OBJECT ORIENTED PROGRAMMING THROUGH PYHTON

B.Tech III Sem (IARE-R18)

COMPUTER SCIENCE AND ENGINEERING
INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
DUNDIGAL, HYDERABAD - 500 043

Introduction to Python

- Python is a general-purpose, dynamic, interpreted high-level programming language.
- Conceptualized in the late 1980's.
- Created by Guido van Rossum(Netherlands) and first released in 1991.
- A descendant of ABC language.
- Open sourced from the beginning, managed by Python Software Foundation.
- Scalable, Object oriented and functional from the beginning.
- Python versions
  - ➢First version 0.9.0 in February 1991
  - ➢Version 1.0 in January 1994
  - ➢ Version 2.0 in October 2000
  - ➢Version 3.0 in 2008

Course outcome / Topic learning outcome

List the course outcome / Topic outcome

Name of the Topic covered	Topic Learning Outcome	Course Outcome
Features of python	The basic and advanced constructs of Python programming for developing object oriented concepts	Recall the basic programming constructs in implementing in Python.

Outcome achieved

Name of the topic: Features of python

Students will be able to do:	
1	Recall the basic programming constructs in implementing in Python.

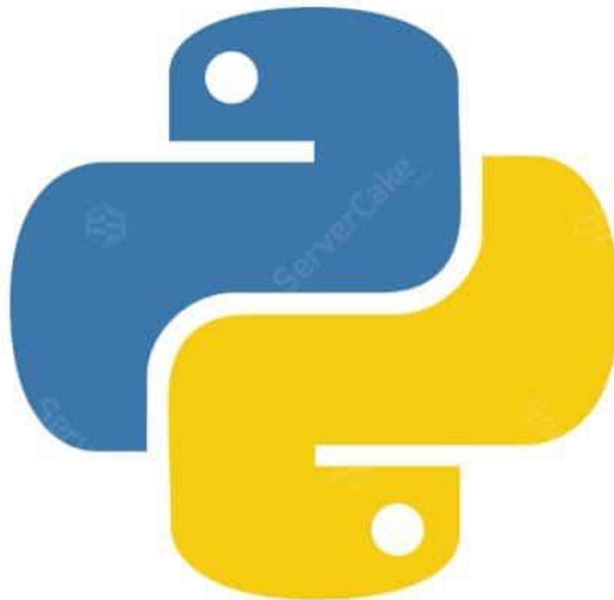
MODULE –I
INTRODUCTION TO PYTHON AND OBJECT ORIENTED CONCEPTS

Contents

- Features of Python
- Data types
- Operators in python
- Input and output
- Control Statements
- Features of object oriented programming system
- Classes and Objects
- Encapsulation
- Inheritance
- Abstraction
- Polymorphism

Guido Van Rossum

DID YOU KNOW?

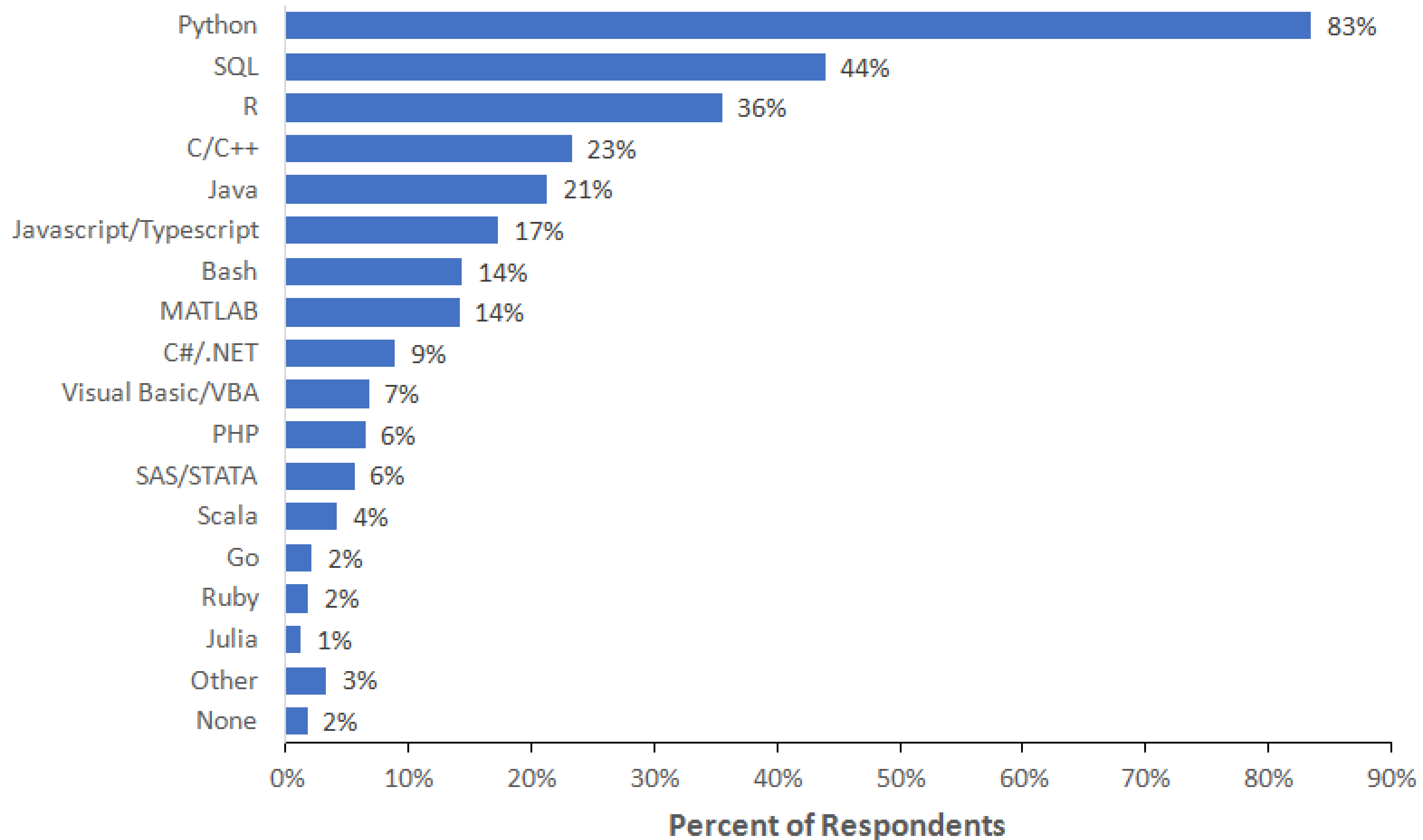


Guido Van Rossum is a Dutch programmer who is best known as the author of the Python Programming Language

Brief History of Python Language

- Python is a general-purpose, dynamic, interpreted high-level programming language.
- Conceptualized in the late 1980's.
- Created by Guido van Rossum(Netherlands) and first released in 1991.
- A descendant of ABC language.
- Open sourced from the beginning, managed by Python Software Foundation.
- Scalable, Object oriented and functional from the beginning.
- Python versions
  - First version 0.9.0 in February 1991
  - Version 1.0 in January 1994
  - Version 2.0 in October 2000
  - Version 3.0 in 2008

Best Programming Language



Features of Python Language

- Simple
- Easy to learn
- Open source
- High level language
- Dynamically typed
- Platform independent
- Portable
- Procedure and object oriented

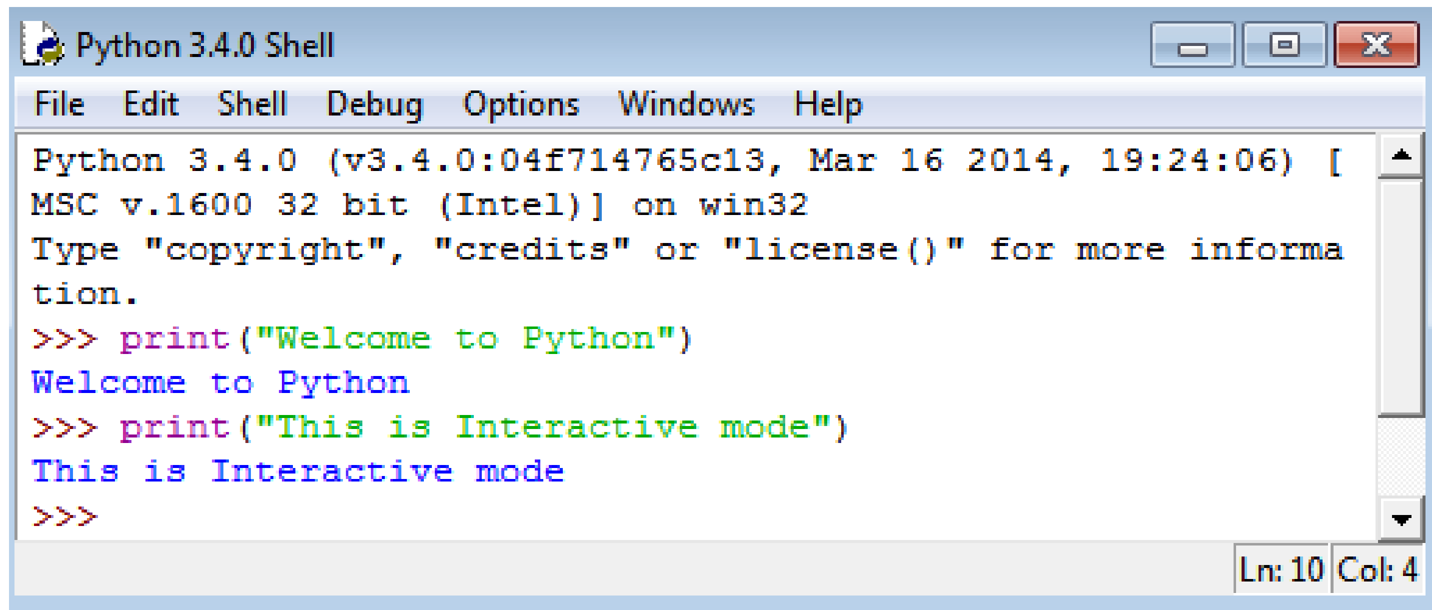
Python Interactive Shell

IARE
EDUCATION FOR LIBERATION
2000

- Python provides an interactive shell, which is used in between the user and operating system
- In other words, Python provides a command line interface with the Python shell known as Python interactive shell.
- Python commands are run using the Python interactive shell.
- User can work with Python shell in two modes: interactive mode and script mode.
- Interactive mode allows the user to interact with the operating system. When the user types any Python statement / expression, the interpreter displays the results instantly.
- In script mode, user types a Python program in a file and then uses the interpreter to execute the file. In interactive mode, user can't save the statements / expressions and need to retype once again to re-run them.

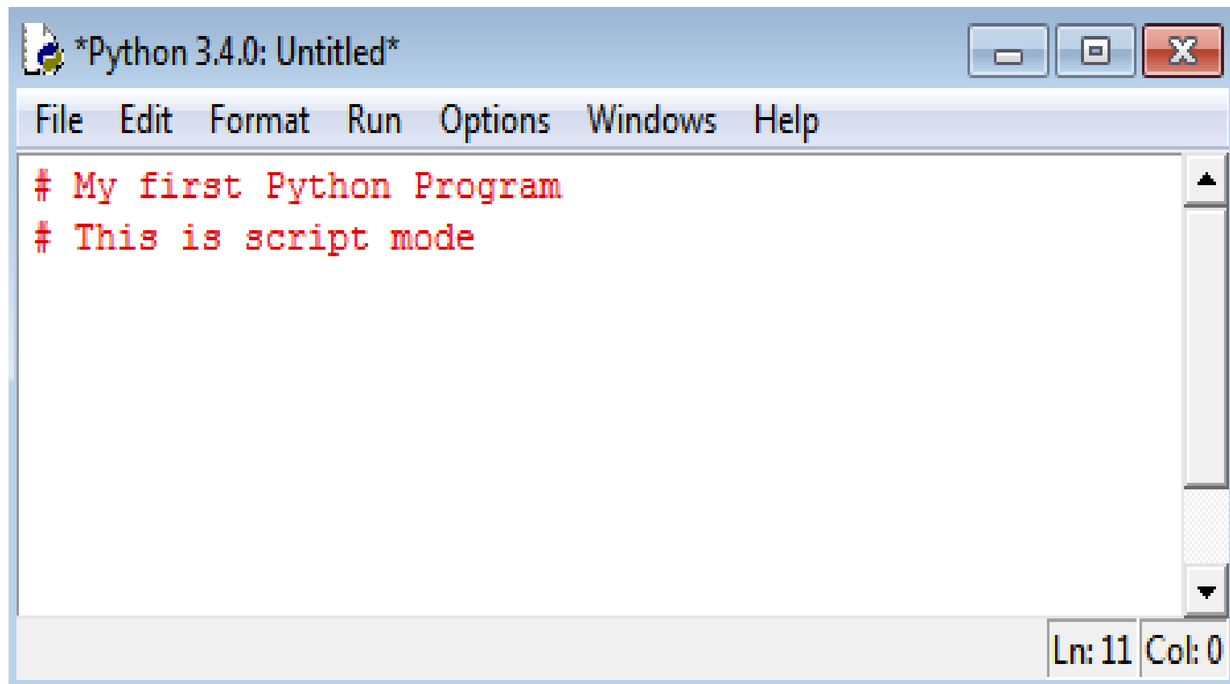
Interactive Mode

- When the user starts the Python IDLE the following window will appear and it shows the interactive shell. This window shows the primary prompt '>>>' where the user types commands to run by the interpreter.



Script Mode

- In this mode, user types a set of statements called a program in a file and then save the program with ‘filename.py’ as extension. Then the interpreter is used to execute the file contents. This mode is convenient when the user wants to write and save multiple lines of code, so that it can be easily modifiable and reusable.



