methods that you can use on dictionaries.

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key. with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

1. **Dictionary clear ():** The clear () method removes all the elements from a dictionary.

Syntax: dictionary. Clear()

Example: Remove all elements from the car list.

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
car.clear ()
print(car)
```

Output: {}

2. **Dictionary copy ():** You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be reference to dict1, and changes made in dict1 will automatically also be made in dict2. The copy () method returns a copy of the specified dictionary.

Syntax: dictionary.copy ()

Example: Copy the car dictionary.

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.copy ()
print(x)
```

Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

3. **Dictionary fromkeys ():** The fromkeys () method returns a dictionary with the specified keys and the specified value.

Syntax: dict.fromkeys (keys, value)

keys :An iterable specifying the keys of the new dictionary

Value (Optional): The value for all keys. Default value is None

Example: 1 Create a dictionary with 3 keys, all with the value 0.

```
x = ('key1', 'key2', 'key3')
y = 0
d = dict.fromkeys(x, y)
print(d)
```

Output: ['key1': 0, 'key2': 0, 'key3': 0]

4. **Dictionary get ():** The get () method returns the value of the item with the specified key.

Syntax: dictionary.get (keyname)

Example-1: Get the value of the "model" item.

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.get("model")
print(x)
```

Output: Mustang

5. **Dictionary items ():** The items () method returns a view object. The view object contains the key-value pairs of the dictionary, as tuples in a list. The view object will reflect any changes done to the dictionary.

Syntax: dictionary.items ()

Example-1: Return the dictionary's key-value pairs:

```
car = {
```

```
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.items()
print(x)
```

Output: dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])

6. **Dictionary keys ():** The keys () method returns a view object. The view object contains the keys of the dictionary, as a list. The view object will reflect any changes done to the dictionary.

Syntax: dictionary.keys ()

Example – 1: Return the keys.

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.keys()
print(x)
```

Output: dict_keys(['brand', 'model', 'year'])

7. **Dictionary pop ():** The pop () method removes the specified item from the dictionary. The value of the removed item is the return value of the pop () method.

Syntax: dictionary.pop (keyname, default value)

Example: 1 Remove "model" from the dictionary.

```
car = {
"brand": "Ford",
"model": "Mustang",
```

```
"year": 1964
}
car.pop("model")
print(car) Output: {'brand': 'Ford', 'year': 1964}
```

8. **Dictionary popitem ():** The popitem () method removes the item that was last inserted into the dictionary. In versions before 3.7, the popitem () method removes a random item. The removed item is the return value of the popitem () method, as a tuple.

Syntax: dictionary.popitem ()

Example: 1 Remove the last item from the dictionary.

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
car.popitem ()
print(car)
```

Output: {'brand': 'Ford', 'model': 'Mustang'}

9. **Dictionary setdefault ():** The setdefault () method returns the value of the item with the specified key. If the key does not exist, insert the key, with the specified value.

Syntax: dictionary.setdefault (keyname, value)

Example: 1 Get the value of the "model" item.

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
```

```
x = car.setdefault ("model", "Bronco")
print(x)
```

Output: Mustang

10. **Dictionary update():** The update() method inserts the specified items to the dictionary. The specified items can be a dictionary, or an iterable object with key value pairs.

Syntax: dictionary.update (iterable)

Example: Insert an item to the dictionary.

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
car.update ({"color": "White"})
print(car)
```

Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'White'}

11. **Dictionary values ():** The values () method returns a view object. The view object contains the values of the dictionary, as a list. The view object will reflect any changes done to the dictionary, see example below.

Syntax: dictionary.values ()

Example: 1 Return the values.

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.values ()
```

```
print(x)
```

Output: dict_values(['Ford', 'Mustang', 1964])

12. del: The del keyword removes the item with the specified key name.

Example: 1

```
thisdict = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
del thisdict["model"]
print(thisdict)
```

Output: {'brand': 'Ford', 'year': 1964}

Check if Key Exists: To determine if a specified key is present in a dictionary use the in keyword.

Example: 1 Check if "model" is present in the dictionary.

```
thisdict = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
if "model" in thisdict:
print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

Output: Yes, 'model' is one of the keys in the thisdict dictionary

Dictionary Length: To determine how many items a dictionary has, use the len () function:

Example: Print the number of items in the dictionary.

```
print(len(thisdict))
```

Output: 3

Loop through a Dictionary: You can loop through a dictionary by using a for loop. When looping through a dictionary, the return values are the keys of the dictionary, but there are methods to return the values as well.

Example: 1 Print all key names in the dictionary, one by one.

```
thisdict = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
for x in thisdict:
print(x)
```

Output:

brand
model
year

Nested Dictionaries: A dictionary can contain dictionaries, this is called nested dictionaries.

Example: 1 Create a dictionary that contain three dictionaries.

```
myfamily = {
"child1" : {
"name" : "Emil",
"year" : 2004
},
"child2" : {
"name" : "Tobias",
"year" : 2007
},
```

```
"child3" : {
"name" : "Linus",
"year" : 2011
}
}
```

Output: {'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias', 'year': 2007}, 'child3': {'name': 'Linus', 'year': 2011}}

III.11 Mutable Vs Immutable

- In Python, everything is an object. An object has its own internal state. Some objects allow you to change their internal state and others don't.
- In Python, Every variable in Python holds an instance of an object. There are two types of objects in Python i.e. Mutable and Immutable objects. Whenever an object is instantiated, it is assigned a unique object id.
- The type of the object is defined at the runtime and it can't be changed afterward. However, its state can be changed if it is a mutable object.
- An object whose internal state can be changed is called a mutable object, while an object whose internal state cannot be changed is called an immutable object.

Immutable Objects in Python: Immutable Objects are of in-built datatypes like int, float, bool, string, Unicode, and tuple. In simple words, an immutable object can't be changed after it is created.

- **The following are examples of immutable objects:**
  - Numbers (int, float, bool,…)
  - Strings
  - Tuples
  - Frozen sets

Mutable Objects in Python: Mutable Objects are of type Python list, Python dict, or Python set. Custom classes are generally mutable.

- **And the following are examples of mutable objects:**

  Lists

  Sets

  Dictionaries

User-defined classes can be mutable or immutable, depending on whether their internal state can be changed or not.

Python's Mutable vs Immutable

- Mutable and immutable objects are handled differently in Python. Immutable objects are quicker to access and are expensive to change because it involves the creation of a copy. Whereas mutable objects are easy to change.
- The use of mutable objects is recommended when there is a need to change the size or content of the object.
- Exception: However, there is an exception in immutability as well. We know that a tuple in Python is immutable. But the tuple consists of a sequence of names with unchangeable bindings to objects

III.12 Arrays Vs Lists

Arrays: An array is a collection of items stored at contiguous memory locations. The array The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).

When to Use Python Arrays :

- Lists are built into the Python programming language, whereas arrays aren't. Arrays are not a built-in data structure, and therefore need to be imported via the array module in order to be used.
- Arrays of the array module are a thin wrapper over C arrays, and are useful when you want to work with homogeneous data.
- They are also more compact and take up less memory and space which makes them more size efficient compared to lists.
- If you want to perform mathematical calculations, then you should use NumPy arrays by importing the NumPy package

How to Use Arrays in Python: In order to create Python arrays, you'll first have to import the array module which contains all the necessary functions.

There are three ways you can import the array module:

1. By using import array at the top of the file. This includes the module array. You would then go on to create an array using array.array().

```
import array
#how you would create an array
array.array()
```

2. Instead of having to type array.array() all the time, you could use import array as arr at the top of the file, instead of import array alone. You would then create an array by typing arr.array(). The arr acts as an alias name, with the array constructor then immediately following it.

```
import array as arr
#how you would create an array
arr.array()
```

3. Lastly, you could also use from array import *, with * importing all the functionalities available. You would then create an array by writing the array() constructor alone.

