ANNAMACHARYA UNIVERSITY, RAJAMPET
(ESTD UNDER AP PRIVATE UNIVERSITIES (ESTABLISHMENT AND REGULATIONS) ACT, 2016
RAJAMPET, Annamayya District, AP, INDIA

Course : Python Programming

Course Code : **24FMCA22T**

Branch : **MCA**

Prepared by: **B. Hari Krishna**

Designation : **Assistant Professor**

Department : **MCA**

Title of the Course : **Python Programming**

Category : **PC**

Year : **I**

Semester : **II**

Course Code : **24FMCA22T**

Branch : **MCA**

Lecture Hours	Tutorial Hours	Practice Hours	Credits
3	0	0	3

COURSE OBJECTIVES:

- To introduce Python programming language through its core language basics and program design techniques suitable for modern applications.
- To utilize high-performance programming constructs available in Python, to develop solutions in real life scenarios.
- To understand the wide range of programming facilities available in Python.
- To import the packages and text files and process them accordingly.
- Understand the object-oriented features of Python.

COURSE OUTCOMES:

The Student will be able to

1. Summarize python fundamental concept.
2. Apply python syntax and semantics use of flow control statements.
3. Analyze functions, turtle graphics using python.
4. Apply the python text-files, exception handling and manipulate dictionary data type.
5. Apply object oriented programming features.

UNIT I **12**

INTRODUCTION TO PYTHON: The process of computational Problem solving, The Python Programming Language, A First Program.

DATA AND EXPRESSIONS: Literals, Variables and Identifiers, I/O Functions, Operators, Operator precedence and associativity rules, Expression and Data types with examples.

UNIT II **11**

CONTROL STRUCTURES: Boolean Expressions (Conditions), Selection Control, Iterative Control and Examples. **LISTS:** List Structures, Lists (Sequences) in Python, Iterating over Lists (Sequences) in Python with Examples, Tuple data type.

UNIT III **12**

FUNCTIONS: Program Routines, More on Functions-Calling Value-Returning Functions, Calling Non-Value-Returning Functions, Parameter Passing, Keyword Arguments in Python, Default Arguments in Python, Variable Scope. Lambda function, Map function, Filters function, **RECURSION**: Recursive Functions, Recursive Problem Solving, Iteration vs. Recursion. **OBJECTS:** Software Objects, Turtle Graphics, Turtle programs

UNIT IV **12**

MODULAR DESIGN: Modules, Top-Down Design, Python Modules. Packages **TEXT FILES**: Using Text Files, String Processing, Exception Handling. **DICTIONARIES AND SETS**: Dictionary Type in Python, Set Data Type.

UNIT V **12**

OBJECT-ORIENTED PROGRAMMING: What is Object-Oriented Programming, features of OOP's, Classes, objects, type's of variables, methods, Constructors with examples, Inheritance, Polymorphism, Data Abstraction, Encapsulation

PRESCRIBED TEXTBOOK:

1. Dr. R. Nageswar Rao, Core Python Programming, Dreamtech Press, 3rd Edition, 2021.

REFERENCE BOOKS:

1. Charles Dierbach, Introduction to Computer Science using Python, Wiley, 2nd Edition, 2012.
2. Wesley J Chun, "Core Python Applications Programming", Prentice Hall, 3rd Edition, 2015.
3. James Payne, Beginning Python: Using Python 2.6 and Python 3, Wiley India, 2nd Edition 2013.
4. Paul Gries, Jennifer Campbell, Jason Montojo, Practical Programming: An Introduction to Computer Science Using Python 3, Pragmatic Bookshelf, 2nd Edition, 2010.

CO-PO MAPPING:

Course Outcomes	Foundation Knowledge	Problem Analysis	Development of Solutions	Modern Tool Usage	Individual and Teamwork	Project Management and Finance	Ethics	Life-long Learning
24FMCA22T.1	2	2	1	-	-	-	-	-
24FMCA22T.2	3	2	1	-	-	-	-	-
24FMCA22T.3	3	2	1	-	-	-	-	-
24FMCA22T.4	3	2	1	-	-	-	-	-
24FMCA22T.5	2	2	1	-	-	-	-	-

UNIT-I

INTRODUCTION TO PYTHON: The process of computational Problem solving, The Python Programming Language, A First Program. DATA AND EXPRESSIONS: Literals, Variables and Identifiers, I/O Functions, Operators, Operator precedence and associativity rules, Expression and Data types with examples

INTRODUCTION TO PYTHON: The process of computational Problem solving

We usually use the term computerization to indicate the use of computer to develop software in order to automate any routine human task efficiently. Computers are used for solving various day-to-day problems and thus problem solving is an essential skill that a computer science student should know. It is mention that computers themselves cannot solve a problem. Precise step-by-step instructions should be given by us to solve the problem. Thus, the success of a computer in solving a problem depends on how correctly and precisely we define the problem, design a solution (algorithm) and implement the solution (program) using
a programming language. Thus, problem solving is the process of identifying a problem, developing an algorithm for the identified problem and finally implementing the algorithm to develop a computer program.

Computational problem is a problem that a computer might be able to solve or a question that a computer may be able to answer. For example, the problem of **factoring**

Computational problems are one of the main objects of study in theoretical computer science. The field of computational complexity theory attempts to determine the amount of resources (computational complexity) solving a given problem will require and explain why some problems are intractable or un decidable. Computational problems belong to complexity classes that define broadly the resources (e.g. time, space/memory, energy, circuit depth) it takes to compute (solve) them with various abstract machines (e.g. classical or quantum machines).

It is typical of many problems to represent both instances and solutions by binary strings, namely elements of {0, 1}* (see regular expressions for the notation used). For example, numbers can be represented as binary strings using binary encoding.

The Process of Computational Problem Solving

The process of computational problem solving involves understanding the problem, designing a solution, and writing the solution (code).

Steps for Problem Solving

Steps for Problem Solving

Algorithm

In our day-to-day life we perform activities by following certain sequence of steps. Examples of activities include getting ready for school, making breakfast, riding bicycle, wearing a tie, solving a puzzle and so on. To complete each activity, we follow a sequence of steps. Suppose following are the steps required for an activity 'riding a bicycle':

1) Remove the bicycle from the stand,
2) sit on the seat of the bicycle,
3) start peddling,
4) Use breaks whenever needed and
5) Stop on reaching the destination.

Coding

After finalizing the algorithm, we need to convert the algorithm into the format which can be understood by the computer to generate the desired solution. Different high level programming languages can be used for writing a program.

It is equally important to record the details of the coding procedures followed and document the solution. This is helpful when revisiting the programs at a later stage.

Testing and Debugging

The program created should be tested on various parameters. The program should meet the requirements of the user. It must respond within the expected time. It

Should generate correct output for all possible inputs. In the presence of syntactical errors, no output will be obtained. In case the output generated is incorrect, then the program should be checked for logical errors, if any. Software industry follows standardized testing methods like unit or component testing, integration testing, system testing, and acceptance testing while developing complex applications.

This is to ensure that the software meets all the business and technical requirements and works as expected. The errors or defects found in the testing phases are debugged or rectified and the program is again tested. This continues still all the errors are removed from the program.

Once the software application has been developed, tested and delivered to the user, still problems in terms of functioning can come up and need to be resolved from time to time. The maintenance of the solution, thus, involves fixing the problems faced by the user, answering the queries of the user and even serving the request for addition or modification of feature

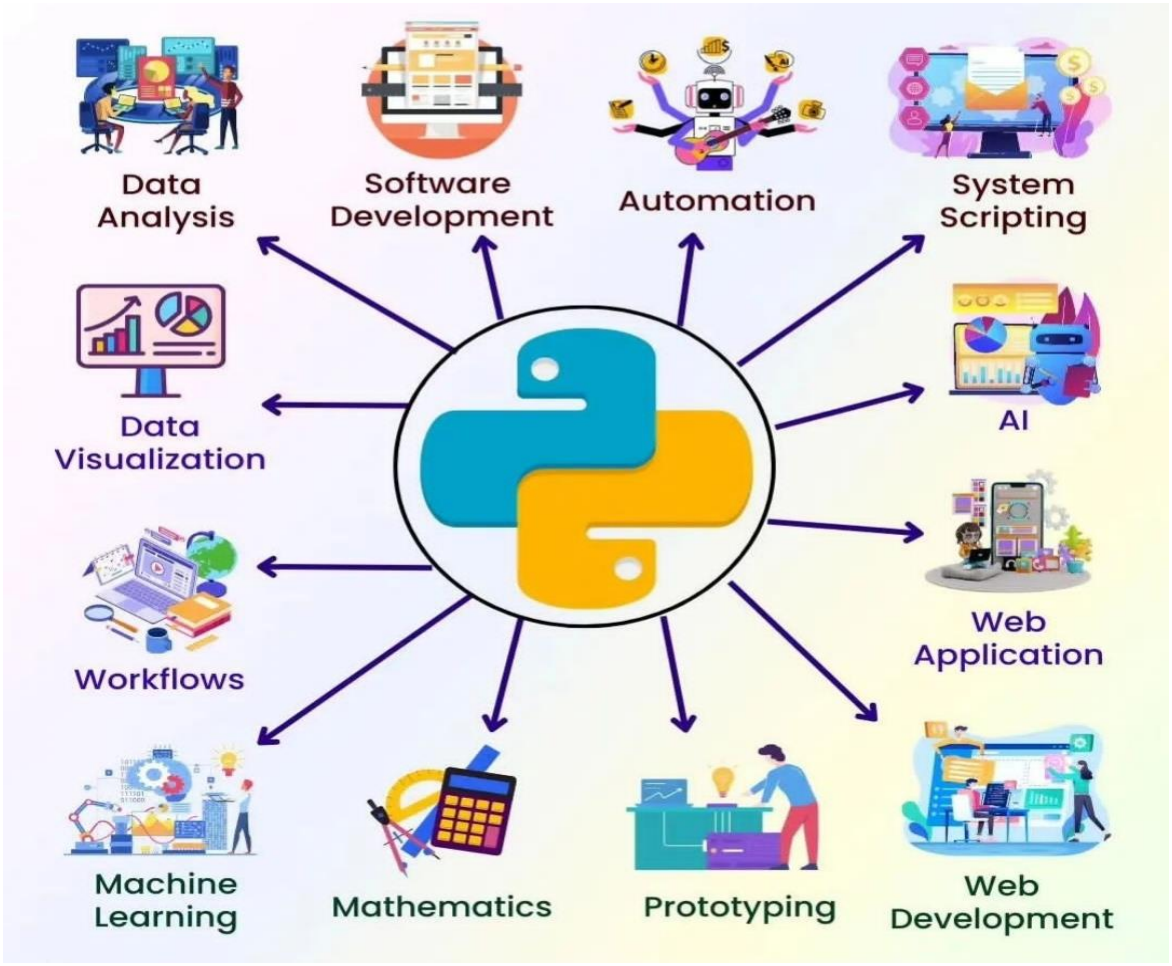
The Python Programming Language, A First Program

Python History

The implementation of Python was started in December 1989 by **Guido Van Rossum** at CWI in Netherland. But officially Python was made available to public in 1991. The official Date of Birth for Python is: Feb 20th 1991.

- In 1994, Python 1.0 was released with new features like lambda, map, filter, and reduce.
- Python 2.0 added new features such as list comprehensions, garbage collection systems.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify the fundamental flaw of the language.
- There is a fact behind choosing the name Python. Guido van Rossum was reading the script of a popular BBC comedy series "Monty Python's Flying Circus". It was late on-air 1970s.
- Van Rossum wanted to select a name which unique, sort, and little-bit mysterious. So he decided to select naming Python after the "Monty Python's Flying Circus" for their newly created programming language.
- Guido developed Python language by taking almost all programming features from Different languages
    - **(i)** Functional Programming Features from C
    - **(ii)** Object Oriented Programming Features from C+
    - **(iii)** Scripting Language Features from Perl and Shell Script
    - **(iv)** Modular Programming Features from Modula-3
    - **(v)** Most of syntax in Python Derived from C and ABC languages.

Where we can use Python:



We can use everywhere. The most common important application areas are

1) For developing Desktop Applications
2) For developing web Applications
3) For developing database Applications
4) For Network Programming
5) For developing game
6) For Data Analysis Applications
7) For Machine Learning
8) For developing Artificial Intelligence Applications
9) For IoT

Definition:

Python is a simple, dynamic, general purpose, interpreter and high level programming language and also object-oriented programming language. Python is much easier than other programming languages and helps you create beautiful applications with less effort and much more ease.

Features of Python:

1) **Simple and easy to learn**:

- Python is a simple programming language. When we read Python program, we can feel like reading English statements.
- The syntaxes are very simple and only 30+ keywords are available.
- When compared with other languages, we can write programs with very less number of lines. Hence more readability and simplicity.
- We can reduce development and cost of the project.

2) **Freeware and Open Source**:

- We can use Python software without any license and it is freeware.
- Its source code is open, so that we can we can customize based on our requirement.
- E.g.: Jython is customized version of Python to work with Java Applications.

3) **High Level Programming language**:

- Python is high level programming language and hence it is programmer friendly Language.
- Being a programmer we are not required to concentrate low level activities like Memory management and security etc.

4) **Platform Independent**:

- Once we write a Python program, it can run on any platform without rewriting once Again.
- Internally PVM is responsible to convert into machine understandable form.

5) **Portability**:

- Python programs are portable. I.e. we can migrate from one platform to another Platform very easily. Python programs will provide same results on any platform.

6 **Dynamically Typed:**

- In Python we are not required to declare type for variables. Whenever we are Assigning the value, based on value, type will be allocated automatically. Hence Python Is considered as dynamically typed language. But Java, C etc. are Statically Typed Languages b'z we have to provide type at the Beginning only.
- This dynamic typing nature will provide more flexibility to the programmer.

7 **Both Procedure Oriented and Object Oriented:**

- Python language supports both Procedure oriented (like C, Pascal etc.) and object Oriented (like C++, Java) features. Hence we can get benefits of both like security and Reusability etc.

8 Interpreted:

- We are not required to compile Python programs explicitly. Internally Python Interpreter will take care that compilation.
- If compilation fails interpreter raised syntax errors. Once compilation success then PVM (Python Virtual Machine) is responsible to execute.

9 Extensible:

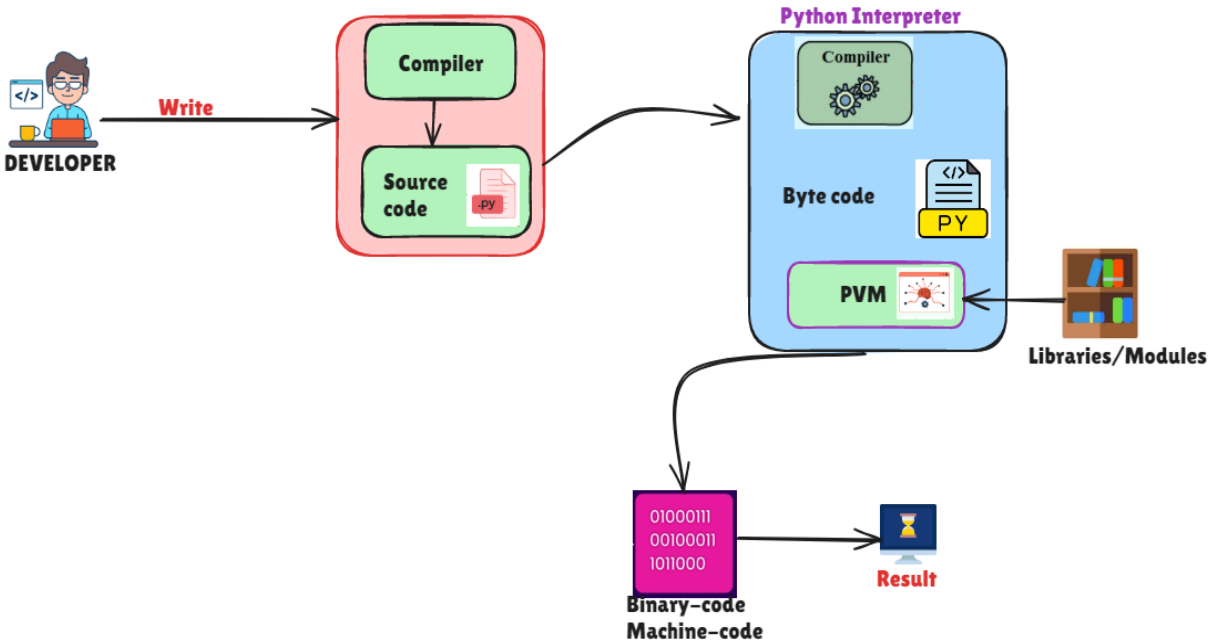
- We can use other language programs in Python.
- The main advantages of this approach are:
- We can use already existing legacy non-Python code
- We can improve performance of the application

10 Embedded:

- We can use Python programs in any other language programs. i.e we can embedded Python programs anywhere.

11 Extensive Library:

- Python has a rich inbuilt library. Being a programmer we can use this library directly and we are not responsible to implement the functionality. Etc.



Python Versions:

- Python 1.0V introduced in Jan 1994
- Python 2.0V introduced in October 2000
- Python 3.0V introduced in December 2008

Note: Python 3 won't provide backward compatibility to Python2 i.e. there is no Guarantee that Python2 programs will run in Python3.

Advantages and Disadvantages of Python

Advantages:

- ❖ It is open-source and readily available to use.
- ❖ It is easy to learn and explore.
- ❖ Third-party modules can be easily integrated.
- ❖ It is high level and object-oriented programming language.
- ❖ It is interactive and portable.
- ❖ Applications can be run on any platform.
- ❖ It is a dynamically typed language.
- ❖ It has great online support and community forums.
- ❖ It has a user-friendly data structure.
- ❖ It has extensive support libraries.
- ❖ It is interpreted language.

Disadvantages:

- ✓ It has limitations with database access.
- ✓ It throws run time issues that cause the issue for the programmers.
- ✓ It consumes more memory because of dynamically typed language.
- ✓ Need more maintenance of application and code

First Python Program (or)

Does python allow you to program in a structured style? Give one example

Python allows you to program in a structured style. Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a program by making extensive use of subroutines, blocks, loops, and conditionals.
One example of structured programming in Python is the use of functions. Functions in Python allow you to break your code down into smaller, more manageable pieces of code that can be reused throughout your program. By using functions, you can make your code more readable, easier to maintain, and more modular

Example:

```
d=eval(input("enter diameter value:"))
r=d/2
c=2*3.14*r
print("The radius value is:",r)
print("The circumference value is:",c)
```

OUTPUT:

```
enter diameter value:4
The radius value is: 2.0
The circumference value is: 12.5
```

We can develop python applications or programs in 2 modes. They are,

1. Interactive mode
2. Batch Mode

1. Interactive mode

- Interactive mode is a command line shell.
- In command line shell if we write any Python statement immediately that statement will execute and gives the result.
- Open command prompt and enter

  python.C:\Users\Lenovo> **python**
- Now we are going to get python prompt area like below

- Now we can test python coding basics like below

```
a = 10
print(a)10
type(a)
<class 'int'>
print("Hello")
Hello
```

NOTE: Interactive mode is not used for development of the businessapplications.
To overcome this problem then we can use Batch Mode.

2 . Batch mode

In Batch mode we write group of python statements in any one of the Editors orIDE's.

Different Editors are:

1. Notepad
2. Notepad++
3. EditPlus
4. nano
5. IDLE

Different IDE's are :

1. Pycharm
2. Visual Studio.
3. Sublimetext
4. Atom
5. Eclipse
6. NetBeen

- After writng the group of Python statements in anyone of the Editor or IDE ,we savethe Python file with extension **.py**

  For example **: FileName.py**

- After developing the .py files , we submit those files to the python **Interpretor** directly.
- Batch mode is used for developent of business applications.

Step 1: Open Notepad Editor

- Open Notepad on your computer.

Step 2: Write the following Python code

```
i = 10
j = 20
print(i + j)
```

Step 3: Save the file

- Save the file as Basic.py
- Choose any location on your computer to save the file, for example, D:\Python_Programs\Basic.py

Step 4: Run the Python file using Command Prompt

- Open Command Prompt
- Navigate to the directory where you saved the file
- cd Python_Programs
- Execute the Python script by running
- python Basic.py

How to Create a Python Program Using IDLE Tool

- Open the **IDLE(Integrated Development and Learning Environment)** tool
- You can find it in your Start menu or by searching "IDLE" on your computer.
- Create a new Python file
- Click on the File menu
- Select New File (or press Ctrl + N)
- This opens a new Python editor window (like a Python Notepad).
- Write your Python program
- In the new window, type your Python code.

IDENTIFIERS

Identifiers: Identifiers are the names given to any variable, function, class, list, methods, etc. for their identification. Python is a case-sensitive language and it has some rules and regulations to name an identifier. Name in Python Program is called Identifier. It can be Class Name OR Function Name OR Module Name OR Variable Name. Here are some rules to name an identifier:

Identifier Rules:

1) Alphabet Symbols (Either Upper case OR Lower case)
2) If Identifier is start with Underscore (_) then it indicates it is private.
3) Identifier should not start with Digits.
4) Identifiers are case sensitive.
5) We cannot use reserved words as identifiers
   - Eg: def = 10
6) There is no length limit for Python identifiers. But not recommended to use too lengthy identifiers.
7) Dollar ($) Symbol is not allowed in Python.

Rules to define Identifiers in Python:

1. The only allowed characters in Python are
   - Alphabet symbols (either lower case or upper c a s e)
     - digits (0 to 9)
   - Underscore symbol (_)
   - By mistake if we are using any other symbol like $ then we will get syntax error.
     - cash = 10 √
     - ca$h =20
2. Identifier should not starts with digit
   - 123total
   - total123 √
3. Identifiers are case sensitive. Of course Python language is case sensitive language.
   - total=10
   - TOTAL=999
   - print(total) #10
   - print(TOTAL) #999

Literals or Values

Literals or Values: Literals are the fixed values or data items used in a source code. Python supports different types of literals such as:

(i) String Literals: The text written in single, double, or triple quotes represents the string literals in Python. For example: "Computer applications", 'milky', etc. We can also use triple quotes to write multi-line strings.

```
a = 'Hello'
b = "Krishna"
c = '''python is good for
                Learning platform'''
# Driver code
print(a) print(b)
print(c)
```

 # String Literals

Output:

```
Hello
Krishna
Python is good for
          Learning platform
```

(ii) Character Literals: Character literal is also a string literal type in which the character is enclosed in single or double-quotes.

 # Character Literals

```
a = 'H'
b = "k"
```

 # Driver code

```
print(a)
print(b)
```

(iii) Numeric Literals: These are the literals written in form of numbers. Python supports the following numerical literals:

- **Integer Literal:** It includes both positive and negative numbers along with 0. It doesn't include fractional parts. It can also include binary, decimal, octal, hexadecimal literal.
- **Float Literal:** It includes both positive and negative real numbers. It also includes fractional parts.

- **Complex Literal:** It includes a+bi numeral, here a represents the real part and b represents the complex part.

 # Numeric Literals

a = 7

b = 8.3

c = -15

 # Driver code

print(a) print(b)

print(c)

(iv) Boolean Literals: Boolean literals have only two values in Python. These are True and False.

a = 3

b = (a == 3)

c = True + 10 #

Driver code

print(a, b, c)

(v) Special Literals: Python has a special literal 'None'. It is used to denote nothing, no values, or the absence of value.

 # Special Literals var

= None print(var)

(vi) Literals Collections: a Literals collection in python includes list, tuple, dictionary, and sets.

1. **List:** It is a list of elements represented in square brackets with commas in between. These variables can be of any data type and can be changed as well.
2. **Tuple:** It is also a list of comma-separated elements or values in round brackets. The values can be of any data type but can't be changed.
3. **Dictionary:** It is the unordered set of key-value pairs.
4. **Set:** It is the unordered collection of elements in curly braces '{}'.

EXAMPLE

```
# Literals collections # List
my_list = [23, "milky", 1.2, 'data'] #
Tuple
my_tuple = (1, 2, 3, 'hari')
# Dictionary
my_dict = {1:'one', 2:'two', 3:'three'}
# Set
my_set = {1, 2, 3, 4} # Driver code
print(my_list) print(my_tuple)
print(my_dict)
print(my_set)
```

DOES PYTHON ALLOW MULTIPLE ASSIGNMENT STATEMENTS

Assigning multiple values to multiple variables. We can assign values to multiple variables in a single line by separating them with commas. Each variable on the left side of the assignment operator (=) corresponds to the respective value on the right side.

```
a, b, c = 1, 2, 3

print(a)  # Output: 1

print(b)  # Output: 2

print(c)  # Output: 3
```

Swapping the values of variables:

Python allows swapping the values of variables without needing a temporary variable.

This is done by utilizing multiple assignments along with tuple unpacking.

```
x = 5

y = 10

x, y = y, x

print(x)  # Output: 10

print(y)  # Output: 5
```

Assigning a single value to multiple variables:

You can assign a single value to multiple variables by chaining them together in a single assignment statement.

```
x = y = z = 0

print(x)  # Output: 0

print(y)  # Output: 0

print(z)  # Output: 0
```

Unpacking a tuple into multiple variables:

You can unpack a tuple into multiple variables by assigning the tuple to those variables.

Python unpacks the tuple elements into the respective variables.

```
mytuple= (10, 20, 30)

x, y, z = mytuple

print(x)  # Output: 10

print(y)  # Output: 20

print(z)  # Output: 30
```

Unpacking a list into multiple variables:

Similar to unpacking a tuple, you can also unpack a list into multiple variables.