Python Shell as a Simple Calculator

2000 IARE EDUCATION FOR LIBERATION

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # division always returns a floating point number
1.6
```

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # result * divisor + remainder
17
```

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

Flavors of Python

- Flavors of Python are nothing but different types of Python compilers available, which are useful to integrate various programming languages into Python. The following are some of the important and popularly used flavors of Python.
  - Cpython
  - Jython
  - IronPython
  - Pypy
  - Pythonxy
  - RubyPython
  - StacklessPython
  - ActivePython

Built-in Data Types in Python

- Every programming language has the ability to create and manipulate object / variable. In a program variables are used to store values so that it can be used later. Every object / variable has an identity, type and a value which it refers. Identity of an object is nothing but its address in memory when it is created. Type or data type indicates is a range of values and operations allowed on those values.

Keywords in Python

- Keywords are reserved words with predefined meaning in any programming languages and these words can’t be used as normal variables. One can check the number of keywords using help() command -> keywords in Python.

```
help> keywords

Here is a list of the Python keywords.  Enter any keyword to get mo
re help.

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

Assigning values to variables

```
>>> a = 100                # a is integer
>>> height = 50.5          #height is float
>>> player = "Sachin"      #player is string
>>> x = y = z = 10         # This statement assign 10 to x, y, z
>>> x = 5
>>> x                      #assigns 5 to x
>>> 5  = x                 #SyntaxError: can't assign to literal
```

Multiple Assignments

- Consider an example where multiple values are assigned to the same variable and when the program runs, it prints different results.

```
x = 5
print ('x = ' + x)
x = 10
print ('x = ' + x)
x = 15
print ('x = ' + x)
```

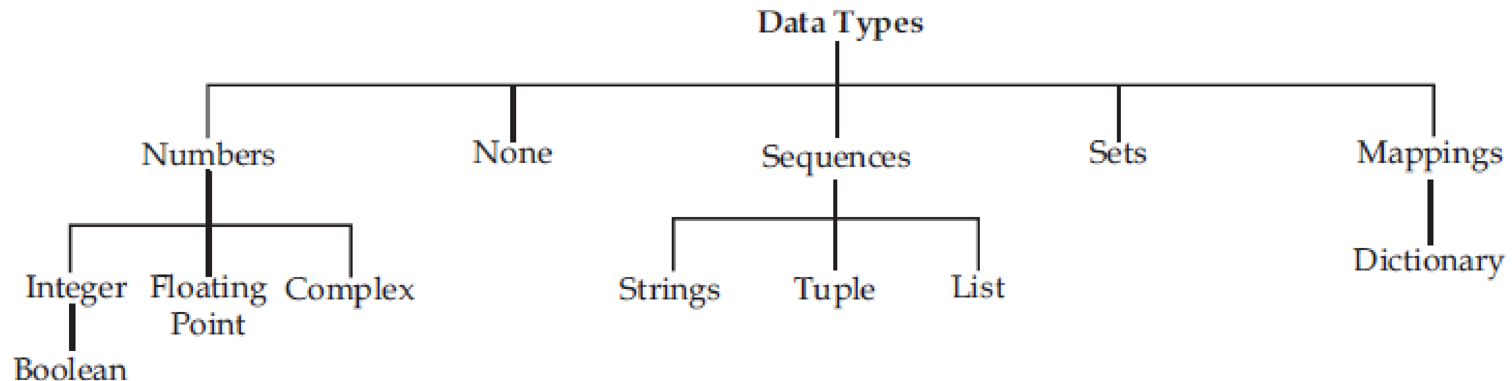
```
x = 5
x = 10
x = 15
```

```
x = 5
print ('x = ' + str(x))
x = 10
print ('x = ' + str(x))
x = 15
print ('x = ' + str(x))
```

```
x = 5
x = 10
x = 15
```

Standard Data Types in Python

- Python has five standard data types, named Numbers, None, Sequences, Sets and Mappings. Python sets the type of variable based on the type of value assigned to it and it will automatically change the variable type if the variable is set to some other value.



Numbers

Python supports the following numeric types.

- **int** - integers of unlimited length in Python 3.x .
- **long** - long integers of unlimited length, but exists only in Python 2.x.
- **float** - floating point numbers.
- **complex** - complex numbers.

```
>>> a = 10
>>> type(a)
<class 'int'>
>>> b = 125.50
>>> type(b)
<class 'float'>
>>> c = 5 + 6j
>>> type(c)
<class 'complex'>
>>> str1 = "Welcome to Python"
>>> type(str1)
<class 'str'>
```

Boolean

- True and False are Boolean literals used in Python and these are used to represent the truth / falsity of any condition / expression.

```
>>> x = True
>>> type(x)
<class 'bool'>
>>> y = (3 > 5)
>>> type(y)
<class 'bool'>
>>> y
False
>>> x
True
```

None

- In Python None keyword is an object which is equivalent to Null. A None can be assigned to a variable during declaration or while evaluating an expression.

```
>>> var = None
>>> type(var)
<class 'NoneType'>
```

Strings

- Strings are identified as group of characters represented in quotation marks. Python allows both a pair of single and double quotes for writing strings. Strings written in triple quotes can span multiple lines of text. Strings in Python are immutable data type i.e. each time a new string object is created when one makes any changes to a string.

```
>>> s1 = 'Hello Python'
>>> s1
'Hello Python'
>>> s2 = "Welcome"
>>> s2
'Welcome'
>>> s3 = s1[0]     #output will be first character
>>> s3
'H'
>>> s4 = s1[0:5]   #output will be first five characters
>>> s4
'Hello'
```

Strings

- Python can also manipulate strings. They can be enclosed in single quotes (‘abc’) or double quotes (“abc”) with the same result.

```
>>> "welcome to Python"
'welcome to Python'
>>> 'Enjoy learning'
'Enjoy learning'
>>> s = 'Beautiful Language'
>>> s
'Beautiful Language'
>>> #Strings can be concatenated with the + operator and repeated with *
>>> #print 3 times hello python
>>> 3 * 'hello' + 'Python'
'hellohellohelloPython'
>>> 'Hello' 'Python'   #another way of concatenation
'HelloPython'
>>> #break long strings
>>> line = ('This is the first line'
            'This is the second line'
            'This is the third line')
>>> line
'This is the first lineThis is the second lineThis is the third line'
>>> #concatenate a variable and a literal using + operator
>>> prefix = 'Py'
>>> prefix + 'thon'
'Python'
```

Tuple

- A tuple contains a list of items enclosed in parentheses and none of the items cannot be updated. Hence tuples are immutable.

```
>>> tuple1 = (100, 200, 'hello', 456.789)
>>> tuple2 = ('Hello', 'World')
>>> tuple1
(100, 200, 'hello', 456.789)
>>> tuple2
('Hello', 'World')
>>> tuple1[0]      #gives first value in the tuple
100
>>> tuple1 + tuple2    #combines both the tuples
(100, 200, 'hello', 456.789, 'Hello', 'World')
>>> tuple1[1] = 300
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    tuple1[1] = 300
TypeError: 'tuple' object does not support item assignment
```

List

- A list contains items separated by commas and enclosed within square brackets. A list in Python can contain heterogeneous data types.

```
>>> list1 = [100, 'happy', 123.456, 'A']
>>> list1
[100, 'happy', 123.456, 'A']
>>> list2 = [10, 20, 30, 40 , 50]
>>> list2
[10, 20, 30, 40, 50]
>>> list3 = ['Hello', 'Python']
>>> list3
['Hello', 'Python']
>>> list1[0:2]     #outputs first two elements of list
[100, 'happy']
>>> list3 * 2      #outputs list3 two times
['Hello', 'Python', 'Hello', 'Python']
>>> list1 + list3  #combines both the lists
[100, 'happy', 123.456, 'A', 'Hello', 'Python']
```

Sets

- In Python sets are unordered collection of objects enclosed in parenthesis and there are basically two types of sets:
- ➢ Sets - These are mutable and can be updated with new elements once sets are defined.
- ➢ Frozen Sets - These are immutable and cannot be updated with new elements once frozen sets are created.

```
>>> flowers = {'Rose', 'Jasmine', 'Rose', 'Lily', 'Rose', 'Jasmine'}
>>> flowers      #eliminates duplicates
{'Lily', 'Jasmine', 'Rose'}
>>> set1 = set('Welcome')
>>> set1         #prints unique letters in set1
{'m', 'c', 'W', 'e', 'l', 'o'}
>>> set1.add('z')  #adds a new element to a set
>>> set1
{'m', 'z', 'c', 'W', 'e', 'l', 'o'}

>>> set2 = frozenset('Welcome')
>>> set2
frozenset({'m', 'c', 'W', 'e', 'l', 'o'})
>>> cities = frozenset(["Delhi", "Hyderabad", "Mumbai"])
>>> cities
frozenset({'Hyderabad', 'Delhi', 'Mumbai'})
>>> set2.add('z')
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    set2.add('z')
```

Dictionary

- In Python dictionary data type consists of key-value pairs and it is enclosed by curly braces. Values can be assigned and accessed using square brackets.

```
>>> dict1 = {'Name':'Happy', 'age': 25}
>>> dict1
{'Name': 'Happy', 'age': 25}
>>> dict1['Name']       #gives the value associated with the key
'Happy'
>>> dict1.values()
dict_values(['Happy', 25])
>>> dict1.keys()
dict_keys(['Name', 'age'])
```

Mutable and Immutable Data Types

- The following table gives examples of mutable and immutable data types in Python.

Mutable Data Types	Immutable Data Types
list	int, long
set	float, complex
dict	str
	tuple
	frozenset

Operators in Python

All the operators in Python are classified according to their nature and type and they are:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Bitwise Operators
- Boolean Operators
- Membership Operators
- Identity Operators

Arithmetic Operators

- These operators perform basic arithmetic operations like addition, subtraction, multiplication, division etc. and these operators are binary operators that means these operators acts on two operands. And there are 7 binary arithmetic operators available in Python.

Operator	Meaning	Example	Result
+	Addition	10 + 7	12
-	Subtraction	10.0 - 1.5	8.5
*	Multiplication	30 * 3	900
/	Float Division	5 / 2	2.5
//	Integer Division	5 // 2	2
**	Exponentiation	3 ** 2	9
%	Remainder	10 % 3	1

Operator	Priority
Parenthesis (( ), [ ])	First
Exponentiation (**)	Second
Multiplication (*), Division (/, //), Modulus (%)	Third
Addition (+), Subtraction (-)	Fourth
Assignment	Fifth

Relational Operators

- Relational operators are used for comparison and the output is either True or False depending on the values we compare. The following table shows the list of relational operators with example.

Operator	Meaning	Example	Result
<	Less than	5 < 7	True
>	Greater than	9 > 5	True
<=	Less than equal to	8 <= 8	True
>=	Greater than equal to	7 >= 9	False
==	Equal to	10 == 20	False
!=	Not equal to	9 != 6	True

Logical Operators

- Logical operators are used to form compound conditions which are a combination of more than one simple condition. Each of the simple conditions are evaluated first and based on the result compound condition is evaluated. The result of the expression is either True or False based on the result of simple conditions.

Operator	Meaning	Example	Result
and	Logical AND	(5 > 7) and (3 < 5)	False
or	Logical OR	(7 == 7) or (5 != 5)	True
not	Logical NOT	not(3 <= 2)	True

```
>>> (5 > 7) and (3 < 5)
False
>>> (7 == 7) or (5 != 5)
True
>>> not(3 <=2)
True
```

Assignment Operators

- These operators are used to store a value into a variable and also useful to perform simple arithmetic operations. Assignment operators are of two types: simple assignment operator and augmented assignment operator. Simple assignment operators are combined with arithmetic operators to form augmented assignment operators. The following table shows a list of assignment operators and its use.

Operator	Meaning	Example	Result
=	Simple assignment	a = 10	10
+=	Addition assignment	a = 5 a += 8	13
-=	Subtraction assignment	b = 5 b -= 8	-3
*=	Multiplication assignment	a =10 a *= 8	80
/=	Float Division assignment	a = 10 a /= 8	1.25
//=	Integer Division assignment	b = 10 b //= 10	1
**=	Exponentiation assignment	a = 10 a %= 5	0
%=	Remainder assignment	b = 10 b **= 8	100000000

Bitwise Operators

2000 IARE EDUCATION FOR LIBERATION

- Bitwise Operators acts on individual bits of the operands. These operators directly act on binary numbers. If we want to use these operators on integers then first these numbers are converted into binary numbers and then bitwise operators act on those bits. The following table shows the list of bitwise operators available in Python.

Operator	Meaning	Example	Result
&	Bitwise AND	a = 10 = 0000 1010 b = 11 = 0000 1011 a & b = 0000 1010 = 10	a & b = 10
|	Bitwise OR	a = 10 = 0000 1010 b = 11 = 0000 1011 a | b = 0000 1011 = 11	a | b = 11
^	Bitwise XOR	a = 10 = 0000 1010 b = 11 = 0000 1011 a ^ b = 0000 0001 = 1	a ^ b = 1
~	Bitwise Complement	a = 10 = 0000 1010 ~a = 1111 0101 = -11	~a = -11
<<	Bitwise Left Shift	a = 10 a << 2 = 40	a << 2 = 40
>>	Bitwise Right Shift	a = 10 a >> 2 = 2	a >> 2 = 2

Boolean Operators

2000 IARE EDUCATION FOR LIBERATION

- There are three boolean operators that act on bool type literals and provide bool type output. The result of the boolean operators are either True or False.

Operator	Meaning	Example	Result
and	Boolean AND	a = True, b = False a and b = True and False	a and b = False
or	Boolean OR	a = True, b = False a or b = True or False	a or b = True
not	Boolean NOT	a = True not a = not True	not a = False

Membership Operators

There are two membership operators in Python that are useful to test for membership in a sequence.

- **in**: This operator returns True if an element is found in the specified sequence, otherwise it returns False.
- **not in: This** operator returns True if any element is not found in the sequence, otherwise it returns True.

Identity Operators

These operators are used to compare the memory locations of two objects. Therefore it is possible to verify whether the two objects are same or not. In Python id() function gives the memory location of an object. Example id(a) returns the identity number or memory location of object a. There are two identity operators available in Python. They are

- **is:** This operator is used to compare the memory location of two objects. If they are same then it returns True, otherwise returns False.
- **is not:** This operator returns True if the memory locations of two objects are not same.If they are same then it returns False.

