SEMESTER: IV

PYTHON PROGRAMMING

UNIT-1

Introduction to Features and Applications of Python; Python Versions; Installation of Python; Python Command Line mode and Python IDEs;Simple Python Program.

Python Basics: Identifiers; Keywords; Statements and Expressions; Variables; Operators; Precedence and Association; Data Types; Indentation; Comments; Built-in Functions- Console Input and Console Output, Type Conversions; Python Libraries; Importing Libraries with Examples.

Python Control Flow: Types of Control Flow; Control Flow Statements- if, else, elif, while loop, break, continue statements, for loop Statement; range() and exit () functions.

Exception Handling: Types of Errors; Exceptions; Exception Handling using try, except and finally.

Python Functions: Types of Functions; Function Definition- Syntax, Function Calling, Passing Parameters/arguments, the return statement; Default Parameters; Command line Arguments; Key Word Arguments; Recursive Functions; Scope and Lifetime of Variables in Functions.

Chapter - 01

Introduction to Python:

Python is a general purpose of the programming language and it is a high level programming language. Python is dynamic language and similar to the English language. It is an open source language.

Python is an interpreter –based programming language and one of the trending programming language in the technical field.

Python is widely used programming language with favourite programming language for the all the age groups. It's is object oriented language and support to develop to the application.

History of python

Python is invented Guido van Rossum in the year of 1980's and first version is released in the year of February 20 1991.

The implementation of Python was started in December 1989 by Guido Van Rossum at CWI in Netherland.

In February 1991, Guido Van Rossum submitted the code (marked version 0.9.0) to alt.sources.

The ABC programming language is said to be the predecessor of Python since it could handle errors and communicate with the Amoeba Operating System.

Python is influenced by the programming languages listed below:

1. The ABC language.
2. Modula-3

Why the Name Python?

There is a fact behind choosing the name Python. **Guido van Rossum** was watching a popular BBC comedy series and is fan of BBC Comedy show"**Monty Python's Flying Circus**". It was late on 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.

Introduction to Features and Applications of Python

Python provides many useful features which make it popular and valuable from the other programming languages. It supports object-oriented programming, procedural programming approaches and provides dynamic memory allocation. The following are the primary factors to use python in day-to-day life:

We have listed below a few essential features or why learn python

1. Easy to Learn and Use
2. Expressive Language
3. Interpreted Language
4. Cross-platform Language
5. Free and Open Source
6. Object-Oriented Language
7. Extensible
8. Large Standard Library
9. GUI Programming Support
10. Integrated
11. Embeddable

12. Dynamic Memory Allocation
13. Indentation
14. It's free (open source)
15. It's Powerful
16. It's Portable
17. Interactive Programming Language
18. Straight forward syntax

1)Easy to Learn and Use

Python is easy to learn as compared to other programming languages. Its syntax is straightforward and much the same as the English language. There is no use of the semicolon or curly-bracket, the indentation defines the code block. It is the recommended programming language for beginners.

2) Expressive Language

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

3) Interpreted Language

Python is an interpreted language; it means the Python program is executed one line at a time. The advantage of being interpreted language, it makes debugging easy and portable.

4) Cross-platform Language

Python can run equally on different platforms such as Windows, Linux, UNIX, and Macintosh, etc. So, we can say that Python is a portable language. It enables programmers to develop the software for several competing platforms by writing a program only once.

5) Free and Open Source

Python is freely available for everyone. It is freely available on its official website www.python.org. It has a large community across the world that is dedicatedly working towards make new python modules and functions. Anyone can contribute to the Python community. The open-source means, "Anyone can download its source code without paying any penny."

6) Object-Oriented Language

Python supports object-oriented language and concepts of classes and objects come into existence. It supports inheritance, polymorphism, and encapsulation, etc. The object-oriented procedure helps to programmer to write reusable code and develop applications in less code.

7) Extensible

It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our Python code. It converts the program into byte code, and any platform can use that byte code.

8) Large Standard Library

It provides a vast range of libraries for the various fields such as machine learning, web developer, and also for the scripting. There are various machine learning libraries, such as Tensor flow, Pandas, Numpy, Keras, and Pytorch, etc. Django, flask, pyramids are the popular framework for Python web development.

9) GUI Programming Support

Graphical User Interface is used for the developing Desktop application. PyQT5, Tkinter, Kivy are the libraries which are used for developing the web application.

10) Integrated

It can be easily integrated with languages like C, C++, and JAVA, etc. Python runs code line by line like C,C++ Java. It makes easy to debug the code.

11. Embeddable

The code of the other programming language can use in the Python source code. We can use Python source code in another programming language as well. It can embed other language into our code.

12. Dynamic Memory Allocation

In Python, we don't need to specify the data-type of the variable. When we assign some value to the variable, it automatically allocates the memory to the variable at run time. Suppose we are assigned integer value 15 to **x,** then we don't need to write **int x = 15.** Just write x = 15.

13.Indentation

Indentation is one of the greatest feature in python

14. It's free (open source)

Downloading python and installing python is free and easy

15. It's Powerful

- Dynamic typing
- Built-in types and tools
- Library utilities
- Third party utilities (e.g. Numeric, NumPy, sciPy)
- Automatic memory management

16. It's Portable

- Python runs virtually every major platform used today
- As long as you have a compatible python interpreter installed, python

17.Interactive Programming Language

Users can interact with the python interpreter directly for writing the programs

18. Straight forward syntax

The formation of python syntax is simple and straight forward which also makes it

Tools and Frameworks of Python

The following lists important tools and frameworks to develop different types of python applications:

1. **Web development**: Django,Pyramid,Bottle,Tornado,Flask,web2py
2. **GUI Development:** tkinter, pyGoobject,PyQt,pyside,Kivy, wxpython
3. **Scientific and Numeric:**Scipy,Pandas,IPython
4. **Software Development**:Buildbot,trac,roundup
5. **System Administration:** Ansible,Salt,OpenStack

Python advantages:

1. ***Simple to Use and Understand***

Python is straightforward to learn and use for newbies. It's a well developed programming language with an English-like grammar. As a result of these qualities, the language is simple to adapt. Python's basics can be implemented faster than those in other programming languages due to their simplicity.

2. ***Open-Source and Free***

Python is offered under an open-source licence recognised by the Open-Source Initiative (OSI). As a consequence, people may build on it and distribute it. Users can obtain the source code, alter it, and even share their own Python version. Companies who want to change a certain behaviour and produce their own version will profit.

3. ***Productivity has improved.***

The Python programming language enables users to construct new types of applications. Because of its adaptability, this language allows the operator to attempt new things. The language does not hinder the user from attempting anything new. Python is preferred in certain cases because other programming languages lack the flexibility and freedom that Python provides.

4. ***Interpreted Language***

It is an interpreted language, which means that the code is executed line by line. This is one of the aspects that makes it simple to use. In the case of an error, it halts the process and reports the

issue. Even if the programme has several faults, Python only displays one error. This makes debugging easy.

5. ***Large library***

Python comes with a plethora of libraries that the user may access. Python's standard library is massive, including practically every function possible. This has been aided by large and supportive communities, as well as corporate backing. Users do not need to utilise any extra libraries when working with Python.

6. ***typed dynamically***

Python has no notion what types of parameters we're talking about until we execute the programme. During execution, it allocates the data type automatically. The programmer is not required to specify variables and their data types.

7. ***Portability***

Many other programming languages, including C/C++, need users to modify their code in order for it to operate on other platforms. Python, on the other hand, is not interchangeable with any other programming language. It simply has to be written once and may then be run elsewhere. The user should, however, avoid using any system-dependent functionality.

8. ***Community that is supportive***

Python is a programming language that was developed many years ago and has a vast community that can help programmers of all levels of expertise, from beginners to experts. Python's community has contributed to its rapid growth in compared to other languages. Python comes with a plethora of tutorials, instructional videos, and highly intelligible documentation to assist developers in learning the language faster and more successfully.

Applications of Python or where to used python

Python is well-known for its general-purpose character, which makes it usable in practically every sector of software development. Python is present in every new sector. It is the fastest-growing programming language and may be used to create any application.

1.Data Science

2.Data Mining

3.Desktop Application

4.Console-Based Application

5.Mobile Application

6.Software Development

7.Artificial intelligence(AI)

8.Web Application/Development (Server side)

9.Enterprise Application

10.Machine learning

11.Computer Vision or Image processing Applications

12.Speech Recognition

13.System Scripting

14.Mathematics

1.Data Science: Data is the new Oil. This statement shows how every modern IT system is driven by capturing, storing and analysing data for various needs. Be it about making decision for business, forecasting weather, studying protein structures in biology or designing a marketing campaign. All of these scenarios involve a multidisciplinary approach of using mathematical models, statistics, graphs, databases and of course the business or scientific logic behind the data analysis.

2.Data Mining: Many programming languages can perform data analysis, and the best language depends on your needs and your use case. For many, Python is considered the best choice for analyzing data. Python can quickly create and manage data structures, allowing you to analyze and manipulate complex data sets.

3.Desktop Application: Python is frequently used for creating desktop applications and GUI (graphical user interface) apps. Python is great for backend web development, and is famous for being simple yet powerful.

4.Console-Based Application: Python console enables executing Python commands and scripts line by line, similar to your experience with Python.

5.Mobile Application: The versatility, simplicity, and high speed of writing Python code allow you to use it in web development, desktop programs, system administration, Data Science, embedded systems, mobile applications, computer games, and a variety of plug-ins and scripts.

6.Software Development: Python is often used as a support language for software developers, for build control and management, testing, and in many other ways. SCons for build control. Buildbot and Apache Gump for automated continuous compilation and testing. Roundup or Trac for bug tracking and project management.

7.Artificial intelligence(AI)

Python is commonly used to develop AI applications, such as improving human to computer interactions, identifying trends, and making predictions. One way that Python is used for human to computer interactions is through chatbots.

8.Web Application/Development (Server side)

Python's role in web development can include sending data to and from servers, processing data and communicating with databases, URL routing, and ensuring security. Python offers several frameworks for web development. Commonly used ones include Django and Flask.

9.Enterprise Application: Using Python, developers can build high-quality enterprise applications to support all major platforms including Android, macOS, Windows, and Linux. There are several Python-based packages like PyInstaller that enable developers to make their code executable across different platforms.

10.Machine learning:Python includes a modular machine learning library known as PyBrain, which provides easy-to-use algorithms for use in machine learning tasks. The best and most

reliable coding solutions require a proper structure and tested environment, which is available in the Python frameworks and libraries.

11.Computer Vision or Image processing Applications

OpenCV. OpenCV is the most popular library for computer vision. Originally written in C/C++, it also provides bindings for Python. Free Bonus: Click here to get the Python Face Detection & OpenCV Examples Mini-Guide that shows you practical code examples of real-world Python computer vision techniques.

12.Speech Recognition : It allows computers to understand human language. Speech recognition is a machine's ability to listen to spoken words and identify them. You can then use speech recognition in Python to convert the spoken words into text, make a query or give a reply. You can even program some devices to respond to these spoken words.

13.System Scripting: Python is an interpreted object-oriented programming language, and PY files are program files or scripts created in Python. Text editors can be used to create and modify it, but a Python interpreter is required to execute it. Web servers, as well as other admin computer systems, are frequently programmed with PY files.

14.Mathematics : The Python math module provides functions that are useful in number theory as well as in representation theory, a related field. These functions allow you to calculate a range of important values, including the following: The factorials of a number. The greatest common divisor of two numbers.

Python Versions:

Python software foundation(PSF)used to support two major versions, python 2.x & python 3.x PSF supported python 2 because a large body of existing code could not be forward ported to python3.So,they supported python 2 until January 202,but now they have stopped supporting it.

Python 3.0 was released on December 3rd,2008.It was designed to rectify certain flaws in earlier version. This version is not completely backward-compatible with previous versions.However,many of its major features have since been back-ported to the python 2.6.x and 2.7.x version series. Release of python 3 include utilities to facilitate the automation of python 2 code translation to python 3.

The following table lists all the important versions of python:

Versions	Release date	Important features
Python 0.9.0	February 1991	1.classes with inheritance exception handling. 2. Functions 3.Modules
Python 1.0	January 1994	1.Functional programming tools(lmbda,map,filter and reduce) 2.support for complex numbers. 3.Functions with keyword arguments
Python 2.0	October 2000	1.List comprehension 2.Cycle-detecting garbage collector. 3.Support for Unicode. Unification of data types and classes

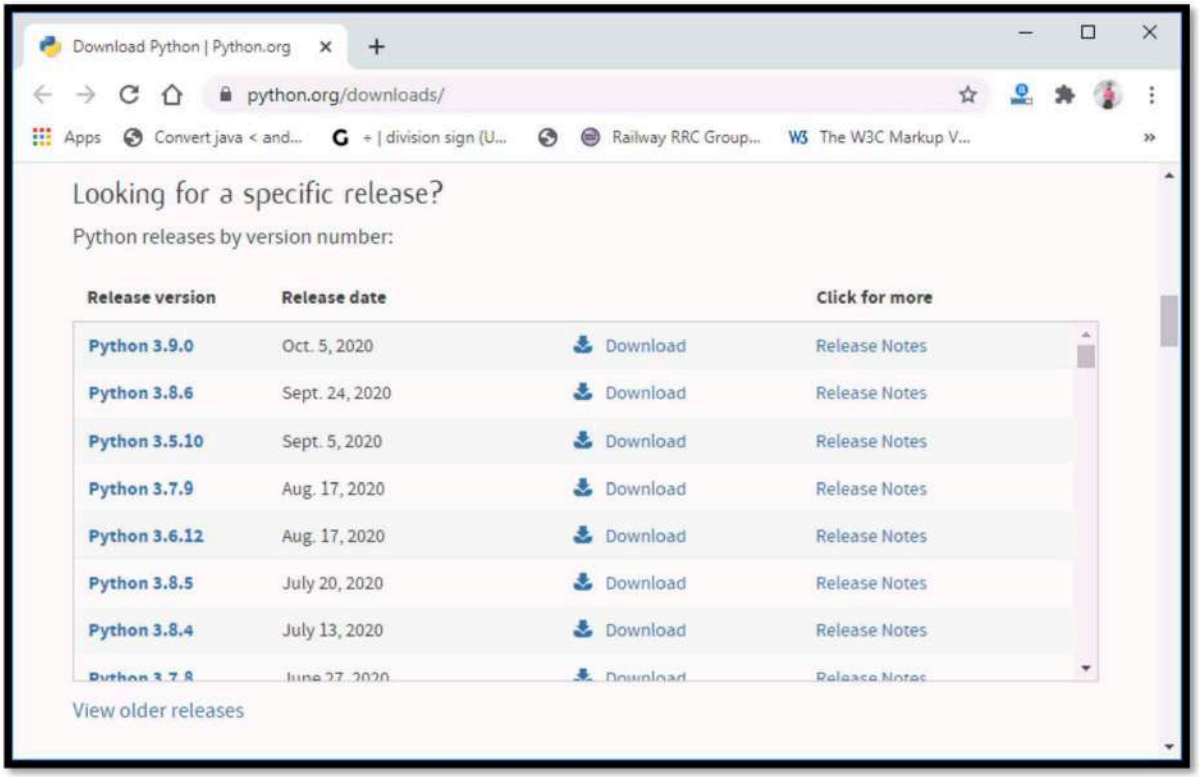
Python 2.7.0 – EOL-Jan 2020	December 2008	1.Backward incompatible 2.print keyword changed to print()function 3.raw_input() function depreciated
Python 3	December 2008	1.Unified str/Unicode types 2.Utilities for automatic conversion of python 2.x code
Python 3.6	December 2016	1.Unified str/Unicode types 2.Utilities for automatic conversion of python 2.x code
Python 3.6.5	March 2018	1.Unified str/Unicode types 2.Utilities for automatic conversion of python 2.x code
Python 3.7.0	May2018	1.New C API for thread-local strage 2.Built-in break point() 3.Data classes 4.Context variabes 5.More…
Python 3.8	October 2019	1.Assignment Expression 2.Positional-Only parameters

		3.Parallel file system cache for compiled bytecode files. 4.More..
Python 3.9	October 2020	1.Dictionary Merge & update operators 2.New removeprefix() and remove suffix() string methods 3.Builin Generic types.
Python 3.10	October 2021	1. Better Error Tracking 2. New Type Union Operator 3. Automatic Text Encoding
Python 3.11	24th, 2022	1. Error Location 2. The 'self' type 3.Exception note

Installation of python: There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from http://python.org/downloads/

Steps to be followed and remembered:

Step 1: Select Version of Python to Install.



Step 2: Download Python Executable Installer.

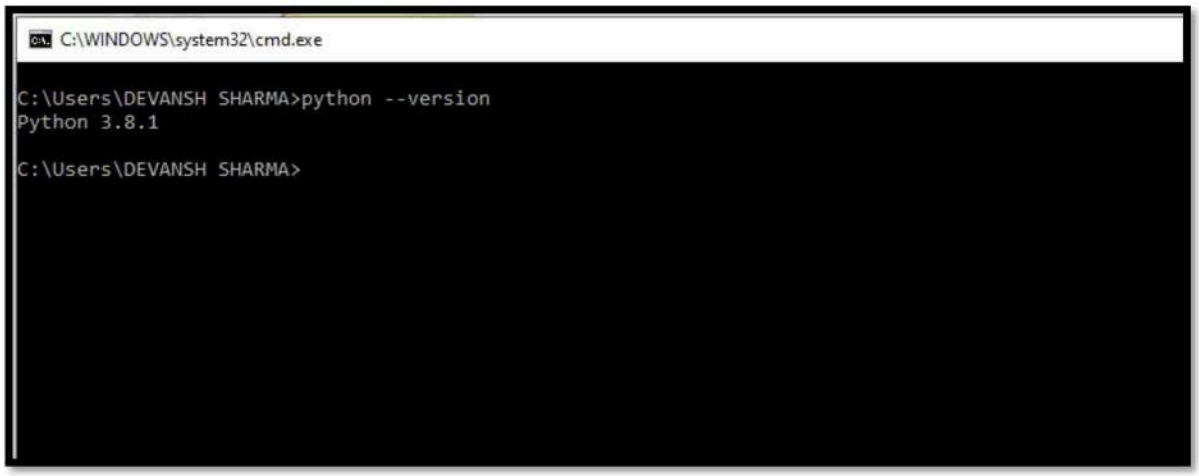
Step 3: Run Executable Installer.

Step 4: Verify Python Was Installed On Windows.

Step 5: Verify Pip Was Installed.

Step 6: Add Python Path to Environment Variables (Optional)

Now, try to run python on the command prompt. Type the command python -version in case of python3.

