



NARSIMHA REDDY ENGINEERING COLLEGE

(UGC AUTONOMOUS)

**PYTHON PROGRAMMING
(CY2105PC)**

(CSE – CS)

**Prepared by
Mr. P Mabu Hussain
Assistant Professor**

UNIT-1

INTRODUCTION TO PYTHON

What is Python

Python is a general purpose, dynamic, and interpreted programming language. It supports Object Oriented programming approach to develop applications. It is simple and easy to learn and provides lots of high-level data structures.



About Author

- Python is an interpreted scripting language also. *Guido Van Rossum* is known as the founder of Python programming.



WHY PYTHON

- Python is a simple, general purpose, high level, and object-oriented programming language.
- It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code.

Why learn Python->

- Python provides many useful features to the programmer. These features make it most popular and widely used language.
 1. Easy to use and Learn
 2. Expressive Language
 3. Interpreted Language
 4. Object-Oriented Language
 5. Open Source Language
 6. Extensible
 7. Learn Standard Library
 8. GUI Programming Support
 9. Integrated
 10. Embeddable
 11. Dynamic Memory Allocation
 12. Wide Range of Libraries and Frameworks

Where is Python used->

- Python is a general-purpose, popular programming language and it is used in almost every technical field. The various areas of Python use are given below.
1. Data Science
 2. Data Mining
 3. Desktop Applications
 4. Console-based Applications
 5. Mobile Applications
 6. Software Development
 7. Artificial Intelligence
 8. Web Applications
 9. Enterprise Applications
 10. 3D CAD Applications
 11. Machine Learning
 12. Computer Vision or Image Processing Applications.
 13. Speech Recognitions

Python Popular Frameworks and Libraries

- **Web development (Server-side)** - Django Flask, Pyramid, CherryPy
- **GUIs based applications** - Tk, PyGTK, PyQt, PyJs, etc.
- **Machine Learning** - TensorFlow, PyTorch, **Scikit-learn**, Matplotlib, Scipy, etc.
- **Mathematics** - Numpy, Pandas, etc.

Who uses Python ?



log2base2.com

Applications of Python

- **Easy-to-learn** – Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read** – Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain** – Python's source code is fairly easy-to-maintain.
- **A broad standard library** – Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode** – Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.

Contd...

- **Portable** – Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable** – You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases** – Python provides interfaces to all major commercial databases.
- **GUI Programming** – Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.

Development of Python



Python's Growth !!

- According to the Google trends, Python is one of the most developing Language.

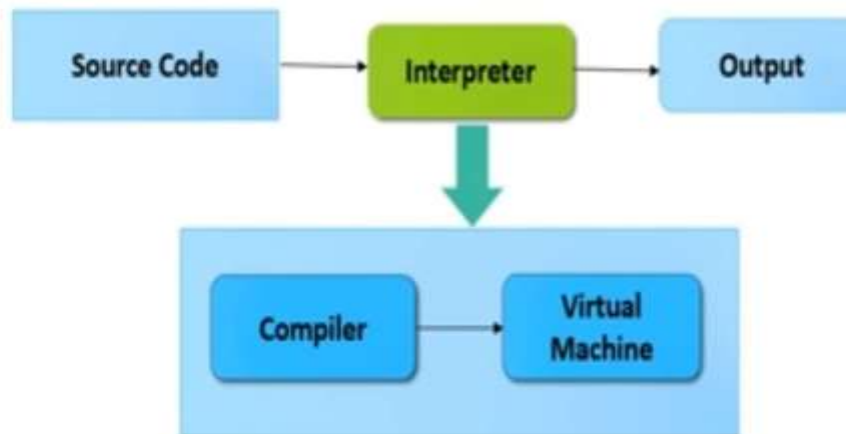


COMPILER & INTERPRETER

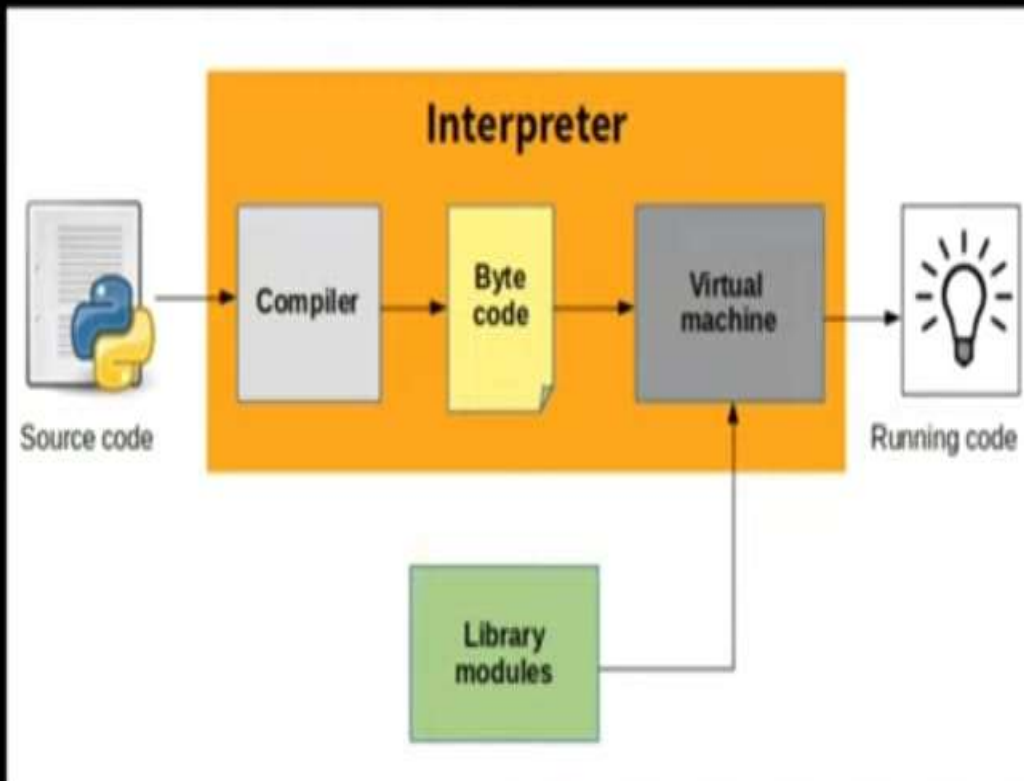
COMPILER



INTERPRETER



How Python Works ?



Byte Code Instructions

| | |
|--|--|
| <pre>import dis def print5(): print(5) dis.dis(print5)</pre> | <pre>RESTART: C:/Users/Arjit Gupta/AppData/Local/Programs/Python/Python39-64/Python.exe 3 0 LOAD_GLOBAL 0 (print) 2 LOAD_CONST 1 (5) 4 CALL_FUNCTION 1 6 POP_TOP 8 LOAD_CONST 0 (None) 10 RETURN_VALUE</pre> |
|--|--|

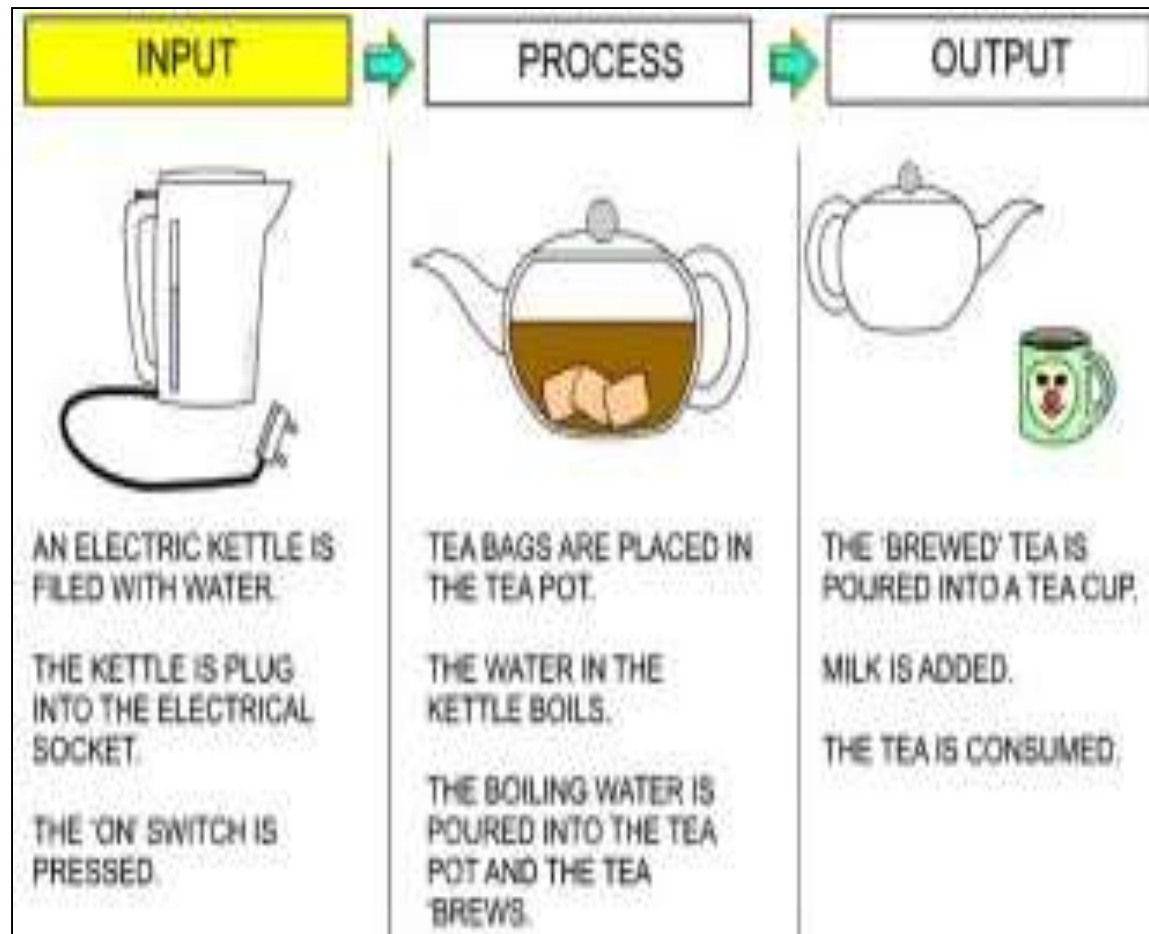
Source Code
in Human Readable Form

Byte Code Instructions

PYTHON SOFTWARES



INPUT, PROCESSING, AND OUTPUT



Displaying Output with the Print Function

- **Python print() function** : It prints the message to the screen or any other standard output device.
- **Syntax:** `print(value(s), sep= ' ', end = '\n', file=file, flush=flush)`

Parameters:

- **value(s)** : Any value, and as many as you like. Will be converted to string before printed
- **sep='separator'** : (Optional) Specify how to separate the objects, if there is more than one. Default : ' '
- **end='end'** : (Optional) Specify what to print at the end. Default : '\n'
- **file** : (Optional) An object with a write method. Default : `sys.stdout`
- **flush** : (Optional) A Boolean, specifying if the output is flushed (True) or buffered (False). Default: False
- **Returns:** It returns output to the screen.

Example program

- `print "Welcome to INDIA"` missing parentheses
- `print("USA")`
- `print (8 * "\n")`
- `print ("Welcome to", end = ' ')`
- `print ("hello", end = '!')`
- `print('can do this work',5)`
- `print('cannot do this work:'+5)`
- **Note:** You cannot use the "+" to join strings with int or float, you must use the ","

Reading Input from the Keyboard

- To get input from the user you can use the `input` function.
- When the input function is called the program stops running the program, prompts the user to enter something at the keyboard by printing a string called the prompt to the screen, and then waits for the user to press the Enter key. The user types a string of characters and presses enter. Then the input function returns that string and Python continues running the program by executing the next statement after the input statement.
- Python provides the function `input()`. `input` has an optional parameter, which is the prompt string.

Comments

Comments in Python start with the hash character, # , and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character.

- #This is a comment
- #print out Hello

Multi-line comments

- Example :

```
"""This is also a  
perfect example of  
multi-line comments"""
```

Variables

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

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

Assigning Values to Variables

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

- A variable can have a short name (like x and y) or a more descriptive name (age, name, total_volume). Rules for Python variables:
 - A variable name must start with a letter or the underscore character
 - A variable name cannot start with a number
 - A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
 - Variable names are case-sensitive (age, Age and AGE are three different variables)
-
- The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

Multiple Assignments to variables

Python allows you to assign a single value to **several variables simultaneously**.

For example :

a = b = c = 1

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

For example :

➤ **a, b, c = 1, 2.5, "ravi"**

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

Reserved Words

- Reserved words (also called keywords) are defined with predefined meaning and syntax in the language. These keywords have to be used to develop programming instructions.
- Python 3 has 33 keywords while Python 2 has 30. The `print` has been removed from Python 2 as keyword and included as built-in function.
- To check the keyword list, type following commands in interpreter .
- **`>>> import keyword >>> keyword.kwlist`**

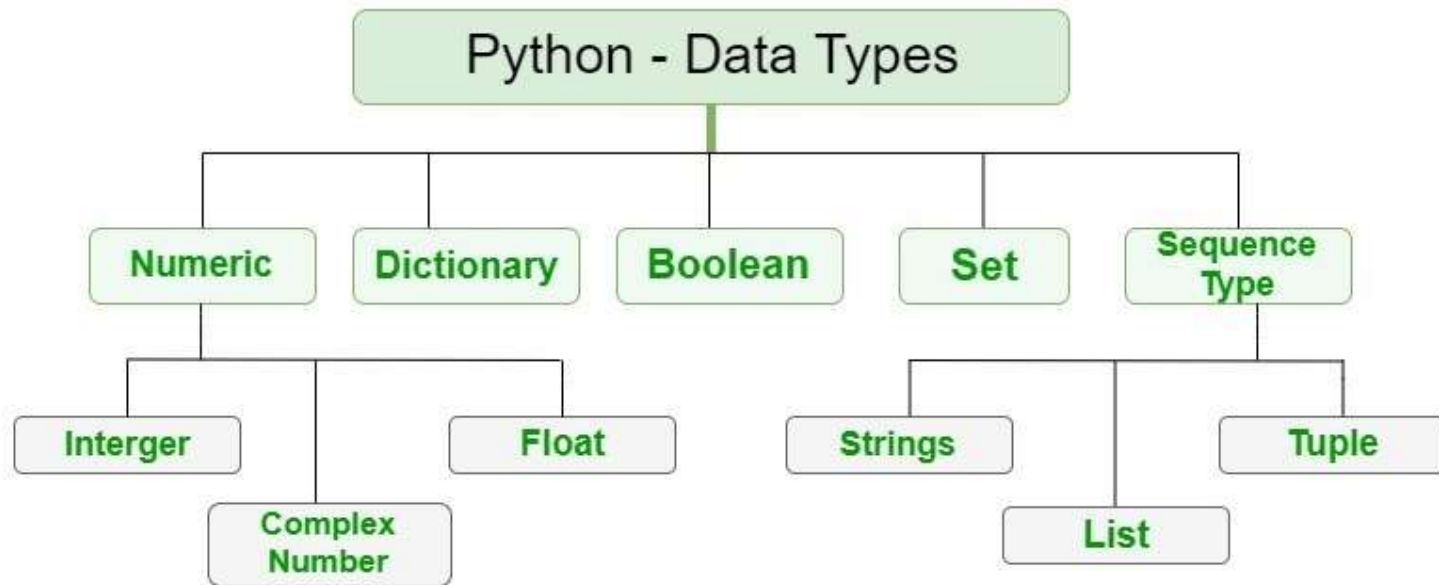
```
>>> import keyword >>> keyword.kwlist
```

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

DATA TYPES

- Data Type represent the type of data present inside a variable. In Python we are not required to specify the type explicitly.
- Based on value provided, the type will be assigned automatically.
- Hence Python is Dynamically Typed Language.

DATA TYPES



1.Numeric: In Python, numeric data type represent the data which has numeric value. Numeric value can be integer, floating number or even complex numbers. These values are defined as int, float and complex class in Python.

2.Integers : This value is represented by int class. It contains positive or negative whole numbers (without fraction or decimal). In Python there is no limit to how long an integer value can be.

3.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.

4.Complex Numbers : Complex number is represented by complex class. It is specified as *(real part) + (imaginary part)j*.

For example – $2+3j$.

❑ **type() function** is used to determine the type of data type.

Example:

#Python program to demonstrate numeric value

```
a = 5
```

```
print("Type of a: ", type(a))
```

```
b = 5.0
```

```
print("\n Type of b: ", type(b))
```

```
c = 2 + 4j
```

```
print("\n Type of c: ", type(c))
```

OUTPUT:

```
Type of a: <class 'int'>
```

```
Type of b: <class 'float'>
```

```
Type of c: <class 'complex'>
```


Python contains the following **in-built** data types

1. int
2. float
3. complex
4. bool
5. str
6. bytes
7. bytearray
8. range
9. list
10. Tuple
11. set
12. frozenset
13. dictionary
14. None

- Python contains several in-built functions
 1. `type()` to check the type of variable.
 2. `id()` to get address of object.
 3. `print()` to print the value.

❑ In Python everything is object

1.Int data type

int data type: We can use int data type to represent whole numbers (integral values)

Eg: `a=10`, `type(a)`, `#int`.

We can represent int values in the following ways

1. Decimal form
2. Binary form
3. Octal form
4. Hexa decimal form

1. Decimal form(base-10):

It is the default number system in Python.

The allowed digits are: 0 to 9

Eg: `a = 10`

2. Binary form(Base-2):

The allowed digits are : 0 & 1

Literal value should be prefixed with `0b` or `0B`

Eg: `a = 0B1111` , `a = 0B123` , `a = b111`

3. Octal Form(Base-8):

The allowed digits are : 0 to 7

Literal value should be prefixed with `0o` or `0O`

Eg: `a = 0o123` , `a = 0o786`

4. Hexa Decimal Form(Base-16):

The allowed digits are : 0 to 9, a-f (both lower and upper cases are allowed)

Literal value should be prefixed with 0x or 0X

Eg: a = 0XFACE

a = 0XBeel

a = 0XBee

- ❑ we can specify literal values in decimal, binary, octal and hexa decimal forms.
- ❑ But **PVM** will always provide values only in decimal form.

Base Conversions

- *Python provide the following in-built functions for base conversions*

1.bin(): We can use bin() to convert from any base to binary

- Eg:
 - 1) >>> bin(15)
'0b1111'
 - 2) >>> bin(0o11)
'0b1001'
 - 3) >>> bin(0X10)
'0b10000'

2. oct(): We can use oct() to convert from any base to octal

Example:

1) >>> oct(10)

'0o12'

2) >>> oct(0B1111)

'0o17'

3) >>> oct(0X123)

'0o443'

3. hex(): We can use hex() to convert from any base to hexadecimal

Example :

1) >>> hex(100)

'0x64'

2) >>> hex(0B111111)

'0x3f'

3) >>> hex(0o12345)

'0x14e5'

2.float data type

- We can use float data type to represent floating point values (decimal values)

Eg: f=1.234
 type(f) float

We can also represent floating point values by using exponential form (scientific notation).

Eg: f=1.2e3
 print(f) 1200.0
 instead of 'e' we can use 'E'

3.Complex Data Type

a and b contain integers or floating point values.

Eg: $3+5j$

$10+5.5j$

$0.5+0.1j$

In the **real part** if we use int value then we can specify that either by decimal,octal,binary or hexa decimal form.

But **imaginary part** should be specified only by using decimal form.

4.bool data type

- We can use this data type to represent boolean values.
 - The only allowed values for this data type are: **True and False.**
- ☐ Internally Python represents True as **1** and False as **0**.

```
b=True  
type(b) ==>bool
```

Example:

```
a=10  b=20  c=a<b  
print(c)==>True
```

```
True True+True==>2
```

```
True-False==>1
```

```
False+False==>0
```

5.str type

str represents String data type.

A String is a sequence of characters enclosed within single quotes or double quotes.

```
s1='python'
```

```
s1=" python"
```

❑ We can also use triple quotes to use single quote or double quote in our String.

```
"""This "Python class very helpful" for java learners"""
```

Creating String

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

Example Program 1:

Creating a String with single Quotes

- `String1 = 'Welcome to the CMREC'`
- `print("String with the use of Single Quotes: ")`
- `print(String1)`

Output:

```
String with the use of Single Quotes:  
Welcome to the CMREC
```

Example Program 2:

Creating a String with double Quotes

- `String1 = "I'm a CMRECIAN"`
- `print("\n String with the use of Double Quotes: ")`
- `print(String1)`
- `print(type(String1))`

Output:

```
String with the use of Double Quotes:  
I'm a CMRECIAN  
<class 'str'>
```

Accessing elements of String

In Python, individual characters of a String can be accessed by using the method of Indexing. Indexing allows negative address references to access characters from the back of the String.

- e.g. -1 refers to the last character, -2 refers to the second last character and so on.

| 0 | 1 | 2 | 3 | 4 |
|----|----|----|----|----|
| H | E | L | L | O |
| -5 | -4 | -3 | -2 | -1 |

Example Program

- **#python Program to Access characters of String**
- `String1 = "venkat"`
- `print("Initial String: ")`
- `print(String1)`
-
- `# Printing First character`
- `print("\n First character of String is: ")`
- `print(String1[0])`
-
- `# Printing Last character`
- `print("\nLast character of String is: ")`
- `print(String1[-1])`

Output:

Initial String:

venkat

First character of String is:

v

Last character of String is:

t

6.bytes Data Type

bytes data type represents a group of byte numbers just like an array.

Eg:

```
x = [10,20,30,40]
```

```
b = bytes(x)
```

```
type(b)==>bytes
```

```
print(b[0])==> 10
```

```
print(b[-1])==> 40
```

7.bytearray Data type

bytearray is exactly same as bytes data type except that its elements can be modified.

Eg: 1

```
x=[10,20,30,40]  
b = bytearray(x)  
for i in b : print(i)
```

output:

```
10  
20  
30  
40
```

Eg:2

```
b[0]=100  
for i in b: print(i)
```

Output:

```
100  
20  
30  
40
```


8.List data type

- ❖ List is a sequence of data values called **items** or **elements**. An item can be of any type.
- ❖ Lists are just like the arrays, declared in other languages which is a ordered collection of data.
- ❖ It is very flexible as the items in a list do not need to be of the same type.

Creating List

Lists in Python can be created by just placing the sequence inside the square brackets [].

- ❖ If we want to represent a group of values as a single entity where insertion order required
 - ❖ To preserve and duplicates are allowed then we should go for list data type.
1. insertion order is preserved
 2. heterogeneous objects are allowed
 3. duplicates are allowed
 4. Growable in nature
 5. values should be enclosed within square brackets.