```
# Identity Operators
a = 100
print("Identity Number of a =  ", id(a))
b = 200
print("Identity Number of b =  ", id(b))
if a is b:
    print(" a and b have same identity")
else:
    print("a and b have different identity")


"""
Identity Number of a =   1528939480
Identity Number of b =   1528941080
a and b have different identity

"""
```

Operator Precedence and Associativity

- An expression may contain several operators and the order in which these operators are executed in sequence is called operator precedence. The following table summarizes the operators in descending order of their precedence.

Operator	Name	Precedence
( )	Parenthesis	1st
**	Exponentiation	2nd
-, ~	Unary minus, bitwise complement	3rd
*, /, //, %	Multiplication, Division, Floor Division, Modulus	4th
+, -	Addition, Subtraction	5th
<<, >>	Bitwise left shift, bitwise right shift	6th
&	Bitwise AND	7th
^	Bitwise XOR	8th
|	Bitwise OR	9th
>, >=, <, <=, = =, !=	Relational Operators	10th
=, %=, /=, //=, -=, +=, *=, **=	Assignment Operators	11th
is, is not	Identity Operators	12th
in, not in	Membership Operators	13th
not	Logical NOT	14th
or	Logical OR	15th
and	Logical AND	16th

Single Line and Multiline Comments

- There are two types of comments used in Python:
- **Single Line Comments**: These are created simply by starting a line with the hash character (#), and they are automatically terminated by the end of line. If a line using the hash character (#) is written after the Python statement, then it is known as inline comment.

- **Multiline Comments**: When multiple lines are used as comment lines, then writing hash character (#) in the beginning of every line is a tedious task. So instead of writing # character in the beginning of every line, we can enclose multiple comment lines within ''' (triple single quotes) or """ (triple double quotes). Multi line comments are also known as block comments.

INPUT AND OUTPUT

- The purpose of a computer is to process data and return results.The data given to the computer is called input. The results returned by the computer are called output. So, we can say that a computer takes input, processes that input and produces the output.

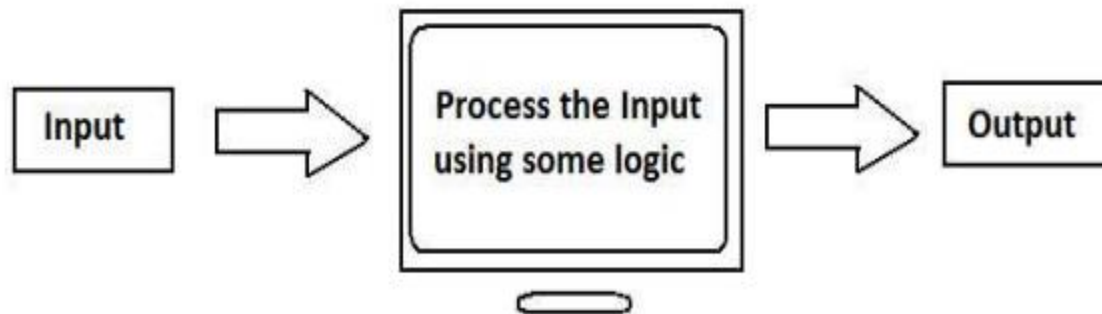


Figure 5.1: Processing Input by the Computer

Control Structures

A control structure is a block of programming that analyzes variables and decides which statement to execute next, based on the given parameters. The term ‘control’ denotes the direction in which the program flows. Usually, loops are used to execute a control statement, a certain number of times.

Basically, control structures determine the flow of events in the program.

If statement: This is used to check a condition and executes the operations/statements within the if block only when the given condition is true.

Syntax:

```
if condition:
    True Statements
```

If…else statements

If…else statements: These statements are used to check a condition and executes the operations/statements within the if block only when the given condition is true. If the given condition is false, the statements in the else block will be executed.

Syntax:

```
if condition:
    True Statements
else:
    False Statements
```

If …elif… else statements

If …elif… else statements: If we want to check more than one condition we can use the elif statements. If a condition is true then the statements within the if block will be executed. If the condition is false, we can provide an elif statement with a second condition and the statements within the elif block will be executed only when the condition is true. We can provide multiple elif statements and an else statement at the end if all the above conditions are false.

Syntax:

```
if condition:
   True Statements
elif condition2:
   True Statements
 elif condition3:
   True Statements
…….
else:
   False Statements
```

Loops in Python

- Loops are used to repeat a set of statements/single statement, a certain number of times. In Python, there are two loops, for loop and while loop. The Python for loop also works as an iterator to iterate over items in list/dictionary or characters in strings.

for Loop: It can be used to iterate over a list/string/dictionary or iterate over a range of numbers.

Syntax:

```
for variable in range(starting number , ending number + 1 , step size):
    statements
 (or)
 for element in sequence:
    statements
```

While Loop

While Loop: This is used, whenever a set of statements should be repeated based on a condition. The control comes out of the loop when the condition is false. In while loop we must explicitly increment/decrement the loop variable (if any) whereas in for, the range function would automatically increment the loop variable.

Syntax:

```
while condition:
    statement(s)

    increment/decrement
```

Break and Continue Statement

break statement: This statement is used to terminate the loop it is present in. Control goes outside the loop it is present in. If a break statement is present in a nested loop, it only comes out of the innermost loop.

Syntax:

```
while condition:
    statments
    if condition:
        break
    statements
```

Break and Continue Statement

Continue statement: This statement is used to skip the current iteration. The loop will not be terminated, it just won't execute the statements below the continue statement. The incrementing will be done in for loop. If the increment statement is written below continue, it won't be executed in while loop.

Syntax:

```
while condition:
    statement(s)
    if condition:
        continue
    statements
```

Pass Statement

Pass statement: This statement is used as placeholder. For example, we want to create a function but are not sure of its content. If we create a function and leave it, an error will occur. To counter this error, we use pass statement.

Syntax:

```
def function(parameters):
    pass
(or)
for elements in sequence:
    pass
(or)
while condition:
    pass
(or)
if condition:
    pass
```

Object Oriented Concepts

- Object oriented programming concept is associated with the concept of class, objects and various other concepts like abstraction, inheritance, polymorphism, encapsulation etc.
- **Class**: - Class is a user defined data type. It is a set of attributes (variables) and methods (functions). It is created using the keyword 'class'.
- **Object:** - Object is a unique instance of a class. We can use the same class as blueprint for creating number of different objects. The class describes what the object will be.
- **Attributes**: - Attributes are the member variables defined inside a class and can be accessed by the objects by using dot operator.
- **Method**: - Methods are functions defined inside a class. They can be accessed by the objects by using dot operator. All the methods in class have self as first parameter.

Example

```
class Car:
    #constructor method where attributes are defined
    def __init__(self):
        self.speed=50
        self.color="white"
        self.modelno=1795
    def accelerate(self):
        self.speed+=5
    def paint(self,newcolor):
        self.color=newcolor
    def speed_down(self):
        self.speed-=1
    def brake(self):
        self.speed=0
    def get_speed(self):
        return self.speed

bmw=Car()
print('Initial speed',end='=')
print(bmw.speed)
bmw.accelerate()
print('Speed after accelerating',end='=')
print(bmw.speed)
print('Initial color',end='=')
print(bmw.color)
bmw.paint('red')
print('Color after painting',end='=')
print(bmw.color)
```

```
Initial speed=50
Speed after accelerating=55
Initial color=white
Color after painting=red
```

__init__ method

- **__init__:** The method_init_is the most important method in the class. This is called when an instance (object) of the class is created, using the class name as a function. The ___init____ method is called as constructor.
- **self:** In class, all methods have self as their first parameter (python adds self as argument which is well known to us) , although it isn't explicitly passed(passed by users). We can't use self while we call the method in a class. Within a method definition, self refers to the instance calling the method.
- In an __init__method, attributes can be used to set the initial value of instance's attributes in a class.

Features of Object Oriented Programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

- **Encapsulation:** Encapsulation refers to binding data and methods together inside a class. It keeps the data and methods safe from outside interference and misuse. Encapsulation prevents accessing data accidentally.

Inheritance

- **Inheritance:** It refers to creating a child class such that the child class would inherit all the properties (variables and methods) of the parent class. The parent class is called super class while the child class is called subclass.
- We have 3 types of inheritance mainly:
- **Single inheritance:** Only one sub class from super class.(superclass->subclass)
- **Hierarchical inheritance:** Inheriting from super class to as many subclasses.
- **Multilevel inheritance:** Inheriting properties from super class to sub class and then other sub classes.

Abstraction and Polymorphism

- **Abstraction:** It refers to creating structure classes that are not implemented. Abstract classes are like a base class and many other classes inherit the properties of abstract class but the abstract class itself is not implemented.

- **Polymorphism:** It is derived from two Greek words, poly (many) and morph (form). Polymorphism allows us to define methods with the same name in two different classes. If the two different classes are parent class and child class then the parent class's method will be overwritten by the child class's method. This is known as Method Overriding.

MODULE-II
PYTHON CLASSES AND OBJECTS

Creating A Class

CLASS

- we write a class with the attributes and actions of objects. Attributes are represented by variables and actions are performed by methods. So, a class contains variable and methods.
- A function written inside a class is called a method. Generally, a method is called using one of the following two ways:
- class name.methodname()
- instancename.methodname()
- The general format of a class is given as follows:

```
Class Classname(object):
""" docstring describing the class """
attributes  def __init__(self):
def method1():
def method2():
```

Creating A CLASS(Contd..)

- A class is created with the keyword class and then writing the Classname. After the Classname, 'object' is written inside the Classname.
- This 'object' represents the base class name from where all classes in Python are derived.
- Even our own classes are also derived from 'object' class. Hence, we should mention 'object' in the parentheses.

```
class Student:
#another way is:
class Student(object):
#the below block defines attributes
def __init__(self):
self.name = 'Vishnu'
self.age = 20
self.marks = 900
#the below block defines a method
```

Creating A CLASS(Contd..)

```
def talk(self):
    print('Hi, I am ', self.name)
    print('My age is', self.age)
    print('My marks are', self.marks)
```

- To create an instance, the following syntax is used:

```
instancename = Classname()
```

So, to create an instance (or object) to the Student class, we can write as:

```
s1 = Student()
```

When we create an instance like this, the following steps will take place internally:

1. First of all, a block of memory is allocated on heap. How much memory is to be allocated is decided from the attributes and methods available in the Student class.
2. After allocating the memory block, the special method by the name '__init__(self)' is called internally. This method stores the initial data into the variables. Since this method is useful to construct the instance, it is called 'constructor'.

Creating A CLASS(Contd..)

3. Finally, the allocated memory location address of the instance is returned into ‘s1’ variable. To see this memory location in decimal number format, we can use id() function as id(s1).

Program

Program 1: A Python program to define Student class and create an object to it. Also, we will call the method and display the student’s details.

```
#instance variables and instance method
class Student:
#this is a special method called constructor.
    def __init__(self):
        self.name = 'Vishnu'
        self.age = 20
        self.marks = 900
    #this is an instance method.
    def talk(self):
        print('Hi, I am', self.name)
```

Creating A CLASS(Contd..)

```
print('My age is', self.age)
print('My marks are', self.marks)
#create an instance to Student class.
s1 = Student()
#call the method using the instance.
s1.talk()
```

Output:

```
C:\>python cl.py
Hi, I am Vishnu
My age is 20
My marks are 900
```

The Self Variable

- 'self' is a default variable that contains the memory address of the instance of the current class.
- For example, we create an instance to Student class as:

s1 = Student()

We use 'self' in two ways:

1. The 'self' variable is used as first parameter in the constructor as:

def __init__(self):

In this case, 'self' can be used to refer to the instance variables inside the constructor.

2. 'self' can be used as first parameter in the instance methods as:

def talk(self):

Here, talk() is instance method as it acts on the instance variables.

Constructor

- A constructor is a special method that is used to initialize the instance variables of a class. In the constructor, we create the instance variables and initialize them with some starting values. The first parameter of the constructor will be ‘self’ variable that contains the memory address of the instance. For example,

```
def __init__(self):
  self.name = ‘Vishnu’
  self.marks = 900
```

Program 2: A Python program to create Student class with a constructor having more than one parameter.

```
#instance vars and instance method - v.20
    class Student:                    #this is constructor.
def __init__(self, n ='', m=0):
    self.name = n
    self.marks = m                    #this is an instance method.
        def display(self):
```

Constructor(Contd..)

```
print('Hi', self.name)
print('Your marks', self.marks)    #constructor is called without any
                                        arguments
s = Student()
s.display()
print('------------------')        #constructor is called with 2 arguments
s1 = Student('Lakshmi Roy', 880)
s1.display()
print('------------------')
```

Output: C:\>python cl.py

```
Hi
Your marks 0
------------------
Hi Lakshmi Roy
Your marks 880
------------------
```

Types of Variables

- The variables which are written inside a class are of 2 types:

1. Instance variables
2. Class variables or Static variables

Program 3: A Python program to understand instance variables.

```
#instance vars example
class Sample:               #this is a constructor.
def __init__(self):
    self.x = 10              #this is an instance method.
def modify(self):
    self.x+=1           #create 2 instances
    s1 = Sample()
    s2 = Sample()
    print('x in s1= ', s1.x)
    print('x in s2= ', s2.x)       #modify x in s1
    s1.modify()
```

Types of Variables (Contd..)

```
print('x in s1= ', s1.x)
print('x in s2= ', s2.x)
```

Output: C:\>python cl.py

x in s1= 10

x in s2= 10

x in s1= 11

x in s2= 10

Program 4: A Python program to understand class variables or static variables.

```
#class vars or static vars example
class Sample:                          #this is a class var
    x = 10                             #this is a class method.
    @classmethod
    def modify(cls):
        cls.x+=1                       #create 2 instances
s1 = Sample()
s2 = Sample()
```

Types of Variables (Contd..)