This allows you to extract the elements of a list into separate variables.

mylist = [100, 200, 300]

```
a, b, c = mylist

print(a)  # Output: 100

print(b)  # Output: 200

print(c)  # Output: 300
```

PYTHON KEYWORDS:

Python keywords are reserved words that are predefined by the Python language and have special meanings and functionalities.
These keywords cannot be used as identifiers (such as variable names or function names) in your Python programs.
Python's keywords are case-sensitive and must be written exactly as shown.
Here is a list of Python keywords as of Python 3.9:

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

In python we have totally 35 keywords. Out of 35 only 3 keywords first letters starts with upper case. They are,

False True None

- ✓ You can use or importing predefined modules using import keyword.
- ✓ You can also retrieve the list of keywords programmatically using the predefined keyword module in Python:

```
import keyword
print( keyword.kwlist )
```

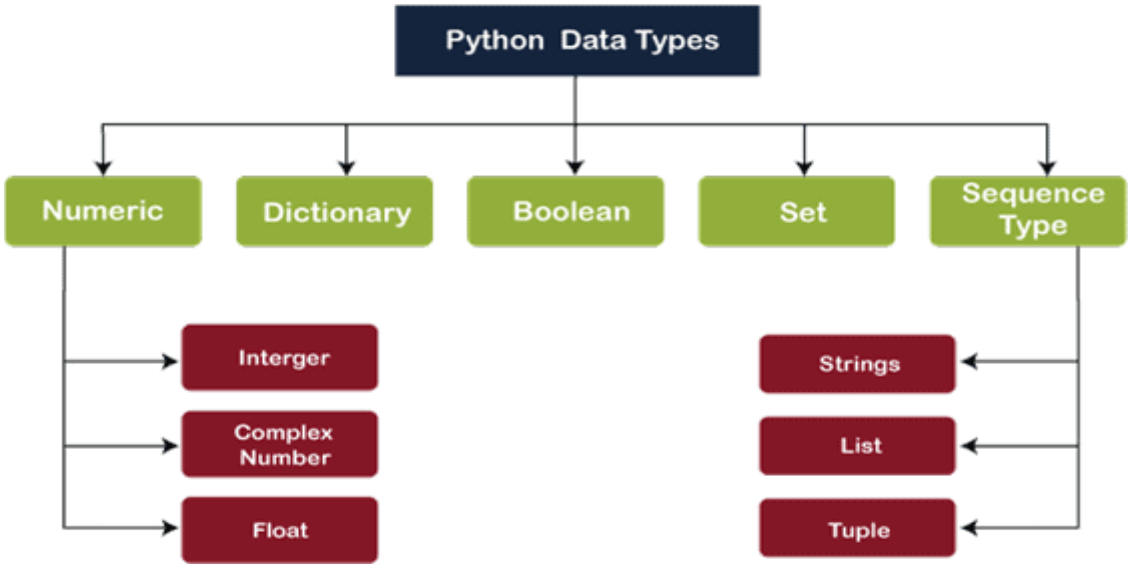
DATA TYPES:

Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.

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.

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**



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. Python

supports three types of numeric data.

Integers – This value is represented by int class. It contains positive or negative whole numbers (without fraction or decimal). 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**

Float – This value is represented by float class. It is a real number with floating point representation. It is specified by a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation. Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate upto 15 decimal points.
Complex Numbers – Complex number is represented by complex class. It is specified as *(real part) + (imaginary part)j*.

Sequence Type

In Python, sequence is the ordered collection of similar or different data types. Sequences allow to store multiple values in an organized and efficient fashion. There are several sequence types in Python

- **String**
- **List**
- **Tuple**

String

In Python, **Strings** are arrays of bytes representing Unicode characters. A string is a collection of one or more characters put in a single quote, double-quote or triple quote. In python there is no character data type, a character is a string of length one. It is represented by str class.

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'.

Creating String

Strings in Python can be created using single quotes or double quotes or even triple quotes.

Eg:

```
# with single Quotes
String1 = 'Welcome to AITS'
print("String with the use of Single Quotes: ")
print(String1)
```

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 strin

Eg

```
# Python program to demonstrate Creation of List
List = []
print("Initial blank List: ")
print(List)
List = ['MILKY']
print("\nList with the use of String: ")
print(List)
List = ["A", "For", "Apple"]
print("\nList containing multiple values: ")
print(List[0])
print(List[2])
```

OUTPUT

```
Initial blank List:

[]

List with the use of String:

['MILKY']

List containing multiple values:

A

For

Apple

>>>
```

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 ().It is represented by tuple class

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

Example :

```
# Python program to demonstrate creation of Set

# Creating an empty tuple

Tuple1 = ()

print("Initial empty Tuple: ")

print (Tuple1)

# Creating a Tuple with the use of Strings

Tuple1 = ('Milky', 'honey')

print("\nTuple with the use of String: ")

print(Tuple1)

# Creating a Tuple with the use of list

list1 = [1, 2, 4, 5, 6]

print("\nTuple using List: ")

print(tuple(list1))
```

OUTPUT

==========

```
Initial empty Tuple:

()

Tuple with the use of String:

('Milky', 'honey')

Tuple using List:

(1, 2, 4, 5, 6)

>>>
```

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'. It is denoted by the class bool.

Note – True and False with capital 'T' and 'F' are valid booleans otherwise python will throw an error.

```
# Python program to demonstrate boolean type

print(type(True))

print(type(False))

print(type(true))
```

OUTPUT

```
<class 'bool'>

<class 'bool'>

Traceback (most recent call last):

  File "C:/Users/Administrator/AppData/Local/Programs/Python/Python37/bo.py", line 4, in <module>

    print(type(true))

NameError: name 'true' is not defined

>>>
```

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.

Eg:

```
# Creating Empty set

set1 = set()

set2 = {'Hanvitha', 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 :

```
{'Hanvitha', 2, 3, 'Python'}
{2, 3, 'Hanvitha', 10, 'Python'}
{3, 'Hanvitha', 10, 'Python'}
>>>
```

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.

Creating Dictionary

In Python, a Dictionary can be created by placing a sequence of elements within curly {} braces, separated by 'comma'. Values in a dictionary can be of any datatype and can be duplicated, whereas keys can't be repeated and must be immutable. Dictionary can also be created by the built-in function dict(). An empty dictionary can be created by just placing it to curly braces{}.

Eg:

```
d = {1:'milky', 2:'Hanvitha', 3:'krishna', 4:'sweety'}

print (d)

print("1st name is "+d[1])

print("2nd name is "+ d[4])

print (d.keys())

print (d.values())
```

output :

```
{1: 'milky', 2: 'Hanvitha', 3: 'krishna', 4: 'sweety'}

1st name is milky

2nd name is sweety

dict_keys([1, 2, 3, 4])

dict_values(['milky', 'Hanvitha', 'krishna', 'sweety'])

>>>
```

TYPE CASTING

Describe about int(),float(),complex(),bool()and str() with examples:

We can convert one type value to another type. This conversion is called Typecasting or Type coersion.
The following are various inbuilt functions for type casting.

1) int()
2) float()
3) complex()
4) bool()
5) str()

int():

We can use this function to convert values from other types to int
1) >>> int(123.987)
2) 123
3) >>> int(10+5j)
4) TypeError: can't convert complex to int
5) >>> int(True)
6) 1
7) >>> int(False)
8) 0
9) >>> int("10")
10) 10
11) >>> int("10.5")
12) ValueError: invalid literal for int() with base 10: '10.5'
13) >>> int("ten")
14) ValueError: invalid literal for int() with base 10: 'ten'
15) >>> int("0B1111")
16) ValueError: invalid literal for int() with base 10: '0B1111'

float():

We can use float() function to convert other type values to float type.
1) >>> float(10)
2) 10.0
3) >>> float(10+5j)

4) TypeError: can't convert complex to float
5) >>> float(True)
6) 1.0
7) >>> float(False)
8) 0.0
9) >>> float("10")
10) 10.0
11) >>> float("10.5")
12) 10.5
13) >>> float("ten")
14) ValueError: could not convert string to float: 'ten'
15) >>> float("0B1111")
16) ValueError: could not convert string to float: '0B1111'

complex():
We can use complex() function to convert other types to complex type.
Form-1: complex(x)
We can use this function to convert x into complex number with real part x and imaginary part 0.
Eg:
1) complex(10)==>10+0j
2) complex(10.5)===>10.5+0j
3) complex(True)==>1+0j
4) complex(False)==>0j
5) complex("10")==>10+0j
6) complex("10.5")==>10.5+0j
7) complex("ten")
8) ValueError: complex() arg is a malformed string
Form-2: complex(x,y)
We can use this method to convert x and y into complex number such that x will be real part and y will be imaginary part.
Eg: complex(10, -2) □ 10-2j
complex(True, False) □ 1+0j

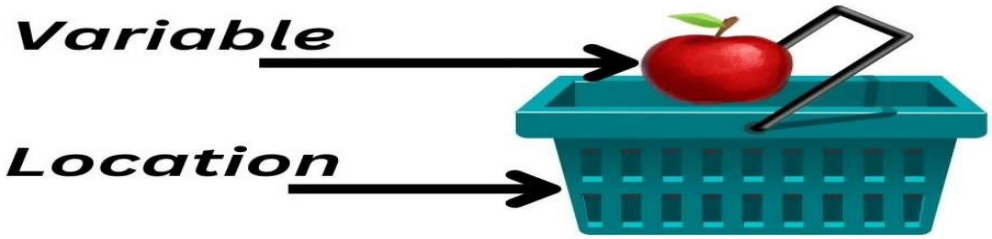
bool():
We can use this function to convert other type values to bool type.
1) bool(0) → False
2) bool(1) → True
3) bool(10) →True
4) bool(10.5) → True
5) bool(0.178) →True
6) bool(0.0) →False
7) bool(10-2j) →True
8) bool(0+1.5j) → True
9) bool(0+0j) → False
10) bool("True") → True
11) bool("False") → True
12) bool("")→ False

str():
We can use this method to convert other type values to str type.
1) >>> str(10)
2) '10'
3) >>> str(10.5)
4) '10.5'
5) >>> str(10+5j)

6) '(10+5j)'
7) >>> str(True)
8) 'True'

Python Variables

Variable is a name that is used to refer to memory location. Python variable is also known as an identifier and used to hold value.



Variables in Python can be understood with a real-world scenario, for example, a basket is required to store the Apples.

In Python, we don't need to specify the type of variable because Python is a infer language and smart enough to get variable type.

Variable names can be a group of both the letters and digits, but they have to begin with a letter or an underscore.

It is recommended to use lowercase letters for the variable name. Milky and milky both are two different variables.

Rules for creating variables in Python:

Variables are the example of identifiers. An Identifier is used to identify the literals used in the program. The rules to name an identifier are given below.

- ✓ The first character of the variable must be an alphabet or underscore (_).
- ✓ All the characters except the first character may be an alphabet of lower-case(a-z), upper-case (A-Z), underscore, or digit (0-9).
- ✓ Identifier name must not contain any white-space, or special character (!, @, #, %, ^, &, *).
- ✓ Identifier name must not be similar to any keyword defined in the language.

- ✓ Identifier names are case sensitive; for example, my name, and MyName is not the same.
- ✓ Examples of valid identifiers: **a123, _n, n_9, etc**.
- ✓ Examples of invalid identifiers: **1a, n%4, n 9, etc.**

Declaring Variable and Assigning Values

Python does not bind us to declare a variable before using it in the application. It allows us to create a variable at the required time.

We don't need to declare explicitly variable in Python. When we assign any value to the variable, that variable is declared automatically.

The equal (=) operator is used to assign value to a variable.

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("Milky")
```

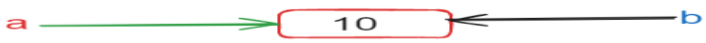
```
Milky
```

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.

```
a = 10
```

In the above image, the variable **a** refers to an integer object. Suppose we assign the integer value 10 to a new variable b.

```
a = 10

b = a
```



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

Object Identity

In Python, every created object identifies uniquely in Python. Python provides the guaranteed that no two objects will have the same identifier. The built-in **id ()** function, is used to identify the object identifier

```
a = 20

b = a

print(id(a))

print(id(b))

# Reassigned variable a

a = 200

print(id(a))
```

OUTPUT:

```
8790323986080

8790323986080

8790323991840
```

We assigned the **b = a, a** and **b** both point to the same object. When we checked by the **id()** function it returned the same number. We reassign **a** to 500; then it referred to the new object identifier.

Variable Names

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 = "Hanvitha"

age = 20

marks = 80.50

print(name)

print(age)

print(marks)
```

output:

```
Hanvitha

20

80.5
```

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.

Assigning single value to multiple variables

```
x=y=z=20

print(x)
print(y)
print(z)
```

output

20

20
20

Assigning multiple values to multiple variables:

```
a,b,c=5,6,7
print a
print b
print c
```

output :

5
6
7

Python Variable Types

There are two types of variables in Python - Local variable and Global variable.

Local Variable

Local variables are the variables that declared inside the function and have scope within the function

Eg:

```
def add():
    a = 20
    b = 30
    c = a + b
    print("The sum is:", c)
# Calling a function
add()
```

output

The sum is: 50

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.

Eg:

```
x = 100

def mainFunction():

    global x

    print(x)

    # modifying a global variable

    x = 'Welcome To Python World'

    print(x)

mainFunction()

print(x)
```

OUTPUT:

```
100

Welcome To Python World

Welcome To Python World
```

Python Comments

Python Comment is an essential tool for the programmers. Comments are generally used to explain the code. We can easily understand the code if it has a proper explanation. A good programmer must use the comments because in the future anyone wants to modify the code as well as implement the new module; then, it can be done easily.

In the other programming language such as C++, It provides the // for single-lined comment and /*.... */ for multiple-lined comment, but Python provides the single- lined Python comment. To apply the comment in the code we use the hash(#) at the beginning of the statement or code.

Types of Comments

Comments can either be

- Single-line
- Multi-line

Single-Line Comments
Python single-line comment starts with the hash tag symbol (#) with no white spaces and lasts till the end of the line. If the comment exceeds one line then put a hash tag on the next line and continue the comment. Python's single-line comments are proved useful for supplying short explanations for variables, function declarations, and expressions.

Print "Python!" to console

print("Welcome to AITS") **Multiline**

Python Comment

We must use the hash (#) at the beginning of every line of code to apply the multiline Python comment.

```
# Variable a holds value 5
# Variable b holds value 10
# Variable c holds sum of a and b
# Print the result
a = 5
b = 10
c = a+b
print("The sum is:", c)
```

We can also use the triple quotes (""") for multiline comment. The triple quotes are also used to string formatting.

Docstring Comments:
Docstrings are not actually comments, but, they are ***documentation strings***. It is used to associate documentation that has been written with Python modules, functions, classes, and methods. It is added right below the functions, modules, or classes to describe what they do. In Python, the docstring is then made available via the_doc_attribute.

```
1. def intro():
2.   """
3.   This function prints Hello milky
4.   """
5.   print("Hello Milky")
6. intro._doc _
```

Python Operators

Operators in python are used for operations between two values or variables. The output varies according to the type of operator used in the operation. Operators are the pillars of a program on which the logic is built in a specific programming language. Suppose if you want to perform addition of two variables or values, you can use the addition operator for this operation. The values in the operands can be variable or any data type that we have in python.

OPERATORS: Are the special symbols. E.g. **- +, *, /, etc.**
OPERAND: It is the value on which the operator is applied

Depending upon the type of operations there are 7 types of operators in python programming language.

Types of Operators

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

Arithmetic operators

Arithmetic operators are used to perform arithmetic calculations in python. Below are the arithmetic operators with names and their symbols. These are the symbols that we use while doing an arithmetic operation in python.

Name	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulus	%
Exponentiation	**
Floor Division	//

```
# Examples of Arithmetic Operator
a = 10
b = 2
# Addition of numbers
add = a + b
# Subtraction of numbers
sub = a - b
# Multiplication of number
mul = a * b
# Division(float) of number
div1 = a / b
# Division(floor) of number
div2 = a // b
# Modulo of both number
```

```
mod = a % b
# Power
p = a ** b
# print results
print(add)
print(sub)
print(mul)
print(div1)
print(div2)
print(mod)
print(p)
```

OUTPUT:

========

```
12
8
20
5.0
5
0
100
>>>
```

Assignment operators

Assignment operators are used to assign values to the variables or any other object in python. Following are the assignment operators that we have in python.

Operator	Description
=	It assigns the value of the right expression to the left operand.
+=	It increases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 10, b = 20 => a+ = b will be equal to a = a+ b and therefore, a = 30.
-=	It decreases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 20, b = 10

	=> a- = b will be equal to a = a- b and therefore, a = 10.
*=	It multiplies the value of the left operand by the value of the right operand and assigns the modified value back to then the left operand. For example, if a = 10, b = 20 => a* = b will be equal to a = a* b and therefore, a = 200.
%=	It divides the value of the left operand by the value of the right operand and assigns the reminder back to the left operand. For example, if a = 20, b = 10 => a % = b will be equal to a = a % b and therefore, a = 0.
**=	a**=b will be equal to a=a**b, for example, if a = 4, b =2, a**=b will assign 4**2 = 16 to a.
//=	A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign 4//3 = 1 to a.

Comparison operators

Comparison operators are used to compare two values. Following are the comparison operators that we have in python

Name	Symbol
Equal	==
Not Equal	!=
Greater than	>
Less than	<
Less than Equal to	<=
Greater than Equal to	>=

```
# Examples of Comparison Operators
a = 10
b = 20
print(a > b)
print(a < b)
print(a == b)
print(a != b)
print(a >= b)
```

```
print(a <= b)
```

OUTPUT:

```
False
True
False
True
False
True
>>>
```

Logical Operators

The logical operators are used primarily in the expression evaluation to make a decision. Python supports the following logical operators.

Operator	Description	Syntax
and	**Logical AND:** True if boththe operands are true	x and y
or	**Logical OR:** True if eitherof the operands is true	x or y
not	**Logical NOT:** True if the operand is false	not x

Examples of Logical Operator

```
a = True
b = False
print(a and b)
print(a or b)
print(not a)
```

OUTPUT:

=========

```
False
True
False
>>>
```

Membership operators

Membership operators are used to check if a sequence is present in an object. Following are the membership operators that we have in python.

in	not in
* Returns true if sequence is present in the object.	* Returns true if sequence is not present in the object.
* Example- X in Y will return true if X is present in Y.	* Example- X not in Y will return true if X is not present in Y.

```
# not 'in' operator
x = 24
y = 20
list = [10, 20, 30, 40, 50]
if (x not in list):
        print("x is NOT present in given list")
else:
        print("x is present in given list")
if (y in list):
        print("y is present in given list")
else:
        print("y is NOT present in given list")
```

OUTPUT:

==========

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

Identity Operators

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

is	is not
* Returns true if both variables are same objects	* Returns true if both variables are not same objects.
* Example- X is Y will return true if both are same objects.	* Example- X is not Y will return true if both are not same objects.

Eg:

```
a = 10
b = 20
```

```
c = a
print(a is not b)
print(a is c)
```

OUTPUT:

```
True
True
>>>
```

Bitwise Operators

The bitwise operators perform bit by bit operation on the values of the two operands. Following are the bitwise operators that we have in python.

Operator	Description
& (binary and)	If both the bits at the same place in two operands are 1, then 1 is copied to the result. Otherwise, 0 is copied.
| (binary or)	The resulting bit will be 0 if both the bits are zero; otherwise, the resulting bit will be 1.
^ (binary xor)	The resulting bit will be 1 if both the bits are different; otherwise, the resulting bit will be 0.
~ (negation)	It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting bit will be 1 and vice versa.
<< (left shift)	The left operand value is moved left by the number of bits present in the right operand.
>> (right shift)	The left operand is moved right by the number of bits present in the right operand.

Example1:

```
x = 5  # binary 0101
y = 3  # binary 0011

result = x & y   # binary 0001

print(result)    # Output: 1
```

Example2:

```
x = 5  # binary 0101
y = 3  # binary 0011

result = x | y    # binary 0111

print(result)     # Output: 7
```

Example3:

```
x = 5  # binary 0101
y = 3  # binary 0011

result = x ^ y    # binary 0110

print(result)     # Output: 6
```

Example4:

```
x = 5  # binary 0101

result = ~x    # binary -0110 (two's complement)

print(result)     # Output: -6
```

Operator Precedence and Associativity

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

Operator precedence and associativity are rules that determine the order in which operators are evaluated in an expression.

Operator precedence defines which operators are evaluated first when multiple operators are used in a single expression. Operators with higher precedence are evaluated before operators with lower precedence. In Python, the operator precedence is as follows, with operators at the top of the list having higher precedence:

- Parentheses ( )
- Exponentiation **
- Unary operators +, -
- Multiplication *, division /, modulo %, and floor division //

Addition + and subtraction -

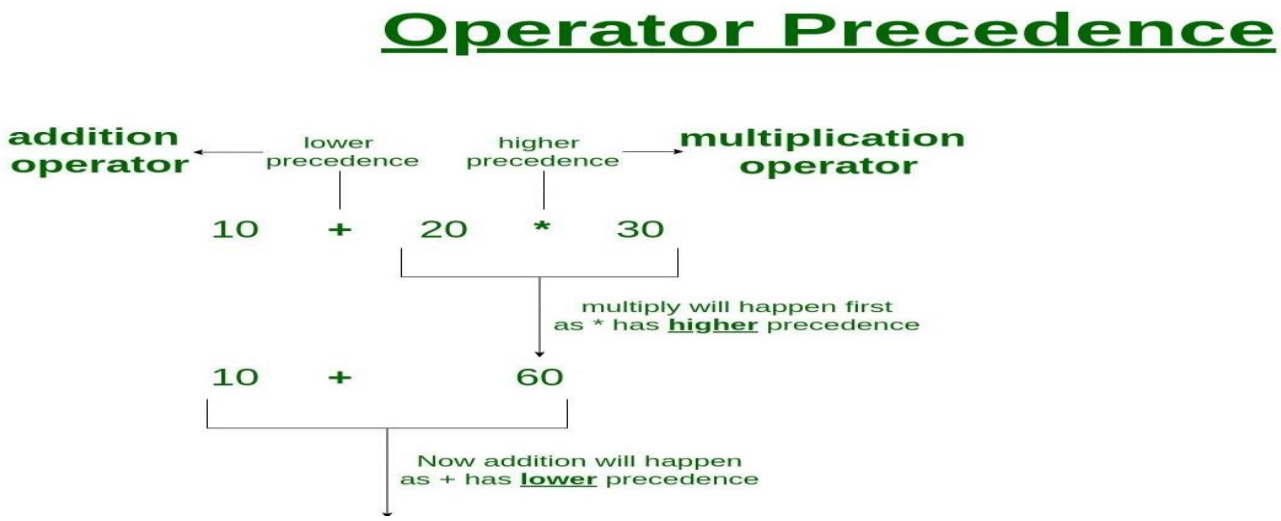
- Comparison operators <, <=, >, >=, ==, and! =

- Logical operators not, and, or

Associativity, on the other hand, determines the order in which operators of the same precedence are evaluated. In Python, most operators are left-associative, which means that expressions are evaluated from left to right. The exponentiation operator ** is right-associative, which means that expressions are evaluated from right to left.

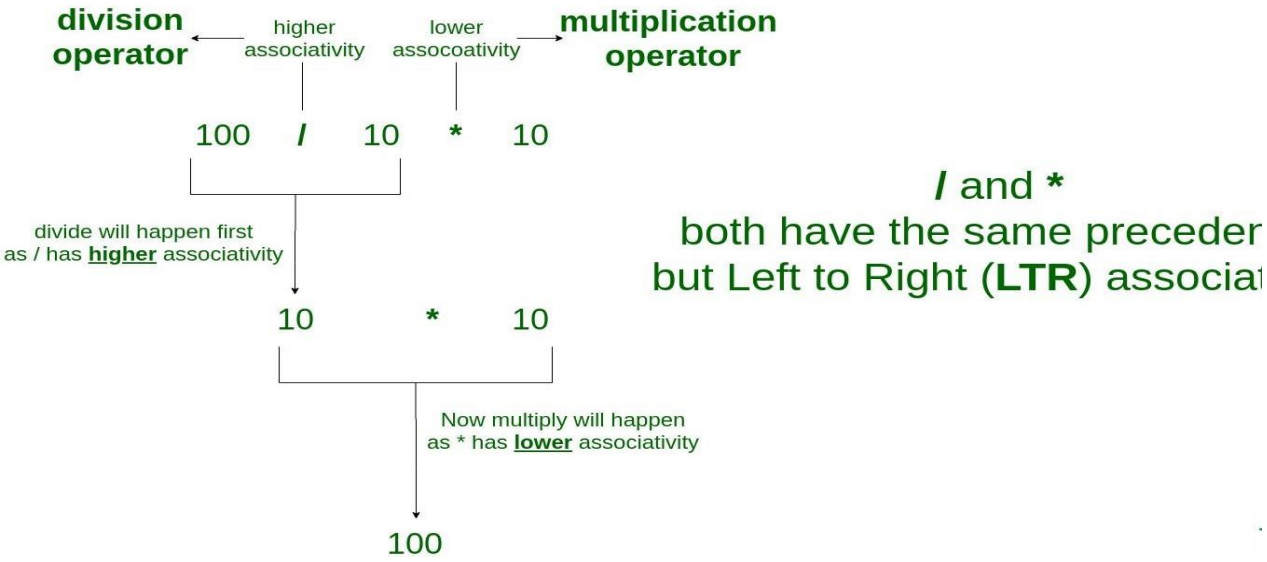
Example:

10 + 20 * 30



Operator Associativity: If an expression contains two or more operators with the same precedence then Operator Associativity is used to determine. It can either be **L**eft to **R**ight or from **R**ight to **L**eft.
Example: '*' and '/' have the same precedence and their associativity is **L**eft **t**o **R**ight, so the expression "100 / 10 * 10" is treated as "(100 / 10) * 10".

Operator Associativity



Operator precedence and Associativity Tabular format:

Operator	Description	Associativity
( )	Parentheses	left-to-right
**	Exponent	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ –	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
is, is not in, not in	Identity Membership operators	left-to-right
&	Bitwise AND	left-to-right

Operator	Description	Associativity
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
not	Logical NOT	right-to-left
and	Logical AND	left-to-right
or	Logical OR	left-to-right
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left

Example1:

```
a = 2 + 3 * 4  # Multiplication has higher precedence than addition, so 3 * 4 is evaluated first, then added to 2
print(a)       # Output: 14

b = (2 + 3) * 4       # Using parentheses to override precedence: addition is evaluated first, then multiplied by 4
print(b)       # Output: 20
```

Input and Output Functions

A program needs to interact with the user to accomplish the desired task; this can be achieved using **Input-Output functions**. The **input ()** function helps to enter data at run time by the user and the output function **print ()** is used to display the result of the program on the screen after execution.

1. The print() function

In Python, the **print ()** function is used to display result on the screen. The syntax for **print ()** is as follows:

Syntax :

```
print (“string to be displayed as output ” )

print (variable )

print (“String to be displayed as output ”, variable)

print (“String1 ”, variable, “String 2”, variable, “String 3” ……)
```

OR

Python provides the print() function to display output to the standard output devices.

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

Parameters:

Value: Any value, and as many as you like. Will be converted to string beforeprinted

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

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

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

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

Returns: It returns output to the screen.

Formatting Output

Formatting output in Python can be done in many ways.We can use formatted string literals**,** by starting a string with f or F before opening quotation marks or triple quotation marks. In this string, we can write Python expressions between **{** and **}** that can refer to a variable or any literal value.