Example:

```
list=[10,10.5,'venkat',True,10]  
print(list)
```

Output:[10,10.5,'venkat',True,10]

Example:

```
list=[10,20,30,40]
```

```
>>> list[0]
```

10

```
>>> list[-1]
```

40

```
>>> list[1:3]
```

[20, 30]

❖ list is growable in nature. i.e based on our requirement we can increase or decrease the size.

- **# Python program to demonstrate Creation of List**

- # Creating a List
- List = []
- print("Initial blank List: ")
- print(List)
-
- # Creating a List with the use of a String
- List = ['cmrec']
- print("\n List with the use of String: ")
- print(List)

- **Output:**

- Initial blank List:
- []
- List with the use of String:
- ['cmrec']

- **# Creating a List with the use of multiple values**

- List = ["Ram", "Laxman", "Bharath"]
- print("\nList containing multiple values: ")
- print(List[0])
- print(List[2])

- **Output:**

- List containing multiple values:
- Ram
- Bharath

- **# Creating a Multi-Dimensional List** (By Nesting a list inside a List)
- You can also use other lists as elements in a list, there by creating a list o lists.

Example:

```
>>>List = [['swapna', 'rani'], ['ganaji']]
>>>print("\nMulti-Dimensional List: ")
>>>print(List)
```

Output:

- Multi-Dimensional List:
- [['swapna', 'rani'], ['ganaji']]

Accessing elements of List

- In order to access the list items refer to the index number. Use the index operator `[]` to access an item in a list.
- In Python, negative sequence indexes represent positions from the end of the array. Instead of having to compute the offset as in `List[len(List)-3]`, it is enough to just write `List[-3]`.
- Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second-last item, etc.

Example Program

- `# Python program to demonstrate accessing of element from list`
- `# Creating a List with the use of multiple values`
- `List = ["AIML", "CSE", "DS"]`
- `# accessing a element from the list using index number`
- `print("Accessing element from the list")`
- `print(List[0])`
- `print(List[2])`

Output:

- Accessing element from the list
- AIML
- DS

9.Tuple data type

- ❖ A tuple is a type of sequence that resembles a list, except that, unlike a list, a tuple is immutable.
- ❖ Tuple data type is exactly same as list data type except that it is immutable. (i.e we cannot change values).
- ❖ Tuple elements can be represented within parenthesis.

Eg:

1) `t=(10,20,30,40)`

2) `type(t)`

3) `<class 'tuple'>`

Creating Tuple

- In Python, [tuples](#) are created by placing a sequence of values separated by 'comma' with or without the use of parentheses for grouping of the data sequence. Tuples can contain any number of elements and of any datatype (like strings, integers, list, etc.).
- **Note:** Tuples can also be created with a single element, but it is a bit tricky. Having one element in the parentheses is not sufficient, there must be a trailing 'comma' to make it a tuple.
- **Example:**
- **# Creating an empty tuple**
- `Tuple1 = ()`
- `print("Initial empty Tuple: ")`
- `print (Tuple1)`
- **Output:**
- `Initial empty Tuple:`
- `()`

- **# Creating a Tuple with the use of Strings**
- `Tuple1 = ('AIML', 'DS')`
- `print("\nTuple with the use of String: ")`
- `print(Tuple1)`

Output:

- `Tuple with the use of String:`
- `('AIML', 'DS')`

- **# Creating a Tuple with the use of list**
- `list1 = [1, 2, 4, 5, 6]`
- `print("\nTuple using List: ")`
- `print(tuple(list1))`

Output:

- `Tuple using List:`
- `(1, 2, 4, 5, 6)`

- **# Creating a Tuple with the use of built-in function**

- Tuple1 = tuple('swapna')
- print("\nTuple with the use of function: ")
- print(Tuple1)

Output:

- Tuple with the use of function:
- ('s', 'w', 'a', 'p', 'n', 'a')

- **# Creating a Tuple with nested tuples**

- Tuple1 = (0, 1, 2, 3)
- Tuple2 = ('swapna', 'rani')
- Tuple3 = (Tuple1, Tuple2)
- print("\nTuple with nested tuples: ")
- print(Tuple3)

Output:

- Tuple with nested tuples:
- ((0, 1, 2, 3), ('swapna', 'rani'))

Note : Creation of Python tuple without the use of parentheses is known as Tuple Packing.

Example:

```
>>>fruits = ("apple", "banana")
```

```
>>>fruits
```

```
Output:('apple', 'banana')
```

```
>>>L1=[1,2,3,4,5]
```

```
>>>L1
```

```
Output:[1, 2, 3, 4, 5]
```

```
>>>print(tuple(L1))
```

```
Output:(1, 2, 3, 4, 5)
```

```
>>>meats=("fish","prawns")
```

```
>>>meats
```

```
Output:('fish', 'prawns')
```

```
>>>food=fruits+meats
```

```
>>>food
```

```
Output:('apple', 'banana', 'fish', 'prawns')
```

10.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.

❖ Example:

Form-1: range(5)

- ❖ generate numbers from 0 to 9

```
>>>r=range(5)
```

```
>>>for i in r:print(i)
```

Output:

0

1

2

3

4

5

- ❖ We can access elements present in the range Data Type by using index.
- ❖ We cannot modify the values of range data type.
- ❖ We can create a list of values with range data type.

11.set Data Type

- In Python, set is an unordered collection of data type that is iterable, mutable and has no duplicate elements.
- The order of elements in a set is undefined though it may consist of various elements.

Creating Sets

- Sets can be created by using the built-in set() function with an iterable object or a sequence by placing the sequence inside curly braces, separated by 'comma'.
- Type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.

If we want to represent a group of values without duplicates where order is not important then we should go for set Data Type.

1. insertion order is not preserved
2. duplicates are not allowed
3. heterogeneous objects are allowed
4. index concept is not applicable
5. It is mutable collection
6. Growable in nature

Python program to demonstrate Creation of Set in Python

- **# Creating a Set**

```
>>>set1 = set()
>>>print("Intial blank Set: ")
>>>print(set1)
```

Output:

```
Intial blank Set:
set()
```

- **# Creating a Set with the use of a String**

```
>>>set1 = set("CMREC")
>>>print("\n Set with the use of String: ")
>>>print(set1)
```

output:

```
Set with the use of String:
{'C', 'M', 'R', 'E', 'C'}
```


- **# Creating a Set with the use of a List**

```
>>>set1 = set(["CSE", "AIML", "DS", "CSE"])
>>>print("\nSet with the use of List: ")
>>>print(set1)
```

OUTPUT:

```
Set with the use of List:
{'CSE', 'AIML', 'DS'}
```

- **# Creating a Set with a mixed type of values (Having numbers and strings)**

```
>>>set1 = set([1, 2, 'AIML', 4, 'DS', 6, 'AIML'])
>>>print("\nSet with the use of Mixed Values")
>>>print(set1)
```

OUTPUT:

```
Set with the use of Mixed Values
{1, 2, 4, 6, 'AIML', 'DS'}
```

Accessing elements of Sets

- Set items cannot be accessed by referring to an index, since sets are unordered the items has no index.
 - But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.
 - **#Python program to demonstrate Accessing of elements in a set**
 - # Creating a set
- ```
>>>set1 = set(["CSE", "AIML", "CSE"])
>>>print("\nInitial set")
>>>print(set1)
```
- OUTPUT:**
- ```
Initial set:
{'CSE', 'AIML', 'CSE'}
```

- **Adding Elements to a Set**

- Elements can be added to the Set by using built-in **add()** function. Only one element at a time can be added to the set by using add() method, loops are used to add multiple elements at a time with the use of add() method.

- **Using update() method**

- For addition of two or more elements Update() method is used. The update() method accepts lists, strings, tuples as well as other sets as its arguments. In all of these cases, duplicate elements are avoided.

- **Removing elements from the Set**

-

- Using remove() method or discard() method

Elements can be removed from the Set by using built-in remove() function but a KeyError arises if element doesn't exist in the set. To remove elements from a set without KeyError, use discard(), if the element doesn't exist in the set, it remains unchanged

Example Program

```
• set1 = set()
• print("Initial blank Set: ")
• print(set1)
• # Adding element and tuple to the Set
• set1.add(8)
• set1.add(9)
• set1.add((6,7))
• print("\nSet after Addition of Three elements: ")
• print(set1)
• # Adding elements to the Set using Iterator
• for i in range(1, 6):
•     set1.add(i)
• print("\nSet after Addition of elements from 1-5: ")
• print(set1)
•
• # Addition of elements to the Set using Update function
• set1.update([10, 11])
• print("\nSet after Addition of elements using Update: ")
• print(set1)
• # Accessing element using for loop
• print("\nElements of set: ")
• for i in set1:
•     print(i, end=" ")
• set1 = set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
• # Removing elements from Set using Remove() method
• set1.remove(8)
• set1.remove(9)
• print("\nSet after Removal of two elements: ")
• print(set1)
• # Removing elements from Set using Discard() method
• set1.discard(10)
• set1.discard(13)
• print("\nSet after Discarding two elements: ")
• print(set1)
• # Removing elements from Set using iterator method
• for i in range(1, 5):
•     set1.remove(i)
• print("\nSet after Removing a range of elements: ")
• print(set1)
```

OUTPUT:

Initial blank Set:

set()

Set after Addition of Three
elements:

{8, 9, (6, 7)}

Set after Addition of elements from
1-5:

{1, 2, 3, (6, 7), 4, 5, 8, 9}

Set after Addition of elements
using Update:

{1, 2, 3, (6, 7), 4, 5, 8, 9, 10,
11}

Elements of set:

1 2 3 (6, 7) 4 5 8 9 10 11

Set after Removal of two elements:

{1, 2, 3, 4, 5, 6, 7, 10, 11, 12}

Set after Discarding two elements:

{1, 2, 3, 4, 5, 6, 7, 11, 12}

Set after Removing a range of
elements:

{5, 6, 7, 11, 12}

12.frozenset Data Type

- ❖ It is exactly same as set except that it is immutable.
- ❖ Hence we cannot use add or remove functions.

Eg:

```
>>> s={10,20,30,40}
```

```
>>> fs=frozenset(s)
```

```
>>> fs
```

output:

```
frozenset({40, 10, 20, 30})
```

```
>>> type(fs)
```

output:

```
<class 'frozenset'> >>> fs
```

13.Dictionary Data Type

- ❖ A Dictionary organizes information by **association**, not position.
- ❖ In python, a dictionary associates set of **keys** with **values**.
- ❖ A dictionary is written as a sequence of key/value pairs separated by commas.
- ❖ These pairs are sometimes called **entries**.
- ❖ The entire sequence of entries is enclosed in curly braces({and}).
- ❖ A colon(:) separates a key and its value.

Example:

```
d={101:'raju',102:'ravi',103:'shiva'}
```

- ❖ Duplicate keys are not allowed but values can be duplicated.
- ❖ If we are trying to insert an entry with duplicate key then old value will be replaced with new value.
- ❖ **Note:** dictionary is mutable and the order wont be preserved

Example program

```
# Creating an empty Dictionary
>>>Dict = {}
>>>print("Empty Dictionary: ")
>>>print(Dict)

# Creating a Dictionary with Integer Keys
>>>Dict = {1: 'CSE', 2: 'DS', 3: 'AIML'}
>>>print("\nDictionary with the use of Integer
Keys: ")
>>>print(Dict)

# Creating a Dictionary with Mixed keys
>>>Dict = {'Name': 'CMREC', 1: [1, 2, 3, 4]}
>>>print("\nDictionary with the use of Mixed
Keys: ")
>>>print(Dict)

# Creating a Dictionary with dict() method
>>>Dict = dict({1: 'AIML', 2: 'CSE', 3: 'DS'})
>>>print("\nDictionary with the use of dict():
")
>>>print(Dict)

# Creating a Dictionary with each item as a
Pair
>>>Dict = dict([(1, 'KIRAN'), (2, 'BHAVYESH')])
>>>print("\nDictionary with each item as a pair:
")
>>>print(Dict)
```

Output:

```
Empty Dictionary:
{}


```

```
Dictionary with the use of
Integer Keys:
{1: 'CSE', 2: 'DS', 3:
'AIML'}
```

```
Dictionary with the use of
Mixed Keys:
{'Name': 'CMREC', 1: [1, 2,
3, 4]}
```

```
Dictionary with the use of
dict():
{1: 'AIML', 2: 'CSE', 3:
'DS'}
```

```
Dictionary with each item
as a pair:
{1: 'KIRAN', 2: 'BHAVYESH'}
```

Accessing elements of Dictionary

- In order to access the items of a dictionary refer to its key name. Key can be used inside square brackets.
- There is also a method called `get()` that will also help in accessing the element from a dictionary.

Python program to demonstrate accessing a element from a Dictionary

Creating a Dictionary

```
>>>Dict = {1: '526', 'name': 'swapna', 'address':  
'hyderabad'}
```

accessing a element using key

```
>>>print("Accessing a element using key:")
```

```
>>>print(Dict['name'])
```

accessing a element using `get()` method

```
>>>print("Accessing a element using get:")
```

```
>>>print(Dict.get(1))
```

Output:

- Accessing a element using key:
- swapna
- Accessing a element using get:
- 526

14.None Data Type

None means Nothing or No value associated.

If the value is not available, then to handle such type of cases None introduced.

It is something like null value in Java.

Eg:

```
def m1():
```

```
    a=10
```

```
print(m1())
```

```
None
```

Python Data Type Cheatsheet

| String | List | Tuple | Set | Dictionary |
|----------------------------------|--|--|---|--|
| Immutable | Mutable | Immutable | Mutable | Mutable |
| Ordered/Indexed | Ordered/Indexed | Ordered/Indexed | Unordered | Unordered |
| Allows Duplicate Members | Allows Duplicate Members | Allows Duplicate Members | Doesn't allow Duplicate Members | Doesn't allow Duplicate keys |
| Empty string = "" | Empty list = [] | Empty tuple = () | Empty set = set() | Empty dictionary = {} |
| String with single element = "H" | List with single item = ["Hello"] | Tuple with single item = ("Hello",) | Set with single item = {"Hello"} | Dictionary with single item = {"Hello": 1} |
| --- | It can store any data type str, list, set, tuple, int and dictionary | It can store any data type str, list, set, tuple, int and dictionary | It can store data types (int, str, tuple) but not (list, set, dictionary) | Inside of dictionary key can be int, str and tuple only values can be of any data type int, str, list, tuple, set and dictionary |

| Datatype | Description | Is Immutable | Example |
|-----------|--|--------------|---|
| Int | We can use to represent the whole/integral numbers | Immutable | <pre>>>> a=10 >>> type(a) <class 'int'></pre> |
| Float | We can use to represent the decimal/floating point numbers | Immutable | <pre>>>> b=10.5 >>> type(b) <class 'float'></pre> |
| Complex | We can use to represent the complex numbers | Immutable | <pre>>>> c=10+5j >>> type(c) <class 'complex'> >>> c.real 10.0 >>> c.imag 5.0</pre> |
| Bool | We can use to represent the logical values(Only allowed values are True and False) | Immutable | <pre>>>> flag=True >>> flag=False >>> type(flag) <class 'bool'></pre> |
| Str | To represent sequence of Characters | Immutable | <pre>>>> s='python' >>> type(s) <class 'str'> >>> s="python programming" >>> s='''cmrec''' >>> type(s) <class 'str'></pre> |
| bytes | To represent a sequence of byte values from 0-255 | Immutable | <pre>>>> list=[1,2,3,4] >>> b=bytes(list) >>> type(b) <class 'bytes'></pre> |
| bytearray | To represent a sequence of byte values from 0-255 | Mutable | <pre>>>> list=[10,20,30] >>> ba=bytearray(list) >>> type(ba)</pre> |

| | | | |
|-----------|--|-----------|---|
| | | | <class 'set'> |
| frozenset | To represent an unordered collection of unique objects | Immutable | <pre>>>> s={11,2,3,'Venkat',100,'Ramu'} >>> fs=frozenset(s) >>> type(fs) <class 'frozenset'></pre> |
| dict | To represent a group of keyvalue pairs | Mutable | <pre>>>> d={101:'venkat',102:'ramu',103:'hari'} >>> type(d) <class 'dict'></pre> |
| range | To represent a range of values | Immutable | <pre>>>> r=range(10) >>> r1=range(0,10) >>> r2=range(0,10,2)</pre> |
| list | To represent an ordered collection of objects | Mutable | <pre>>>> l=[10,11,12,13,14,15] >>> type(l) <class 'list'></pre> |
| tuple | To represent an ordered collections of objects | Immutable | <pre>>>> t=(1,2,3,4,5) >>> type(t) <class 'tuple'></pre> |
| set | To represent an unordered collection of unique objects | Mutable | <pre>>>> s={1,2,3,4,5,6} >>> type(s)</pre> |

| Data Type | Category | Mutable | Hashable |
|-------------------|-----------|---------|----------|
| Boolean (bool) | Condition | No | Yes |
| Integer (int) | Numeric | No | Yes |
| Float | Numeric | No | Yes |
| Complex | Numeric | No | Yes |
| List | Sequence | Yes | No |
| Tuple | Sequence | No | Yes |
| Range | Sequence | No | Yes |
| String (str) | Text | No | Yes |
| Set | Set | Yes | No |
| Frozenset | Set | No | Yes |
| Dictionary (dict) | Mapping | Yes | No |

Operators

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

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

Python Arithmetic Operators

- ❖ Python Arithmetic Operators: Arithmetic operators are used with numeric values to perform common mathematical operations.

Python Arithmetic Operators

| Operator | Name | Example |
|----------|----------------|----------|
| + | Addition | $x + y$ |
| - | Subtraction | $x - y$ |
| * | Multiplication | $x * y$ |
| / | Division | x / y |
| % | Modulus | $x \% y$ |
| ** | Exponentiation | $x ** y$ |
| // | Floor division | $x // y$ |

Program:

```
a=int(input('Enter a value:'))
b=int(input('Enter b value:'))
print('Addition=',a+b)
print('Subtraction=',a-b)
print('Multiplication=',a*b)
print('Floor division=',a//b)
print('Exponenciation=',a**b)
print('Division=',a/b)
print('Modulus=',a%b)
```

Output:

```
Enter a value:10
Enter b value:15
Addition= 25
Subtraction= -5
Multiplication= 150
Floor division= 0
Exponenciation= 10000000000000000
Division= 0.6666666666666666
Modulus= 10
```

Python Assignment Operators:

Assignment operators are used to assign values to variables

| Operator | Example | Same As |
|----------|----------------------------|-------------------------------|
| = | <code>x = 5</code> | <code>x = 5</code> |
| += | <code>x += 3</code> | <code>x = x + 3</code> |
| -= | <code>x -= 3</code> | <code>x = x - 3</code> |
| *= | <code>x *= 3</code> | <code>x = x * 3</code> |
| /= | <code>x /= 3</code> | <code>x = x / 3</code> |
| %= | <code>x %= 3</code> | <code>x = x % 3</code> |
| //= | <code>x //= 3</code> | <code>x = x // 3</code> |
| **= | <code>x **= 3</code> | <code>x = x ** 3</code> |
| &= | <code>x &= 3</code> | <code>x = x & 3</code> |
| = | <code>x = 3</code> | <code>x = x 3</code> |
| ^= | <code>x ^= 3</code> | <code>x = x ^ 3</code> |
| >>= | <code>x >>= 3</code> | <code>x = x >> 3</code> |
| <<= | <code>x <<= 3</code> | <code>x = x << 3</code> |

Python Comparison Operators:

Comparison operators are used to compare two values

| Operator | Name | Example |
|--------------------|--------------------------|------------------------|
| <code>==</code> | Equal | <code>x == y</code> |
| <code>!=</code> | Not equal | <code>x != y</code> |
| <code>></code> | Greater than | <code>x > y</code> |
| <code><</code> | Less than | <code>x < y</code> |
| <code>>=</code> | Greater than or equal to | <code>x >= y</code> |
| <code><=</code> | Less than or equal to | <code>x <= y</code> |

Program on Relational operators:

```
a=int(input('Enter a value:'))  
b=int(input('Enter b value:'))  
print("a > b is ",a>b)  
print("a >= b is ",a>=b)  
print("a < b is ",a<b)  
print("a <= b is ",a<=b)
```

Output:

```
Enter a value:10  
Enter b value:5  
a > b is True  
a >= b is True  
a < b is False  
a <= b is False
```

Python Logical Operators:

Logical operators are used to combine conditional statements

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

Program on Logical operators:

```
a=int(input('Enter a value:'))  
b=int(input('Enter b value:'))  
print("a or b is ",a or b)  
print("a and b is ",a and b)  
print("a not b is ", not a)
```

Output:

Enter a value:5

Enter b value:3

a or b is 5

a and b is 3

a not b is False

Python Identity Operators:

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

| Operator | Description | Example |
|----------|--|------------|
| is | Returns true if both variables are the same object | x is y |
| is not | Returns true if both variables are not the same object | x is not y |

Python Membership Operators:

Membership operators are used to test if a sequence is presented in an object:

| Operator | Description | Example |
|----------|--|------------|
| in | Returns True if a sequence with the specified value is present in the object | x in y |
| not in | Returns True if a sequence with the specified value is not present in the object | x not in y |

Python Bitwise Operators:

Bitwise operators are used to compare (binary) numbers

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

Program on Bitwise Operators:

```
a=int(input('Enter a value:'))
b=int(input('Enter b value:'))
print("a & b is ",a & b)
print("a | b is ",a | b)
print("~ a is ", ~ a)
print("a ^ b is ",a ^ b)
print("a << 2 is ",a << 2)
print("a >> 2 is ",a >> 2)
```

Output:

Enter a value:4

Enter b value:5

a & b is 4

a | b is 5

~ a is -5

a ^ b is 1

a << 2 is 16

a >> 2 is 1

Special operators

Python defines the following 2 special operators

1. Identity Operators
2. Membership operators

1. Identity Operators:

We can use identity operators for address comparison.

Two identity operators are available.

1. is
2. is not

`r1 is r2` returns True if both `r1` and `r2` are pointing to the same object

`r1 is not r2` returns True if both `r1` and `r2` are not pointing to the same object.

Example:

- 1) a=10
- 2) b=10
- 3) print(a is b) True
- 4) x=True
- 5) y=True
- 6) print(x is y) True

Example:

- 1) list1=["one","two","three"]
- 2) list2=["one","two","three"]
- 3) print(id(list1))
- 4) print(id(list2))
- 5) print(list1 is list2) False
- 6) print(list1 is not list2) True
- 7) print(list1 == list2) True

Note: We can use is operator for address comparison where as == operator for content comparison.

2. Membership operators

We can use Membership operators to check whether the given object present in the given collection.(It may be String,List,Set,Tuple or Dict).

Example:

- 1) `x="hello learning Python is very easy!!!"`
- 2) `print('h' in x)` True
- 3) `print('d' in x)` False
- 4) `print('d' not in x)` True
- 5) `print('Python' in x)` True

Operator Precedence

If multiple operators present then which operator will be evaluated first is decided by operator precedence.