```
print('x in s1= ', s1.x)
print('x in s2= ', s2.x)
#modify x in s1
s1.modify()
print('x in s1= ', s1.x)
print('x in s2= ', s2.x)
```

Output: C:\>python cl.py

```
x in s1= 10
x in s2= 10
x in s1= 11
x in s2= 11
```

Namespaces (Contd..)

Namespaces

A namespace represents a memory block where names are mapped (or linked) to objects. Suppose we write:

```
    n = 10           #understanding class namespace
class Student:            #this is a class var
    n=10                  #access class var in the class namespace
    print(Student.n)      #displays 10
    Student.n+=1          #modify it in class namespace
    print(Student.n)      #displays 11
```

Types of Methods

- The purpose of a method is to process the variables provided in the class or in the method.
- We can classify the methods in the following 3 types:

1. Instance methods (a) Accessor methods (b) Mutator methods
2. Class methods
3. Static methods

Instance Methods

- Instance methods are the methods which act upon the instance variables of the class. Instance methods are bound to instances (or objects) and hence called as: instancename.method().
- Program: A Python program to store data into instances using mutator methods and to retrieve data from the instances using accessor methods.

```
#accessor and mutator methods
class Student:                    #mutator method
    def setName(self, name):
        self.name = name          #accessor method
```

Types of Methods (Contd...)

```
def getName(self):
    return self.name       #mutator method
def setMarks(self, marks):
    self.marks = marks   #accessor method
def getMarks(self):
    return self.marks       #create instances with some data from keyboard
    n = int(input('How many students? '))
    i=0
    while(i<n):               #create Student class instance
        s = Student()
        name = input('Enter name: ')
        s.setName(name)
        marks = int(input('Enter marks: '))
        s.setMarks(marks)        #retrieve data from Student class instance
     print('Hi', s.getName())
```

Types of Methods (Contd...)

```
print('Your marks', s.getMarks())
i+=1  print('-------------------')
```

Output: C:\>python cl.py

How many students? 2

Enter name: Vinay Krishna

Enter marks: 890

Hi Vinay Krishna

Your marks 890

Enter name: Vimala Rao

Enter marks: 750

Hi Vimala Rao

Your marks 750

Types of Methods (Contd...)

IARE
EDUCATION FOR LIBERATION

Class Methods

- These methods act on class level. Class methods are the methods which act on the class variables or static variables. These methods are written using @classmethod decorator above them. By default, the first parameter for class methods is 'cls' which refers to the class itself.
- Program 7: A Python program to use class method to handle the common feature of all the instances of Bird class.

```
#understanding class methods
    class Bird:                #this is a class var
        wings = 2                                   #this is a class method
    @classmethod
      def fly(cls, name):
          print('{} flies with {} wings'.format(name, cls.wings))   #display
          Bird.fly('Sparrow')
         Bird.fly('Pigeon')
```

Output: C:\>python cl.py   Sparrow flies with 2 wings   Pigeon flies with 2 wings

Types of Methods (Contd...)

Static Methods

- We need static methods when the processing is at the class level but we need not involve the class or instances. Static methods are used when some processing is related to the class but does not need the class or its instances to perform any work.

Program : A Python program to create a static method that counts the number of instances created for a class.

```
class Myclass:
#this is class var or static var          n=0
#constructor that increments n when an instance is created
    def __init__(self):
        Myclass.n = Myclass.n+1
        @staticmethod  def noObjects():
        print('No. of instances created: ', Myclass.n)
        obj1 = Myclass()    obj2 = Myclass()   obj3 = Myclass()
Myclass.noObjects()
```

Output: C:\>python cl.py   No. of instances created: 3

Inheritance

A programmer in the software development is creating Teacher class with setter() and getter() methods as shown in Program 1. Then he saved this code in a file ‘teacher.py’.

Program

Program 1: A Python program to create Teacher class and store it into teacher.py module. #this is Teacher class. save this code in teacher.py file

```
class Teacher:
    def setid(self, id):
        self.id = id
    def getid(self):
        return self.id
    def setname(self, name):
        self.name = name
    def getname(self):
        return self.name
```

Inheritance(Contd..)

```
def setaddress(self, address):
        self.address = address
def getaddress(self):
      return self.address
def setsalary(self, salary):
      self.salary = salary
def getsalary(self):
    return self.salary
```

Program 2: A Python program to use the Teacher class.

```
#save this code as inh.py file
#using Teacher class  from teacher import Teacher
#create instance  t = Teacher()
#store data into the instance        t.setid(10)
t.setname('Prakash')
t.setaddress('HNO-10, Rajouri gardens, Delhi')
```

t.setsalary(25000.50) #retrieve data from instance and display

print('id=', t.getid())

print('name=', t.getname())

print('address=', t.getaddress())

print('salary=', t.getsalary())

Output: C:\>python inh.py

id= 10

name= Prakash

address= HNO-10, Rajouri gardens, Delhi

salary= 25000.5

A Python program to create Student class by deriving it from the Teacher class. #Student class - v2.0.save it as student.py

from teacher import Teacher

class Student(Teacher):

def setmarks(self, marks): self.marks = marks

def getmarks(self): return self.marks

Constructors in Inheritance

Program 6: A Python program to access the base class constructor from sub class.

```
#base class constructor is available to sub class
    class Father:
        def __init__(self):
            self.property = 800000.00
        def display_property(self):
            print('Father\'s property=', self.property)
    class Son(Father):
        pass                    #we do not want to write anything in the sub class
    #create sub class instance and display father's property
        s = Son()
        s.display_property()
```

Output: C:\>python inh.py

Father's property= 800000.0

Super() Method

- super() is a built-in method which is useful to call the super class constructor or methods from the sub class

Program 8: A Python program to call the super class constructor in the sub class using super().

```
#accessing base class constructor in sub class        class Father:
def __init__(self, property=0):
      self.property = property
def display_property(self):
     print('Father\'s property=', self.property)
class Son(Father):
         def __init__(self, property1=0, property=0):
               super().__init__(property)
               self.property1= property1
         def display_property(self):
               print('Total property of child=', self.property1 + self.property)
#create sub class instance and display father's property
```

Super() Method (Contd...)

```
s = Son(200000.00, 800000.00)
s.display_property()
```

Output:

```
C:\>python inh.py
Total property of child= 1000000.0
```

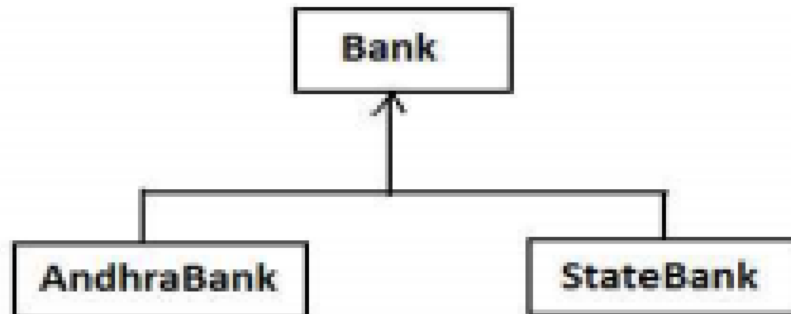
Types of Inheritance

There are mainly 2 types of inheritance available. They are:

1. Single inheritance
2. Multiple inheritance

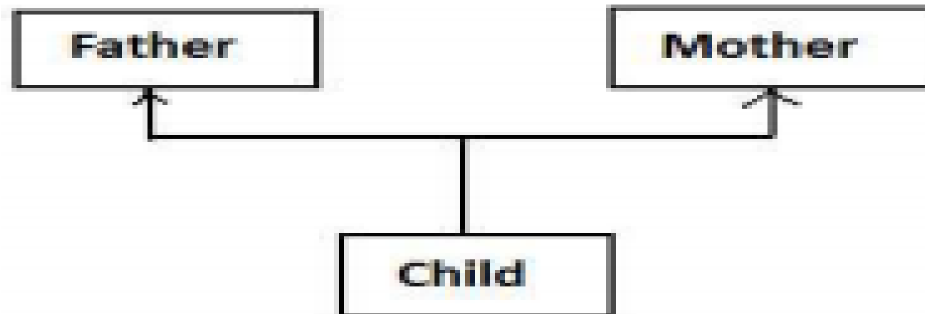
Single Inheritance

Deriving one or more sub classes from a single base class is called 'single inheritance'. In single inheritance, we always have only one base class, but there can be n number of sub classes derived from it. For example,



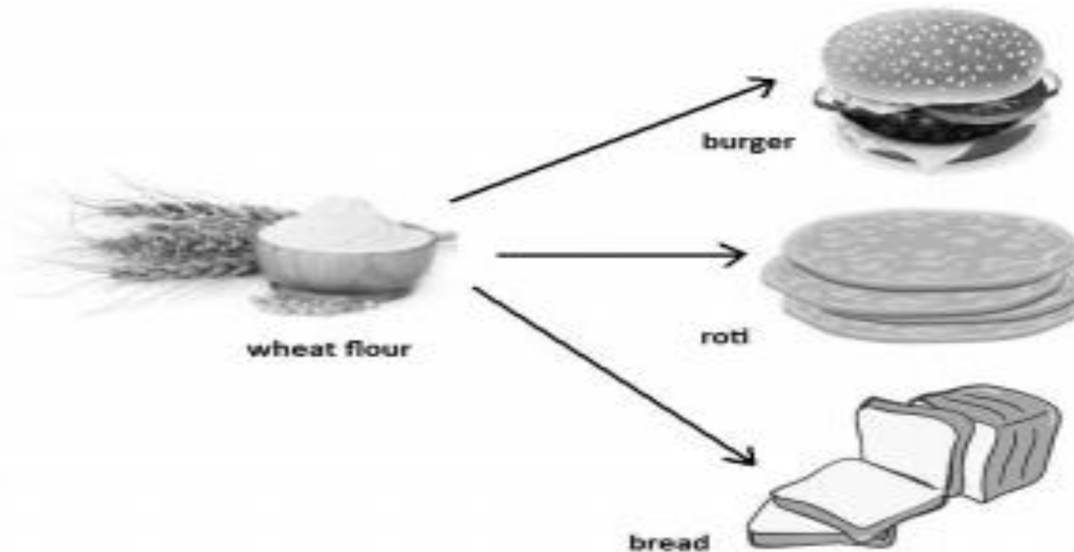
Multiple Inheritance

- Deriving sub classes from multiple (or more than one) base classes is called ‘multiple inheritance’. In this type of inheritance, there will be more than one super class and there may be one or more sub classes.



Polymorphism

- Polymorphism is a word that came from two Greek words, poly means many and morphos means forms. If something exhibits various forms, it is called polymorphism. Let's take a simple example in our daily life. Assume that we have wheat flour. Using this wheat flour, we can make burgers, rotis, or loaves of bread. It means same wheat flour is taking different edible forms and hence we can say wheat flour is exhibiting polymorphism. Consider Figure:



Polymorphism(Contd..)

The following topics are examples for polymorphism in Python:

- Duck typing philosophy of Python
- Operator overloading
- Method overloading
- Method overriding

Abstract classes and interfaces

- An abstract method is a method whose action is redefined in the sub classes as per the requirement of the objects. To mark a method as abstract, we should use the decorator @abstractmethod.
- An abstract class is a class that generally contains some abstract methods.

Program : A Python program to create abstract class and sub classes which implement the abstract method of the abstract class.

```
#abstract class example
from abc import ABC, abstractmethod
class Myclass(ABC):
@abstractmethod
    def calculate(self, x):
        pass
#empty body, no code
#this is sub class of Myclass
class Sub1(Myclass):
```

Abstract classes and interfaces(Contd...)

```
def calculate(self, x):
    print('Square value=', x*x)
#this is another sub class for Myclass
import math
class Sub2(Myclass):
    def calculate(self, x):
        print('Square root=', math.sqrt(x))
#third sub class for Myclass
class Sub3(Myclass):
    def calculate(self, x):
        print('Cube value=', x**3)
#create Sub1 class object and call calculate() method
obj1 = Sub1()
obj1.calculate(16)
#create Sub2 class object and call calculate() method
obj2 = Sub2()
```

```
obj2.calculate(16)
#create Sub3 class object and call calculate() method
obj3 = Sub3()
obj3.calculate(16)
```

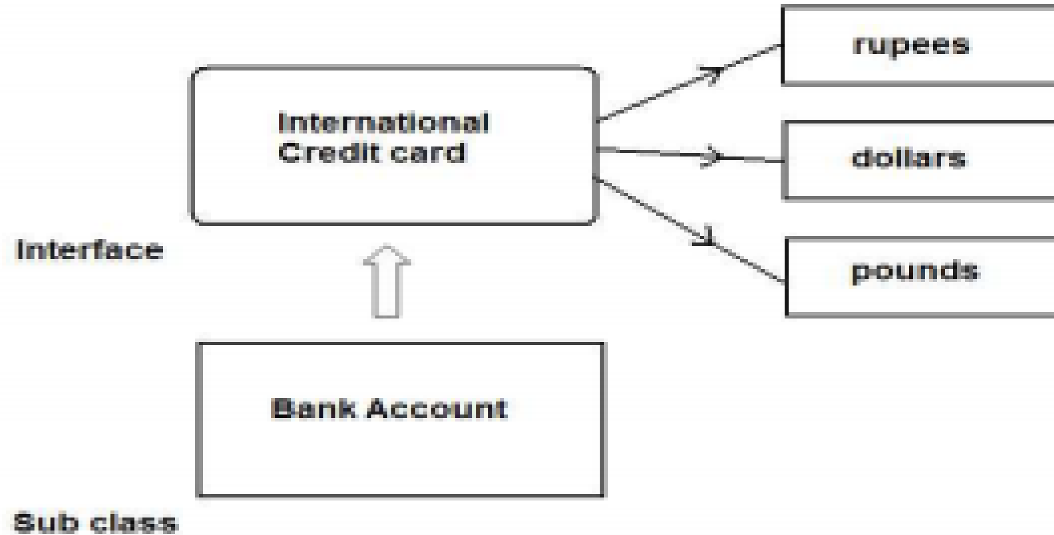
Output:

```
C:\>python abs.py
Square value= 256
Square root= 4.0
Cube value= 4096
```

Interfaces in Python

- An interface can be defined as a specification of method headers. Since, we write only abstract methods in the interface, there is possibility for providing different implementations (body) for those abstract methods depending on the requirements of objects. We have to use abstract classes as interfaces in Python. Since an interface contains methods without body, it is not possible to create objects to an interface.

Abstract classes and interfaces(Contd..)