```
# Declaring a variable

name = "Milky"

# Output

print(f'Hello {name}! How are you?')
```

OUTPUT:

```
Hello Milky! How are you?

>>>
```

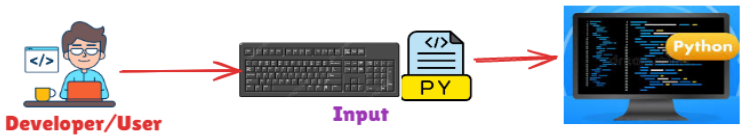
2. input() function

In Python, **input ( )** function is used to accept data as input at run time. The syntaxfor **input ()** function is,

```
Variable = input (“prompt string”)
```

Where, **prompt string** in the syntax is a statement or message to the user, to know what input can be given.

If a prompt string is used, it is displayed on the monitor; the user can provide expected data from the input device. The **input ( )** takes whatever is typed from the keyboard and stores the entered data in the given variable.



 # Taking input from the user

```
name = input("Enter your name: ")

print("Hello, " + name)
```

OUTPUT:

```
Enter your name: milky

Hello, milky

>>>
```

#Example 2:

```
x = int (input("Enter Number 1:"))

y = int (input("Enter Number 2:"))

print ("The sum = ", x+y)
```

OUTPUT:

```
Enter Number 1:20
```

```
Enter Number 2:10

The sum =  30
```

Using format ()

We can also use format () function to format our output to make it look presentable. The curly braces **{ }** work as placeholders. We can specify the order in which variables occur in the output.

 # initializing variables

```
a = 20

b = 10

# Addition

sum = a + b

# subtraction

sub = a- b

# Output
```

```
print('The value of a is {} and b is {}'.format(a,b))

print('{2} is the sum of {0} and {1}'.format(a,b,sum))
```

OUTPUT:

```
The value of a is 20 and b is 10

30 is the sum of 20 and 10
```

Using % Operator

We can use **'%'** operator. % values are replaced with zero or more value of elements. The formatting using % is similar to that of 'printf' in the C programming language.

- %d – integer
- %f – float
- %s – string
- %x – hexadecimal
- %o – octal

 # Taking input from the user

```
num = int(input("Enter a value: "))
add = num + 5
# Output
print("The sum is %d" %add)
```

OUTPUT:

```
Enter a value: 15

The sum is 20
```

UNIT-II

CONTROL STRUCTURES: Boolean Expressions (Conditions), Selection Control, Iterative Control andExamples.

LISTS: List Structures, Lists (Sequences) in Python, Iterating Over Lists (Sequences) in Python with Examples. Tuple data Type

CONTROL STRUCTURES: Boolean Expressions(Conditions)

Boolean Expressions

Python bool () condition is used to return or convert a value to a Boolean value i.e., True or 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'.

Syntax: bool ([x])

bool () parameters

The bool() method in general takes only one parameter(here x), on which the standard truth testing procedure can be applied. If no parameter is passed, then by default it returns False. So, passing a parameter is optional.

- ✓ Return value from bool ()
- ✓ It can return one of the two values.
- ✓ It returns True if the parameter or value passed is True.
- ✓ It returns False if the parameter or value passed is False.

Here are a few cases, in which Python's bool() method returns false. Except these all other values return True.

- ✓ If a False value is passed.
- ✓ If None is passed.
- ✓ If an empty sequence is passed, such as (), [], ", etc
- ✓ If Zero is passed in any numeric type, such as 0, 0.0 etc
- ✓ If an empty mapping is passed, such as {}.
- ✓ if Objects of Classes having bool() orlen()method, returning 0 or False

Example:

```
#case:1
x = False
print(bool(x)) #False
#case:2
x = True
print(bool(x))  #True
#case:3
```

```
x = 5
y = 10
print(bool(x == y)) #False
#case:4
x = None
print(bool(x))  #False
#case:5
x = ()
print(bool(x)) #False
#case:6
x = {}
print(bool(x)) #False
#case:7
x = 0.0
print(bool(x)) # False
#case:8
x = 'Hanvitha'
print(bool(x))
```

output:

True

Python indentation

In Python, indentation plays a crucial role in defining the structure and flow of the code. Unlike many other programming languages that use braces {} or keywords like begin and end to denote blocks of code, Python uses indentation to indicate blocks of code.

The key points about Python indentation are:

1. Indentation Levels: Python code blocks are defined by their indentation level. A block of code with the same level of indentation is considered part of the same block.

2. Whitespace: Python uses whitespace (spaces or tabs) to define indentation.

While either spaces or tabs can be used for indentation, it's recommended to be consistent within your codebase. Most Python style guides, including PEP 8 (the official Python style guide), recommend using four spaces for each indentation level.

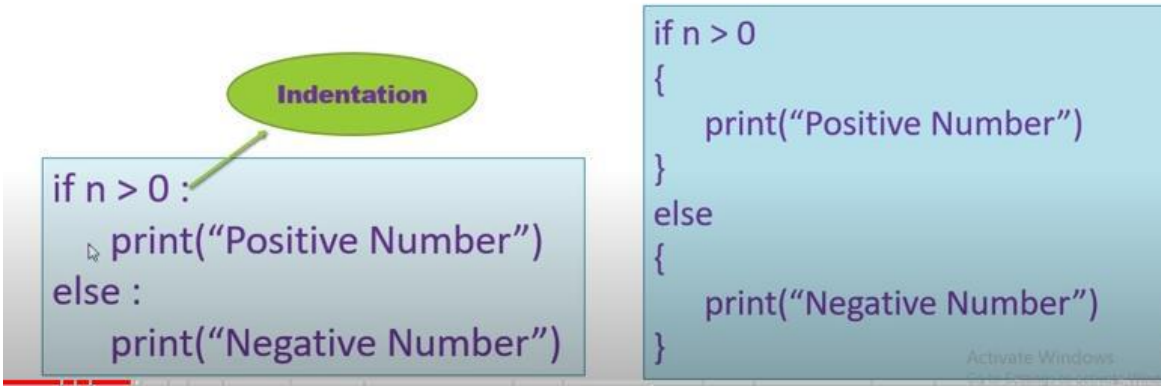
3. Indentation Errors: Incorrect indentation can lead to syntax errors or unintended behavior in Python programs.

For example, if the indentation within a block is inconsistent, Python raises an Indentation Error. It's essential to ensure that all lines within the same block have the same indentation level.

4. No Explicit Block Delimiters: Unlike many other programming languages, Python

does not use explicit block delimiters (such as braces {}) to mark the beginning and end of blocks. Instead, the indentation level determines the structure of the code. This makes Python code more readable but also requires careful attention to indentation.

- Python requires indentation as a part of syntax.
- Indentation signifies the start and end of block of code.
- Programs will not run without correct indentation.

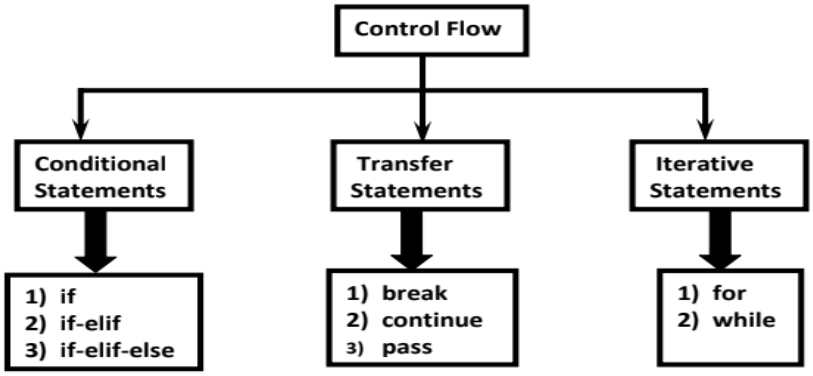


Selection Control, Iterative Control and Examples.

Python Control Statements

Control statements are used to control the flow of program execution. These are classified into three types.

1. Conditional or Decision Making or Selection Statements.
2. Looping or Iterative Statements.
3. Transfer or Jump Statements.



Python Selection Statements

In Python, the selection statements are also known as decision making statements or branching statements. The selection statements are used to select a part of the program to be

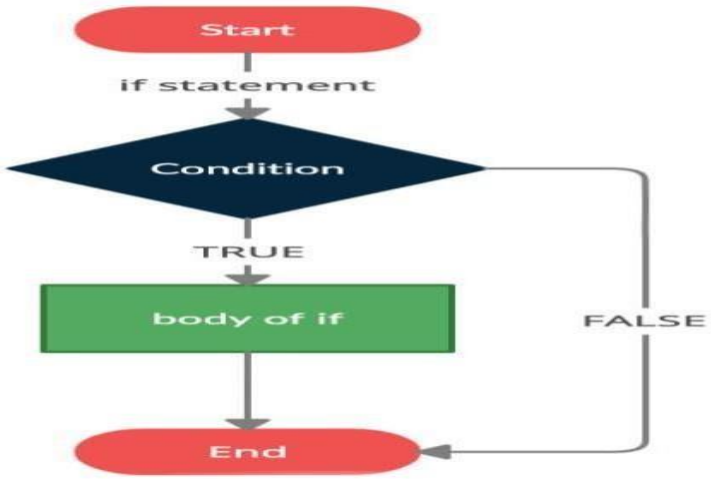
executed based on a condition. Python provides the following selection statements.

- if statement
- if-else statement
- if-elif statement

if statement in Python

The Python if statement is same as it is with other programming languages.

It executes a set of statements conditionally, based on the value of a logical expression



The general syntax of if statement in Python is as follows

```
If condition:
    Statement-1
    Statement-2
    ..............
```

In the above syntax, expression specifies the conditions it produces either true or false. If the expression evaluates true then the same amount of indented statement(s) following it will be executed. This group of the statement(s) is called a block

```
marks = int(input("Enter your marks to know pass or failed: "))
if marks >= 35:
print("You are passed ...")
```

Output:

```
Enter your marks to know pass or failed: 45
You are passed...
```

if -- else Statement:

In Python if-else statement has two blocks, first block follows the expression and the other block follows the else clause. Here is the syntax.

```
if expression :
```

```
    statement1
else :
    statement2
```

In the above case, if the expression evaluates to true then the same amount of indented statements(s) executed. if the expression evaluates to false then same amount of indented statements(s) follow the else block will executed.

Example1: Write python script to check whether the given number is even or odd number?

```
a = int(input('Enter Your number: '))
if a % 2 == 0:
    print(a," is a even number")
else:
   print(a," is a odd number")
```

Output:

```
Enter Your number: 20
20 is a even number
Or
Enter Your number: 21
21 is a odd number
```

Write a Program to check whether the given Number is in between 1 and 100?

```
n=int(input("Enter Number:"))
if n>=1 and n<=10 :
    print("The number",n,"is in between 1 to 10")
else:
   print("The number",n,"is not in between 1 to 10")
```

3) if-elif-else:

```
if condition1:
   statement-1
elif condition2:
   statement-2
...
else: statement
```

Based condition the corresponding action will be executed.

Example1: How to check a Student result based on the marks ?

```
marks = int(input("Enter your percentage of marks : "))
if marks < 35 and marks > 0:
      print("You are failed")
elif marks >= 35 and marks < 50:
      print("Your got 3rd class")
elif marks >= 50 and marks < 60:
      print("You got 2nd class")
elif marks >= 60 and marks < 72:
      print("you got 1st class")
else:
     print("you got distinction")
```

Output:

```
Enter your percentage of marks : 50
You got 2nd class
Enter your percentage of marks : 90
you got distinction
Enter your percentage of marks : 20
You are failed
```

Example 2: How to find the biggest value of three given values ?

```
a = int(input("Enter First Value: "))
b = int(input("Enter Second Value: "))
c = int(input("Enter third Value: "))
if a > b and a > c:
      print(a,'is greater then ',b, 'and',c)
elif b > c:
     print(b,'is greater then ',a, 'and',c)
else:
      print(c,'is greater then ',a,'and',b)
```

Output 1:

```
Enter First Value: 10
Enter Second Value: 20
Enter third Value: 15
20 is greater then 10 and 15
```

Write a python program to read a number and display the respective value in word format?

```
num = int(input("Enter a number: "))
if num == 0:
  print("Zero")
elif num == 1:
  print("One")
elif num == 2:
  print("Two")
elif num == 3:
  print("Three")
elif num == 4:
  print("Four")
elif num == 5:
  print("Five")
elif num == 6:
  print("Six")
elif num == 7:
  print("Seven")
elif num == 8:
  print("Eight")
elif num == 9:
  print("Nine")
else:
   print("Number out of range.")
```

Python Looping Statements

In Python, the iterative statements are also known as looping statements or repetitive statements. The iterative statements are used to execute a part of the program repeatedly as long as a given condition is True. Python provides the following iterative statements.

- While statement
- for statement

While statement

In Python, the while statement is used to execute a set of statements repeatedly. In Python, the while statement is also known as entry control loop statement because in the case of the while statement, first, the given condition is verified then the execution of statements is determined based on the condition result

```
while (condition):
   statement1
```

```
    statement2
    statement3
    statement4


num = 1
while(num <= 5):
    print("the count is: ",num)
    num += 1 # num = num + 1
```

Output

```
the count is: 1
the count is: 2
the count is: 3
the count is: 4
the count is: 5
```

Q) WAP to display 0 to n number Square numbers?

```
num = int(input('enter any number :'))
i = 0
while ( i < num+1 ):
        print('Square of '+str(i) +' is '+str(i *2))
         i += 1
```

Output:

```
Enter any number: 10
Square of 0 is 0
Square of 1 is 1
Square of 2 is 4
Square of 3 is 9
Square of 4 is 16
Square of 5 is 25
Square of 6 is 36
Square of 7 is 49
Square of 8 is 64
Square of 9 is 81
Square of 10 is 100
```

While statement with 'else' clause in Python

In Python, the else clause can be used with a while statement. The else block is gets executed whenever the condition of the while statement is evaluated to false. But, if the while loop is terminated with break statement then else doesn't execute.

```
count = int(input('How many times you want to say "Hello": '))
i = 1
while i <= count: if count > 10:
      print('I cann\'t say more than 10 times!')
      break
print('Hello')
i += 1
else:
   print('Job is done! Thank you!!')
```

Range Data Type:

Range Data Type represents a sequence of numbers. The elements present in range Data type are not modifiable. i.e range Data type is immutable.

Form-1: range (n)

Here **n** means o to n-1

Eg:

```
r = range (10)
for i in r : print(i) #0 to 9
```

To print output in same line (horizontal) format then use end=' ' attribute inside print () method.

```
for i in range(10):
print (i, end=' ')
```

Output

```
0 1 2 3 4 5 6 7 8 9
```

Form-2: range (bengin-value,end-value)

Eg:

```
r = range (10, 20)
for i in r : print(i) #10 to 19
```

Form-3: range (begin,end,inc/dec)

Eg:

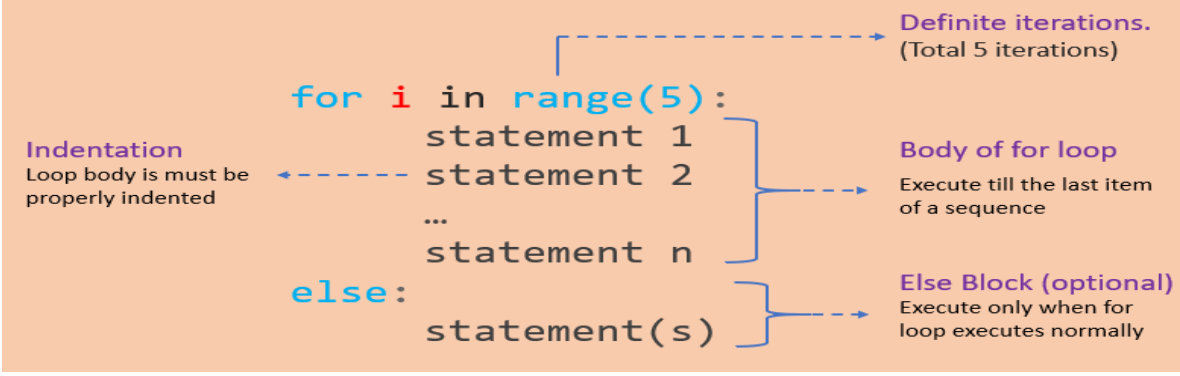
```
r = range(10,20,2)
for i in r : print(i) # 10,12,14,16,18
```

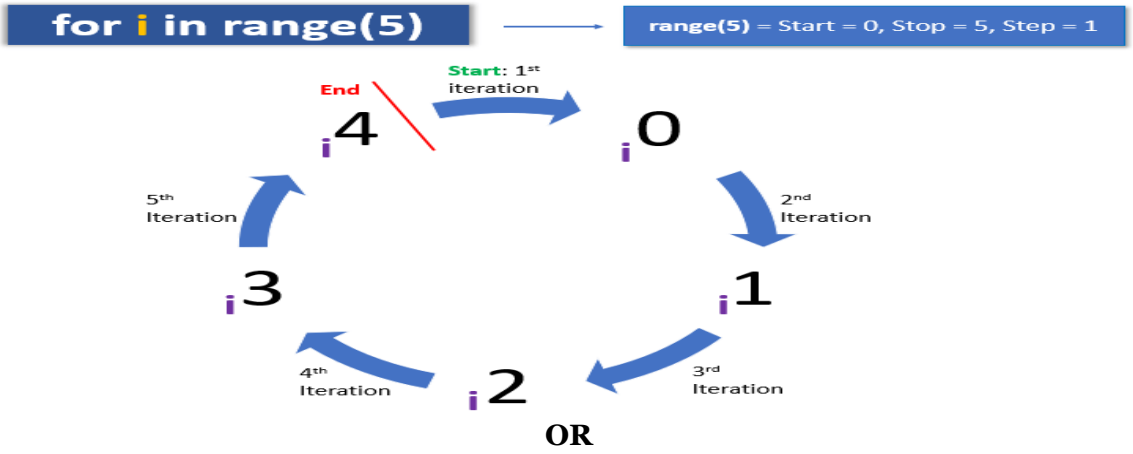
For loops:-

For loop is a programming language statement, i.e. an iteration statement, which allows a code block to be repeated a certain number of times.

Python for loop

A for loop is **used for iterating over a sequence and iterables** (like range, list, a tuple, a dictionary, a set, or a string).



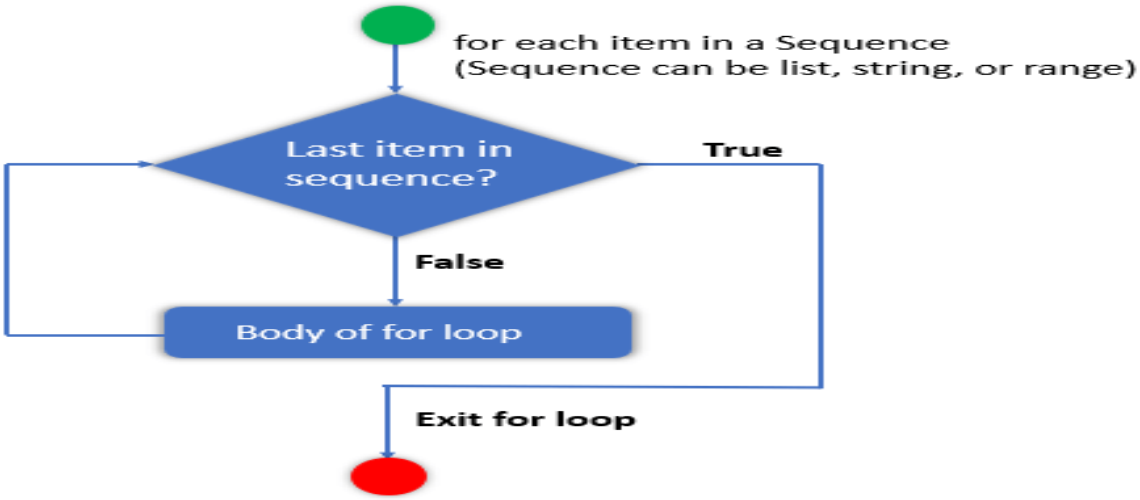


OR

Syntax: for variable in sequence:

statement1

statement2



Write a Python program that prints multiplication table of a given number

```
num = int(input("Enter a number: "))
print("Multiplication Table of", num)
for i in range(1, 11):
    print(num, "x", i, "=", num * i)
```

Output

```
Enter a number: 10
Multiplication Table of 10
10 x 1 = 10
10 x 2 = 20
10 x 3 = 30
10 x 4 = 40
10 x 5 = 50
10 x 6 = 60
10 x 7 = 70
10 x 8 = 80
10 x 9 = 90
10 x 10 = 100
```

For statement with 'else' clause in Python

In Python, the else clause can be used with a for a statement. The else block is gets executed whenever the for statement is does not terminated with a break statement. But, if the for loop is terminated with break statement then else block doesn't execute.

```
for i in range(1, 11):
   if i == 5:
      continue
   print(i,end=" ")
else:
   print("\nLoop completed successfully.")
```

Transfer Statements

Transfer statements are used to transfer the program control from one location to another location. We have different types of transfer statements,

1. break
2. continue
3. pass

1. break

- ✓ Break is a keyword which is used only in looping statements.

- ✓ Whenever break occurs it will stop entire iteration and control goes outside the loop.

2. Continue

Continue is a keyword which is used only in looping statements. Whenever continue occurs it will stop current iteration and executes from next iteration onwards.

For example:

```
for i in range(1,10):
    if(i % 2 == 0):
        continue
    if(i % 5 == 0):
        break
print(i,end=' ')
```

Output

1 3

3. pass

Pass is a keyword which is used in program at the time of partial development of code.

Example:

```
if 10 > 5:
pass
```

Assert

- ✓ Assert is a keyword which is used to generate exceptions.
- ✓ If you want to execute certain statements only when the condition is satisfied otherwise program has to stop with error message.

Example:

```
n = int(input("Enter any number : "))
assert n > 0, "Invalid number"
print("Your number is accepted")
print("We will process your request ")
```

Example:

- ✓ The following example uses the assert statement in the function.

```
def square(x):
assert x>=0, 'Only positive numbers are allowed'
return x*x
n = square(2) # returns 4
n = square(-2) # raise an AssertionError
```

Output

```
Traceback (most recent call last):
assert x > 0, 'Only positive numbers are allowed'
AssertionError: Only positive numbers are allowed
```

Return

- ✓ return is a keyword which is used in functions concept.
- ✓ if you want to transfer any value in functions then we have to use return keyword.
- ✓ in functions we can return single value as well as multiple value.

Example:

```
def sum():
a = 10;
b = 20;
c = a + b
print(c)
return None
result = sum()
print("sum is:", result )
```

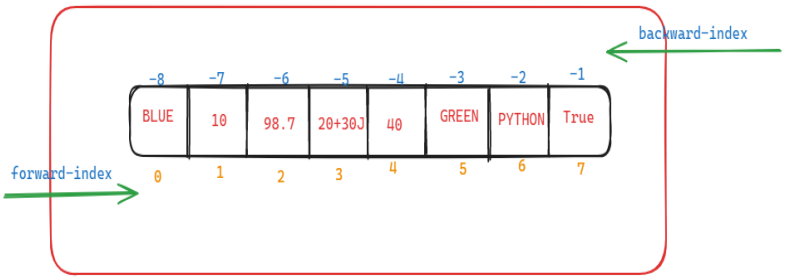
Output:

```
30
Sum is: None
```

LISTS: List Structures, Lists (Sequences) in Python, Iterating Over Lists (Sequences) in Python with Examples

A list in Python is used to store the sequence of various types of data. Python lists are mutable type its mean we can modify its element after it created. However, Python consists of six data-types that are capable to store the sequences, but the most common and reliable type is the list.

A list can be defined as a collection of values or items of different types. The items in the list are separated with the comma (,) and enclosed with the square brackets [].



- A list is a collection of elements. These elements may be homogeneous (same types) or heterogeneous (different types).
- A list also allows duplicate elements.
- Insertion order is preserved in list.
- List elements are separated by commas and enclosed within square brackets [10,20,'abc'].
- Every element in the list has its own unique index number.
- List supports both forward indexing and backward indexing, forward index starts from 0 and backward index starts from -1.
- We access either specific element by using "indexing" or set of elements by using "slicing" from the List.
- For example, l[0] --->> indexing l[0:3] ----->> slicing
- We can create list in different ways.
    a. By using "list()" function,
    b. By using square brackets []
    c. By using range() function.
- List objects are mutable means we can change the list elements.

Creating List by using list():

1. This list() allows only one string value with set of characters.

2. If we give int type data in the list() function then interpreter will throw '**TypeError**' error.

Example:

```
>>> List1 = list() #creating empty list
>>> print(List1) []
>>> type(List1) <class 'list'>
>>> List1 = list('python') #creating list with set of characters
>>> print(List1) ['p', 'y', 't', 'h', 'o', 'n']
```

>>> type(List1) <class 'list'>

Creating list by using square brackets []

List1 = [] #creating empty list

>>> print(List1) []

>>> type(List1) <class 'list'>

>>> **List1 = [1,2,3,4,5] #creating list with homogeneous elements**

>>> print(List1) [1, 2, 3, 4, 5]

>>> type(List1) <class 'list'>

>>> List1 = [10,11,'Python',5.5,True,2+3j] #creating list with heterogeneous elements

>>> print(List1) [10, 11, 'Python', 5.5, True, (2+3j)]

>>> type(List1) <class 'list'>

Creating list by using range() function:

We can also use range() function to create list.

Syntax: range(StartingIndexValue, LastValue-1, RangeValue)

For example: range(10)

a. Here, both StartingIndexValue and RangeValue are optional.

b. The default StartingIndexValue is 0.

c. The default RangeValue is 1.

Example:

>>> List1 = list(range(10))

>>> print(List1) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> type(List1) <class 'list'>

>>> List1 = list(range(0,10))

>>> print(List1) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> type(List1) <class 'list'>

>>> List1 = list(range(2,8))

>>> print(List1) [2, 3, 4, 5, 6, 7]

>>> type(List1) <class 'list'>

List Indexing

1. By using list indexing we can fetch specific element from the list.

2. It supports both forward and backward indexing

0 1 2 3 4 5 6

List1 =[10, 20, 30, 'Python', True, 1.5, 2+3j]

-7 -6 -5 -4 -3 -2 -1

Example:

>>> List1 = [10,20,30,'Python',True,1.5,2+3j]

>>> print(List1[0]) 10

>>> print(List1[1]) 20

List Slicing

- By using slicing we can fetch set of characters from list.
- It also supports both forward and backward indexing
- **Colon (:)** is the slicing operator.

Example:

>>> List1 = [10,20,30,'Python',True,1.5,2+3j]

>>> print(List1[0:2]) [10, 20]

>>> print(List1[2:5]) [30, 'Python', True]

NOTE : List is a "mutable" object that means we can update or replace the existing list with new elements. But id reference value is not changed.

List Concatenation:

Python supports concatenating two or more lists into single list.

Example:

>>> list1 = [10,20,50]

>>> list2 = [70,10,20,50]

>>> list1 + list2 # [10, 20, 50, 70, 10, 20, 60]

List Multiplication or List Repetition

Python supports multiplying the given list into N number of times.

Example:

>>> List1 = [10,'Python',5.5]

>>> List2 = List1*3

>>> print(List2) # [10, 'Python', 5.5, 10, 'Python', 5.5, 10, 'Python', 5.5

List functions or methods:

1. len():

This function counts number of elements in the list.

Example:

>>> List1 = [10,20,'Python',30,True,'milky',10,3+4j]

>>> len(List1) # 8

2. count(element):

This function counts the no.of occurrences of specific element in the list.

Example:

>>> List1 = [10,20,'Python',30,True,'milky',10,3+4j]

>>> List1.count(10) #2

Example:

>>> lst = [1,2,3,True,False,1,0,False]

>>> lst.count(1) 3

Example:

>>> lst = [1,2,3,True,False,1,0,False,1+9j,1.1]

>>> lst.count(0) 3

>>> lst.count(False) 3

3. index(object, index_value):

This function finds the index value for specific element.

Example:

>>> List1 = [10,20,'Python',30,True,'milky',10,3+4j]

>>> List1.index(10) 0

Example:

>>> lst = [1,2,3,True,False,1,0,False,1+9j,1.1]

>>> lst.index(1,1) 3

Example:

>>> List1 = [10,20,'Python',30,True,'milky',10,3+4j]

>>> List1.index(10,1) # 6

Index on nested list:

>>> lst = [100, True, 1, 2, 3, 4, 5, 0, [8, 9], 10, 20, 4]

>>> lst[8].index(9) # 1

>>> lst[8].index(8) # 0

0 1 2 3 4 5 6 7 8 9 10 11

lst = [100, True, 1, 2, 3, 4, 5, 0, [8, 9], 10, 20, 4]

-12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

4. append():

This function adds new element at the end of the existing list.

Example:

>>>List1 = [10,20,'Python',30,True,'milky',10,3+4j]

>>> List1.append(50)

>>> print(List1) # [10, 20, 'Python', 30, True, 'milky', 10, (3+4j), 50]

Example:

>>> List1 = [10,20,'Python',30,True,'milky',10,3+4j]

>>> List1.append(60)

>>> print(List1) # [10, 20, 'Python', 30, True, 'milky', 10, (3+4j), 60]

NOTE:

We can also add multiple elements in the list by using append method but those multiple elements work like nested list or sub list in the existing list and also getting single index value for this sub list

```
>>> List1 = [10,20,'Python',30,True,'milky',10,3+4j]
>>> List1.append([0,1,2])
>>> print(List1)
[10, 20, 'Python', 30, True, 'milky', 10, (3+4j), [0, 1, 2]]
```

5. extend():

This function adds multiple elements at the end of the existing list as multiple elements

Example:

```
>>>List1=[10,20,'Python',30,True,'milky',10,3+4j]
>>> List1.extend([70,80,90])
>>> print(List1)
[10, 20, 'Python', 30, True, 'milky', 10, (3+4j), 70, 80,90]
```

6. insert(index_position, object):

This function adds an element at any required index place in the existing list.

Example:

```
>>>List1=[10,20,'Python',30,True,'Srinivas',10,3+4j]
>>> List1.insert(2,100)
>>> print(List1)
[10, 20, 100, 'Python', 30, True, 'milky', 10, (3+4j)]
```

Example:

```
>>> List1=[10,20,'Python',30,True,'milky',10,3+4j]
>>> List1.insert(3,'Durga')
>>> print(List1)
[10, 20, 'Python', 'krishna', 30, True, 'milky', 10, (3+4j)]
```

We can also add multiple elements in the list at required place by using insert method but those multiple elements work like nested list or sub list in the existing list and it contains single index value.

```
>>> List1=[10,20,'Python',30,True,'milky',10,3+4j]
>>> List1.insert(0,[0,1,2,3])
>>> print(List1)
[[0, 1, 2, 3], 10, 20, 'Python', 30, True, 'milky', 10, (3+4j)]
```

7. remove (object):

This function removes specific element in the existing list. This function allows one argument and that should be element name. Search for removing element from 0 index onwards and removing first occurrence.

Example:

```
>>> List1 = [10,20,'Python',30,True,'milky',10,3+4j]
>>>List1.remove(10)
```

```
>>> print(List1)
[20, 'Python', 30, True, 'milky', 10, (3+4j)]
```

Example:

```
>>> List1 = [10,20,'Python',30,True,'milky',10,3+4j]
>>> List1.remove(True)
>>> print(List1)
[10, 20, 'Python', 30, 'milky', 10, (3+4j)]
```

Example:

```
>>> l = [1, 2, 3, 1, 3]
>>> l
[1, 2, 3, 1, 3]
>>> l.remove(1) # only first occurence element deleted
>>> l
[2, 3, 1, 3]
>>> l.remove(6)
```

ValueError: list.remove(x): x not in list

Note : if value is not in given list then it returns ValueError

```
>>> l = [10, 30, 10]
>>> l.remove()
```

TypeError: remove() takes exactly one argument (0 given)

8. pop(index):

This function removes specific element from given list. This function also allows only one argument and that should be index value for an element. If index value not given then by default last element removed automatically from given list and also returns this removed value.

Example:

```
>>> List1 = [10,20,'Python',30,True,'milky',10,3+4j]
>>> List1.pop(1) # 20
>>> print(List1) # [10, 'Python', 30, True,'milky', 10, (3+4j)]
```

Example:

```
>>> List1 = [10,20,'Python',30,True,'milky',10,3+4j]
>>> List1.pop(2) # 'Python'
>>> print(List1) # [10, 20, 30, True, 'milky', 10, (3+4j)]
```

Example:

```
>>> l = [10,30,60,10,40]
>>> l.pop()
40
```

```
>>> l
[10, 30, 60, 10]
>>> l.pop(2) # index position 2 value removed and returns also.
60
>>> l
[10, 30, 10]
```

Example:

```
>>> l = [10, 30, 10]
>>> id(l)
67974328
>>> l2 = l.pop() # removed value store in l2.
>>> l
[10, 30]
>>> id(l)
67974328
>>> l2
10
>>> id(l2)
1834313024
```

9. reverse ():

This function reverses the existing list elements.

Example:

```
>>> List1 = [10,20,'Python',30,True,'milky',10,3+4j]
>>> List1.reverse()
>>> print(List1) # [(3+4j), 10, 'milky', True, 30, 'Python', 20, 10]
```

Example:

```
>>> l = [10,40,60,20,30,40]
>>> l2 = l.reverse() # reverse the elements in given list only but not assign to l2
>>> l2
>>> # no output returns l2.
>>> l
[40, 30, 20, 60, 40, 10]
>>>
```

10. reversed(object) :

This method reverse the given list elements but returns the list_reverseiterator object

Example:

```
>>> l = [40, 30, 20, 60, 40, 10]
```

```
>>> reversed(l)
<list_reverseiterator object at 0x040EAE10>
---->> To get elements from this list_reverseiterator object then use list() method.
>>> list(reversed(l))
[10, 40, 60, 20, 30, 40]
```

10. copy():

This function copies the existing list into new variable.

Example:

```
>>> List1 = [10,20,'Python',30,True,'milky',10,3+4j]
>>> x = List1.copy()
>>> print(List1) # [10, 20, 'Python', 30, True, 'milky', 10, (3+4j)]
>>> print(x) # [10, 20, 'Python', 30, True, 'milky', 10, (3+4j)]
```

After copying the given list elements into another list gets new object reference.

Example:

```
>>> l = [40, 30, 20, 60, 40, 10]
>>> l2 = l.copy()
>>> l2
[40, 30, 20, 60, 40, 10]
>>> id(l)
67976448
>>> id(l2)
63174056
```

11.clear():

This function clears the entire elements from the given list.

Example:

```
>>> List1 = [10,20,'Python',30,True,'Srinivas',10,3+4j]
>>> List1.clear()
>>> print(List1) []
>>> del List1
```

12.max():

This functions finds the maximum value in the given list

Example:

```
>>>List2=[10,20,30,40]
>>> max(List2) 40
```

13.min():

This function finds the minimum value in the given list

Example:

>>> List2=[10,20,30,40]

>>> min(List2) 10

14. sort():

This function sorts the elements of given list.

```
lst=[1,9,5,11,2]
lst.sort()
print(lst) [1, 2, 5, 9, 11]
l1=[1,2,5,3,7,4,2,True]
l1.sort()
print(l1) [1, True, 2, 2, 3, 4, 5, 7]
```

Note: sort() method sorts the elements in given list only but not assign to new variable.

Example:

>>> l = [10,40,60,30,20]

>>> l2 = l.sort()

>>> l2

>>>

>>> l

[10, 20, 30, 40, 60]

Note: by default this function sorts in ascending order, we can also get in descending order by setting True for reverse.

Example:

>>> lst=[1,9,5,11,2]

>>> lst.sort(reverse=True)

>>> print(lst) [11, 9, 5, 2, 1]

Example:

>>> l1=[1,2,5,3,7,4,2,True]

>>> l1.sort(reverse=True)

>>>print(l1) [7, 5, 4, 3, 2, 2, 1, True]

del command:

This command is used to remove any specific element in the list or to remove entire list object permanently.

Removing specific element:

>>> List1 = [10,20,'Python',30,True,'milky',10,3+4j]

>>> del List1[0]

>>> print(List1) # [20, 'Python', 30, True, 'milky', 10, (3+4j)]

>>> List1 = [10,20,'Python',30,True,'milky',10,3+4j]

>>> del List1[4]

>>> print(List1) [10, 20, 'Python', 30, 'milky', 10, (3+4j)]

Removing entire list object:

Example:

```
>>> List1 = [10,20,'Python',30,True,'milky',10,3+4j]
>>> del List1 # deleting lst
>>> print(List1) # after deleting
```

NameError: name 'lst' is not defined

Nested List:

```
Python supports Nested lists also, it means a list contains another lists.
List1 = [10,'Python',5.5]
List2 = [20,30,'milky',3+4j]
List3 = [1,True,2,'krishna']
print(List1) #[10, 'Python', 5.5]
print(List2) #[20, 30, 'milky', (3+4j)]
print(List3) #[1, True, 2, 'krishna']
 NestList=[List1,List2,List3] #creating a list by using existing lists.
print(NestList)
[[10, 'Python', 5.5], [20, 30, 'milky', (3+4j)], [1, True, 2,'krishna']]
type(NestList) <class 'list'>
print(NestList[0]) #[10, 'Python', 5.5]
print(NestList[1]) #[20, 30, 'milky', (3+4j)]
print(NestList[2]) #[1, True, 2, 'krishna']
print(NestList[0][0]) #10
print(NestList[1][0]) #20
print(NestList[2][0]) #1
```

Converting a String into List :

```
Str1 = 'Python is very simple and easy language'
print(Str1) # Python is very simple and easy language
type(Str1) # <class 'str'>
List1 = Str1.split()
print(List1) # ['Python', 'is', 'very', 'simple', 'and', 'easy', 'language']
type(List1) # <class 'list'>
```

Converting a List to String :

```
List1=['Python', 'is', 'very', 'simple', 'and', 'easy', 'language']
print(List1) # ['Python', 'is', 'very', 'simple', 'and', 'easy', 'language']
type(List1) <class 'list'>
Str2=" ".join(List1)
```

```
print(Str2) Python is very simple and easy language
type(Str2) <class 'str'>
```

The main difference between String and List is mutation.

1. String is immutable whereas List is mutable.

2. Mutable objects can be altered whereas immutable objects can't be altered.

List Packing:

A list can be created by using a group of variables, it is called list packing

```
a=10
b=20
c=True
d='Py'
list1=[a,b,c,d]
print(list1) [10, 20, True, 'Py']
type(list1) <class 'list'>
```

List Unpacking:

1. List unpacking allows extracting list elements automatically.

2. List unpacking is the list of variables on the left has the same number of elements as the length of the list.

```
lst=[10,20,'Python',True]
print(lst) [10, 20, 'Python', True]
type(lst) <class 'list'>
a,b,c,d=lst #list unpacking
print(a) 10
type(a) <class 'int'>
print(b) 20
type(b) <class 'int'>
print(c) Python
type(c) <class 'str'>
print(d) True
type(d) <class 'bool'>
```

TUPLE DATA TYPE

Tuple is used to represent a set of homogeneous or heterogeneous elements into a single entity.

- Tuple objects are immutable that means once if we create a tuple object later we cannot modify those tuple elements.
- All elements are separated by commas (,) and enclosed by parentheses. Parentheses are optional. ()
- Tuple allows duplicate elements.
- Every element in the tuple has its own index number
- Tuple supports both forward indexing and also backward indexing, forward indexing starts from 0 and backward indexing starts from -1.
- If we take only one element in the tuple then we should use comma (,) after that single element.

```
t = (10,) --->> tuple type
t1 = (10) --->> int type
```

Tuples can be used as keys to the dictionary.

We can create a tuple in different ways, like with tuple(), with () or without () also.

The main difference between lists and tuples is- Lists are enclosed in square brackets like [] and their elements and size can be changed, while tuples are enclosed in parentheses like () and their elements and size cannot be updated.

Creating a tuple with tuple() :

```
>>> tup = tuple([10,20,30,True,'Python'])
>>> print(tup) (10, 20, 30, True, 'Python')
>>> type(tup) <class 'tuple'>
>>> id(tup) 52059760
```

Creating an empty tuple:

Example:

```
>>> tup = () #creating empty tuple
>>> print(tup) ()
>>> type(tup) <class 'tuple'>
>>> id(tup) 23134256
```

Creating a tuple with ()

Example:

```
>>> tup2 = (10,20,30,40,50) #creating homogeneous tuple
>>> print(tup2) (10, 20, 30, 40, 50)
>>> type(tup2) <class 'tuple'>
>>> id(tup2) 63484864
```

Creating a tuple without ()

Example:

```
>>> tup = 10,20,True,'Py' #creating tuple without parenthesis
>>> print(tup) (10, 20, True, 'Py')
>>> type(tup) <class 'tuple'>
>>> id(tup) 67086688
```

Creating a tuple with heterogeneous elements

Example:

```
>>> tup1 = (10,20,30,True,"Python",10.5,3+5j) #creating heterogeneous tuple
>>> print(tup1)
(10, 20, 30, True, 'Python', 10.5, (3+5j))
```

Creating a tuple with homogeneous elements

Example:

```
>>> t = (10,20,30,40) # creating homogeneous tuple
>>> print(t) # (10,20,30,40)
```

NOTE: tuple with Single value

Creating a tuple with a single element is tricky, if we take only one element then the type of that tuple will be based on specified element type.

```
>>> t2 = (1)
>>> t2 1
>>> type(t2) <type 'int'>
>>> t2 = (True)
>>> print(t2) True
>>> type(t2) <type 'bool'>
>>> t2 = ('a')
>>> print(t2) 'a'
>>> type(t2) <type 'str'>
```

Solution:

So to solve the above problem we should use comma (,) after the element in the tuple if tuple contains single element.

For example:

```
>>> t2=(1,)
>>> print(t2) (1,)
>>> type(t2) <type 'tuple'>
>>> t2=(False,)
>>> print(t2) (False,)
>>> type(t2) <type 'tuple'>
```

>>> t2 = ('a')

>>> print(t2) ('a',)

>>> type(t2) <type 'tuple'>

Tuple Indexing:

Tuple indexing is nothing but fetching a specific element from the existing tuple by using its index value.

Tuple Slicing:

Tuple slicing is nothing but fetching a sequence of elements from the existing tuple by using their index values.

Example:

>>>tup = (10,20,30,True,"Python",10.5,3+5j,10)

>>> print(tup) # (10, 20, 30, True, 'Python', 10.5, (3+5j), 10)

>>> type(tup) # <class 'tuple'>

Tuple concatenation:

We can concatenate two or more tuples in python.

Example:

>>> tup1=(1,2,3,'a',True) #creating first tuple tup1

>>> print(tup1) (1, 2, 3, 'a', True)

>>> type(tup) <class 'tuple'>

>>> tup2=(10,20,False,'b') #creating second tuple tup2

>> print(tup2) (10, 20, False, 'b')

>>> type(tup2) <class 'tuple'>

>>> tup3 = tup1+tup2 #concatenating tup1 and tup2 as tup3

>>> print(tup3) (1, 2, 3, 'a', True, 10, 20, False, 'b')

>>> type(tup3) <class 'tuple'>

Tuple multiplication or repetition:

We can multiply or repeat a tuple n number of times.

>>> tup1=(1,2,3,'a',True)

>>> print(tup1) (1, 2, 3, 'a', True)

>>> type(tup1) <class 'tuple'>

>>> tup1*3

(1, 2, 3, 'a', True, 1, 2, 3, 'a', True, 1, 2, 3, 'a', True)

Tuple Data type Methods:

1. len():

This function returns no.of elements in the tuple.

>>> tup = (1,2,3,4,'a',5.5)

>>> len(tup) 6

2. count():

This function counts the number of occurences of a specific elements.

This function takes exactly one argument like element.

Example:

>>> tup = (1,10,20,True,0)

>>> tup.count(1) 2

>>> tup.count(0) 1

3. index(object, index_value,end_index):

This function is used to find the index value of specific given element. This function returns by default first occurrence of given element index value. It is also accepts the second parameter as index value, it is used for from where you want search the given index. By default index value starts from zero.

Example:

>>> tup=(1,10,20,True,0)

>>> tup.index(0) 4

>>> tup.index(10) 1

>>> tup.index(20) 2

4. max():

This function returns maximum value from the tuple elements.

Example:

>>> tup=(1,3,2,55,3,5,23)

>>> max(tup) 55

5. min():

This function returns minimum value from the tuple elements.

Example:

>>> tup=(1,3,2,55,3,5,23)

>>> min(tup) 1

6. sum():

this function returns sum of all the elements.

Example:

>>> tup=[1,9,5,11,2]

>>> sum(tup) 28

7. sorted(object):

Sorted () is going to take the elements from given object and arranging all the elements by default in a ascending order. After arranging all the elements in ascending order then result store in a new variable. Sorted () method is not doing any changes in a original object and the result store in a new object. Sorted () method returns result in a list format by default. if

you want to get in tuple format then use tuple() method

Example:

>>> tup = (1,3,2,55,3,5,23)

>>> sorted(tup) [1, 2, 3, 3, 5, 23, 55]

Note: by default this function sorts the data in ascending order. We can also get in descending order by setting True for reverse.

Example:

>>> tup=(1,3,2,55,3,5,23)

>>> sorted(tup,reverse=True) [55, 23, 5, 3, 3, 2, 1]

Or

>>> t1 = tuple([1,2,3,7,4])

>>> t1 (1, 2, 3, 7, 4)

>>> t2 = reversed(t1)

>>> tuple(t2) (4, 7, 3, 2, 1)

8. reversed():

Reversed () is going to take the elements from given object and arranging all the elements by default in a reversing order. After arranging all the elements in reversing order then result store in a new variable. reversed () method is not doing any changes in a original object and the result store in a new object. reversed() method returns result in a <reversed object at 0x03EFFC30> format by default. Internally elements are reversed.

if you want to get in tuple format then use tuple() method

>>> t = (10, 40, 60, 20)

>>> t2 = reversed(t)

>>> t2

<reversed object at 0x03EFFC30>

>>> tuple(t2) # (20, 60, 40, 10)

Note:

tuple object is not supporting both sort() and reverse() and copy() and clear() also.

>>> t = (10, 40, 60, 20)

>>> t2 = sort(t)

NameError: name 'sort' is not defined

>>> t3 = reverse(t)

NameError: name 'reverse' is not defined

>>> t = (10, 40, 60, 20)

>>> id(t) # 65890704

>>> t2 = t

>>> print(t2) # (10, 40, 60, 20)

>>> id(t8) # 65890704

>>> t2 = t.copy()

AttributeError: 'tuple' object has no attribute 'copy'

DEL Command :

We cannot delete the elements of existing tuple but we can delete the entire tuple object by using del command.

Example:

>>> tup = (10,20,"Python",1.3)

>>> print(tup) # (10, 20, 'Python', 1.3)

>>> type(tup) # <class 'tuple'>

>>> del tup # deleting tuple by using del command.

>>> print(tup) # after deleting

NameError: name 'tup' is not defined

We can replace the elements of list but not tuple, like

>>> lst=[10,20,30,'Py',True]

>>> lst[4]=False # it is possible in list

>>> print(lst) [10, 20, 30, 'Py', False]

>>> tup = (10,20,30,'Py',True)

>>> tup[4]=False # it is not possible in tuple

TypeError: 'tuple' object does not support item assignment

Tuple packing:

We can create a tuple by using existing variables, so its called tuple packing.

>>> a=10

>>> b=20

>>> c='Python'

>>> d=2+5j

>>> tup=(a,b,c,d)

>>> print(tup) (10, 20, 'Python', (2+5j))

>>> type(tup) <class 'tuple'>

>>> id(tup) 62673808

Tuple Unpacking

Tuple unpacking allows to extract tuple elements automatically.

Tuple unpacking is the list of variables on the left has the same number of elements as the length of the tuple

>>> tup=(1,2,3,4)

>>> a,b,c,d=tup # tuple unpacking

>>> print(a) 1

>>> print(b) 2

>>> print(c) 3

>>> print(d) 4

Nested tuple:

Python supports nested tuple, means a tuple in another tuple.Tuple allows list as its element.

Example:

>>> t1=(1,'a',True)

>>> print(t1) (1, 'a', True)

>>> type(t1) <class 'tuple'>

>>> t2=(10,'b',False)

>>> print(t2) (10, 'b', False)

>>> type(t2) <class 'tuple'>

>>> t3=(t1,100,'Python',t2) # creating a tuple with existing tuples t1 and t2

>>> print(t3) ((1, 'a', True), 100, 'Python', (10, 'b', False))

>>> type(t3) <class 'tuple'>

>>> print(t3[0]) (1, 'a', True)

>>> print(t3[1]) 100

>>> print(t3[2]) Python

>>> print(t3[3]) (10, 'b', False)

>>> print(t3[3][0]) 10

>>> print(t3[3][1]) b

>>> print(t3[3][2]) False

>>> print(t3[0][0]) 1

>>> print(t3[0][1]) a

>>> print(t3[0][2]) True

>>> t3[0:2] ((1, 'a', True), 100)

>>> t3[2:4] ('Python', (10, 'b', False))

>>> t3[-2:4] ('Python', (10, 'b', False))

Note:

We can't modify any element of the above tuples because tuples are immutable.

If the tuple contains a list as a element then we can modify the elements of the list as it a mutable object.

Example:

>>> tup = (1,2,[10,12,'a'],(100,200,300),3,'Srinivas')

>>> print(tup) # (1, 2, [10, 12, 'a'], (100, 200, 300), 3, 'Srinivas')

>>> tup[0] 1

>>> tup[1] 2

```
>>> tup[2] [10, 12, 'a']
>>> tup[3] (100, 200, 300)
>>> tup[4] 3
>>> tup[5] 'Srinivas'
>>> tup[0]=50
# trying to replace element 1 with 50, interpreter throws error.
TypeError: 'tuple' object does not support item assignment
>>> tup[2][0]=50
# trying to replace element of list 10 with 50, interpreter accepts.
>>> print(tup) (1, 2, [50, 12, 'a'], (100, 200, 300), 3, 'Srinivas')
```

Converting tuple to list :

```
>>> tup=(1,2,4,9,8) #creating a tuple
>>> print(tup) (1, 2, 4, 9, 8)
>>> type(tup) <class 'tuple'>
>>> lst = list(tup) # converting tuple to list by using list()
>>> print(lst) [1, 2, 4, 9, 8]
>>> type(lst) <class 'list'>
```

Converting list to tuple:

```
>>> lst=[10,20,30,40,'a'] #creating a list
>>> print(lst) [10, 20, 30, 40, 'a']
>>> type(lst) <class 'list'>
>>> tup=tuple(lst) #converting list to tuple by using tuple()
>>> print(tup) (10, 20, 30, 40, 'a')
>>> type(tup) <class 'tuple'>
```

Converting tuple to string:

```
>>> tup=('a','b','c') #creating tuple
>>> print(tup) ('a', 'b', 'c')
>>> type(tup) <class 'tuple'>
>>> str1=''.join(tup) #converting tuple to string by using join method
>>> print(str1) abc
>>> type(str1) <class 'str'>
```

Converting string to tuple:

```
str1="Python by krishna"
print(str1)
print(type(str1))
tup=tuple(str1)
print(tup)
```

print(type(tup))

Advantages of Tuple over List:

- ✓ Generally we use tuple for heterogeneous elements and list for homogeneous elements.
- ✓ Iterating through tuple is faster than list because tuples are immutable, So there might be a slight performance boost.
- ✓ Tuples can be used as key for a dictionary. With list, this is not possible because list is a mutable object.
- ✓ If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

SNo.	Key	List	Tuple
1	Type	List is mutable.	Tuple is immutable.
2	Iteration	List iteration is slower and is time consuming.	Tuple iteration is faster.
3	Appropriate for	List is useful for insertion and deletion operations.	Tuple is useful for readonly operations like accessing elements.
4	Memory Consumption	List consumes more memory.	Tuples consumes less memory.
5	Methods	List provides many in-built methods.	Tuples have less in-built methods.
6	Error prone	List operations are more error prone.	Tuples operations are safe.

PROGRAMS

Write a Program to print numbers from 1 to 10, skipping 5 and breaking at 8

```
for i in range(1, 11):
   if i == 5:
      continue  # skip 5
   if i == 8:
      break  # break at 8
   print(i)
```

Write a program to accept some list elements and find sum of elements?