Example:

`print(3+10*2) ->23`

`print((3+10)*2) -> 26`

The following list describes operator precedence in Python

- ✓ `()` ----> Parenthesis
- ✓ `**` ----> exponential operator
- ✓ `~, -` ----> Bitwise complement operator , unary minus operator
- ✓ `*, /, %, //` ----> multiplication, division, modulo, floor division
- ✓ `+, -` ----> addition , subtraction
- ✓ `<<, >>` ----> Left and Right Shift
- ✓ `&` ----> bitwise And
- ✓ `^` ----> Bitwise X-OR
- ✓ `|` ----> Bitwise OR
- ✓ `>, >=, <, <=, ==, !=` ==> Relational or Comparison operators
- ✓ `=, +=, -=, *= ...` ==> Assignment operators
- ✓ `is, is not` ----> Identity Operators
- ✓ `in, not in` ----> Membership operators
- ✓ `not` ----> Logical not
- ✓ `and` ----> Logical and
- ✓ `or` ----> Logical or

Example:

1) $a=30$

2) $b=20$

3) $c=10$

4) $d=5$

5) $\text{print}((a+b)*c/d) \text{ } 100.0$

6) $\text{print}((a+b)*(c/d)) \text{ } 100.0$

7) $\text{print}(a+(b*c)/d) \text{ } 70.0$

8) $3/2*4+3+(10/5)**3-2$

9) $3/2*4+3+2.0**3-2$

10) $3/2*4+3+8.0-2$

11) $1.5*4+3+8.0-2$

12) $6.0+3+8.0-2$

13) 15.0

Type conversions

- Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.
- For example, it is not possible to perform “2”+4 since one operand is integer and the other is string type. To perform this we have convert string to integer i.e., **int(“2”) + 4 = 6.**
- There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

Example: (int)

1) >>> int(123.987)

123

2) >>> int(10+5j)

TypeError: can't convert complex to int 5)

3) >>> int(True)

1

4) >>> int(False)

0

5) >>> int("10")

10

6) >>> int("10.5")

ValueError: invalid literal for int() with base 10: '10.5' 13)

7) >>> int("ten")

ValueError: invalid literal for int() with base 10: 'ten'

Example:(float)

1) >>> float(10)

10.0

2) >>> float(10+5j)

TypeError: can't convert complex to float

3) >>> float(True)

1.0

4) >>> float(False)

0.0

5) >>> float("10")

10.0 11)

6)>>> float("10.5")

10.5

Example:(complex)

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")`

ValueError: complex() arg is a malformed string

Example: **(Boolean)**

- 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`

Type conversions

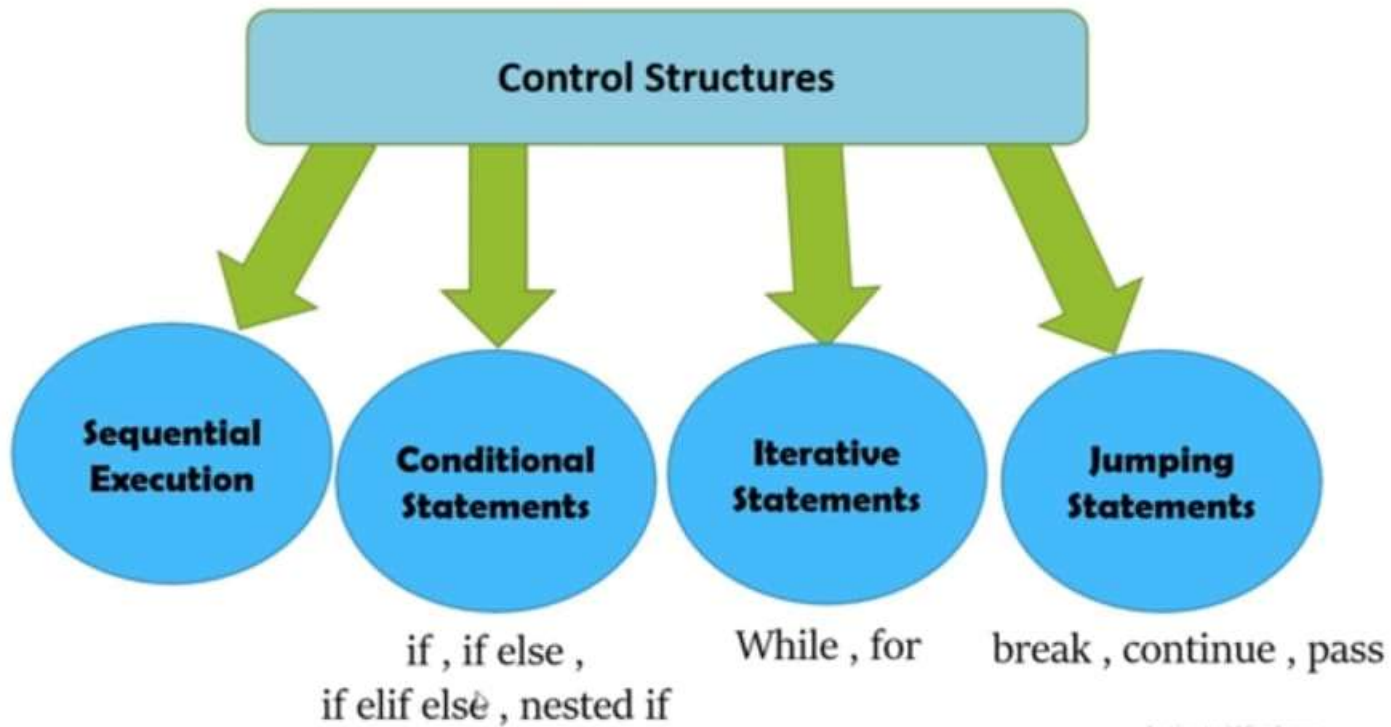
| Function | Description |
|------------------------------------|--|
| <code>int(x [,base])</code> | Converts x to an integer. |
| <code>long(x [,base])</code> | Converts x to a long integer. |
| <code>float(x)</code> | Converts x to a floating-point number. |
| <code>complex(real [.imag])</code> | Creates a complex number. |
| <code>str(x)</code> | Converts object x to a string representation. |
| <code>repr(x)</code> | Converts object x to an expression string. |
| <code>eval(str)</code> | Evaluates a string and returns an object. |
| <code>tuple(s)</code> | Converts s to a tuple. |
| <code>list(s)</code> | Converts s to a list. |
| <code>set(s)</code> | Converts s to a set. |
| <code>dict(d)</code> | Creates a dictionary, d must be a sequence of (key, value) tuples. |
| <code>frozenset(s)</code> | Converts s to a frozen set. |
| <code>chr(x)</code> | Converts an integer to a character. |
| <code>unichr(x)</code> | Converts an integer to a Unicode character. |
| <code>ord(x)</code> | Converts a single character to its integer value. |
| <code>hex(x)</code> | Converts an integer to a hexadecimal string. |
| <code>oct(x)</code> | Converts an integer to an octal string. |

Expressions

- An expression is a combination of variables constants and operators written according to the syntax of Python language.
- In Python every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable.
- Some examples of Python expressions are shown in the table given below.

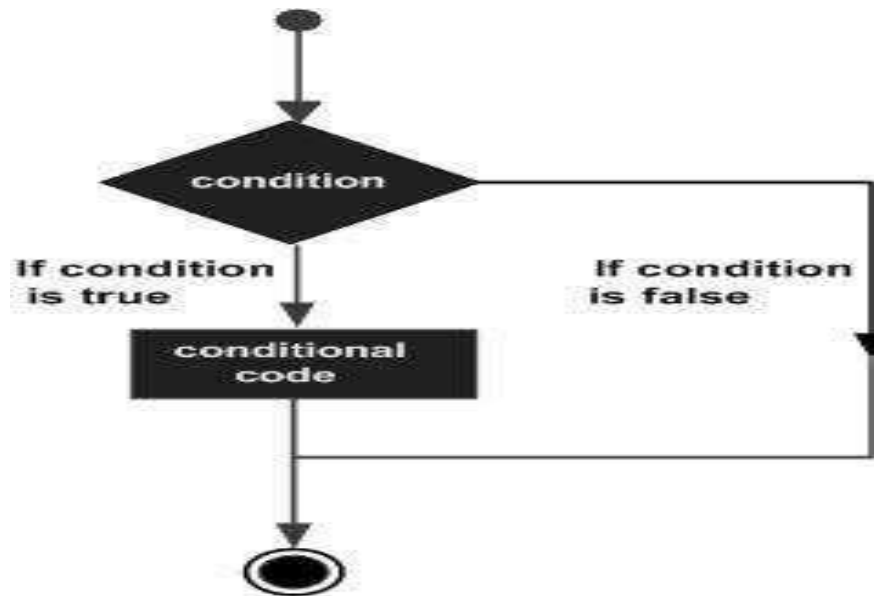
| Algebraic Expression | Python Expression |
|----------------------|--------------------------------|
| $a \times b - c$ | <code>a * b - c</code> |
| $(m + n) (x + y)$ | <code>(m + n) * (x + y)</code> |
| (ab / c) | <code>a * b / c</code> |
| $3x^2 + 2x + 1$ | <code>3*x*x+2*x+1</code> |
| $(x / y) + c$ | <code>x / y + c</code> |

Control structures



Decision Structures and Boolean Logic: if, if-else, if-elif-else Statements

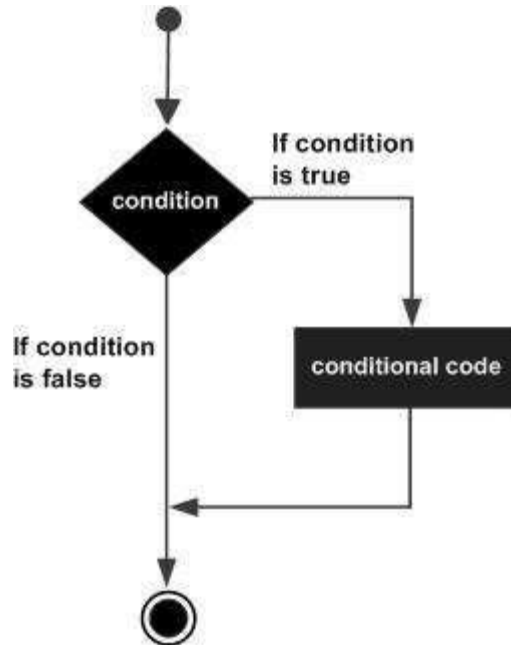
- Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.
- Decision structures evaluate multiple expressions which produce True or False as outcome. You need to determine which action to take and which statements to execute if outcome is True or False otherwise.



| Statement | Description |
|-----------------------------|--|
| if statements | if statement consists of a boolean expression followed by one or more statements. |
| if...else statements | if statement can be followed by an optional else statement, which executes when the boolean expression is FALSE. |
| nested if statements | You can use one if or else if statement inside another if or else if statement(s). |

If statement

- It is similar to that of other languages. The **if** statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.



- **Syntax:**

```
if condition:
    statements
```

- First, the condition is tested. If the condition is True, then the statements given after colon (:) are executed.
- We can write one or more statements after colon (:).

- **Example:**

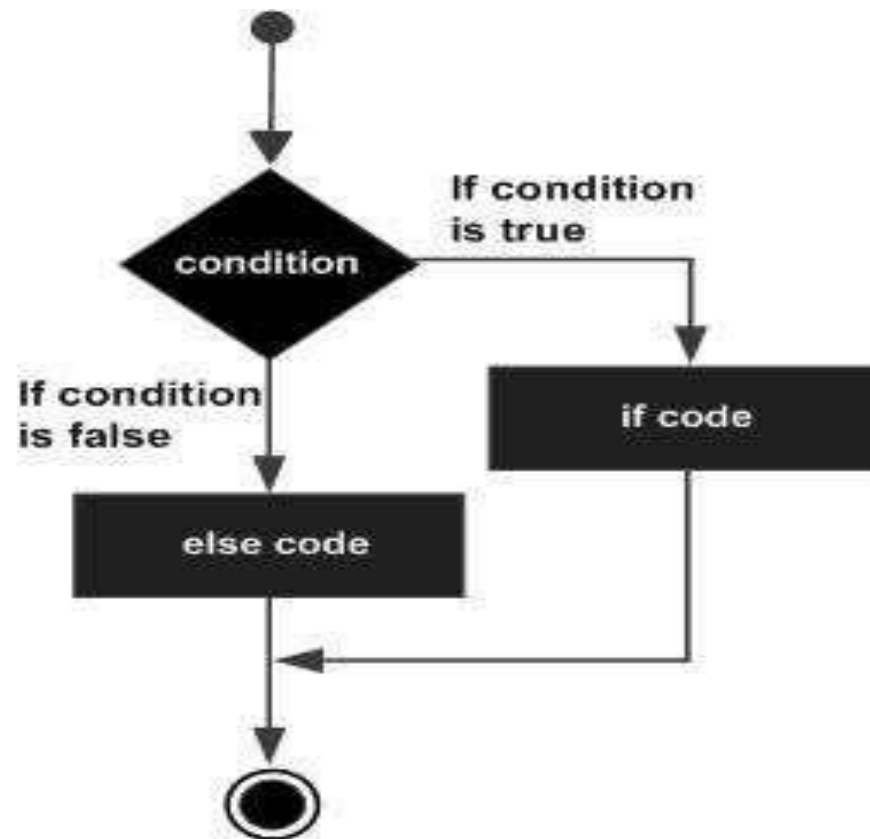
```
>>>a=10
>>>b=15
>>>if a < b:
    print ("a is lessthan b")
```

output:

a is lessthan b

The *if ... else* statement

- The **if-else statement** is the most common type of selection statement. It is also called as **two-way selection statement**, because it directs the computer to make a choice between two alternative courses of action.
- An **else** statement can be combined with an **if** statement. An **else** statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.
- The *else* statement is an optional statement and there could be at most only one **else** statement following **if**.
- **Syntax:**
 - if condition:
 statement(s)
 - else:
 statement(s)



Example:

```
>>>a=100
```

```
>>>b=200
```

```
>>>if(a<b):
```

```
    print("b is greater:",b)
```

```
>>>else:
```

```
    print("a is greater:",a)
```

Output:

- b is greater:200

Write a program for checking whether the given number is even or odd.

Program:

```
a= int(input("enter a value:"))  
if(a%2==0):  
    print(" a is even")  
else:  
    print("a is odd")
```

Output:

```
enter a value:56  
a is even
```

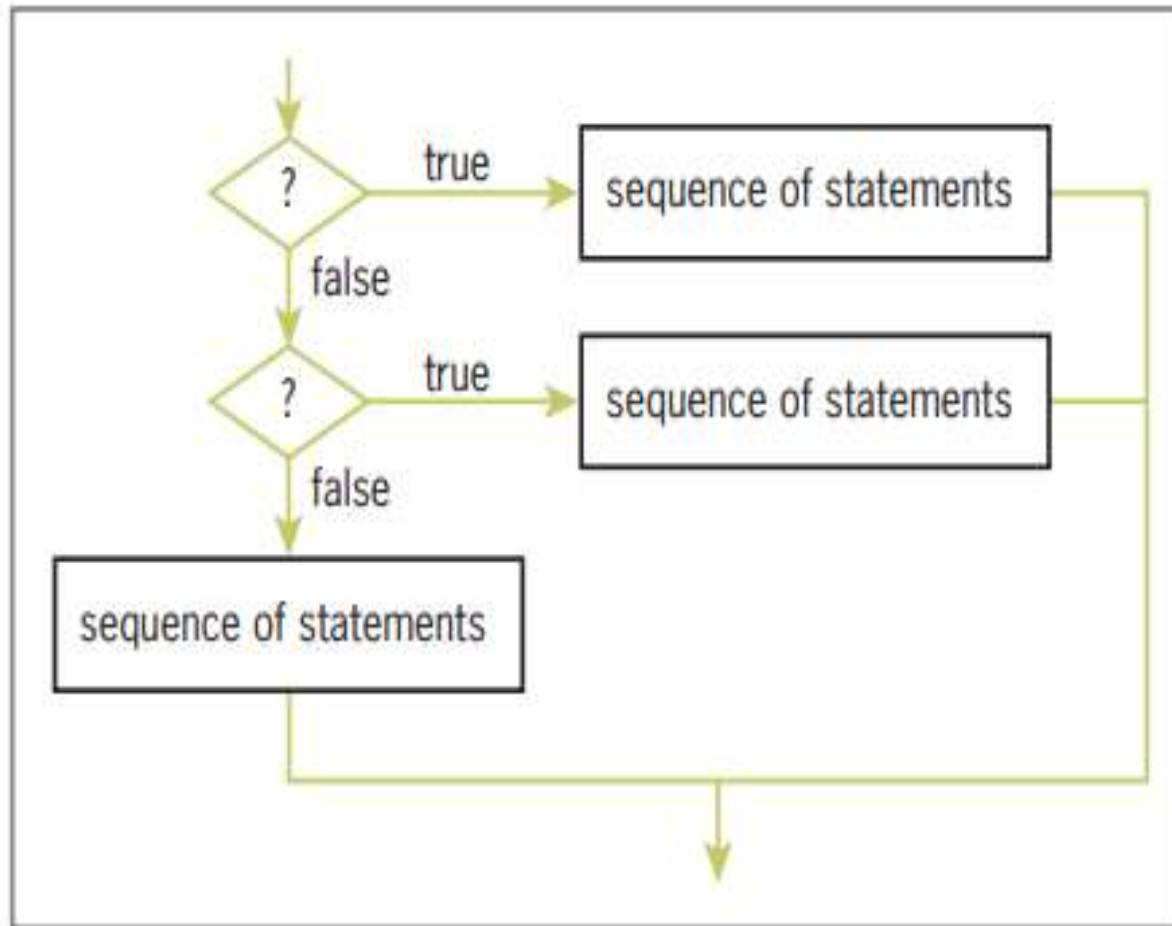

The *elif* Statement

- The process of testing several conditions and responding accordingly can be described in code by a **multi-way selection statement**.
- The multi-way **if** statement considers each condition until one evaluates to **False**.
- When a condition evaluates to **True**, the corresponding action is performed and control skips to the end of the entire selection statement.
- If no condition evaluates to **True**, then the action after the trailing **else** is performed.
- **Syntax:**

```
if condition1:
    statement(s)

elif condition2:
    statements(s)

else:
    statement(s)
```



Program:

```
a=int(input("Enter a value: "))
b=int(input("Enter b value: "))
c=int(input("Enter c value: "))
if((a>=b) and (a>=c)):
    print("a is greater")
elif((b>=a)and (b>=c)):
    print("b is big")
else:
    print("c is big")
```

Output:

```
Enter a value: 56
Enter b value: 23
Enter c value: 21
a is greater
```

Program:

```
Number=int(input("Enter the Number: "))
if(Number>99):
    print("letter=A")
elif(Number>89):
    print('letter=B')
elif(Number>79):
    print('letter=C')
else:
    print("The Number is:", Number)
```

Output:

```
1.Enter the Number: 100
letter=A
2.Enter the Number: 1
The Number is: 1
```

Nested IF statements

- There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested **if** construct.
- In a nested **if** construct, you can have an **if...elif...else** construct inside another **if...elif...else** construct.

- Syntax:

```
if expression1:  
    statement(s)  
    if expression2:  
        statement(s)  
    elif expression3:  
        statement(s)  
    elif expression4:  
        statement(s)  
    else: statement(s)
```

Program:

```
num = float(input("Enter a number: "))  
if num >= 0:  
    if num == 0:  
        print("Zero")  
    else:  
        print("Positive number")  
else:  
    print("Negative number")
```

Output:

```
1)Enter a number: 52  
    Positive number  
2)Enter a number: -9  
    Negative number
```

Program:

```
age = int(input(" Please Enter Your Age Here: "))  
if (age < 18):  
    print(" You are Minor ")  
    print(" You are not Eligible to Work ")  
else:  
    if (age >= 18 and age <= 60):  
        print(" You are Eligible to Work ")  
        print(" Please fill in your details and apply")  
    else:  
        print(" You are too old to work ")  
        print(" Please Collect your pension!")
```

Output:

```
Please Enter Your Age Here: 9  
You are Minor  
You are not Eligible to Work
```

for loop

- The *for* loop is useful to iterate over the elements of a sequence. It means, the *for* loop can be used to execute a group of statements repeatedly depending upon the number of elements in the sequence. The *for* loop can work with sequence like string, list, tuple, range etc.

The **syntax** of the for loop is given below:

for var **in** sequence:

statement (s)

- The first element of the sequence is assigned to the variable written after „for“ and then the statements are executed.
- The second element of the sequence is assigned to the variable and then the statements are executed second time.

- The first word of the statement starts with the **keyword “for”** which signifies the beginning of the for loop.
- Then we have the **iterator variable** which iterates over the sequence and can be used within the loop to perform various functions.
- The next is the **“in” keyword** in Python which tells the iterator variable to loop for elements within the sequence.
- And finally, we have the **sequence variable** which can either be a list, a tuple, or any other kind of iterator.
- The statements part of the loop is where you can play around with the iterator variable and perform various function.

Example 1:

```
fruits = ["apple", "banana", "cherry"]
```

```
for x in fruits:
```

```
    print(x)
```

Output:

apple

banana

Cherry

Example-2: Program to print the sum of the given list.

```
list = [10,30,23,43,65,12]
```

```
sum = 0
```

```
for i in list:
```

```
    sum = sum+i
```

```
print("The sum is:",sum)
```

Output:

The sum is: 10

The sum is: 40

The sum is: 63

The sum is: 106

The sum is: 171

The sum is: 183

For loop Using range() function

The range() function

This function returns a sequence of integers that we can use to determine how many iterations (repetitions) of the loop will be completed. The loop will complete one iteration per integer.

The range function has three parameters:

- **start:** where the sequence of integers will start. By default, it's 0.
- **stop:** where the sequence of integers will stop (without including this value).
- **step:** the value that will be added to each element to get the next element in the sequence. By default, it's 1.

This is the general syntax to write a for loop with range():

```
for <loop_variable> in range(<start>, <stop>, <step>):  
    <code>
```

- You can pass 1, 2, or 3 arguments to `range()`:
- With 1 argument, the value is assigned to the stop parameter and the default values for the other two parameters are used.
- With 2 arguments, the values are assigned to the start and stop parameters and the default value for step is used.
- With 3 arguments, the values are assigned to the start, stop, and step parameters (in order)

Range function

```
for var in range(m,n):  
    print var
```

The function **range(m,n)** returns the sequence of integers starting from m, m+1, m+2, m+3.....n-1.