```
from array import *
#how you would create an array
array()
```

How to Define Arrays in Python: Once you've imported the array module, you can then go on to define a Python array. The general syntax for creating an array looks like this:

variable_name = array(typecode,[elements])

- variable_name would be the name of the array.
- The typecode specifies what kind of elements would be stored in the array. Whether it would be an array of integers, an array of floats or an array of any other Python data type. Remember that all elements should be of the same data type.
- Inside square brackets you mention the elements that would be stored in the array, with each element being separated by a comma. You can also create an empty array by just writing variable_name = array(typecode) alone, without any elements.

Below is a typecode table, with the different typecodes that can be used with the different data types when defining Python arrays:

TYPECODE	C TYPE	PYTHON TYPE	SIZE
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	wchar_t	Unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

Example:

```
import array as arr
numbers = arr.array('i',[10,20,30])
print(numbers)
```

output: array('i', [10, 20, 30])

- First we included the array module, in this case with import array as arr .
- Then, we created a numbers array.
- We used arr.array() because of import array as arr .
- Inside the array() constructor, we first included i, for signed integer. Signed integer means that the array can include positive and negative values. Unsigned integer, with H for example, would mean that no negative values are allowed.
- Lastly, we included the values to be stored in the array in square brackets.

How to Find the Length of an Array in Python: To find out the exact number of elements contained in an array, use the built-in len() method.

Example:

```
import array as arr
numbers = arr.array('i',[10,20,30])
print(len(numbers))
```

output: 3

Array Indexing and How to Access Individual Items in an Array in Python

- Each item in an array has a specific address. Individual items are accessed by referencing their index number.Indexing in Python, and in all programming languages and computing in general, starts at 0. It is important to remember that counting starts at 0 and not at 1.
- To access an element, you first write the name of the array followed by square brackets. Inside the square brackets you include the item's index number.The general syntax would look something like this:

Example:

```
array_name[index_value_of_item]
import array as arr
numbers = arr.array('i',[10,20,30])
print(numbers[0]) # gets the 1st element
print(numbers[1]) # gets the 2nd element
print(numbers[2]) # gets the 3rd element
```

output:

```
10
20
30
```

How to Search Through an Array in Python: You can find out an element's index number by using the index() method. You pass the value of the element being searched as the argument to the method, and the element's index number is returned.

Example:

```
import array as arr
numbers = arr.array('i',[10,20,30])
#search for the index of the value 10
print(numbers.index(10))
```

Output: 0

- If there is more than one element with the same value, the index of the first instance of the value will be returned

How to Loop through an Array in Python : You've seen how to access each individual element in an array and print it out on its own. If you want to print each value one by one, This is where a loop comes in handy. You can loop through the array and print out each value, one-by-one, with each loop iteration.

Example:

```
import array as arr
numbers = arr.array('i',[10,20,30])
for number in numbers:
  print(number)
```

Output: 10
20
30

You could also use the range() function, and pass the len() method as its parameter. This would give the same result as above:

Example:

```
import array as arr
values = arr.array('i',[10,20,30])
#prints each individual value in the array
for value in range(len(values)):
  print(values[value])
```

Output: 10
20
30

How to Slice an Array in Python: To access a specific range of values inside the array, use the slicing operator, which is a colon :

- When using the slicing operator and you only include one value, the counting starts from 0 by default. It gets the first item, and goes up to but not including the index number you specify.

Example:

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
```

```
#get the values 10 and 20 only
print(numbers[:2])  #first to second position
```

Output : array('i', [10, 20])

Methods For Performing Operations on Arrays in Python

- Arrays are mutable, which means they are changeable. You can change the value of the different items, add new ones, or remove any you don't want in your program anymore.
- Let's see some of the most commonly used methods which are used for performing operations on arrays.

How to Change the Value of an Item in an Array: You can change the value of a specific element by speficying its position and assigning it a new value:

Example:

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
#change the first element
#change it from having a value of 10 to having a value of 40
numbers[0] = 40
print(numbers)
```

Output: array('i', [40, 20, 30])

How to Add a New Value to an Array: To add one single value at the end of an array, use the append() method:

Example:

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
#add the integer 40 to the end of numbers
numbers.append(40)
```

```
print(numbers)
```

Output: array('i', [10, 20, 30, 40])

Be aware that the new item you add needs to be the same data type as the rest of the items in the array.

Extend method: If you want to add more than one value to the end an array. Use the extend() method, which takes an iterable (such as a list of items) as an argument. Again, make sure that the new items are all the same data type.

Example:

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30])
#add the integers 40,50,60 to the end of numbers
#The numbers need to be enclosed in square brackets
numbers.extend([40,50,60])
print(numbers)
```

Output: array('i', [10, 20, 30, 40, 50, 60])

Insert method: The insert() method, is to add an item at a specific position. The insert() function takes two arguments: the index number of the position the new element will be inserted, and the value of the new element.

Example:

```
#add the integer 40 in the first position remember indexing starts at 0
import array as arr
numbers = arr.array('i',[10,20,30]) #original array
numbers.insert(0,40)
print(numbers)
```

Output: array('i', [40, 10, 20, 30])

How to Remove a Value from an Array : To remove an element from an array, use the remove() method and include the value as an argument to the method.

Example:

```
import array as arr
```

```
#original array
numbers = arr.array('i',[10,20,30])
numbers.remove(10)
print(numbers)
```

Output: array('i', [20, 30])

With remove(), only the first instance of the value you pass as an argument will be removed.

Pop method: You can also use the pop() method, and specify the position of the element to be removed:

Example:

```
import array as arr
#original array
numbers = arr.array('i',[10,20,30,10,20])
#remove the first instance of 10
numbers.pop(0)
print(numbers)
```

Output: array('i', [20, 30, 10, 20])

Difference Between List and Array in Python

List	Array
Can consist of elements belonging to different data types	Only consists of elements belonging to the same data type
No need to explicitly import a module for the declaration	**Need to explicitly import the array module for declaration**
Cannot directly handle arithmetic operations	Can directly handle arithmetic operations
Preferred for a shorter sequence of data items	Preferred for a longer sequence of data items
Greater flexibility allows easy modification (addition, deletion) of data	Less flexibility since addition, and deletion has to be done element-wise
The entire list can be printed without any explicit looping	A loop has to be formed to print or access the components of the array

Consume larger memory for easy addition of elements	Comparatively more compact in memory size
Nested lists can be of variable size	Nested arrays has to be of same size.
Can perform direct operations using functions like: count() , sort()	Need to import proper modules to perform these operations.
Example: my_list = [1, 2, 3, 4]	**Example:** import array arr = array.array('i', [1, 2, 3])

III.13 Map method in python

The map() function in Python: The map() function (which is a built-in function in Python) is used to apply a function to each item in an iterable (like a Python list or dictionary). It returns a new iterable (a map object) that you can use in other parts of your code.

The general syntax: map(function, iterable, [iterable1, iterable2, ...])

Let's see an example: imagine you have a list of numbers, and you want to create a new list with the cubes of the numbers in the first list. A traditional approach would involve using the for loop

The map() function simplifies and reduce the code in trditional aproach :

```
org_list = [1, 2, 3, 4, 5]
# define a function that returns the cube of `num`
def cube(num):
  return num**3
fin_list = list(map(cube, org_list))
print(fin_list) # [1, 8, 27, 64, 125]
```

- Instead of writing a separate function to calculate the cube of a number, we can use a lambda expression in its place. Here's how you'd do that:

```
fin_list = list(map(lambda x:x**3, org_list))
```

```
print(fin_list) # [1, 8, 27, 64, 125]
```

For example if you had a list of strings, you can easily create a new list with the length of each string in the list.

```
org_list = ["Hello", "world", "freecodecamp"]
fin_list = list(map(len, org_list))
print(fin_list) # [5, 5, 12]
```

Note: The map function takes n number of values as input and it returns n values as a output

III.14 Filter method in python

- Python's built-in filter() function is a powerful too; for performing data filtering procedure on sequences like lists, tuples, and strings. The filter() function is utilized to apply a function to each element of an iterable (like a list or tuple) and return another iterable containing just the elements for which the function brings True back. Along these lines, filter() permits us to filter out elements from a grouping based on some condition. The first argument can be None if the function is not available and returns only elements that are True.

Syntax: filter (function, iterable)

Parameters

function: It is a function. If set to None returns only elements that are True.

Iterable: Any iterable sequence like list, tuple, and string.

Example 1: This simple example returns values higher than 5 using filter function

```
# Python filter() function example
def filterdata(x):
  if x>5:
    return x
# Calling function
result = filter(filterdata,(1,2,6))
# Displaying result
```

print(list(result))

using lambda function in filter : Instead of writing a separate function to fin the a number greater than 5, we can use a lambda expression in its place. Here's how you'd do that:

result = filter(lambda x:x>5,(1,2,6))

Displaying result

print(list(result))

Note: The filter function takes n number of values as input and it returns less than or equal to n values as a output

III.15 Reduce method in python

In Python, reduce() is a built-in function that applies a given function to the elements of an iterable, reducing them to a single value.

The syntax for reduce() is as follows:

functools.reduce(function, iterable [, initializer])

- The function argument is a function that takes two arguments and returns a single value. The first argument is the accumulated value, and the second argument is the current value from the iterable.
- The iterable argument is the sequence of values to be reduced.
- The optional initializer argument is used to provide an initial value for the accumulated result. If no initializer is specified, the first element of the iterable is used as the initial value.

Example that demonstrates how to use reduce () to find the sum of a list of numbers:

```
# Examples to understand the reduce() function
from functools import reduce
 # Function that returns the sum of two numbers
def add(a, b):
   return a + b
```

```
# Our Iterable
num_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# add function is passed as the first argument, and num_list is passed as the second argument
sum = reduce(add, num_list)
print(f"Sum of the integers of num_list : {sum}")
# Passing 10 as an initial value
sum = reduce(add, num_list, 10)
print(f"Sum of the integers of num_list with initial value 10 : {sum}")
```

Output:

Sum of the integers of num_list : 55

Sum of the integers of num_list with initial value 10 : 65

Let us use the lambda function as the first argument of the reduce() function