```
#an interface to connect to any database
class Myclass(ABC):
@abstractmethod
   def connect(self):
        pass
@abstractmethod
     def disconnect(self):
        pass
```

Abstract Classes vs. Interfaces

- Python does not provide interface concept explicitly. It provides abstract classes which can be used as either abstract classes or interfaces.
- It is the discretion of the programmer to decide when to use an abstract class and when to go for an interface.
- For example, take a class WholeSaler which represents a whole sale shop with text books and stationery like pens, papers and note books as:

```
#an abstract class
class WholeSaler(ABC):
    @abstractmethod
    def text_books(self):
        pass
    @abstractmethod
    def stationery(self):
        pass
```

MODULE-III

STRINGS AND FUNCTIONS

Creating Strings

String

- String is group of characters. We can create a string in Python by assigning a group of characters to a variable. The group of characters should be enclosed inside single quotes or double quotes as:

```
s1 = 'Welcome to Core Python learning'
s2 = "Welcome to Core Python learning“
```

- There is no difference between the single quotes and double quotes while creating the strings. Both will work in the same manner.

- It is possible to display quotation marks to mark a sub string in a string.

```
s1 = 'Welcome to "Core Python" learning'
print(s1)
```

Creating Strings(Contd..)

- It is possible to use escape characters like \t or \n inside the strings.
- The escape character \t releases tab space of 6 or 8 spaces and the escape character \n throws cursor into a new line.
- Table below summarizes the escape characters that can be used in strings:

Escape Character	Meaning
\a	Bell or alert
\b	Backspace
\n	New line
\t	Horizontal tab space
\v	Vertical tab space
\r	Enter button
\x	Character x
\\	Displays single\

Basic operations on strings

The following are the basic operations we can perform on strings

1.Length of string

- Length of a string represents the number of characters in a string.

- To know the length of a string, we can use the len() function.
-
- This function gives the number of characters including spaces in the string.

```
str = 'Core Python'
n = len(str)
 print(n)
```

- The preceding lines of code will display the following output:

```
11
```

Basic operations on strings(Contd..)

2.Indexing in Strings

- Index represents the position number. Index is written using square braces [].

- By specifying the position number through an index, we can refer to the individual elements (or characters) of a string.

- For example, str[0] refers to the 0th element of the string and str[1] refers to the 1st element of the string. Thus, str[i] can be used to refer to ith element of the string.

- Here, ‘i’ is called the string index because it is specifying the position number of the element in the string.

- When we use index as a negative number, it refers to elements in the reverse order. Thus, str[-1] refers to the last element and str[-2] refers to second element from last.

Basic operations on strings(Contd..)

IARE
EDUCATION FOR LIBERATION
2000

3.Slicing the Strings

- A slice represents a part or piece of a string. The format of slicing is:

stringname[start: stop: stepsize]

- If 'start' and 'stop' are not specified, then slicing is done from 0th to n-1th elements. If 'stepsize' is not written, then it is taken to be 1
- See the following example:

```
str = 'Core Python'
str[0:9:1] #access string from 0th to 8th element in steps of 1
```

Output: Core Pyth

- Consider the following code snippet:

```
str = 'Core Python'
str[::] #access string from 0th to last character
```

- The preceding lines of code will display the following output:

Core Python

4.Concatenation of Strings

- We can use ‘+’ on strings to attach a string at the end of another string.

- This operator ‘+’ is called addition operator when used on numbers. But, when used on strings, it is called ‘concatenation’ operator since it joins or concatenates the strings.
- For example:

```
s1='Core'
s2="Python"
s3=s1+s2 #concatenate s1 and s2
print(s3) #display the total string s3
```

- The output of the preceding statement is as follows:

CorePython

5.Comparing Strings

- We can use the relational operators like >, >=, <, <=, == or != operators to compare two strings.

- They return Boolean value, i.e. either True or False depending on the strings being compared.

- For example:

```
s1='Box'
s2='Boy'
 if(s1==s2):
print('Both are same')
 else:
print('Not same')
```

- This code returns ‘Not same’ as the strings are not same.

Basic operations on strings(Contd..)

6.Removing Spaces from a String

- A space is also considered as a character inside a string.
- Sometimes, the unnecessary spaces in a string will lead to wrong results.
- For example, a person typed his name ‘Mukesh‘ (observe two spaces at the end of the string) instead of typing ‘Mukesh’.
- If we compare these two strings using ‘==’ operator as:

```
if 'Mukesh '=='Mukesh':
print('Welcome')
else: print('Name not found')
```

- The output will be ‘Name not found’.

Basic operations on strings(Contd..)

- Hence such spaces should be removed from the strings before they are compared.
- This is possible using rstrip(), lstrip() and strip() methods.
- The rstrip() method removes the spaces which are at the right side of the string.
- The lstrip() method removes spaces which are at the left side of the string.
- strip() method removes spaces from both the sides of the strings.
- These methods do not remove spaces which are in the middle of the string.

Basic operations on strings(Contd..)

7.Finding Sub Strings

- The find(), rfind(), index() and rindex() methods are useful to locate sub strings in a string. These methods return the location of the first occurrence of the sub string in the main string.

- The find() and index() methods search for the sub string from the beginning of the main string.

- The rfind() and rindex() methods search for the sub string from right to left, i.e. in backward order.

- The find() method returns -1 if the sub string is not found in the main string.

- The index() method returns 'ValueError' exception if the sub string is not found. The format of find() method is: mainstring.find(substring, beginning, ending)

2000 IARE EDUCATION FOR LIBERATION

8.Splitting and Joining Strings

- The split() method is used to brake a string into pieces. These pieces are returned as a list.

- For example, to brake the string ‘str’ where a comma (,) is found, we can write:

str.split(‘,’)

- In the following example, we are cutting the string ‘str’ wherever a comma is found. The resultant string is stored in ‘str1’ which is a list.

```
str = 'one,two,three,four'
str1 = str.split(',')
print(str1)
```

- The output of the preceding statements is as follows:

['one', 'two', 'three', 'four']

Basic operations on strings(Contd..)

- In the following example, we are taking a list comprising 4 strings and we are joining them using a colon (:) between them.

```
str = ['apple', 'guava', 'grapes', 'mango']
sep =':'
str1 = sep.join(str)
print(str1)
```

- The output of the preceding statements is as follows:

apple:guava:grapes:mango

Basic operations on strings(Contd..)

9.Changing Case of a String

- Python offers 4 methods that are useful to change the case of a string. They are upper(), lower(), swapcase(), title().

- The upper() method is used to convert all the characters of a string into uppercase or capital letters.

- The lower() method converts the string into lowercase or into small letters.

- The swapcase() method converts the capital letters into small letters and vice versa.

- The title() method converts the string such that each word in the string will start with a capital letter and remaining will be small letters.

String testing methods

- There are several methods to test the nature of characters in a string. These methods return either True or False.

- For example, if a string has only numeric digits, then isdigit() method returns True.

- These methods can also be applied to individual characters. Below table shows the string and character testing methods:

Method	Description
isalnum()	This method returns True if all characters in the string are alphanumeric (A to Z, a to z, 0 to 9) and there is at least one character; otherwise it returns False.
isalpha()	Returns True if the string has at least one character and all characters are alphabetic (A to Z and a to z); otherwise, it returns False.
isdigit()	Returns True if the string contains only numeric digits (0 to 9) and False otherwise.
islower()	Returns True if the string contains at least one letter and all characters are in lower case; otherwise, it returns False.

String testing methods(Contd..)

Method	Description
isupper()	Returns True if the string contains at least one letter and all characters are in upper case; otherwise, it returns False.
istitle()	Returns True if each word of the string starts with a capital letter and there is at least one character in the string; otherwise, it returns False.
isspace()	Returns True if the string contains only spaces; otherwise, it returns False.

Table: String and character testing methods

Functions

- A function is similar to a program that consists of a group of statements that are intended to perform a specific task.
- The main purpose of a function is to perform a specific task or work. Thus when there are several tasks to be performed, the programmer will write several functions.
- There are several 'built-in' functions in Python to perform various tasks.
- For example, to display output, Python has print() function. Similarly, to calculate square root value, there is sqrt() function and to calculate power value, there is power() function

Functions(Contd..)

Advantages

- Functions are important in programming because they are used to process data
- Once a function is written, it can be reused as and when required.
- Functions provide modularity for programming. A module represents a part of the program.
- Code maintenance will become easy because of functions.
- When there is an error in the software, the corresponding function can be modified without disturbing the other functions in the software. Thus code debugging will become easy.
-
- The use of functions in a program will reduce the length of the program.

Defining Functions

- We can define a function using the keyword def followed by function name.
- After the function name, we should write parentheses () which may contain parameters.

Syntax:

def functionname(parameter1,parameter2,….):

“””function docstring”””

function statements

Example:

def add(a,b):

“””This function finds sum of two numbers”””

c=a+b

print(c)

Calling a function

- A function cannot run on its own. It runs only when we call it. So, the next step is to call the function using its name.

- While calling the function, we should pass the necessary values to the function in the parentheses as:
- sum(10, 15)

- Here, we are calling the 'sum' function and passing two values 10 and 15 to that function.

- When this statement is executed, the Python interpreter jumps to the function definition and copies the values 10 and 15 into the parameters 'a' and 'b' respectively.

Calling a function(Contd..)

Example:

A function that accepts two values and finds their sum.

```
#a function to add two numbers
def sum(a, b):
    """ This function finds sum of two numbers """
    c = a+b
    print('Sum=', c)
#call the function
sum(10, 15)
sum(1.5, 10.75) #call second time
```

Output:

```
C:\>python fun.py
Sum= 25
Sum= 12.25
```

-
-

Returning Results from a Function

- We can return the result or output from the function using a ‘return’ statement in the body of the function.

- For example,

return c #returns c value out of function

return 100 #returns 100

return lst #return the list that contains values

return x, y, c #returns 3 values

- When a function does not return any result, we need not write the return statement in the body of the function.

Returning Multiple Values from a Function

- A function returns a single value in the programming languages like C or Java. But in Python, a function can return multiple values.

- When a function calculates multiple results and wants to return the results, we can use the return statement as:

```
return a, b, c
```

- we can use three variables at the time of calling the function as:

```
x, y, z = function()
```

Example:

```
 def sum_sub(a, b):
     c = a + b
     d = a – b
return c, d
```

Returning Multiple Values from a Function

Example:

A Python program to understand how a function returns two values.

```
#a function that returns two results
def sum_sub(a, b):
""" this function returns results of addition and subtraction of a, b """
     c = a + b
    d = a – b
    return c, d
#get the results from the sum_sub() function
 x, y = sum_sub(10, 5)
#display the results
print("Result of addition:", x)
print("Result of subtraction:", y)
```

Output: C:\>python fun.py

Result of addition: 15

Result of subtraction: 5

Functions are First Class Objects

- In Python, functions are considered as first class objects. It means we can use functions as perfect objects.

- In fact when we create a function, the Python interpreter internally creates an object.

- Since functions are objects, we can pass a function to another function just like we pass an object (or value) to a function.

- The following possibilities are noteworthy:

1. It is possible to assign a function to a variable.
2. It is possible to define one function inside another function.
3. It is possible to pass a function as parameter to another function.
4. It is possible that a function can return another function.

Functions are First Class Objects(Contd..)

1. Assign a function to variable

A Python program to see how to assign a function to a variable.

```
#assign a function to a variable
def display(str):
      return 'Hai '+str
#assign function to variable x
     x = display("Krishna")
     print(x)
```

Output: C:\>python fun.py

Hai Krishna

Functions are First Class Objects(Contd..)

2.Defining one function inside another function

A Python program to know how to define a function inside another function.

```
#define a function inside another function
def display(str):
    def message():
        return 'How are U?’
     result = message()+str
return result
#call display() function
print(display("Krishna"))
```

Output: C:\>python fun.py

How are U? Krishna

Functions are First Class Objects(Contd..)

3.Pass a function as parameter to another function

A Python program to know how to pass a function as parameter to another function.

```
#functions can be passed as parameters to other functions
def display(fun):
     return 'Hai '+ fun
def message():
    return 'How are U? '
#call display() function and pass message() function
 print(display(message()))
```

Output: C:\>python fun.py

Hai How are U?

4.A function can return another function

A Python program to know how a function can return another function.

```
#functions can return other functions
def display():
     def message():
           return 'How are U?’
      return message
#call display() function and it returns message() function
#in the following code, fun refers to the name: message.
fun = display()
print(fun())
```

Output: C:\>python fun.py

How are U?

Formal and Actual Arguments

- When a function is defined, it may have some parameters. These parameters are useful to receive values from outside of the function. They are called ‘formal arguments’.

- When we call the function, we should pass data or values to the function. These values are called ‘actual arguments’.

- In the following code, ‘a’ and ‘b’ are formal arguments and ‘x’ and ‘y’ are actual arguments.