Example with **one parameter**:

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

Output:

0

1

2

3

4

5

6

7

8

9

Example Program:

```
for i in range(10):  
    print(i*2)
```

Output:

```
0  
2  
4  
6  
8  
10  
12  
14  
16  
18
```

- In the example below, we repeat a string as many times as indicated by the value of the loop variable:

Program:

```
for num in range(8):  
    print("Hello" * num)
```

Output:

Hello

HelloHello

HelloHelloHello

HelloHelloHelloHello

HelloHelloHelloHelloHello

HelloHelloHelloHelloHelloHello

HelloHelloHelloHelloHelloHelloHello

Example with **Two** parameters:

```
for i in range(2,10):  
    print(i)
```

Output:

2

3

4

5

6

7

8

9

The expression **range(2,10)** creates an object known as an iterable .

This allows loop to assign the values 2,3,4,5,6,7,8 and 9 to the iteration variable i.

```
for i in range(2,6):  
    print("hello"*i)
```

Output:

hellohello

hellohellohello

hellohellohellohello

hellohellohellohellohello

The **len()** function is combined with **range()** function which iterate through a sequence using indexing. Consider the following example.

```
list = ['Peter','Joseph','Ricky','Devansh']  
for i in range(len(list)):  
    print("Hello",list[i])
```

Output:

Hello Peter

Hello Joseph

Hello Ricky

Hello Devansh

Example with **Three parameters**:

```
for i in range(2,25,2):  
    print(i)
```

Output:

```
2  
4  
6  
8  
10  
12  
14  
16  
18  
20  
22
```

Program to print even number using step size in range().

```
n = int(input("Enter the number "))  
for i in range(2,n,2):  
    print(i)
```

Output:

Enter the number 10

2

4

6

8

Nested for loop in python

Python allows us to nest any number of for loops inside a **for** loop. The inner loop is executed n number of times for every iteration of the outer loop. The syntax is given below.

Syntax

```
for iterating_var1 in sequence:    #outer loop
    for iterating_var2 in sequence: #inner loop
        #block of statements
#Other statements
```

Example- 1: Nested for loop

```
# User input for number of rows
rows = int(input("Enter the rows:"))
# Outer loop will print number of rows
for i in range(0,rows+1):
# Inner loop will print number of Astrisk
    for j in range(i):
        print("*",end = '')
    print()
```

Output:

Enter the rows:5

```
*
**
***
****
*****
```

Example-2: Program to number pyramid.

```
rows = int(input("Enter the rows"))  
    for i in range(0, rows+1):  
        for j in range(i):  
            print(i, end = ' ')  
        print()
```

Output:

1

22

333

4444

55555

While Loop

- The while loop is a loop control statement in Python and frequently used in programming for repeated execution of statement(s) in a loop.
- It executes a sequence of statements repeatedly as long as a condition remains true.

syntax :

```
while (test-condition):  
    sequence of statement(s)
```

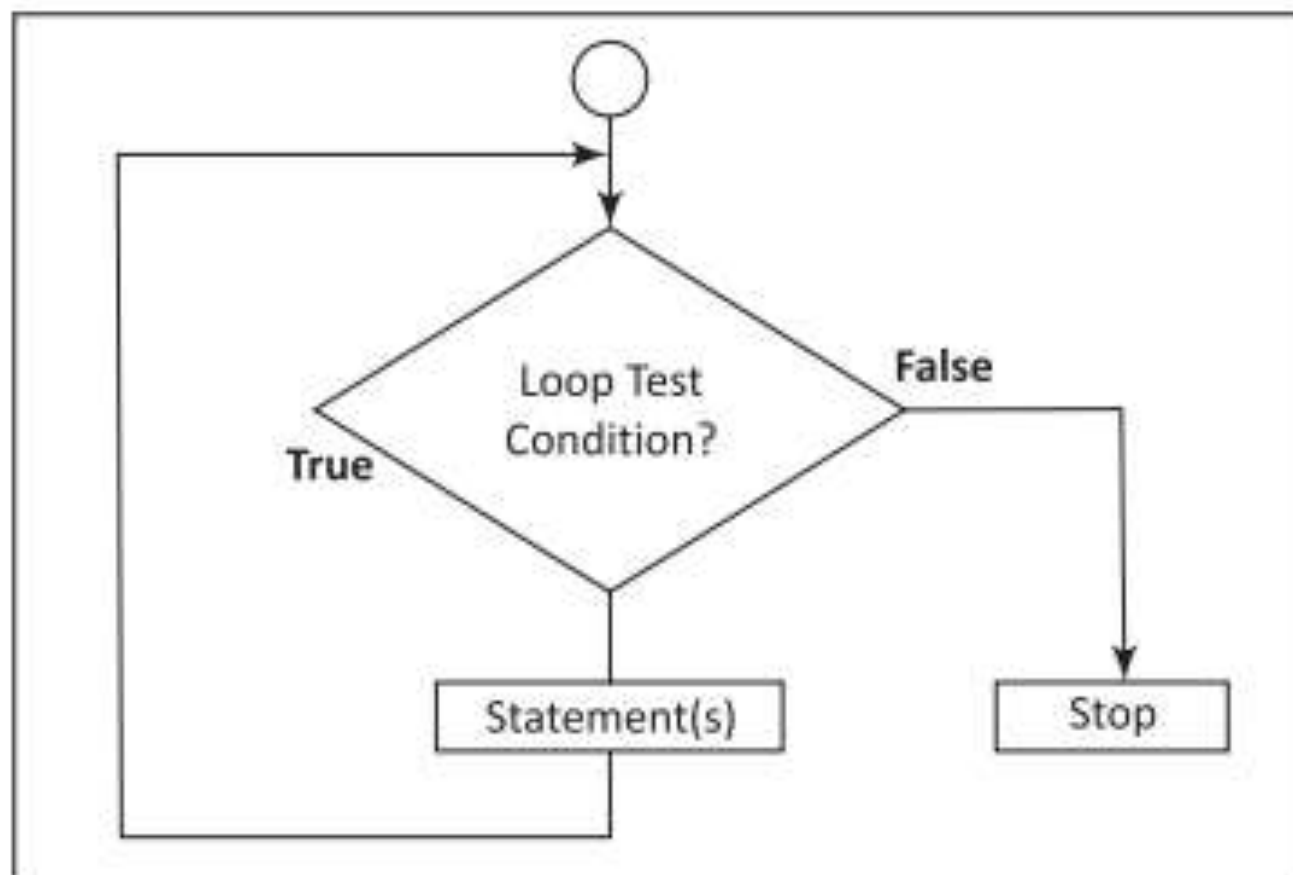



Figure 5.1 Flowchart of while loop

➤ Write a program to print the numbers from one to five using the while loop.

Program:

```
count=0
while count<=5:
    print("Count = ",count)
    count=count+1
```

Output:

```
Count = 0
Count = 1
Count = 2
Count = 3
Count = 4
Count = 5
```

Break statement

- The keyword allows a programmer to terminate a loop
- When the **break** statement is encountered inside a loop, the loop is immediately terminated and the program control automatically goes to the first statement following the loop.

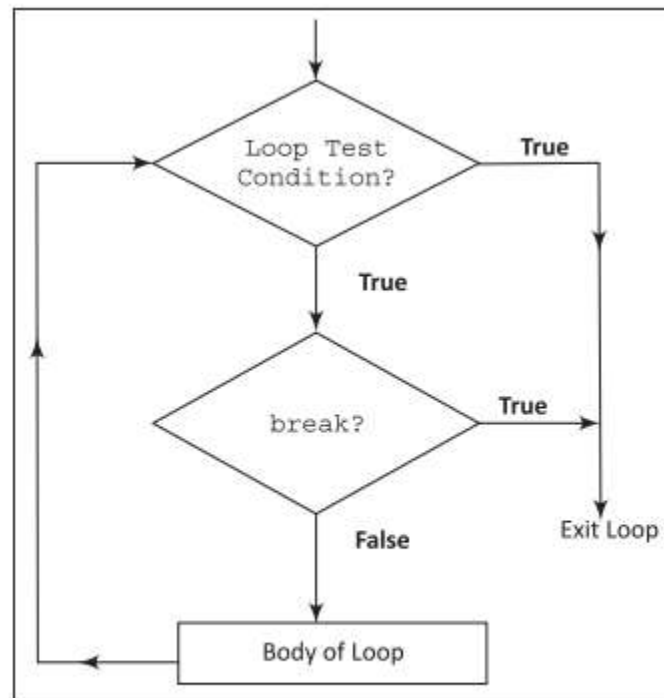
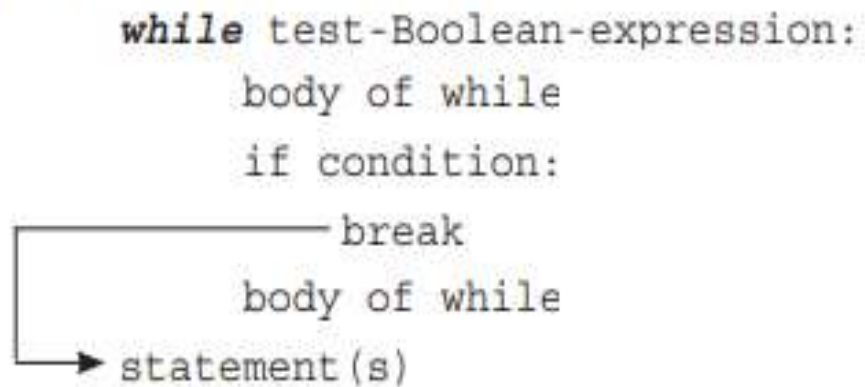
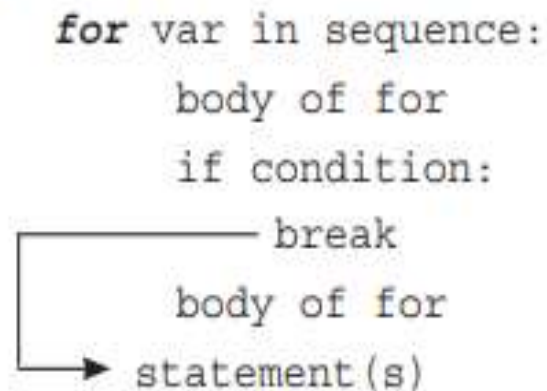


Figure 5.3 Flowchart for **break** statement

Working of break in while loop:



Working of break in for loop:



Write a program to demonstrate the use of the break statement

Program:

```
print("The Numbers from 1 to 10 are as follows: ")
for i in range(1,100,1):
    if(i==11):
        break
    else:
        print(i,end=" ")
```

Output:

The Numbers from 1 to 10 are as follows:
1 2 3 4 5 6 7 8 9 10

continue statement

- The continue statement is exactly opposite of the break statement.
- When continue is encountered within a loop, the remaining statements within the body are skipped but the loop condition is checked to see if the loop should continue or exit.

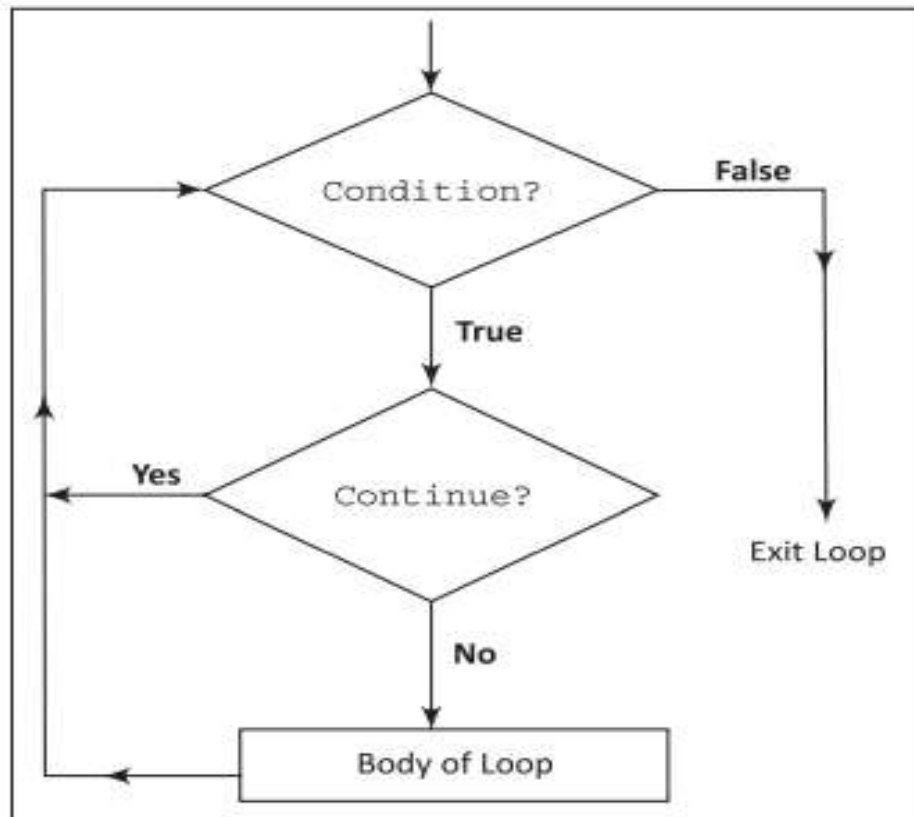



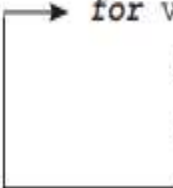
Figure 5.4 Flowchart for **continue** statement

The working of **continue** in while loop is shown as follows:



```
→ while test-boolean-expression:  
    body of while  
    if condition:  
        continue  
    body of while  
    statement(s)
```

Alternatively, the working of **continue** in for loop is shown as follows:



```
→ for var in sequence:  
    body of for  
    if condition:  
        continue  
    body of for  
    statement(s)
```

| <i>Break</i> | <i>Continue</i> |
|---------------------------------------|--|
| Exits from current block or loop. | Skips the current iteration and also skips the remaining statements within the body. |
| Control passes to the next statement. | Control passes at the beginning of the loop. |
| Terminates the loop. | Never terminates the loop. |

Program:

```
for i in range(1,11,1):  
    if i==5:  
        continue  
    print(i,end=" ")
```

Output:

1 2 3 4 6 7 8 9 10

Pass statement

- The pass statement does not do anything . It is used with 'if' statement or inside a loop to represent no operation.
- We use pass statement when we need a statement syntactically but we do not want to do any operation.

A program to know that the pass does nothing

```
X=0
```

```
While x<5:
```

```
    X+=1
```

```
    if X>3:
```

```
        pass
```

```
    print('X=',X)
```

```
Print ("out of the loop)
```

Output:

```
X=1
```

```
X=2
```

```
X=3
```

```
X=4
```

```
X=5
```

UNIT – II

Files and Exceptions

Files

When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order.

- **Open a file**
- **Read or write (perform operation)**
- **Close the file**

File Built-in Function [open ()]

Python has a built-in function `open()` to open a file. Which accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

Syntax:

```
Fileobject = open (file-name, access-mode)
```

file-name: It specifies the name of the file to be opened.

access-mode: There are following access modes for opening a file:

| access-mode | Description |
|-------------|---|
| "w" | Write - Opens a file for writing, creates the file if it does not exist. |
| "r" | Read - Default value. Opens a file for reading, error if the file does not exist. |
| "a" | Append - Opens a file for appending, creates the file if it does not exist. |
| "x" | Create - Creates the specified file, returns an error if the file exists. |
| "w+" | Open a file for updating (reading and writing), overwrite if the file exists. |
| "r+" | Open a file for updating (reading and writing), doesn't overwrite if the file exists. |

In addition you can specify if the file should be handled as binary or text mode

| access-mode | Description |
|-------------|--------------------------------------|
| "t" | Text - Default value. Text mode. |
| "b" | - Binary - Binary mode (e.g. images) |

Example:

```
fileptr = open("myfile.txt","r")
if fileptr:
    print("file is opened successfully with read mode only")
```

Output:

```
file is opened successfully with read mode only
```

Example:

```
fileptr = open("myfile1.txt","x")
if fileptr:
    print("new file was created successfully")
```

Output:

```
new file was created successfully
```

File Built-in Methods

Python supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. For this, python provides following built-in functions, those are

- `close()`
- `read()`
- `readline()`
- `write()`
- `writelines()`
- `tell()`
- `seek()`

close()

The close() method used to close the currently opened file, after which no more writing or Reading can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

Syntax:

```
Fileobject.close()
```

Example:

```
f = open("myfile.txt", "r")  
f.close()
```

read()

The read () method is used to read the content from file. To read a file in Python, we must open the file in reading mode.

Syntax: `Fileobject.read([size])`

Where 'size' specifies number of bytes to be read

readline()

Python facilitates us to read the file line by line by using a function readline(). The readline() method reads the lines of the file from the beginning, i.e., if we use the readline() method two times, then we can get the first two lines of the file.

Syntax: `Fileobject.readline()`

write()

The write () method is used to write the content into file. To write some text to a file, we need to open the file using the open method with one of the following access modes.

w: It will overwrite the file if any file exists. The file pointer point at the beginning of the file in this mode.

a: It will append the existing file. The file pointer point at the end of the file.

*Syntax:*Fileobject.write(content)

writelines()

The writelines () method is used to write multiple lines of content into file. To write some lines to a file

*Syntax:*Fileobject.writelines(list)

list – This is the Sequence of the strings.

write()

The write () method is used to write the content into file. To write some text to a file, we need to open the file using the open method with one of the following access modes.

w: It will overwrite the file if any file exists. The file pointer point at the beginning of the file in this mode.

a: It will append the existing file. The file pointer point at the end of the file.

*Syntax:*Fileobject.write(content)

writelines()

The writelines () method is used to write multiple lines of content into file. To write some lines to a file

*Syntax:*Fileobject.writelines(list)

list – This is the Sequence of the strings.

File Positions

Methods that set or modify the current position within the file

tell()

The tell() method returns the current file position in a file stream. You can change the current file position with the seek() method.

Syntax: Fileobject.tell()

seek()

The seek() method sets and returns the current file position in a file stream.

Syntax: Fileobject.seek(offset)

File Built-in Attributes

Python Supports following built-in attributes, those are
file.name - returns the name of the file which is already opened.

file.mode - returns the access mode of opened file.

file.closed - returns true, if the file closed, otherwise false.

Command-Line Arguments in Python

Till now, we have taken input in python using `raw_input()` or `input()`. There is another method that uses command line arguments. The command line arguments must be given whenever we want to give the input before the start of the script, while on the other hand, `input()` is used to get the input while the python program / script is running.

The `sys` module also provides access to any command-line arguments via `sys.argv`. Command-line arguments are those arguments given to the program in addition to the script name on invocation.

`sys.argv` is the list of command-line arguments

`len(sys.argv)` is the number of command-line arguments.

To use `argv`, you will first have to import it (`import sys`) The first argument, `sys.argv[0]`, is always the name of the program as it was invoked, and `sys.argv[1]` is the first argument you pass to the program. It's common that you slice the list to access the actual command line arguments.

File System in Python

In python, the file system contains the files and directories. To handle these files and directories python supports “os” module. Python has the “os” module, which provides us with many useful methods to work with directories (and files as well).

The os module provides us the methods that are involved in file processing operations and directory processing like renaming, deleting, get current directory, changing directory etc.

Renaming the file - rename()

The os module provides us the rename() method which is used to rename the specified file to a new name.

Syntax:

```
os.rename (“current-name”, “new-name”)
```

Example:

```
]import os;
```

```
#rename file2.txt to file3.txt
```

```
os.rename("file2.txt","file3.txt")
```

Removing the file – remove()

The os module provides us the remove() method which is used to remove the specified file.

Syntax:

```
os.remove("file-name")
```

Example:

```
import os;  
#deleting the file named file3.txt  
os.remove("file3.txt")
```

Creating the new directory – mkdir()

The mkdir() method is used to create the directories in the current working directory.

Syntax:

```
os.mkdir("directory-name")
```

Example:

```
import os;  
#creating a new directory with the name new  
os.mkdir("dirnew")
```

Changing the current working directory – chdir()

The `chdir()` method is used to change the current working directory to a specified directory.

Syntax:

```
os.chdir("new-directory")
```

Example:

```
import os;  
#changing the current working directory to new  
os.chdir("dir2")
```

Get current working directory – getcwd()

This method returns the current working directory.

Syntax:

```
os.getcwd()
```

Example:

```
import os;  
#printing the current working directory  
print(os.getcwd())
```


Deleting directory - rmdir()

The rmdir() method is used to delete the specified directory.

Syntax:

```
os.rmdir("directory name")
```

Example:

```
import os;  
#removing the new directory os.rmdir("dir2")
```

List Directories and Files – listdir()

All files and sub directories inside a directory can be known using the listdir() method. This method takes in a path and returns a list of sub directories and files in that path. If no path is specified, it returns from the current working directory.

Syntax:

```
os.listdir(["path"])
```

Example:

```
import os;  
#list of files and directories in current working directory  
print(os.listdir()) #list of files and directories in specified  
path print(os.listdir("D:\\\\"))
```

Exceptions

An exception can be defined as an abnormal condition in a program. It interrupts the flow of the program. Whenever an exception occurs, the program halts the execution, and thus the further code is not executed.

In general an exception is an error that happens during execution of a program. When that error occurs, it terminates program execution. In Python, an error can be a syntax error or an exception. Now we will see what an exception is and how it differs from a syntax error.

Syntax errors occur when the parser detects an incorrect statement.

expdemo.py

a=5

b=0

print(a/b))

Output:

File "expdemo.py", line 3 print(a/b))

SyntaxError: invalid syntax

Standard Exceptions in Python

Python supports various built-in exceptions, the commonly used exceptions are

- **NameError:** It occurs when a name is not found.i.e attempt to access an undeclared variable

Example:

```
a=5
```

```
c=a+b
```

```
print("Sum =",c)
```

Output:

Traceback (most recent call last):

File "expdemo.py", line 2, in c=a+b

NameError: name 'b' is not defined

ZeroDivisionError: Occurs when a number is divided by zero.

Example:

```
a=5
```

```
b=0
```

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

Output:

Traceback (most recent call last):

File "expdemo.py", line 3, in print(a/b)

ZeroDivisionError: division by zero

ValueError: Occurs when an inappropriate value assigned to variable.

Example:

```
a=int(input("Enter a number : ")) b=int(input("Enter  
a number : ")) print("Sum =",a+b)
```

Output:

Enter a number : 23

Enter a number : abc

Traceback (most recent call last): File "expdemo.py",
line 2, in

```
b=int(input("Enter a number : ")) ValueError: invalid  
literal for int() with base 10: 'abc'
```

IndexError: Occurs when we request for an out-of-range index for sequence

Example:

```
ls=['c','java','python']  
print("list item is :",ls[5])
```

Output:

```
Traceback (most recent call last): File "expdemo.py",  
line 2, in
```

```
print("list item is :",ls[5])
```

```
IndexError: list index out of range.
```


KeyError: Occurs when we request for a non-existent dictionary key

Example:

```
dic={"name":"Madhu","location":"Hyd"}  
print("The age is :",dic["age"])
```

Output:

Traceback (most recent call last):

File "expdemo.py", line 2, in

```
print("The age is :",dic["age"])
```

KeyError: 'age'

KeyError: Occurs when we request for a non-existent dictionary key

Example:

```
dic={"name":"Madhu","location":"Hyd"}  
print("The age is :",dic["age"])
```

Output:

Traceback (most recent call last):

File "expdemo.py", line 2, in

```
print("The age is :",dic["age"])
```

KeyError: 'age'

IOError: Occurs when we request for a non-existent input/output file.