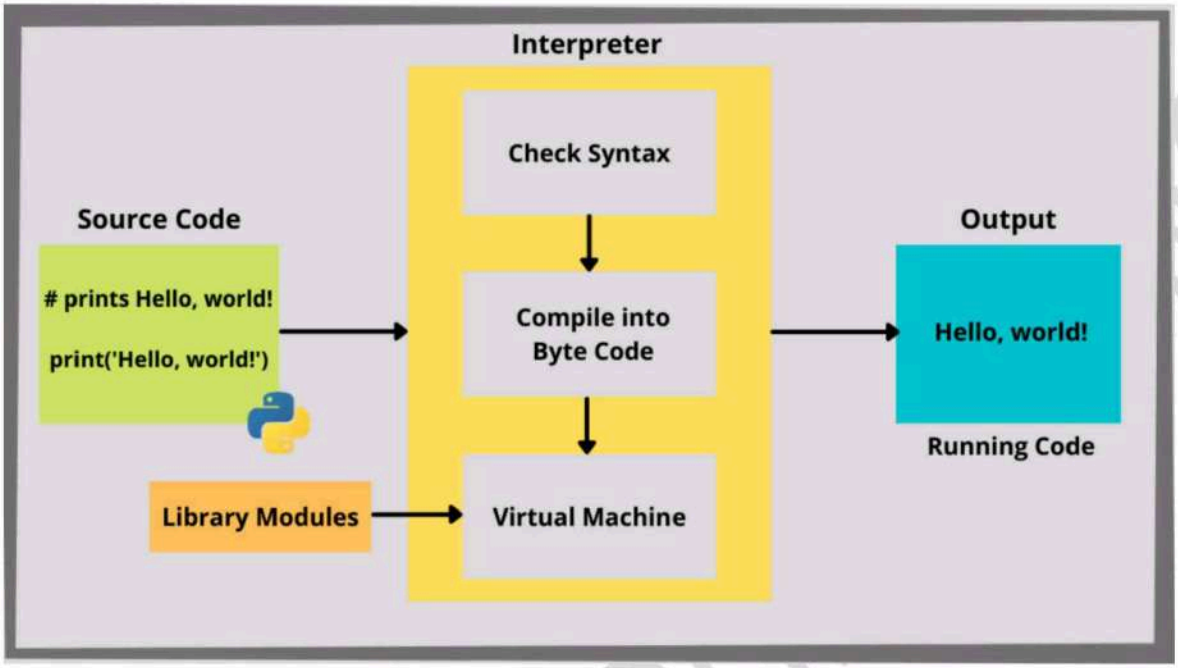


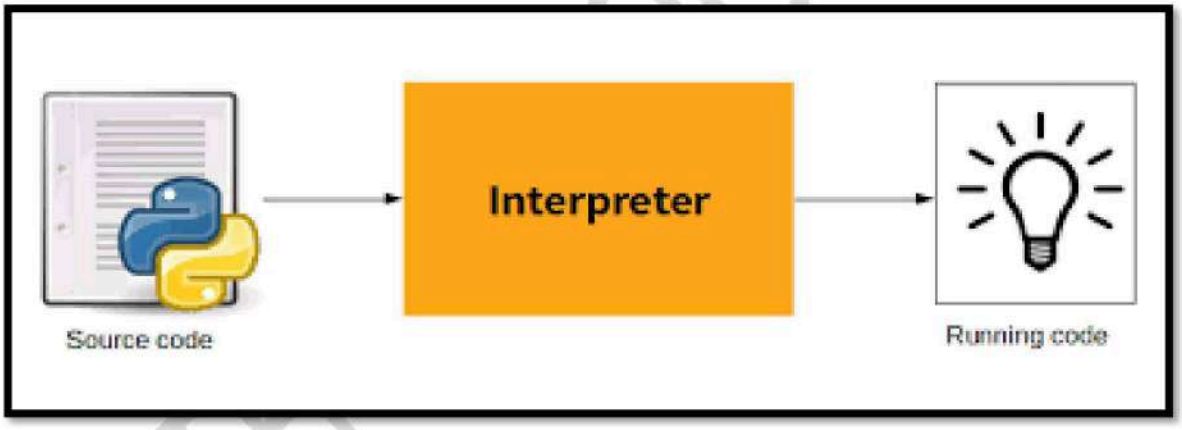
We are ready to work with the Python.

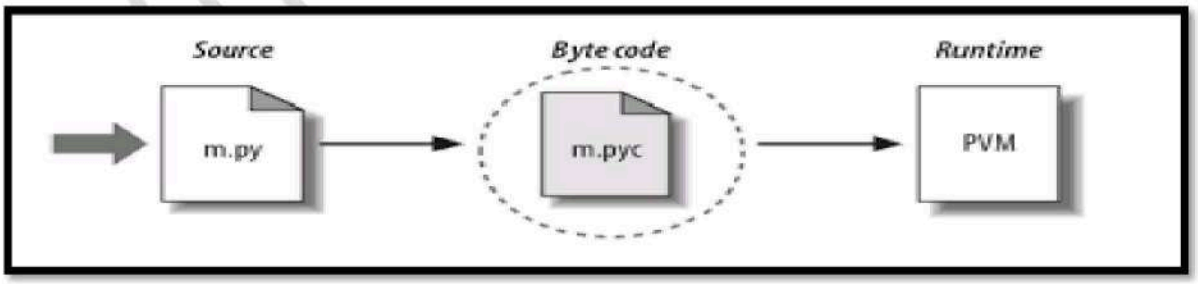
Working with Python

Python Code Execution:

Python's traditional runtime execution model: Source code you type is translated to bytecode, which is then run by the Python Virtual Machine (PVM). Your code is automatically compiled, but then it is interpreted.







Source code extension is .py

Byte code extension is .pyc (Compiled python code)

Python Command Line mode and Python IDEs

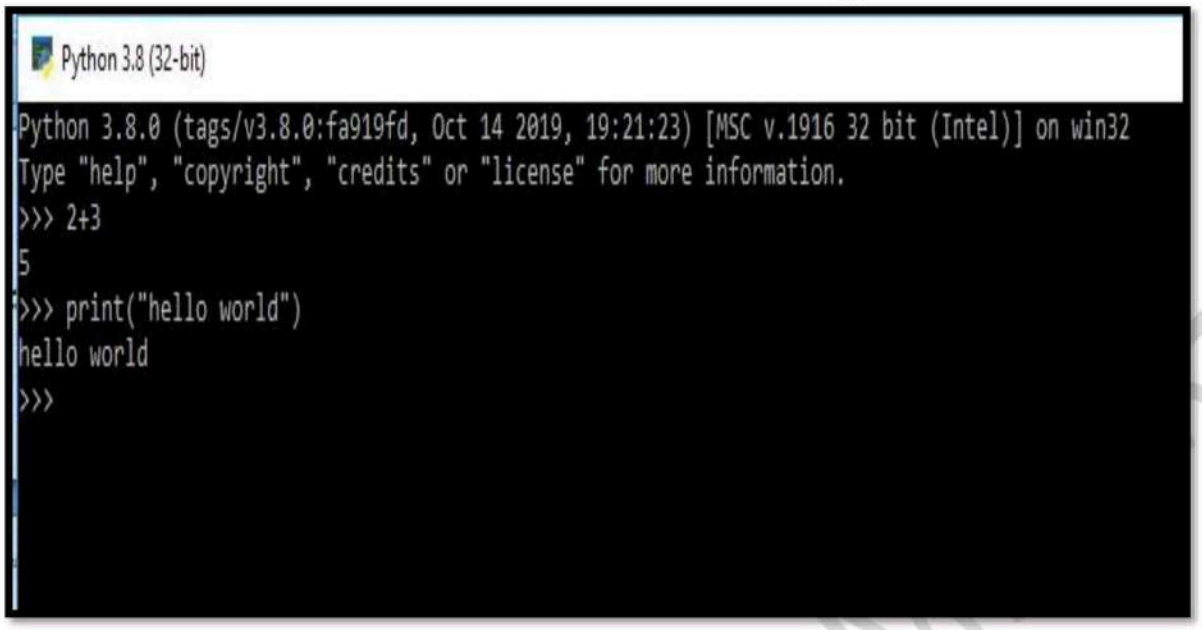
Python Command Line mode: There are two modes for using the Python interpreter:

- Using interpreter Interactive Mode or prompt
- Using Script Mode

Running Python in interactive mode:

Without passing python script file to the interpreter, directly execute code to Python prompt. Once you're inside the python interpreter, then you can start.

```
>>>print("Hello world")

# Relevant output is displayed on subsequent lines without the >>> symbol

>>> x=[0,1,2]

# Quantities stored in memory are not displayed by default.

>>> x

#If a quantity is stored in memory, typing its name will display it.

[0, 1, 2]

>>> 2+3

5
```



The screen at the beginning of the 1st line, i.e., the symbol >>> is a prompt the python interpreter uses to indicate that it is ready. If the programmer types 2+6, the interpreter replies 8.

Running Python in script mode:

Alternatively, programmers can store Python script source code in a file with the .py extension, and use the interpreter to execute the contents of the file. To execute the script by the interpreter, you have to tell the interpreter the name of the file. For example, if you have a script name MyFile.py and you're working on Unix, to run the script you have to type:

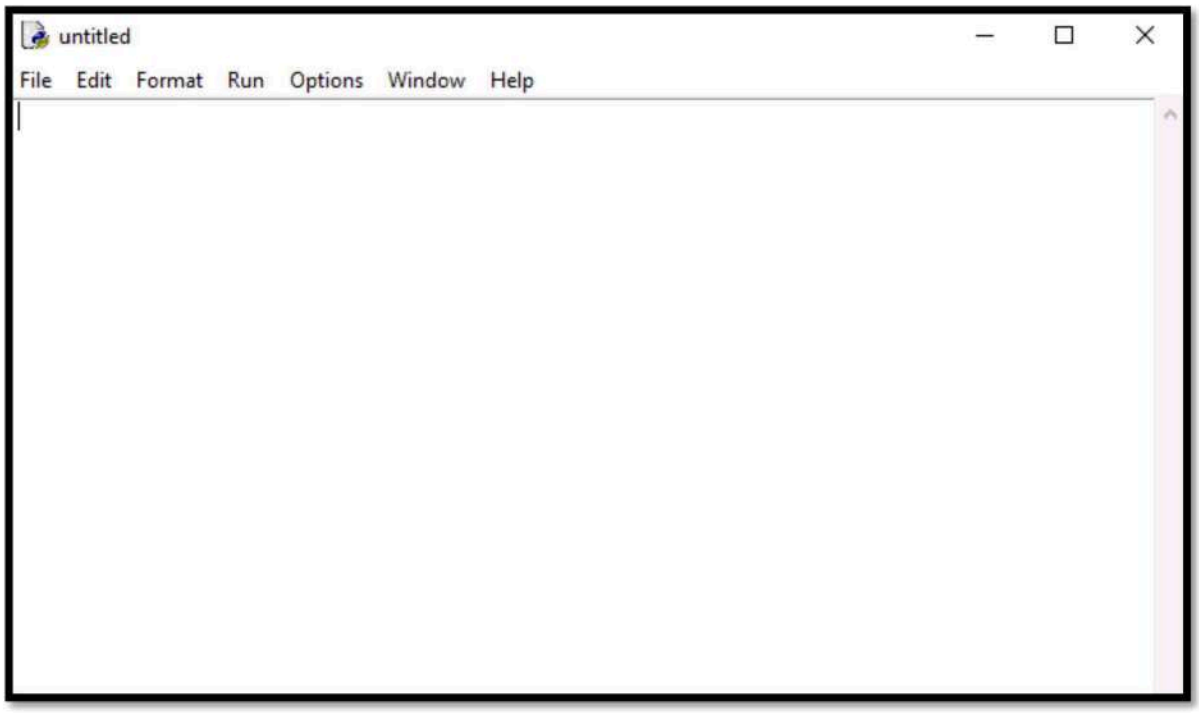
python MyFile.py

Working with the interactive mode is better when Python programmers deal with small pieces of code as you can type and execute them immediately, but when the code is more than 2-4 lines, using the script for coding can help to modify and use the code in future.

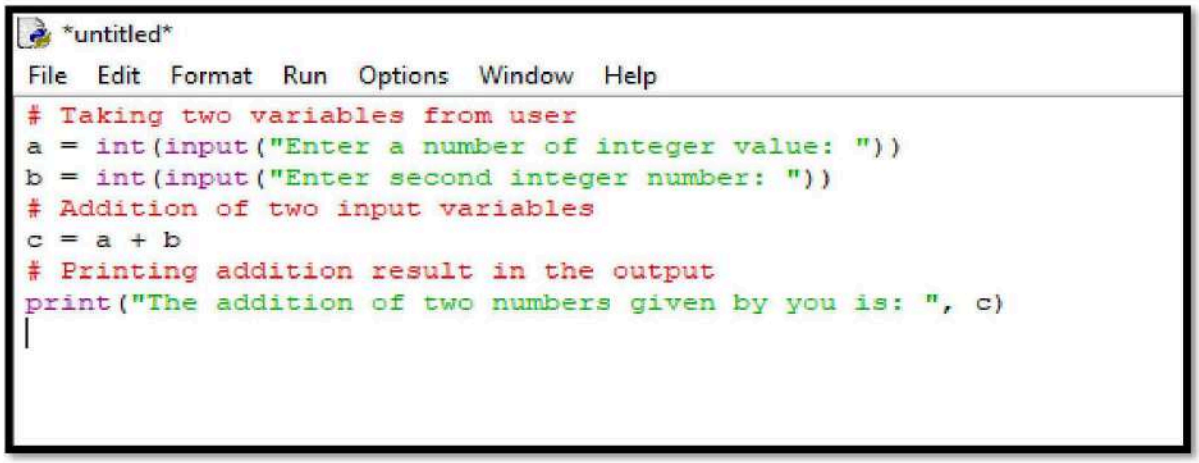
Creating and saving a Python program script file:

We have to follow the following steps in sequence in order to create and save a Python program script in our device:

Step 1: Open the text editor of Python or any other text editor of our choice (We will be using Python file editor here).



Step 2: Write the following Python program inside the text editor:



Step 3: After writing the Python program, we have to save the file inside the folder where our Python IDE is installed, and we have to save it with Python file extension, i.e., '.py' extension. We have saved the file with the 'code.py' name in our device for this program.

code.py - C:\Users\Manish\Downloads\code.py (3.9.0)

File Edit Format Run Options Window Help

```
# Taking two variables from user
a = int(input("Enter a number of integer value: "))
b = int(input("Enter second integer number: "))
# Addition of two input variables
c = a + b
# Printing addition result in the output
print("The addition of two numbers given by you is: ", c)
```

A Python script having a Python program is successfully created and saved in our device, and now we can move forward to execute the script.

Interactive mode	Script mode
A way of using the Python interpreter by typing commands and expressions at the prompt.	A way of using the Python interpreter to read and execute statements in a script.
Cant save and edit the code	Can save and edit the code
If we want to experiment with the code, we can use interactive mode.	If we are very clear about the code, we can use script mode.
we cannot save the statements for further use and we have to retype all the statements to re-run them.	we can save the statements for further use and we no need to retype all the statements to re-run them.
We can see the results immediately.	We cant see the code immediately.

Get Started with PyCharm

In our first program, we have used gedit on our CentOS as an editor. On Windows, we have an alternative like notepad or notepad++ to edit the code. However, these editors are not used as IDE for python since they are unable to show the syntax related suggestions.

JetBrains provides the most popular and a widely used cross-platform IDE **PyCharm** to run the python programs.

PyCharm installation

As we have already stated, PyCharm is a cross-platform IDE, and hence it can be installed on a variety of the operating systems. In this section of the tutorial, we will cover the installation process of PyCharm on Windows, MacOS, CentOS, and Ubuntu.

Windows

Installing PyCharm on Windows is very simple. To install PyCharm on Windows operating system, visit the link https://www.jetbrains.com/pycharm/download/download-thanks.html?platform=windows to download the executable installer.

Double click the installer (.exe) file and install PyCharm by clicking next at each step.

To create a first program to Pycharm follows the following step.

Step - 1. Open Pycharm editor. Click on "Create New Project" option to create new project.

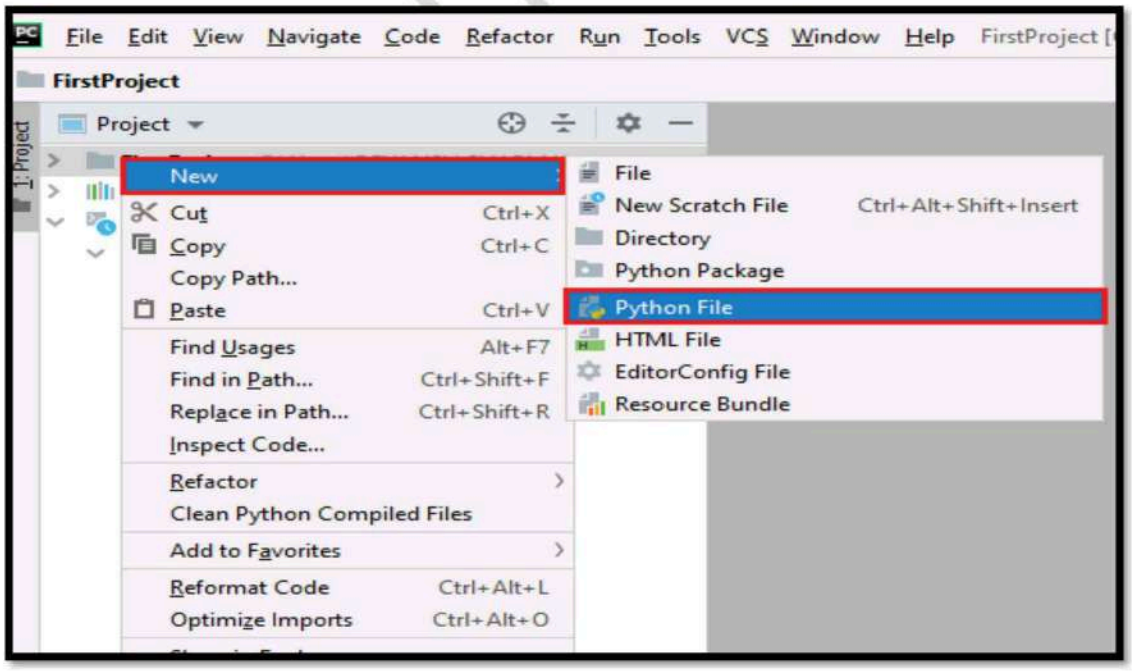
Step - 2. Select a location to save the project.

We can save the newly created project at desired memory location or can keep file location as it is but atleast change the project default name **untitled** to **"FirstProject"** or something meaningful.

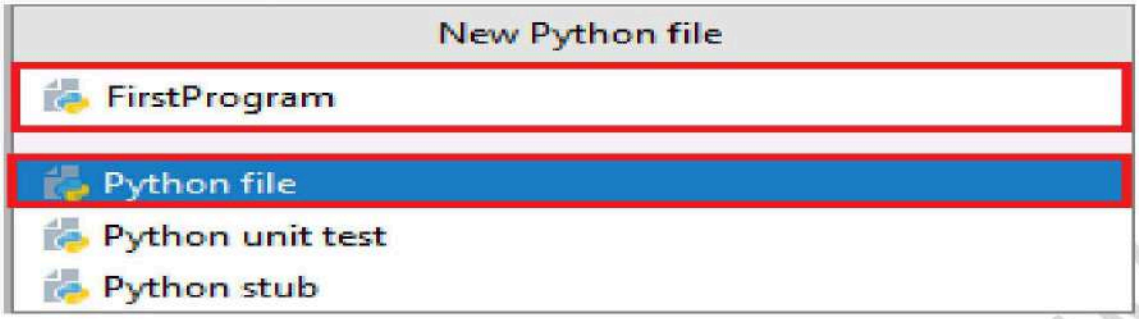
a. Pycharm automatically found the installed Python interpreter.

b. After change the name click on the "Create" Button.

Step - 3. Click on **"File"** menu and select **"New"**. By clicking "New" option it will show various file formats. Select the "Python File".

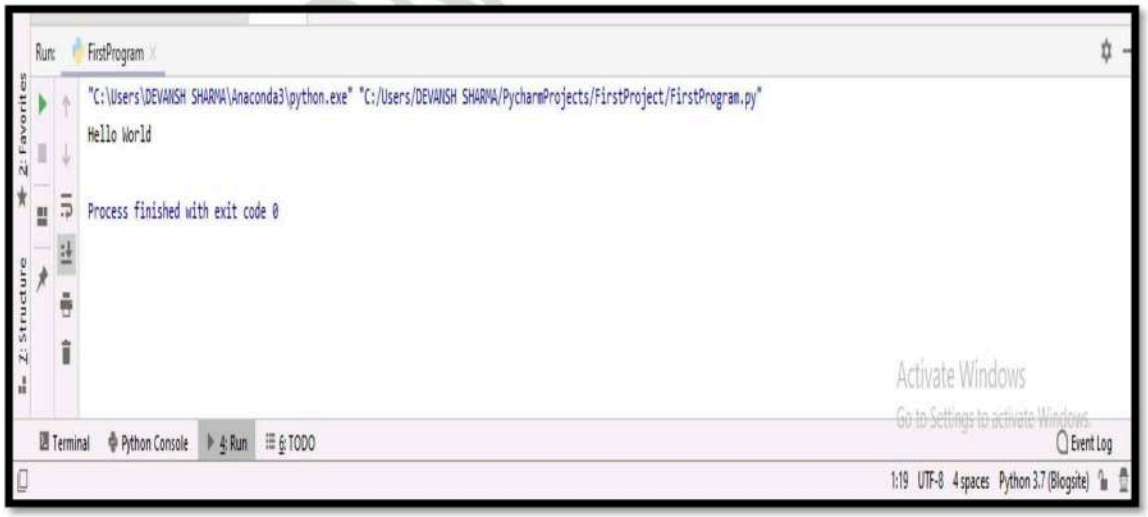


Step - 4. Now type the name of the Python file and click on "OK". We have written the "FirstProgram".



Step - 5. Now type the first program - print("Hello World") then click on the "Run" menu to run program.

Step - 6. The output will appear at the bottom of the screen.



Simple Python Program. A simple program that displays "Hello, World!". It's often used to illustrate the syntax of the language. Python is a popular, general-purpose programming language and has straightforward syntax. Python also provides many built-in functions.

This is the most basic Python program. It specifies how to print any statement in Python.

In old Python (up to Python 2.7.0), the print command is not written in parenthesis, but in new python software (Python 3.4.3), it is mandatory to add parenthesis to print a statement.

Let' see the following example to print Hello World

Example:

```
1. print ('Hello Python')
```

Output:

```
Hello World
```

In the above code, we have used the built-in print() method to print the string 'Hello World' on the terminal. Python takes only one line to print the 'hello world' program, we have used the built-in `print()` function to print the string `Hello, world!` on our screen.

By the way, a string is a sequence of characters. In Python, strings are enclosed inside single quotes, double quotes, or triple quotes.

CHAPTER -02

Python Basics:

Identifiers:

Identifier is a name used to identify a variable, function, class, module, etc. The identifier is a combination of character digits and underscore. The identifier should start with a character or Underscore then use a digit. The characters are A-Z or a-z, an Underscore (_) , and digit (0-9).

Rules for writing Identifiers in Python

1. The identifier is a combination of character digits and underscore and the character includes letters in lowercase (a-z), letters in uppercase (A-Z), digits (0-9), and an underscore (_).
2. An identifier cannot begin with a digit. If an identifier starts with a digit, it will give a Syntax error.
3. In Python, keywords are the reserved names that are built-in to Python, so a keyword cannot be used as an identifier - they have a special meaning and we cannot use them as identifier names.
4. Special symbols like !, @, #, $, %, etc. are not allowed in identifiers.
5. Python identifiers cannot only contain digits.
6. There is no restriction on the length of identifiers.
7. Identifier names are case-sensitive.

Python Valid Identifiers Example

1. abc123
2. abc_de
3. _abc
4. ABC
5. abc

Python Invalid Identifiers Example

1. 123abc
2. abc@
3. 123
4. For

Keywords:

Keywords are some predefined and reserved words in python that have special meanings. Keywords are used to define the syntax of the coding. The keyword cannot be used as an identifier, function, and variable name. All the keywords in python are written in lower case except True and False. There are 33 keywords in Python 3.7 let's go through all of them one by one.

Python Keywords

Here is the list of some reserved keywords in Python that cannot be used as identifiers.

False	def	If	raise
None	del	Import	return
True	elif	In	try
and	else	Is	while
as	except	Lambda	with
assert	finally	Nonlocal	yield

break	for	Not	await
class	form	Or	async
continue	global	Pass	

Statements and Expressions

A statement is an instruction that the Python interpreter can execute. We have normally two basic statements, the assignment statement and the print statement. Some other kinds of statements that are if statements, while statements, and for statements generally called as control flows.

Examples:

An assignment statement creates new variables and gives them values:

```
>>> x=10
```

>>> college="JSS"

An print statement is something which is an input from the user, to be printed / displayed on to the screen (or) monitor. >>> print("mrcet colege") mrcet college

Expressions: An expression is a combination of operators and operands that is interpreted to produce some other value. In any programming language, an expression is evaluated as per the precedence of its operators. So that if there is more than one operator in an expression, their precedence decides which operation will be performed first. We have many different types of expressions in Python. Let's discuss all types along with some exemplar codes :

1. Constant Expressions: These are the expressions that have constant values only.

Example:

```
# Constant Expressions

x = 15 + 1.3

print(x)
```

Output

16.3

2. Arithmetic Expressions: An arithmetic expression is a combination of numeric values, operators, and sometimes parenthesis. The result of this type of expression is also a numeric value. The operators used in these expressions are arithmetic operators like addition, subtraction, etc. Here are some arithmetic operators in Python:

Operators	Syntax	Functioning
+	x + y	Addition
–	x – y	Subtraction
*	x * y	Multiplication

/	x / y	Division
//	x // y	Quotient
%	x % y	Remainder
**	x ** y	Exponentiation

Example:

Let's see an exemplar code of arithmetic expressions in Python :

```
# Arithmetic Expressions
x = 40
y = 12

add = x + y
sub = x - y
pro = x * y
div = x / y

print(add)
print(sub)
print(pro)
print(div)
```

Output

52

28

480

3.3333333333333335

3. Integral Expressions: These are the kind of expressions that produce only integer results after all computations and type conversions.

Example:

```
# Integral Expressions
a = 13
b = 12.0

c = a + int(b)
print(c)
```

Output

25

4. Floating Expressions: These are the kind of expressions which produce floating point numbers as result after all computations and type conversions.

Example:

```
# Floating Expressions
a = 13
b = 5

c = a / b
print(c)
```

Output

2.6

5. Relational Expressions: In these types of expressions, arithmetic expressions are written on both sides of relational operator (> , < , >= , <=). Those arithmetic expressions are evaluated first, and then compared as per relational operator and produce a boolean output in the end. These expressions are also called Boolean expressions.

Example:

 # Relational Expressions

a = 21

b = 13

c = 40

d = 37

p = (a + b) >= (c - d)

Operator	Syntax	Functioning
and	P and Q	It returns true if both P and Q are true otherwise returns false
or	P or Q	It returns true if at least one of P and Q is true

(10 == 9) **Q = (7 > 5)** **# Logical Expressions** **R = P and Q** **S = P or Q** **T = not P** **print(R)** **print(S)** **print(T)** **not**	not P	It returns true if condition P is false

print(p)

Output

True

6. Logical Expressions: These are kinds of expressions that result in either *True* or *False*. It basically specifies one or more conditions. For example, (10 == 9) is a condition if 10 is equal to 9. As we know it is not correct, so it will return False. Studying logical expressions, we also come across some logical operators which can be seen in logical expressions most often. Here are some logical operators in Python:

Example:

Let's have a look at an exemplar code :

Output

False

True

True

7. Bitwise Expressions: These are the kind of expressions in which computations are performed at bit level.

Example:

```
# Bitwise Expressions
a = 12

x = a >> 2
y = a << 1

print(x, y)
```

Output

3 24

8. Combinational Expressions: We can also use different types of expressions in a single expression, and that will be termed as combinational expressions.

Example:

```
# Combinational Expressions
a = 16
b = 12

c = a + (b >> 1)
```

```
print(c)
```

Output

22

But when we combine different types of expressions or use multiple operators in a single expression, operator precedence comes into play.

Multiple operators in expression (Operator Precedence)

It's a quite simple process to get the result of an expression if there is only one operator in an expression. But if there is more than one operator in an expression, it may give different results on basis of the order of operators executed. To sort out these confusions, the *operator precedence* is defined. Operator Precedence simply defines the priority of operators that which operator is to be executed first. Here we see the operator precedence in Python, where the operator higher in the list has more precedence or priority:

Precedence	Name	Operator
1	Parenthesis	( ) [ ] { }
2	Exponentiation	**
3	Unary plus or minus, complement	-a , +a , ~a
4	Multiply, Divide, Modulo	/ * // %
5	Addition & Subtraction	+ −
6	Shift Operators	>> <<
7	Bitwise AND	&
8	Bitwise XOR	^
9	Bitwise OR	|
10	Comparison Operators	>= <= > <
11	Equality Operators	== !=
12	Assignment Operators	= += -= /= *=
13	Identity and membership operators	is, is not, in, not in

14	Logical Operators	and, or, not

So, if we have more than one operator in an expression, it is evaluated as per operator precedence. For example, if we have the expression "10 + 3 * 4". Going without precedence it could have given two different outputs 22 or 52. But now looking at operator precedence, it must yield 22. Let's discuss this with the help of a Python program:

```
# Multi-operator expression

a = 10 + 3 * 4
print(a)

b = (10 + 3) * 4
print(b)

c = 10 + (3 * 4)
print(c)
```

Output

22

52

22

Hence, operator precedence plays an important role in the evaluation of a Python expression.

Variables: Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Rules for Python variables:

- A variable name must start with a letter or the underscore character

- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

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 −

```
a= 100 # An integer assignment
b = 1000.0 # A floating point
c = "John" # A string
print (a)
print (b)
print (c)
```

This produces the following result −

```
100
1000.0
John
```

Multiple Assignment:

Python allows us to assign a value to multiple variables in a single statement, which is also known as multiple assignments.

We can apply multiple assignments in two ways, either by assigning a single value to multiple variables or assigning multiple values to multiple variables. Consider the following example.

1. Assigning single value to multiple variables

```
>>>x=y=z=50

>>>print(x)

>>>print(y)
>>>print(z)
```

Output:

```
50
50
50
```

2. Assigning multiple values to multiple variables:

```
>>>a,b,c=5,10,15
>>>print a
>>>print b
>>>print c
```

Output:

```
5
10
15
```

Object References

It is necessary to understand how the Python interpreter works when we declare a variable. The process of treating variables is somewhat different from many other programming languages.

Python is the highly object-oriented programming language that's why every data item belongs to a specific type of class. Consider the following example.

```
>>>print("John")
```

Output:

```
John
```

The Python object creates an integer object and displays it to the console. In the above print statement, we have created a string object. Let's check the type of it using the Python built-in **type**() function.

```
>>>type("John")
```

Output:

```
<class 'str'>
```

In Python, variables are a symbolic name that is a reference or pointer to an object. The variables are used to denote objects by that name.

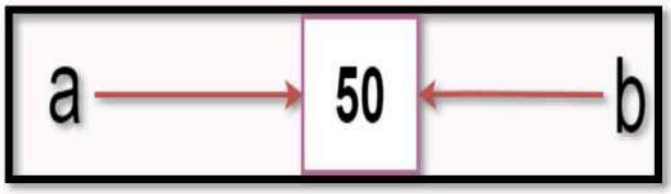
Let's understand the following example

```
>>>a = 50
```

In the above image, the variable **a** refers to an integer object.

Suppose we assign the integer value 50 to a new variable b.

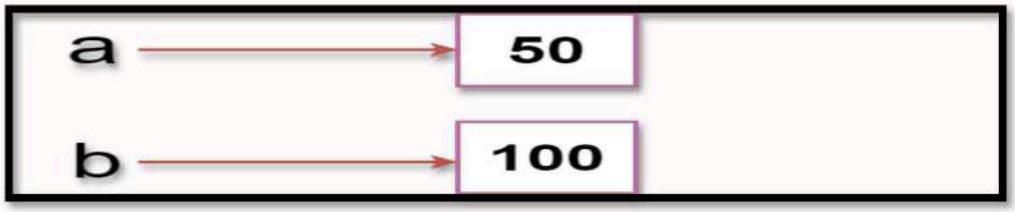
```
a = 50

b = a
```



The variable b refers to the same object that a points to because Python does not create another object.

Let's assign the new value to b. Now both variables will refer to the different objects.

```
a = 50

b =100
```



Python manages memory efficiently if we assign the same variable to two different values.

Variable Names

We have already discussed how to declare the valid variable. Variable names can be any length can have uppercase, lowercase (A to Z, a to z), the digit (0-9), and underscore character(_). Consider the following example of valid variables names.

```
>>>name = "Devansh"
>>>age = 20
>>>marks = 80.50
>>>print(name)
>>>print(age)
>>>print(marks)
```

Output:

```
Devansh
20
80.5
```

Consider the following valid variables name.

```
>>>name = "A"
>>>Name = "B"
>>>naMe = "C"
>>>NAME = "D"
>>>n_a_m_e = "E"
>>>_name = "F"
>>>name_ = "G"
>>>_name_ = "H"
>>>na56me = "I"

>>>print(name,Name,naMe,NAME,n_a_m_e, NAME, n_a_m_e, _name, name_,_name, na56me)
```

Output:

```
A B C D E D E F G F I
```

In the above example, we have declared a few valid variable names such as name, _name_ , etc. But it is not recommended because when we try to read code, it may create confusion. The variable name should be descriptive to make code more readable.

Python Variable Types

There are two types of variables in Python

1. Local variable and
2. Global variable.

Let's understand the following variables.

Local Variable

Local variables are the variables that declared inside the function and have scope within the function. Let's understand the following example.

Example :

```
# Declaring a function
def add():
    # Defining local variables. They has scope only within a function
    a = 20
    b = 30
    c = a + b
    print("The sum is:", c)
# Calling a function
    add()
```

Output:

```
The sum is: 50
```

Explanation:

In the above code, we declared a function named **add()** and assigned a few variables within the function. These variables will be referred to as the **local variables** which have scope only inside the function. If we try to use them outside the function, we get a following error.

```
    add()
# Accessing local variable outside the function
print(a)
```

Output:

```
The sum is: 50
   print(a)
NameError: name 'a' is not defined
```

We tried to use local variable outside their scope; it threw the **NameError.**

Global Variables

Global variables can be used throughout the program, and its scope is in the entire program. We can use global variables inside or outside the function.A variable declared outside the function is the global variable by default. Python provides the **global** keyword to use global variable inside the function. If we don't use the **global** keyword, the function treats it as a local variable. Let's understand the following example.

Example:

```
# Declare a variable and initialize it
x = 101
# Global variable in function
def mainFunction():
 # printing a global variable
 global x
 print(x)
 # modifying a global variable
```

```
 x = 'Welcome To  python'
 print(x)


mainFunction()
print(x)
```

Output:

```
101
Welcome To Python
Welcome To Python
```

Explanation:

In the above code, we declare a global variable **x** and assign a value to it. Next, we defined a function and accessed the declared variable using the **global** keyword inside the function. Now we can modify its value. Then, we assigned a new string value to the variable x.

Now, we called the function and proceeded to print **x**. It printed the as newly assigned value of x.

Delete a variable

We can delete the variable using the **del** keyword. The syntax is given below.

Syntax :

del <variable_name>

In the following example, we create a variable x and assign value to it. We deleted variable x, and print it, we get the error **"variable x is not defined"**. The variable x will no longer use in future.

Example :

```
# Assigning a value to x
x = 6
print(x)
# deleting a variable.
del x
print(x)
```

Output:

```
6
Traceback (most recent call last):
  File "C:/Users/DEVANSH SHARMA/PycharmProjects/Hello/multiprocessing.py", line 389, in
    print(x)
NameError: name 'x' is not defined
```

Output Variables:

The Python print statement is often used to output variables.

Variables do not need to be declared with any particular type and can even change type after they have been set.

```
x = 5 # x is of type int
x = "jsscs " # x is now of type str
print(x)
```

Output: jsscs

To combine both text and a variable, Python uses the **"+"** character:

Example:

```
x = "awesome"
print("Python is " + x)
```

Output

Python is awesome

You can also use the + character to add a variable to another variable:

Example

```
x = "Python is "
y = "awesome"
z = x + y
print(z)
```

Output:

Python is awesome

BASIC INPUT AND OUTPUT:

Python Output

In Python, we can simply use the print() function to print output. For example,

```
>>>print('Python is powerful')
```

Output: Python is powerful

Here, the print() function displays the string enclosed inside the single quotation.

Syntax of print()

In the above code, the print() function is taking a single parameter.

However, the actual syntax of the print function accepts **5** parameters

Print(object= separator= end= file= flush=)

Here,**object -** value(s) to be printed

sep **(optional) -** allows us to separate multiple **objects** inside print().

end (optional) - allows us to add add specific values like new line "\n", tab "\t"

file (optional) - where the values are printed. It's default value is sys.stdout (screen)

flush (optional) - boolean specifying if the output is flushed or buffered. Default: False

Example 1: Python Print Statement

```
>>>print('Good Morning!')
print('It is rainy today')
```

Output: Good Morning!

It is rainy today

In the above example, the print() statement only includes the **object** to be printed. Here, the value for **end** is not used. Hence, it takes the default value '\n'.

So we get the output in two different lines.

Example 2: Python print() with end Parameter

```
# print with end whitespace
>>>print('Good Morning!', end= ' ')
>>>print('It is rainy today')
```

Output: Good Morning! It is rainy today

Notice that we have included the end= ' ' after the end of the first print() statement.

Hence, we get the output in a single line separated by space.

Example 3: Python print() with sep parameter

```
>>> print('New Year', 2023, 'See you soon!', sep= '. ')
```

Output: New Year. 2023. See you soon!

In the above example, the print() statement includes multiple **items** separated by a comma.Notice that we have used the optional parameter sep= ". " inside the print() statement.Hence, the output includes items separated by . not comma.

Example: Print Python Variables and Literals

We can also use the print() function to print Python variables. For example,

```
>>>number = -10.6
name = "JSSCOLLEGE"
# print literals
print(5)
# print variables
print(number)
print(name)
```

Output:5

-10.6

JSSCOLLEGE

Example: Print Concatenated Strings

We can also join two strings together inside the print() statement. For example,

>>>print('OOTY is ' + 'awesome.')

Output: OOTY is awesome.

Here,the + operator joins two strings 'OOTY is ' and 'awesome.' And the print() function prints the joined string

Output formatting:

Sometimes we would like to format our output to make it look attractive. This can be done by using the str.format() method. For example,

```
x = 5
y = 10
print('The value of x is {} and y is {}'.format(x,y))
```

Output: The value of x is 5 and y is 10

Here, the curly braces {} are used as placeholders. We can specify the order in which they are printed by using numbers (tuple index).

Python Input

While programming, we might want to take the input from the user. In Python, we can use the input() function.

Syntax of input(): Input(prompt)

Here, prompt is the string we wish to display on the screen. It is optional.

Example: Python User Input

```
# using input() to take user input
num = input('Enter a number: ')
```

```
print('You Entered:', num)
print('Data type of num:', type(num))
```

Output:

Enter a number: 20

You Entered: 20

Data type of num: <class 'str'>

In the above example, we have used the input() function to take input from the user and stored the user input in the num variable.

It is important to note that the entered value **10** is a string, not a number. So, type(num) returns <class 'str'>.

To convert user input into a number we can use int() or float() functions as:

num=int(input("Enter a number:"))

Here, the data type of the user input is converted from string to integer .

Operators: The operator is a symbol that performs a certain operation between two operands, according to one definition. In a particular programming language, operators serve as the foundation upon which logic is constructed in a programme. The different operators that Python offers are listed here.

1. Arithmetic operators
2. Assignment Operators
3. Comparison Operators
4. Logical Operators
5. Bitwise Operators
6. Special Operators
7. Membership Operators
8. Identity Operators

1.Arithmetic Operators

Operator	Operation	Example
+	Addition	5 + 2 = 7
-	Subtraction	4 - 2 = 2
*	Multiplication	2 * 3 = 6
/	Division	4 / 2 = 2
//	Floor Division	10 // 3 = 3
%	Modulo	5 % 2 = 1
**	Power	4 ** 2 = 16

Arithmetic operations between two operands are carried out using arithmetic operators. It includes the exponent (**) operator as well as the + (addition), - (subtraction), * (multiplication), / (divide), % (reminder), and // (floor division) operators.

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc. For example,

```
sub = 10 - 5
```

Here, - is an arithmetic operator that subtracts two values or variables.

Example 1: Arithmetic Operators in Python

```
a = 7
b = 2

# addition
print ('Sum: ', a + b)

# subtraction
print ('Subtraction: ', a - b)
```

```
# multiplication
print ('Multiplication: ', a * b)

# division
print ('Division: ', a / b)

# floor division
print ('Floor Division: ', a // b)

# modulo
print ('Modulo: ', a % b)

# a to the power b
print ('Power: ', a ** b)
```

Output: Sum: 9
Subtraction: 5
Multiplication: 14
Division: 3.5
Floor Division: 3
Modulo: 1
Power: 49

In the above example, we have used multiple arithmetic operators,

- + to add a and b
- - to subtract b from a
- * to multiply a and b
- / to divide a by b

- // to floor divide a by b
- % to get the remainder
- ** to get a to the power b

2.Python Assignment Operators

The right expression's value is assigned to the left operand using the assignment operators.

Assignment operators are used to assign values to variables.

For example,

```
# assign 5 to x
var x = 5
```

Here, = is an assignment operator that assigns 5 to x.

Operator	Name	Example
=	Assignment Operator	a = 7
+=	Addition Assignment	a += 1 # a = a + 1
-=	Subtraction Assignment	a -= 3 # a = a - 3
*=	Multiplication Assignment	a *= 4 # a = a * 4
/=	Division Assignment	a /= 3 # a = a / 3
%=	Remainder Assignment	a %= 10 # a = a % 10
**=	Exponent Assignment	a **= 10 # a = a ** 10

Here's a list of different assignment operators available in Python.

Example 2: Assignment Operators

```
# assign 10 to a
a = 10
```

```
# assign 5 to b
b = 5

# assign the sum of a and b to a
a += b      # a = a + b

print(a)

# Output: 15
```

Here, we have used the `+=` operator to assign the sum of a and b to a.

Similarly, we can use any other assignment operators according to the need.

3.Python Comparison Operators

Comparison operators compare the values of the two operands and return a true or false Boolean value in accordance.

Comparison operators compare two values/variables and return a boolean result: `True` or `False`.

For example,

```
a = 5
b =2

print (a > b)

#Output: True
```

Here, the `>` comparison operator is used to compare whether `a` is greater than `b` or not.

Operator	Meaning	Example

==	Is Equal To	3 == 5 gives us **False**
!=	Not Equal To	3 != 5 gives us **True**
>	Greater Than	3 > 5 gives us **False**
<	Less Than	3 < 5 gives us **True**
>=	Greater Than or Equal To	3 >= 5 give us **False**
<=	Less Than or Equal To	3 <= 5 gives us **True**

Example 3: Comparison Operators

```
a = 5

b = 2

# equal to operator
print('a == b =', a == b)

# not equal to operator
print('a != b =', a != b)

# greater than operator
print('a > b =', a > b)

# less than operator
print('a < b =', a < b)

# greater than or equal to operator
print('a >= b =', a >= b)
```

less than or equal to operator

print('a <= b =', a <= b)

Output: a == b = False

a != b = True

a > b = True

a < b = False

a >= b = True

a <= b = False

Note: Comparison operators are used in decision-making and loops. We'll discuss more of the comparison operator and decision-making in later tutorials.

Operator	Example	Meaning
and	a **and** b	**Logical AND**: True only if both the operands are True
or	a **or** b	**Logical OR**: True if at least one of the operands is True
not	**not** a	**Logical NOT**: True if the operand is False and vice-versa.

3.Python Logical Operators: Logical operators are used to check whether an expression is True or False. They are used in decision-making. The assessment of expressions to make decisions typically makes use of the logical operators. For example,

a = 5

b = 6

print((a > 2) and (b >= 6))

Output:TruHere, and is the logical operator **AND**. Since both a > 2 and b >= 6 are True, the result is True.

Example 4: Logical Operators

```
# logical AND
print(True and True)    # True
print(True and False)   # False

# logical OR
print(True or False)    # True

# logical NOT
print(not True)         # False
```

Output: True
False
True
False

Note: Here is the truth table for these logical operators.

5.Python Bitwise operators

The two operands' values are processed bit by bit by the bitwise operators.

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name.

For example, **2** is 10 in binary and **7** is 111.

In the table below: Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

Operator	Meaning	Example

&	Bitwise AND	x & y = 0 (0000 0000)
|	Bitwise OR	x | y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x >> 2 = 2 (0000 0010)
<<	Bitwise left shift	x << 2 = 40 (0010 1000)

For example,

if a = 7

b = 6

then, binary (a) = 0111

binary (b) = 0110

hence, a & b = 0011

a | b = 0111

a ^ b = 0100

~ a = 1000

6. Python Special operators

Python language offers some special types of operators like the **identity** operator and the **membership** operator. They are described below with examples.

Identity operatorsIn Python, is and is not are used to check if two values are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True

is not	True if the operands are not identical (do not refer to the same object)	x is not True

Example 4: Identity operators in Python

```
x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'
x3 = [1,2,3]
y3 = [1,2,3]

print(x1 is not y1)  # prints False

print(x2 is y2)  # prints True

print(x3 is y3)  # prints False
```

Output: True

False

True

Here, we see that x1 and y1 are integers of the same values, so they are equal as well as identical. Same is the case with x2 and y2 (strings).

But x3 and y3 are lists. They are equal but not identical. It is because the interpreter locates them separately in memory although they are equal.

7.Membership operators

The membership of a value inside a Python data structure can be verified using Python membership operators. The result is true if the value is in the data structure; otherwise, it returns false.In Python, in and not in are the membership operators. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

In a dictionary we can only test for presence of key, not the value.

Operator	Meaning	Example
in	True if value/variable is **found** in the sequence	5 in x
not in	True if value/variable is **not found** in the sequence	5 not in x

Example 5: Membership operators in Python

```
x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'
x3 = [1,2,3]
y3 = [1,2,3]

print(x1 is not y1)  # prints False

print(x2 is y2)  # prints True

print(x3 is y3)  # prints False
```

Output: True

False

True

False

Here, 'H' is in x but 'hello' is not present in x (remember, Python is case sensitive).

Similarly, 1 is key and 'a' is the value in dictionary y. Hence, 'a' in y returns False.

8. Identity Operators

Operator	Description
is	If the references on both sides point to the same object, it is determined to be true.
is not	If the references on both sides do not point at the same object, it is determined to be true.

Precedence and Association:

Precedence : Operator precedence in Python simply refers to the order of operations. Operators are used to perform operations on variables and values. Python classifies its operators in the following groups: Arithmetic operators.

It would be best if the reader understood how Python assesses the ordering of its operators after checking it. Some operators prioritize others; for example, the division operator takes precedence over the multiplication operator; therefore, division comes first.

The Python interpreter executes operations of higher precedence operators first in any given logical or arithmetic expression. Except for the exponent operator (**), all other operators are executed from left to right.

Precedence of Python Operators

An expression is a collection of numbers, variables, operations, and built-in or user-defined function calls. The Python interpreter can evaluate a valid expression.

```
>>>2 - 7
```

Output:

-5

The expression 2 - 7 is used as an example here. In an expression, we can add multiple operators. There is a principle of precedence in Python for evaluating these types of expressions. It directs the sequence in which certain tasks are completed.

Division, for instance, takes precedence over addition.

```
# Division has higher precedence than addition
>>>16 + 16 / 2
```

Output:

```
24.0
```

However, we can reverse this sequence by employing parenthesis (), which takes precedence over division.

```
# Parentheses () holds higher precedence
(16 + 16) / 2
```

Output:

```
16.0
```

The following table shows the precedence of Python operators. It's in reverse order (the upper operator holds higher precedence than the lower operator).

Operators	Meaning
()	Parentheses
**	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT

*, /, //, %	Multiplication, Division, Floor division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
==, !=, >, >=, <, <=, is, is not, in, not in	Comparisons, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR

Consider the following examples:

Assume we're building an if...else block that only executes if the color is red or green and quantity is greater than or equal to 5.

```
# Precedence of and and or operators
>>>colour = "red"
>>>quantity = 0
>>>if colour == "red" or colour == "green" and quantity >= 5:
>>>print("Your parcel is dispatched")
>>>else:
>>>print("your parcel cannot be dispatched")
```

Output:

Your parcel is dispatched

Even though quantity is 0, this program runs if block. Because and takes precedence over or does not produce the anticipated result.

Appropriately employing parenthesis (), we may get the required result:

Example:

```
# Precedence of and and or operators
>>>colour = "red"
>>>quantity = 0
>>>if (colour == "red" or colour == "green") and quantity >= 5:
>>> print("Your parcel is dispatched")
>>>else:
>>> print("Your parcel cannot be dispatched")
```

Output:

Your parcel cannot be dispatched

Associativity /Association of Python Operators

We can observe that a given category contains many operators in the list above. The order of these operators is identical.

Associativity/Association aids in determining the sequence of operations when two operators share the same priority.

The direction in which any given expression with more than one operator having the same precedence is assessed is associativity. Almost every operator is associative from left to right.

```
# This shows Left to right associativity
```

```
>>>print(4 * 9 // 3)
# Using parenthesis this time to show left to right associativity
>>>print(4 * (9 // 3))
```

Output:

12

12

Note: In Python, the exponent operation ** possesses the right to left associativity.

```
# This example demonstrates the right to left associativity of exponent operator **
>>>print (3 ** 2 ** 3)
# If we want to first solve 5 ** 7 then we can use ()
>>>print ((3 ** 2) ** 3)
```

Output:

6561

729

Non Associative Operators

Several operators, such as comparison or assignment operators, don't have such associativity rules in Python. Patterns of this type of operator have their own rules that can not be represented as associativity. For instance, a < b < c is not the same as (a < b) < c or a < (b < c). a < b < c is assessed from left to right and is similar to a < b and b < c.

Additionally, while linking assignment operators such as a = b = c = 3 is entirely acceptable, a = b = c += 2 is not acceptable.

```
# Defining the variables a, b, and, c
>>>a = b = c = 3
#If the operator += follows associativity answer will shown otherwise error will be raised
```

```
>>>a = b = c += 2
```

Output:

```
a = b = c += 2

^SyntaxError: invalid syntax
```

Hence, this operator does not follow left-to-right associativity

DataTypes: The data stored in memory can be of many types. For example, a student roll number is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Variables can hold values, and every value has a data-type. Python is a dynamically typed language; hence we do not need to define the type of the variable while declaring it. The interpreter implicitly binds the value with its type.

```
>>>a = 5
```

The variable **a** holds integer value five and we did not define its type. Python interpreter will automatically interpret variables **a** as an integer type.

Python enables us to check the type of the variable used in the program. Python provides us the **type()** function, which returns the type of the variable passed.

Consider the following example to define the values of different data types and checking its type.

```
>>>a=10
b="Hi Python"
c = 10.5
print(type(a))
print(type(b))
print(type(c))
```

Output:

```
<type 'int'>
<type 'str'>
<type 'float'>
```

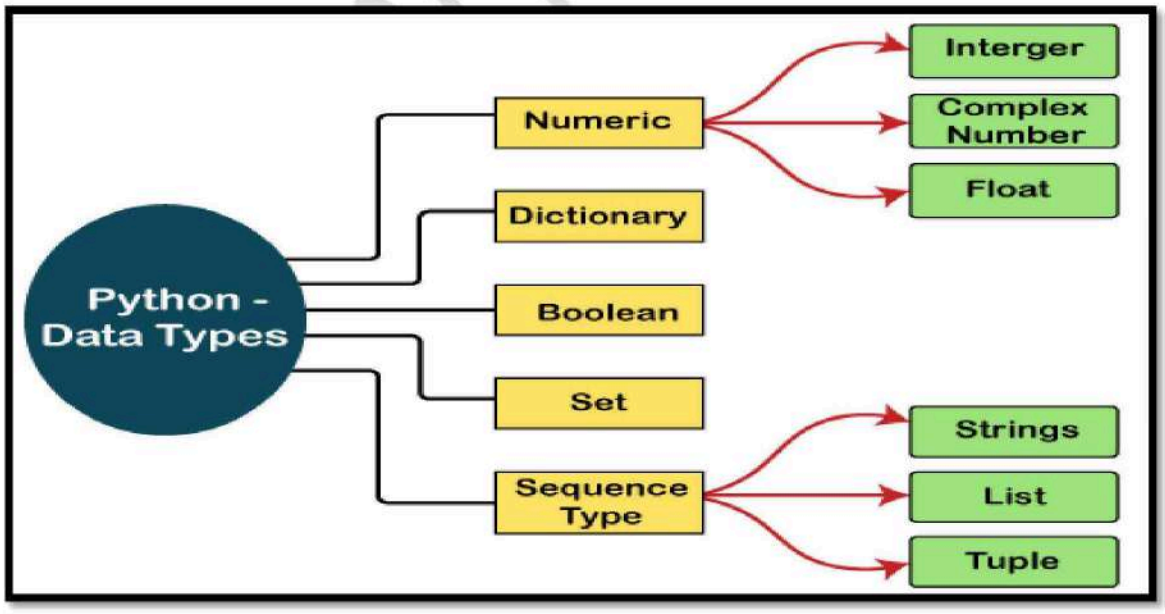
Standard data types

A variable can hold different types of values. For example, a person's name must be stored as a string whereas its id must be stored as an integer.

Python provides various standard data types that define the storage method on each of them.

The data types defined in Python are given below.

1. **Numbers**
2. **Sequence Type**
3. **Boolean**
4. **Set**
5. **Dictionary**



In this section of the tutorial, we will give a brief introduction of the above data-types. We will discuss each one of them in detail later in this tutorial.

1.Numbers

Number stores numeric values. The integer, float, and complex values belong to a Python Numbers data-type. Python provides the **type()** function to know the data-type of the variable. Similarly, the **isinstance()** function is used to check an object belongs to a particular class.

Python creates Number objects when a number is assigned to a variable. For example;

```
>>>a = 5
print("The type of a", type(a))
    b = 40.5
    print("The type of b", type(b))
c = 1+3j
print("The type of c", type(c))
print(" c is a complex number", is instance(1+3j,complex))
```

Output:

```
The type of a <class 'int'>
The type of b <class 'float'>
The type of c <class 'complex'>
c is complex number: True
```

Python supports three types of numeric data.

1. **Int -** Integer value can be any length such as integers 10, 2, 29, -20, -150 etc. Python has no restriction on the length of an integer. Its value belongs to **int**
2. **Float -** Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate upto 15 decimal points.
3. **complex -** A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts, respectively. The complex numbers like 2.14j, 2.0 + 2.3j, etc.

2.Sequence Type

String

The string can be defined as the sequence of characters represented in the quotation marks. In Python, we can use single, double, or triple quotes to define a string.

String handling in Python is a straightforward task since Python provides built-in functions and operators to perform operations in the string.

In the case of string handling, the operator + is used to concatenate two strings as the operation *"hello"+" python"* returns *"hello python"*.

The operator * is known as a repetition operator as the operation "Python" *2 returns 'Python Python'.

The following example illustrates the string in Python.

Example :

```
    >>>str = "string using double quotes"
print(str)
s = '''A multiline
    string'''
print(s)
```

Output:

```
string using double quotes
A multiline
string
```

Consider the following example of string handling.

Example 2:

```
>>>str1 = 'hello world' #string str1
str2 = ' how are you' #string str2
print (str1[0:2]) #printing first two character using slice operator
print (str1[4]) #printing 4th character of the string
print (str1*2) #printing the string twice
print (str1 + str2) #printing the concatenation of str1 and str2
```

Output:

```
he
o
hello worldhello world
hello world how are you
```

3.List

Python Lists are similar to arrays in C. However, the list can contain data of different types. The items stored in the list are separated with a comma (,) and enclosed within square brackets [].

We can use slice [:] operators to access the data of the list. The concatenation operator (+) and repetition operator (*) works with the list in the same way as they were working with the strings.

Consider the following example.

```
>>>list1 = [1, "hi", "Python", 2]
#Checking type of given list
print(type(list1))

#Printing the list1
print (list1)

# List slicing
print (list1[3:])
```

```
# List slicing
print (list1[0:2])

    # List Concatenation using + operator
print (list1 + list1)

    # List repetation using * operator
print (list1 * 3)
```

Output:

```
[1, 'hi', 'Python', 2]
[2]
[1, 'hi']
[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]
[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]
```

4.Tuple

A tuple is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types. The items of the tuple are separated with a comma (,) and enclosed in parentheses ().

A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple.

Let's see a simple example of the tuple.

```
>>>tup = ("hi", "Python", 2)
# Checking type of tup
print (type(tup))

#Printing the tuple
```

```
print (tup)

    # Tuple slicing
    print (tup[1:])
    print (tup[0:1])
     # Tuple concatenation using + operator
    print (tup + tup)

    # Tuple repatation using * operator
    print (tup * 3)

    # Adding value to tup. It will throw an error.
    t[2] = "hi"
```

Output:

```
<class 'tuple'>
('hi', 'Python', 2)
('Python', 2)
('hi',)
('hi', 'Python', 2, 'hi', 'Python', 2)
('hi', 'Python', 2, 'hi', 'Python', 2, 'hi', 'Python', 2)
Traceback (most recent call last):
File "main.py", line 14, in <module>
t[2] = "hi";
TypeError: 'tuple' object does not support item assignment
```

5.Dictionary

Dictionary is an unordered set of a key-value pair of items. It is like an associative array or a hash table where each key stores a specific value. Key can hold any primitive data type, whereas value is an arbitrary Python object.

The items in the dictionary are separated with the comma (,) and enclosed in the curly braces {}.

Consider the following example.

```
d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'}

# Printing dictionary
print (d)

# Accesing value using keys
print("1st name is "+d[1])
print("2nd name is "+ d[4])

print (d.keys())
print (d.values())
```

Output:

```
1st name is Jimmy
2nd name is mike
{1: 'Jimmy', 2: 'Alex', 3: 'john', 4: 'mike'}
dict_keys([1, 2, 3, 4])
dict_values(['Jimmy', 'Alex', 'john', 'mike'])
```

Boolean

Boolean type provides two built-in values, True and False. These values are used to determine the given statement true or false. It denotes by the class bool. True can be represented by any non-zero value or 'T' whereas false can be represented by the 0 or 'F'. Consider the following example.

```
# Python program to check the boolean type
print(type(True))
```

```
print(type(False))
    print(false)
```

Output:

```
<class 'bool'>
<class 'bool'>
NameError: name 'false' is not defined
```

6.Set

Python Set is the unordered collection of the data type. It is iterable, mutable(can modify after creation), and has unique elements. In set, the order of the elements is undefined; it may return the changed sequence of the element. The set is created by using a built-in function **set(),** or a sequence of elements is passed in the curly braces and separated by the comma. It can contain various types of values. Consider the following example.

```
# Creating Empty set
    >>>set1 = set()

set2 = {'James', 2, 3,'Python'}

    #Printing Set value
    print(set2)

    # Adding element to the set

    set2.add(10)
    print(set2)

    #Removing element from the set
    set2.remove(2)
```

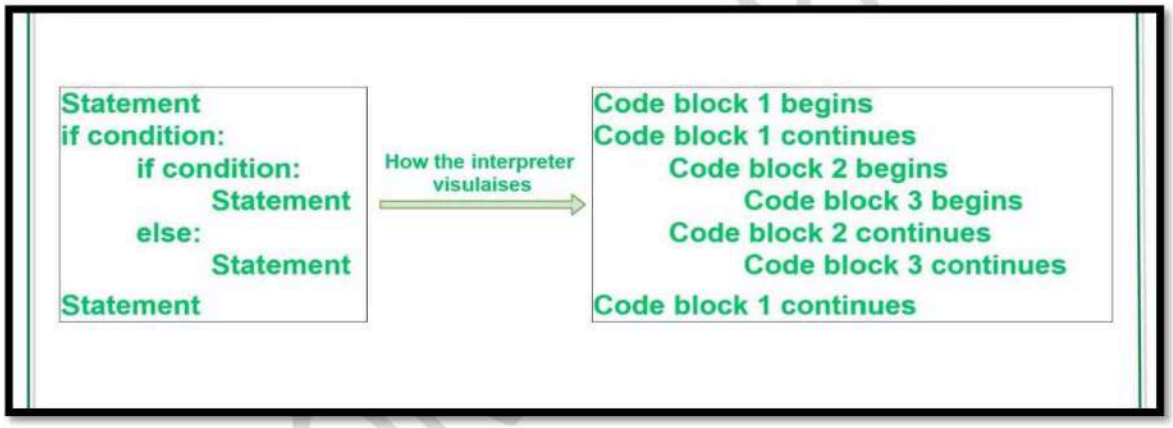
```
print(set2)
```

Output:

```
{3, 'Python', 'James', 2}
{'Python', 'James', 3, 2, 10}
{'Python', 'James', 3, 10}
```

Indentation:

- Indentation is a very important concept of Python because without properly indenting the Python code, you will end up seeing IndentationError and the code will not get compiled.
- Python indentation refers to adding white space before a statement to a particular block of code. In another word, all the statements with the same space to the right, belong to the same code block.



- Indentation refers to the spaces at the beginning of a code line.
- Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
- Python uses indentation to indicate a block of code.

Example:

```
if 5 > 2:
 print("Five is greater than two!")
if 5 > 2:
      print("Five is greater than two!")
```

Output:

```
File "demo_indentation2_error.py", line 3
    print("Five is greater than two!")
    ^
IndentationError: unexpected indent
```

Correct way

The number of spaces is up to you as a programmer, but it has to be at least one.

Example

```
if 5 > 2:
 print("Five is greater than two!")
if 5 > 2:
      print("Five is greater than two!")
```

Comments: Below are some of the most common uses for comments:

- Readability of the Code
- Restrict code execution
- Provide an overview of the program or project metadata
- To add resources to the code

Types of Comments in Python

In Python, there are 3 types of comments. They are described below:

Single-Line Comments

Single-line remarks in Python have shown to be effective for providing quick descriptions for parameters, function definitions, and expressions. A single-line comment of Python is the one that has a hashtag # at the beginning of it and continues until the finish of the line. If the comment continues to the next line, add a hashtag to the subsequent line and resume the conversation. Consider the accompanying code snippet, which shows how to use a single line comment:

Code

```
# This code is to show an example of a single-line comment
print( 'This statement does not have a hashtag before it' )
```

Output:

```
This statement does not have a hashtag before it
```

The following is the comment:

```
# This code is to show an example of a single-line comment
```

The Python compiler ignores this line.