```
def sum(a, b):
#a, b are formal arguments
    c = a+b
    print(c)
#call the function x=10; y=15
   sum(x, y)
#x, y are actual arguments
```

Formal and Actual Arguments(Contd..)

IARE EDUCATION FOR LIBERATION 2000

The actual arguments used in a function call are of 4 types:

1. Positional arguments
2. Keyword arguments
3. Default arguments
4. Variable length arguments

1.Positional arguments

These are the arguments passed to a function in correct positional order. Here, the number of arguments and their positions in the function definition should match exactly with the number and position of the argument in the function call.

2. Keyword arguments

Keyword arguments are arguments that identify the parameters by their names. For example, the definition of a function that displays grocery item and its price can be written as:

```
def grocery(item, price):
```

3.Default Arguments

We can mention some default value for the function parameters in the definition. Let's take the definition of grocery() function as:

```
def grocery(item, price=40.00):
```

- **4.Variable Length Arguments**

Sometimes, the programmer does not know how many values a function may receive. In that case, the programmer cannot decide how many arguments to be given in the function definition.

The variable length argument is written with a ' * ' symbol before it in the function definition as:

```
def add(farg, *args):
```

Recursive Functions

- A function that calls itself is known as ‘recursive function’.

- For example, we can write the factorial of 3 as:

factorial(3) = 3 * factorial(2)

Here, factorial(2) = 2 * factorial(1)

And, factorial(1) = 1 * factorial(0)

- Now, if we know that the factorial(0) value is 1, all the preceding statements will evaluate and give the result as:

factorial(3) = 3 * factorial(2)
= 3 * 2 * factorial(1)
= 3 * 2 * 1 * factorial(0)
= 3 * 2 * 1 * 1 = 6

- From the above statements, we can write the formula to calculate factorial of any number ‘n’ as: factorial(n) = n * factorial(n-1)

Recursive Functions (Contd..)

Example:

A Python program to calculate factorial values using recursion.

```
 #recursive function to calculate factorial
def factorial(n):
 """ to find factorial of n """
if n==0:
   result=1
else:
   result=n*factorial(n-1)
 return result
 #find factorial values for first 10 numbers
for i in range(1, 11):
print('Factorial of {} is {}'.format(i, factorial(i)))
```

MODULE-IV
Errors & Exceptions of Python

Errors in Python

- The error is something that goes wrong in the program, e.g., like a syntactical error.

- It occurs at compile time. Let’s see an example.
  - if a<5
  - File "<interactive input>", line 1
  -     if a < 5
  -           ^

- SyntaxError: invalid syntax

Errors in Python(Contd..)

Syntax Errors

- Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.

  - >>> if a < 3
  - File "<interactive input>", line 1
  - if a < 3
  - ^
  - SyntaxError: invalid syntax

- We can notice here that a colon is missing in the if statement.

Errors in Python(Contd..)

- Syntax errors, also known as parsing errors, are perhaps the most common kind of error you encounter while you are still learning Python.

```
>>> while True print 'Hello world'
  File "<stdin>", line 1, in ?
    while True print 'Hello world'
                   ^
SyntaxError: invalid syntax
```

- The parser repeats the offending line and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected. The error is detected at the token *preceding* the arrow. File name and line number are printed so you know where to look in case the input came from a script.

Exceptions

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.
- Errors detected during execution are called *exceptions* and are not unconditionally fatal.
- Most exceptions are not handled by programs, however, and result in error messages like “cannot divide by zero” or “cannot concatenate ‘str’ and ‘int’ objects”.

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Handling Exceptions

2000 IARE EDUCATION FOR LIBERATION

- It is possible to write programs that handle selected exceptions. Consider the following, where a user-generated interruption is signaled by raising the Keyboard Interrupt exception.

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops!  That was no valid number.  Try again..."
```

- First the 'try' clause is executed until an exception occurs, in which case the rest of 'try' clause is skipped and the 'except' clause is executed (depending on type of exception), and execution continues. If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops.

Handling Exceptions(Contd...)

- The last except clause (when many are declared) may omit the exception name(s), to serve as a wildcard. This makes it very easy to mask a real programming error. It can also be used to print an error message and then re-raise the exception.
- The try-except statement has an optional *else clause*, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception.

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

Raising Exceptions

- The raise statement allows the programmer to force a specified exception to occur.

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

- The sole argument to raise indicates the exception to be raised.
- A simpler form of the raise statement allows one to re-raise the exception (if you don’t want to handle it):

Raising Exceptions(Contd..)

A simpler form of the raise statement allows one to re-raise the exception (if you don't want to handle it):

```
>>> try:
...         raise NameError('HiThere')
... except NameError:
...         print 'An exception flew by!'
...         raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

User-defined Exceptions

- Programs may name their own exceptions by creating a new exception class. These are derived from the Exception class, either directly or indirectly.

```
>>> class MyError(Exception):
...         def __init__(self, value):
...             self.value = value
...         def __str__(self):
...             return repr(self.value)
...
>>> try:
...         raise MyError(2*2)
...     except MyError as e:
...         print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
   File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

- Here, the def__init__() of Exception has been overridden. The new behavior simply creates the value attribute.

Defining Clean-up Actions

2000 IARE EDUCATION FOR LIBERATION

- The try statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances.

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
KeyboardInterrupt
```

- A finally clause is executed before leaving the try statement, whether an exception has occurred or not. When an exception has occurred in the try clause and has not been handled by an except clause, it is re-raised after the finally clause has been executed. The finally clause is also executed “on the way out” when any other clause of the try statement is exited using break/continue/return.

Predefined Clean-up Actions

- Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed.

```
for line in open("myfile.txt"):
    print line
```

- The problem with this code is that it leaves the file open for an indefinite amount of time after the code has finished executing.
- The 'with' statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```
with open("myfile.txt") as f:
    for line in f:
        print line
```

MODULE-V
GRAPHICAL USER INTERFACE

UNIT-V GUI- SYLLABUS

- GUI in Python
- The Root window
- Fonts and colors
- Working with containers
- Canvas
- Frames
- Widgets
- Button widget
- Label Widget
- Message widget
- Text widget
- Radio button Widget
- Entry widget

GUI IN PYTHON

GUI offers the following advantages:

- It is user-friendly. The user need not worry about any commands. Even a layman will be able to work with the application developed using GUI.
- It adds attraction and beauty to any application by adding pictures, colors, menus, animation, etc. For example, all websites on Internet are developed using GUI to lure their visitors and improve their business.

GUI IN PYTHON

- It is possible to simulate the real life objects using GUI. For example, a calculator program may actually display a real calculator on the screen. The user feels that he is interacting with a real calculator and he would be able to use it without any difficulty or special training. So, GUI eliminates the need of user training.
- GUI helps to create graphical components like push buttons, radio buttons, check buttons, menus, etc. and use them effectively.

GUI IN PYTHON

- Python offers tkinter module to create graphics programs.
- The tkinter represents ‘toolkit interface’ for GUI. This is an interface for Python programmers that enable them to use the classes of TK module of TCL/TK language.
- Let’s see what this TCL/TK is. The TCL (Tool Command Language) is a powerful dynamic programming language, suitable for web and desktop applications, networking, administration, testing and many more.
- It is open source and hence can be used by any one freely. TCL language uses TK (Tool Kit) language to generate graphics.

GUI IN PYTHON

- TK provides standard GUI not only for TCL but also for many other dynamic programming languages like Python.

- Hence, this TK is used by Python programmers in developing GUI applications through Python’s tkinter module.

The general steps involved in basic GUI programs:

- First of all, we should create the root window. The root window is the top level window that Provides rectangular space on the screen where we can display text, colors, images, components, etc.

GUI IN PYTHON

- In the root window, we have to allocate space for our use. This is done by creating a canvas or frame. So, canvas and frame are child windows in the root window.

- Generally, we use canvas for displaying drawings like lines, arcs, circles, shapes, etc. We use Frame for the purpose of displaying components like push buttons, check buttons, menus, etc. These components are also called 'widgets'.

- When the user clicks on a widget like push button, we have to handle that event. It means we have to respond to the events by performing the desired tasks.

THE ROOT WINDOW

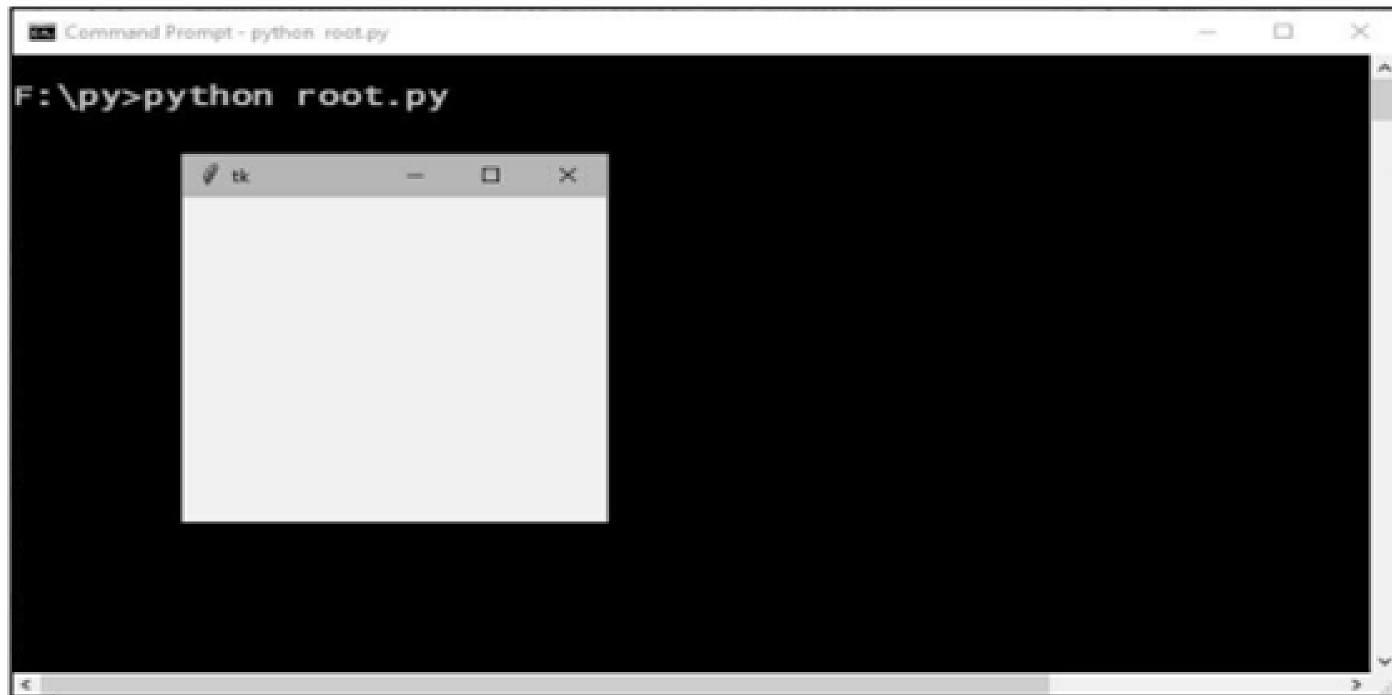
- To display the graphical output, we need space on the screen. This space that is initially allocated to every GUI program is called ‘top level window’ or ‘root window’.
- the root window is the highest level GUI component in any tkinter application.
- Root window by creating an object to Tk class. The root window will have a title bar that contains minimize, resize and close options.
- When you click on close ‘X’ option, the window will be destroyed.

Program : A Python program to create root window or top level window.

```
# import all components from tkinter
from tkinter import *
```

```
# create the root window
root = Tk()

# wait and watch for any events that may take place
# in the root window
root.mainloop()
```

Output:



FONTS AND COLORS

- A font represents a type of displaying letters and numbers. In tkinter, fonts are mentioned using a tuple that contains font family name, size and font style as:

```
fnt =('Times', -40, 'bold italic underline overstrike')
```

FONTS AND COLORS

Program: A Python program to know the available font families.

```
from tkinter import *
from tkinter import font

# create root window
root = Tk()

# get all the supported font families
list_fonts = list(font.families())

# display them
print(list_fonts)
```

FONTS AND COLORS

Colors in tkinter can be displayed directly by mentioning their names as: blue, light blue, dark blue, red, light red, dark red, black, white, yellow, magenta, cyan, etc. We can also specify colors using the hexadecimal numbers in the format:

```
#rrggbb# 8 bits per color
#rrrgggbbb# 12 bits per color
```

For example, #000000 represents black and #ff0000 represents red. In the same way, #000fff000 represents pure green and#00ffff is cyan (green plus blue).

WORKING WITH CONTAINERS

- A container is a component that is used as a place where drawings or widgets can be displayed. In short, a container is a space that displays the output to the user.

There are two important containers:

- **Canvas:** This is a container that is generally used to draw shapes like lines, curves, arcs and circles.

- **Frame:** This is a container that is generally used to display widgets like buttons, check buttons or menus. After creating the root window, we have to create space, i.e. the container in the root window so that we can use this space for displaying any drawings or widgets.