Example:

```
fn=open("exam.py")  
print(fn)
```

Output:

Traceback (most recent call last):

File "expdemo.py", line 1, in

```
fn=open("exam.py")
```

FileNotFoundError: [IOError] No such file or directory: 'exam.py'

Exceptions as Strings

Prior to Python 1.5, standard exceptions were implemented as strings. However, this became limiting in that it did not allow for exceptions to have relationships to each other. As of python 1.5, all standard exceptions are now classes. It is still possible for programmers to generate their own exceptions as strings, but we recommend using exception classes from now on.

Python 2.5 begins the process of deprecating string exceptions from Python forever. In 2.5, raise of string exceptions generates a warning. In 2.6, the catching of string exceptions results in a warning. Since they are rarely used and are being deprecated, we will no longer consider string exceptions.

Exceptions as Strings

Prior to Python 1.5, standard exceptions were implemented as strings. However, this became limiting in that it did not allow for exceptions to have relationships to each other. As of python 1.5, all standard exceptions are now classes. It is still possible for programmers to generate their own exceptions as strings, but we recommend using exception classes from now on.

Python 2.5 begins the process of deprecating string exceptions from Python forever. In 2.5, raise of string exceptions generates a warning. In 2.6, the catching of string exceptions results in a warning. Since they are rarely used and are being deprecated, we will no longer consider string exceptions.

Exceptions as Strings

Prior to Python 1.5, standard exceptions were implemented as strings. However, this became limiting in that it did not allow for exceptions to have relationships to each other. As of python 1.5, all standard exceptions are now classes. It is still possible for programmers to generate their own exceptions as strings, but we recommend using exception classes from now on.

Python 2.5 begins the process of deprecating string exceptions from Python forever. In 2.5, raise of string exceptions generates a warning. In 2.6, the catching of string exceptions results in a warning. Since they are rarely used and are being deprecated, we will no longer consider string exceptions.

Exception handling in Python

Python provides us with the way to handle the Exception so that the other part of the code can be executed without any interrupt.

For Exception handling, python uses following keywords or statements

try

except

finally

raise

assert

try – except statement

If the python program contains suspicious code that may throw the exception, we must place that code in the try block. The try block must be followed with the except statement which contains a block of code that will be executed if there is some exception in the try block.

Syntax:

try:

 #block of code except

Exception1:

 #block of code except

Exception2:

 #block of code #other code

try – except - else statement

We can also use the else statement with the try-except statement in which, we can place the code which will be executed if no exception occurs in the try block.

Syntax:

try:

 #block of code

except Exception1:

 #block of code

else:

 #this code executes if no except block is executed

finally statement

We can use the finally block with the try block in which, we can place the important code which must be executed either try throws an exception or not.

Syntax:

try:

 #block of code

except Exception1:

 #block of code

else:

 #this code executes if no except block is executed

finally:

 # this code will always be executed

try:

Run this Code

except:

Execute this code when there is an exception

else:

No Exception? Run this code.

finally:

Always run this code

raise statement (or) Raising exceptions

We can use raise to throw an exception if a condition occurs. i.e. If you want to throw an error when a certain condition occurs we can use raise keyword.

Syntax:

```
raise Exception_class
```

Example:

try:

```
    age = int(input("Enter the age : "))
```

```
    if age<18:
```

```
        raise Exception;
```

```
    else:
```

```
        print("the age is valid")
```

```
except Exception:
```

```
    print("The age is not valid")
```

Output: Case 1

Enter the age : 34

the age is valid

Output: Case 2

Enter the age : 12

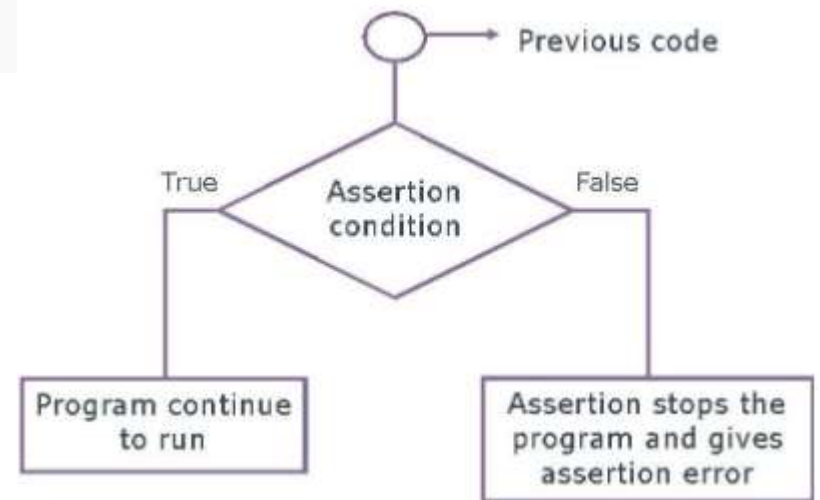
The age is not valid

assert statement (or) Assertions

Assertions are simply Boolean expressions that checks if the conditions return true or not. If it is true, the program does nothing and moves to the next line of code.

However, if it's false, the program stops and throws an error.

It is also a debugging tool as it brings the program on halt as soon as any error is occurred and shows on which point of the program error has occurred.



assert Statement

Python has built-in assert statement to use assertion condition in the program. assert statement has a condition or expression which is supposed to be always true. If the condition is false assert halts the program and gives an AssertionError.

Syntax:

```
assert condition [, error_message]
```

In Python we can use assert statement in two ways as mentioned above.

- assert statement has a condition and if the condition is not satisfied the program will stop and give AssertionError.
- assert statement can also have a condition and a optional error message. If the condition is not satisfied assert stops the program and gives AssertionError along with the error message.

Example:

```
x = int(input("Enter x :"))  
y = int(input("Enter y :"))  
# It uses assert to check for 0 print ("x / y value is : ")  
assert y != 0, "Divide by 0 error"
```

Output: Case 1

```
Enter x :33 Enter y :11  
x / y value is : 3.0
```

Output: Case 2

```
Enter x :33 Enter y :0  
x / y value is :  
AssertionError: Divide by 0 error
```


Modules in Python

In python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module.

Modules in Python provides us the flexibility to organize the code in a logical way. To use the functionality of one module into another, we must have to import the specific module.

Creating Module

Example: demo.py

```
# Python Module example
```

```
def sum(a,b):  
    return a+b
```

```
def sub(a,b):  
    return a-b
```

```
def mul(a,b):  
    return a*b
```

```
def div(a,b):  
    return a/b
```

In the above example we have defined 4 functions sum(), sub(), mul() and div() inside a module named demo.

Loading the module in our python code

We need to load the module in our python code to use its functionality. Python provides two types of statements as defined below.

- 1. import statement**
- 2. from-import statement**

1. import statement

The import statement is used to import all the functionality of one module into another. Here, we must notice that we can use the functionality of any python source file by importing that file as the module into another python source file.

We can import multiple modules with a single import statement.

Syntax:

```
import module1,module2..
```

Example:

```
import demo
#importing entire Module
a=int(input("Enter a :"))
b=int(input("Enter b :"))
print("Sum is :",demo.sum(a,b))
print("Sub is :",demo.sub(a,b))
print("Mul is :",demo.mul(a,b))
print("Div is :",demo.div(a,b))
```

Output:

Enter a :12

Enter b :6

Sum is : 18

Sub is : 6

Mul is : 72

Div is : 2.0

2. from-import statement

Instead of importing the whole module into the namespace, python provides the flexibility to import only the specific attributes of a module. This can be done by using from - import statement. In such case we don't use the dot operator.

Syntax:

```
from module-name import
```

Example:

```
from demo import *  
a=int(input("Enter a :"))  
b=int(input("Enter b :"))  
print("Sum is :",sum(a,b))  
    print("Sub is :",sub(a,b))  
print("Mul is :",mul(a,b))  
print("Div is :",div(a,b))
```

Output:

Enter a :12

Enter b :6

Sum is : 18

Sub is : 6

Mul is : 72

Div is : 2.0

We can import specific function from a module without importing the module as a whole. Here is an example.

Syntax:

```
from module-name import function1,function2...
```

Example:

```
from demo import sub,mul
#importing specific functionality from Module
a=int(input("Enter a :"))
b=int(input("Enter b :"))
print("Sub is :",sub(a,b))
print("Mul is :",mul(a,b))
```

Output:

Enter a :12

Enter b :6

Sub is : 6

Mul is : 72

Renaming a module

Python provides us the flexibility to import some module with a specific name so that we can use this name to use that module in our python source file.

Syntax:

```
import module-name as specific-name
```

Example:

```
import demo as c
a=int(input("Enter a :"))
b=int(input("Enter b :"))
print("Sum is :",c.sum(a,b))
print("Sub is :",c.sub(a,b))
```

Output:

Enter a :25

Enter b :12

Sum is : 37

Sub is : 13

Namespaces in Python

A namespace is basically a system to make sure that all the names in a program are unique and can be used without any conflict.

A namespace is a system to have a unique name for each and every object in Python. An object might be a variable or a method. Python itself maintains a namespace in the form of a Python dictionary.

Let's go through an example, a directory-file system structure in computers. Needless to say, that one can have multiple directories having a file with the same name inside of every directory.

Real-time example, the role of a namespace is like a surname. One might not find a single "Kumar" in the class there might be multiple "Kumar" but when you particularly ask for "N Kumar" or "S Kumar" (with a surname), there will be only one (time being don't think of both first name and surname are same for multiple students).

Types of namespaces

Local Namespace: This namespace includes local names inside a function. This namespace is created when a function is called, and it only lasts until the function returns.

Global Namespace: This namespace includes names from various imported modules that you are using in a project. It is created when the module is included in the project, and it lasts until the script ends.

Built-in Namespace: This namespace includes built-in functions and built-in exception names. Like `print ()`, `input ()`, `list ()` and etc.

Example:

```
print("Namespace Example") #built-in namespace
a=10 #global namespace
def func1():
    b=20 #local namespace
    print(a+b)
func1()
```

Output:30

In above code, print () is built-in namespace, 'a' is in the global namespace in python and 'b' is in the local namespace of func1.

Python supports “global” keyword to update global namespaces in local.

Example:

```
count = 5
def func1():
    global count
    #To update global namespace
    count = count + 1
    print(count)
func1()
```

Output:

6

Example:

```
a=10 #global namespace
def func1():
    b=20 #non-local namespace
    def func2():
        nonlocal b
        c=30 #local namespace
        global a
        a=a+c
        b=b+c
    func2()
    print(a,b)
func1()
```

Output:

40 50

To func2, 'c' is local, 'b' is nonlocal, and 'a' is global. By nonlocal, we mean it isn't global, but isn't local either. Of course, here, you can write 'c', and read both 'b' and 'a'. To update 'b' (non-local namespace), we need to use "nonlocal" keyword and to update 'a' (global namespace), we need to "global" keyword.

globals() and locals()

The `globals()` and `locals()` functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

If `locals()` is called from within a function, it will return all the names that can be accessed locally from that function.

If `globals()` is called from within a function, it will return all the names that can be accessed globally from that function.

The return type of both these functions is dictionary. Therefore, names can be extracted using the `keys()` function.

Example:

```
xy="Madhu"
def sum(a,b):
    c=0
    c=a+b
    print(c)
    print(globals())
    print(locals())
sum(2,3)
```

Output:

5

```
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x006BDBD0>, '__spec__': None, '__annotations__': {}, '__builtins__': , '__file__': 'module1.py', '__cached__': None, 'xy': 'Madhu', 'sum': }
{'a': 2, 'b': 3, 'c': 5}
```


Packages in Python

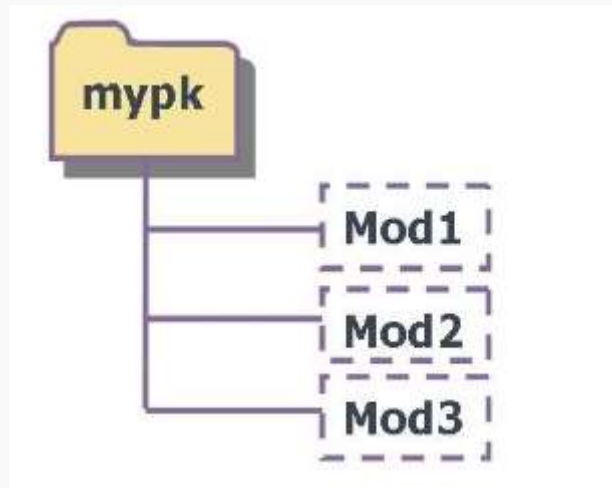
Suppose you have developed a very large application that includes many modules. As the number of modules grows, it becomes difficult to keep track of them all if they are dumped into one location. This is particularly so if they have similar names or functionality.

In python a Package contains collection of modules and sub-packages. Packages are a way of structuring many packages and modules which help in a well-organized hierarchy of data set, making the packages and modules easy to access. Just like there are different drives and folders in an OS to help us store files, similarly packages help us in storing other sub-packages and modules, so that it can be used by the user when necessary.

To create a package in Python, we need to follow these three simple steps:

First, we create a directory and give it a package name, preferably related to its operation. Then we put the all modules in it.

Finally we create an **__init__.py** file inside the directory(folder), to let Python know that the directory is a package. (This is optional from python 3.3)



If a file named **__init__.py** is present in a package directory, it is invoked when the package or a module in the package is imported. This can be used for execution of package initialization code, such as initialization of package-level data, this data can able to use any modules under package.

UNIT – III

Regular Expressions and Multithreading

Regular Expressions

What is Regular Expression

A **Regular Expression** (RegEx) is a sequence of characters that defines a search pattern. For example,

```
^a.....s$
```

The above code defines a RegEx pattern. The pattern is: **any five letter string starting** with a and ending with s.

Python has a module named **re** to work with regular expression. Here's an example:

```
import re

pattern = '^a...s$'
test_string = 'abyss'
Result = re.match(pattern, test_string)

if result:
    print("Search successful.")
else:
    print("Search unsuccessful")
```

Here, we used `re.match` function to grab pattern within the `test_string`. The method returns a match object if the search is successful. If not, it returns **No**

Specify Pattern Using RegEx:

To specify regular expressions, metacharacters are used. In the above example, ^ and \$ are metacharacters.

Meta characters:

Metacharacters are characters that are interpreted in a special way by a RegEx engine. Here's a list of metacharacters:

[] . ^ \$ *

[] – Square brackets:

Square brackets specifies a set of characters you wish to match

| Expression | String | Matched? |
|------------|-----------|-----------|
| [abc] | a | 1 match |
| | ac | 2 matches |
| | Hey Jude | No match |
| | abc de ca | 5 matches |

Above, `[abc]` will match if the string you are trying to match contains any of the a, b or c.

You can also specify a range of characters using `-` inside square brackets.

- `[a-e]` is the same as `[abcde]`.
- `[1-4]` is the same as `[1234]`.
- `[0-39]` is the same as `[01239]`.

You can complement (invert) the character set by using caret `^` symbol at the start of a square-bracket.

- `[^abc]` means any character except a or b or c.
- `[^0-9]` means any non-digit character.

– Period:

period matches any single character (except newline '\n').

| Expression | String | Matched? |
|------------|--------|-----------------------------------|
| ... | a | No match |
| | ac | 1 match |
| | acd | 1 match |
| | acde | 2 matches (contains 4 characters) |

^ – Caret:

The caret symbol ^ is used to check if a string starts with a certain character.

| Expression | String | Matched? |
|------------|--------|--|
| ^a | a | 1 match |
| | abc | 1 match |
| | bac | No match |
| ^ab | abc | 1 match |
| | acb | No match (starts with a but not followed by b) |

\$ – Dollar:

The dollar symbol \$ is used to check if a string ends with a certain character.

| Expression | String | Matched? |
|------------|---------|----------|
| a\$ | a | 1 match |
| | formula | 1 match |
| | cab | No match |

+ – Plus:

The plus symbol + matches one or more occurrences of the pattern left to it.

| Expression | String | Matched? |
|------------|--------|-----------------------------------|
| ma+n | mn | No match (no a character) |
| | man | 1 match |
| | maan | 1 match |
| | main | No match (a is not followed by n) |
| | woman | 1 match |

Python RegEx

Python has a module named **re** to work with regular expressions. To use it, we need to import the module.

```
import re
```

The module defines several functions and constants to work with RegEx.

re.findall()

The re.findall() method returns a list of strings containing all matches

Example:

```
#program to extract numbers from a string
```

```
import re
```

```
string = 'hello 12 hi 89. Howdy 34'
```

```
pattern = '\d+'
```

```
result = re.findall(pattern, string)
```

```
print(result)
```

```
#Output: ['12', '89', '34']
```

If the pattern is no found, re.findall() returns an empty list.

re.split()

The re.split method splits the string where there is a match and returns a list of strings where the splits have occurred.

Example:

```
import re

String = 'Twelve:12 Eighty nine:89'
pattern = '\d+'

result = re.split(pattern, string)
Print( result )

#Output: [ 'Twelve:', ' Eighty nine:', ' .' ]
```

If the pattern is no found, re.split() returns a list containing an empty string.

re.sub()

GXTCS

The syntax of re.sub() is:

```
re.sub(pattern, replace, string)
```

Example:

```
#program to remove all white spaces
Import re
string = 'abc 12\ de 23 \n f4 6'

#matches all whitespace characters
pattern = '\s+'

#empty string
replace = ''

new_string = re.sub(pattern, replace, string)
```

If the pattern is not found, re.sub() returns the original string.

re.subn ()

The re.subn() is similar to re.sub() expect it returns a tuple of 2 items containing the new string and the number of substitutions made.

Example:

```
#program to remove all whitespaces
Import re
#multiline string
string = 'abc 12\ de 23 \n f45 6'

# matches all whitespace characters
pattern = '\s+'

#empty string
replace = ''
new_string = re.subn(pattern, replace, string)
print(new_string)

#Output: ( 'abc12de23f456', 4 )
```

re.search()

The `re.search()` method takes two arguments: a pattern and a string. The method looks for the first location where the RegEx pattern produces a match with the string. If the search is successful, `re.search()` returns a match object; if not, it returns `None`.

```
match = re.search(pattern, str)
```

Example:

```
import re
String = "Python is fun"

#check if 'Python' is at the beginning
match = re.search('\APython', string)

if match:
    print("pattern found inside the string")
else:
    print("pattern not found")

#Output : pattern found inside the string
```

Here, match contains a match object.

Math Object

You can get methods and attributes of a match object using dir() function. Some of the commonly used methods and attributes of match objects are:

```
Import re
String = ` 39801 356, 2102 1111 `

#Three digit number followed by space followed by two digit number
Pattern = ` (\d{3}) (\d{2}) `

#match variable contains a Match object.
Match = re.search(pattern, string)

If match:
    print(match.group())
Else:
    print("` pattern not found ")
# Output: 801 35
```

ere, match variable contains a match object.

ur pattern `(\d{3}) (\d{2})` has two subgroups `(\d{3})` and `(\d{2})`. You can get the part of the string of these parenthesized subgroups. Here's how:

```
>>> match.group(1)  
'801'
```

```
>>> match.group(2)  
'35'
```

```
>>> match.group(3)  
('801', '35')
```

```
>>> match.group()  
('801', '35')
```

`match.start()`, `match.end()` and `match.span()`

The `start()` function returns the index of the matched substring. Similarly, `end()` returns the end index of the matched substring.

```
>>> match.start()  
2  
>>> match.end()  
8
```

The `span()` function returns a tuple containing start and end index of the matched part.

```
>>> match.span()  
(2, 8)
```

match.re and match.string

The `re` attribute of a matched object returns a regular expression object. Similarly, `string` attribute returns the passed string.

```
>>> match.re  
Re.compile('\\d{3}) (\\d{2}) '
```

```
>>> match.string  
' 39801 356, 2102 1111 '
```

Multithreaded Programming (or) Multithreading in python

Introduction:

- Multitasking is a process of executing multiple tasks simultaneously, we use multitasking to utilize the CPU.
- Multitasking can be achieved by two ways or classified into two types
 - **Process-Based Multitasking(Multiprocessing)**
 - **Thread-Based Multitasking(Multithreading)**
- **Process-Based Multitasking(Multiprocessing):**
Executing multiple tasks simultaneously, where each task is separate independent process (or) program is called as process based multitasking.

Example:

- Typing a python program in notepad
- Listening audio songs
- Download a file from internet
- The above three tasks are performed simultaneously in a system, but there is no dependence between one task and another task.
- Process based multitasking is best suitable at “Operating System” level not at programming level.

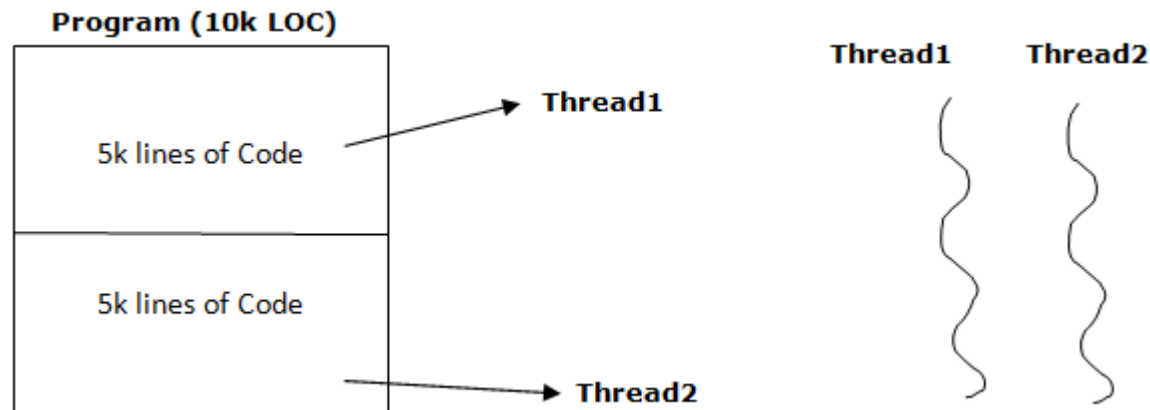
- **Thread-Based Multitasking(Multithreading):**

Executing multiple tasks simultaneously, where each task is separate independent part of process (or) program is called as thread based multitasking.

- The each independent part is called as thread. The thread based multitasking is best suitable at programming level.