```
# Importing reduce function from functools
from functools import reduce
 # Creating a list
my_list = [1, 2, 3, 4, 5]
# Calculating the product of the numbers in my_list
# using reduce and lambda functions together
product = reduce(lambda x, y: x * y, my_list)
# Printing output
print(f"Product = {product}")
```

Output : Product = 120

Differences between map,reduce and filter in python

Key characteristics	Map	Filter	Reduce
Syntax	map(function, iterable object)	filter(function, iterable object)	reduce(function, iterable object)
Effect of the function on the iterable	Applies the function evenly to all the items in the iterable object	Function is a boolean condition that rejects all the items in the iterable object that are not true	Breaks down the entire process of applying the function into pair-wise operations
Input to output variables	N to N	N to M where N>=M	N to 1
Use Case	Transformation/Mapping	Splitting the data	Single output operations
Example	List of the square of all numbers in a list	All even numbers from a list	Product of all the items in the list

III.15 Comprehensions in python

Comprehensions in Python provide us with a short and concise way to construct new sequences (such as lists, sets, dictionaries, etc.) using previously defined sequences. Python supports the following 4 types of comprehension:

1. List Comprehensions
2. Dictionary Comprehensions
3. Set Comprehensions
4. Generator Comprehensions

1. **List Comprehensions:** List Comprehensions provide an elegant way to create new lists. The following is the basic structure of list comprehension:

Syntax: output_list = [output_exp for var in input_list if (var satisfies this condition)]

Note that list comprehension may or may not contain an if condition. List comprehensions can contain multiple

Example 1: Generating an Even list WITHOUT using List comprehensions

Suppose we want to create an output list that contains only the even numbers which are present in the input list

Example:

```
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]
output_list = []
for var in input_list:
    if var % 2 == 0:
        output_list.append(var)
print("Output List using for loop:", output_list)
```

Output: Output List using for loop: [2, 4, 4, 6]

Example 2: Generating Even list using List comprehensions

Here we use the list comprehensions in Python. It creates a new list named list_using_comp by iterating through each element var in the input_list. Elements are included in the new list only if they satisfy the condition, which checks if the element is even. As a result, the output list will contain all even numbers.

Example:

```
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]
list_using_comp = [var for var in input_list if var % 2 == 0]
print("Output List using list comprehensions:",list_using_comp)
```

output: Output List using list comprehensions: [2, 4, 4, 6]

2. **Dictionary Comprehensions**: Extending the idea of list comprehensions, we can also create a dictionary using dictionary comprehensions. The basic structure of a dictionary comprehension looks like below.

```
output_dict = {key:value for (key, value) in iterable if (key, value satisfy this condition)}
```

Example 1: Generating odd number with their cube values without using dictionary comprehension

```
input_list = [1, 2, 3, 4, 5, 6, 7]
output_dict = {}
for var in input_list:
 if var % 2 != 0:
        output_dict[var] = var**3
print("Output Dictionary using for loop:",output_dict )
```

Example 2: Generating odd number with their cube values with using dictionary comprehension

```
input_list = [1,2,3,4,5,6,7]
dict_using_comp = {var:var ** 3 for var in input_list if var % 2 != 0}
print("Output Dictionary using dictionary comprehensions:",dict_using_comp)
```

Output:

Output Dictionary using dictionary comprehensions: {1: 1, 3: 27, 5: 125, 7: 343}

3. **Set Comprehensions:** Set comprehensions are pretty similar to list comprehensions. The only difference between them is that set comprehensions use curly brackets { }

Example 1 : Checking Even number Without using set comprehension

```
input_list = [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]
output_set = set()
for var in input_list:
 if var % 2 == 0:
        output_set.add(var)
print("Output Set using for loop:", output_set)
```

Output:

Output Set using for loop: {2, 4, 6}

Example 2: Checking Even number using set comprehension

```
input_list = [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]
set_using_comp = {var for var in input_list if var % 2 == 0}
print("Output Set using set comprehensions:",set_using_comp)
```

Output:

Output Set using set comprehensions: {2, 4, 6}

4. **Generator Comprehensions:** Generator Comprehensions are very similar to list comprehensions. One difference between them is that generator comprehensions use circular brackets whereas list comprehensions use square brackets.

   The major difference between them is that generators don't allocate memory for the whole list. Instead, they generate each value one by one which is why they are memory efficient

   Example:

```
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]
output_gen = (var for var in input_list if var % 2 == 0)
print("Output values using generator comprehensions:", end = ' ')
for var in output_gen:
 print(var, end = ' ')
```

   Output:

   Output values using generator comprehensions: 2 4 4 6

All the best

Module-IV: Files, Modules and Packages

Files, Modules and Packages: Files- Persistent, Text Files, Reading and Writing Files, Format Operator, Filename and Paths, Command Line Arguments, File methods, Modules- Creating Modules, Import Statement, Form Import Statement, name spacing, Packages- Introduction to PIP, Installing Packages via PIP(Numpy).

Python provides inbuilt functions for creating, writing, and reading files. There are two types of files that can be handled in python, normal text files and binary files (written in binary language 0s, and 1s).

Text files: In this type of file, each line of text is terminated with a special character called EOL (End of Line), which is the new line character ('\n') in python by default.

Binary files: In this type of file, there is no terminator for a line, and the data is stored after converting it into machine-understandable binary language**.**

File Access Modes: Access modes govern the type of operations possible in the opened file. It refers to how the file will be used once it's opened. These modes also define the location of the File Handle in the file. File handle is like a cursor, which defines from where the data has to be read or written in the file. There are 6 access modes in python.

1. **Read only ('r'):** Open text file for reading. The handle is positioned at the beginning of the file. If the file does not exists, raises the I/O error. This is also the default mode in which a file is opened.
2. **Read and Write ('r+'):** Open the file for reading and writing. The handle is positioned at the beginning of the file. Raises I/O error if the file does not exist.
3. **Write Only ('w'):** Open the file for writing. For the existing files, the data is truncated and over-written. The handle is positioned at the beginning of the file. Creates the file if the file does not exist.
4. **Write and Read ('w+'):** Open the file for reading and writing. For an existing file, data is truncated and over-written. The handle is positioned at the beginning of the file.

5. **Append only ('a'):** Open the file for writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.
6. **Append and Read ('a+'):** Open the file for reading and writing. The file is created if it does not exist. The handle is positioned at the end of the file. The data being written will be inserted at the end, after the existing data.

IV.6 Command Line Arguments

Python Command Line Arguments provides a convenient way to accept some information at the command line while running the program. The arguments that are given after the name of the Python script are known as Command Line Arguments and they are used to pass some information to the program.

For example -

```
python script.py arg1 arg2 arg3
```

Here Python script name is script.py and rest of the three arguments - arg1 arg2 arg3 are command line arguments for the program.

There are following three Python modules which are helpful in parsing and managing the command line arguments:

1. sys module
2. getopt module
3. argparse module

1. **sys module - System-specific parameters:** The Python sys module provides access to any command-line arguments via the sys.argv. This serves two purposes −
    - sys.argv is the list of command-line arguments.
    - len(sys.argv) is the number of command-line arguments.

    Here sys.argv[0] is the program ie. script name.

    Example: Consider the following script test.py −

```
import sys

print("Number of arguments:", len(sys.argv), 'arguments.')

print ("Argument List:", str(sys.argv)
```

- Now run above script as below. All the programs in this tutorial need to be run from the command line, so we are unable to provide online compile & run option for these programs. Kindly try to run these programs at your computer.

```
python test.py arg1 arg2 arg3
```

This produce following result −

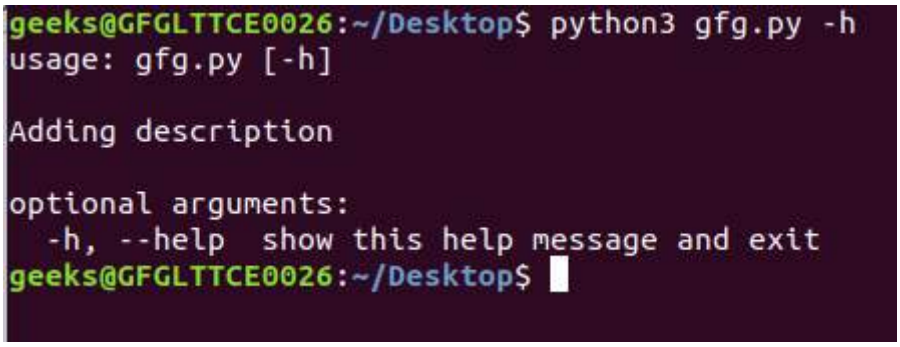
```
Number of arguments: 4 arguments.