CANVAS

A canvas is a rectangular area which can be used for drawing pictures like lines, circles, polygons, arcs, etc. To create a canvas, we should create an object to Canvas class as:

```
c = Canvas(root, bg="blue", height=500, width=600, cursor='pencil')
```

Here, ‘c’ is the Canvas class object. ‘root’ is the name of the parent window. ‘bg’ represents background color, ‘height’ and ‘width’ represent the height and width of the canvas in pixels. A pixel (picture element) is a minute dot with which all the text and pictures on the monitor are composed

CANVAS

2000 IARE EDUCATION FOR LIBERATION

Once the canvas is created, it should be added to the root window. Then only it will be visible. This is done using the pack() method, as follows:

```
c.pack()
```

After the canvas is created, we can draw any shapes on the canvas. For example, to create a line, we can use create line () method, as:

```
id = c.create_line(50, 50, 200, 50, 200, 150, width=4, fill="white")
```

To create an oval, we can use the create oval () method. An oval is also called ellipse.

```
id = c.create_oval(100, 100, 400, 300, width=5, fill="yellow",
     outline="red", activefill="green")
```

CANVAS

A polygon represents several points connected by either straight lines or smooth lines. To create a polygon, we can use the create polygon () method as:

```
id = c.create_polygon(10, 10, 200, 200, 300, 200, width=3,
     fill="green", outline="red", smooth=1, activefill="lightblue")
```

Similarly, to create a rectangle or square shaped box, we can use the create rectangle() method as:

```
id = c.create_rectangle(500, 200, 700, 600, width=2, fill="gray",
     outline="black", activefill="yellow")
```

It is also possible to display some text in the canvas. For this purpose, we should use the create text () method as:

```
id = c.create_text(500, 100, text="My canvas", font= fnt,
     fill="yellow", activefill="green")
```

Here, the 'font' option is showing 'fnt' object that can be created as:

```
fnt =('Times', 40, 'bold')
fnt =('Times', -40, 'bold italic underline')
```

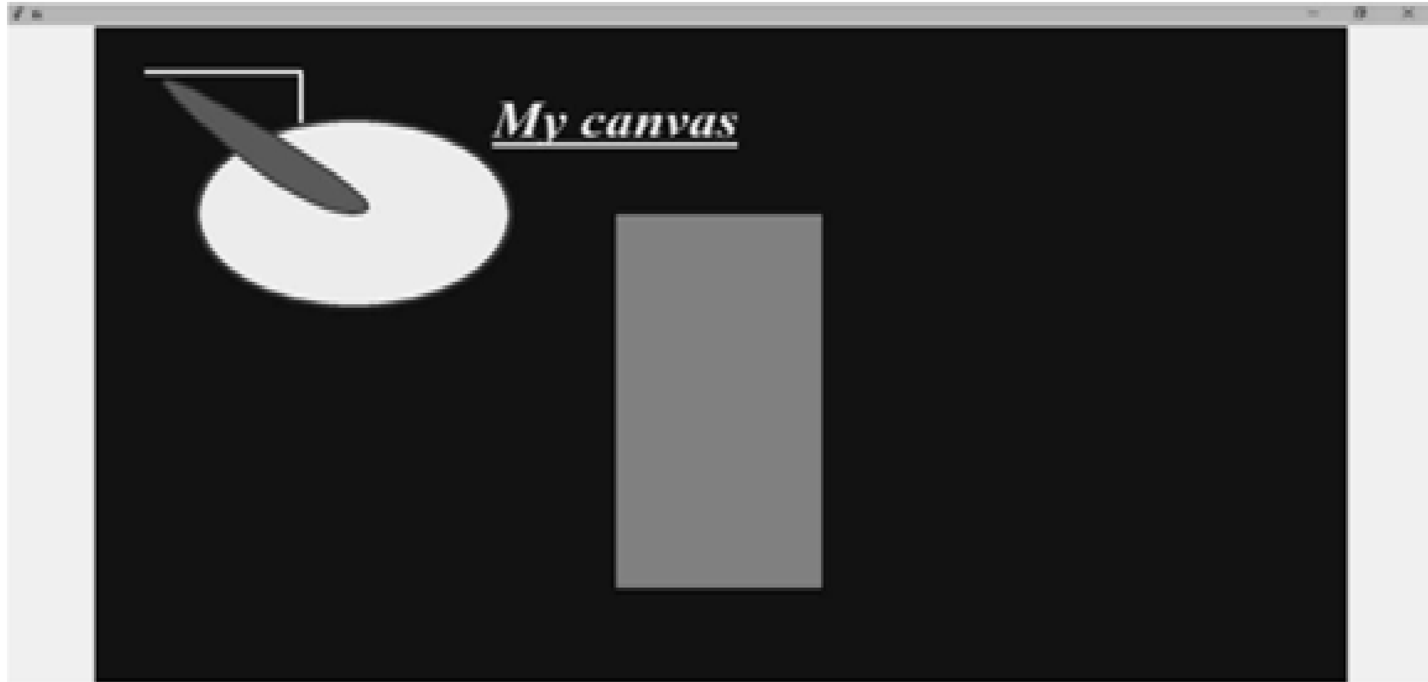
CANVAS

Program : A GUI program that demonstrates the creation of various shapes in canvas.

```
from tkinter import *

# create root window
root = Tk()

# create Canvas as a child to root window
c = Canvas(root, bg="blue", height=700, width=1200, cursor='pencil')

# create a line in the canvas
id = c.create_line(50, 50, 200, 50, 200, 150, width=4, fill="white")

# create an oval in the canvas
id = c.create_oval(100, 100, 400, 300, width=5, fill="yellow",
     outline="red", activefill="green")

# create a polygon in the canvas
id = c.create_polygon(10, 10, 200, 200, 300, 200, width=3,
     fill="green", outline="red", smooth=1, activefill="lightblue")

# create a rectangle in the canvas
id = c.create_rectangle(500, 200, 700, 600, width=2, fill="gray",
     outline="black", activefill="yellow")

# create some text in the canvas
fnt =('Times', 40, 'bold italic underline')
id = c.create_text(500, 100, text="My canvas", font= fnt,
     fill="yellow", activefill="green")

# add canvas to the root window
c.pack()

# wait for any events
root.mainloop()
```

CANVAS

Output:



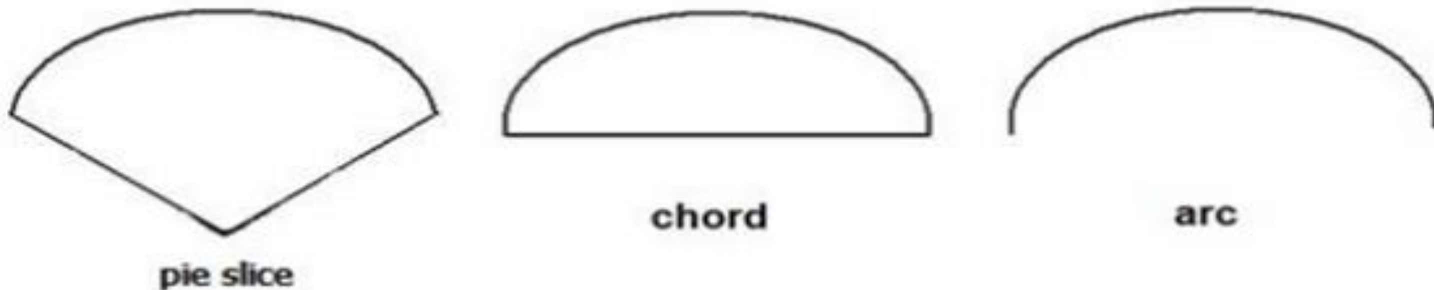
CANVAS

Another important shape that we can draw in the canvas is an arc. An arc represents a part of an ellipse or circle. Arcs can be created using the create arc () method as:

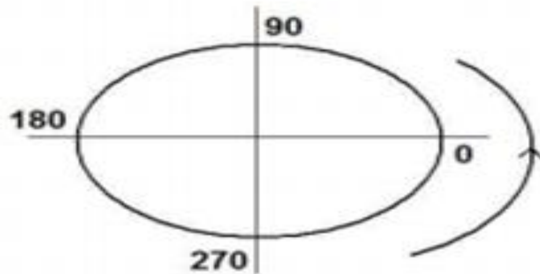
```
id = c.create_arc(100, 100, 400, 300, width=3, start=270, extent=180,
     outline="red", style="arc")
```

Here, the arc is created in the rectangular space defined by the coordinates (100, 100) and (400, 300). The width of the arc will be 3 pixels. The arc will start at an angle 270 degrees and extend for another 180 degrees (i.e. up to 450 degrees means 450 – 360 = 90 degrees).
The outline of the arc will be in red color. ‘style’ option can be “arc” for drawing arcs. ‘style’ can be “pie slice” and “chord”.

Another important shape that we can draw in the canvas is an arc. An arc represents a part of an ellipse or circle. Arcs can be created using the create arc () method as:



As mentioned, the option 'start' represents an angle of the arc where it has to start and 'extent' represents the angle further which the arc should extend. These angles should be taken in counter clock-wise direction, taking the 3 O' clock position as 0 degrees. Thus, the 12 O' clock position will show 90 degrees, the 9 O' clock will be 180 and the 6 O' clock will represent 270 degrees.



start = 270, extent = 180

start = 90, extent = 180

start = 0, extent = 180

start = 180, extent = 180

start = 90, extent = 90

The value of the extent should be added to the starting angle so that we can understand where the arc will stop. For example,

```
id = c.create_arc(500, 100, 800, 300, width=3, start=90, extent=180,
outline="red", style="arc")
```

CANVAS

To display an image in the canvas with the help of create image() method. Using this method, we can display the images with the formats .gif,.pgm,or.ppm. We should first load the image into a file using Photo Image class as:

```
file1 = PhotoImage(file="cat.gif")# load cat.gif into file1
```

Now, the image is available in ‘file1’. This image can be displayed in the canvas using create image() method as:

```
id = c.create_image(500, 200, anchor=NE, image=file1,
     activeimage=file2)
```

CANVAS

Program: A Python program to display images in the canvas.

```
from tkinter import *

# create root window
root = Tk()

# create Canvas as a child to root window
c = Canvas(root, bg="white", height=700, width=1200)

# copy images into files
file1 = PhotoImage(file="cat.gif")
file2 = PhotoImage(file="puppy.gif")

# display the image in the canvas in NE direction
# when mouse is placed on cat image, we can see puppy image
id = c.create_image(500, 200, anchor=NE, image=file1,
     activeimage=file2)
# display some text below the image
id = c.create_text(500, 500, text="This is a thrilling photo", font=
     ('Helvetica', 30, 'bold'), fill="blue")

# add canvas to the root
c.pack()

# wait for any events
root.mainloop()
```

CANVAS

Output:

FRAME

A frame is similar to canvas that represents a rectangular area where some text or widgets can be displayed. Our root window is in fact a frame. To create a frame, we can create an object of Frame class as:

```
f= Frame(root, height=400, width=500, bg="yellow", cursor="cross")
```

Here, 'f' is the object of Frame class. The frame is created as a child of 'root' window. The options 'height' and 'width' represent the height and width of the frame in pixels. 'bg' represents the back ground color to be displayed and 'cursor' indicates the type of the cursor to be displayed in the frame.Once the frame is created, it should be added to the root window using the pack () method as follows:

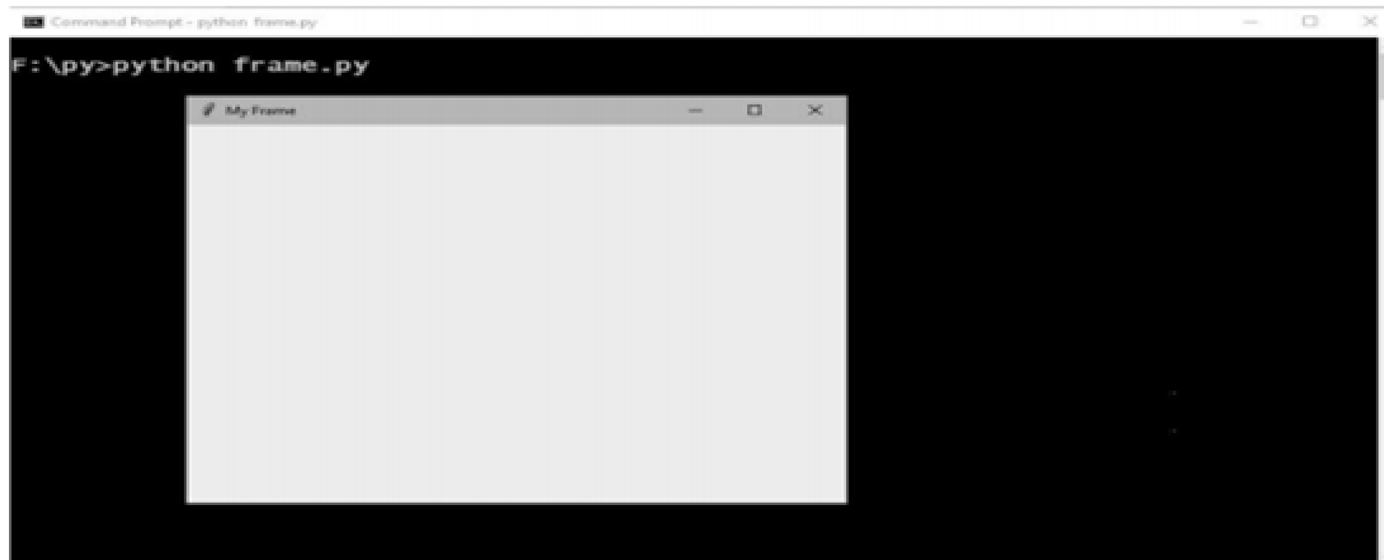
```
f.pack()
```

FRAME

Program: A GUI program to display a frame in the root window.

```
from tkinter import *

# create root window
root = Tk()

# give a title for root window
root.title("My Frame")

# create a frame as child to root window
f= Frame(root, height=400, width=500, bg="yellow", cursor="cross")

# attach the frame to root window
f.pack()

# let the root window wait for any events
root.mainloop()
```

Output:



WIDGETS

- A widget is a GUI component that is displayed on the screen and can perform a task as desired by the user. We create widgets as objects.
- For example, a push button is a widget that is nothing but an object of Button class. Similarly, label is a widget that is an object of Label class. Once a widget is created, it should be added to canvas or frame.
- The following are important widgets in Python:
- 1 Button 4 Text 7 Radio button 10 List box
- 2 Label 5 Scrollbar 8 Entry 11 Menu
- 3 Message 6 Check button 9 Spin box

WIDGETS

In general, working with widgets takes the following four steps:

1. Create the widgets that are needed in the program. A widget is a GUI component that is represented as an object of a class. For example, a push button is a widget that is represented as Button class object. As an example, suppose we want to create a push button, we can create an object to Button class as:

```
b = Button(f, text='My Button')
```

Here, ‘f’ is Frame object to which the button is added. ‘My Button’ is the text that is displayed on the button.

2. When the user interacts with a widget, he will generate an event. For example, clicking on a push button is an event. Such events should be handled by writing functions or routines. These functions are called in response to the events. Hence they are called 'callback handlers' or 'event handlers'. Other examples for events are pressing the Enter button, right clicking the mouse button, etc. As an example, let's write a function that may be called in response to button click.

```
    def buttonClick(self):
        print('You have clicked me')
```

Here, 'f' is Frame object to which the button is added. 'My Button' is the text that is displayed on the button.

3. When the user clicks on the push button, that ‘clicking’ event should be linked with the ‘callback handler’ function. Then only the button widget will appear as if it is performing some task. As an example, let’s bind the button click with the function as:

```
b.bind('<Button-1>', buttonClick)
```

Here, ‘b’ represents the push button. <Button-1> indicates the left mouse button. When the user presses the left mouse button, the ‘button Click’ function is called as these are linked by bind () method in the preceding code.