Example:

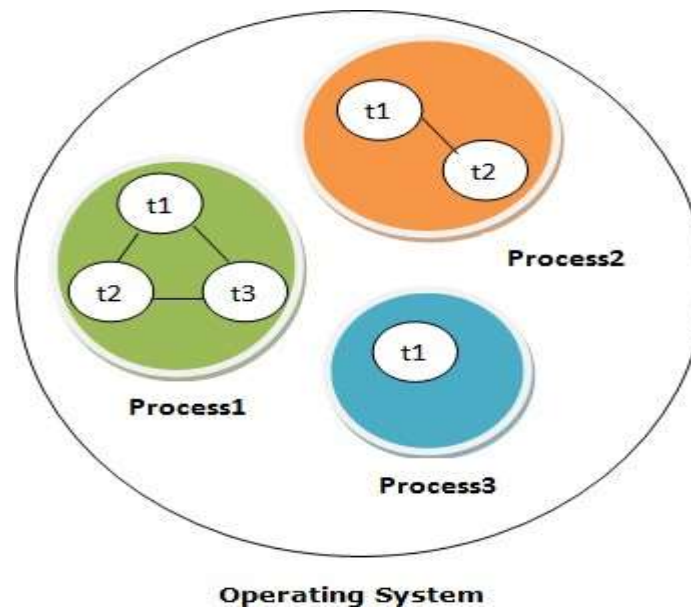
Let a program has 10k line of code, where last 5k lines of code doesn't depend on first 5k lines of code, then both are the execution simultaneously. So takes less time to complete the execution.



Note: Any type of multitasking is used to reduce response time of system and Improves performance.

Multithreading:

- A **thread** is a lightweight process; In simple words, a **thread** is a sequence of some instructions within a program that can be executed independently of other code.
- Threads are independent; if there is an exception in one thread it doesn't affect remaining threads.
- Threads shares common memory area.



- As shown in the figure, a thread is executed inside the process. There can be multiple processes inside the OS, and each process can have multiple threads.

- ❑ **Definition:** Multithreading is a process of executing multiple threads simultaneously. Multithreading allows you to break down an application into multiple sub-tasks and run these tasks simultaneously.
- In other words, the ability of a process to execute multiple threads parallelly is called multithreading. Ideally, multithreading can significantly improve the performance of any program.
 - Multiprocessing and Multithreading both are used to achieve multitasking, but we use multithreading than multiprocessing because threads shares a common memory area and context-switching between threads takes less time than process.

Advantages:

- Multithreading can significantly improve the speed of computation on multiprocessor
- Multithreading allows a program to remain responsive while one thread waits for input, and another runs a GUI at the same time.

Disadvantages:

- It raises the possibility of deadlocks.
- It may cause starvation when a thread doesn't get regular access to shared resources.

❑ **Global Interpreter Lock (GIL):**

- Execution of Python code is controlled by the Python Virtual Machine.
- Python was designed in such a way that only one thread may be executing in Python Virtual Machine. similar to how multiple processes in a system share a single CPU.
- Many programs may be in memory, but only one is live on the CPU at any given moment.
- Likewise, although multiple threads may be "running" within the Python interpreter, only one thread is being executed by the interpreter at any given time.
- Access to the Python Virtual Machine is controlled by the global interpreter lock (GIL). This lock is what ensures that exactly one thread is running.

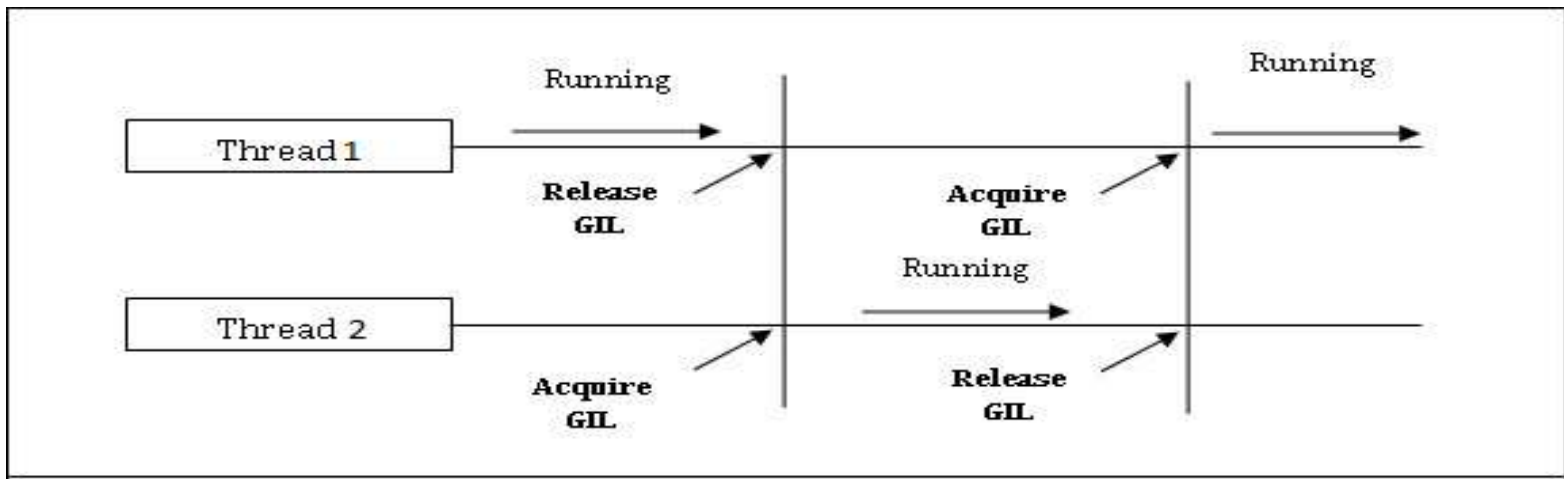
- This essentially means is a process can run only one thread at a time. When a thread starts running, it acquires GIL and when it waits for I/O, it releases the GIL, so that other threads of that process can run.

For example

Let us suppose process P1 has threads Thread1 and Thread2.

Thread1 running (acquire GIL) -> **Thread1** waiting for I/O (releases GIL)

-> **Thread2** running (acquires GIL) -> **Thread2** waiting for I/O (releases GIL)



❑ Multithreading Modules:

Python offers two modules to implement threads in programs.

- **thread module** and
- **threading module**.

Thread Module(**_thread**):

- This module provides low-level primitives for working with multiple threads .
- The `<_thread>` module supports one method to create thread. That is
`thread.start_new_thread(function, args)`
- This method starts a new thread and returns its identifier. It'll invoke the function specified as the “function” parameter with the passed list of arguments. When the `<function>` returns, the thread would silently exit.
- Here, args is a tuple of arguments; use an empty tuple to call `<function>` without any arguments.

Note: Python 2.x used to have the `<thread>` module. But it got deprecated in Python 3.x and renamed to `<_thread>` module for backward compatibility.

Example:

```
from _thread import start_new_thread
from time import sleep

def disp(n):
    for i in range(5):
        print(n)

start_new_thread(disp, ("hai", ))
start_new_thread(disp, ("hello", ))
sleep(2)
print("threads are executed...")
```

Output:

```
>>>python multithr1.py hai
hello hai hai hai hai
hello hello hello hello
threads are executed...
```


Threading Module:

- The threading module provides more features and good support for threads than thread module.
- This Module provides **Thread** class, and this **Thread** class provide following methods
 - **start()** – The start() method starts a thread by calling the run method.
 - **join([time])** – The join() waits for threads to terminate.
 - **isAlive()** – The isAlive() method checks whether a thread is still executing.
 - **getName()** – The getName() method returns the name of a thread.
 - **setName()** – The setName() method sets the name of a thread.

Creating Thread Using Threading Module

- To implement a new thread using the threading module, use following code snippet

Syntax:

threading.Thread(target=None, name=None, args=())

This method has following arguments. Those are:

- **target** is the callable function to be invoked by the [run\(\)](#) method. Defaults to None, meaning nothing is called.
- **name** is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number.
- **args** is the argument tuple for the function invocation. Defaults to ().

Example:

```
import threading #function display def display(msg):
```

```
    for i in range(5):
```

```
        print(msg)
```

```
# creating thread
```

```
t1 = threading.Thread(target=display, args=("Thread1",))
```

```
t2 = threading.Thread(target=display, args=("Thread2",))
```

```
# starting thread 1
```

```
t1.start()
```

```
# starting thread 2
```

```
t2.start()
```

```
# wait until thread 1 is completely executed
```

```
t1.join()
```

```
# wait until thread 2 is completely executed
```

```
t2.join()
```

```
# both threads completely executed
```

```
print("Done!")
```

Output:

```
>>>python multithr.py
```

```
Thread1
```

```
Thread1
```

```
Thread2
```

```
Thread1
```

```
Thread1
```

```
Thread2
```

```
Thread1
```

```
Thread2
```

```
Thread2
```

```
Thread2
```

```
Done!
```

Let us try to understand the above code:

- To import the threading module, we do:

import threading

- To create a new thread, we create an object of Thread class. It takes following arguments:

target: the function to be executed by thread

args: the arguments to be passed to the target function

- In above example, we created 2 threads with different target functions:

t1 = threading.Thread(target=display, args=("Thread1",))

t2 = threading.Thread(target=display, args=("Thread2",))

- To start a thread, we use start method of Thread class.

t1.start()

t2.start()

- Once the threads start, the current program also keeps on executing. In order to stop execution of current program until a thread is complete, we use join method.

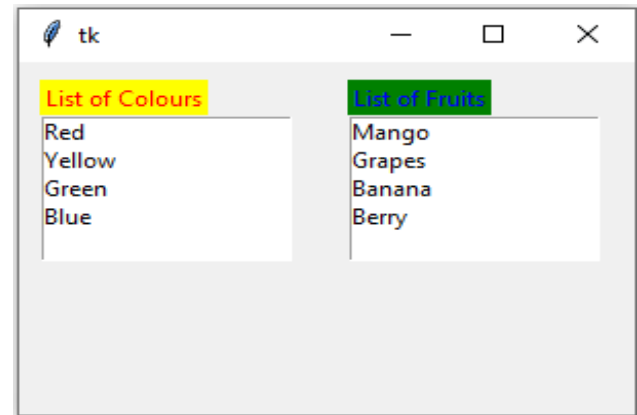
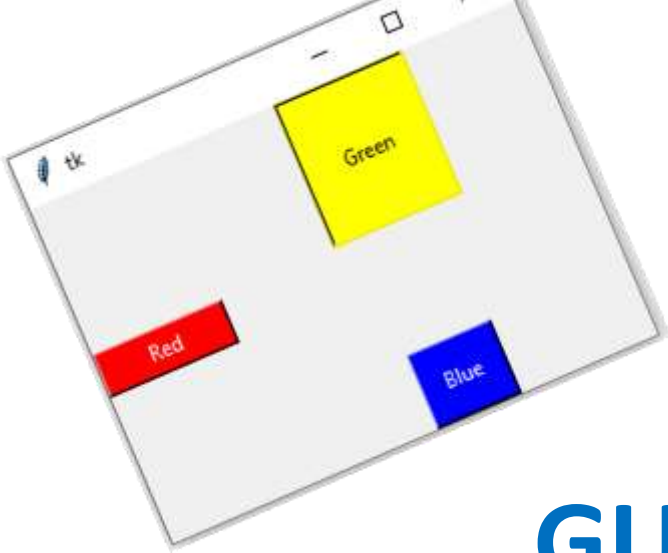
t1.join()

t2.join()

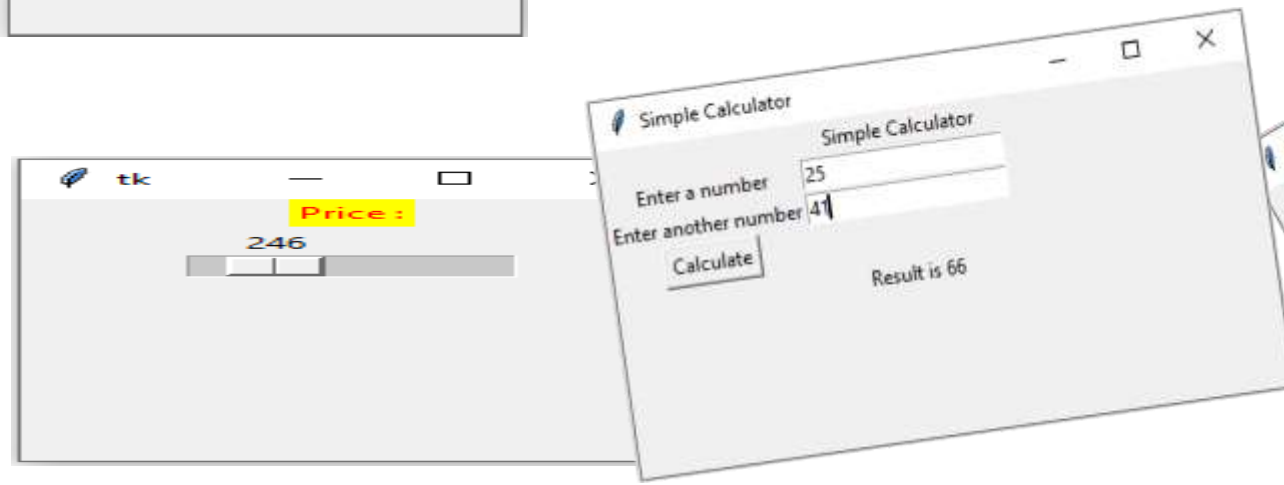
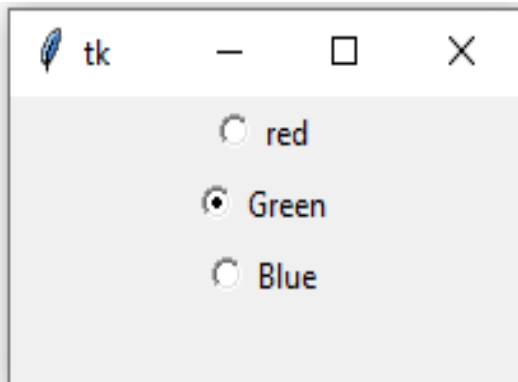
- As a result, the current program will first wait for the completion of t1 and then t2. Once, they are finished, the remaining statements of current program are executed.

UNIT – IV

**GUI Programming and Web
Programming**



GUI Programming in Python



Introduction:

- A graphical user interface is an application that has buttons, windows, and lots of other widgets that the user can use to interact with your application.
- A good example would be a web browser. It has buttons, tabs, and a main window where all the content loads.
- In GUI programming, a **top-level root** windowing object contains all of the **little windowing objects** that will be part of your complete GUI application.
- These windowing objects can be text labels, buttons, list boxes, etc. These individual little GUI components are known as **widgets**.

The image is a screenshot of a Tkinter window titled 'Students'. The window has a standard title bar with minimize, maximize, and close buttons. Inside the window, there are two text input fields. The first field is labeled 'Name :' and contains the text 'abc'. The second field is labeled 'Regd No :' and contains the text '123'. Below these fields is a button labeled 'Submit'.

- Python offers multiple options for developing GUI (Graphical User Interface). The most commonly used **GUI method** is **tkinter**.
- **Tkinter** is the easiest among all to get started with. It is Python's standard GUI (Graphical User Interface) package. It is the most commonly used toolkit for **GUI Programming** in Python
- since Tkinter is the Python interface to Tk (Tea Kay), it can be pronounced as **Tea-Kay-inter**. i.e tkinter = **t k inter**.

tkinter - GUI for Python:

- Python provides the standard library **tkinter** for creating the graphical user interface for **desktop based applications**.
- Developing desktop based applications with **tkinter** is not a complex task.
- A Tkinter window application can be created by using the following steps.
 1. **Import** the **tkinter** module.
 2. Create the **main application window**.
 3. Add the **widgets** like labels, buttons, frames, etc. to the window.
 4. Call the **main event loop** so that the actions can take place on the user's computer screen.

1. Importing tkinter is same as importing any other module in the python code. Note that the name of the module in **Python 2.x** is '**Tkinter**' and in **Python 3.x** is '**tkinter**'.

import tkinter (or) **from tkinter import ***

2. After importing **tkinter** module we need to create a main window, tkinter offers a method '**Tk()**' to create **main window**. The basic code used to create the main window of the application is:

top = tkinter.Tk() (or) **top=Tk()**

3. After creating main window, we need to **add components** or **widgets** like labels, buttons, frames, etc.

4. After adding widgets to **main window**, we need to run the application, tkinter offers a method '**mainloop()**' to run application. The basic code used to run the application is:

top.mainloop ()

Example: tkndemo.py

```
import tkinter
top = tkinter.Tk()           #creating the application main window.
top.title("Welcome")         #title of main window
top.geometry("400x300")      #size of main window
top.mainloop()               #calling the event main loop
```

Output:

```
>>> python tkndemo.py
```

Title of window



**Main Window
(400x300)**

- tkinter also offers access to the geometric configuration of the widgets which can organize the widgets in the parent windows.

Tkinter provides the following geometry methods

1. pack () method:

The **pack()** method is used to organize components or widgets in main window.

Syntax:

widget.pack (options)

The possible options are

side: it represents the side to which the widget is to be placed on the window. Side may be **LEFT** or **RIGHT** or **TOP(default)** or **BOTTOM**.

Example: tknpack.py

```
from tkinter import *
```

```
top = Tk()
```

```
top.geometry("300x200")
```

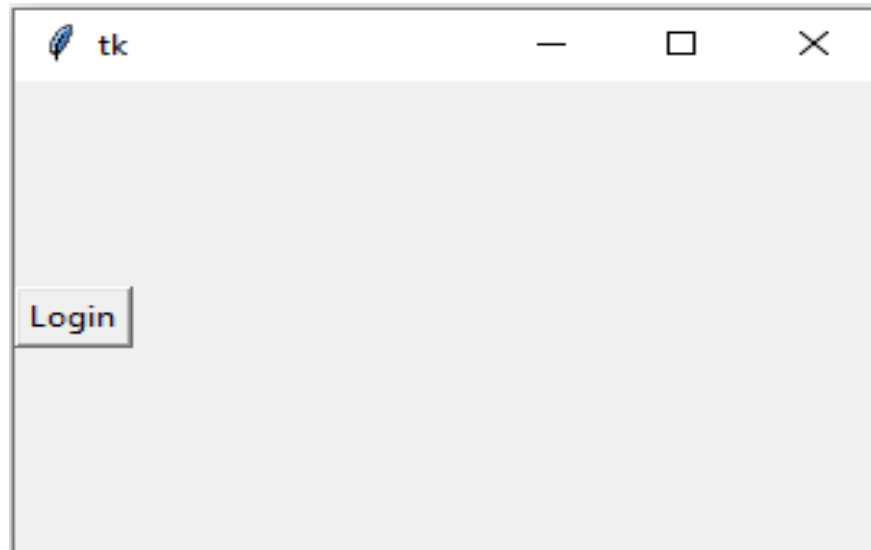
```
btn1 = Button(top, text = "Login")
```

```
btn1.pack( side = LEFT)
```

```
top.mainloop()
```

Output:

```
>>> python tknpack.py
```



2. grid() method:

The **grid()** method organizes the widgets in the tabular form. We can specify the rows and columns as the options in the method call.

This is a more organized way to place the widgets to the python application.

Syntax:

widget.grid (options)

The possible options are

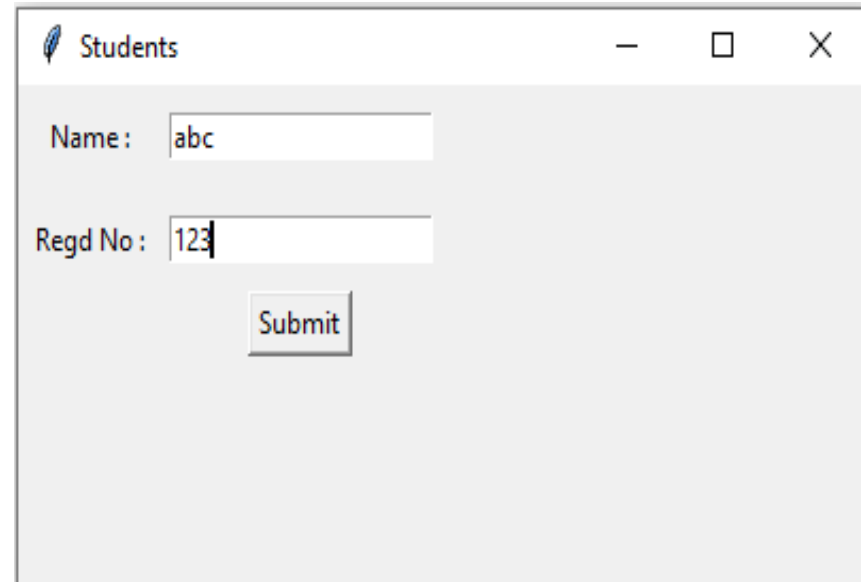
- **Column**
The column number in which the widget is to be placed. The leftmost column is represented by **0**.
- **padx, pady**
It represents the number of pixels to pad the widget outside the widget's border.
- **row**
The row number in which the widget is to be placed. The topmost row is represented by **0**.

Example: tkngrid.py

```
from tkinter import *
parent = Tk()
parent.title("Students")
parent.geometry("300x200")
name = Label(parent, text = "Name : ")
name.grid(row = 0, column = 0, pady=10, padx=5)
e1 = Entry(parent)
e1.grid(row = 0, column = 1)
regno = Label(parent, text = "Regd No : ")
regno.grid(row = 1, column = 0, pady=10, padx=5)
e2 = Entry(parent)
e2.grid(row = 1, column = 1)
btn = Button(parent, text = "Submit")
btn.grid(row = 3, column = 1)
parent.mainloop()
```

Output:

```
>>>python tkngrid.py
```



3. place() method:

The place() method organizes the widgets to the specific **x** and **y** coordinates.

Syntax:

widget.place(x,y)

- **x, y:** It refers to the horizontal and vertical offset in the pixels.