Argument List: ['test.py', 'arg1', 'arg2', 'arg3']
```

2. **Getopt module:** Python getopt module is similar to the getopt () function of C. Unlike sys module getopt module extends the separation of the input string by parameter validation. It allows both short, and long options including a value assignment. However, this module requires the use of the sys module to process input data properly. To use getopt module, it is required to remove the first element from the list of command-line arguments.
   Syntax: getopt.getopt (args, options, [long_options])
   Parameters:
   args: List of arguments to be passed.
   Options: String of option letters that the script want to recognize. Options that require an argument should be followed by a colon (:).
   long_options: List of string with the name of long options. Options that require arguments should be followed by an equal sign (=).
   Return Type: Returns value consisting of two elements: the first is a list of (option, value) pairs. The second is the list of program arguments left after the option list was stripped.
3. **Python argparse Module:** argparse module is a better option than the above two options as it provides a lot of options such as positional arguments, default value for arguments, help message, specifying data type of argument etc. Note: As a default optional argument, it includes -h, along with its long version –help.

Example:

```
# Python program to demonstrate
# command line arguments
import argparse
# Initialize parser
parser = argparse.ArgumentParser()
parser.parse_args()
```

Output:



IV.7 File Methods

Python File Methods Python has a set of methods available for the file object.

Method	Description
close()	Closes the file
detach()	Returns the separated raw stream from the buffer
fileno()	Returns a number that represents the stream, from the operating system's perspective
flush()	Flushes the internal buffer
isatty()	Returns whether the file stream is interactive or not
read()	Returns the file content
readable()	Returns whether the file stream can be read or not
readline()	Returns one line from the file
readlines()	Returns a list of lines from the file

seek()	Change the file position
seekable()	Returns whether the file allows us to change the file position
tell()	Returns the current file position
truncate()	Resizes the file to a specified size
writable()	Returns whether the file can be written to or not
write()	Writes the specified string to the file
writelines()	Writes a list of strings to the file

1. **The close () method** closes an open file.

- You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

  Syntax: file.close()

2. **The read() method** returns the specified number of bytes from the file. Default is -1 which means the whole file.

   Syntax: file.read (size)

   Parameter Values: Size is Optional, it represent the number of bytes to return. Default -1, which means the whole file.

3. **The readline () method** returns one line from the file. You can also specified how many bytes from the line to return, by using the size parameter.

   Syntax: file.readline(size)

   Size: Optional. The number of bytes from the line to return. Default -1, which means the whole line.

4. **The readable () method** returns True if the file is readable, False if not.

   Syntax: file.readable()

   Example: f = open ("file.txt", "r")

   print(f.readable())

   Output: True

5. **The readlines () method** returns a list containing each line in the file as a list item. Use the hint parameter to limit the number of lines returned. If the total number of bytes returned exceeds the specified number, no more lines are returned.

Syntax: file.readlines(hint)

6. The seek () method sets the current file position in a file stream. The seek () method also returns the new postion.

   Syntax: file.seek (offset)

   Offset (Required): A number representing the position to set the current file stream position.

   Example:
   ```
   f = open ("file.txt", "r")
    f.seek(4)
   print(f.readline())
   ```

   Output: o! Welcome to First.txt

7. **The writelines() method** writes the items of a list to the file. Where the texts will be inserted depends on the file mode and stream position. "a": The texts will be inserted at the current file stream position, default at the end of the file. "w": The file will be emptied before the texts will be inserted at the current file stream position, default 0.

   Syntax: file.writelines (list)

   List: The list of texts or byte objects that will be inserted.

   Example:

   ```
   f = open ("First.txt", "a")
   f.writelines(["\n See you soon!", "\nOver and out."])
   f.close()
   #open and read the file after the appending:
   f = open ("demofile3.txt", "r")
   print(f.read())
   ```

   Output: Hello! Welcome to First.txt

   This file is for testing purposes.

   Good Luck!

   See you soon!

   Over and out.

8. **The writable () method** returns True if the file is writable, False if not. A file is writable if it is opened using "a" for append or "w" for write.

   Syntax: file.writable()

   Example: f = open ("First.txt", "a")

   print(f.writable())

   Output: True

V.8 Modules- Creating Modules

- A module is a file with the extension .py and contains executable Python or C code. A module contains several Python statements and expressions. We can use pre-defined variables, functions, and classes with the help of modules.
- This saves a lot of developing time and provides reusability of code. Most modules are designed to be concise, unambiguous, and are meant to solve specific problems of developers

Types of Modules in Python: There are two types of modules in Python, they are built-in modules and user-defined modules.

1. Built-in Modules in Python
2. User-defined Modules

1. **Built-in Modules in python:** Modules that are pre-defined are called built-in modules. We don't have to create these modules in order to use them. Built-in modules are mostly written in C language. All built-in modules are available in the Python Standard Library. Few examples of built-in modules are:
   i. math module
   ii. os module
   iii. random module
   iv. datetime module
2. **User-defined Modules in Python:** User-defined modules are modules that we create. These are custom-made modules created specifically to cater to the needs of a certain project**.**

   Creating Modules in python: Creating a module is a simple two-step process. First, we need to write some Python code in a file using any text editor or an IDE. Then, we need to save the file with the extension .py.

We can name the file anything but the extension should always be .py. For example, we can name the file hello.py. We can call this the hello module. Example on creating a module in Python.

1. First, we write some Python code in a file**.**

Code in file hello.py

```
def greet(name=""):
    print(f"Hello! Nice to meet you {name}")
```

2. Second, we save the file with the name hello and extension .py. That's it! We created our module, hello and it is ready to use whenever we want.

V.9 importing modules in python

Importing Modules in Python: Modules are meant to be used with other files. To use a module, we need to first import it. There are two different keywords we can use to import a module. They are "import" and "from".

1. Using Keyword: import: The simplest way to import a module is to use the import keyword followed by the module name.

Syntax: import Module-Name

Example on using import keyword: If we want to import the module hello.py into another file main.py, we use the following code.

Code in file main.py

```
import hello        #Access the module's objects in Python
```

The import keyword creates a reference to the specified module name. Hence, to access its objects, we need to use the module name and a dot operator.

Syntax to access a variable in Python: ModuleName.VariableName

Syntax to call a function in Python: ModuleName.FunctionName()

Example on accessing a module's object using import keyword

Example module: main.py

```
import hello

hello.greet()
```

Output: Hello! Nice to meet you

In the above code example, we called the function greet (), which is inside the hello.py module, from the main.py script.

Importing multiple objects in Python: We can access multiple objects of a module by using the import keyword.

Example on importing multiple objects using import keyword

Example module with (main.py)

```
import math

print(math.log2(2))

print(math.sqrt(16))

print(math.pow(2,4))
```

Output:
```
1.0
4.0
16.0
```

Importing Multiple Modules in Python: We can import multiple modules using a single import keyword.

Syntax: import moduleName1, moduleName2,..,ModuleNameN

Example on importing multiple modules using import keyword

Code in file (main.py)
```
import hello, math, random

hello.greet()

print(math.sqrt(4))

print(random.randint(1, 2)) #generates a random number
```

Output: Hello! Nice to meet you

2.0

2

In the above code example, we used the import keyword to import multiple modules

Pros of using import keyword in Python:

1. It is the simplest and easiest way of importing.
2. With just one line, we can access multiple objects of multiple modules.

Cons of using import keyword in Python:

i. We need to specify the module name every time we want to access an object from the module. This makes the code clunky, lengthy, and time-consuming.

V.9 Using Keyword From in Python

When we need to import only a certain part of the module and not the entire module, we can use the from keyword. This is the most precise way of importing.

Syntax: from Module-Name import Object-Name

Example on using from keyword (Code in file main.py)

```
from hello import greet          #Access the module's objects
```

The keyword from imports the module and creates references to the specified functions. Hence we can directly access them.

Example on accessing a module's object using from keyword (Code in file main.py)

```
from hello import greet

greet()
```

Output: Hello! Nice to meet you

In the above code example, we called the function greet () without using the module name.

Importing Multiple objects in Python: We can import multiple objects of a module by using the from keyword.

Syntax: from Module-Name import ObjectName1,Object-Name2,.., Object-Name-N

Example on importing multiple objects using from keyword (Code in file main.py)

```
from math import sqrt, exp, log2
print(sqrt(4))
print(exp(0))
print(log2(2))
```

Output:
```
2.0
1.0
1.0
```

In the above code example, we called multiple functions without using the module name.

Importing Multiple Modules in Python: Importing multiple modules using a single from keyword is not possible. We need to use multiple from keywords to import multiple modules.

Example on importing multiple modules using from keyword (Code in file main.py)

```
from hello import greet
from math import sqrt
greet()
print(sqrt(4))
```

Output:
```
Hello! Nice to meet you
2.0
```

Pros of using from keyword in Python: Since we have to specify every object name while importing, this is the most unambiguous and precise way of importing. It is the recommended way of importing.

Cons of using from keyword in Python: We need to write multiple from statements to import multiple modules.

- When we need to import a large number of objects from a module, we have to specify all those objects' names in the form statement. This is a time-consuming process.
- from Keyword with Asterisk (*) in Python
- We can also import the entire module using the from keyword. To do this we need to use the symbol asterisk (*). This allows us to import multiple objects without specifying their names in the from import statement.

Syntax: from Module-Name import *

Example on using asterisk (*) with from keyword (Code in file main.py)

```
from math import *      #Access the module's objects
```

This makes python import the specified module and creates references to all the functions and attributes inside the module. We do not need to use the module name to access any of the module's attributes. This is the main difference to the import keyword.

Example on accessing module's objects using * with from keyword (Code in file main.py)

```
from math import *

print(sqrt(4))

print(pow(2, 2))

print(log2(2))
```

Output:

```
2.0

4.0

1.0
```

In the above code example, although we used the from keyword, we called multiple functions without specifying their names in the import statement.

Pros of using * with from keyword in Python: This brings the best of both worlds. Like the import statement, we can import an entire module. Like the from statement, we can access any function without using the module name.

Cons of using * with from keyword in Python: When two modules have an object with the same name, the object from the latest imported module replaces the object from the earlier imported module.

IV.12 Namespace?

- A name or identifier is the name given to objects when we create objects in python. The name is simply the variable name that we use in our programs. In python, we can declare variable names and assign them to the objects as follows

  myInt = 117

  myString = "Python"
- In Python, a way to give each object a unique name is through a namespace. Variables and methods are examples of objects in Python. It is a collection of the known symbolic names and the details about the thing that each name refers to.
- It is used to store the values of variables and other objects in the program, and to associate them with a specific name.
- In Python, there are four types of namespaces which are given below.
  1. Built-in
  2. Global
  3. Enclosing
  4. Local

1. **The Built-in Namespace:** As its name suggests, it contains pre-defined names of all of Python's built-in objects already available in Python. Let's list these names with the following command. Open the Python terminal and type the following command. Print(dir (__builtins__)). The built-in namespace creates by the Python interpreter when its starts up. These are terminated when Python interpreter terminates.
2. The Global Namespace: The global namespace consists of any names in Python at any level of the main program. It is created when the main body executes and remains in

existence until the interpreter terminates. The Python interpreter creates a global namespace for any module that our Python loads with the import statement.

3. The Local Namespaces: The local namespaces are used by the function; when the function is run, the Python interpreter creates a new namespace. The local namespaces continue to exist after the function has finished running.
4. **Enclosing Namespace:** The name space that contains black of code in nested format then we can call it as enclosing name space

   Example: Example -

```
def f():
        print('Initiate f()')
def g():
        print('Initiate g()')
        print('End g()')
        return
g()
        print('Initiate f()')
        return
f()
```

- In the above program, the capability g() is characterized inside the collection of f(). We called the g() function within the f() and the main f() function. Let's look at how the above function works:
- Python creates a new namespace for f() when we call it. Likewise, the f() calls g(), g() gets its own different namespace. The local namespace g() was created for the enclosing namespace, f(). Each of these namespaces is terminated when the function is terminated.

Module-V: Object Oriented Programming, Errors and Exceptions
OOP in Python: Object Oriented Features, Classes, self-variable, Methods, Constructors, Destructors, Inheritance, Overriding Methods, Data hiding, Polymorphism. **Error and Exceptions**: Difference between an error and Exception, Handling Exception, try except block, Raising Exceptions

V.1 Object Oriented Features

- The main features of object oriented programming are as follows:
    1. Classes
    2. Objects
    3. Abstraction
    4. Polymorphism
    5. Inheritance
    6. Encapsulation

1. **Classes:** A class is a template or blue print or a model that consists of the data members or variables and functions and defines the properties and methods for a group of objects. A class is a user-defined data type. It consists of data members and member functions, which can be accessed and used by creating an instance of that class.

   It is a logical entity that has some specific attributes and methods. For example: if you have an employee class, then it should contain an attribute and method, i.e. an email id, name, age, salary, etc.
2. **Object:** It is a basic unit of Object-Oriented Programming and represents the real-life entities. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated. An object has an identity, state, and behavior. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects. Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.
3. **Data Abstraction:** Data abstraction is one of the most essential and important features of object-oriented programming. Data abstraction refers to providing only essential information

about the data to the outside world, hiding the background details or implementation. Data Abstraction in Python can be achieved by creating abstract classes. Advantages of Abstraction

1. It enables code reuse by avoiding code duplication.
2. It enhances software security by making only necessary information available to the users and hiding the complex ones.

Example: A man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car, but he does not know about how on pressing the accelerator the speed is increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

4. **Polymorphism:** The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. For example, A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possess different behavior in different situations. This is called polymorphism. The word polymorphism means to have many forms. So, by using polymorphism, you can add different meanings to a single component. There are two types of polymorphism:
    1. Run-time polymorphism
    2. Compile-time polymorphism
5. **Inheritance:** Inheritance is one of the most important features of object oriented programming. It allows a class to inherit the properties and methods of another class called the parent class, the base class, or the super-class. The class that inherits is called the child class or sub-class.
    i. It provides the reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
    ii. It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Types of Inheritance

b. **Single Inheritance:** Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.

c. **Multilevel Inheritance:** Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.

d. **Hierarchical Inheritance:** Hierarchical-level inheritance enables more than one derived class to inherit properties from a parent class.

e. **Multiple Inheritance**: Multiple-level inheritance enables one derived class to inherit properties from more than one base class.

6. **Encapsulation:** Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. In Encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of their class in which they are declared. As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.

V.2 Classes

A class is a template or blue print or a model that consists of the data members or variables and functions and defines the properties and methods for a group of objects. A class is a user-defined data type. It consists of data members and member functions, which can be accessed and used by creating an instance of that class.

It is a logical entity that has some specific attributes and methods. For example: if you have an employee class, then it should contain an attribute and method, i.e. an email id, name, age, salary, etc.

Define/create Python Class: In Python, a class can be created by using the keyword class, followed by the class name. The syntax to create a class is given below.

Syntax: class <Class Name>:

#Statements..

Example: create a python class with name Bike

```
class Bike:
    name = ""
    gear = 0
```

Here,

Bike - the name of the class

Name/gear - variables inside the class with default values "" and 0 respectively. The variables inside a class are called attributes.

Pass Statement in Python: Empty classes raise an error. To create an empty class without raising an error, we use the pass statement.

Example of Python Pass Statement

```
class Dog:
    pass
print(Dog)
```

output: <class '__main__.Dog'>

Examples of class:

```
class Dog:
    vari1 = "mammal"        #class attributes /instance variables
    vari2 = "dog"
    def fun(self):          # A sample method
        print("I'm a", self.vari1)
        print("I'm a", self.vari2 )
```

Attributes in Python: Variables that are declared inside a class are called Attributes. Attributes are two types: Class Attributes and Instance Attributes.

1. Class Attributes in Python

Class attributes are variables that are declared inside the class and outside methods. These attributes belong to the class itself and are shared by all objects of the class.

Example of Python Class Attributes

```
class Vehicle:
```

```
    wheels = 4
car = Vehicle()
van = Vehicle()
print(car.wheels)
print(van.wheels)
```

Output: 4 4

In the above code example, the attribute wheels is a class attribute. Since it is a class attribute, its value is shared by all objects of the class and in the above code example, the attribute wheels is shared by the two objects, car and van

Class attributes can also be accessed by class name rather than an object name as shown in the below code example.

```
print(Vehicle.wheels)
```

Changing the value of a class attribute using the class name causes the value change in all objects of the class.

2. Instance Attributes in Python: Instance attributes provide a state to an object. They are declared inside the __init__() method. Instance attributes are unique to each object and are not shared by other objects. We can declare instance attributes in the __init__() method by using the dot operator and the self-parameter.

Class Attributes vs Instance Attributes

Class Attributes	Instance Attributes
Shared by all objects of the class.	Not shared by all objects of the class.
Same value for all objects.	A Unique value for each object.
Declared outside the __init__() method.	Declared inside the __init__() method.
Dot operator and self-parameter are not needed.	Dot operator and self-parameter are needed to declare instance attributes.
Can be accessed with the class name and object name.	Can only be accessed with object name.

Object: An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated. An object has an

identity, state, and behavior. Each object contains data and code to manipulate the data. Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

With the help of an object we can access all the attributes or instance variables defined in class or methods in class.

Creating Object in Python: To create an object from the class, we use the following syntax:

Syntax: object_name = ClassName(*args)

Dot Operator in Python: In python, the dot operator is used to access variables and functions which are inside the class.

Syntax: object_name.attribute_name or object_name.method_name ()

Examples-1: Creation of an object for Bike class defined in the above example is

```
Splendor = Bike ()
Splendor.name="Hero"
Slpendor.gear=4
```

Here splendor is the object of class Bike, with the help of object (splendor) we can access or assign values to attributes of Bike class

Example-2: Creation of an object for dog class

```
Pet=dog ()
```

Here pet is object or instance of a dog class, with the help of object (pet) we can access or assign values to attributes and methods of dog class

Example:

```
class Dog:
  vari1 = "animal"                    # attributes or instance variables
  vari2 = "dog"
  def fun(self):                      # method of a class
     print ("I'm a", self.vari1)
     print ("I'm a", self.vari2)
pet = Dog()                           # creating object  to dog class
```

```
print (pet.vari1)
pet.fun()
```

Class variables and Instance Variables: Instance variables are variables whose value is assigned inside a constructor or method with self-whereas class variables are variables whose value is assigned in the class.

V.3 self-variable

When working with classes in Python, The self-parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class

Self: Pointer to Current Object: The self is always pointing to the Current Object. When you create an instance of a class, you're essentially creating an object with its own set of attributes and methods.

Example:

```
class First-self:
        def __init__(self):
        print("Address of self = ",id(self))
obj = First-self()
print("Address of class object = ",id(obj))
```

Self in Constructors and Methods: Self is the first argument to be passed in Constructor and Instance Method. Self must be provided as a First parameter to the Instance method and constructor. If you don't provide it, it will cause an error.

Self is a convention and not a Python keyword. Self is a parameter in Instance Method and the user can use another parameter name in place of it. But it is advisable to use self because it increases the readability of code, and it is also a good programming practice.

Example:

```
class employee:
        def __init__(self, name, age):
                self.name = name
                self.age = age
        def myfunc(star):
```

```
        print ("Hello my name is " + star.name)
e1 = Person("Smith", 21)
e1.myfunc()
```

In the above example, self can be written as star in myfunc method, the first argument of method or constructor must be a self-parameter

V.4 Methods

- Functions inside a class are called Methods. Methods are defined inside classes and are called on instances of that class. They have access to the instance attributes and can modify its state.
- The class constructor or __init__ method is a special method that is called when an object of the class is created. It is used to initialize the instance variables of a class.

An abstract class method is a method that is declared but contains no implementation. It is used as a template for other methods that are defined in a subclass.

A class method is a method that is bound to the class and not the instance of the class. It can be accessed using the class name.

A static method is a method that is bound to the class and not the instance of the class. It does not take any argument like self or cls.

Class methods vs. instance methods: The following table illustrates the differences between class methods and instance methods:

Features	class methods	Instance methods
Binding	Class	An instance of the class
Calling	Class. method()	object. method()
Accessing	Class attributes	Instance & class attributes

Python Abstract Class Method: An abstract class method is a method that is declared in an abstract base class but does not have an implementation. It is used to define a method that must be implemented by any class that inherits from the abstract class. Abstract class methods are created using the @abstract method decorator. Here is an example of an abstract class method in Python:

```
from abc import ABC, abstract-method
class Shape(ABC):
        def __init__(self, type):
                self.type = type

        @abstractmethod
        def area(self):
                pass
class Square(Shape):
        def __init__(self, side):
                super().__init__("Square")
                self.side = side
        def area(self):
                return self.side ** 2
my_square = Square(5)
print(my_square.area())
```

Output: 25

Access specifiers in python: In Python, we can differentiate between public, private, and protected methods based on their access level.

Public methods are those that can be accessed from anywhere within or outside the class.

Private methods in a Python's class are those that can only be accessed from inside the class.

Protected methods are those that can only be accessed from inside the class and its subclasses.

Public Methods: Public methods are accessible from anywhere within or outside the class. They play a significant role in interacting with the class's attributes and functionality. When

developers create a method without any underscore prefix, it automatically becomes a public method.

Example:
```
class MyClass:
    def public_method(self):
        print("This is a public method")
obj = MyClass()
obj.public_method () # Accessing the public method
```

Python Private Methods: Private methods in Python are designed to be accessed only from within the class in which they are defined. They are indicated by prefixing the method name with double underscores _ _

```
class MyClass:
    def __private_method(self):
        print("This is a private method")
obj = MyClass()
obj.__private_method() # Attempting to access the private method (Raises an error)
```

Protected Methods: Protected methods are indicated by prefixing the method name with a single underscore _. They can be accessed from within the class itself and its subclasses.

```
class MyClass:
  def _protected_method(self):
    print("This is a protected method")
class SubClass(MyClass):
  def access_protected(self):
    self._protected_method() # Accessing the protected method from a subclass
obj = SubClass()
obj.access_protected() # Accessing the protected method from the subclass
```

- Protected methods provide a way to allow subclasses to access certain methods while still preventing direct access from external code. However, unlike some other languages, Python doesn't enforce strict visibility restrictions.

V.5 Constructors

Constructors are generally used for instantiating an object. The task of constructors is to initialize (assign values) to the data members of the class when an object of the class is created. In Python the __init__ () method is called the constructor and is always called when an object is created Constructors are special methods that are automatically invoked when an object is created.

Syntax:

```
def __init__(self):
    # Body of the constructor
```

Constructors can be of two types.

1. Parameterized Constructor
2. Non-parameterized Constructor

1. **Default constructor:** The default constructor is a simple constructor which doesn't accept any arguments. Its definition has only one argument which is a reference to the instance being constructed.

Example:

```
class DefaultCons:
    def __init__(self):                # default constructor
        self.msg = "welcome to lab"
    def print_msg(self):               # a method for printing data members
        print (self.msg)
obj = DefaultCons ()                   # creating object of the class
obj.print_msg()      # calling the instance method using the object obj
```

Output: welcome to lab

2. **Parameterized constructor:** constructor with parameters is known as parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

Example:

```
class Student:
    # Constructor - parameterized
    def __init__(self, name):
    print("This is parameterized constructor")
    self.name = "Samantha"
    def show(self):
    print("Hello",self.name)
```

```
s1= Student("John")
s1.show()
```

Output: This is parameterized constructor
Hello Samantha

Advantages of using constructors in Python:

1. **Initialization of objects:** Constructors are used to initialize the objects of a class. They allow you to set default values for attributes or properties, and also allow you to initialize the object with custom data.
2. **Easy to implement:** Constructors are easy to implement in Python, and can be defined using the __init__() method.
3. **Better readability:** Constructors improve the readability of the code by making it clear what values are being initialized and how they are being initialized.
4. **Encapsulation:** Constructors can be used to enforce encapsulation, by ensuring that the object's attributes are initialized correctly and in a controlled manner.

Disadvantages of using constructors in Python:

1. Overloading not supported with constructors
2. Limited functionality with constructors

V.6 Destructors

- A destructor is a special method that is invoked when an object is destroyed or its scope expires. Its primary function is to perform cleansing operations or discharge any resources the object holds.
- Python's garbage collector will eventually designate an object as unreachable and invoke its destructor when it is no longer required.
- Python manages memory using automatic garbage collection. Destructors collaborate with the garbage collector to guarantee memory management efficiency.
- The __del__() method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

Syntax of destructor declaration:

```
def __del__(self):
    # body of destructor
```

- A reference to objects is also deleted when the object goes out of reference or when the program ends.

Example: # Python program to illustrate destructor

```
class Employee:
# Initializing
    def __init__(self):
        print('Employee created.')
    # Deleting (Calling destructor)
    def __del__(self):
        print('Destructor called, Employee deleted.')
obj = Employee()
del obj
```

Output:

```
Employee created.
Destructor called, Employee deleted.
```

Advantages of using destructors in Python:

1. **Automatic cleanup:** Destructors provide automatic cleanup of resources used by an object when it is no longer needed.
2. **Easy to use:** Destructors are easy to implement in Python, and can be defined using the __del__() method.
3. **Helps with debugging:** Destructors can be useful for debugging, as they can be used to trace the lifecycle of an object and determine when it is being destroyed.

V.7 Inheritance

- Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.
- Inheritance allows us to create a new class from an existing class.
- The new class that is created is known as subclass (child or derived class) and the existing class from which the child class is derived is known as superclass (parent or base class).

Python Inheritance Syntax

```
# define a superclass
class super_class:
    # attributes and method definition
    # inheritance
```

```
class sub_class(super_class):
        # attributes and method of super_class
        # attributes and method of sub_class
```

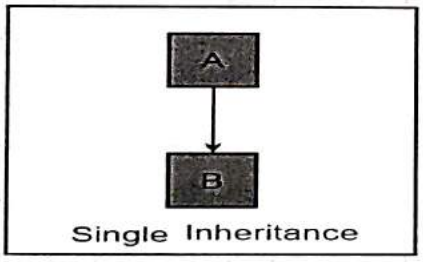
- Here, we are inheriting the sub_class class from the super_class class.

Benefits of inheritance:

1. It represents real-world relationships well.
2. It provides the reusability of a code.
3. It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.
4. Less development and maintenance expenses result from an inheritance

Different types of Python Inheritance: There are 5 different types of inheritance in Python. They are as follows:

1. **Single inheritance:** When a child class inherits from only one parent class, it is called single inheritance. In the following diagram class A is parent class and B is child class, class B inherit the properties from class A



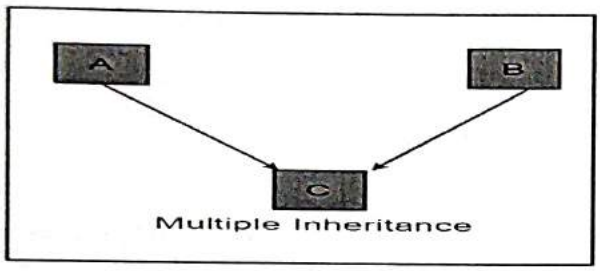
Example:

```
class Animal:
        def speak(self):
                print("Animal Speaking")
                #child class Dog inherits the base class Animal
class Dog(Animal):
        def bark(self):
         print("dog barking")
d = Dog()
d.bark()
d.speak()
```

Output: dog barking

Animal Speaking

2. **Multiple inheritances:** When a child class inherits from multiple parent classes, it is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class.



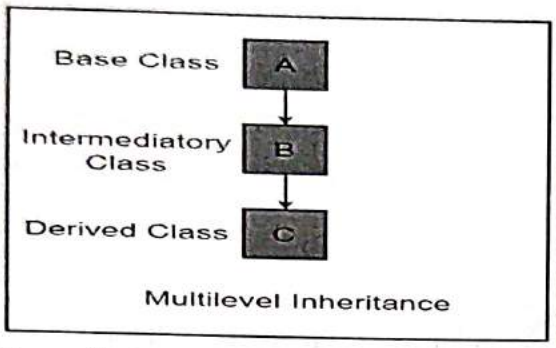
Example:
```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(d.Summation(10,20))
print (d.Multiplication(10, 20))
print (d.Divide(10, 20))
```

Output: 30

200

0.5

3. **Multilevel inheritance:** When we have a child and grandchild relationship. This means that a child class will inherit from its parent class, which in turn is inheriting from its parent class. Here is no limit on the number of levels up to which, the multi-level inheritance is archived in python.



Example:

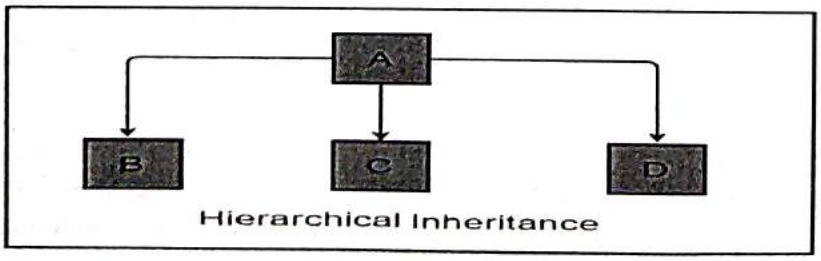
```
class Animal:
        def speak(self):
        print("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
        def bark(self):
        print("dog barking")
#The child class Dog child inherits another child class Dog
class DogChild(Dog):
        def eat(self):
        print("Eating bread...")
d = DogChild()
d.bark()
d.speak()
d.eat()
```

Output: dog barking

Animal Speaking

Eating bread..

4. **Hierarchical inheritance** More than one derived class can be created from a single base. In this program, we have a parent (base) class and two child (derived) classes.



Example:

Python program to demonstrate Hierarchical inheritance

```
# Base class
class Parent:
        def func1(self):
                print("This function is in parent class.")
# Derived class1
class Child1(Parent):
        def func2(self):
                print("This function is in child 1.")
# Derivied class2
class Child2(Parent):
        def func3(self):
                print("This function is in child 2.")
# Driver's code
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

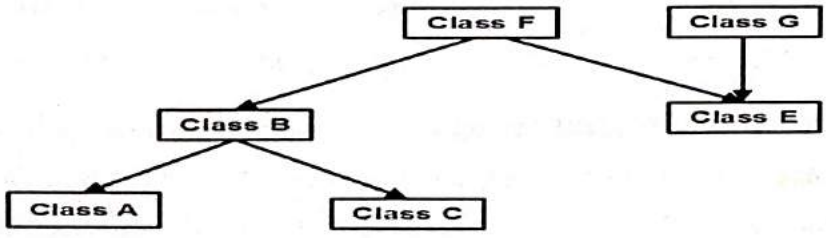
Output: This function is in parent class.

This function is in child 1.

This function is in parent class.

This function is in child 2.

5. **Hybrid inheritance:** This form combines more than one form of inheritance. Basically, it is a blend of more than one type of inheritance.



Example: # Python program to demonstrate hybrid inheritance

```
class School:
        def func1(self):
        print("This function is in school.")
class Student1(School):
        def func2(self):
        print("This function is in student 1. ")
class Student2(School):
        def func3(self):
        print("This function is in student 2.")
class Student3(Student1, School):
        def func4(self):
        print("This function is in student 3.")
# Driver's code
object = Student3()
object.func1()
object.func2()
```

Output: This function is in school.
This function is in student 1.

The super () Function: The super () function is a built-in function that returns the objects that represent the parent class. It allows to access the parent class's methods and attributes in the child class.

```
# parent class
class Person():
        def __init__(self, name, age):
                self.name = name
                self.age = age
        def display(self):
                print(self.name, self.age)
# child class
class Student(Person):
```

```
        def __init__(self, name, age):
            self.sName = name
            self.sAge = age
            # inheriting the properties of parent class
            super().__init__("samantha", age)
        def displayInfo(self):
            print(self.sName, self.sAge)
obj = Student("python", 23)
obj.display()
obj.displayInfo ()
```

Output: Rahul 23
Mayank 23

V.8 Overriding Methods

- Method Overriding in Python is a OOPs concept closely related to inheritance. When a child class method overrides the parent class method of the same name, parameters and return type, it is known as method overriding.
- In this case, the child class's method is called the overriding method and the parent class's method is called the overridden method.
- Method overriding is completely different from the concept of method overloading. Method overloading occurs when there are two functions with the same name but different parameters. And, method overloading is not directly supported in Python.

Parent class: The class being inherited is called the Parent or Superclass.

Child class: The class that inherits the properties and methods of the parent class is called the Child or Subclass.

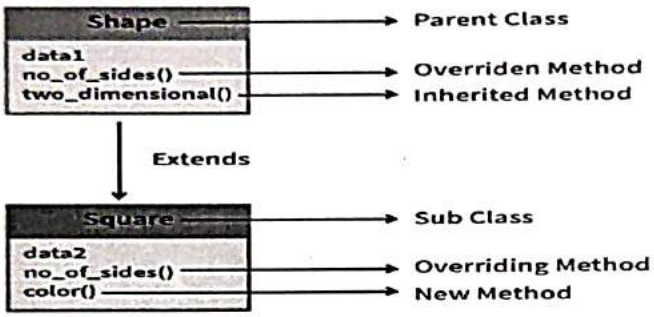
Key features of Method Overriding in Python

These are some of the key features and advantages of method overriding in Python

- Method Overriding is derived from the concept of object oriented programming
- Method Overriding allows us to change the implementation of a function in the child class which is defined in the parent class.

- Method Overriding is a part of the inheritance mechanism
- Method Overriding avoids duplication of code
- Method Overriding also enhances the code adding some additional properties.

Example of Method Overriding in Python:



Example:

```
# Parent class

class Shape:

 # properties

        data1 = "abc"

        # function no_of_sides

        def no_of_sides(self):

            print("My sides need to be defined. I am from shape class.")

         # function two_dimensional

        def two_dimensional(self):

                print("I am a 2D object. I am from shape class")

#Sub-class

class Square(Shape):

        data2 = "xyz"

        def no_of_sides(self):
```

```
                print("I have 4 sides. I am from Square class")
        def color(self):
                print("I have teal color. I am from Square class.")
    # Create an object of Square class
    sq = Square()
    # Override the no_of_sides of parent class
    sq.no_of_sides()
    # Will inherit this method from the parent class
    sq.two_dimensional()
    # It's own method - color
    sq.color()
    print ("Old value of data1 = ", sq.data1)
    # Override property of the Parent class
    sq.data1 = "New value"
print("The value of data1 in Shape class overridden by the Square class = ", sq.data1)
```

Output:

I have 4 sides. I am from Square class

I am a 2D object. I am from shape class

I have teal color. I am from Square class.

Old value of data1 = abs

The value of data1 in Shape class overridden by the Square class = New value

V.9 Data Hiding

- Data hiding is also known as data encapsulation and it is the process of hiding the implementation of specific parts of the application from the user. Data hiding combines members of class thereby restricting direct access to the members of the class.
- Data hiding plays a major role in making an application secure and more robust
- Data hiding in python is a technique of preventing methods and variables of a class from being accessed directly outside of the class in which the methods and variables are initialized. Data hiding of essential member function prevents the end user from viewing the implementation of the program hence increasing security.
- The use of data hiding also helps in reducing the complexity of the program by reducing interdependencies.
- Data hiding in python can be achieved by declaring class members as private by putting a double underscore (_ _) as prefix before the member name.

Syntax: The syntax for hiding data in python

```
__variable-name
```

Example: In this example, data hiding is performed by declaring the variable in the class as private _ _

```
class hidden:
# declaring private member of class
    __hidden-private = 0
    def sum(self, counter):
        self.__hidden-private+= counter
        print (self.__hidden-private)
hidden-obj = hidden()
hidden-obj.sum(5)
hidden-obj.sum(10)
```

```
# print statement throws error as __hidden-private is private
print(hidden-obj.__hidden-private)
```

Output: 5
15
Traceback (most recent call last):
File "main.py", line 12, in <module>
print(hiddenobj.__hidden-private)
AttributeError: 'hidden' object has no attribute '__hidden-private'

Example:

```
class Solution:
    __private-Counter = 0

    def sum(self):
        self.__private-Counter += 1
        print(self.__private-Counter)
count = Solution ()
count.sum()
count.sum()
# Here it will show error because it unable to access private member
print(count.__private-Count)
```

Output:

Traceback (most recent call last):
File "/home/db01b918da68a3747044d44675be8872.py", line 11, in <module>
print(count.__private-Count)
AttributeError: 'Solution' object has no attribute '__private-Count'

- To rectify the error, we can access the private member through the class name
- In the above program error can be resolved, we have accessed the private data member through class name.

```
print (count._Solution__private-Counter)
```

V.10 Polymorphism

- The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function
- Polymorphism is the principle that one kind of thing can take a variety of forms. In the context of programming, this means that a single entity in a programming language can behave in multiple ways depending 0on the context.

Operator Polymorphism (overloading): Operator polymorphism, or operator overloading, means that one symbol can be used to perform multiple operations. One of the simplest examples of this is the addition operator +. Python's addition operator works differently in relation to different data types. For example, if the + operates on two integers, the result is additive, returning the sum of the two integers.

```
int1 = 10
int2 = 15
print(int1 + int2)
```

Output: returns 25

In the example above, the + operator adds 10 and 15, such that the output is 25. However, if the addition operator is used on two strings, it concatenates the strings. Here's an example of how the + operator acts on string data types

```
str1 = "10"
str2 = "15"
print(str1 + str2)
```

Output: returns 1015

In this case, the output is 1015 because the + operator concatenates the strings "10" and "15". This is one example of how a single operator can perform distinct operations in different contexts.

Function Polymorphism (Method Overloading): Method overloading means a class containing multiple methods with the same name but may have different arguments. Basically python does not support method overloading, but there are several ways to achieve method overloading. Though method overloading can be achieved, the last defined methods can only be usable.

Certain functions in Python are polymorphic as well, meaning that they can act on multiple data types and structures to yield different kinds of information.

Python's built-in len () function, for instance, can be used to return the length of an object. However, it will measure the length of the object differently depending on the object's data type and structure.

For instance, if the object is a string, the len () function will return the number of characters in the string. If the object is a list, it will return the number of entries in the list. Here's an example of how the len () function acts on strings and lists:

Example:

```
str1 = "animal"
print(len(str1))
# returns 6
list1 = ["giraffe","lion","bear","dog"]
print(len(list1))
# returns 4
```

The len () function counts the number of characters in a string and the number of entries in a list. The function operates differently depending on the nature of the data it's acting on.

V.11 Difference between an error and Exception

Errors are the problems in a program due to which the program will stop the execution.

Exceptions An exception is an event that occurs during the execution of programs that disrupt the normal flow of execution (e.g., Key Error Raised when a key is not found in a dictionary.)

An exception is a Python object that represents an error. Two types of Error occurs in python.

1. Syntax errors
2. Logical errors (Exceptions)

The syntax error occurs when we are not following the proper structure or syntax of the language. A syntax error is also known as a parsing error. When Python parses the program and finds an incorrect statement it is known as a syntax error. When the parser found a syntax error it exits with an error message without running anything.

Common Python Syntax errors:

Incorrect indentation

Missing colon, comma, or brackets

Putting keywords in the wrong place.

Logical errors (Exception): Even if a statement or expression is syntactically correct, the error that occurs at the runtime is known as a Logical error or Exception. In other words, Errors detected during execution are called exceptions.

Common Python Logical errors:

Indenting a block to the wrong level

using the wrong variable name

making a mistake in a boolean expression

Python automatically generates many exceptions and errors. Runtime exceptions, generally a result of programming errors, such as:

- Reading a file that is not present
- Trying to read data outside the available index of a list
- Dividing an integer value by zero

Error vs. Exceptions

Error	Exceptions
All errors in Python are the unchecked type	Exceptions include both checked and unchecked type.
Errors occur at run time which unknown to the compiler.	Exceptions can be recover by handling them with the help of try-catch blocks.
Errors are mostly caused by the environment in which an application is running.	The application itself causes exceptions.
Examples:	Examples:
OutofMemoryError	Checked Exceptions, SQL exception, NullPointerException,etc.

Built-in Exceptions: The below table shows different built-in exceptions.

Python Built-in Exceptions

Exception	Description

AssertionError	Raised when an assert statement fails.
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when the input() function hits the end-of-file condition.
FloatingPointError	Raised when a floating-point operation fails.
GeneratorExit	Raise when a generator's close() method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when the index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits the interrupt key (Ctrl+C or Delete)
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in the local or global scope.
OSError	Raised when system operation causes system related error.
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.

V.12 Exception Handling in Python

An exception is an error which happens at the time of execution of a program. However, while running a program, Python generates an exception that should be handled to avoid your program to crash

- An exception is an error which happens at the time of execution of a program. However, while running a program, Python generates an exception that should be handled to avoid your program to crash
- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions during the execution of a program
- Exception handling is managed by the following 5 keywords:
    1. Try
    2. except
    3. Finally
    4. Else

Try and Except Statement – Catching Exceptions

- Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause. Every try block is followed by an except block.
- When an exception occurs, it is caught by the except block. The except block cannot be used without the try block.

```
try:
    # code that may cause exception
except:
    # code to run when exception occurs
```

Example: Exception Handling Using try...except

```
try:
    numerator = 10
    denominator = 0
    result = numerator/denominator
    print(result)
except:
    print("Error: Denominator cannot be 0.")
```

Output: Error: Denominator cannot be 0.

In the example, we are trying to divide a number by 0. Here, this code generates an exception. To handle the exception, we have put the code, result = numerator/denominator inside the try block. Now when an exception occurs, the rest of the code inside the try block is skipped. The except block catches the exception and statements inside the except block are executed

Catching Specific Exceptions in Python: For each try block, there can be zero or more except blocks. Multiple except blocks allow us to handle each exception differently.

The argument type of each except block indicates the type of exception that can be handled by it.

For example:

```
try:
   even_numbers = [2,4,6,8]
   print(even_numbers[5])
except ZeroDivisionError:
   print("Denominator cannot be 0.")
Except IndexError:
   print("Index Out of Bound.")
```

Output: Index Out of Bound

In the above example, we have created a list named even_numbers. Since the list index starts from 0, the last element of the list is at index 3. Notice the statement, Here, we are trying to access a value to the index 5. Hence, IndexError exception occurs. When the IndexError exception occurs in the try block, The ZeroDivisionError exception is skipped. The set of code inside the IndexError exception is executed.

Python try with else clause: In some situations, we might want to run a certain block of code if the code block inside try runs without any errors. For these cases, you can use the optional else keyword with the try statement.

Let's look at an example:

```
# program to print the reciprocal of even numbers
try:
   num = int(input("Enter a number: "))
   assert num % 2 == 0
except:
   print("Not an even number!")
else:
   reciprocal = 1/num
   print(reciprocal)
```

Output: If we pass an odd number:

```
Enter a number: 1
```

Not an even number!

If we pass an even number, the reciprocal is computed and displayed.

Enter a number: 4

0.25

However, if we pass 0, we get ZeroDivisionError as the code block inside else is not handled by preceding except.

Enter a number: 0

Traceback (most recent call last):

File "<string>", line 7, in <module>

reciprocal = 1/num

ZeroDivisionError: division by zero

Python try...finally: In Python, the finally block is always executed no matter whether there is an exception or not. The finally block is optional. And, for each try block, there can be only one finally block.

Example:

```
try:
   numerator = 10
   denominator = 0
   result = numerator/denominator
   print(result)
except:
   print("Error: Denominator cannot be 0.")
finally:
   print("This is finally block.")
```

Output

Error: Denominator cannot be 0.

This is finally block.

V.13 Python Raise an Exception

The raise statement in Python is used to raise an exception. Try-except blocks can be used to manage exceptions, which are errors that happen while a programme is running. When an exception is triggered, the programme goes to the closest exception handler, interrupting the regular flow of execution.

- The raise keyword is typically used inside a function or method, and is used to indicate an error condition.
- We can throw an exception and immediately halt the running of your programme by using the raise keyword.
- Python looks for the closest exception handler, which is often defined using a try-except block, when an exception is triggered.
- If an exception handler is discovered, its code is performed, and the try-except block's starting point is reached again.
- If an exception handler cannot be located, the software crashes and an error message appears.

Example:

Example-1:

```
# Python program to raise an exception divide function
def divide(a, b):
  if b == 0:
   raise Exception(" Cannot divide by zero. ")
 return a / b
try:
        result = divide(10, 0)
except Exception as e:
        print(" An error occurred: ", e)
```

Output: An error occurred: Cannot divide by zero.

ALL THE BEST