Everything following the # is omitted. As a result, we may put the program mentioned above in one line as follows:

Code

```
print( 'This is not a comment' ) # this code is to show an example of a single-line comment
```

Output:

```
This is not a comment
```

This program's output will be identical to the example above. The computer overlooks all content following #.

Multi-Line Comments

Python does not provide the facility for multi-line comments. However, there are indeed many ways to create multi-line comments.

With Multiple Hashtags (#)

In Python, we may use hashtags (#) multiple times to construct multiple lines of comments. Every line with a (#) before it will be regarded as a single-line comment.

Code

```
    # it is a
# comment
# extending to multiple lines
```

In this case, each line is considered a comment, and they are all omitted.

Using String Literals

Because Python overlooks string expressions that aren't allocated to a variable, we can utilize them as comments.

Code

```
'it is a comment extending to multiple lines'
```

We can observe that on running this code, there will be no output; thus, we utilize the strings inside triple quotes(""") as multi-line comments.

Python Docstring

The strings enclosed in triple quotes that come immediately after the defined function are called Python docstring. It's designed to link documentation developed for Python modules, methods, classes, and functions

together. It's placed just beneath the function, module, or class to explain what they perform. The docstring is then readily accessible in Python using the __doc__ attribute.

Code

```
# Code to show how we use docstrings in Python
def add(x, y):
    """This function adds the values of x and y"""
```

Built-in Functions:- Python Built-in Functions

The Python built-in functions are defined as the functions whose functionality is pre-defined in Python. The python interpreter has several functions that are always present for use. These functions are known as Built-in Functions. There are several built-in functions in Python which are listed below:

Python abs() Function

The python **abs()** function is used to return the absolute value of a number. It takes only one argument, a number whose absolute value is to be returned. The argument can be an integer and floating-point number. If the argument is a complex number, then, abs() returns its magnitude.

Python abs() Function Example

```
# integer number
integer = -20
print('Absolute value of -40 is:', abs(integer))

# floating number
floating = -20.83
print('Absolute value of -40.83 is:', abs(floating))
```

Output:

```
Absolute value of -20 is: 20
Absolute value of -20.83 is: 20.83
```

Python all() Function

The python **all()** function accepts an iterable object (such as list, dictionary, etc.). It returns true if all items in passed iterable are true. Otherwise, it returns False. If the iterable object is empty, the all() function returns True.

Python all() Function Example

```
1. # all values true
2. k = [1, 3, 4, 6]
3. print(all(k))
4.
5. # all values false
6. k = [0, False]
7. print(all(k))
8.
9. # one false value
10. k = [1, 3, 7, 0]
11. print(all(k))
12.
13. # one true value
14. k = [0, False, 5]
15. print(all(k))
16.
17. # empty iterable
18. k = []
19. print(all(k))
```

Output:

```
True
False
False
False
True
```

Python bin() Function

The python **bin()** function is used to return the binary representation of a specified integer. A result always starts with the prefix 0b.

Python bin() Function Example

```
1. x = 10
2. y = bin(x)
3. print (y)
```

Output:

```
0b1010
```

Python bool()

The python **bool()** converts a value to boolean(True or False) using the standard truth testing procedure.

Python bool() Example

```
1. test1 = []
2. print(test1,'is',bool(test1))
3. test1 = [0]
4. print(test1,'is',bool(test1))
5. test1 = 0.0
6. print(test1,'is',bool(test1))
7. test1 = None
8. print(test1,'is',bool(test1))
9. test1 = True
10. print(test1,'is',bool(test1))
11. test1 = 'Easy string'
12. print(test1,'is',bool(test1))
```

Output:

```
[] is False
[0] is True
0.0 is False
None is False
True is True
Easy string is True
```

Python bytes()

The python **bytes()** in Python is used for returning a **bytes** object. It is an immutable version of the bytearray() function.

It can create empty bytes object of the specified size.

Python bytes() Example

```
1. string = "Hello World."
2. array = bytes(string, 'utf-8')
3. print(array)
```

Output:

```
b ' Hello World.'
```

Python callable() Function

A python **callable()** function in Python is something that can be called. This built-in function checks and returns true if the object passed appears to be callable, otherwise false.

Python callable() Function Example

```
1. x = 8
2. print(callable(x))
```

Output:

```
False
```

Python compile() Function

The python **compile()** function takes source code as input and returns a code object which can later be executed by exec() function.

Python compile() Function Example

```
1. # compile string source to code
2. code_str = 'x=5\ny=10\nprint("sum =",x+y)'
3. code = compile(code_str, 'sum.py', 'exec')
4. print(type(code))
5. exec(code)
6. exec(x)
```

Output:

```
<class 'code'>
sum = 15
```

Python exec() Function

The python **exec()** function is used for the dynamic execution of Python program which can either be a string or object code and it accepts large blocks of code, unlike the eval() function which only accepts a single expression.

Python exec() Function Example

```
1. x = 8
2. exec('print(x==8)')
3. exec('print(x+4)')
```

Output:

```
True
12
```

Python sum() Function

As the name says, python **sum()** function is used to get the sum of numbers of an iterable, i.e., list.

Python sum() Function Example

```
1. s = sum([1, 2,4 ])
2. print(s)
3.
4. s = sum([1, 2, 4], 10)
5. print(s)
```

Output:

```
7
17
```

Python any() Function

The python **any()** function returns true if any item in an iterable is true. Otherwise, it returns False.

Python any() Function Example

```
1. l = [4, 3, 2, 0]
2. print(any(l))
3.
```

```
l = [0, False]
print(any(l))

l = [0, False, 5]
print(any(l))

l = []
print(any(l))
```

Output:

```
True
False
True
False
```

Python ascii() Function

The python **ascii()** function returns a string containing a printable representation of an object and escapes the non-ASCII characters in the string using \x, \u or \U escapes.

Python ascii() Function Example

```
normalText = 'Python is interesting'
print(ascii(normalText))

otherText = 'Pythön is interesting'
print(ascii(otherText))

print('Pyth\xf6n is interesting')
```

Output:

```
'Python is interesting'
'Pyth\xf6n is interesting'
Pythön is interesting
```

Python bytearray()

The python **bytearray()** returns a bytearray object and can convert objects into bytearray objects, or create an empty bytearray object of the specified size.

Python bytearray() Example

```
string = "Python is a programming language."

# string with encoding 'utf-8'
arr = bytearray(string, 'utf-8')
print(arr)
```

Output:

```
bytearray(b'Python is a programming language.')
```

Python eval() Function

The python **eval()** function parses the expression passed to it and runs python expression(code) within the program.

Python eval() Function Example

```
x = 8
print(eval('x + 1'))
```

Output:

```
9
```

Python float()

The python **float()** function returns a floating-point number from a number or string.

Python float() Example

```
# for integers
print(float(9))

# for floats
print(float(8.19))

# for string floats
print(float("-24.27"))

# for string floats with whitespaces
print(float("    -17.19\n"))
```

```
13. # string float error
14. print(float("xyz"))
```

Output:

```
9.0
8.19
-24.27
-17.19
ValueError: could not convert string to float: 'xyz'
```

Python format() Function

The python **format()** function returns a formatted representation of the given value.

Python format() Function Example

```
1. # d, f and b are a type
2.
3. # integer
4. print(format(123, "d"))
5.
6. # float arguments
7. print(format(123.4567898, "f"))
8.
9. # binary format
10. print(format(12, "b"))
```

Output:

```
123
123.456790
1100
```

Python frozenset()

The python **frozenset()** function returns an immutable frozenset object initialized with elements from the given iterable.

Python frozenset() Example

```
1. # tuple of letters
2. letters = ('m', 'r', 'o', 't', 's')
3.
```

```
4. fSet = frozenset(letters)
5. print('Frozen set is:', fSet)
6. print('Empty frozen set is:', frozenset())
```

Output:

```
Frozen set is: frozenset({'o', 'm', 's', 'r', 't'})
Empty frozen set is: frozenset()
```

Python getattr() Function

The python **getattr()** function returns the value of a named attribute of an object. If it is not found, it returns the default value.

Python getattr() Function Example

```
1. class Details:
2.     age = 22
3.     name = "Phill"
4.
5. details = Details()
6. print('The age is:', getattr(details, "age"))
7. print('The age is:', details.age)
```

Output:

```
The age is: 22
The age is: 22
```

Python globals() Function

The python **globals()** function returns the dictionary of the current global symbol table.

A **Symbol table** is defined as a data structure which contains all the necessary information about the program. It includes variable names, methods, classes, etc.

Python globals() Function Example

```
1. age = 22
2.
3. globals()['age'] = 22
4. print('The age is:', age)
```

Output:

```
The age is: 22
```

Python hasattr() Function

The python **any()** function returns true if any item in an iterable is true, otherwise it returns False.

Python hasattr() Function Example

```
1. l = [4, 3, 2, 0]
2. print(any(l))
3.
4. l = [0, False]
5. print(any(l))
6.
7. l = [0, False, 5]
8. print(any(l))
9.
10. l = []
11. print(any(l))
```

Output:

```
True
False
True
False
```

Python iter() Function

The python **iter()** function is used to return an iterator object. It creates an object which can be iterated one element at a time.

Python iter() Function Example

```
1. # list of numbers
2. list = [1,2,3,4,5]
3.
4. listIter = iter(list)
5.
6. # prints '1'
7. print(next(listIter))
8.
```

```
9. # prints '2'
10. print(next(listIter))
11.
12. # prints '3'
13. print(next(listIter))
14.
15. # prints '4'
16. print(next(listIter))
17.
18. # prints '5'
19. print(next(listIter))
```

Output:

```
1
2
3
4
5
```

Python len() Function

The python **len()** function is used to return the length (the number of items) of an object.

Python len() Function Example

```
1. strA = 'Python'
2. print(len(strA))
```

Output:

```
6
```

Python list()

The python **list()** creates a list in python.

Python list() Example

```
1. # empty list
2. print(list())
3.
4. # string
5. String = 'abcde'
6. print(list(String))
```

```
7.
8.  # tuple
9.  Tuple = (1,2,3,4,5)
10. print(list(Tuple))
11. # list
12. List = [1,2,3,4,5]
13. print(list(List))
```

Output:

```
[]
['a', 'b', 'c', 'd', 'e']
[1,2,3,4,5]
[1,2,3,4,5]
```

Python locals() Function

The python **locals()** method updates and returns the dictionary of the current local symbol table.

A **Symbol table** is defined as a data structure which contains all the necessary information about the program. It includes variable names, methods, classes, etc.

Python locals() Function Example

```
1. def localsAbsent():
2.     return locals()
3.
4. def localsPresent():
5.     present = True
6.     return locals()
7.
8. print('localsNotPresent:', localsAbsent())
9. print('localsPresent:', localsPresent())
```

Output:

```
localsAbsent: {}
localsPresent: {'present': True}
```

Python map() Function

The python **map()** function is used to return a list of results after applying a given function to each item of an iterable(list, tuple etc.).

Python map() Function Example

```
1. def calculateAddition(n):
2.   return n+n
3.
4. numbers = (1, 2, 3, 4)
5. result = map(calculateAddition, numbers)
6. print(result)
7.
8. # converting map object to set
9. numbersAddition = set(result)
10. print(numbersAddition)
```

Output:

```
<map object at 0x7fb04a6bec18>
{8, 2, 4, 6}
```

Python memoryview() Function

The python **memoryview()** function returns a memoryview object of the given argument.

Python memoryview () Function Example

```
1. #A random bytearray
2. randomByteArray = bytearray('ABC', 'utf-8')
3.
4. mv = memoryview(randomByteArray)
5.
6. # access the memory view's zeroth index
7. print(mv[0])
8.
9. # It create byte from memory view
10. print(bytes(mv[0:2]))
11.
12. # It create list from memory view
13. print(list(mv[0:3]))
```

Output:

```
65
b'AB'
```

```
[65, 66, 67]
```

Python object()

The python **object()** returns an empty object. It is a base for all the classes and holds the built-in properties and methods which are default for all the classes.

Python object() Example

```
1. python = object()
2.
3. print(type(python))
4. print(dir(python))
```

Output:

```
<class 'object'>
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__']
```

Python open() Function

The python **open()** function opens the file and returns a corresponding file object.

Python open() Function Example

```
1. # opens python.text file of the current directory
2. f = open("python.txt")
3. # specifying full path
4. f = open("C:/Python33/README.txt")
```

Output:

```
Since the mode is omitted, the file is opened in 'r' mode; opens for reading.
```

Python chr() Function

Python **chr()** function is used to get a string representing a character which points to a Unicode code integer. For example, chr(97) returns the string 'a'. This function takes an integer argument and throws an error if it exceeds the specified range. The standard range of the argument is from 0 to 1,114,111.

Python chr() Function Example

```
# Calling function
result = chr(102) # It returns string representation of a char
result2 = chr(112)
# Displaying result
print(result)
print(result2)
# Verify, is it string type?
print("is it string type:", type(result) is str)
```

Output:

```
ValueError: chr() arg not in range(0x110000)
```

Python complex()

Python **complex()** function is used to convert numbers or string into a complex number. This method takes two optional parameters and returns a complex number. The first parameter is called a real and second as imaginary parts.

Python complex() Example

```
# Python complex() function example
# Calling function
a = complex(1) # Passing single parameter
b = complex(1,2) # Passing both parameters
# Displaying result
print(a)
print(b)
```

Output:

```
(1.5+0j)
(1.5+2.2j)
```

Python delattr() Function

Python **delattr()** function is used to delete an attribute from a class. It takes two parameters, first is an object of the class and second is an attribute which we want to delete. After deleting the attribute, it no longer available in the class and throws an error if try to call it using the class object.

Python delattr() Function Example

```
class Student:
```

```
    id = 101
    name = "Pranshu"
    email = "pranshu@abc.com"
# Declaring function
    def getinfo(self):
        print(self.id, self.name, self.email)
s = Student()
s.getinfo()
delattr(Student,'course') # Removing attribute which is not available
s.getinfo() # error: throws an error
```

Output:

```
101 Pranshu pranshu@abc.com
AttributeError: course
```

Python dir() Function

Python **dir()** function returns the list of names in the current local scope. If the object on which method is called has a method named __dir__(), this method will be called and must return the list of attributes. It takes a single object type argument.

Python dir() Function Example

```
# Calling function
att = dir()
# Displaying result
print(att)
```

Output:

```
['__annotations__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__']
```

Python divmod() Function

Python **divmod()** function is used to get remainder and quotient of two numbers. This function takes two numeric arguments and returns a tuple. Both arguments are required and numeric

Python divmod() Function Example

```
# Python divmod() function example
# Calling function
```

```
3. result = divmod(10,2)
4. # Displaying result
5. print(result)
```

Output:

```
(5, 0)
```

Python enumerate() Function

Python **enumerate()** function returns an enumerated object. It takes two parameters, first is a sequence of elements and the second is the start index of the sequence. We can get the elements in sequence either through a loop or next() method.

Python enumerate() Function Example

```
1. # Calling function
2. result = enumerate([1,2,3])
3. # Displaying result
4. print(result)
5. print(list(result))
```

Output:

```
<enumerate object at 0x7ff641093d80>
[(0, 1), (1, 2), (2, 3)]
```

Python dict()

Python **dict()** function is a constructor which creates a dictionary. Python dictionary provides three different constructors to create a dictionary:

- If no argument is passed, it creates an empty dictionary.
- If a positional argument is given, a dictionary is created with the same key-value pairs. Otherwise, pass an iterable object.
- If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument.

Python dict() Example

```
1. # Calling function
2. result = dict() # returns an empty dictionary
3. result2 = dict(a=1,b=2)
```

```
4. # Displaying result
5. print(result)
6. print(result2)
```

Output:

```
{}
{'a': 1, 'b': 2}
```

Python filter() Function

Python **filter()** function is used to get filtered elements. This function takes two arguments, first is a function and the second is iterable. The filter function returns a sequence of those elements of iterable object for which function returns **true value**.

The first argument can be **none**, if the function is not available and returns only elements that are **true**.

Python filter() Function Example

```
1. # Python filter() function example
2. def filterdata(x):
3.     if x>5:
4.         return x
5. # Calling function
6. result = filter(filterdata,(1,2,6))
7. # Displaying result
8. print(list(result))
```

Output:

```
[6]
```

Python hash() Function

Python **hash()** function is used to get the hash value of an object. Python calculates the hash value by using the hash algorithm. The hash values are integers and used to compare dictionary keys during a dictionary lookup. We can hash only the types which are given below:

Hashable types: * bool * int * long * float * string * Unicode * tuple * code object.

Python hash() Function Example

```
1. # Calling function
2. result = hash(21) # integer value
```

```
result2 = hash(22.2) # decimal value
# Displaying result
print(result)
print(result2)
```

Output:

```
21
461168601842737174
```

Python help() Function

Python **help()** function is used to get help related to the object passed during the call. It takes an optional parameter and returns help information. If no argument is given, it shows the Python help console. It internally calls python's help function.

Python help() Function Example

```
# Calling function
info = help() # No argument
# Displaying result
print(info)
```

Output:

```
Welcome to Python 3.5's help utility!
```

Python min() Function

Python **min()** function is used to get the smallest element from the collection. This function takes two arguments, first is a collection of elements and second is key, and returns the smallest element from the collection.

Python min() Function Example

```
# Calling function
small = min(2225,325,2025) # returns smallest element
small2 = min(1000.25,2025.35,5625.36,10052.50)
# Displaying result
print(small)
print(small2)
```

Output:

```
325
1000.25
```

Python set() Function

In python, a set is a built-in class, and this function is a constructor of this class. It is used to create a new set using elements passed during the call. It takes an iterable object as an argument and returns a new set object.

Python set() Function Example

```
1. # Calling function
2. result = set() # empty set
3. result2 = set('12')
4. result3 = set('javatpoint')
5. # Displaying result
6. print(result)
7. print(result2)
8. print(result3)
```

Output:

```
set()
{'1', '2'}
{'a', 'n', 'v', 't', 'j', 'p', 'i', 'o'}
```

Python hex() Function

Python **hex()** function is used to generate hex value of an integer argument. It takes an integer argument and returns an integer converted into a hexadecimal string. In case, we want to get a hexadecimal value of a float, then use float.hex() function.

Python hex() Function Example

```
1. # Calling function
2. result = hex(1)
3. # integer value
4. result2 = hex(342)
5. # Displaying result
6. print(result)
7. print(result2)
```

Output:

```
0x1
0x156
```

Python id() Function

Python **id()** function returns the identity of an object. This is an integer which is guaranteed to be unique. This function takes an argument as an object and returns a unique integer number which represents identity. Two objects with non-overlapping lifetimes may have the same id() value.

Python id() Function Example

```
1. # Calling function
2. val = id("Javatpoint") # string object
3. val2 = id(1200) # integer object
4. val3 = id([25,336,95,236,92,3225]) # List object
5. # Displaying result
6. print(val)
7. print(val2)
8. print(val3)
```

Output:

```
139963782059696
139963805666864
139963781994504
```

Python setattr() Function

Python **setattr()** function is used to set a value to the object's attribute. It takes three arguments, i.e., an object, a string, and an arbitrary value, and returns none. It is helpful when we want to add a new attribute to an object and set a value to it.

Python setattr() Function Example

```
1. class Student:
2.     id = 0
3.     name = ""
4.
5.     def __init__(self, id, name):
6.         self.id = id
7.         self.name = name
8.
9. student = Student(102,"Sohan")
10. print(student.id)
11. print(student.name)
12. #print(student.email) product error
13. setattr(student, 'email','sohan@abc.com') # adding new attribute
```

14. **print**(student.email)
Output:

```
102
Sohan
sohan@abc.com
```

Python slice() Function

Python **slice()** function is used to get a slice of elements from the collection of elements. Python provides two overloaded slice functions. The first function takes a single argument while the second function takes three arguments and returns a slice object. This slice object can be used to get a subsection of the collection.

Python slice() Function Example

```
1. # Calling function
2. result = slice(5) # returns slice object
3. result2 = slice(0,5,3) # returns slice object
4. # Displaying result
5. print(result)
6. print(result2)
```
Output:

```
slice(None, 5, None)
slice(0, 5, 3)
```

Python sorted() Function

Python **sorted()** function is used to sort elements. By default, it sorts elements in an ascending order but can be sorted in descending also. It takes four arguments and returns a collection in sorted order. In the case of a dictionary, it sorts only keys, not values.

Python sorted() Function Example

```
1. str = "javatpoint" # declaring string
2. # Calling function
3. sorted1 = sorted(str) # sorting string
4. # Displaying result
5. print(sorted1)
```
Output:

```
['a', 'a', 'i', 'j', 'n', 'o', 'p', 't', 't', 'v']
```

Python next() Function

Python **next()** function is used to fetch next item from the collection. It takes two arguments, i.e., an iterator and a default value, and returns an element.

This method calls on iterator and throws an error if no item is present. To avoid the error, we can set a default value.

Python next() Function Example

```
1. number = iter([256, 32, 82]) # Creating iterator
2. # Calling function
3. item = next(number)
4. # Displaying result
5. print(item)
6. # second item
7. item = next(number)
8. print(item)
9. # third item
10. item = next(number)
11. print(item)
```

Output:

```
256
32
82
```

Python input() Function

Python **input()** function is used to get an input from the user. It prompts for the user input and reads a line. After reading data, it converts it into a string and returns it. It throws an error **EOFError** if EOF is read.

Python input() Function Example

```
1. # Calling function
2. val = input("Enter a value: ")
3. # Displaying result
4. print("You entered:",val)
```

Output:

```
Enter a value: 45
You entered: 45
```

Python int() Function

Python **int()** function is used to get an integer value. It returns an expression converted into an integer number. If the argument is a floating-point, the conversion truncates the number. If the argument is outside the integer range, then it converts the number into a long type.

If the number is not a number or if a base is given, the number must be a string.

Python int() Function Example

```
# Calling function
val = int(10) # integer value
val2 = int(10.52) # float value
val3 = int('10') # string value
# Displaying result
print("integer values :",val, val2, val3)
```

Output:

```
integer values : 10 10 10
```

Python isinstance() Function

Python **isinstance()** function is used to check whether the given object is an instance of that class. If the object belongs to the class, it returns true. Otherwise returns False. It also returns true if the class is a subclass.

The **isinstance()** function takes two arguments, i.e., object and classinfo, and then it returns either True or False.

Python isinstance() function Example

```
class Student:
    id = 101
    name = "John"
    def __init__(self, id, name):
        self.id=id
        self.name=name

student = Student(1010,"John")
lst = [12,34,5,6,767]
# Calling function
print(isinstance(student, Student)) # isinstance of Student class
print(isinstance(lst, Student))
```

Output:

```
True
False
```

Python oct() Function

Python **oct()** function is used to get an octal value of an integer number. This method takes an argument and returns an integer converted into an octal string. It throws an error **TypeError**, if argument type is other than an integer.

Python oct() function Example

```
# Calling function
val = oct(10)
# Displaying result
print("Octal value of 10:",val)
```

Output:

```
Octal value of 10: 0o12
```

Python ord() Function

The python **ord()** function returns an integer representing Unicode code point for the given Unicode character.

Python ord() function Example

```
# Code point of an integer
print(ord('8'))

# Code point of an alphabet
print(ord('R'))

# Code point of a character
print(ord('&'))
```

Output:

```
56
82
38
```

Python pow() Function

The python **pow()** function is used to compute the power of a number. It returns x to the power of y. If the third argument(z) is given, it returns x to the power of y modulus z, i.e. (x, y) % z.

Python pow() function Example

```
# positive x, positive y (x**y)
print(pow(4, 2))

# negative x, positive y
print(pow(-4, 2))

# positive x, negative y (x**-y)
print(pow(4, -2))

# negative x, negative y
print(pow(-4, -2))
```

Output:

```
16
16
0.0625
0.0625
```

Python print() Function

The python **print()** function prints the given object to the screen or other standard output devices.

Python print() function Example

```
print("Python is programming language.")

x = 7
# Two objects passed
print("x =", x)

y = x
# Three objects passed
print('x =', x, '= y')
```

Output:

```
Python is programming language.
```

```
x = 7
x = 7 = y
```

Python range() Function

The python **range()** function returns an immutable sequence of numbers starting from 0 by default, increments by 1 (by default) and ends at a specified number.

Python range() function Example

```
1. # empty range
2. print(list(range(0)))
3.
4. # using the range(stop)
5. print(list(range(4)))
6.
7. # using the range(start, stop)
8. print(list(range(1,7 )))
```

Output:

```
[]
[0, 1, 2, 3]
[1, 2, 3, 4, 5, 6]
```

Python reversed() Function

The python **reversed()** function returns the reversed iterator of the given sequence.

Python reversed() function Example

```
1. # for string
2. String = 'Java'
3. print(list(reversed(String)))
4.
5. # for tuple
6. Tuple = ('J', 'a', 'v', 'a')
7. print(list(reversed(Tuple)))
8.
9. # for range
10. Range = range(8, 12)
11. print(list(reversed(Range)))
```

```
12.
13. # for list
14. List = [1, 2, 7, 5]
15. print(list(reversed(List)))
```

Output:

```
['a', 'v', 'a', 'J']
['a', 'v', 'a', 'J']
[11, 10, 9, 8]
[5, 7, 2, 1]
```

Python round() Function

The python **round()** function rounds off the digits of a number and returns the floating point number.

Python round() Function Example

```
1. # for integers
2. print(round(10))
3.
4. # for floating point
5. print(round(10.8))
6.
7. # even choice
8. print(round(6.6))
```

Output:

```
10
11
7
```

Python issubclass() Function

The python **issubclass()** function returns true if object argument(first argument) is a subclass of second class(second argument).

Python issubclass() Function Example

```
1. class Rectangle:
2.   def __init__(rectangleType):
3.     print('Rectangle is a ', rectangleType)
4.
```

```
5. class Square(Rectangle):
6.   def __init__(self):
7.     Rectangle.__init__('square')
8.
9. print(issubclass(Square, Rectangle))
10. print(issubclass(Square, list))
11. print(issubclass(Square, (list, Rectangle)))
12. print(issubclass(Rectangle, (list, Rectangle)))
```

Output:

```
True
False
True
True
```

Python str

The python **str()** converts a specified value into a string.

Python str() Function Example

```
1. str('4')
```

Output:

```
'4'
```

Python tuple() Function

The python **tuple()** function is used to create a tuple object.

Python tuple() Function Example

```
1. t1 = tuple()
2. print('t1=', t1)
3.
4. # creating a tuple from a list
5. t2 = tuple([1, 6, 9])
6. print('t2=', t2)
7.
8. # creating a tuple from a string
9. t1 = tuple('Java')
10. print('t1=',t1)
```

```
11.
12. # creating a tuple from a dictionary
13. t1 = tuple({4: 'four', 5: 'five'})
14. print('t1=',t1)
```

Output:

```
t1= ()
t2= (1, 6, 9)
t1= ('J', 'a', 'v', 'a')
t1= (4, 5)
```

Python type()

The python **type()** returns the type of the specified object if a single argument is passed to the type() built in function. If three arguments are passed, then it returns a new type object.

Python type() Function Example

```
List = [4, 5]
1. print(type(List))
2.
3. Dict = {4: 'four', 5: 'five'}
4. print(type(Dict))
5.
6. class Python:
7.     a = 0
8.
9. InstanceOfPython = Python()
10. print(type(InstanceOfPython))
```

Output:

```
<class 'list'>
<class 'dict'>
<class '__main__.Python'>
```

Python vars() function

The python **vars()** function returns the __dict__ attribute of the given object.

Python vars() Function Example

```
1. class Python:
```

```
2.  def __init__(self, x = 7, y = 9):
3.    self.x = x
4.    self.y = y
5.
6.  InstanceOfPython = Python()
7.  print(vars(InstanceOfPython))
```

Output:

```
{'y': 9, 'x': 7}
```

Python zip() Function

The python **zip()** Function returns a zip object, which maps a similar index of multiple containers. It takes iterables (can be zero or more), makes it an iterator that aggregates the elements based on iterables passed, and returns an iterator of tuples.

Python zip() Function Example

```
1.  numList = [4,5, 6]
2.  strList = ['four', 'five', 'six']
3.
4.  # No iterables are passed
5.  result = zip()
6.
7.  # Converting itertor to list
8.  resultList = list(result)
9.  print(resultList)
10.
11. # Two iterables are passed
12. result = zip(numList, strList)
13.
14. # Converting itertor to set
15. resultSet = set(result)
16. print(resultSet)
```

Output:

```
[]
{(5, 'five'), (4, 'four'), (6, 'six')}
```

Console Input and Console Output:

- The purpose of a computer is to process data and return results.
- It means that first of all, we should provide data to the computer. The data given to the computer is called input. The results returned by the computer are called output. So, we cansay that a computer takes input, processes that input and produces the output.
- To provide input to a computer, Python provides some statements which are called **Input statements**. Similarly, to display the output, there are **Output statements** available in Python

We should use some logic to convert the input into output.Output statements

- To display output or results, Python provides the print() function.
- This function can be used in different formats.

print() Statement :-When the print() function is called simply, it will throw the cursor to the next line. It means that a blank line will be displayed.

The print(formatted string) Statement :-The output displayed by the print() function can be formatted as we like. The special operator „%“ (percent) can be used for this purpose. It joins a string with a variable or value in the following format:

print (“formatted string”% (variables list))

In the “formatted string”, we can use %i or %d to represent decimal integer numbers. We can use %f to represent float values. Similarly, we can use %s to represent strings. See the example below:

```
x=10
print('value= %i'% x)
value= 10
```

As seen above, to display a single variable (i.e. „x“), we need not wrap it inside parentheses.

Input Statements

To accept input from keyboard, Python provides the input () function. This function takes a value from the keyboard and returns it as a string. For example

```
str = input('Enter
your name:') Enter
your name: Raj
print(str)
Raj
```

Type conversion:

Type conversion in python

- Type conversion refers to changing a given object from one data type to another datatype.
- **Type conversion in Python** is used to convert values from one data type to another, either automatically by the Python interpreter or manually by the programmer using in-built functions.
- Note that not all data types can be converted into each other. For example, there's no waya string consisting of English letter alphabets can be converted into an integer.

Implicit Type Conversion

- Implicit type conversion is when the Python interpreter converts an object from one datatype to another on its own without the need for the programmer to do it manually.
- The smaller data type is covered into higher data type to prevent any loss of data during the runtime. Since the conversion is automatic, we do not require to use any function explicitly in the code.

Example:

```
english, science, maths, sst, it = 95, 92,
97, 90, 98 total = english + science +
maths + sst + it print("Total marks:", total,
"is of type", type(total)) percentage = total
```

/ 5

print("Percentage:", percentage, "is of type", type(percentage))

Output:

Total marks: 472 is of type

<class 'int'> Percentage: 94.4 is

of type <class 'float'>

Explicit Type Conversion

- Suppose we need to convert a value from a higher to a lower data type. Implicit typeconversion cannot be used here.
- Explicit type conversion means that the programmer manually changes the data type of an object to another with the help of in-built *Python* functions like str() for converting tostring type and int() for converting to the integer type.

Note: In explicit type conversion, loss of data may occur as we enforce a given value to alower data type. For example, converting a float value to int will round off the decimal places in the output.

Example:

Conversion from string to list

```
Value="h
ello"
Print(valu
e)
Value1=li
st(value)
Print(valu
e1)
```

Output

```
Value:hel
lo
Value1:[„h“,“e“,“l“,“l“,“o“]
```

PYTHON LIBRARY

- A Python library is a collection of related modules.
- It contains bundles of code that can be used repeatedly in different programs.
- It makes Python Programming simpler and convenient for the programmer. As we don't need to write the same code again and again for different programs.
- Modules on the other hand refer to any python file saved with the **.py** extension. Modules often contain code such as functions, classes and statements that can be imported and used within other programs.

Python libraries are pre-written sets of code that provide various functionalities to developers.

These libraries contain modules and packages that can be imported into your python programs to extend their capabilities.

Python libraries are created and maintained by the Python community and cover a wide range of areas, including data analysis, web development, machine learning, scientific computing, and more.

To import a library in Python, you can use the import statement. Here are the steps to import a library:

- **Identify the library:** Determine the name of the library you want to import. Libraries in Python often come with built-in functionality or provide additional features for specific purposes.
- **Import the library:** Use the import statement followed by the name of the library. You can place the import statement at the beginning of your Python script or module.
- **Utilize the library's features:** Once the library is imported, you can access its functions, classes, or other elements using the library's name followed by a dot (.) notation.

Importing library with example:

Matplotlib: This library is responsible for plotting numerical data. And that's why it is used in data analysis. It is also an open-source library and plots high-defined figures like pie charts, histograms, scatterplots, graphs, etc.

Pandas: It is an open-source machine learning library that provides flexible high-level data structures and a variety of analysis tools. It eases data analysis, data manipulation, and cleaning of data. Pandas support operations like Sorting, Re-indexing, Iteration, Concatenation, Conversion of data, Visualizations, Aggregations, etc.

Numpy: The name "Numpy" stands for "Numerical Python". It is the commonly used library. It is a popular machine learning library that supports large matrices and multi-dimensional data. It consists of in-built mathematical functions for easy computations..

SciPy: The name SciPy stands for "Scientific Python". It is an open- source library used for high-level scientific computations. This library is built over an extension of Numpy. It works with Numpy to handle complex computations. While Numpy allows sorting and indexing of array data, the numerical data code is stored in SciPy.

- **Scrapy:** It is an open-source library that is used for extracting data from websites. It provides very fast web crawling and high-level screen scraping. It can also be used for data mining and automated testing of data.
- **PyGame:** This library provides an easy interface to the Standard Directmedia Library (SDL) platform-independent graphics, audio, and input libraries. It is used for developing video games using computer graphics and audio libraries along with Python programming language.

Here's an example that demonstrates the steps to import and use the math library, which provides various mathematical functions:

```
# Step 1: Identify the library
# The math library provides mathematical functions # Step 2:
Import the library
import math
# Step 3: Utilize the library's features
# Calculate the square root of a number

num = 16
sqrt = math.sqrt(num)
print("The square root of", num, "is", sqrt)#
Calculate the value of pi
pi = math.pi
print("The value of pi is", pi)
```

- In this example, the math library is imported using the import statement. After importing, we can access the library's functions and constants using the math. prefix.
- We calculate the square root of a number (math.sqrt()) and assign it to the variable sqrt. We then print the result. Additionally, we access the value of pi (math.pi) and print it.
- By importing the math library, we gain access to its features and can use them in our program.

CHAPTER-03

Control Flow

- A program's **control flow** is the order in which the program's code executes.
- The control flow of a Python program is regulated by conditional statements, loops, and function calls.

Python has three types of control structures:

- **Sequential** - default mode
- **Selection** - used for decisions and branching
- **Repetition** - used for looping, i.e., repeating a piece of code multiple times.

1. Sequential

Sequential statements are a set of statements whose execution process happens in a sequence. The problem with sequential statements is that if the logic has broken in any one of the lines, then the complete source code execution will break.

```
## This is a Sequential statement
a=20
b=10
c=a-b
print("Subtraction is : ",c)
```

Output

```
Subtraction is : 10
```

2. Selection/Decision control statements

- In Python, the selection statements are also known as *Decision control statements* or *branching statements*.
- The selection statement allows a program to test several conditions and execute instructions based on which condition is true.

Some Decision Control Statements are:

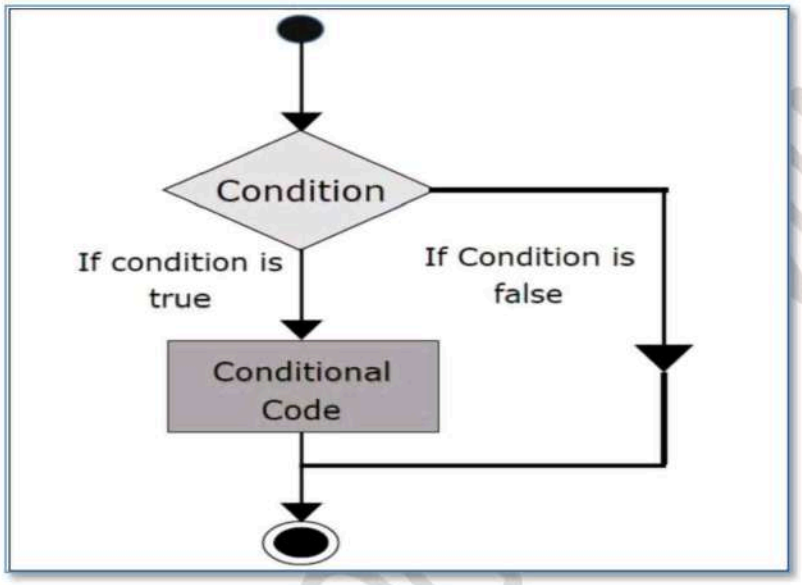
- Simple if
- if-else
- nested if
- if-elif-else

Simple if: *If statements* are control flow statements that help us to run a particular code, but only when a certain condition is met or satisfied. A *simple if* only has one condition to check.

- In control statements, The if statement is the simplest form. It takes a condition and evaluates to either True or False.
- If the condition is True, then the True block of code will be executed, and if the condition is False, then the block of code is skipped, and The controller moves to the next line

Syntax of the if statement

if condition:
statemen1t
statement 2
statement n




```
n = 10
if n % 2 == 0:
   print("n  is an even number")
```

Output: n is an even number

if-else: The *if-else statement* evaluates the condition and will execute the body of if if the test condition is True, but if the condition is False, then the body of else is executed.

- The if-else statement checks the condition and executes the if block of code when the condition is True, and if the condition is False, it will execute the else block of code.
- If the condition is True, then statement 1 will be executed If the condition is False, statement 2 will be executed.

Syntax of the if-else statement

```
if condition:
   statement 1
else:
   statement 2
```

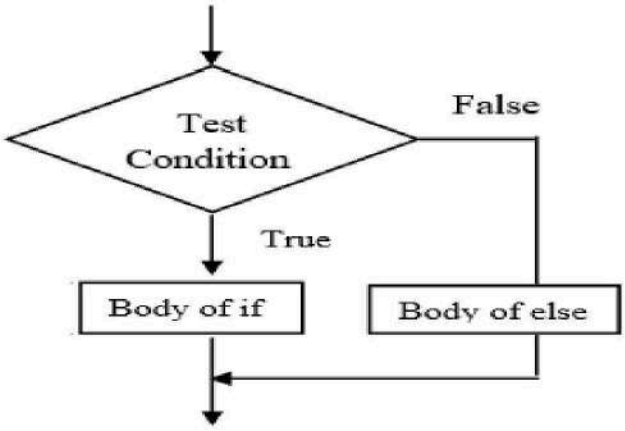


Fig. Flowchart of if-else

```
password = input('Enter  password ')

if password == "PYnative@#29":
   print("Correct  password")
else:
   print("Incorrect  Password")
```

Output 1:

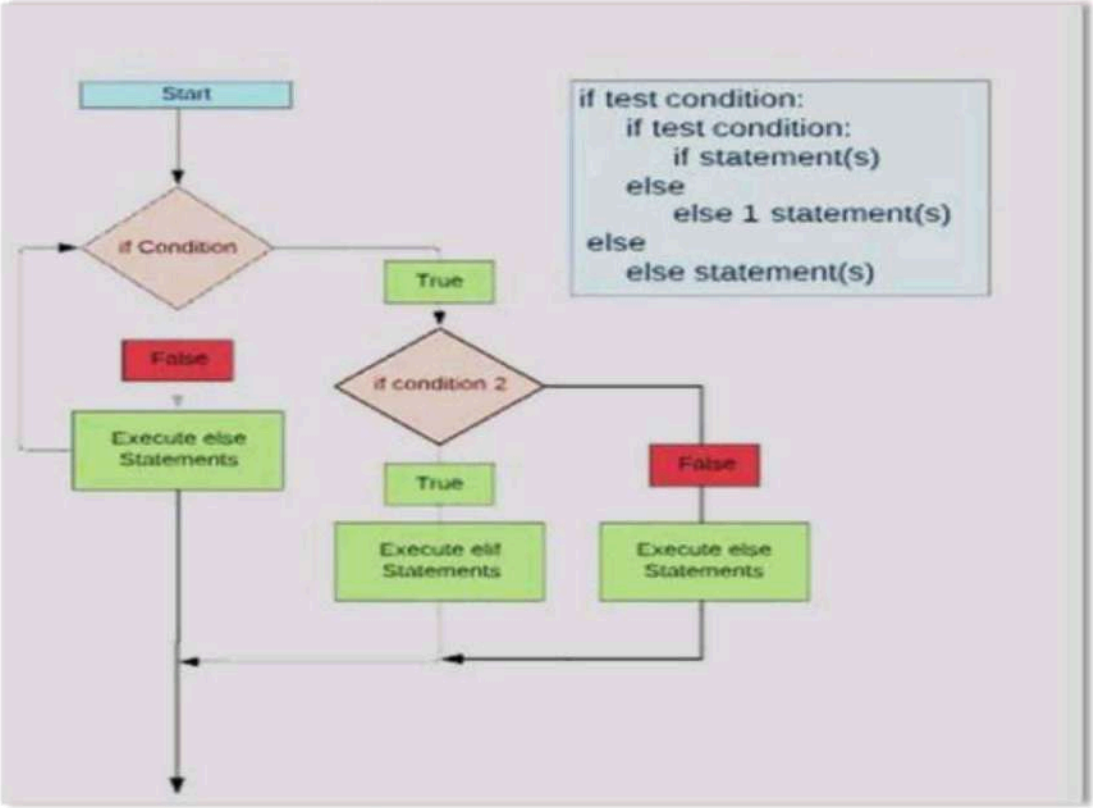
```
Enter  password PYnative@#29

Correct password
```

Output 2:

```
Enter  password PYnative

Incorrect  Password
```

nested if: *Nested if statements* are an if statement inside another if statement.

- In Python, the nested if-else statement is an if statement inside another if-else statement. It is allowed in Python to put any number of if statements in another if statement.
- Indentation is the only way to differentiate the level of nesting. The nested if-else is useful when we want to make a series of decisions.




```
a = 5
b = 10
c = 15
if a > b:
  if a > c:
    print("a  value  is big")
  else:
     print("c  value  is big")
elif b > c:
   print("b  value  is big")
else:
    print("c  is big")
```

Output 1:

```
Enter first number 56
Enter second number 15
56 is greater than 15
```

Output 2:

```
Enter first number 29
Enter second number 78
29 is smaller than 78
```

if-elif-else: The *if-elif-else statement* is used to conditionally execute a statement or a block of statements.

- the if-elif-else condition statement has an elif blocks to chain multiple conditions one after another. This is useful when you need to check multiple conditions.
- With the help of if-elif-else we can make a tricky decision. The elif statement checks multiple conditions one by one and if the condition fulfills, then executes that code.

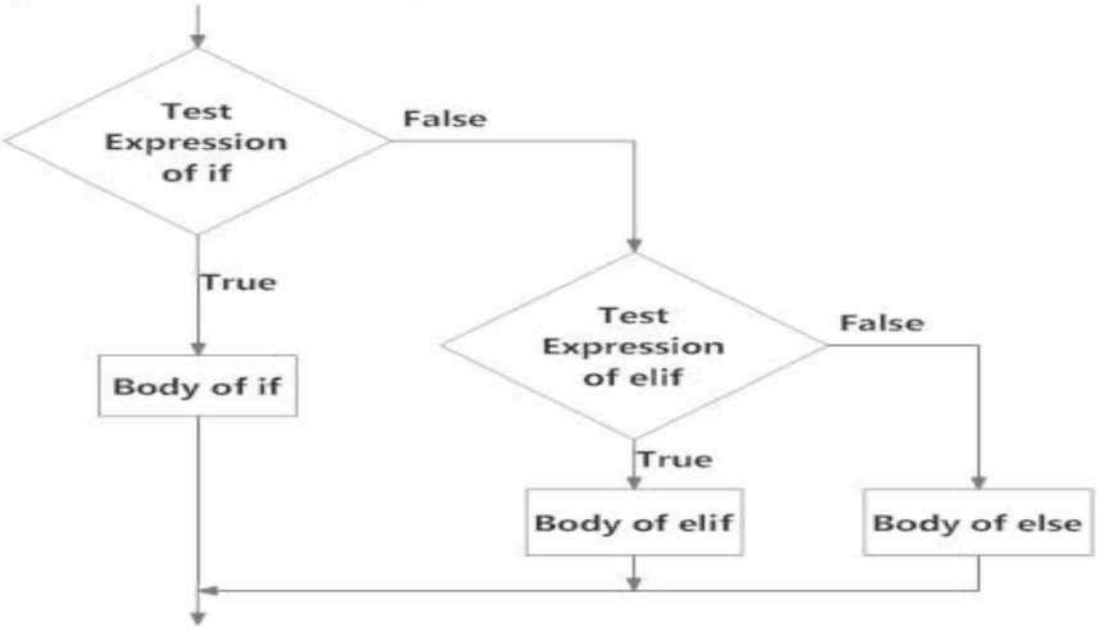


Fig: Operation of if...elif...else statement

Syntax of the if-elif-else statement:

```
if condition-1:
    statement 1
elif condition-2:
    stetement 2
elif condition-3:
    stetement 3
    ...
else:
    statement
```

```
x = 15
y = 12
if x == y:
  print("Both  are Equal")
elif x > y:
    print("x  is greater than y")
else:
```

```
print("x is smaller than y")
```

Output:

x is greater than y

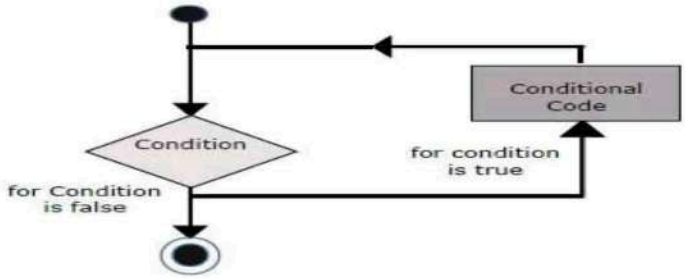
Repetition

A **repetition statement** is used to repeat a group(block) of programming instructions.

In Python, we generally have two loops/repetitive statements:

- for loop
- while loop

for loop: A *for loop* is used to iterate over a sequence that is either a list, tuple, dictionary, or a set. We can execute a set of statements once for each item in a list, tuple, or dictionary.



syntax of for loop:

```
for element in sequence:
    body of for loop
```

Example to display first ten numbers using for loop

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

Output

```
1
2
3
4
5
6
7
8
9
10
```

While loop in Python

- In Python, The while loop statement repeatedly executes a code block while a particular condition is true.
- In a while-loop, every time the condition is checked at the beginning of the loop, and if it is true, then the loop's body gets executed. When the condition became False, the controller comes out of the block.

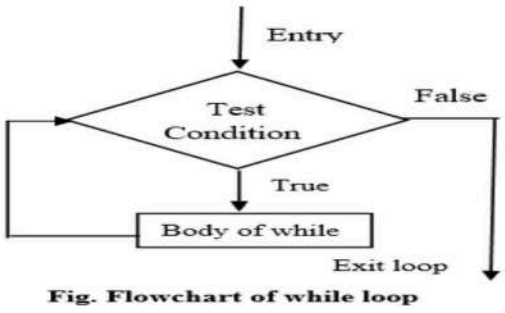


Fig. Flowchart of while loop

Syntax of while-loop

```
while condition :
  body of while loop
```

Example to calculate the sum of N Natural numbers

Lab program 3

Control Statements

Within an imperative programming language, a control flow statement is a statement that results in a choice being made as to which of two or more paths to follow.

There are three control staments in python

1. Break statement.
2. Continue statement.
3. Pass statement.

1.Break Statement

The break statement is used inside the loop to exit out of the loop. It is useful when we want to terminate the loop as soon as the condition is fulfilled instead of doing the remaining iterations. It reduces execution time. Whenever the controller encountered a break statement, it comes out of that loop immediately

Example of using a break statement

```
for num in range(10):
  if num > 5:
    print("stop processing.")
```

```
    break
  print(num)
```

Output

```
0

1

2

3

4

5

stop processing.
```

2.Continue statement

The continue statement is used to skip the current iteration and continue with the next iteration.

Let's see how to skip a for a loop iteration if the number is 5 and continue executing the body of the loop for other numbers.

Example of a continue statement

```
for num in range(3, 8):
  if num == 5:
    continue
  else:
    print(num)
```

Output

```
3
4
6
7
```

3.Pass statement

- The pass is the keyword In Python, which won't do anything. Sometimes there is a situation in programming where we need to define a syntactically empty block. We can define that block with the pass keyword.
- A pass statement is a Python null statement. When the interpreter finds a pass statement in the program, it returns no operation. Nothing happens when the pass statement is executed.
- It is useful in a situation where we are implementing new methods or also in exception handling. It plays a role like a placeholder.

Example

```
months = ['January', 'June', 'March', 'April']
for mon in months:
   pass
print(months)
```

Output

```
['January', 'June', 'March', 'April']
```

Range () and exit() functions:

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.

It creates the sequence of numbers from start to stop -1 . For example, range(5) will produce [0, 1, 2, 3, 4] . The result contains numbers from 0 to up to 5 but not five.

There are three ways you can call range():

- range(stop) takes one argument.
- range(start, stop) takes two arguments.
- range(start, stop, step) takes three arguments.

➢ **range(stop)**

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

```
for i in range(3):
   print(i)
```

Output:

```
0
1
2
```

we have all the whole numbers from 0 up to but not including 3, the number you provided as the stop.

- **range(start, stop)**

When you call range() with two arguments, you get to decide not only where the series of numbers stops but also where it starts, so you don't have to start at 0 all the time. You can use range() to generate a series of numbers from *A* to *B* using a range(A, B). Let's find out how to generate a range starting at 1.

To calling range() with two arguments:

```
for i in range(1, 8):
    print(i)
```

Output:

```
1
2
3
4
5
6
7
```

All the whole numbers from 1 (the number you provided as the start) up to but not including 8 (the number you provided as the stop).

But if you add one more argument, then you'll be able to reproduce the output you got earlier when you were using the list named numbers_divisible_by_three.

- **range(start, stop, step)**

When you call range() with three arguments, you 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 you don't provide a step, then range() will automatically behave as if the step is 1.

```
for i in range(3, 16, 3):
    quotient = i / 3
    print(f"{i} divided by 3 is {int(quotient)}.")
```

Your output will look exactly like the output of the for-loop you saw earlier in this article, when you were using the list named numbers_divisible_by_three:

Now we will get the correct result:

```
3 divided by 3 is 1.
6 divided by 3 is 2.
9 divided by 3 is 3.
12 divided by 3 is 4.
15 divided by 3 is 5.
```

Note: step can be a positive number or a negative number, but it can't be 0.

```
range(1, 4, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: range() arg 3 must not be zero
```

Exit (): The exit() function in Python is used to exit or terminate the current running script or program. You can use it to stop the execution of the program at any point.

When the **exit()** function is called, the program will immediately stop running and exit.

The syntax of the exit()

```
exit([status])
```

- Here, `status` is an optional argument that represents the exit status of the program.
- The exit status is an integer value that indicates the reason for program termination.
- By convention, a status of 0 indicates successful execution, and any non-zero status indicates an error or abnormal termination.
- Python provides many exit functions using which a user can exit the python program.

Types of python exit function

Python mainly provides four commands using which you can exit from the program and stop the script execution at a certain time. The 4 types of exit commands are:-

- exit()
- quit()
- sys.exit()
- os._exit()

Python exit() function

The python exit() function is available in the site module. It is used for the exit or termination of a Python script. The exit command should not be used in production and can only be used for interpreters. Below is the sample python code with the exit function.

```
age = 45
if age < 50:
# exit the program
print(exit)
exit(0)
```

Python quit() function

In Python, the `quit()` function is equivalent to the `exit()` function. However, the quit() function is only available n python2 but not in python 3.Quit() function should only be used in the interpreter. Below is the sample python code which uses the quit() function.

```
age = 45
if age &amp;lt; 50:
# quit the program
print(quit)
quit()
```

The above code defines a variable age as 45 and then checks if the value of age is less than 50. If it is, it will print the value of the quit function and then call the quit() function to stop the program from running.

Python sys.exit() function

- In Python, the `sys.exit()` function is used to exit or terminate a Python script.

  The **sys.exit()** function can be used in production because it raises an `SystemExit` exception that causes the interpreter to exit.
- You can use the sys.exit() program with or without arguments. With an argument, the program will exit and return a successful exit code on 0, whereas with an argument, which is the exit code like:

  sys.exit(0) # success

  sys.exit(1) # failure

```
import sys
for i in range(1,5):
print(i)
```

```
if i == 3:
sys.exit()
```

- The above code will print the numbers from 1 to 3, and then the `sys.exit()` function will be called when the value of `i` is 3. This will cause the script to terminate and end the program with an exit code of 0, indicating a successful termination.

python os._exit() function

- In Python, the os._exit() function is used to immediately exit the current process without performing any cleanup or finalization tasks. The os._exit() does not call any cleanup handlers, close open files, or flush stdio buffers.
- Below is the sample python code using the os._exit() function

```
import os
for i in range(1,5):
print(i)
if i == 3:
os._exit(0)
```

After the number 3 is printed, the `os. exit(0)` function will be called, causing the script to immediately terminate and exit the process without printing the remaining numbers (4) in the range.

CHAPTER-04

Exception Handling: when a Python script encounters a situation that it cannot cope with, it raises an exception.

- An exception is a Python object that represents an error.
- When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Types of errors:

1. Syntax Errors

2. Runtime Errors

3. Logical Errors

1. Syntax Errors

- The Python Syntax Error occurs when the interpreter encounters invalid syntax in code.
- When Python code is executed, the interpreter parses it to convert it into bytecode.
- If the interpreter finds any invalid syntax during the parsing stage, a Syntax Error is thrown.

Example:
```
x = 10
if x == 10
```

```
print("x is 10")
```

output:

```
File "c:\Users\name\OneDrive\Desktop\demo.py", line 2
    If x == 10
             ^
Syntax Error: expected ':'
```

`Note:` It shows that there is a `Syntax Error` on `line 2`

Now we will get the correct results

The `Syntax Error` occurs on `line 2` because the *if* statement is missing a colon `:` at the end of the line.

The correct code should be:

```
x = 10
if x == 10:
    print ("x is 10")
```

This code will execute correctly and print `x is 10` to the console because the `Syntax Error` has been resolved.

2. Runtime Errors:

- A runtime error is a type of error that occurs during program execution.
- The Python interpreter executes a script if it is syntactically correct.
- However, if it encounters an issue at runtime, which is not detected when the script is parsed, script execution may halt unexpectedly.

Types of runtime errors

1. Name Error

2. Type Error

3. Index Error

4. Attribute Error

1.Name Error:

- The Name Error occurs when you try to use a variable, function, or module that doesn't exist or wasn't used in a valid way.
- Some of the common mistakes that cause this error are: Using a variable or function name that is yet to be defined

Example program:

```
def calculate_sum(a, b):
    total = a + b
    return total
```

```
x = 5
y = 10
z = calculate_sum(x, w)
    print(z)
```

Output:

```
Traceback (most recent call last):
  File "c:\Users\name\OneDrive\Desktop\demo.py", line 7, in <module>
    Z = calculate_sum(x, w)
                ^
Name Error: name 'w' is not defined
```

Now we will get the correct result:

This error message indicates that the interpreter could not find the variable `w` in the current scope. To fix this error, we need to correct the misspelling of the variable name to `y`, like so:

```
def calculate_sum(a, b):
    total = a + b
    return total

x = 5
y = 10
z = calculate_sum(x, y)
print(z)
```

Now the code will run without any errors, and the output will be `15`, which is the correct result of adding `x` and `y`

2. Type Error

The Python Type Error is an exception that occurs when the data type of an object in an operation is inappropriate. This can happen when an operation is performed on an object of an incorrect type, or it is not supported for the object.

Example program:

```
x = "10"
y = 5
Z = x + y
print(z)
```

Output:

```
Traceback (most recent call last):
    File "c:\Users\name\OneDrive\Desktop\demo.py", line 3, in <module>
      Z = x + y
```

```
            ~~^~~
Type Error: can only concatenate str (not "int") to str
```

Now we will get the correct result:

This error message indicates that we cannot `concatenate` a `string` and an `integer` using the + operator. To fix this error, we need to convert the integer `y` to a string before concatenating it with `x`, like so:

```
x = "10"
y = 5
Z = x + str(y)
print(z)
```

The `str()` method to convert our integer to a string. Now the code will run without any errors, and the output will be `105`, which is the result of concatenating `x` and `y` as strings.

3. Index Error:

This error occurs when an attempt is made to access an item in a list at an index which is out of bounds. The range of a list in Python is [0, n-1], where n is the number of elements in the list.

Example program:

```
my_list = [100, 200, 300, 400, 500]
print(my_list[5])
```

When the above code is executed in an IDE, we get the following error message:

```
  Traceback (most recent call last):
      File "c:\Users\name\OneDrive\Desktop\demo.py", line 2, in <module>
        print(my list[p
              ~~^~~
Index Error: list index out of range
```

Now we will get the correct result:

This error message indicates that we are trying to access an index that is outside the range of valid indices for the list. To fix this error, we need to make sure that we are only accessing valid indices of the list, like so:

```
my_list = [100, 200, 300, 400, 500]
print(my_list[4])
```

Now the code will run without any errors, and the output will be `500`, which is the element at `index 4` of the list.

➢ *Attribute Error:*

The Python Attribute Error is raised when an invalid attribute reference is made. This can happen if an attribute or function not associated with a data type is referenced on it. For example, if a method is called on an integer value, an Attribute Error is raised.

Example program in python

```
my_string = "Hello, world!"
my_string.reverse()
```

Output:

```
Traceback (most recent call last):
  File "c:\Users\name\OneDrive\Desktop\demo.py", line 2, in <module>
    my_string.reverse ()

AttributeError: 'str' object has no attribute 'reverse'
```

This error message indicates that we are trying to access an `attribute (reverse)` that is not defined for the type of `object (str)` we are working with.

Now we will get the correct result:

To fix this error, we need to use a different method or attribute that is defined for strings, like `[::-1]` to reverse the string:

```
my_string = "Hello, world!"
reversed_string = my_string[::-1]
print(reversed_string)
```

Now the code will run without any errors, and the output will be `!dlrow ,olleH`, which is the reversed string of `my_string`.

3.Logical Error

- A logical error occurs in Python when the code runs without any syntax or runtime errors but produces incorrect results due to flawed logic in the code.
- These types of errors are often caused by incorrect assumptions, an incomplete understanding of the problem, or the incorrect use of algorithms or formulas.

Example program:

```
x = float(input('Enter a number: '))
y = float(input('Enter a number: '))

z = x+y/2
```

```
print('The average of the two numbers you have entered is:',z)
```

Output:

```
Enter a number: 3
Enter a number: 4
The average of the two numbers you have entered is: 5.0
```

The example above should calculate the average of the two numbers the user enters. But, because of the order of operations in arithmetic (the division is evaluated before addition) the program will not give the correct answer:

To rectify this problem, we will simply add the parentheses: z = (x+y)/2

```
x = float(input('Enter a number: '))
y = float(input('Enter a number: '))

z = (x+y)/2
print('The average of the two numbers you have entered is:',z)
```

Now we will get the correct result:

```
Enter a number: 3
Enter a number: 4
The average of the two numbers you have entered is: 3.5
```

Exceptions:

- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
- Exceptions are raised when the program is syntactically correct, but the code results in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.
- An exception in Python is an incident that happens while executing a program that causes the regular course of the program's commands to be disrupted.
- When a Python code comes across a condition it can't handle, it raises an exception. An object in Python that describes an error is called an exception.
- An exception in Python is an incident that happens while executing a program that causes the regular course of the program's commands to be disrupted.

- When a Python code comes across a condition it can't handle, it raises an exception. An object in Python that describes an error is called an exception.
- When a Python code throws an exception, it has two options: handle the exception immediately or stop and quit.
- An Exception is an Event, which occurs during the execution of the program.
- It is also known as a **run time error**.
- When that error occurs, Python generates an exception during the execution and that can be handled, which avoids your program to interrupt.

Example:

```
a = 5

b = 0

print(a/b)
```

Output:

```
Traceback (most recent call last):
  File "/home/8a10be6ca075391a8b174e0987a3e7f5.py", line 3, in <module>
    print(a/b)
ZeroDivisionError: division by zero
```

In this code, The system can not divide the number with zero so an exception is raised.

- Exception handling enables a program to deal with exceptions and continue its normal execution.
- An error that occurs at runtime is also called an exception.
- The run-time exceptions immediately terminate a running program. Python provides a standard mechanism called exception handling that allows the program to catch the error and prompt the user to correct the issue
- The syntax for exception handling is to wrap the code that might raise (or throw) an exception in a try clause, as follows:

  try:
  <body>
  except <ExceptionType>:
  <handler>
- Here, <body> contains the code that may raise an exception. When an exception occurs, the rest of the code in <body> is skipped.
- If the exception matches an exception type, the corresponding handler is executed.
- <handler> is the code that processes the exception.

Example:

```
Def divide(a,b):
try:
c = a/b
return c
except :
print ("Error in divide function")
print(divide(10,0))#function call
```

Advantages of Exception Handling:

- **Improved program reliability**: By handling exceptions properly, you can prevent your program from crashing or producing incorrect results due to unexpected errors or input.
- **Simplified error handling**: Exception handling allows you to separate error handling code from the main program logic, making it easier to read and maintain your code.
- **Cleaner code:** With exception handling, you can avoid using complex conditional statements to check for errors, leading to cleaner and more readable code.
- **Easier debugging**: When an exception is raised, the Python interpreter prints a traceback that shows the exact location where the exception occurred, making it easier to debug your code.

Disadvantages of Exception Handling:

- **Performance overhead:** Exception handling can be slower than using conditional statements to check for errors, as the interpreter has to perform additional work to catch and handle the exception.
- **Increased code complexity**: Exception handling can make your code more complex, especially if you have to handle multiple types of exceptions or implement complex error handling logic.
- **Possible security risks:** Improperly handled exceptions can potentially reveal sensitive information or create security vulnerabilities in your code, so it"s important to handle exceptions carefully and avoid exposing too much information about your program.

Exception Handling using try, except and finally

- **Try**: This block will test the excepted error to occur
- **Except**: Here you can handle the error
- **Else**: If there is no exception then this block will be executed
- **Finally**: Finally block always gets executed either exception is generated or not

Syntax:

```
try:
# Some Code....

except:
# optional block
# Handling of exception (if required)

else:
# execute if no exception

finally:
# Some code .....(always executed)
```

Let's first understand how the try and except works –

- First **try** clause is executed i.e. the code between **try** and **except** clause.
- If there is no exception, then only **try** clause will run, **except** clause will not get executed.
- If any exception occurs, the **try** clause will be skipped and **except** clause will run.
- If any exception occurs, but the **except** clause within the code doesn't handle it, it is passed on to the outer **try** statements. If the exception is left unhandled, then the execution stops.
- A **try** statement can have more than one **except** clause.

Example: Let us try to take user integer input and throw the exception in except block.

```
# Python code to illustrate
# working of try()
def divide(x, y):
    try:
        # Floor Division : Gives only Fractional
        # Part as Answer
        result = x // y
        print("The answer is :", result)
    except ZeroDivisionError:
        print("Sorry ! You are dividing by zero ")

# Look at parameters and note the working of Program
```

```
divide(3, 2)
divide(3, 0)
```

Output:

```
The answer is : 1
Sorry ! You are dividing by zero
```

Else Clause

The code enters the else block only if the try clause does not raise an exception.

Example: Else block will execute **only when no exception occurs**.

```
# Python code to illustrate
# working of try()
def divide(x, y):
    try:
        # Floor Division : Gives only Fractional
        # Part as Answer
        result = x // y
    except ZeroDivisionError:
        print("Sorry ! You are dividing by zero ")
    else:
        print("The answer is :", result)

# Look at parameters and note the working of Program
divide(3, 2)
divide(3, 0)
```

Output:

```
The answer is : 1
Sorry ! You are dividing by zero
```

Finally Keyword

Python provides a keyword finally, which is **always executed** after try and except blocks. The finally block always executes after normal termination of try block or after try block terminates due to some exception. Even if you return in the except block still the finally block will execute

Example: Let's try to throw the exception in except block and Finally will execute either exception will generate or not

```
# Python code to illustrate
# working of try()
def divide(x, y):
    try:
        # Floor Division : Gives only Fractional
        # Part as Answer
        result = x // y
    except ZeroDivisionError:
```

```
        print("Sorry ! You are dividing by zero ")
    else:
        print("The answer is :", result)
    finally:
        # this block is always executed
        # regardless of exception generation.
        print('This is always executed')

# Look at parameters and note the working of Program
divide(3, 2)
divide(3, 0)
```

Output: The answer is : 1

This is always executed

Sorry ! You are dividing by zero

This is always executed

Python try...except Block

The try..except block is used to handle exceptions in Python.

Here's the syntax of try..except block:

```
try:
# code that may cause exception
except:
# code to run when exception occurs
```

- Here, we have placed the code that might generate an exception inside the try block. 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.

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.
```

Run Code

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.

If none of the statements in the try block generates an exception, the except block is skipped.

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
```

Run Code

In this 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, print(even_numbers[5])

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)
```

Run Code

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
```

Note: Exceptions in the else clause are not handled by the preceding except clauses.

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.

Let's see an example, try:

```
numerator = 10
denominator = 0
result = numerator/denominator
print(result)
except:
print("Error: Denominator cannot be 0.") finally:
print("This is finally block.")
```

Run Code

Output

Error: Denominator cannot be 0.This is finally block.

In the above example, we are dividing a number by **0** inside the try block. Here, this codegenerates an exception.

The exception is caught by the except b

CHAPTER-05

Python Functions:

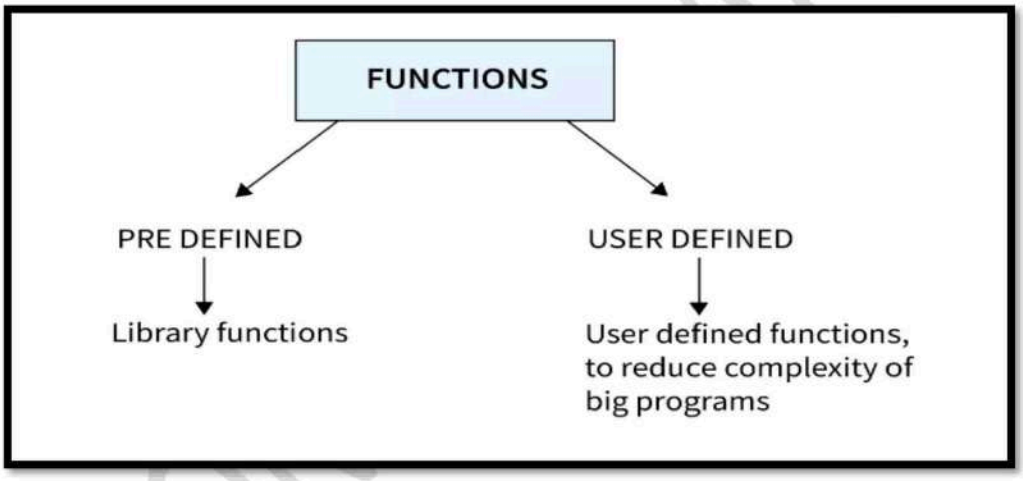
- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

- Functions are the basic building block of any Python program, defined as the organized block of reusable code, which can be called whenever required.
- A function is used to carry out a specific task. The function might require multiple inputs. When the task is done executing, the function can or can not return one or more values.

Types of functions in python

There are two types of functions in python:

- **User-Defined Functions** - these types of functions are defined by the user to perform any specific task
- **Built-in Functions** - These are pre-defined functions in python.



Function definition

- A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.
- Function is a group of related statements that perform 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.
- It avoids repetition and makes code reusable.
- Python functions once defined can be called many times in a program.
- User-defined and built-in functions are the two main categories of functions in Python.

Built-in Functions

Built-in functions are already defined in python. A user has to remember the name and parameters of a particular function. Since these functions are pre-defined, there is no need to define them again.

Some of the widely used built-in functions are given below:

Example:

```
Lis1= [2, 5, 19, 7, 43]

print("Length of string is",len(list1))
print("Maximum number in list is ",max(llist1))
print("Type is",type(list1))
```

Output:

```
Length of string is 5
Maximum number in list is 43
Type is <class 'list'>
```

In the above program, we created a list and stored a few numbers in it, then called various in-built functions on the list and displayed the output.

Some of the widely used python built-in functions are:

Function	Description
len()	Returns the length of a python object
abs()	Returns the absolute value of a number
max()	Returns the largest item in a python iterable
min()	Returns the largest item in a python iterable
sum()	Sum() in Python returns the sum of all the items in an iterator
type()	The type() in Python returns the type of a python object
help()	Executes the python built-in interactive help console
input()	Allows the user to give input

format()	Formats a specified value
bool()	Returns the boolean value of an object

User-Defined Functions

These functions are defined by a programmer to perform any specific task or to reduce the complexity of big problems and use that function according to their need.

The syntax to declare a function is:

```
def function_name(arguments):
    # function body

    return
```

Here,

- def - keyword used to declare a function
- function_name - any name given to the function
- arguments - any value passed to function
- return (optional) - returns value from a function

Example program:

```
def sub(x, y):
    return x-y

print(sub(5,2))
```

Output:

```
3
```

In the above program, we have created our program that subtracts two numbers passed as arguments and displays the result. Such types of functions are known as user-defined functions.

Advantages of functions in Python

- ***Helps in increasing modularity of code*** – Functions in python help the user to divide the program into smaller parts and solve them individually, thus making it easier to implement.
- ***Minimizes Redundancy*** – Python functions help us to save the effort of rewriting the whole code. All we got to do is call the function once it is defined.
- ***Maximizes Code Reusability*** – Once a function is defined in python, it can be called as many times as needed, thus enhancing code reusability.

- ***Improves Clarity of Code*** – Since a large program is divided into sections with the help of functions it helps increase the readability of code while ensuring easy debugging.

Function calling:

Once you have defined a function, you can call it in your code as many times as you need. To call a functior in Python, you simply type the name of the function followed by parentheses (). If the function takes any arguments, they are included within the parentheses.

we have declared a function named greet().

```
def greet():
   print('A  P J Abdul kalam')
```

Now, to use this function, we need to call it.

Here's how we can call the greet() function in Python.

```
# call the function
greet()
```

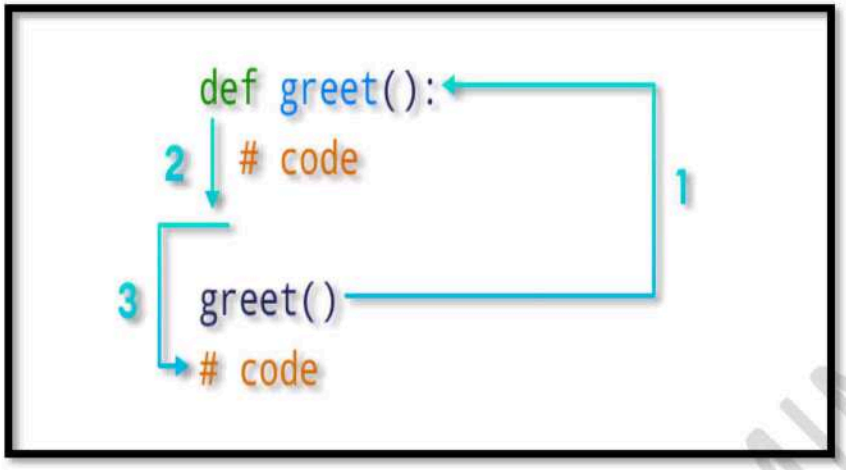
Example: Python Function

```
def greet():
   print("A  P J Abdul kalam')

# call the function
greet()

print('Missile  man of india')
Run Code
```

Output

```
A P J Abdul kalam
Missile  man of india
```

In the above example, we have created a function named greet(). Here's how the program works:



Here,

- When the function is called, the control of the program goes to the function definition.
- All codes inside the function are executed.
- The control of the program jumps to the next statement after the function call.

Defining a function

```
def function_name(
    parameters ): # code
    bloc
```

- The beginning of a function header is indicated by a keyword called def.
- Function name is the function's name that we can use to separate it from others. We will use this name to call the function later in the program.
- The function header is terminated by a colon (:).
- The body of the function is made up of several valid Python statements. The indentation depth of the whole code block must be the same (usually 4 spaces).
- We can use a return expression to return a value from a defined function.

Calling a Function

- Calling a function executes the code in the function.
- In a function's definition, you define what it is to do. To use a function, you have to call or invoke it.
- The programthat calls the function is called a caller.
- There are two ways to call a function, depending on whether or not it returns a value. If the function returns a value, a callto that function is usually treated as a value.

Parameters and arguments:

Parameters are passed during the definition of function while Arguments are passed during the function call.

Example:

```
#here a and b are parameters

def add(a,b): #//function
definition return a+b
#12 and 13 are arguments#function call result=add(12,13)
print(result)
```

Types of function arguments

There are various ways to use arguments in a function. In Python, we have thefollowing 4 types of function arguments.

- Keyword arguments
- Default arguments
- Variable length arguments
- Positional arguments

Keyword Arguments

- When we call a function with some values, these values get assigned to the argumentsaccording to their position.
- Python allows functions to be called using keyword arguments. When we call functions inthis way, the order (position) of the arguments can be changed.
- If you have some functions with many parameters and you want to specify only some of them, then you can give values for such parameters by naming them - this iscalled **keyword arguments** - we use the name (keyword) instead of the position to specify the arguments to the function.

Example:

```
Def display(a,b):
  Print(a,b)
Display(b=20,a=10)
```

Default Arguments

- In a function, arguments can have default values. We assign default values to the argument using the '=' (assignment) operator at the time of function definition. You can define a function with any number of default arguments.
- The default value of an argument will be used inside a function if we do not pass a value to that argument at the time of the function call. Due to this, the default arguments become optional during the function call.

- It overrides the default value if we provide a value to the default arguments during function calls.

Example:

- Let's define a function student() with four arguments name, age, grade, and school.
- If you call a function without school and grade arguments, then the default values of grade and school are used.
- The age and name parameters do not have default values and are required (mandatory) during a function call.

```
def student(name, age, grade="Five", school="ABC
  School"): print('Student Details:', name, age, grade,
  school)
student('Jon', 12)
```

 # Output: Student Details: Jon 12 Five ABC School

Variable-length arguments

- Sometimes you may need more arguments to process function than you mentioned in the definition.
- If we don't know in advance about the arguments needed in function, you can use variable-length arguments also called arbitrary arguments.

For this an asterisk (*) is placed before a parameter in function definition which can hold non-keyword variable-length arguments and a double asterisk (**) is placed before a parameter in function which can hold keyword variable-length arguments.

Example

```
def wish(*names):
"""This function greets all
the person in the names tuple."""
print("Hello",name)
wish("MRCET","CSE","SIR","MADAM")
```

Output:

Hello MRCET Hello CSE Hello SIR Hello MADAM

Positional Arguments:

- Positional arguments are those arguments where values get assigned to the arguments by their position when the function is called.

For example, the 1st positional argument must be 1st when the function is called.

The 2nd positional argument needs to be 2nd when the function is called, etc.

By default, Python functions are called using the positional arguments.

Example: Program to subtract 2 numbers using positional arguments. def data(name, rollno):

Example:

```
Def display(a,b):
  Print(a,b)
Display(a=10,b=20) #For a 10 will be assigned and for b 20will be assigned
```

Recursion

- Recursion is the process of defining something in terms of itself.
- A function can call other functions. It is even possible for the function to call itself.
- These type of construct are termed as recursive functions.

Following is an example of recursive function to find the factorial of an integer

```
def fact(x):

if X==0:

result=1

else:

result = x * fact(x-1)

return result
```

Parameter passing mechanism

- When we define a custom or user defined function in python we may need to specify parameter names between the function's parentheses.
- If we specify parameters in the function definition, then we need to pass argument values to the function's parameters while calling it.
- The function use that passed values during its execution by referencing it via parameter name.
  - Basically, there are two ways to pass argument values to the function's parameters.
    - Call/Pass by Value
    - Call/Pass by Reference

Call/Pass by Value in Python

- In pass by value (also known as call by value), the argument passed to the function is the copy of of its original object.
- If we change or update the value of object inside the function, then original object will not change. If the argument is variable, the copy of the current value of the variable is passed to the function's parameter.
- The value of the variable in the function call is not affected by what happens inside the function. If the argument is passed by value, a copy of it is made and passed to the function.
- If the argument values being passed to the function is large, the copying can take up a lot of time and memory.

Example

```
def my_pass_by_value(b):
 b += 2
print(b)
 c=my_pass_by_value
 (20) print(c)
```

Output: 22

Call/Pass by Reference

- In pass by reference (also known as call by reference), the argument passed to the function's parameter is the original object.
- If we change the value of object inside the function, the original object will also change.

Example:

```
def my_pass_by_reference(r):
r += 10
print(r)
ref1=100
h=my_pass_by_reference(ref1)
print(h)
```

Output 110

Range () and exit() functions:

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.

It creates the sequence of numbers from start to stop -1 . For example, range(5) will produce [0, 1, 2, 3, 4] . The result contains numbers from 0 to up to 5 but not five.

There are three ways you can call range():

- range(stop) takes one argument.
- range(start, stop) takes two arguments.
- range(start, stop, step) takes three arguments.

➢ **range(stop)**

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

```
for i in range(3):
    print(i)
```

Output:

```
0
1
2
```

we have all the whole numbers from 0 up to but not including 3, the number you provided as the stop.

➢ **range(start, stop)**

When you call range() with two arguments, you get to decide not only where the series of numbers stops but also where it starts, so you don't have to start at 0 all the time. You can use range() to generate a series of numbers from *A* to *B* using a range(A, B). Let's find out how to generate a range starting at 1.

To calling range() with two arguments:

```
for i in range(1, 8):
    print(i)
```

Output:

```
1
2
3
4
5
6
7
```

All the whole numbers from 1 (the number you provided as the start) up to but not including 8 (the number you provided as the stop).

But if you add one more argument, then you'll be able to reproduce the output you got earlier when you were using the list named numbers_divisible_by_three.

➢ **range(start, stop, step)**

When you call range() with three arguments, you 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 you don't provide a step, then range() will automatically behave as if the step is 1.

```
for i in range(3, 16, 3):
    quotient = i / 3
    print(f"{i} divided by 3 is {int(quotient)}.")
```

Your output will look exactly like the output of the for-loop you saw earlier in this article, when you were using the list named numbers_divisible_by_three:

Now we will get the correct result:

```
3 divided by 3 is 1.
6 divided by 3 is 2.
9 divided by 3 is 3.
12 divided by 3 is 4.
15 divided by 3 is 5.
```

Note: step can be a positive number or a negative number, but it can't be 0.

```
range(1, 4, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: range() arg 3 must not be zero
```

Exit (): The exit() function in Python is used to exit or terminate the current running script or program. You can use it to stop the execution of the program at any point.

When the **exit()** function is called, the program will immediately stop running and exit.

The syntax of the exit()

```
exit([status])
```

- Here, `status` is an optional argument that represents the exit status of the program.
- The exit status is an integer value that indicates the reason for program termination.

- By convention, a status of 0 indicates successful execution, and any non-zero status indicates an error or abnormal termination.
- Python provides many exit functions using which a user can exit the python program.

Types of python exit function

Python mainly provides four commands using which you can exit from the program and stop the script execution at a certain time. The 4 types of exit commands are:-

- exit()
- quit()
- sys.exit()
- os._exit()

Python exit() function

The python exit() function is available in the site module. It is used for the exit or termination of a Python script. The exit command should not be used in production and can only be used for interpreters. Below is the sample python code with the exit function.

```
age = 45
if age < 50:
# exit the program
print(exit)
exit(0)
```

Python quit() function

In Python, the `quit()` function is equivalent to the `exit()` function. However, the quit() function is only available n python2 but not in python 3.Quit() function should only be used in the interpreter. Below is the sample python code which uses the quit() function.

```
age = 45
if age &amp;lt; 50:
# quit the program
print(quit)
quit()
```

The above code defines a variable age as 45 and then checks if the value of age is less than 50. If it is, it will print the value of the quit function and then call the quit() function to stop the program from running.

Python sys.exit() function

- In Python, the `sys.exit()` function is used to exit or terminate a Python script.

  The **sys.exit()** function can be used in production because it raises an `SystemExit` exception that causes the interpreter to exit.

- You can use the sys.exit() program with or without arguments. With an argument, the program will exit and return a successful exit code on 0, whereas with an argument, which is the exit code like:

  sys.exit(0) # success

  sys.exit(1) # failure

```
import sys
for i in range(1,5):
print(i)
if i == 3:
sys.exit()
```

- The above code will print the numbers from 1 to 3, and then the `sys.exit()` function will be called when the value of `i` is 3. This will cause the script to terminate and end the program with an exit code of 0, indicating a successful termination.

python os._exit() function

- In Python, the os._exit() function is used to immediately exit the current process without performing any cleanup or finalization tasks. The os._exit() does not call any cleanup handlers, close open files, o flush stdio buffers.
- Below is the sample python code using the os._exit() function

```
import os
for i in range(1,5):
print(i)
if i == 3:
os._exit(0)
```

After the number 3 is printed, the `os._exit(0)` function will be called, causing the script to immediately terminate and exit the process without printing the remaining numbers (4) in