Example: tknplace.py

```
from tkinter import *
```

```
parent = Tk()
```

```
parent.title("Students")
```

```
parent.geometry("300x200")
```

```
name = Label(parent,text = "Name : ")
```

```
name.place(x=50,y=50)
```

```
e1 = Entry(parent)
```

```
e1.place(x=100,y=50)
```

```
regno = Label(parent,text = "Regd No : ")
```

```
regno.place(x=50,y=100)
```

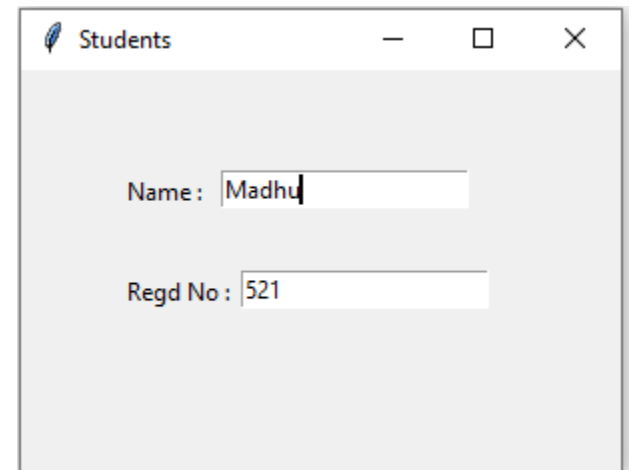
```
e2 = Entry(parent)
```

```
e2.place(x=110,y=100)
```

```
parent.mainloop()
```

Output:

```
>>>python tknplace.py
```



- **Tkinter widgets or components:**

Tkinter supports various widgets or components to build GUI application in python.

| Widget | Description |
|--------------------|--|
| Button | Creates various buttons in Python Application. |
| Checkbutton | Select one or more options from multiple options.(Checkbox) |
| Entry | Allows the user to enter single line of text(Textbox) |
| Frame | Acts like a container which can be used to hold the other widgets |
| Label | Used to display non editable text on window |
| Listbox | Display the list items, The user can choose one or more items. |
| Radiobutton | Select one option from multiple options. |
| Text | Allows the user to enter single or multiple line of text(Textarea) |
| Scale | Creates the graphical slider, the user can slide through the range of values |
| Toplevel | Used to create and display the top-level windows(Open a new window) |

❖ Button Widget in Tkinter:

- The Button is used to add various kinds of buttons to the python application. We can also associate a method or function with a button which is called when the button is pressed.

Syntax: **name = Button(parent, options)**

The options are

- **activebackground:**It represents the background of the button when it is active.
- **activeforeground:**It represents the font color of the button when it is active..
- **bd:** It represents the border width in pixels.
- **bg:** It represents the background color of the button.
- **command:**It is set to the function call which is scheduled when the function is called.
- **text:** It is set to the text displayed on the button.
- **fg:** Foreground color of the button.
- **height:**The height of the button.
- **padx:**Additional padding to the button in the horizontal direction.
- **pady:**Additional padding to the button in the vertical direction.
- **width:**The width of the button.

Example: **btndemo1.py**

```
from tkinter import *
from tkinter import messagebox

top = Tk()
top.geometry("300x200")


def fun():
    messagebox.showinfo("Hello", "Blue Button clicked")


btn1 = Button(top, text = "Red",bg="red",fg="white",width=10)
btn1.pack( side = LEFT)

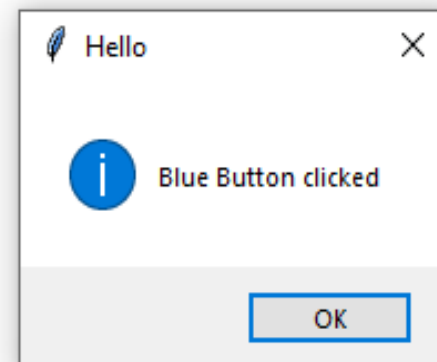
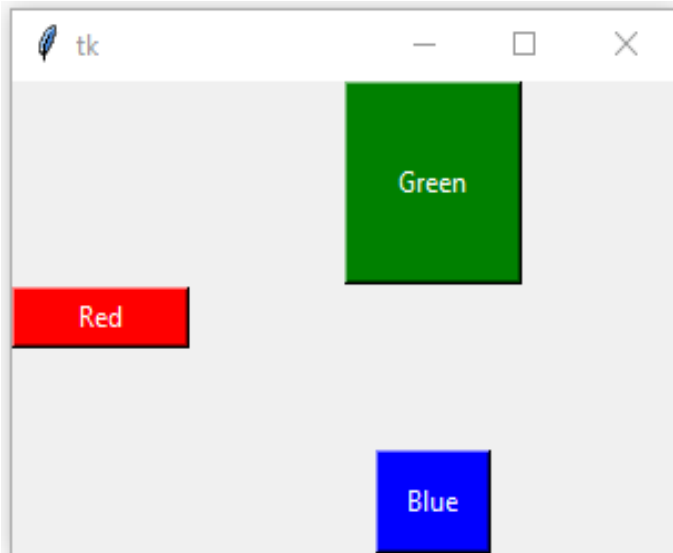
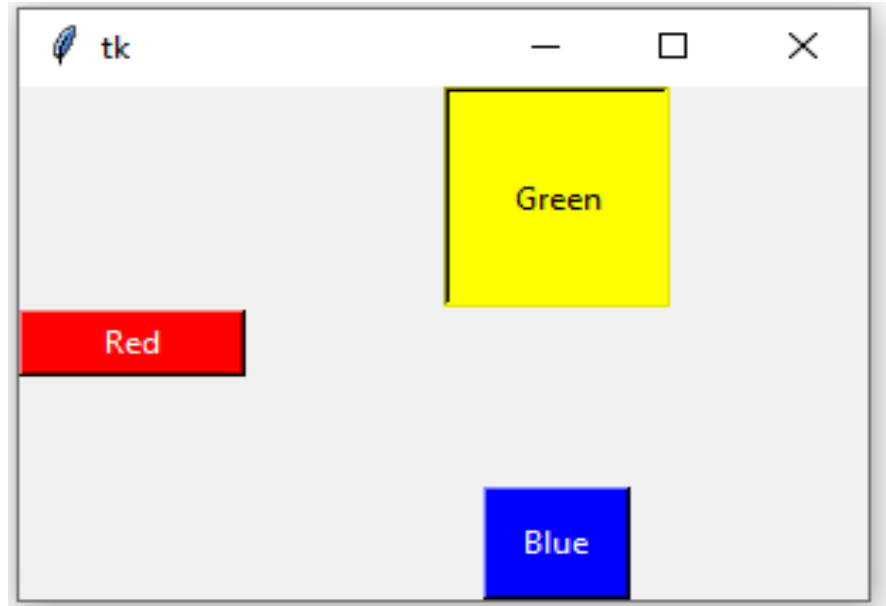
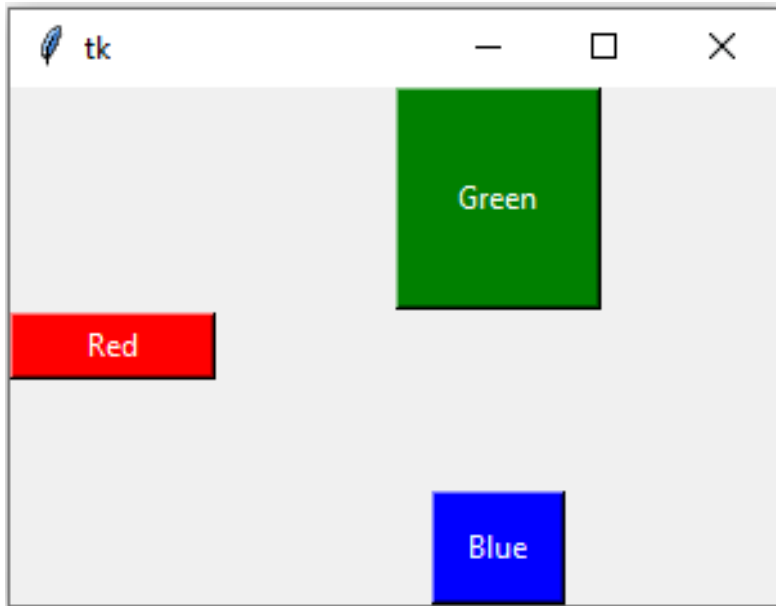
btn2 = Button(top, text = "Green",bg="green",fg="white",width=10,height=5,
    activebackground="yellow")
btn2.pack( side = TOP)

btn3 = Button(top, text ="Blue",bg="blue",fg="white",padx=10,pady=10,
    command=fun)
btn3.pack( side = BOTTOM)

top.mainloop()
```

Output:

```
>>>python btndemo1.py
```



❖ Checkbutton Widget in Tkinter:

- The Checkbutton is used to display the CheckButton on the window. The Checkbutton is mostly used to provide many choices to the user among which, the user needs to choose the one. It generally implements many of many selections.

Syntax: **name = Checkbutton(parent, options)**

The options are

- **activebackground:** It represents the background of the Checkbutton when it is active.
- **activeforeground:** It represents the font color of the Checkbutton when when it is active.
- **bd:** It represents the border width in pixels.
- **bg:** It represents the background color of the Checkbutton.
- **command:** It is set to the function call which is scheduled when the function is called.
- **text:** It is set to the text displayed on the Checkbutton.
- **fg:** Foreground color of the Checkbutton.
- **height:** The height of the Checkbutton.
- **padx:** Additional padding to the Checkbutton in the horizontal direction.
- **pady:** Additional padding to the Checkbutton in the vertical direction.
- **width:** The width of the Checkbutton.

Example: **chbtndemo.py**

```
from tkinter import *
```

```
top = Tk()
```

```
top.geometry("300x200")
```

```
cbtn1 = Checkbutton(top, text="red",fg="red")
```

```
cbtn1.pack()
```

```
cbtn2 = Checkbutton(top, text="Green",fg="green",activebackground="orange")
```

```
cbtn2.pack()
```

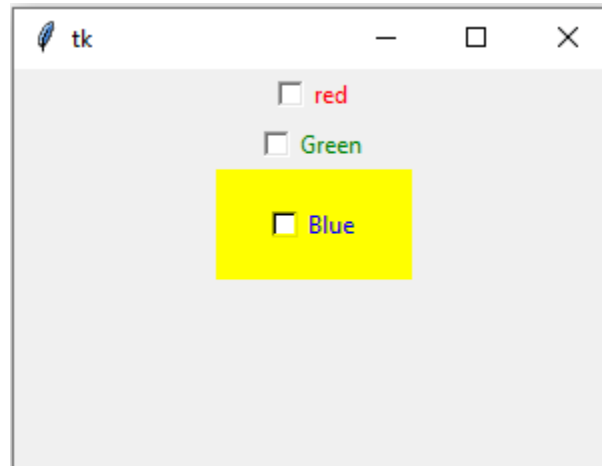
```
cbtn3 = Checkbutton(top, text="Blue",fg="blue",bg="yellow",width=10,height=3)
```

```
cbtn3.pack()
```

```
top.mainloop()
```

Output:

```
>>>python chbtndemo.py
```



❖ Entry Widget in Tkinter:

- The Entry widget is used to provide the single line text-box to the user to accept a value from the user. We can use the Entry widget to accept the text strings from the user.

Syntax: **name = Entry(parent, options)**

The options are

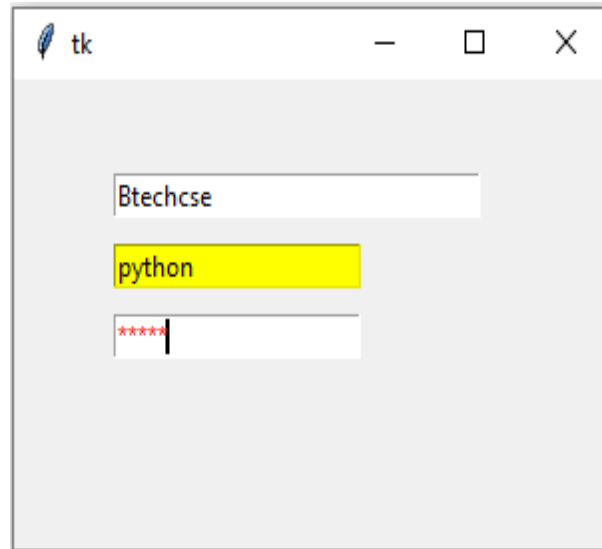
- **bd:** It represents the border width in pixels.
- **bg:** It represents the background color of the Entry.
- **show:** It is used to show the entry text of some other type instead of the string. For example, the password is typed using stars (*).
- **fg:** Foreground color of the Entry.
- **width:** The width of the Entry.

Example: **entrydemo.py**

```
from tkinter import *  
top = Tk()  
top.geometry("300x200")  
enty0 = Entry(top,width="30")  
enty0.place(x=50,y=40)  
enty1 = Entry(top,bg="yellow")  
enty1.place(x=50,y=70)  
enty2 = Entry(top,fg="red",show="*")  
enty2.place(x=50,y=100)  
top.mainloop()
```

Output:

>>>python entrydemo.py



❖ Frame Widget in Tkinter:

- Frame widget is used to organize the group of widgets. It acts like a container which can be used to hold the other widgets. The rectangular areas of the screen are used to organize the widgets to the python application.

Syntax: **name = Frame(parent, options)**

The options are

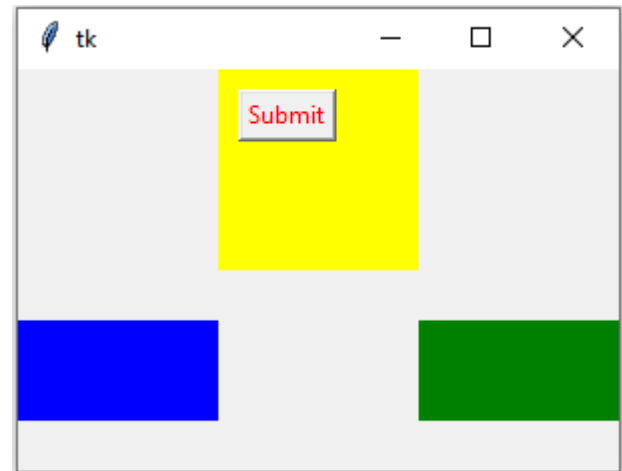
- **bd:** It represents the border width in pixels.
- **bg:** It represents the background color of the frame.
- **width:** The width of the frame.
- **height:** The height of the frame.

Example: **framedemo.py**

```
from tkinter import *  
top = Tk()  
top.geometry("300x200")  
tframe = Frame(top,width="100",height="100",bg="yellow")  
tframe.pack()  
lframe = Frame(top,width="100",height="50",bg="blue")  
lframe.pack(side = LEFT)  
rframe = Frame(top,width="100",height="50",bg="green")  
rframe.pack(side = RIGHT)  
btn1 = Button(tframe, text="Submit", fg="red")  
btn1.place(x=10,y=10)  
top.mainloop()
```

Output:

>>>python framedemo.py



❖ Label Widget in Tkinter:

- The Label is used to specify the container box where we can place the text or images.

Syntax: **name = Label(parent, options)**

The options are

- **bd:** It represents the border width in pixels.
- **bg:** It represents the background color of the label.
- **text:** It is set to the text displayed on the label.
- **fg:** Foreground color of the label.
- **height:** The height of the label.
- **image:** It is set to the image displayed on the label.
- **padx:** Additional padding to the label in the horizontal direction.
- **pady:** Additional padding to the label in the vertical direction.
- **width:** The width of the label.

Example: **labeldemo.py**

```
from tkinter import *
```

```
top = Tk()
```

```
top.geometry("300x200")
```

```
lbl1 = Label(top, text="Name")
```

```
lbl1.place(x=10,y=10)
```

```
lbl2 = Label(top, text="Password", fg="red",bg="yellow")
```

```
lbl2.place(x=10,y=40)
```

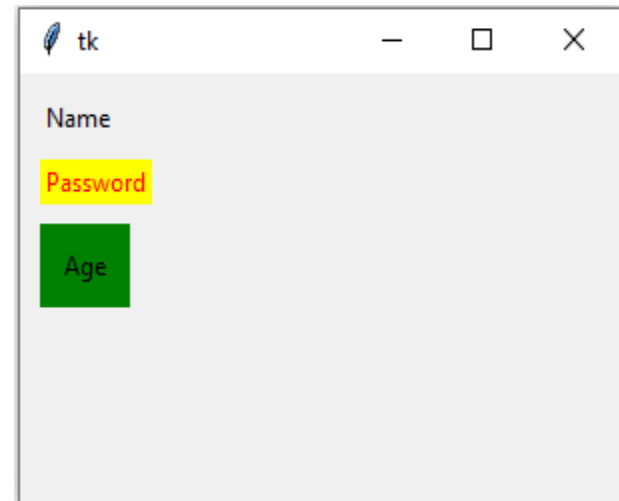
```
lbl3 = Label(top, text="Age", padx=10,pady=10,bg="green")
```

```
lbl3.place(x=10,y=70)
```

```
top.mainloop()
```

Output:

```
>>>python labeldemo.py
```



❖ Listbox Widget in Tkinter:

- The Listbox widget is used to display the list items to the user. We can place only text items in the Listbox. The user can choose one or more items from the list.

Syntax: **name = Listbox(parent, options)**

The options are

- **bd:** It represents the border width in pixels.
- **bg:** It represents the background color of the listbox.
- **fg:** Foreground color of the listbox.
- **width:** The width of the listbox.
- **height:** The height of the listbox.

The following method is associated with the Listbox to insert list item to listbox at specified index.i.e, **insert ()**.

Syntax:

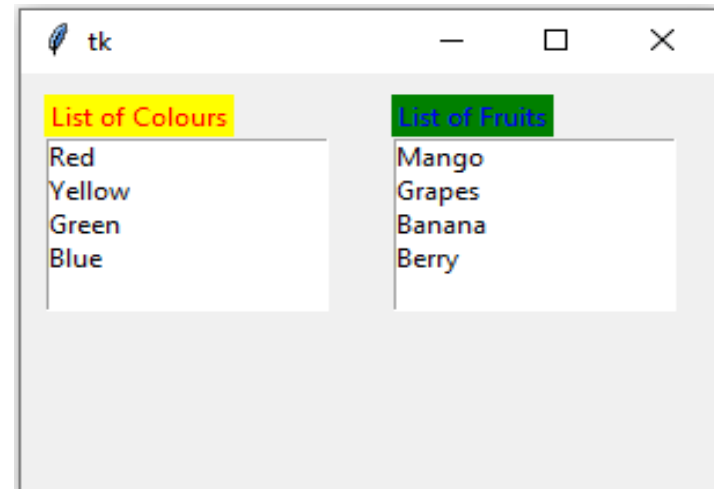
Listbox.insert (index, item)

Example: listboxdemo.py

```
from tkinter import *
top = Tk()
top.geometry("300x200")
lbl1 = Label(top, text="List of Colours",fg="red",bg="yellow")
lbl1.place(x=10,y=10)
lb = Listbox(top,height=5)
lb.insert(1,"Red")
lb.insert(2, "Yellow")
lb.insert(3, "Green")
lb.insert(4, "Blue")
lb.place(x=10,y=30)
lbl2 = Label(top, text="List of Fruits",fg="blue",bg="green")
lbl2.place(x=160,y=10)
lb1 = Listbox(top,height=5)
lb1.insert(1,"Mango")
lb1.insert(2, "Grapes")
lb1.insert(3, "Banana")
lb1.insert(4, "Berry")
lb1.place(x=160,y=30)
top.mainloop()
```

Output:

```
>>>python listboxdemo.py
```



❖ Radiobutton Widget in Tkinter:

- The Radiobutton widget is used to select one option among multiple options. The Radiobutton is different from a checkbutton. Here, the user is provided with various options and the user can select only one option among them.

Syntax: **name = Radiobutton(parent, options)**

The options are

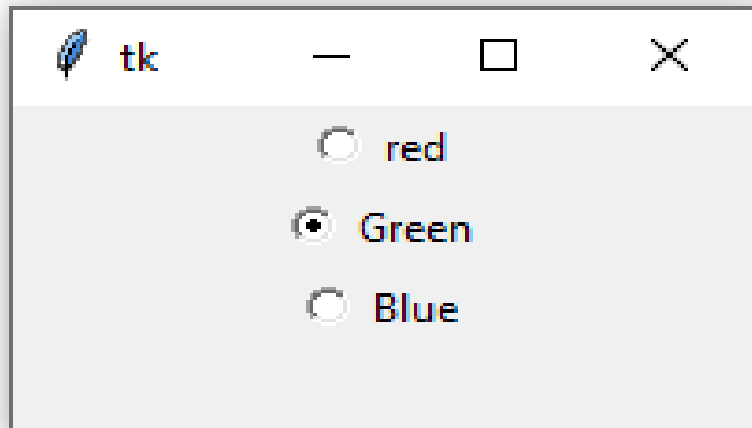
- **activebackground:** It represents the background of the Radiobutton when it is active.
- **activeforeground:** It represents the font color of the Radiobutton when when it is active.
- **bd:** It represents the border width in pixels.
- **bg:** It represents the background color of the Radiobutton.
- **command:** It is set to the function call which is scheduled when the function is called.
- **text:** It is set to the text displayed on the Radiobutton.
- **fg:** Foreground color of the Radiobutton.
- **height:** The height of the Radiobutton.
- **padx:** Additional padding to the Radiobutton in the horizontal direction.
- **pady:** Additional padding to the Radiobutton in the vertical direction.
- **width:** The width of the Radiobutton.
- **Variable:** It is used to keep track of the user's choices. It is shared among all the radiobuttons.

Example: `rbtndemo.py` `from tkinter import *` `top = Tk()`
`top.geometry("200x100")` `radio = IntVar()`

```
rbtn1 = Radiobutton(top, text="red",variable=radio,value="1")  
rbtn1.pack()  
rbtn2 = Radiobutton(top, text="Green",variable=radio,value="2")  
rbtn2.pack()  
rbtn3 = Radiobutton(top, text="Blue",variable=radio,value="3")  
rbtn3.pack()  
top.mainloop()
```

Output:

>>>python rbtndemo.py



❖ Text Widget in Tkinter:

- The Text widget allows the user to enter multiple lines of text. It is different from Entry because it provides a multi-line text field to the user so that the user can write the text and edit the text inside it.

Syntax: **name = Text(parent, options)**

The options are

- **bd:** It represents the border width in pixels.
- **bg:** It represents the background color of the Text.
- **show:** It is used to show the entry text of some other type instead of the string. For example, the password is typed using stars (*).
- **fg:** Foreground color of the Text.
- **width:** The width of the Text.
- **height:** The vertical dimension of the widget in lines.

Example: textdemo.py

```
from tkinter
```

```
import * top =
```

```
Tk()
```

```
top.title("Address
```

```
")
```

```
top.geometry("300x200")
```

```
lbl=Label(top,text="Address
```

```
top.mainloop()
```

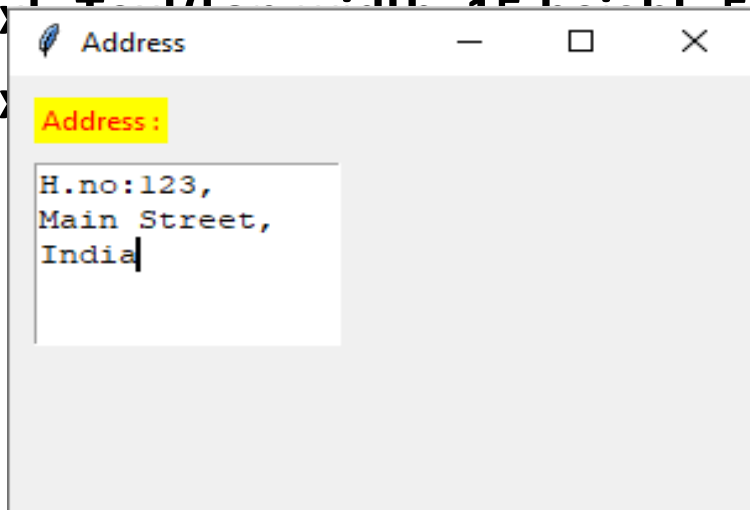
```
"fg="red",bg="yellow")
```

Output:

```
lbl.place(x=10,y=10)
```

```
top.mainloop()
```

```
top.mainloop()
```



❖ Scale Widget in Tkinter:

- The Text widget allows the user to enter multiple lines of text. It is different from Entry because it provides a multi-line text field to the user so that the user can write the text and edit the text inside it.

Syntax: **name = Scale(parent, options)**

The options are

- **activebackground:** It represents the background of the Scale when it is active.
- **bd:** It represents the border width in pixels.
- **bg:** It represents the background color of the Scale.
- **command:** It is set to the function call which is scheduled when the function is called.
- **fg:** Foreground color of the Scale.
- **from_:** It is used to represent one end of the widget range.
- **to:** It represents a float or integer value that specifies the other end of the range represented by the scale.
- **orient:** It can be set to horizontal or vertical depending upon the type of the scale.

Example: scaledemo.py

from tkinter

import *

top =

Tk()

top.geometry("200x200")

l = Label(top, text = "Price",

scale.pack(anchor=CENTER)

Output:

>>>python scaledemo.py



❖ Toplevel Widget in Tkinter:

- The Toplevel widget is used to create and display the toplevel windows which are directly managed by the window manager.

Syntax: **name = Toplevel(options)**

The options are

- **bd:** It represents the border width in pixels.
- **bg:** It represents the background color of the Toplevel.
- **fg:** Foreground color of the Toplevel.
- **width:** The width of the Toplevel.
- **height:** The vertical dimension of the widget in lines.

Example: topleveldemo.py

```
from tkinter import *
```

```
top = Tk() top.geometry("300x200") def fun():
```

```
    chld = Toplevel(top)
```

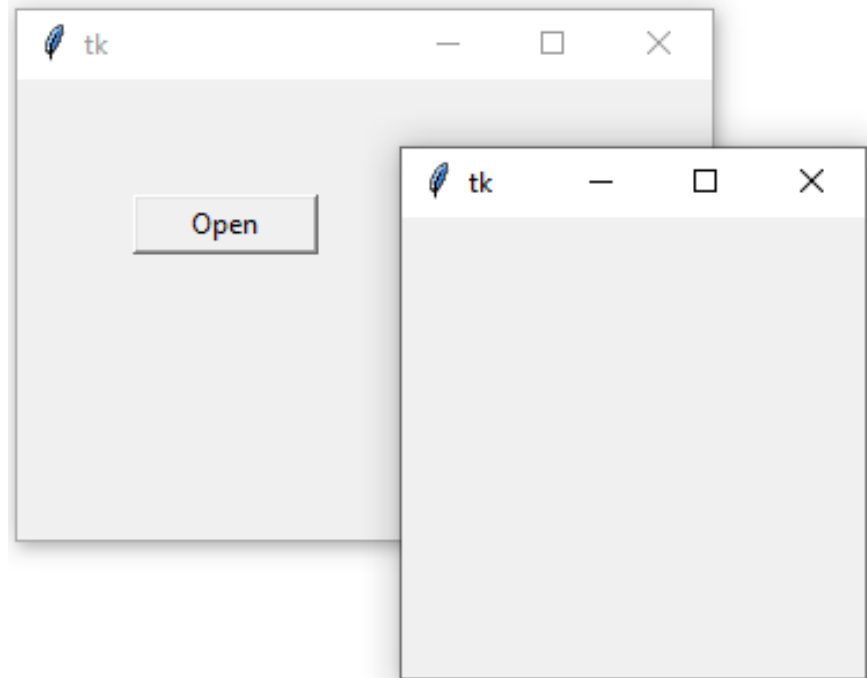
```
    chld.mainloop()
```

```
    btn1 = Button(top, text = "Open",width=10,command=fun)
```

```
    btn1.place(x=50,y=50)
```

```
    top.mainloop() Output:
```

```
>>>python topleveldemo.py
```



Example: simplecalc.py

import tkinter as

tk
from functools import partial

def call_result(label_result, n1, n2):

num1 = (n1.get())

num2 = (n2.get())

result = int(num1)+int(num2)

label_result.config(text="Result is %d" % result)

return

root = tk.Tk()

root.geometry('400x200+100+200')

root.title('Simple Calculator')

number1 = tk.StringVar()

number2 = tk.StringVar()


```
labelTitle = tk.Label(root, text="Simple Calculator").grid(row=0, column=2)
labelNum1 = tk.Label(root, text="Enter a number").grid(row=1, column=0)
labelNum2 = tk.Label(root, text="Enter another number").grid(row=2,
    column=0)
labelResult = tk.Label(root)
labelResult.grid(row=7, column=2)
entryNum1 = tk.Entry(root, textvariable=number1).grid(row=1, column=2)
entryNum2 = tk.Entry(root, textvariable=number2).grid(row=2, column=2)
call_result = partial(call_result, labelResult, number1, number2)
buttonCal = tk.Button(root, text="Calculate",
    command=call_result).grid(row=3, column=0)
root.mainloop()
```

Simple Calculator

Enter a number

Enter another number

Simple Calculator

Enter a number

Enter another number

Result is 66

❖ **Brief Tour of Other GUIs:**

- Python offers multiple options for developing GUI (Graphical User Interface). The most commonly used GUI methods are

1. Tix (Tk Interface eXtensions):

- Tix, which stands for Tk Interface Extension, is an extension library for Tcl/Tk. Tix adds many new widgets, image types and other commands that allows you to create compelling Tcl/Tk-based GUI applications.
- Tix includes the standard, widgets those are `tixGrid`, `tixHList`, `tixInputOnly`, `tixTlist` and etc.

2. Pmw (Python MegaWidgets Tkinter extension):

- Pmw is a toolkit for building high-level compound widgets in Python using the Tkinter module.
- It consists of a set of base classes and a library of flexible and extensible megawidgets built on this foundation. These megawidgets include notebooks, comboboxes, selection widgets, paned widgets, scrolled widgets and dialog windows.

3. wxPython (Python binding to wxWidgets):

- wxPython is a blending of the wxWidgets GUI classes and the Python programming language.
- wxPython is a Python package that can be imported at runtime that includes a collection of Python modules and an extension module (native code). It provides a series of Python classes that mirror (or shadow) many of the wxWidgets GUI classes.

UNIT – V

Database Programming

Introduction:

- To build the real world applications, connecting with the databases is the necessity for the programming languages. However, python allows us to connect our application to the databases like MySQL, SQLite, MongoDB, and many others.
- Python also supports Data Definition Language (DDL), Data Manipulation Language (DML) and Data Query Statements. For database programming, the Python **DB-API** is a widely used module that provides a database application programming interface.

The Python Programming language has powerful features for database programming, those are

- Python is famous for its portability.
- It is platform independent.
- In many programming languages, the application developer needs to take care of the open and closed connections of the database, to avoid further exceptions and errors. In Python, these connections are taken care of.
- Python supports relational database systems.
- Python database APIs are compatible with various databases, so it is very easy to migrate and port database application interfaces.

Environment Setup:

- In this topic we will discuss Python-MySQL database connectivity, and we will perform the database operations in python.
- The Python DB API implementation for MySQL is possible by **MySQLdb** or **mysql.connector**.

Note: The Python DB-API implementation for MySQL is possible by **MySQLdb** in **python2.x** but it deprecated in python3.x. In **Python3.x**, DB -API implementation for MySQL is possible by **mysql.connector**.

- ***You should have MySQL installed on your computer***

Windows:

You can download a free MySQL database at

<https://www.mysql.com/downloads/>.

Linux(Ubuntu):

sudo apt-get install mysql-server

- ***You need to install MySQLdb: (in case of Python2.x)***
- MySQLdb is an interface for connecting to a MySQL database server from Python. The **MySQLdb** is not a built-in module, We need to install it to get it working.
- Execute the following command to install it.
- For (Linux)Ubuntu, use the following command -
sudo apt-get install python2.7-mysqldb
- For Windows command prompt, use the following command -
pip install MySQL-python
- To test if the installation was successful, or if you already have "MySQLdb" installed, execute following python statement at terminal or CMD.
import MySQLdb
- If the above statement was executed with no errors, "MySQLdb " is installed and ready to be used.

(OR)

- ***You need to install mysql.connector: (in case of Python3.x)***
- To connect the python application with the MySQL database, we must import the **mysql.connector** module in the program.
- The **mysql.connector** is not a built-in module, We need to install it to get it working.
- Execute the following command to install it using pip installer.
- For (Linux)Ubuntu, use the following command -
pip install mysql-connector-python
- For Windows command prompt, use the following command -
pip install mysql-connector
- To test if the installation was successful, or if you already have "**mysql.connector**" installed, execute following python statement at terminal or CMD.
import mysql.connector
- If the above statement was executed with no errors, "**mysql.connector**" is installed and ready to be used.

Python Database Application Programmer's Interface (DB-API):

- Python DB-API is independent of any database engine, which enables you to write Python scripts to access any database engine.
- The Python DB API implementation for MySQL is possible by **MySQLdb** or **mysql.connector**.
- Using Python structure, **DB-API** provides standard and support for working with databases.

The API consists of:

1. Import module(**mysql.connector** or **MySQLdb**)
2. Create the connection object.
3. Create the cursor object
4. Execute the query
5. Close the connection

1. Import module(mysql.connector or MySQLdb):

- To interact with MySQL database using Python, you need first to import **mysql.connector** or **MySQLdb** module by using following statement.

- **MySQLdb(in python2.x)**

import MySQLdb

- **mysql.connector(in python3.x)**

import mysql.connector

2. Create the connection object:

- After importing **mysql.connector** or **MySQLdb** module, we need to create connection object, for that python DB-API supports one method i.e. **connect ()** method.
- It creates connection between MySQL database and Python Application.
- If you import **MySQLdb**(in python2.x) then we need to use following code to create connection.

Syntax:

Conn-name=MySQLdb.connect(<hostname>,<username>,<password>,<database>)

Example:

Myconn =MySQLdb.**connect** ("localhost","root","root","emp")

(Or)

- If you import **mysql.connector**(in python3.x) then we need to use following code to create connection.

Syntax:

*conn-name= mysql.connector.connect (host=<host-name>,
user=<username>,**passwd**=<pwd>,**database**=<dbname>)*

Example:

myconn=mysql.connector.connect(host="localhost",user="root",
passwd="root",database="emp")

3. Create the cursor object:

- After creation of connection object we need to create cursor object to execute SQL queries in MySQL database.
- The cursor object facilitates us to have multiple separate working environments through the same connection to the database.
- The Cursor object can be created by using **cursor ()** method.

Syntax:

cur_came = conn-name.cursor()

Example:

my_cur=myconn.**cursor()**

4. Execute the query:

- After creation of cursor object we need to execute required queries by using cursor object. To execute SQL queries, python DB-API supports following method i.e. **execute ()**.

Syntax:

cur-name.execute(query)

Example:

```
my_cur.execute ("select * from Employee")
```

5. Close the connection:

- After completion of all required queries we need to close the connection.

Syntax:

conn-name.close()

Example:

```
conn-name.close()
```

MySQLdb(in python2.x):

- MySQLdb is an interface for connecting to a MySQL database server from Python. The following are example programs demonstrate interactions with MySQL database using **MySQLdb** module.
- **Note** – Make sure you have root privileges of MySQL database to interact with database.i.e. Userid and password of MySQL database.
- We are going to perform the following operations on MySQL database.
 - Show databases
 - Create database
 - Create table
 - To insert data into table
 - Read/Select data from table
 - Update data in table
 - Delete data from table

Example Programs:

To display databases :

We can get the list of all the databases by using the following MySQL query.

>show databases;

Example: **showdb.py**

```
import MySQLdb  
  
#Create the connection object  
myconn = MySQLdb.connect("localhost","root","root")  
  
#creating the cursor object  
cur = myconn.cursor()  
  
#executing the query  
dbs = cur.execute("show databases")  
  
#display the result  
for x in cur:  
    print(x)  
  
#close the connection  
myconn.close()
```

Output:

```
>>>python showdb.py  
(information_schema',)  
(mysql',)  
(performance_schema',)  
(phpmyadmin',)  
(test',)  
(Sampledb',)
```

To Create database :

The new database can be created by using the following SQL query.

> create database <database-name>

Example: **createdb.py**

```
import MySQLdb
#Create the connection object
myconn = MySQLdb.connect("localhost","root","root")
#creating the cursor object
cur = myconn.cursor()
#executing the query
cur.execute("create database Collegedb")
print("Database created successfully")
#close the connection
myconn.close()
```

Output:

>>>python **createdb.py**

Database created successfully

```
MariaDB [(none)]> show databases;
+-----+
| Database |
+-----+
| collegedb |
| information_schema |
| mysql |
| performance_schema |
| phpmyadmin |
| test |
+-----+
6 rows in set (0.00 sec)
```


To Create table :

The new table can be created by using the following SQL query.

> *create table <table-name> (column-name1 datatype, column-name2 datatype,...)*

Example: **createtable.py**

```
import MySQLdb
```

```
#Create the connection object
```

```
myconn = MySQLdb.connect("localhost","root","root","Colleged")
```

```
#creating the cursor object
```

```
cur = myconn.cursor()
```

```
#executing the query
```

```
cur.execute("create table students(sid varchar(20)primary key,sname varchar(25),age int(10))")
```

```
print("Table created successfully")
```

```
#close the connection
```

```
myconn.close()
```

Output:

```
>>>python createtable.py
```

```
Table created successfully
```

```
MariaDB [(none)]> use Collegedb
Database changed
MariaDB [Collegedb]> show tables;
+-----+
| Tables_in_collegedb |
+-----+
| students             |
+-----+
1 row in set (0.00 sec)

MariaDB [Collegedb]>
```

To Insert data into table :

The data can be inserted into table by using the following SQL query.

> insert into <table-name> values (value1, value2,...)

Example: **insertdata.py**

```
import MySQLdb
#Create the connection object
myconn = MySQLdb.connect("localhost","root","root","Collegedb")
#creating the cursor object
cur = myconn.cursor()
#executing the query
cur.execute("INSERT INTO students VALUES ('501', 'ABC', 23)")
cur.execute("INSERT INTO students VALUES ('502', 'XYZ', 22)")
#commit the transaction
myconn.commit()
print("Data inserted successfully")
#close the connection
myconn.close()
```

Output:

```
>>>python insertdata.py
Data inserted successfully
```

```
MariaDB [Collegedb]> select * from students;
+----+-----+-----+
| sid | sname | age  |
+----+-----+-----+
| 501 | ABC   | 23   |
| 502 | XYZ   | 22   |
+----+-----+-----+
2 rows in set (0.00 sec)
```

To Read/Select data from table ::

The data can be read/select data from table by using the following SQL query.

>select column-names from <table-name>

Example: **selectdata.py**

fetchall() method returns all rows in the table.
fetchone() method returns one row from table.

```
import MySQLdb
```

```
#Create the connection object
```

```
myconn = MySQLdb.connect("localhost","root","root","Colleged")
```

```
#creating the cursor object
```

```
cur = myconn.cursor()
```

```
#executing the query
```

```
cur.execute("select * from students")
```

```
#fetching all the rows from the cursor object
```

```
result = cur.fetchall()
```

```
print("Student Details are :")
```

```
#printing the result
```

```
for x in result:
```

```
    print(x);
```

```
#close the connection
```

```
myconn.close()
```

Output:

```
>>>python selectdata.py
```

Student Details are:

('501', 'ABC', 23)

('502', 'XYZ', 22)

fetchall() method returns all rows in the table.
fetchone() method returns one row from table.

Example: **selectone.py**

```
import MySQLdb

#Create the connection object
myconn = MySQLdb.connect("localhost","root","root","Colleged")
#creating the cursor object
cur = myconn.cursor()
#executing the query
cur.execute("select * from students")
#fetching all the rows from the cursor object
result = cur.fetchone()
print("One student Details are :")
#printing the result
print(result)
#close the connection
myconn.close()
```

Output:

```
>>>python selectone.py
```

One student Details are:
('501', 'ABC', 23)

To Update data into table :

The data can be updated in table by using the following SQL query.

> update <table-name> set column-name=value where condition

Example: **updatedata.py**

```
import MySQLdb
```

```
#Create the connection object
```

```
myconn = MySQLdb.connect("localhost","root","root","Collegedb")
```

```
#creating the cursor object
```

```
cur = myconn.cursor()
```

```
#executing the query
```

```
cur.execute("update students set sname='Kumar' where sid='502'")
```

```
#commit the transaction
```

```
myconn.commit()
```

```
print("Data updated successfully")
```

```
#close the connection
```

```
myconn.close()
```

Output:

```
>>>python updatedata.py
```

```
Data updated successfully
```

```
MariaDB [Collegedb]> select * from students;
+----+-----+-----+
| sid | sname | age |
+----+-----+-----+
| 501 | ABC   | 23  |
| 502 | Kumar | 22  |
+----+-----+-----+
2 rows in set (0.00 sec)
```

To Delete data from table :

The data can be deleted from table by using the following SQL query.

> delete from <table-name> where condition

Example: **deletedata.py**

```
import MySQLdb

#Create the connection object
myconn = MySQLdb.connect("localhost","root","root","Collegedb")

#creating the cursor object
cur = myconn.cursor()

#executing the query
cur.execute("delete from students where sid='502'")

#commit the transaction
myconn.commit()

print("Data deleted successfully")

#close the connection
myconn.close()
```

Output:

```
>>>python deletedata.py
Data deleted successfully
```

```
MariaDB [Collegedb]> select * from students;
+-----+-----+-----+
| sid | sname | age |
+-----+-----+-----+
| 501 | ABC   | 23  |
+-----+-----+-----+
1 row in set (0.00 sec)
```

DB-API for MySQL in Python

MySQLdb (**python2.x**)

```
#Import MySQLdb
import MySQLdb
#Create the connection object
myconn =MySQLdb.connect
("localhost","root","root","Colleged")
```

Mysql.connector(**python3.x**)

```
#Import mysql.connector
import mysql.connector
#Create the connection object
myconn=mysql.connector.connect
(host="localhost",user="root",
passwd="root",database="Colleged")
```

mysql.connector(in python3.x)::

MySQL Connector enables Python programs to access MySQL databases.

Example: **deletedata.py**

```
import mysql.connector
#Create the connection object
myconn=mysql.connector.connect(host="localhost",user="root",passwd="root",
database="Collegedb")
#creating the cursor object
cur = myconn.cursor()
#executing the querys
cur.execute("delete from students where sid='502'")
#commit the transaction
myconn.commit()
print("Data deleted successfully")
#close the connection
myconn.close()
```

Output:

```
>>>python deletedata.py
Data deleted successfully
```

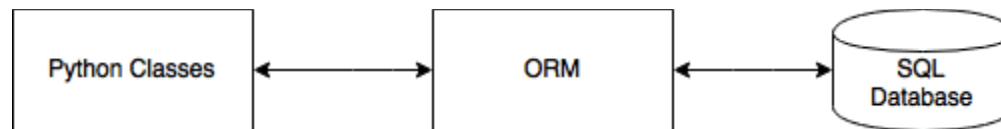
```
MariaDB [Collegedb]> select * from students;
+-----+-----+-----+
| sid | sname | age |
+-----+-----+-----+
| 501 | ABC   | 23  |
+-----+-----+-----+
1 row in set (0.00 sec)
```


Object **Relational **M**apping (ORM)**

in python

Introduction:

- **Object Relational Mapping** is a system of mapping objects to a database. That means it automates the transfer of data stored in relational databases tables into objects that are commonly used in application code.
- An object relational mapper maps a relational database system to objects. The ORM is independent of which relational database system is used. From within Python, you can talk to objects and the ORM will map it to the database.



- **ORMs** provide a high-level abstraction upon a [relational database](#) that allows a developer to write Python code instead of SQL to interact (create, read, update and delete data and schemas) with database.

The mapping like this...

- Python Class == SQL Table
 - Instance of the Class == Row in the Table
-
- Developers can use the programming language they are comfortable with to work with a database instead of writing SQL statements or stored procedures.
 - There are many ORM implementations written in Python, including
 - **SQLAlchemy**
 - Peewee
 - The Django ORM
 - PonyORM
 - SQLAlchemy
 - Tortoise ORM
 - We are going to discuss about **SQLAlchemy**, it pronounced as **SQL-All-Chemy**.

SQLAlchemy:

- **SQLAlchemy** is a library used to interact with a wide variety of databases. It enables you to create data models and queries in a manner that feels like normal Python classes and statements.
- It can be used to connect to most common databases such as Postgres, MySQL, SQLite, Oracle, and many others.
- **SQLAlchemy** is a popular SQL toolkit and Object Relational Mapper. It is written in Python and gives full power and flexibility of SQL to an application developer.
- It is necessary to install SQLAlchemy. To install we have to use following code at Terminal or CMD.
pip install sqlalchemy
- To check if SQLAlchemy is properly installed or not, enter the following command in the Python prompt
>>>import sqlalchemy
- If the above statement was executed with no errors, “sqlalchemy ” is installed and ready to be used.

Connecting to Database:

- To connect with database using SQLAlchemy, we have to create engine for this purpose SQLAlchemy supports one function is **create_engine()**.
- The **create_engine()** function is used to create engine; it takes overall responsibilities of database connectivity.

Syntax:

Database-server[+driver]://user:password@host/dbname

Example:

mysql+mysqldb://root:root@localhost/collegedb

- The main objective of the ORM-API of SQLAlchemy is to facilitate associating user-defined Python classes with database tables, and objects of those classes with rows in their corresponding tables.

Declare Mapping:

- First of all, `create_engine()` function is called to set up an engine object which is subsequently used to perform SQL operations.

To create engine in case of MySQL:

Example:

```
from sqlalchemy import create_engine  
engine = create_engine('mysql+mysqldb://root:@localhost/Collegedb')
```

- When using ORM, we first configure database tables that we will be using. Then we define classes that will be mapped to them. Modern SQLAlchemy uses *Declarative* system to do these tasks.
- A *declarative base class* is created, which maintains a catalog of classes and tables. A declarative base class is created with the `declarative_base()` function.
- The `declarative_base()` function is used to create base class. This function is defined in ***sqlalchemy.ext.declarative*** module.

To create declarative base class:

Example:

```
from sqlalchemy.ext.declarative import declarative_base  
Base = declarative_base()
```

Example: **tabledef.py**

```
from sqlalchemy import Column, Integer, String
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
# create a engine
engine =create_engine('mysql+mysqldb://root:@localhost/Sampledbs')
# create a declarative base class
Base = declarative_base()

class Students(Base):
    __tablename__= 'students'
    id = Column(Integer, primary_key=True)
    name = Column(String(10))
    address = Column(String(10))
    email = Column(String(10))

Base.metadata.create_all(engine)
```