```
mylist = input("Enter a list of numbers separated by space: ")
mylist = list(map(int, mylist.split()))
sum = 0
for num in mylist:
  sum += num
print("The sum of the numbers is:", sum)
```

Solve a Python program that prints all the numbers from 0 to 6 except 3 and 6

```
for num in range(7):
    if num !=3 nd num != 6:
      print(num)
```

Write a python program to find sum of the elements in a list?(input is[8,2,3,0,7])

```
mylist = [8,2,3,0,7]
sum = 0
for i in mylist:
   sum = sum+i
print("The sum is:",sum)
```

Output

```
The sum is: 20
```

Write a python program to accept N numbers and store them in list, then print the list without odd numbers in it

```
n = int(input("Enter the value of n: "))
mylist = []
for i in range(n):
   num = int(input("enter a number:"))
   mylist.append(num)
print("Even numbers are")
print("[", end="")
for num in mylist:
   if num % 2 == 0:
      print(num, end=", ")
print("]")
```

Output

```
Enter the value of n: 5
enter a number:10
enter a number:15
enter a number:20
enter a number:25
enter a number:33
Even numbers are
[10, 20, ]
```

Write a python program to accept a string and converted into a list?

```
s = input("Enter the string: ")
print(s.split())
```

Output

```
Enter the string: python is very easy learning program
['python', 'is', 'very', 'easy', 'learning', 'program']
```

Example: For a=['hello', 'how', [1,2,3], [[10,20,30]]] what is the output of following statement (i) print(a[: :]) (ii) print(a[-3][0]) (iii) print(a[2][: -1]) (iv) print(a[0][: : -1])

(i) print(a[::]) : This statement prints the entire list a. The slicing a[::] means start from the beginning (:), end at the end (:), and take steps of size 1 (:). So, it essentially returns a copy of the entire list a.

(ii) print(a[-3][0]) : a[-3] refers to the third element from the end of the list a, which is [1,2,3], and [0] indexes the first element of that sublist, which is 1. So, the output will be 1.

(iii) print(a[2][: -1]) - This will print a slice of the third element of a, which is [1,2,3]. a[2] refers to [1,2,3], and [:-1] means to take all elements except the last one. So, it will print [1,2].

(iv) print(a[0][::-1]) - This statement will reverse the first element of a, which is 'hello'. a[0] refers to 'hello', and [::-1] means to reverse the string. So, it will print 'olleh'.

So, the outputs will be:

(i) ['hello', 'how', [1, 2, 3], [[10, 20, 30]]]

(ii) 1

(iii) [1, 2]

(iv) 'olleh'

Write a python program to print without duplicate numbers in a list?

```
mylist = [10, 20, 30, 40, 20, 30, 50, 60, 10]
```

```
print("my original list is:",mylist)
newlist = []
for num in mylist:
    if num not in newlist:
        newlist.append(num)
print("List without duplicates:", newlist)
```

Output:

```
my original list is: [10, 20, 30, 40, 20, 30, 50, 60, 10]
List without duplicates: [10, 20, 30, 40, 50, 60]
```

Develop a Python program weather a given number is prime or not.

```
n = int(input("Enter a number: "))
if n > 1:
    for i in range(2, n):
        if (n % i) == 0:
            print(n, "is not a prime number")
            break
    else:
        print(n, "is a prime number")
else:
    print(n, "is not a prime number")
```

Output:

```
Enter a number: 5
5 is a prime number
```

Write a python program to find the given number is palindrome or not?

```
num = input("Enter a number: ")
rev = num[::-1]
if num == rev:
    print(num, "is a palindrome")
else:
    print(num, "is not a palindrome")
```

Output:

```
Enter a number: 191
191 is a palindrome
```

Write a python program to print numbers from 1 to 10, skipping multiples of 2

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

```
    if i % 2 == 0:
        continue
    print(i,end=" ")
```

Output:

```
1 3 5 7 9
```

Write a python program to find squares of numbers from 1 to 10 using list comprehension

```
squares = [x ** 2 for x in range(1, 11)]
print("Squares of numbers from 1 to 10:", squares)
```

Output:

```
Squares of numbers from 1 to 10: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Write a python program to print Fibonacci series up to n terms?

```
n = int(input("Enter the number of terms: "))
a, b = 0, 1
count = 0
while count < n:
    print(a)
    c = a + b
    a = b
    b = c
    count += 1
```

Sample output:

```
Enter the number of terms: 5
0
1
1
2
3
```

Write a python Program to print numbers from 1 to 10, skipping 5

```
for i in range(1, 11):
    if i == 5:
        continue
    print(i,end=" ")
else:
    print("\nLoop completed successfully.")
```

Sample output:

```
1 2 3 4 6 7 8 9 10
Loop completed successfully.
```

Write a python Program to print numbers from 10 to 1, skipping 5 and breaking at 4?

```
for num in range(10, 0, -1):
    if num == 5:
        continue
    print(num)
    if num == 4:
        break
```

Sample output:

```
10 9 8 7 6 4
```

Write a python program to convert tuple data type into list data type?

```
tup=(1,2,4,9,8)
print(tup)
print(type(tup))
lst = list(tup)
print(lst)
print(type(lst))
```

Sample output:

```
(1, 2, 4, 9, 8)
<class 'tuple'>
[1, 2, 4, 9, 8]
<class 'list'>
```

Write a python program to convert list data type into tuple format?

```
lst=[10,20,30,40,'a']
print(lst)
print(type(lst))
tup=tuple(lst)
print(tup)
print(type(tup))
```

Sample output:

```
[10, 20, 30, 40, 'a']
<class 'list'>
(10, 20, 30, 40, 'a')
<class 'tuple'>
```

Write a python program to covert tuple format into string format?

```
tup=('a','b','c')
print(tup)
print(type(tup))
str1=''.join(tup)
print(str1)
print(type(str1))
```

Sample output:

```
('a', 'b', 'c')
<class 'tuple'>
abc
<class 'str'>
```

Write a python program to convert string format into tuple format?

```
str1="Python by krishna"
print(str1)
print(type(str1))
tup=tuple(str1)
print(tup)
```

Sample output:

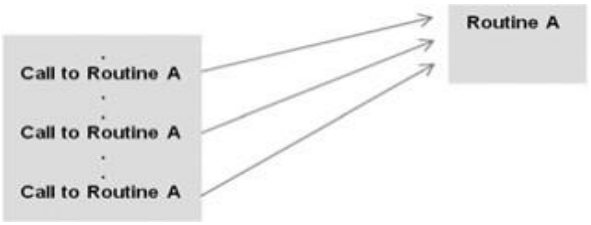
```
Python by krishna
<class 'str'>
('P', 'y', 't', 'h', 'o', 'n', ' ', 'b', 'y', ' ', 'k', 'r', 'i', 's', 'h', 'n', 'a')
```

UNIT-III

FUNCTIONS: Program Routines, More on Functions-Calling Value-Returning Functions, Calling Non-Value-Returning Functions, Parameter Passing, Keyword Arguments in Python, Default Arguments in Python, Variable Scope. Lambda function, Map function, Filters function, **RECURSION**: Recursive Functions, Recursive Problem Solving, Iteration vs. Recursion. **OBJECTS:** Software Objects, Turtle Graphics, Turtle programs

FUNCTIONS: Program Routines, More on Functions-Calling Value-Returning Functions

A routine is a named group of instructions performing some task. A routine can be invoked (called) as many times as needed in a given program, as shown below:



When a routine terminates, execution automatically returns to the point from which it was called. Such routines may be predefined in the programming language, or designed and implemented by the programmer. A function is Python's version of a program routine. Some functions are designed to return a value, while others are designed for other purposes.

Python functions are simple to define and essential to intermediate-level programming. The exact criteria hold to function names as they do to variable names. The goal is to group up certain often performed actions and define a function. Rather than rewriting the same code block over and over for varied input variables, we may call the function and repurpose the code included within it with different variables.

A function can be related to the process of making tea. If you were to make tea for the first time, your parents would have defined the steps to make it. The next time they want to drink tea, they'll call out to you directly by saying 'Make Tea'.

Similarly, in the case of functions, you first define the sequence of steps that you would like

to carry out to achieve a certain task. Later, you can call out the Function by its name, and these steps will be performed.

Definition:

- A function in Python is a named block of organized and connected statements that performs a specific task.
- In other words, a function is a block of code that performs a specific and well-defined task. It organizes the code into a logical way to perform a certain task.
- Function is syntax of structure in which we represents the reusable business logic.
- Every function is called by a certain name and the block of code inside the function gets executed when we call the function.
- If you have a block of code that gets invoked more than once, put it into a function. This is because function allows us to use a block of code repeatedly in different sections of a program.
- It makes the program more efficient by minimizing the repetition. It makes the code reusable.
- Functions help to break down long and complex programs into smaller and manageable segments and enhance program readability and comprehensibility.
- They provide better modularity for our application program and a high degree of code reusability. Sometimes, we call the function with another name called method. But, there is a slight difference between a function and method.
- Functions are defined outside the class, whereas methods are defined inside the class.

Syntax to define User defined Function in Python The user creates or defines user-

defined functions. The general syntax to declare a user defined function in Python is as follows

```
def function_name(parameters): # Function header.
    """docstring"""
    ......
    ......

    body of the function
    ......
    ......
    return [expression or values]
```

In the above syntax, the first line of function definition is header, and the rest is the body of function. The function definition begins with **def** keyword followed by the function name, parentheses (()), and then colon (:). The parameter list placed within parentheses is optional. Let's define a simple function that prints the message "Hello Python".

Advantages of Functions in Python

- By including functions, we can prevent repeating the same code block repeatedly in a program.
- Python functions, once defined, can be called many times and from anywhere in a program.
- If our Python program is large, it can be separated into numerous functions which is simple to track.
- The key accomplishment of Python functions is we can return as many outputs as we want with different arguments.

```
def function_name (parameters):
    """docstring """
   statement1
   statement 2

   return [expr]
```

The following elements make up define a function

- The beginning of a function header is indicated by a keyword called def.
- **Name_of_function** 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 same criteria apply to naming functions as to naming variables in Python.
- We pass arguments to the defined function using parameters. They are optional, though. The function header is terminated by a colon (:).
- We can use a documentation string called docstring in the short form to explain the purpose of the function.
- 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.

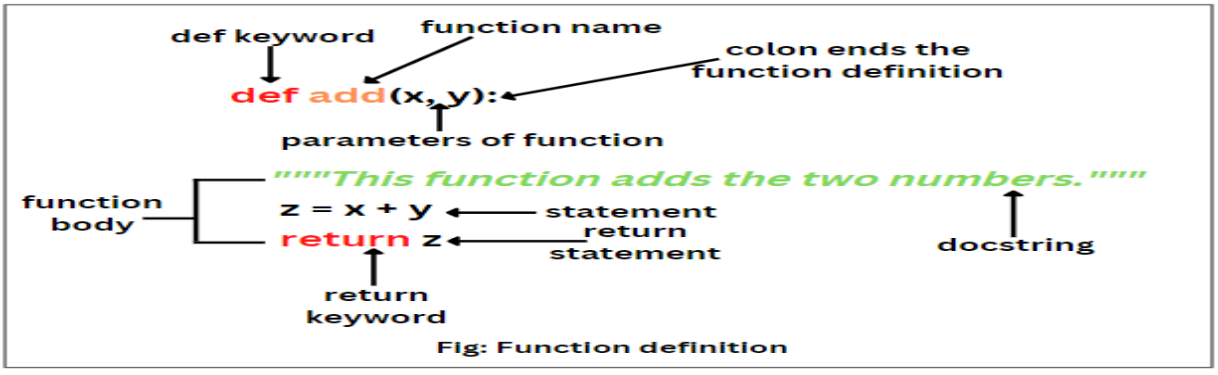


Fig: Function definition

1. **Keyword def:** Function definition starts with the keyword 'def' that defines the function. It tells the beginning of the function header.
2. **Function name:** The function_name represents the name of a function. Every function

must be a unique user-defined name or an identifier and given to it in the function header statement.

3**. Parameters**: The formal parameters (arguments) placed inside the parentheses are optional. A parameter is a piece of data or information that we use inside the function to perform a specific task. Generally, parameters are a list of local variables that will be assigned with values when the function gets called. They are used to passing values to functions. They can be zero, or more parameters, separated by commas inside the parentheses. In the above example, there are two parameters in the function definition.If any function contains parameters, At the time of calling their function, we have to pass values to those parameters.

If not pass parameters values then we will get error

4**. Colon (:):** A colon indicates the end of function headers.

5. **Docstring:** A docstring is a documentation string that is an optional component. We commonly use it to describe what the function does. We write it on the line next to the function header. A docstring must enclose in triple quotes and can write up to in several lines. We can access it with: function_name.__doc__.

6. **Statement(s):** The body of function consists of one or more valid statements. Each statement must be intended with the same indentation to form a block. In general, 4 spaces of indentation are standard from the header line in Python. When the function gets called, the statements inside the function's body execute. Once the function is invoked, the formal parameters inside the parentheses become arguments.

7. **Return statement**: A return statement ends the function call and returns a value from a function back to the function calling code. It is optional. If we do not define return statement inside the function's body, the function returns the object 'None'. After the return statement if we are creating any other function statements then those all statements are not executing. Because return keyword statement only last statement in every function body.

Rules for defining User defined Function

1. Functions must begin either with a letter or an underscore.
2. They should be lowercase.
3. They can comprise numbers but shouldn't begin with one.
4. Functions can be of any length.
5. The name of function should not be a keyword in Python.
6. We can use an underscore to separate more than one word. For example, function_name(), person_name(), etc. It improves the readability and maintains conventions.
7. There should not be any space between the two words.

Syntax:-

```
def function_name(parameters_lists):
statements1
statements2 # block / suite / scope
-----------
-----------
function_name(actual_arguments_list)
```

Write a function to add the given values and store result in a variable

```
def add(a,b):
return a+b
x = add(10,20)
print(x)
```

Output: 30

Write a function to print a message?

```
def m1():
print('hi')
m1()
```

Output : hi

Types of Functions

Based on "argument" and "return" type functions are classified into four types.

1. Functions with no-arguments and no-return type.
2. Functions with arguments and no-return type.
3. Functions with no-arguments and return type.
4. Functions with arguments and with return type.

The general syntax of all functions,

```
def functionname(p1,p2,p3..):
statement1
statement2
statement3
return value1, value2,..
```

1. **Functions with No-Arguments and No-return type.**

Syntax:

```
def functionname():
statement1
statement1
statement3
```

For example:

```
def sum():
a = 10
b = 20
c = a + b
print("Sum is : ",c)
sum()
```

2. Functions with Arguments and No-return type.

```
def functionName(p1,p2,...):
statement1
statement1
statement3
```

For example:

```
def sum(a,b):
c = a + b
print("Sum is : ",c)
sum(10,20)
```

3. **Functions with No-Arguments and with return type.**

```
def functionname():
statement1
statement1
statement3
return value1, value2
```

For example:

```
def sum():
a = 10
b = 20
c = a + b
return c
s = sum()
print("Sum is :", s)
```

4. **Functions with arguments and with return type.**

```
def functionname(p1,p2,..):
statement1
```

```
statement1
statement3
return value1, value2,...
```

For example:

```
def sum(a,b):
c = a + b
return c
s = sum(10,20)
print("Sum is :", s)
```

Parameter Passing, Keyword Arguments in Python, Default Arguments in Python

```
def f1(a,b):
------
------
f1(10,20)
```

a, b are **formal arguments** whereas **10,20 are actual arguments**.

There are 4 types are actual arguments are allowed in Python.

1) Positional Arguments

2) Keyword Arguments

3) Default Arguments

4) Variable Length Arguments

1) Positional Arguments:

These are the arguments passed to function in correct positional order.

```
def sub(a, b):
   print(a-b)
sub(100, 200)
sub(200, 100)
```

2) Keyword Arguments:

We can pass argument values by keyword i.e by parameter name.

```
def wish(name,msg):
   print("Hello",name,msg)
wish(name="Krishna",msg="Good Morning")
wish(msg="Good Morning",name="Krishna")
```

Output:

```
Hello Krishna Good Morning
Hello Krishna Good Morning
```

Here the order of arguments is not important but number of arguments must be matched.

3) Default Arguments:

Sometimes we can provide default values for our positional arguments.

```
def wish(name="Guest"):
    print("Hello",name,"Good Morning")
wish("Krishna")
wish()
```

Output

```
Hello Krishna Good Morning
Hello Guest Good Morning
```

If we are not passing any name then only default value will be considered.

4) Variable Length Arguments:

Sometimes we can pass variable number of arguments to our function, such type of arguments are called variable length arguments. We can declare a variable length argument with * symbol as follows

def f1(*n):

We can call this function by passing any number of arguments including zero number. Internally all these values represented in the form of tuple.

```
def sum(*n):
    total = 0
    for n1 in n:
        total += n1
    print("The Sum =", total)
sum()
sum(10)
sum(10, 20)
sum(10, 20, 30, 40)
```

Output:

```
The Sum = 0
The Sum = 10
The Sum = 30
```

```
The Sum = 100
```

We can call this function by passing any number of keyword arguments. Internally these keyword arguments will be stored inside a dictionary.

```
def display(**kwargs):
   for k,v in kwargs.items():
      print(k,"=",v)
display(n1=10,n2=20,n3=30)
display(rno=100,name="Krishna",marks=70,subject="Python")
```

Output:

```
n1 = 10
n2 = 20
n3 = 30
rno = 100
name = Krishna
marks = 70
subject = Python
```

Variable Scope

All variables in a program may not be access at all locations in that program. They are accessible depends on where you have declared the variables. Variables are divided in to 2-types. They are,

1. Global variable
2 . Local Variable.

1. Global variables:

The variables which are declared outside of all the functions are known as global variables. Global variables of one module can be used throughout module.

```
a=100
def f1():
print(a)
f1()
```

2. Local variables:

The variables declared with in a function are known as local variables. Local variables of one function cannot be accessed outside of that function.

```
a=100 # local
print(a)
```

```
def f2() :
print(a)
a=200 # global
f1()
f2()
```

Lambda function, Map function, Filters function

1. Lambda function is a small anonymous function, i.e. functions without a name.
2. These functions are throw-away functions, i.e. they are just needed where they have been created.
3. We use a lambda function when we require a nameless function for short time.

The general syntax of a lambda function is quite simple:

lambda argument list : expression

The argument list consists of a comma separated list of arguments and the expression is an arithmetic expression using these arguments.

Q) Write a function to add two numbers ?

By using def

```
def add_nums(x , y):
return x + y
print(add_nums(1,2)) # 3
```

by using Lambda:

```
a = lambda x,y:x+y
print(a(1,2))
```

Q. **Write a Python program to print the only Even numbers from given list elements?**

```
lst =[1,5,4,6,8,11]
even_list =list(filter( lambda x : ( x%2 == 0, lst)))
print(even_list)
```

Q 2) Write a python function to find maximum value of two given values ?

```
def max(x,y) :
if x > y :
return x
else :
return y
max(1,2) # 2
```

```
max(10,2) # 10
```

Q 3)Write a python function to find max value of three given values ?

```
def maxofthree(x,y,z):
if x > y and x > z :
return x
elif y > z :
return y
else :
return z
maxofthree(100,3,20)
maxofthree(10,3,20)
maxofthree(10,30,20)
```

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 iterrable object. The filter function returns a sequence from those elements of iterable for which function returns True. The first argument can be None if the function is not available and returns only elements that are True.

```
lst = [1,2,3,4,5,6,7,8,9,10]
x = list(filter( None, lst))
print(x)
```

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

Q. **Write a program to returns values higher than 5 using filter function?**

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

Output: [6]

Q. Write a program to find numbers which are divisible by 3 from given sequence?

```
def multiplicationof3(val):
```

```
if (val % 3) == 0:
return val
result = filter(multiplicationof3,(1,3,5,6,8,9,12,14))
result = list(result)
print(result)
```

Output: [3, 6, 9, 12]

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 iterrable (list, tuple etc.).

Example:

```
def calculateAddition(n):
return n+n
numbers = (1, 2, 3, 4)
result = map(calculateAddition, numbers)
print(result)
numbersAddition = set(result)
print(numbersAddition)
```

Output:

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

Q. **Add 5 value to every list item and returns result?**

```
lst = [1,4,7,9,6,11,16,15]
result = list(map( lambda x : x + 5, lst))
print(result)
```

Output: [6, 9, 12, 14, 11, 16, 21, 20]

Q**. Write a function to find given value is a palindrome or not?**

```
def is_palondrom(n):
if n == n[::-1]:
print(n,'is a palindrom value')
else:
print(n,'is not a palindrom value')
n = input('Enter any number :') # 5
is_palondrom(n)
```

```
Enter any number :121
121 is a palindrom value
Enter any number :madam
madam is a palindrom value
```

Q. **How to find the Fibonacci sequence numbers and its sum value?**

```
Number = int(input("Please Enter the Fibonacci Number Range = "))
First = 0
Second = 1
Sum = 0
for Num in range(0, Number):
print(First, end = ' ')
Sum = Sum + First
Next = First + Second
First = Second
Second = Next
print("\nThe Sum of Fibonacci Series Numbers = %d" %Sum)
```

Output:

```
Please Enter the Fibonacci Number Range = 8
0 1 1 2 3 5 8 13
The Sum of Fibonacci Series Numbers = 33
```

RECURSION: Recursive Functions, Recursive Problem Solving

Term Recursion can be defined as the process of defining something in terms of itself. In simple words, it is a process in which a function calls itself directly or indirectly.

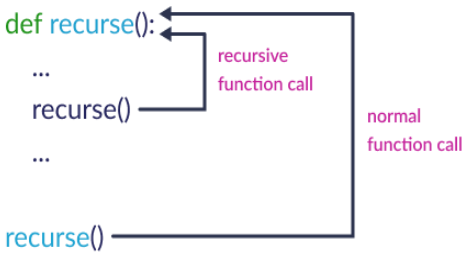
Advantages of using recursion

- A complicated function can be split down into smaller sub-problems utilizing recursion.
- Sequence creation is simpler through recursion than utilizing any nested iteration.
- Recursive functions render the code look simple and effective.

Disadvantages of using recursion

- A lot of memory and time is taken through recursive calls which make it expensive for use.
- Recursive functions are challenging to debug.

- The reasoning behind recursion can sometimes be tough to think through.



The factorial of a number is calculated by multiply number itself and n-1 until we get one. Suppose we want to want factorial of 5,

```
0! = 1
1! = 1x0! = 1x1 = 1
2! = 2x1! = 2
3! = 3x2! = 3x2 = 6
4! = 4x3! = 4x6 = 24
5! = 5x4! = 5x24 = 120
```

It is calculating n*(n-1) repeatedly. Hence we can use recursion. Here the factorial of 0 is 1.

```
def factorial(n):
  if n==0:
    return 1
    fact(n) = n * f(n-1) n<1
if n==0:
   return 1
else:
return n*factorial(n-1)
num = int(input("Enter a number: "))
print("The factorial of a {0} is: ".format(num), factorial(num))
```

Sample output:

```
Enter a number: 5
The factorial of a 5 is: 120
```

Fibonacci sequence

It is a most famous mathematical problem and sometime also asked in interview at entry level jobs. In this sequence, each number in the sequence is the sum of the two numbers that precede it. The series will be 0, 1, 1, 2, 3, 5, 8,....so on. The first element is 0 and second is 1; hence the third element is sum of first and third element. Suppose we want 6th element of Fibonacci series, the preceding two elements will be 5 and 4 -

```
fib(6) = fib(5) + fib(4) => 5 + 3 = 8
fib(5) = fib(4) + fib(3) => 2 + 3 = 5
fib(4) = fib(3) + fib(2) => 2 + 1 = 3
fib(3) = fib(2) + fib(1) => 1 + 1 = 2
```

```
def fib(n) :
  if n==0:
     return 0
 elif n ==1 :
    return 1
else :
return fib(n-1) +fib(n-2)
num = int(input("Enter a number: "))
print("The element of fibonacci series is: ".format(num), fib(7))
```

sample output:

```
Enter a number: 7
The element of fibonacci series is: 13
```

Difference between Recursion and Iteration

The following table highlights all the important differences between recursion and iteration –

On the basis of	Recursion	Iteration
Basic	Recursion is the process of calling a function itself within its own code.	In iteration, there is a repeated execution of the set of instructions. In Iteration, loops are used to execute the set of instructions repetitively until the condition is false.
Syntax	There is a termination condition is specified.	The format of iteration includes initialization, condition, and increment/decrement of a variable.
Termination	The termination condition is defined within the recursive function.	The termination condition is defined in the definition of the loop.

Code size	The code size in recursion is smaller than the code size in iteration.	The code size in iteration is larger than the code size in recursion.
Applied	It is always applied to functions.	It is applied to loops.
Speed	It is slower than iteration.	It is faster than recursion.
Usage	Recursion is generally used where there is no issue of time complexity, and code size requires being small.	It is used when we have to balance the time complexity against a large code size.
Stack	It has to update and maintain the stack.	There is no utilization of stack.
Memory	It uses more memory as compared to iteration.	It uses less memory as compared to recursion.

OBJECTS: Software Objects, Turtle Graphics, Turtle programs

Turtle is a Python library which used to create graphics, pictures, and games. It was developed by Wally Feurzeig, Seymour Parpet and Cynthina Slolomon in 1967. It was a part of the original Logo programming language.The Logo programming language was popular among the kids because it enables us to draw attractive graphs to the screen in the simple way. It is like a little object on the screen, which can move according to the desired position. Similarly, turtle library comes with the interactive feature that gives the flexibility to work with Python. Turtle is a pre-installed library in Python that is similar to the virtual canvas that we can draw pictures and attractive shapes.The turtle Library is primarily designed to introduce children to the world of programming. With the help of Turtle's library, new programmers can get an idea of how we can do programming with python in a fun and interactive way.

The Python turtle library consists of all important methods and functions that we will need to create our designs and images. Import the turtle library using the following command.

Syntax

```
from turtle import *
# or
import turtle
```

Turtle" is a Python feature like a drawing board, which lets us command a turtle to draw all over it! We can use functions like turtle.forward(...) and turtle.right(...) which can move the turtle around. Commonly used turtle methods are

Method	Parameter	Description
Turtle()	None	Creates and returns a new turtle object
forward()	amount	Moves the turtle forward by the specified amount
backward()	amount	Moves the turtle backward by the specified amount
right()	angle	Turns the turtle clockwise
left()	angle	Turns the turtle counterclockwise
penup()	None	Picks up the turtle's Pen
pendown()	None	Puts down the turtle's Pen
up()	None	Picks up the turtle's Pen
down()	None	Puts down the turtle's Pen
color()	Color name	Changes the color of the turtle's pen
fillcolor()	Color name	Changes the color of the turtle will use to fill a polygon
heading()	None	Returns the current heading
position()	None	Returns the current position
goto()	x, y	Move the turtle to position x,y
begin_fill()	None	Remember the starting point for a filled polygon
end_fill()	None	Close the polygon and fill with the current fill color
dot()	None	Leave the dot at the current position
stamp()	None	Leaves an impression of a turtle shape at the current location
shape()	shapename	Should be 'arrow', 'classic', 'turtle' or 'circle'

Write a python program to print forward arrow directions?

```
import turtle
t = turtle.Turtle()
t.forward(100)
turtle.mainloop()
```

Sample output:

Write a python program using turtle graphics to print circle?

```
import turtle
t = turtle.Turtle()
t.circle(50)
turtle.mainloop()
```

Sample output:

Write a program to print background color using turtle graphics?

```
import turtle
t = turtle.Turtle()
turtle.bgcolor("red")
turtle.mainloop()
```

Sample and output:

Write a python program to print square shape using turtle graphics?

```
import turtle
my_pen = turtle.Turtle()
for i in range(4):
  my_pen.forward(50)
  my_pen.right(90)
turtle.done()
```

Sample output:

Write a python program to print a star symbol using turtle graphics?

```
import turtle
my_pen = turtle.Turtle()
```

```
for i in range(50):
  my_pen.forward(50)
  my_pen.right(144)
turtle.done()
```

Sample output:

Write a python program to print hexagon shape using turtle graphics?

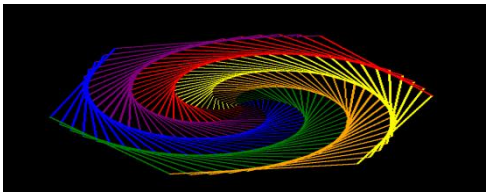
```
import turtle
polygon = turtle.Turtle()
my_num_sides = 6
my_side_length = 70
my_angle = 360.0 / my_num_sides
for i in range(my_num_sides):
  polygon.forward(my_side_length)
  polygon.right(my_angle)
turtle.done()
```

Sample output:

Write a python program to print another pattern using turtle?

```
import turtle
colors = [ "red","purple","blue","green","orange","yellow"]
my_pen = turtle.Pen()
turtle.bgcolor("black")
for x in range(360):
  my_pen.pencolor(colors[x % 6])
  my_pen.width(x/100 + 1)
  my_pen.forward(x)
  my_pen.left(59)
```

Sample output:



FUNCTIONS-PROGRAMS-LIST

1. Design a Python function to sum all the numbers in a list.

Sample List: (8, 2, 3, 0, 7) Expected Output: 20

```
def sum(n):
    total = 0
    for num in n:
        total = total+num
    return total
slist = [8, 2, 3, 0, 7]
result = sum(slist)
print("Sum of the list:", result)
```

Output:

Sum of the list: 20

2. Write a python program to find GCd of two numbers using function?

```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
num1 = int(input("Enter the first number: "))
num2 = int(input("Enter the second number: "))
result = gcd(num1, num2)
print("The GCD of", num1, "and", num2, "is", result)
```

Sample Input and Output:

Enter the first number: 18
Enter the second number: 36
The GCD of 18 and 36 is 18

3. Write a python program to find even or odd number using function?

```
def evenodd(number):
```

```
    if number % 2 == 0:
        return "Even"
    else:
        return "Odd"
num = int(input("Enter a number: "))
result = evenodd(num)
print("The number is", result)
```

Sample and Output:

```
Enter a number: 8
The number is Even
```

4. Write a python program to find sum of a list using function?

```
def sum(numbers):
    total = 0
    for num in numbers:
        total += num
    return total
list = []
n = int(input("Enter the number of elements in the list: "))
for i in range(n):
    num = eval(input("Enter element {}: ".format(i+1)))
    list.append (num)
result = sum(list)
print("The sum of the list is", result)
```

Sample input and output:

```
Enter the number of elements in the list: 5
Enter element 1: 10
Enter element 2: 20
Enter element 3: 30
Enter element 4: 40
Enter element 5: 50
The sum of the list is 150
```

5. Write a python program to find factorial values 1 to 10 using function?

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
for i in range(1, 11):
    result = factorial(i)
    print("Factorial of", i, "is", result)
```

Sample input and output:

```
Factorial of 1 is 1
Factorial of 2 is 2
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
Factorial of 6 is 720
Factorial of 7 is 5040
Factorial of 8 is 40320
Factorial of 9 is 362880
Factorial of 10 is 3628800
```

6. Write a Python function that prints all factors of a given number

```
def fact(n):
    ls = []
    for i in range(1, n + 1):
        if n % i == 0:
            ls.append(i)
    for x in ls:
        print(x, end=' ')
num = 36
print(f"Factors of {num} are:", end=' ')
fact(num)
```

Sample output:

```
Factors of 36 are: 1 2 3 4 6 9 12 18 36
```

7. Read a string from the user and redisplays the same string after removing vowels from it without using built-in function. Write a Python program to illustrate it. Input: “INDIA” Output: “ND”

```
def rv(input):
    vowels = "AEIOUaeiou"
    result = []
    for X in input:
        if X not in vowels:
            result.append(X)
    return ''.join(result)
input = input("Enter a string: ")
output= rv(input)
print("String after removing vowels:", output)
```

SAMPLE OUTPUT:

```
Enter a string: INDIA
String after removing vowels: ND
```

Python Objects

• Objects **contain data and functions (methods)**: Yes, objects are instances of classes in Python, and they can store data (attributes) and have functions (called methods) to interact with or manipulate that data.

• Data **types can vary**: Correct. The attributes of an object can be of any type—strings, integers, lists, other objects, etc.

• Objects **are like mini-programs**: That's a creative and not inaccurate analogy. Objects encapsulate data and behavior, which gives them a sort of "self-contained" nature, much like small programs.

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

    def greet(self):
        if self.is_student:
```

```
        return f"Hi, I'm {self.name}, a student and I'm {self.age} years old."
    else:
        return f"Hi, I'm {self.name} I'm {self.age} years old."
person1 = Person("Tillu", 23, True)
print(person1.greet())
person1.age = 22
print(person1.greet())
```

Sample Output:

```
Hi, I'm Tillu, a student and I'm 23 years old.
Hi, I'm Tillu, a student and I'm 22 years old.
```

UNIT-IV

MODULAR DESIGN: Modules, Top-Down Design, Python Modules. **TEXT FILES:** Using Text Files, String Processing, Exception Handling. **DICTIONARIES AND SETS**: Dictionary Type in Python, Set Data Type.

MODULAR DESIGN: Modules, Top-Down Design, Python Modules

If we need to reuse the code in python then we can define functions , but if we need to reuse number of functions then we have to go for modules concept. A module is nothing but a file with extention of .py , which may contains methods , variables & also classes. In Python , Every python file itself is known as a module . Modules of python .py files that concists of python code. Any python file can be referenced as a Module.In order to use the properties of one module into another module we use the "**import**" statement.

syntax : import required_moduleName

Modules are three types

1) Standard / predefined / built-in modules
2) User defined Modules.
3) 3rd party modules.

1) Predefined modules

These modules are already defined and kept in python software. So when we install python then automatically these standard modules will install in our Machine.For example,math, calender, os , sys, json , datetime, ...

2) User defined Modules.

These modules are defined by users as per their requirement . So here User defined module is nothing but the .py file which contains methods, variables, and also classes.For example: demo.py, sample.py , add.py(), subtract.py

3) 3rd party modules.

These modules are already defined by some other people and kept in internet.So we can download and install in our machines by using "pip" (Python Installer Package).PIP is a

package management system used to install and manage software packages written in python like pymysql , cx_oracle.In python, modules are accessed by using "import" statement.When our current file needed to use the code which is already existed in other files we can import that file(modules)When python import a module called as "Mymodule" for example , the interpreter will first search for a built-in module called Mymodule.If a built-in module is not found , the python interpretor will then search for a file named Mymodule.py in a list of directories that it receives from the sys.path variables.We can import modules in 3 diffrent ways ,

1) import <module_name>

```
import math

print(math.pow(2,4))
```

It is not recomended way in real time programs. Because when importing module name directly then this module contains all members like functions,variables anc classes.So if we want to use only one member of this module then remaining all members are loading not good. here every time we need to use module name when we want to use member of this module. to overcome this problem we can use another way of importing.

2) from <module_name> import *

```
from math import *

print(pow(2,4)) # 16
```

here we can import all members of required module at a time. we can use these members directly without module name.

3) from <module_name> import *

```
from math import pow

print(pow(2,4)) # 16
```

here we can directly importing required module name only what we want. we can use this member directly.

Example:

module1.py file

```
x = 100
def f1():
 print("in f1 of mod1")
def f2():
 print("in f2 of mod1")
```

module2.py file

```
import module1.py
print(module1.x)
module1.f1()
module1.f2()
print("end")
```

output:

100

```
in f1 of mod1

in f2 of mod2

end
```

Note: Whenever we import one module into another module then imported module file will be generated and stored that file into computer hard disc premenently.After generating the compiled file for python module with out sharing the .py file of that module, we can import that module into other module.

Example:

mod3.py

```
def add(a,b):
 return a+b
def sub(a,b):
 return a-b
def abs(a):
```

```
 if a >= 0:
  return a
 else:
  return (-a)
```

mod4.py

```
import mod3
sum = mod3.add(30,20)
print(sum)
sub = mod3.sub(30,20)
print(sub)
pr = mod3.abs(-10)
print(pr)
```

output:

```
50
10
10
```

Renaming a module at the time of importing :

When ever we import a module , to access the properties of that module then compulsary we have to use **modulename.propertyname** .If we rename a module at the time of importing, we can access the propterties of that module by using alicename.propertyName

For Example:

We can import the "math" module by using alias name , which is providing all mathematicl properties

```
import math as m

print("the value of pi is " , m.pi)
```

Output: the value of pi is 3.14159265359

from ... import … :

Inorder to import the specific properties of one module into another module we use

from <module_name> import <property_name>

Example:

mod6.py

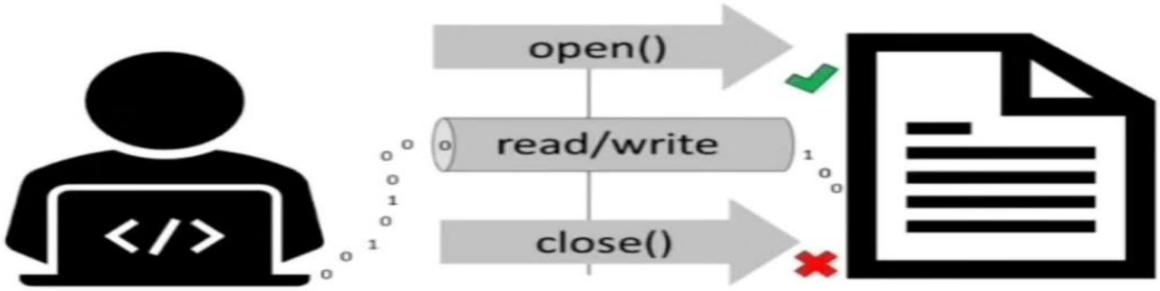
```
from math import pi,e
```

```
print("the value of pi is",pi)
print("the value of e is",e)
```

Output:

the value of pi is 3.14159265354

TEXT FILES: Using Text Files



In any programming languages, program memory allocation takes place in RAM area atthe time of execution of program .RAM is a volatile, so that after execution of the program the memory is going to bede-allocated.Any programming languages programs are good at processing the data but they can notstore data in permanent mannerAfter processing the data , we have to store the data in perminent manner.If the data is stored in permanent manner, we can use the data whenever we want.We can store the data in permanent manner by using file.File is a named location on disk , which is used to store the related data inpermanent manner in the hard disk.By using a python programs, we can put the data into a file , we can get the datafrom a file & we can modify the data of the file.Before going to read or write or modify the data of the file, we have to open the file.

- ✓ By using open() , we can open the file.
- ✓ At the time of opening the file, we have to specify the file modes.
- ✓ Mode of the file indicates what purpose we are going to open the file.

Syntax: file_object = open('demo.txt' , 'r')

here open() method taking file_name as first parameter and required file_mode assecond parameter.Various Properties of File Object:Once we opend a file and we got file object, we can get various details related to that file by using its properties.

name : Name of opened file

mode : Mode in which the file is opened

closed : Returns boolean value indicates that file is closed or not

readable(): Retruns boolean value indicates that whether file is readable or not

writable(): Returns boolean value indicates that whether file is writable or not.

```
f=open("abc.txt",'w')
print("File Name: ",f.name)
print("File Mode: ",f.mode)
print("Is File Readable: ",f.readable())
print("Is File Writable: ",f.writable())
print("Is File Closed : ",f.closed)
f.close()
print("Is File Closed : ",f.closed)
```

OUTPUT:

```
File Name:  abc.txt
File Mode:  w
Is File Readable:  False
Is File Writable:  True
Is File Closed :  False
Is File Closed :  True
```

MODES OF FILE

S.No.	Modes & Description
1	**r** Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	**r+** Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
3	**w** Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
4	**w+** Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.

5	**a** Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
6	**a+** Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
7	**x** open for exclusive creation, failing if the file already exists

```
syntax : x = open( "fileName" , "modes of the file")
```

- ✓ After executing the open function, it will create a file object with the specified modes like x.
- ✓ BY calling the methods on the file object, we can read the data from the file , we can write into file or update the data of a file.
- ✓ After performing the operations on the file object we have to close the file object.
- ✓ By using close() we can close the file object .

READ DATA:

- ✓ To read the data from file object , in python we have read(), readline() and readlines() methods.
- ✓ Using empty read() method we can read entire file object data from where a file_pointer is available and it returns in the form of string type object.

Note: Create a file with name **demo.txt** with some content

```
python is easy language
python is more powerfull language
python is dynamic
```

Q) Write a python program to read the data from the file?

```
file_object = open("demo.txt" , "r" )
print ("file object is opened")
```

```
print(file_object)
data = file_object.read()
print(data)
read(count_number)
```

read() will take the 'int' value as input and It reads given integer number of characters of the file and return as a string format.

```
file_object = open("demo.txt")
print ("file is opened")
print(file_object)
data = file_object.read(10)
print(data)
```

readline():

- ✓ Python facilitates to read the file line by line by using a function readline() method.
- ✓ The empty readline() method reads the current line of the file from the file pointer location, i.e., if we use the readline() method two times, then we can get the first two lines of the file.
- ✓ It is used to Read the entire single line of file where our file pointer is available.
- ✓ Consider the following example which contains a function readline() that reads the first line of our file "demo.txt" containing three lines.

Q) Write a python program to read the first line data from the file ?

```
file_object = open("demo.txt")
print ("file is opened")
print(file_object)
data = file_object.readline()
print(data)
readline(count_number)
```

- ✓ The readline(int) will reads the given specified number of charecters only from the file pointer location but not upto end of the line.
- ✓ If given integer value is more than given file data length then readline(int) will read the data which is available length in a file.

```
fileObject = open('demo.txt', 'r')
data = fileObject.readline(2)
print(data) # 'py'
```

readlines()

- ✓ Is used to Read the entire all lines of file where our file pointer is available.
- ✓ It returns output as a list of items, where each line is a new string of list format.

```
fileObject = open('demo.txt', 'r')
data = fileObject.readlines() # ['line1' , 'line2' , 'line3']
print(data)
```

Output : ['python is easy language\n' , 'python is more powerfull language\n' , 'python is dynamic']

Writing Data to Text Files:

We can write character data to the text files by using the following 2 methods.

write(str)

writelines(list of lines)

```
f=open("abcd.txt",'w')
f.write("Krishna\n")
f.write("Software\n")
f.write("Solutions\n")
print("Data written to the file successfully")
f.close()
```

writelines():

```
f=open("abcd.txt",'w')
list=["sunny\n","bunny\n","vinny\n","chinny"]
f.writelines(list)
print("List of lines written to the file successfully")
f.close()
```

The seek() and tell() Methods:

tell(): We can use tell() method to return current position of the cursor(file pointer) from beginning of the file.The position(index) of first character in files is zero just like string index.

```
f=open("abc.txt","r")
```

```
print(f.tell())
print(f.read(2))
print(f.tell())
print(f.read(3))
print(f.tell())
```

seek():

We can use seek() method to move cursor (file pointer) to specified location

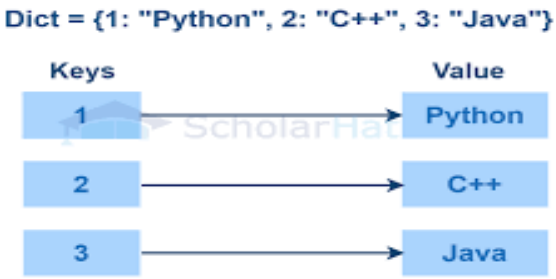
f.seek(offset, fromwhere) ------>offset represents the number of positions

```
data="All Students are GOOD"
f=open("abc.txt","w")
f.write(data)
with open("abc.txt","r+") as f:
text=f.read()
print(text)
print("The Current Cursor Position: ",f.tell())
f.seek(17)
print("The Current Cursor Position: ",f.tell())
f.write("GEMS!!!")
f.seek(0)
text=f.read()
print("Data After Modification:")
print(text)
```

DICTIONARIES AND SETS: Dictionary Type in Python, Set Data Type

A Python dictionary is a data structure that allows us to easily write very efficient codePython dictionaries allow us to associate a value to a unique key, and then to quickly access this value. Dictionaries contains set of key/value pairs which are enclosed between curly braces separated by comma. Here keys are unique.A pair of curly braces creates an empty dictionary: { }The main operations on a dictionary are storing a value with some key andextracting the value with given key.Dictionary keys are not allowed duplicates but dictionary values are allowedduplicates.We can use homogeneous and heterogeneous elements for both keys andvalues.Insertion order is not preserved.Dictionary keys are only

immutable and values are mutable | immutable.Dictionary will not allow indexing and slicing.Dictionaries are indexed by keys, which can be any immutable types likestrings and numbers can always be keys But Tuples can be used as keys if they contain only strings, numbers, or tuples.



We can create dictionary in different ways, ----->> using **{} , dict()**

1. Creating empty dictionary and adding key : value pairs.

```
dic1 = {}
print(dic1)
type(dic1)
dic1['a']=10
dic1['b']=20
dic1['c']=30
dic1['a']=10
dic1['a']=50
print(dic1)
```

2. Creating a dictionary with dict() function:

```
dic1=dict( [ ( 'a ', 10 ) , ( 'b' , 20 ) , ( 'c' , 30 ) , ( 'd' , 40 ) ] )
print(dic1)
type(dic1)
```

3. Creating a dictionary with "curly braces" including "key:value" pairs.

```
dic1={'a':10,'b':20,'c':30,'d':40}
print(dic1)
type(dic1)
```

Access Data from the Dictionary: We can access data by using keys.

```
d = {100:'krishna',200:'milky', 300:'Tillu'}
print(d[100])
print(d[300])
```

If the specified key is not available then we will get KeyError

Update Dictionaries

d[key] = value

If the key is not available then a new entry will be added to the dictionary with the specified key-value pair if the key is already available then old value will be replaced with new value.

```
d={100:"krishna",200:"milky",300:"Tillu"}
print(d)
d[400]="veena"
print(d)
d[100]="sweety"
print(d)
```

Output

```
{100: 'krishna', 200: 'milky', 300: 'Tillu'}
{100: 'krishna', 200: 'milky', 300: 'Tillu', 400: 'veena'}
{100: 'sweety', 200: 'milky', 300: 'Tillu', 400: 'veena'}
```

Delete Elements from Dictionary

del d[key]

- ✓ It deletes entry associated with the specified key.
- ✓ If the key is not available then we will get KeyError.

```
d={100:"krishna",200:"milky",300:"Tillu"}
print(d)
del d[100]
print(d)
del d[400]
```

Output

```
{100: 'krishna', 200: 'milky', 300: 'Tillu'}
{200: 'milky', 300: 'Tillu'}
KeyError: 400
```

2) d.clear()

To remove all entries from the dictionary.

```
d={100:"krishna",200:"milky",300:"Tillu"}
print(d)
```

```
d.clear()
print(d)
```

Output

```
{100:"krishna",200:"milky",300:"Tillu"}
{}
```

del d

To delete total dictionary.Now we cannot access d.

```
d={100:"krishna",200:"milky",300:"Tillu"}
print(d)
del d
print(d)
```

Output

```
{100:"krishna",200:"milky",300:"Tillu"}
NameError: name 'd' is not defined
```

update():

- ✓ The update() method updates the dictionary with the elements from another dictionary object or from an iterable of key/value pairs.
- ✓ The current dictionary will be updated with the all key:value pairs fromanother dictionary.

```
stuDetails={'Id':100,'Name':'Sai', 'subjects':['SQL Server', 'Oracle','Python']}
stuDetails1={'id':1000,'name':'nani'}
stuDetails.update(stuDetails1)
print(stuDetails)
```

{'Id': 100, 'Name': 'Sai', 'subjects': ['SQL Server', 'Oracle', 'Python'], 'id': 1000,'name': 'nani'}

Dictionary functions:

dict():

To create a dictionary

d = dict() =➔It creates empty dictionary

d = dict({100:"krishna",200:"ravi"}) =====>It creates dictionary with specified elements

d = dict([(100,"krishna"),(200,"milky"),(300,"Tillu")]) ➔It creates dictionary with the given list of tuple elements

2) len()

Returns the number of items in the dictionary.

```
dict1 = {101: 'krishna', 102:'milky'};
dict2 = {1001: 'milky', 1002:'19' ,1003:'Tillu'};
print("Length of dict1",len(dict1))
print("Length of dict2",len(dict2))
```

output:

```
Length of dict1 2
Length of dict2 3
```

3) clear():

To remove all elements from the dictionary.

4) get():

To get the value associated with the key

d.get(key)

If the key is available then returns the corresponding value otherwise returns None

5)d.get(key,defaultvalue)

If the key is available then returns the corresponding value otherwise returns default value.

```
d={100:"krishna",200:"milky",300:"Tillu"}
print(d[100]) #krishna
print(d[400]) # KeyError:400
print(d.get(100)) #krishna
print(d.get(400)) #None
print(d.get(100,"Guest")) #krishna
print(d.get(400,"Guest")) #Guest
```

6) pop():

d.pop(key)

It removes the entry associated with the specified key and returns the corresponding value.

If the specified key is not available then we will get KeyError.

```
d={100:"krishna",200:"milky",300:"Tillu"}
print(d.pop(100))
print(d)
print(d.pop(400))
```

Output

```
krishna
{200: 'milky', 300: 'Tillu'}
KeyError: 400
```

7) popitem():

It removes an arbitrary item(key-value) from the dictionaty and returns it.

```
d={100:"durga",200:"milky",300:"Tillu"}
print(d)
print(d.popitem())
print(d)
```

Output

```
{100: 'durga', 200: 'milky', 300: 'Tillu'}
(300, 'Tillu')
{100: 'durga', 200: 'milky'}
```

If the dictionary is empty then we will get KeyError

d={}

print(d.popitem()) ==>**KeyError:** 'popitem(): dictionary is empty'

8) keys():

It returns all keys associated eith dictionary.

```
d={100:"durga",200:"milky",300:"Tillu"}
print(d.keys())
for k in d.keys():
   print(k)
```

Output

```
dict_keys([100, 200, 300])
100
200
300
```

9) values():

It returns all values associated with the dictionary.

```
d={100:"durga",200:"milky",300:"Tillu"}
print(d.values())
```

```
for v in d.values():
print(v)
```

Output

```
dict_values(['durga', 'milky', 'Tillu'])
durga
milky
Tillu
```

10) items():

It returns list of tuples representing key-value pairs.

[(k,v),(k,v),(k,v)]

```
d={100:"durga",200:"milky",300:"Tillu”}
for k,v in d.items():
    print(k,"--",v)
```

Output

```
100 -- durga
200 -- milky
300 -- Tillu
```

Q)Merging two Dictionaries into a New Dictionary using ** symbol OR

Create a Python program to merging two dictionaries

Method-one: Using the | Operator

```
dict_1 = {1: 'a', 2: 'b'}
dict_2 = {2: 'c', 4: 'd'}
print(dict_1 | dict_2)
```

OR

Using copy() and update()

```
dict_1 = {1: 'a', 2: 'b'}
dict_2 = {2: 'c', 4: 'd'}
dict_3 = dict_2.copy()
dict_3.update(dict_1)
print(dict_3)
```

OR

```
d1 = {1:10, 2:20}
d2 = {2:100, 3:30}
d3 = {**d1, **d2}
d3
{1: 10, 2: 100, 3: 30}
```

Explanation :

- ✓ This is generally considered a trick in Python where a single expression is used to merge two dictionaries and stored in a third dictionary.
- ✓ The single expression is **. This does not affect the other two dictionaries.
- ✓ ** implies that an argument is a dictionary. Using ** [double star] is a shortcut that allows you to pass multiple arguments to a function directly using a dictionary.
- ✓ For more information refer **kwargs in Python. Using this we first pass all the elements of the first dictionary into the third one and then pass the second dictionary into the third. This will replace the duplicate keys of the first dictionary.

Q)Discuss the following methods on dictionary i) index() ii) sorted() iii) max()

A dictionary is a data structure that consists of key and value pairs. We can sort a dictionary using two criterias

Sort by key : The dictionary is sorted in ascending order of its keys. The values are not taken care of.

Sort by value : The dictionary is sorted in ascending order of the values.

Sort the dictionary by key

```
dic={2:90, 1: 100, 8: 3, 5: 67, 3: 5}
dic2={}
for i in sorted(dic):
  dic2[i]=dic[i]
print(dic2)
```

output: {1: 100, 2: 90, 3: 5, 5: 67, 8: 3}

Sort dictionary by values

==========

```
dic={2:90, 1: 100, 8: 3, 5: 67, 3: 5}
dic2=dict(sorted(dic.items(),key= lambda x:x[1]))
print(dic2)
```

Output:

```
{8: 3, 3: 5, 5: 67, 2: 90, 1: 100}
```

The **index()** method in Python dictionaries is used to find the index or position of a specified key within the dictionary. index() in dictionaries is not a direct method available.

Key-Value Pairs: A dictionary in Python consists of key-value pairs. Each key is unique and associated with a value.

Accessing Values: You can access the value associated with a key using square brackets [].

```
my_dict = {'a': 1, 'b': 2, 'c': 3}
print(my_dict['b'])
```

The **max()** function in Python is used to find the maximum value among the keys of a dictionary. It can also accept a key function similar to sorted() to find the maximum based on values rather than keys.

```
my_dict = {'c': 3, 'a': 1, 'b': 2}
max_key = max(my_dict)
print(max_key)  # Output: 'c'
```

Dictionary Comprehension:

Comprehension concept applicable for dictionaries also.

```
squares={x:x*x for x in range(1,6)}
print(squares)
doubles={x:2*x for x in range(1,6)}
print(doubles)
```

Output

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
{1: 2, 2: 4, 3: 6, 4: 8, 5: 10}
```

Q) How to perform arithmetic operations on the values of a dictionary?

```
d1={'sub1':80,'sub2':90,'sub3':70,'sub4':80}
print(d1)
{'sub1': 80, 'sub2': 90, 'sub3': 70, 'sub4': 80}
s = sum(d1.values())
print(s) 320
mx = max(d1.values())
print(mx) 90
```

```
mn = min(d1.values())
print(mn) 70
cnt = len(d1.values())
print(cnt) 4
```

String Processing

- ✓ A group/sequence of characters is called String.
- ✓ Python supports str data type to represent string type data.
- ✓ String objects are immutable objects that mean we can't modify the existing string object.
- ✓ Insertion order is preserved in string objects.
- ✓ Every character in the string object is represented with unique index.
- ✓ Python supports both forward and backward indexes.
- ✓ Forward index starts with 0 and negative index starts with -1
- ✓ Python string supports both "concatenation" and "multiplication" of string objects.
- ✓ Strings can be created by enclosing characters inside a single quote or double quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

Quotations in Python:

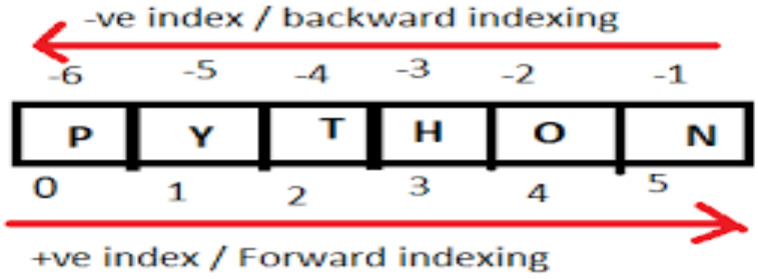
- ✓ Python accepts single ('), double (") and triple ('" or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.
- ✓ Generally triple quotes are used to write the string across multiple lines.

For example1 :

```
word = 'word'
sentence = "This is a sentence."
Paragraph = """This is a paragraph. It is
 made up of multiple lines and sentences.""
```

Accessing Characters By using Index:

Python supports both +ve and -ve Index.



+ve Index means Left to Right (Forward Direction)

-ve Index means Right to Left (Backward Direction)

```
s="krishna"
print(s[3])
print(s[0])
print(s[-2])
print(s[10])
```

output:

```
s
k
n
IndexError: string index out of range
```

Q) Write a Program to Accept some String from the Keyboard and display its Characters by Index wise (both Positive and Negative Index)

```
s=input("Enter Some String:")
l=len(s)
i=0
for x in s:
  print("The character present at positive index {} and at Negative index {} is {}".format(i,i-l,x))
  i=i+1
```

output:

```
Enter Some String:Krishna
The character present at positive index 0 and at Negative index -7 is K
The character present at positive index 1 and at Negative index -6 is r
The character present at positive index 2 and at Negative index -5 is i
```

```
The character present at positive index 3 and at Negative index -4 is s
The character present at positive index 4 and at Negative index -3 is h
The character present at positive index 5 and at Negative index -2 is n
The character present at positive index 6 and at Negative index -1 is a
```

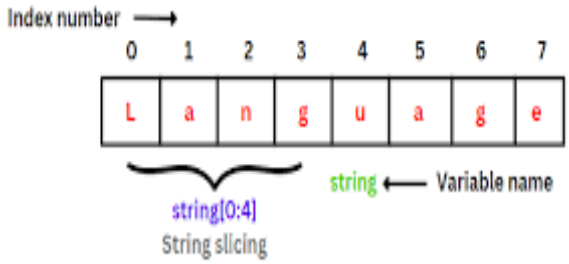
Accessing Characters by using Slice Operator:

Syntax: s[beginindex:endindex:step value]

Begin Index: From where we have to consider slice (substring)

End Index: We have to terminate the slice (substring) at endindex-1

Step value: Incremented Value.




```
s="Learning Python is very very easy!!!"
print(s[1:7:1])
print(s[1:7])
print(s[1:7:2])
print(s[:7])
print(s[7:])
print(s[:])
print(s[::])
print(s[::-1])
```

Output:

```
earnin
earnin
eri
Learnin
g Python is very very easy!!!
Learning Python is very very easy!!!
Learning Python is very very easy!!!
```

```
!!!ysae yrev yrev si nohtyP gninraeL
```

Mathematical Operators for String:

We can apply the following mathematical operators for Strings.

1) + operator for concatenation

2) * operator for repetition

To use + operator for Strings, compulsory both arguments should be str type.

To use * operator for Strings, compulsory one argument should be str and other argument should be int.

len() in-built Function:

We can use len() function to find the number of characters present in the string.

```
s = 'krishna'
print(len(s))
```

Checking Membership:

We can check whether the character or string is the member of another string or not by using **in** and **not in** operators

```
s = input("Enter main string:")
subs = input("Enter sub string:")
if subs in s:
print(subs,"is found in main string")
else:
print(subs,"is not found in main string")
```

Comparison of Strings:

We can use comparison operators (<, <=, >, >=) and equality operators (==, !=) for strings. Comparison will be performed based on alphabetical order.

Example: develop a python program to find substring within a given string?

```
s1=input("Enter first string:")
2s2=input("Enter Second string:")
if s1==s2:
print("Both strings are equal")
elif s1<s2:
print("First String is less than Second String")
else:
```

```
print("First String is greater than Second String")
```

find():

syntax: **s.find(substring)**

Returns index of first occurrence of the given substring. If it is not available then we will get -1.

```
s="Learning Python is very easy"
print(s.find("Python")) #9
print(s.find("Java")) # -1
print(s.find("r"))#3
print(s.rfind("r"))#21
```

Note: By default find() method can search total string. We can also specify the boundaries to search.

s.find(substring,bEgin,end)

It will always search from bEgin index to end-1 index.

```
s="durgaravipavanshiva"
print(s.find('a'))#4
print(s.find('a',7,15))#10
print(s.find('z',7,15))#-1
```

Counting substring in the given String:

We can find the number of occurrences of substring present in the given string by using **count()** method.

1) **s.count(substring) :** It will search through out the string.

2) **s.count(substring, begin, end) :** It will search from bEgin index to end-1 index.

```
s="abcabcabcabcadda"
print(s.count('a'))
print(s.count('ab'))
print(s.count('a',3,7))
```

Replacing a String with another String:

Syntax: **s.replace(oldstring, newstring)**

inside s, every occurrence of old String will be replaced with new String.

```
s = "Learning Python is very difficult"
```

```
s1 = s.replace("difficult","easy")
print(s1)
```

Output: Learning Python is very easy

Splitting of Strings:

We can split the given string according to specified seperator by using split() method.

```
l = s.split(seperator)
```

The default seperator is space. The return type of split() method is List.

```
s="krishna software solutions"
l=s.split()
for x in l:
  print(x)
```

Output:

```
krishna
software
solutions
```

Joining of Strings:

We can join a Group of Strings (List OR Tuple) wrt the given Seperator.

```
s = seperator.join(group of strings)
```

Eg 1:

```
t = ('sunny', 'bunny', 'chinny')
s = '-'.join(t)
print(s)
```

Output: sunny-bunny-chinny

Eg 2:

```
l = ['hyderabad', 'singapore', 'london', 'dubai']
s = ':'.join(l)
print(s)
```

Output: hyderabad:singapore:london:dubai

Changing Case of a String:

We can change case of a string by using the following 4 methods.

1) **upper**() :To convert all characters to upper case

2) **lower**() :To convert all characters to lower case

3) **swapcase**() :Converts all lower case characters to upper case and all upper case characters to lower case

4) **title**() : To convert all character to title case. i.e first character in every word should be upper case and all remaining characters should be in lower case.

5) **capitalize**() :Only first character will be converted to upper case and all remaining characters can be converted to lower case

```
s = 'learning Python is very Easy'
print(s.upper())
print(s.lower())
print(s.swapcase())
print(s.title())
print(s.capitalize())
```

Output:

```
LEARNING PYTHON IS VERY EASY
learning python is very easy
LEARNING pYTHON IS VERY eASY
Learning Python Is Very Easy
Learning python is very easy
```

Checking Starting and Ending Part of the String:

Python contains the following methods for this purpose

1) s.startswith(substring)

2) s.endswith(substring)

```
s = 'learning Python is very easy'
print(s.startswith('learning'))
print(s.endswith('learning'))
print(s.endswith('easy'))
```

To Check Type of Characters Present in a String:

Python contains the following methods for this purpose.

1) **isalnum**(): Returns True if all characters are alphanumeric(a to z , A to Z ,0 to9)

2) **isalpha**(): Returns True if all characters are only alphabet symbols(a to z,A to Z)

3) **isdigit**(): Returns True if all characters are digits only(0 to 9)

4) **islower**(): Returns True if all characters are lower case alphabet symbols

5) **isupper**(): Returns True if all characters are upper case aplhabet symbols

6) **istitle**(): Returns True if string is in title case

7) **isspace**(): Returns True if string contains only spaces

```
s=input("Enter any character:")
if s.isalnum():
   print("Alpha Numeric Character")
 if s.isalpha():
   print("Alphabet character")
 if s.islower():
   print("Lower case alphabet character")
 else:
   print("Upper case alphabet character")
else:
  print("it is a digit")
elif s.isspace():
  print("It is space character")
 else:
  print("Non Space Special Character")
```

Q) Write a Program to Reverse the given String

Input: python

Output: nohtyp

1st Way:

```
s = input("Enter Some String:")
print(s[::-1])
```

2nd Way:

```
s = input("Enter Some String:")
print(''.join(reversed(s)))
```

3rd Way:

```
s = input("Enter Some String:")
i=len(s)-1
target=''
```

```
while i>=0:
  target=target+s[i]
  i=i-1
print(target)
```

Write a Program to Print Characters at Odd Position and Even Position for the given String?

1st Way:

```
s = input("Enter Some String:")
print("Characters at Even Position:",s[0::2])
print("Characters at Odd Position:",s[1::2])
```

2nd Way:

```
s=input("Enter Some String:")
i=0
print("Characters at Even Position:")
while i< len(s):
   print(s[i],end=',')
   i=i+2
 print()
print("Characters at Odd Position:")
i=1
while i< len(s):
    print(s[i],end=',')
    i=i+2
```

SET- DATA Type:

- ✓ A set is unordered collection of unique elements.
- ✓ Set is commonly used in membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.
- ✓ Set will not allow duplicate values.
- ✓ Insertion order is not preserved but elements can be sorted
- ✓ The major advantage of using a set is as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.

- ✓ Sets do not support indexing, slicing,
- ✓ Sets do not support concatenation and multiplication.
- ✓ There are currently two built-in set types,

Set:

- ✓ The set type is mutable means the contents of set can be changed using methods like add(), update() and remove(), discard(), pop() , clear().
- ✓ Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set.

Frozenset:

- ✓ The frozen sets are the immutable form of the normal sets, i.e., the items of the frozen set cannot be changed and therefore it can be used as a key in the dictionary.
- ✓ The elements of the frozen set cannot be changed after the creation. We cannot change or append the content of the frozen sets by using the methods like add() or remove().
- ✓ The frozenset() method is used to create the frozenset object. The iterable sequence is passed into this method which is converted into the frozen set as a return type of the method.

```
Frozenset = frozenset([1,2,3,4,5])
print(type(Frozenset))
print("\n printing the content of frozen set...")
for i in Frozenset:
 print(i);
Frozenset.add(6)
```

We can create a set in different ways,

1. Creating an empty set using set() and add elements to that empty set.

Example:

```
set1 = set()
set1.add(10)
set1.add(20)
set1.add(30)
set1.add(10)
print(set1)
```

2. Creating a set with elements using set().

```
set2=set([1,2,4,'a',2+4j,True])
print(set2) {1, 2, 4, (2+4j), 'a'}
type(set2) <class 'set'>
```

3.Creating a set with curly braces ----->> { }

```
set3={1,2,3,4,"krishna",True}
print(set3) {1, 2, 3, 4, 'krishna'}
type(set3) <class 'set'>
```

Set Functions:

Adding items to the set:

Python provides the add() method and update() method which can be used to add some particular item to the set. The add() method is used to add a single element whereas the update() method is used to add multiple elements to the set.

add():

This method is used to add new elements in to existing set.

```
set1 = {1,2,3,4,5}
print(set1)
set1.add(6)
set1.add(7)
print(set1)
```

update():

- ✓ To add more than one item in the set, Python provides the update() method. It accepts iterable object as an argument.

```
s1 = set()
s1.update([10,20,30,40])
s1
{40, 10, 20, 30}
s1.update((50,60))
s1
```

Removing items from the set:

remove(element): It will remove elements from the set, if that element is not found then it will throw error like KeyError

```
se1={1,2,3,4,5}
print(se1)
type(se1)
se1.remove(5)
se1.remove(4)
se1.remove(15)
```

Error: KeyError: 15

discard():

- ✓ It will remove elements from the set, if that element is not found in the set then it will do nothing. means it will not return any exception here.

```
se1={1,2,3,4,5}
print(se1)
se1.discard(7)
se1.discard(20)
se1.discard(5)
print(se1)
```

pop():

We can also use the pop() method to remove the item. Generally, the pop() method will always remove the last item but the set is unordered, we can't determine which element will be popped out from set.

```
s1 = {90,40, 10, 20, 30}
s1.pop()
40
s1.pop()
10
```

Assignment operator

By using assignment operator, if we assigning given set object in to another object then both can share the same memory address.

```
set1 = {1, 2, 3, 4, 5}
set2 = set1
 set1
{1, 2, 3, 4, 5}
```

```
set2
{1, 2, 3, 4, 5}
id(set1)
2693878076136
id(set3)
2693878076136
```

copy():

This function copies the elements of one set to another new set and also it creates new memory value.

copy() method always creates new memory for new set object. so both memories are different but values are same.

```
set1={1,2,3,4,5}
set2=set1.copy() #copying se1 elements to se2
set1 {1, 2, 3, 4, 5}
set2 {1, 2, 3, 4, 5} # id values different.
id(set1)
2693878076136
id(set2)
2693878076360
```

clear():

By using clear() function we can clear or remove all elements from the given set object.

```
se1={1,2,3,4,5}
print(se1) {1, 2, 3, 4, 5}
type(se1) <class 'set'>
se1.clear() #clearing the se1, so se1 will become empty set.
 print(se1) set()
```

Python Set Operations:

Set can be performed mathematical operation such as union, intersection, difference, and symmetric difference. Python provides the facility to carry out these operations with operators or methods. We describe these operations as follows.

isdisjoint():

This function returns True if both are "empty sets" or if both sets "contains

non-matching" elements.if atleast one elemet matching also returns False value.

```
se1 = set()
se2 = set()
se1.isdisjoint(se2) True
se1=set(5)
se2={1,2,3}
se1.isdisjoint(se2) True
se1={1,2,3}
se2={1,2,3,4}
se1.isdisjoint(se2) False
```

issubset():

- ✓ x.issubset(y) returns True, if x is a subset of y.
- ✓ " <= " is an abbreviation for "Subset of".

```
se1={1,2,3,4,5}
se2={1,2,3}
se2.issubset(se1) True
se1.issubset(se2) False
```

Or

```
se2 <= se1 True
se1 <= se2 False
```

issuperset()

- ✓ x.issuperset(y) returns True, if x is a superset of y.
- ✓ " >= " is an abbreviation for "issuperset of"

```
se1={1,2,3,4,5}
se2={1,2,3}
se2.issuperset(se1) False
se1.issuperset(se2) True
se2 >= se1 False
se1 >= se2 True
```

Membership:

- ✓ We can also check the elements whether they belong to set or not,

```
se1={1,2,3,"Python",3+5j,8}
```

```
4 in se1 False
1 in se1 True
"Python" in se1 True
10 not in se1 True
"krishna" not in se1 True
```

union():

It returns the union of two sets, that means it returns all the values from both sets except duplicate values. The same result we can get by using '|' between two sets

Syntax: `<First_Set>.union(<Second_Set>) or<First_Set> | <Second_Set>`

```
se1={1,2,3,4,5}
se2={1,2,3,6,7}
se1.union(se2) {1, 2, 3, 4, 5, 6, 7} or
se1|se2 {1, 2, 3, 4, 5, 6, 7}
```

Or

```
se2.union(se1) {1, 2, 3, 4, 5, 6, 7}
se2|se1 {1, 2, 3, 4, 5, 6, 7}
```

intersection():

It returns an intersection elements of two sets, that means it returns only common elements from both sets.That same operation we can get by sing '&' operator.

Syntax: `<First_Set>.intersection(<Second_Set>) or <First_Set> & <Second_Set>`

```
se1={1,2,3,4,5}
se2={1,2,3,6,7}
se1.intersection(se2) {1, 2, 3} or
se1&se2 {1, 2, 3}
```

Or

```
se2.intersection(se1) {1, 2, 3}
se2&se1 {1, 2, 3}
```

diffferenece():

It returns all elements from first set which are not there in the second set.

Syntax: `<First_set>.difference(<Secnd_Set>) or <First_Set> - <Second_Set>`

```
se1={1,2,3,4,5}
se2={1,2,3,6,7}
se1.difference(se2) {4, 5} or
se1-se2 {4, 5}
```

Or

```
se2.difference(se1) {6, 7} or
se2-se1 {6, 7}
```

intersection_update():

- ✓ The intersection_update() method removes the items from the original set that are not present in both the sets (all the sets if more than one are specified).
- ✓ The intersection_update() method is different from the intersection() method since it modifies the original set by removing the unwanted items, on the other hand, the intersection() method returns a new set.

Syntax: <First_Set>.intersection_update(<Second_Set>)

```
se1={1,2,3,4,5}
se2={1,2,3,6,7}
se1.intersection_update(se2)
print(se1) {1, 2, 3}
print(se2) {1, 2, 3, 6, 7}
```

differenece_update():

The result of difference between two sets will in First_Set.

Syntax: <First_Set>.difference_update(<Second_Set>)

```
se1={1,2,3,4,5}
se2={1,2,3,6,7}
se1.difference_update(se2)
print(se1)
print(se2)
```

Q) Give a comparison between lists, tuples, dictionaries and sets

List	Tuple	Dictionary

List contains heterogeneous elements	Tuple contains heterogeneous elements	Whereas a dictionary contains key-value pairs.
A List is represented by [].	A Tuple is represented by ().	A Dictionary is represented by {}.
Lists in Python are mutable.	These are immutable.	These are mutable.
Lists are ordered.	Tuples are unordered.	Dictionaries are ordered.

- ✓ The list and tuple can be created by using the elements without any defining the key whereas the dictionary uses the key and value pairs.
- ✓ If we want to create a group of elements with some key name, then we can go for dictionary as it accepts key and value.
- ✓ When we want to list out few elements and want to make changes later as per our requirement we can go for list.
- ✓ When we want to combine few elements into group and don't want to apply any changes further then we can go for tuple. Let's see the combined example of the list, tuple and the dictionary.

Exception Handling

Exception definition :An exception is an event, which occurs during the excution of a program,that disrupts the normal flow of the program instructions We can handle the exceptions at runtime. But we cannot handle errors.Exceptions are related to application, where as errors are related to environment in which the application is running.An error is a term which is used to describe any issue that arisesunexpectedly that cause a computer do not functioning properly.

Debugging: The process of finding and eliminating the errors is called Debugging. In Python We have 2 types of Errors:

1) SyntaxError

2) RuntimeError

1 Syntax error :

The errors which occurs because of "invalid syntax" are known as "Syntax errors". Interpreter will check for syntax errors when ever we run the python program

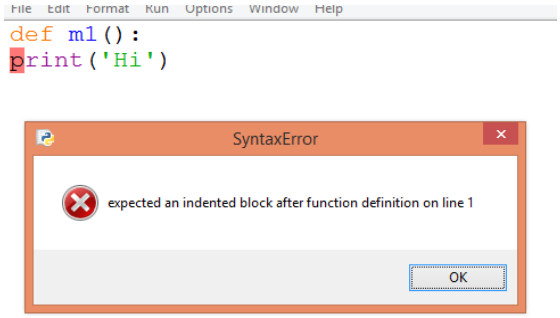
- ✓ If syntax error is found , Then no byte code is generating.
- ✓ With out byte code, program excution is not possible.
- ✓ Developers only responsible to solve these errors.

```
def m1():
print('Hi')
```

Example1:

```
def m1():
print('Hi')
```

Output : SyntaxError : Expected an indented block



2) RUNTIME ERRORS :

- ✓ The errors which occurs at the time of excution of a program are known as runtime errors.
- ✓ We get Runtime Errors because of programing logic, invalid input,memory related issues etc...
- ✓ For every RuntimeError , Python is providing corresponding exception class is available.

  Example1: KeyError , IndexError, ValueError, NameError etc ....
- ✓ At the time of execution of a program if any Runtime Error is occur then internally corresponding Runtime Error representation classes object will be created

What is Exception

An unwanted and unexpected event that disturbs normal flow of program is called exception.

Eg:

- ✓ ZeroDivisionError
- ✓ TypeError
- ✓ ValueError
- ✓ FileNotFoundError
- ✓ EOFError

It is highly recommended to handle exceptions. The main objective of exception handling

is Graceful Termination of the program (i.e we should not block our resources and we should not miss anything). Exception handling does not mean repairing exception. We have to define alternative way to continue rest of the program normally

Default Exception Handing in Python:

- ✓ Every exception in Python is an object. For every exception type the corresponding classes are available.
- ✓ Whevever an exception occurs PVM will create the corresponding exception object and will check for handling code. If handling code is not available then Python interpreter terminates the program abnormally and prints corresponding exception information to the console.
- ✓ The rest of the program won't be executed

```
print("Hello")
print(10/0)
print("Hi")
```

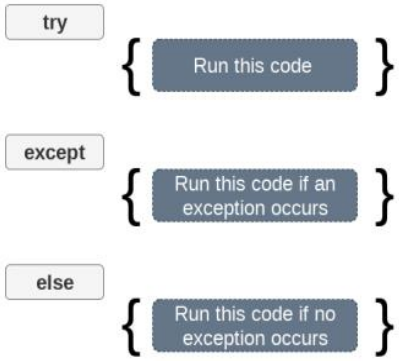
D:\Python_classes>py test.py

```
Hello
Traceback (most recent call last):
 File "test.py", line 2, in <module>
 print(10/0)
ZeroDivisionError: division by zero
```

Customized Exception Handling by using try-except:

It is highly recommended to handle exceptions.

The code which may raise exception is called risky code and we have to take risky code inside try block. The corresponding handling code we have to take inside except block.



Without try-except:

```
print("stmt-1")
print(10/0)
print("stmt-3")
```

Output

```
stmt-1
ZeroDivisionError: division by zero
Abnormal termination/Non-Graceful Termination
```

With try-except:

```
print("stmt-1")
try:
print(10/0)
except ZeroDivisionError:
print(10/2)
print("stmt-3")
```

Output

```
stmt-1
5.0
stmt-3
Normal termination/Graceful Termination
```

`Control Flow in try-except:`

```
try:
 stmt-1
 stmt-2
 stmt-3
except xxxxxxx:
 stmt-4
stmt-5
```

Case-1: If there is no exception

1,2,3,5 and Normal Termination

Case-2: If an exception raised at stmt-2 and corresponding except block matched

1,4,5 Normal Termination

Case-3: If an exception rose at stmt-2 and corresponding except block not matched

1, Abnormal Termination

Case-4: If an exception rose at stmt-4 or at stmt-5 then it is always abnormal termination.

How to Print Exception Information:

```
try:
print(10/0)
except ZeroDivisionError as msg:
 print("exception raised and its description is:",msg)
```

Output exception raised and its description is: division by zero

try with Multiple except Blocks:

The way of handling exception is varied from exception to exception.Hence for every exception type a seperate except block we have to provide. i.e try with multiple except blocks is possible and recommended to use.

Eg:

```
try:
 -------
 -------
 -------
except ZeroDivisionError:
 perform alternative arithmetic operation
except FileNotFoundError:
 use local file instead of remote file
```

If try with multiple except blocks available then based on raised exception the corresponding except block will be executed.

```
try:
x=int(input("Enter First Number: "))
y=int(input("Enter Second Number: "))
print(x/y)
except ZeroDivisionError :
print("Can't Divide with Zero")
except ValueError:
print("please provide int value only")
```

D:\Python_classes>py test.py

```
Enter First Number: 10
Enter Second Number: 2
5.0
```

D:\Python_classes>py test.py

```
Enter First Number: 10
Enter Second Number: 0
Can't Divide with Zero
```

D:\Python_classes>py test.py

```
Enter First Number: 10
Enter Second Number: ten
please provide int value only
```

If try with multiple except blocks available then the order of these except blocks is important .Python interpreter will always consider from top to bottom until matched except block identified.

Single except Block that can handle Multiple Exceptions:

We can write a single except block that can handle multiple different types of exceptions.

except (Exception1,Exception2,exception3,..): OR

except (Exception1,Exception2,exception3,..) as msg :

Parentheses are mandatory and this group of exceptions internally considered as tuple.

```
try:
x=int(input("Enter First Number: "))
y=int(input("Enter Second Number: "))
print(x/y)
 except (ZeroDivisionError,ValueError) as msg:
print("Plz Provide valid numbers only and problem is: ",msg)
```

D:\Python_classes>py test.py

```
Enter First Number: 10
Enter Second Number: 0
Plz Provide valid numbers only and problem is: division by zero
```

finally Block:

☕ It is not recommended to maintain clean up code(Resource Deallocating Code or Resource Releasing code) inside try block because there is no guarentee for the execution of every statement inside try block always.

☕ It is not recommended to maintain clean up code inside except block, because if there is no exception then except block won't be executed.

☕ Hence we required some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether exception handled or not handled. Such type of best place is nothing but finally block.

☕ Hence the main purpose of finally block is to maintain clean up code

Syntax:

```
try:
 Risky Code
except:
 Handling Code
finally:
 Cleanup code
```

The speciality of finally block is it will be executed always whether exception raised or not raised and whether exception handled or not handled.

Case-1: If there is no exception

```
try:
print("try")
except:
print("except")
finally:
print("finally")
```

Output

```
try
finally
```

Case-2: If there is an exception raised but handled

```
try:
print("try")
print(10/0)
```

```
except ZeroDivisionError:
print("except")
finally:
print("finally")
```

Output

```
try
except
finally
```

Types of Exceptions:

In Python there are 2 types of exceptions are possible.

1) Predefined Exceptions

2) User Definded Exceptions

1)Predefined Exceptions:

- ✓ Also known as inbuilt exceptions.
- ✓ The exceptions which are raised automatically by Python virtual machine whenver a

particular event occurs are called pre defined exceptions.

Eg 1: Whenever we are trying to perform Division by zero, automatically Python will raise ZeroDivisionError.

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

2)User Defined Exceptions:

- ✓ Also known as Customized Exceptions or Programatic Exceptions
- ✓ Some time we have to define and raise exceptions explicitly to indicate that something goes wrong, such type of exceptions are called User Defined Exceptions or Customized Exceptions
- ✓ Programmer is responsible to define these exceptions and Python not having any idea about these. Hence we have to raise explicitly based on our requirement by using "raise" keyword.

Eg:

- ✓ InSufficientFundsException
- ✓ InvalidInputException
- ✓ TooYoungException
- ✓ TooOldException

PROGRAMS:

1. Write a Python code to search a string in the given list

```
l = [1, 2.0, 'milky','Tillu','krishna','Ridwa']
s = input("ener a string:")
if s in l:
        print(f'{s} is present in the list')
else:
        print(f'{s} is not present in the list')
```

2. Develop a python program to access the elements of a Nested dictionary

In Python, a nested dictionary is a dictionary inside a dictionary. It's a collection of dictionaries into one single dictionary.

```
nested_dict = { 'dictA': {'key_1': 'value_1'},
        'dictB': {'key_2': 'value_2'}}
```

Create a Nested Dictionary

```
people = {1: {'name': 'Tillu', 'age': '22', 'sex': 'Male'},
    2: {'name': 'Milky', 'age': '22', 'sex': 'Female'}}
print(people)
```

Access elements of a Nested Dictionary

To access element of a nested dictionary, we use indexing [] syntax in Python.

```
people = {1: {'name': 'Tillu', 'age': '22', 'sex': 'Male'},
    2: {'name': 'Milky', 'age': '22', 'sex': 'Female'}}
print(people[1]['name'])
print(people[1]['age'])
print(people[1]['sex'])
```

3. Explain the process of how to delete dictionary from multiple dictionaries

```
mylist = [{"id" : 1, "data" : "HappY"},
```

```
                    {"id" : 2, "data" : "BirthDaY"},
                    {"id" : 3, "data" : "Tillu"}]
print("The original list is: " + str(mylist))
index = None
for i, d in enumerate(mylist):
        if d['id'] == 2:
                index = i
                break
if index is not None:
        mylist.pop(index)
print("List after deletion of dictionary: " + str(mylist))
```

sample output:

```
The original list is: [{'id': 1, 'data': 'HappY'}, {'id': 2, 'data': 'BirthDaY'}, {'id': 3, 'data': 'Tillu'}]
List after deletion of dictionary: [{'id': 1, 'data': 'HappY'}, {'id': 3, 'data': 'Tillu'}]
```

How to convert list to a dictionary

```
pets= ['dog','cat','guinea pig', 'parrot']
# add one value to all
pets_owner = {animal:'Junnu' for animal in pets}
print(pets_owner)
```

sample output:

{'dog': 'Junnu', 'cat': 'Junnu', 'guinea pig': 'Junnu', 'parrot': 'Junnu'}

What is a Nested Dictionary? How is it created?

A dictionary inside the dictionary is known as a “Nested Dictionary”. For ex

```
dictionary1 = {
 1 : {'roll': '101', 'name': 'sam'},
2 : {'roll': '102', 'name': 'ram'}
 }
print(dictionary1)
```

Output: {1: {'roll': '101', 'name': 'sam'}, 2: {'roll': '102', 'name': 'ram'}}

The elements of nested dictionary can be accessed using

```
print(dictionary[1]['roll'])
```

Output: 101

Create a list of tuples from the dictionary

The list of tuples can be created in following way:

```
dict1 = { 1: 'a', 2: 'b', 3: 'c' }
lst1 = list(dict1.items())
print(lst1)
```

Output: [(1, 'a'), (2, 'b'), (3, 'c')]

UNIT-V

OBJECT-ORIENTED PROGRAMMING: What is Object-Oriented Programming, features of OOP, Classes, objects, type's of variables, methods, Constructors with examples, Inheritance, Polymorphism, Data Abstraction, Encapsulation

OBJECT-ORIENTED PROGRAMMING: What is Object-Oriented Programming

Generally all Programming languages are basically classified into two types.

1. Functional Oriented Programming.

2. Object Oriented Programming.

- If you want to develop one-to-one applications then use functional oriented programming.
- To develop operating system, drivers, compilers we have to use functional programming.
- If you want to develop one-to-many applications then we have to use object oriented programming.
- To develop web applications like Amazon, Swiggy, IRCTC, etc we have to use object oriented programming.
- In 'OOPL' we have to develop programs by using class and objects.
- Like other general-purpose programming languages, Python is also an object oriented programming language since its beginning.
- It allows us to develop applications using an Object-Oriented approach. In
- Python , we can easily create and use classes and objects.
- An object-oriented paradigm is to design the program using classes and objects.
- The object is related to real-word entities such as book, house, pencil, etc.
- The oops concept focuses on writing the reusable code.
- It is a widespread technique to solve the problem by creating objects.

CLASS:

- In Python every thing is an object. To create objects we required some Model or Plan or Blue print, which is nothing but class.
- We can write a class to represent properties (attributes) and actions (behaviour) ofobject.
- Properties can be represented by variables

- Actions can be represented by Methods.
- Hence class contains both variables and methods.

To define a Class

We can define a class by using **class** keyword.

Syntax:

```
class className:

 ''' documenttation string '''
 variables:instance variables,static and local variables
 methods: instance methods,static methods,class methods
```

Documentation string represents description of the class. Within the class doc string is always optional. We can get doc string by using the following 2 ways.

```
print(classname.__doc__)

help(classname)
```

Example:

```
class Student:
""" This is student class with required data'''
print(Student.__doc__)
help(Student)
```

Object:

- The object is an entity that has state and behavior.
- It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.
- Everything in Python is an object, and almost everything has attributes and methods.

- All functions have a built-in attribute __doc__, which returns the docstring defined in the function source code.
- When we define a class, it needs to create an object to allocate the memory.

Syntax to Create Object: reference variable = classname()

Example: `s = Student()`

Type's of variables

Types of Variables:

Inside Python class 3 types of variables are allowed.

1) Instance Variables (Object Level Variables)

2) Static Variables (Class Level Variables)

3) Local variables (Method Level Variables)

1)Instance Variables:

- If the value of a variable is varied from object to object, then such type of variables are called instance variables.
- For every object a separate copy of instance variables will be created.

2)Static Variables:

If the value of a variable is not varied from object to object, such type of variables we have to declare with in the class directly but outside of methods. Such types of variables are called Static variables.

For total class only one copy of static variable will be created and shared by all objects of that class.

We can access static variables either by class name or by object reference. But recommended to use class name.

What is Reference Variable

- The variable which can be used to refer object is called reference variable.
- By using reference variable, we can access properties and methods of object.

Program: **Write a Python program to create a Student class and Creates an object to it. Call the method talk() to display student details**

```
class Student:
def __init__(self,name,rollno,marks):
self.name=name
self.rollno=rollno
self.marks=marks
def talk(self):
print("Hello My Name is:",self.name)
print("My Rollno is:",self.rollno)
print("My Marks are:",self.marks)
s1=Student("krishna",101,80)
s1.talk()
```

The self-parameter

- The self-parameter refers to the current instance of the class and accesses the class variables.
- We can use anything instead of self, but it must be the first parameter of any function which belongs to the class.

Example:

```
class car:
 def __init__(self,modelname, year):
 self.modelname = modelname
 self.year = year
 def display(self):
 print("The Car modelname is :", self.modelname)
 print("The Car launch year :",self.year)
c1 = car("Toyota", 2016)
c1.display()
```

Output:

```
Toyota 2016
```

- In the above example, we have created the class named car, and it has two attributes modelname and year.
- We have created a c1 object to access the class attribute. The c1 object will allocate memory for these values.

Method

- A method is a set of statements to perform specific operations on data.
- The method is a function that is associated with an object.
- In Python, a method is not unique to class instances. Any object type can have methods.

```
def display(self):
   print('Employee id:101)
```

Q. Create a class with some variables and methods. How to display all members of class?

```
a = 10
 b = 20
 def message(self):
 print('hello')
 return 'ok'
obj = Sample()
print('a value is :',obj.a)
print('b value is :',obj.b)
print(obj.message())
```

Constructors with examples

- A constructor is a special type of method (function) which is used to initialize the instance members of the class.

- In C++ or Java, the constructor has the same name as its class, but it treats constructor differently in Python. It is used to create an object.

Creating the constructor in python:

- In Python, the method the __init__() represents the constructor of the class.
- This method is called when the class is instantiated (object created).
- It accepts the self-parameter as a first argument which allows accessing the attributes or method of the class.
- We can pass any number of arguments at the time of creating the class object, depending upon the __init__() definition.
- It is mostly used to initialize the class attributes. Every class must have a constructor, even if it simply relies on the default constructor.

Example:

```
class Employee:
 def __init__(self, name, id):
 self.id = id
 self.name = name
 def display(self):
print(f"ID: {self.id} \nName: {self.name}")
emp1 = Employee("Milky", 101)
emp2 = Employee("Tillu", 102)
emp1.display()
emp2.display()
```

Output:

```
ID: 101
Name: Milky
ID: 102
Name: Tillu
```

Constructors can be of two types.

- Parameterized Constructor
- Non-parameterized Constructor

Constructor definition is executed when we create the object of this class. Constructors also verify that there are enough resources for the object to perform any start-up task.

Python Non-Parameterized Constructor

- The non-parameterized constructor uses when we do not want to manipulate the value or the constructor that has only self as an argument.

Example:

```
class Student:
def __init__(self):
 print("This is non parametrized constructor")
 def show(self,name):
 print("Hello",name)
student = Student()
student.show("Milky")
```

Python Parameterized Constructor:

- The parameterized constructor has multiple parameters along with the self.

```
class Student:
def __init__(self, name):
 print("This is parametrized constructor")
 self.name = name
 def show(self):
 print("Hello",self.name)
student = Student("Tillu")
```

```
student.show()
```

Output:

```
This is parametrized constructor

Hello Tillu
```

Inheritance

Inheritance is one of the powerful features of Python since it is an Object Oriented Programming Language.

What is Inheritance

Inheritance in python is a way through which we can use the property of one class into another class. This ensures the reusability of code and hence reduces the complexity of the program.The class which inherits other class is called Base Class and the class which has been inherited from other class is known as derived class.Python Inheritance Block Diagram:

```
Base Class

 |

 |

 |

Derive Class
```

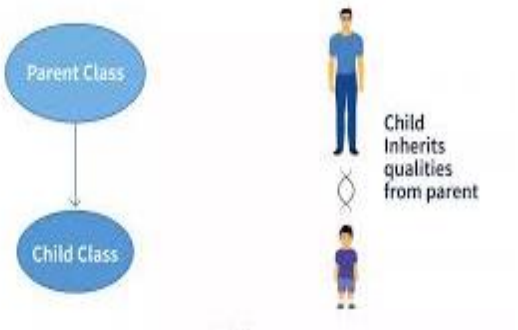
Python Inheritance Syntax:

```
class Base_Class:
   #statements
 pass
class Derived_Class(Base_Class):
 #statements
 Pass
```

In Python, we have several types of Inheritance concepts, they are like,

- Single Level Inheritance
- MultiLevel Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

1. Single Level Inheritance: The concept of inheriting the properties from one class to another class is known as single inheritance

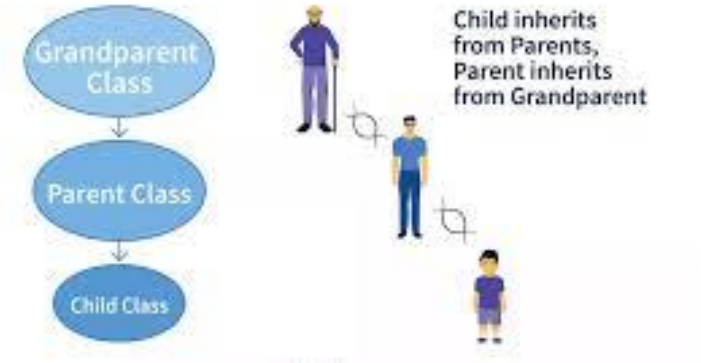



```
class P:
def m1(self):
    print("Parent Method")
class C(P):
def m2(self):
    print("Child Method")
c=C()
c.m1()
c.m2()
```

Output:

```
Parent Method
Child Method
```

Multilevel Inheritance : The concept of inheriting the properties from multiple classes to single class with the concept of one after another is known as multilevel inheritance

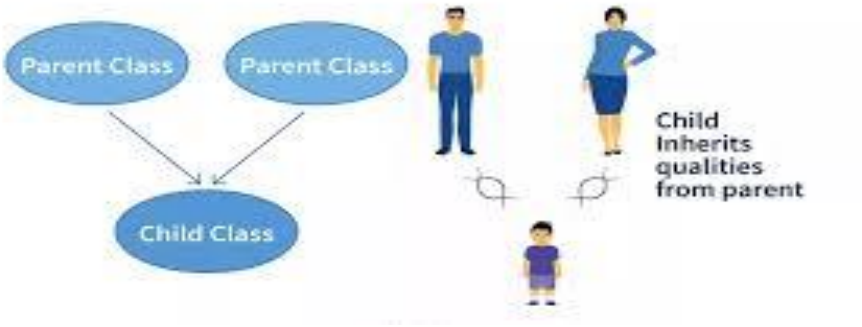



```
class P:
def m1(self):
print("Parent Method")
class C(P):
def m2(self):
print("Child Method")
class CC(C):
def m3(self):
 print("Sub Child Method")
10) c=CC()
c.m1()
c.m2()
c.m3()
```

Output:

```
Parent Method
Child Method
Sub Child Method
```

Multiple Inheritance: The concept of inheriting the properties from multiple classes into a single class at a time, is known as multiple inheritance.




```
class P1:
def m1(self):
print("Parent1 Method")
class P2:
def m2(self):
print("Parent2 Method")
class C(P1,P2):
def m3(self):
print("Child2 Method")
c=C()
c.m1()
c.m2()
c.m3()
```

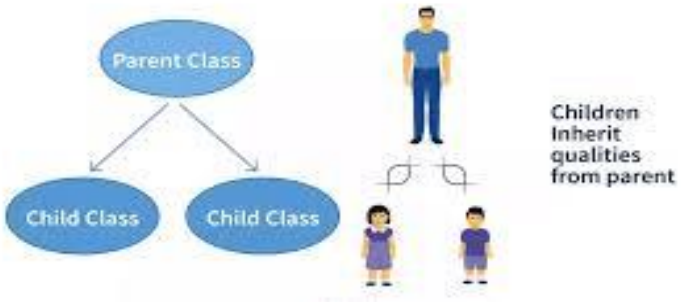
Sample Output:

```
Parent1 Method
Parent2 Method
Child2 Method
```

If the same method is inherited from both parent classes, then Python will always consider the order of Parent classes in the declaration of the child class.

- **class C(P1, P2):** P1 method will be considered
- **class C(P2, P1):** P2 method will be considered

Hierarchical Inheritance: The concept of inheriting properties from one class into multiple classes which are present at same level is known as Hierarchical Inheritance

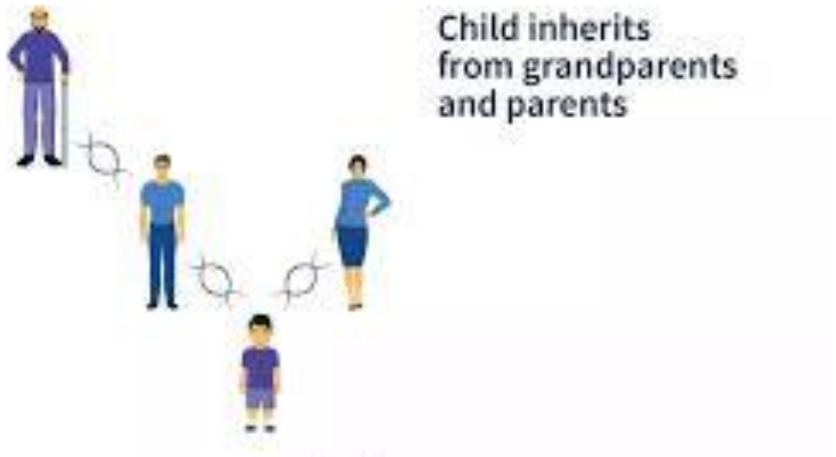



```
class one:
    def display(self):
        self.x=1000;
        self.y=2000;
        print("This is the method in class one");
        print("Value of X= ",self.x);
        print("Value of Y= ",self.y);

class two (one):
    def add(self):
        print("This is the method in class two");
        print("X+Y= ",(self.x+self.y));

class three(one):
    def mul(self):
        print("This is the method in class three");
        print("X*Y= ",(self.x * self.y));

t1=two();
t2=three();
t1.display();
t2.display();
t1.add();
t2.mul();
```

Output-

```
This is the method in class one
Value of X= 1000
Value of Y= 2000
This is the method in class one
Value of X= 1000
Value of Y= 2000
This is the method in class two
X+Y= 3000
This is the method in class three
X*Y= 2000000
```

Hybrid Inheritance: Combination of Single, Multi level, multiple and Hierarchicalinheritance is known as Hybrid Inheritance.



```
class stud:
    def setstud(self,sno,sname):
        self.sno = sno;
        self.sname = sname;

    def putstud(self):
        print("Student No : " ,self.sno);
        print("Student Name : ",self.sname);

class marks(stud):
    def setmarks(self, m1,m2):
        self.mark1 = m1;
        self.mark2 = m2;

    def putmarks(self):
        print("Mark1 : " , self.mark1);
        print("Mark2 : " , self.mark2);

class pratical:
    def getpractial(self,p1):
        self.p1=p1;

    def putpractial(self):
        print("Practial mark=",self.p1);


class result(marks,pratical):
    def calc(self):
        self.total = self.mark1 + self.mark2+self.p1;

    def puttotal(self):
        print("Total : " , self.total);

r =result();
r.setstud(60,"Ash");
r.setmarks(50,60);
r.getpractial(100);
r.calc();
r.putstud();
r.putmarks();
r.putpractial()
r.puttotal();
```

Output-

```
Student No : 60
Student Name : Ash
Mark1 : 50
Mark2 : 60
Practial mark= 100

Total : 210
```

POLYMORPHISM

- poly means many. Morphs means forms.
- Polymorphism means 'Many Forms'.

Eg1: Yourself is best example of polymorphism.In front of Your parents You will have one type of behaviour and with friends another type of behaviour.Same person but different behaviours at different places,which is nothing but polymorphism.

Eg2: + operator acts as concatenation and arithmetic addition

Eg3: * operator acts as multiplication and repetition operator

Eg4: The Same method with different implementations in Parent class and child classes.(overriding)

Polymorphism can be carried out through inheritance, with sub classes making use of base class methods or overriding them.

Polymorphism is classified into two types.

- ☑ Method Overloading
- ☑ 2. Method Overriding

Overloading:

Overloading occurs when two or more methods in one class have the same method name but different parameters.

Example :

```
class A:
 def m1(self,a):
 print(a)
 def m1(self,a,b):
 print(a + b)
 def m1(self, name , age , a):
```

```
 print('Name is :' , name)
 print('Age is :' , age)
 print(a)
obj = A()
obj.m1('Milky',20,10)
```

Output:

```
Name is : Milky
Age is : 20
10
```

Python Overriding :

Overriding means having two methods with the same method name and same parameters (i.e., method signature). One of the method is in the Parent class and the other method is in the Child class.

Syntax:

```
class Person:
 def read(self):
 pass
class Employee(Person):
 def read(self):
 pass
```

Example 1:

```
class Person:
 def read(self):
 print('Person class read() method')
class Employee(Person):
 def read(self):
 super().read()
```

```
print('Employee class read() method')
e = Employee()
e.read()
```

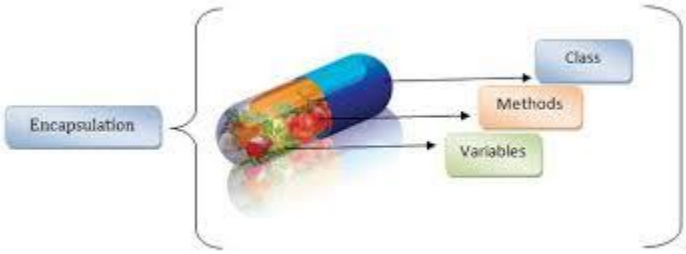
Output:

```
Person class read() method
Employee class
```

Python Encapsulation

The wrapping up of data and function together, into a single unit is called as encapsulation.

Data + Function ----->> Class



- This feature keeps the data safe from outside interface and misuse.
- This led to a concept of data hiding.
- In an object oriented python program, you can restrict access to methods and variables.
- This can prevent the data from being modified by accident and is known as encapsulation.
- Encapsulation restricted accesss to methods or variables. read() method
- Compared to languages like Java that offer access modifiers (public or private) for variables and methods, Python provides access to all the variables and methods globally

Example:

```
class Person:
 def __init__(self, name, age=0):
 self.name = name
```

```
 self.age = age
 def display(self):
 print(self.name)
 print(self.age)
person = Person('MILKY', 30)
#accessing using class method
person.display()
#accessing directly from outside
print(person.name)
print(person.age)
```

Output:

```
Milky
30
Milky
30
```

Benefits of Encapsulation in Python:

- Encapsulation not only ensures better data flow but also protects the data from outside sources.
- The concept of encapsulation makes the code self-sufficient.
- It is very helpful in the implementation level, as it prioritizes the 'how' type questions, leaving behind the complexities.
- You should hide the data in the unit to make encapsulation easy and also to secure the data.

Access Modifiers in Python : Public, Private and Protected

Access modifiers are used by object oriented programming languages like C++,java,python etc. to restrict the access of the class member variable and methods from outside the class. Python supports three types of access modifiers which are public,private and protected. These access

modifiers provide restrictions on the access of member variables and methods of the class from any object outside the class.

Public Access Modifier

By default the member variables and methods are public which means they can be accessed from anywhere outside or inside the class. No public keyword is required to make the class or methods and properties public.

```
class Student:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def display(self):
    print("Name:", self.name)
    print("Age:", self.age)
s = Student("Tillu", 11)
s.display()
```

Sample Output:

```
Name: Tillu
Age: 11
```

Private Access Modifier

Class properties and methods with private access modifier can only be accessed within the class where they are defined and cannot be accessed outside the class. In Python private properties and methods are declared by adding a prefix with two underscores('__') before their declaration.

```
class BankAccount:
  def __init__(self, account_number, balance):
    self.__account_number = account_number
    self.__balance = balance
```

```
  def __display_balance(self):
    print("Balance:", self.__balance)
b = BankAccount(1234567890, 9000)
b.__display_balance()
```

Sample Output:

```
AttributeError: 'BankAccount' object has no attribute '__display_balance'
```

Protected Access Modifier

Class properties and methods with protected access modifier can be accessed within the class and from the class that inherits the protected class. In python, protected members and methods are declared using single underscore('_') as prefix before their names.

```
class Person:
  def __init__(self, name, age):
    self._name = name
    self._age = age
  def _display(self):
    print("Name:", self._name)
    print("Age:", self._age)
class Student(Person):
  def __init__(self, name, age, roll_number):
    super().__init__(name, age)
    self._roll_number = roll_number
  def display(self):
    self._display()
    print("Roll Number:", self._roll_number)
s = Student("Milky", 19, 101)
s.display()
```

Sample Output:

Name: Milky

Age: 19

Roll Number: 101

SUPER() FUNCTION

The super() function is a built-in function in Python that provides a convenient way to access and delegate methods and attributes of parent classes. When used, it allows one class to access the methods and properties of another class in the same hierarchy. It is commonly used to avoid redundant code and make it more organized and easier to maintain.

Syntax of super() function

When using the Python super() function, you must include the self-argument in your method. This tells Python that you are referring to the current class, not another class in the hierarchy

```
class ParentClass():
   def __init__(self):
   self.name = "ParentClass"
class ChildClass(ParentClass):
   def __init__(self):

   super().__init__()

self.name = "ChildClass"
```

In the example, we create two classes — ParentClass and ChildClass — and set them up in an inheritance relationship. Then, inside the ChildClass's init method, we use the super() function to call the init method of its parent class, thus allowing us to access its properties and methods. The Python super() function can accept two parameters:

The first parameter is the type of class you are referring to — ParentClass in our example above — and The second parameter is the self-argument of the class you are referring to.

```
class Parent:
    def __init__(self, name):
    self.name = name
def greet(self):
    print(f "Hello, {self.name}!")
class Child(Parent):
    def __init__(self, name, age):
    super().__init__(name)
self.age = age
child = Child("Tillu", 10)
child.greet()
```

1. **Define a class named as Circle. Use a class variable to define the value of constant PI. Use this class variable to calculate area and circumference of a circle with specified radius?**

```
class Circle():
    def __init__(self, r):
        self.radius = r
    def area(self):
        return self.radius**2*3.14
    def perimeter(self):
        return 2*self.radius*3.14
NewCircle = Circle(8)
print(NewCircle.area())
print(NewCircle.perimeter())
```

SAMPLE OUTPUT:

```
200.96
50.24
```