4. The preceding 3 steps make the widgets ready for the user. Now, the user has to interact with the widgets. This is done by entering text from the keyboard or pressing mouse button. These are called events. These events are continuously monitored by our program with the help of a loop, called 'event loop'. As an example, we can use the main loop() method that waits and processes the events as:

```
root.mainloop()
```

Here, 'root' is the object of root window in Python GUI. The events in root window are continuously observed by the main loop() method. It means clicking the mouse or pressing a button on the keyboard are accepted by main loop() and then the main loop() calls the corresponding even handler function.

BUTTON WIDGET

A push button is a component that performs some action when clicked. These buttons are created as objects of Button class as:

```
b = Button(f, text='My Button', width=15, height=2, bg='yellow',
        fg='blue', activebackground='green', activeforeground='red')
```

Here, ‘b’ is the object of Button class. ‘f’ represents the frame for which the button is created as a child. It means the button is shown in the frame. The ‘text’ option represents the text to be displayed on the button. ‘width’ represents the width of the button in characters. If an image is displayed on the button instead of text, then ‘width’ represents the width in pixels. ‘height’ represents the height of the button in textual lines. If an image is displayed on the button, then ‘height’ represents the height of the button in pixels.

BUTTON WIDGET

‘bg’ represents the foreground color and ‘fg’ represents the back ground color of the button. ‘activebackground’ represents the background color when the button is clicked. Similarly, ‘activeforeground’ represents the foreground color when the button is clicked.

We can also display an image on the button as:

```
# first load the image into file1
file1 = PhotoImage(file="cat.gif")

# create a push button with image
b = Button(f, image=file1, width=150, height=100, bg='yellow',
    fg='blue', activebackground='green', activeforeground='red')
```

In the preceding statement, observe that the width and height of the button are mentioned in pixels.

First Create a frame and then create a push button with some options and add the button to the frame. Then we link the mouse left button with the buttonClick () method using bind () method as:

```
b.bind('<Button-1>', buttonClick)
```

Here, <Button-1> represents the mouse left button that is linked withbuttonClick() method. It means when the mouse left button is clicked, the buttonClick() method is called. This method is called event handler.

BUTTON WIDGET

Program: A Python program to create a push button and bind it with an event handler function.

```
from tkinter import *

# method to be called when the button is clicked
def buttonClick(self):
    print('You have clicked me')

# create root window
root = Tk()

# create frame as child to root window
f = Frame(root, height=200, width=300)

# let the frame will not shrink
f.propagate(0)

# attach the frame to root window
f.pack()

# create a push button as child to frame
b = Button(f, text='My Button', width=15, height=2, bg='yellow',
     fg='blue', activebackground='green', activeforeground='red')

# attach button to the frame
b.pack()
# bind the left mouse button with the method to be called
b.bind("<Button-1>", buttonClick)

# the root window handles the mouse click event
root.mainloop()
```

Output:

LABEL WIDGET

A label represents constant text that is displayed in the frame or container. A label can display one or more lines of text that cannot be modified. A label is created as an object of Label class as:

```
lbl = Label(f, text="Welcome to Python", width=20, height=2,
        font=('Courier', -30, 'bold underline '), fg='blue', bg='yellow')
```

Here, 'f' represents the frame object to which the label is created as a child. 'text' represents the text to be displayed. 'width' represents the width of the label in number of characters and 'height' represents the height of the label in number of lines. 'font' represents a tuple that contains font name, size and style. 'fg' and 'bg' represents the foreground and background colors for the text.

LABEL WIDGET

Program: A Python program to display a label upon clicking a push button.

```
from tkinter import *

class MyButtons:
    # constructor
    def __init__(self, root):
        # create a frame as child to root window
        self.f = Frame(root, height=350, width=500)

        # let the frame will not shrink
        self.f.propagate(0)

        # attach the frame to root window
        self.f.pack()

        # create a push button and bind it to buttonClick method
        self.b1 = Button(self.f, text='Click Me', width=15, height=2,
        command=self.buttonClick)

        # create another button that closes the root window upon
        # clicking
        self.b2 = Button(self.f, text='Close', width=15, height=2,
        command=quit)

        # attach buttons to the frame
        self.b1.grid(row=0, column=1)
        self.b2.grid(row=0, column=2)
        # the event handler method
        def buttonClick(self):
        # create a label with some text
        self.lbl = Label(self.f, text="Welcome to Python", width=20,
                    height=2, font=('Courier', -30, 'bold underline '),
                    fg='blue')

        # attach the label in the frame
        self.lbl.grid(row=2, column=0)

# create root window
root = Tk()

# create an object to MyButtons class
mb = MyButtons(root)

# the root window handles the mouse click event
root.mainloop()
```

Output:

MESSAGE WIDGET

A message is similar to a label. But messages are generally used to display multiple lines of text where as a label is used to display a single line of text. All the text in the message will be displayed using the same font. To create a message, we need to create an object of Message class as:

```
m = Message(f, text='This is a message that has more than one line of
    text.', width=200, font=('Roman', 20, 'bold italic'), fg='dark
    goldenrod')
```

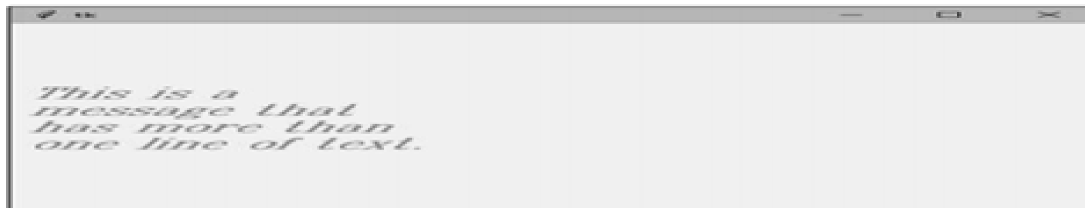
Here, ‘text’ represents the text to be displayed in the message. The ‘width’ option specifies the message width in pixels. ‘font’ represents the font for the message. We can use options ‘fg’ for specifying foreground color and ‘bg’ for specifying background color for the message text.

MESSAGE WIDGET

Program: A Python program to display a message in the frame.

```
from tkinter import *

class MyMessage:
    # constructor
    def __init__(self, root):
        # create a frame as child to root window
        self.f = Frame(root, height=350, width=500)

        # let the frame will not shrink
        self.f.propagate(0)

        # attach the frame to root window
        self.f.pack()

        # create a Message widget with some text
        self.m = Message(self.f, text='This is a message that has more
        than one line of text.', width=200, font=('Roman', 20, 'bold
            italic'), fg='dark goldenrod')

        # attach Message to the frame
        self.m.pack(side=LEFT)

# create root window
root = Tk()

# create an object to MyMessage class
mb = MyMessage(root)

# the root window handles the mouse click event
root.mainloop()
```

Output:



TEXT WIDGET

Text widget is same as a label or message. But Text widget has several options and can display multiple lines of text in different colors and fonts. It is possible to insert text into a Text widget, modify it or delete it. We can also display images in the Text widget. One can create a Text widget by creating an object to Text class as:

```
t = Text(root, width=20, height=10, font=('Verdana', 14, 'bold'),
    fg='blue', bg='yellow', wrap=WORD)
```

Once the Text widget is created, we can insert any text using the insert() method as:

```
t.insert(END, 'Text widget\nThis text is inserted into the Text
    widget.\n This is second line\n and this is third line\n')
```

TEXT WIDGET

It is possible to display an image like a photo using the image create() method as:

```
img = PhotoImage(file='moon.gif')# store moon.gif into img object
t.image_create(END, image=self.img)# append img to Text widget at the
# end
```

It is possible to mark some part of the text as a tag and provide different colors and font for that text. For this purpose, first we should specify the tag using the tag add() method as:

```
t.tag_add('start', '1.0', '1.11')
```

Here, the tag name is ‘start’. It contains characters (or text) from 1st row 0th character till 1st row 11th character. Now, we can apply colors and font to this tag text using the config() method as:

```
t.tag_config('start', background='red', foreground='white',
    font=('Lucida console', 20, 'bold italic'))
```

TEXT WIDGET

Program: A Python program to create a Text widget with a vertical scroll bar attached to it. Also, highlight the first line of the text and display an image in the Text widget.

```
from tkinter import *

class MyText:
    # constructor
    def __init__(self, root):
        # create a Text widget with 20 chars width and 10 lines height
        self.t = Text(root, width=20, height=10, font=('Verdana', 14,
            'bold'), fg='blue', bg='yellow', wrap=WORD)

        # insert some text into the Text widget
        self.t.insert(END, 'Text widget\nThis text is inserted into the
        Text widget.\n This is second line\n and this is third line\n')

        # attach Text to the root
        self.t.pack(side=LEFT)

        # show image in the Text widget
        self.img = PhotoImage(file='moon.gif')
        self.t.image_create(END, image=self.img)

        # create a tag with the name 'start'
        self.t.tag_add('start', '1.0', '1.11')

        # apply colors to the tag
        self.t.tag_config('start', background='red',
        foreground='white',font=('Lucida console', 20, 'bold italic'))

        # create a Scrollbar widget to move the text vertically
        self.s = Scrollbar(root, orient=VERTICAL, command=
        self.t.yview)

        # attach the scroll bar to the Text widget
        self.t.configure(yscrollcommand=self.s.set)

        # attach the scroll bar to the root window
        self.s.pack(side=RIGHT, fill=Y)

# create root window
root = Tk()

# create an object to MyText class
mt = MyText(root)
# the root window handles the mouse click event
root.mainloop()
```

TEXT WIDGET

Program: A Python program to create a Text widget with a vertical scroll bar attached to it. Also, highlight the first line of the text and display an image in the Text widget.

Output:

RADIO BUTTON WIDGET

A radio button is similar to a check button, but it is useful to select only one option from a group of available options. A radio button is displayed in the form of round shaped button. The user cannot select more than one option in case of radio buttons. When a radio button is selected, there appears a dot in the radio button. We can create a radio button as an object of the Radio button class as:

```
r1 = Radiobutton(f, bg='yellow', fg= 'green', font=('Georgia', 20,
     'underline'), text='Male', variable= var, value=1,
     command=display)
```

The option 'text' represents the string to be displayed after the radio button. 'variable' represents the object of IntVar class. 'value' represents a value that is set to this object when the radio button is clicked. The object of IntVar class can be created as:

```
var = IntVar()
```

RADIO BUTTON WIDGET

Program: A Python program to create radio buttons and know which button is selected by the user.

```
from tkinter import *

class Myradio:
    # constructor
    def __init__(self, root):
        # create a frame as child to root window
        self.f = Frame(root, height=350, width=500)

        # let the frame will not shrink
        self.f.propagate(0)

        # attach the frame to root window
        self.f.pack()

        # create IntVar class variable
        self.var = IntVar()

        # create radio buttons and bind them to display method
        self.r1 = Radiobutton(self.f, bg='yellow', fg= 'green',
            font=('Georgia', 20, 'underline'), text='Male',
            variable= self.var, value=1, command=self.display)
        self.r2 = Radiobutton(self.f, text='Female', variable=
        self.var,value=2, command=self.display)

        # attach radio buttons to the frame
        self.r1.place(x=50, y=100)
        self.r2.place(x=200, y=100)

    def display(self):
        # retrieve the control variable value
        x = self.var.get()

        # string is empty initially
        str = ''

        # catch user choice
        if x==1:
            str += 'You selected: Male '
        if x==2:
            str+= 'You selected: Female '

        # display the user selection as a label
        lbl = Label(text=str, fg='blue').place(x=50, y=150, width=200,
              height=20)
# create root window
root = Tk()

# create an object to MyButtons class
mb = Myradio(root)
# the root window handles the mouse click event
root.mainloop()
```

RADIO BUTTON WIDGET

Output:

ENTRY WIDGET

Entry widget is useful to create a rectangular box that can be used to enter or display one line of text. For example, we can display names, passwords or credit card numbers using Entry widgets. An Entry widget can be created as an object of Entry class as:

```
e1 = Entry(f, width=25, fg='blue', bg='yellow', font=('Arial', 14),
     show='*')
```

After typing text in the Entry widget, the user presses the Enter button. Such an event should be linked with the Entry widget using bind() method as:

```
e1.bind("<Return>", self.display)
```

When the user presses Enter (or Return) button, the event is passed to display () method. Hence, we are supposed to catch the event in the display method, using the following statement:

```
def display(self, event):
```

ENTRY WIDGET

Program: A Python program to create Entry widgets for entering user name and password and display the entered text.

```
from tkinter import *

class MyEntry:
    # constructor
    def __init__(self, root):
        # create a frame as child to root window
        self.f = Frame(root, height=350, width=500)

        # let the frame will not shrink
        self.f.propagate(0)

        # attach the frame to root window
        self.f.pack()

        # labels
        self.l1 = Label(text='Enter User name: ')
        self.l2 = Label(text='Enter Password: ')

        # create Entry widget for user name
        self.e1 = Entry(self.f, width=25, fg='blue', bg='yellow',
                  font=('Arial', 14))

        # create Entry widget for pass word. the text in the widget is
            # replaced by stars (*)
        self.e2 = Entry(self.f, width=25, fg='blue', bg='yellow',
                  show='*')

        # when user presses Enter, bind that event to display method
        self.e2.bind("<Return>", self.display)

        # place labels and entry widgets in the frame
        self.l1.place(x=50, y=100)
        self.e1.place(x=200, y=100)
        self.l2.place(x=50, y=150)
        self.e2.place(x=200, y=150)
    def display(self, event):
        # retrieve the values from the entry widgets
        str1 = self.e1.get()
        str2 = self.e2.get()

        # display the values using labels
        lbl1 = Label(text='Your name is: '+str1).place(x=50, y=200)
        lbl2 = Label(text='Your password is: '+str2).place(x=50, y=220)
# create root window
root = Tk()

# create an object to MyButtons class
mb = MyEntry(root)

# the root window handles the mouse click event
root.mainloop()
```

ENTRY WIDGET

Output: