LECTURE NOTES ON

OBJECT ORIENTED PROGRAMMING THROUGH PYTHON

B.Tech III Sem (IARE-R18)

By

Dr. K Suvarchala, Professor
Dr. D Govardhan, Professor

COMPUTER SCIENCE AND ENGINEERING

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

Module-I

Introduction to Python and Object Oriented Concepts

Introduction to Python

Programming languages like C, Pascal or FORTRAN concentrate more on the functional aspects of programming. In these languages, there will be more focus on writing the code using functions. For example, we can imagine a C program as a combination of several functions. Computer scientists thought that programming will become easy for human beings to understand if it is based on real life examples.

Hence, they developed Object Oriented Programming languages like Java and .NET where programming is done through classes and objects. Programmers started migrating from C to Java and Java soon became the most popular language in the software community. In Java, a programmer should express his logic through classes and objects only. It is not possible to write a program without writing at least one class! This makes programming lengthy.

For example, a simple program to add two numbers in Java looks like this:

```
//Java program to add two numbers

class Add //create a class

{

 public static void main(String args[]) //start execution

 {

int a, b; //take two variables

 a = b = 10; //store 10 in to a, b

 System.out.println("Sum= "+ (a+b)); //display their sum

}

}
```

Python

Python is a programming language that combines the features of C and Java. It offers elegant style of developing programs like C. When the programmers want to go for object orientation, Python offers classes and objects like Java. In Python, the program to add two numbers will be as follows:

```
#Python program to add two numbers

a = b = 10  #take two variables and store 10 in to them

 print("Sum=", (a+b))  #display their sum
```

Van Rossum picked the name Python for the new language from the TV show, Monty Python's Flying Circus. Python's first working version was ready by early 1990 and Van Rossum released it for

the public on February 20, 1991. The logo of Python shows two intertwined snakes as shown in Figure 1.1.

Figure 1.1: Python Official Logo

Python is open source software, which means anybody can freely download it from www.python.org and use it to develop programs. Its source code can be accessed and modified as required in the projects.

Features of Python

There are various reasons why Python is gaining good popularity in the programming community. The following are some of the important features of Python:

1. **Simple**: Python is a simple programming language. When we read a Python program, we feel like reading English sentences. It means more clarity and less stress on understanding the syntax of the language. Hence, developing and understanding programs will become easy.

2. **Easy to learn**: Python uses very few keywords. Its programs use very simple structure. So, developing programs in Python become easy. Also, Python resembles C language. Most of the language constructs in C are also available in Python. Hence, migrating from C to Python is easy for programmers.

3. **Open source**: There is no need to pay for Python software. Python can be freely downloaded from www.python.org website. Its source code can be read, modified and can be used in programs as desired by the programmers.

4. **High level language**: Programming languages are of two types: low level and high level. A low level language uses machine code instructions to develop programs. These instructions directly interact with the CPU. Machine language and assembly language are called low level languages. High level languages use English words to develop programs. These are easy to learn and use. Like COBOL, PHP or Java, Python also uses English words in its programs and hence it is called high level programming language.

5. **Dynamically typed**: In Python, we need not declare anything. An assignment statement binds a name to an object, and the object can be of any type. If a name is assigned to an object of one type, it may later be assigned to an object of a different type. This is the meaning of the saying that Python is a dynamically typed language. Languages like C and Java are statically typed. In these languages, the variable names and data types should be mentioned properly. Attempting to assign an object of the wrong type to a variable name triggers error or exception.

6. **Platform independent**: When a Python program is compiled using a Python compiler, it generates byte code. Python's byte code represents a fixed set of instructions that run on all operating systems and hardware. Using a Python Virtual Machine (PVM), anybody can run these byte code instructions on any computer system. Hence, Python programs are not dependent on any specific operating system. We can use Python on almost all operating systems like UNIX, Linux, Windows,Macintosh, Solaris, OS/2, Amiga, AROS, AS/400, etc. This makes Python an ideal programming language for any network or Internet.

7. **Portable**: When a program yields the same result on any computer in the world, then it is called a portable program. Python programs will give the same result since they are platform independent. Once a Python program is written, it can run on any computer system using PVM. However, Python also contains some system dependent modules (or code), which are specific to operating system. Programmers should be careful about such code while developing the software if they want it to be completely portable.

8. **Procedure and object oriented**: Python is a procedure oriented as well as an object oriented programming language. In procedure oriented programming languages (e.g. C and Pascal), the programs are built using functions and procedures. But in object oriented languages (e.g. C++ and Java), the programs use classes and objects.

Let's get some idea on objects and classes. An object is anything that exists physically in the real world. Almost everything comes in this definition. Let's take a dog with the name Snoopy. We can say Snoopy is an object since it physically exists in our house. Objects will have behavior represented by their attributes (or properties) and actions. For example, Snoopy has attributes like height, weight, age and color. These attributes are represented by variables in programming. Similarly, Snoopy can perform actions like barking, biting, eating, running, etc. These actions are represented by methods (functions) in programming. Hence, an object contains variables and methods.

A class, on the other hand, does not exist physically. A class is only an abstract idea which represents common behavior of several objects. For example, dog is a class. When we talk about dog, we will have a picture in our mind where we imagine a head, body, legs, tail, etc. This imaginary picture is called a class. When we take Snoopy, she has all the features that we have in our mind but she exists physically and hence she becomes the object of dog class. Similarly all the other dogs like Tommy, Charlie, Sophie, etc. exhibit same behavior like Snoopy. Hence, they are all objects of the same class, i.e. dog class. We should understand the point that the object Snoopy exists physically but the class dog does not exist physically. It is only a picture in our mind with some attributes and actions at abstract level. When we take Snoopy, Tommy, Charlie and Sophie, they have these attributes and actions and hence they are all objects of the dog class.

A class indicates common behavior of objects. This common behavior is represented by attributes and actions. Attributes are represented by variables and actions are performed by methods (functions). So, a class also contains variables and methods just like an object does. Figure 1.2 shows relationship between a class and its object:

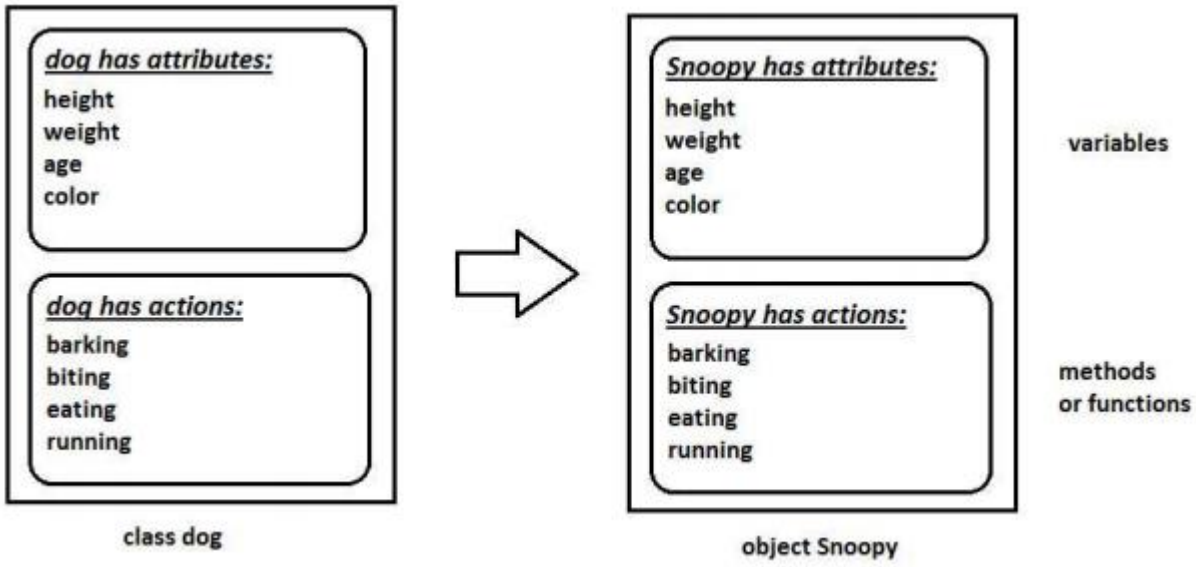


Fig:A Class and its object

Similarly, parrot, sparrow, pigeon and crow are objects of the bird class. We should understand that bird (class) is only an idea that defines some attributes and actions. A parrot and sparrow have the same attributes and actions but they exist physically. Hence, they are objects of the bird class.

Object oriented languages like Python, Java and .NET use the concepts of classes and objects in their programs. Since class does not exist physically, there will not be any memory allocated when the class is created. But, object exists physically and hence, a separate block of memory is allocated when an object is created. In Python language, everything like variables, lists, functions, arrays etc. are treated as objects.

a. **Interpreted**: A program code is called source code. After writing a Python program, we should compile the source code using Python compiler. Python compiler translates the Python program into an intermediate code called byte code. This byte code is then executed by PVM. Inside the PVM, an interpreter converts the byte code instructions into machine code so that the processor will understand and run that machine code to produce results.

b. **Extensible**: The programs or pieces of code written in C or C++ can be integrated into Python and executed using PVM. This is what we see in standard Python that is downloaded from www.python.org. There are other flavors of Python where programs from other languages can be integrated into Python. For example, Jython is useful to integrate Java code into Python programs and run on JVM (Java Virtual Machine). Similarly, Iron Python is useful to integrate .NET programs and libraries into Python programs and run on CLR (Common Language Runtime).

c. **Embeddable**: We can insert Python programs into a C or C++ program. Several applications are already developed in Python which can be integrated into other programming languages like C, C++, Delphi, PHP, Java and .NET. It means programmers can use these applications for their advantage in various software projects.

d. **Huge library**: Python has a big library which can be used on any operating system like UNIX, Windows or Macintosh. Programmers can develop programs very easily using the modules available in the Python library.

e. **Scripting language**: A scripting language is a programming language that does not use a compiler for executing the source code. Rather, it uses an interpreter to translate the source code into machine code on the fly (while running). Generally, scripting languages perform supporting tasks for a bigger application or software. For example, PHP is a scripting language that performs supporting task of taking input from an HTML page and send it to Web server software. Python is considered as a scripting language as it is interpreted and it is used on the Internet to support other software.

f. **Database connectivity**: A database represents software that stores and manipulates data. For example, Oracle is a popular database using which we can store data in the form of tables and manipulate the data. Python provides interfaces to connect its programs to all major databases like Oracle, Sybase or MySql.

g. **Scalable**: A program would be scalable if it could be moved to another operating system or hardware and take full advantage of the new environment in terms of performance. Python programs are scalable since they can run on any platform and use the features of the new platform effectively.

h. **Batteries included**: The huge library of Python contains several small applications (or small packages) which are already developed and immediately available to programmers. These small packages can be used and maintained easily. Thus the programmers need not download separate packages or applications in many cases. This will give them a head start in many projects. These libraries are called 'batteries included'.

Some interesting batteries or packages are given here:

1. **botois** Amazon web services library
2. **cryptography** offers cryptographic techniques for the programmers
3. **Fiona** reads and writes big data files
4. **jellyfish** is a library for doing approximate and phonetic matching of strings
5. **mysql-connector-pythonis** a driver written in Python to connect to MySQL database
6. **numpy** is a package for processing arrays of single or multidimensional type
7. **pandas** is a package for powerful data structures for data analysis, time series and statistics
8. **matplotlib** is a package for drawing electronic circuits and 2D graphs.
9. **pyquery** represents jquery-like library for Python
10. **w3lib** is a library of web related functions

Data types in Python

We have already written a Python program to add two numbers and executed it. Let's now view the program once again here:

```
#First Python program to add two numbers

a = 10

b = 15

c = a + b

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

When we compile this program using Python compiler, it converts the program source code into byte code instructions. This byte code is then converted into machine code by the interpreter inside the Python Virtual Machine (PVM). Finally, the machine code is executed by the processor in our computer and the result is produced as:

```
F:\PY>python first.py

Sum= 25
```

Observe the first line in the program. It starts with the #symbol. This is called the comment line. A comment is used to describe the features of a program. When we write a program, we may write some statements or create functions or classes, etc. in our program. All these things should be described using comments. When comments are written, not only our selves but also any programmer can easily understand our program. It means comments increase readability (understandability) of our programs.

Comments in Python

There are two types of comments in Python: single line comments and multi line comments. Single line comments .These comments start with a hash symbol (#) and are useful to mention that the entire line till the end should be treated as comment. For example,

```
#To find sum of two numbers

a = 10 #store 10 into variable a
```

Here, the first line is starting with a #and hence the entire line is treated as a comment. In the second line, a = 10 is a statement. After this statement, #symbol starts the comment describing that the value 10 is stored into variable 'a'. The part of this line starting from #symbol to the end is treated as a comment. Comments are non-executable statements. It means neither the Python compiler nor the PVM will execute them.

Multi line comments

When we want to mark several lines as comment, then writing #symbol in the beginning of every line will be a tedious job. For example,

```
#This is a program to find net salary of an employee

#based on the basic salary, provident fund, house rent allowance,
```

#dearness allowance and income tax.

Instead of starting every line with #symbol, we can write the previous block of code inside """ (triple double quotes) or ''' (triple single quotes) in the beginning and ending of the block as:

""" This is a program to find net salary of an employee based on the basic salary, provident fund, house rent allowance, dearness allowance and income tax. """

The triple double quotes (""") or triple single quotes (''') are called 'multi line comments' or 'block comments'. They are used to enclose a block of lines as comments.

Data types in Python

A data type represents the type of data stored into a variable or memory. The data types which are already available in Python language are called Built-in data types. The data types which can be created by the programmers are called User-defined data types. The built-in data types are of five types:

1. None Type
2. Numeric types
3. Sequences
4. Sets
5. Mappings

None Type

In Python, the 'None' data type represents an object that does not contain any value. In languages like Java, it is called 'null' object. But in Python, it is called 'None' object. In a Python program, maximum of only one 'None' object is provided. One of the uses of 'None' is that it is used inside a function as a default value of the arguments. When calling the function, if no value is passed, then the default value will be taken as 'None'. If some value is passed to the function, then that value is used by the function. In Boolean expressions, 'None' data type represents 'False'.

Numeric Types

The numeric types represent numbers. There are three sub types:

- int
- float
- complex

Int Data type

The int data type represents an integer number. An integer number is a number without any decimal point or fraction part. For example, 200, -50, 0, 9888998700, etc. are treated as integer numbers. Now, let's store an integer number -57 into a variable 'a'.

```
a = -57
```

Here, ‘a’ is called int type variable since it is storing -57 which is an integer value. In Python, there is no limit for the size of an int data type. It can store very large integer numbers conveniently.

Float Data type

The float data type represents floating point numbers. A floating point number is a number that contains a decimal point. For example, 0.5, -3.4567, 290.08, 0.001 etc. are called floating point numbers. Let’s store a float number into a variable ‘num’ as:

num = 55.67998

Here num is called float type variable since it is storing floating point value. Floating point numbers can also be written in scientific notation where we use ‘e’ or ‘E’ to represent the power of 10. Here ‘e’ or ‘E’ represents ‘exponentiation’. For example, the number 2.5 × 104 is written as 2.5E4. Such numbers are also treated as floating point numbers. For example,

x = 22.55e3

Here, the float value 22.55 × 103 is stored into the variable ‘x’. The type of the variable ‘x’ will be internally taken as float type. The convenience in scientific notation is that it is possible to represent very big numbers using less memory.

Complex Data type

A complex number is a number that is written in the form of a + bj or a + bJ. Here, ‘a’ represents the real part of the number and ‘b’ represents the imaginary part of the number. The suffix ‘j’ or ‘J’ after ‘b’ indicates the square root value of -1. The parts ‘a’ and ‘b’ may contain integers or floats. For example, 3+5j, -1-5.5J, 0.2+10.5J are all complex numbers. See the following statement:

c1 = -1-5.5J

Representing Binary, Octal and Hexadecimal Numbers

A binary number should be written by prefixing 0b (zero and b) or 0B (zero and B) before the value. For example, 0b110110, 0B101010011 are treated as binary numbers. Hexadecimal numbers are written by prefixing 0x (zero and x) or 0X (zero and big X) before the value, as 0xA180 or 0X11fb91 etc. Similarly, octal numbers are indicated by prefixing 0o (zero and small o) or 0O (zero and then O) before the actual value. For example, 0O145 or 0o773 are octal values. Converting the Data types Explicitly Depending on the type of data, Python internally assumes the data type for the variable. But sometimes, the programmer wants to convert one data type into another type on his own. This is called type conversion or coercion. This is possible by mentioning the data type with parentheses. For example, to convert a number into integer type, we can write int(num).

int(x) is used to convert the number x into int type. See the example:

x = 15.56

int(x) #will display 15

float(x) is used to convert x into float type.

For example,

num = 15 float(num) #will display 15.0

Bool Data type

The bool data type in Python represents boolean values. There are only two boolean values True or False that can be represented by this data type. Python internally represents True as 1 and False as 0. A blank string like "" is also represented as False. Conditions will be evaluated internally to either True or False. For example,

a = 10

b = 20

if(a<b):

print("Hello") #displays Hello.

In the previous code, the condition a<b which is written after if - is evaluated to True and hence it will execute print("Hello").

a = 10>5 #here ‘a’ is treated as bool type variable

print(a) #displays True

a = 5>10

print(a) #displays False

True+True will display 2 #True is 1 and false is 0

True-False will display 1

Sequences in Python

Generally, a sequence represents a group of elements or items. For example, a group of integer numbers will form a sequence. There are six types of sequences in Python:

1. str
2. bytes
3. bytearray
4. list
5. tuple
6. range

Str Data type

In Python, str represents string data type. A string is represented by a group of characters. Strings are enclosed in single quotes or double quotes. Both are valid.

str = “Welcome” #here str is name of string type variable

We can also write strings inside """ (triple double quotes) or ''' (triple single quotes) to span a group of lines including spaces.

Bytes Data type

The bytes data type represents a group of byte numbers just like an array does. A byte number is any positive integer from 0 to 255 (inclusive). bytes array can store numbers in the range from 0 to 255 and it cannot even store negative numbers. For example,

```
elements = [10, 20, 0, 40, 15] #this is a list of byte numbers

x = bytes(elements) #convert the list into bytes array

print(x[0]) #display 0th element, i.e 10
```

We cannot modify or edit any element in the bytes type array. For example, x[0] = 55 gives an error. Here we are trying to replace 0th element (i.e. 10) by 55 which is not allowed.

Bytearray Data type

The bytearray data type is similar to bytes data type. The difference is that the bytes type array cannot be modified but the bytearray type array can be modified. It means any element or all the elements of the bytearray type can be modified. To create a bytearray type array, we can use the function bytearray as:

```
elements = [10, 20, 0, 40, 15] #this is a list of byte numbers

x = bytearray(elements) #convert the list into bytearray type array

print(x[0]) #display 0th element,  i.e 10
```

We can modify or edit the elements of the bytearray. For example, we can write: x[0] = 88 #replace 0th element by 88 x[1] = 99 #replace 1st element by 99.

List Data type

Lists in Python are similar to arrays in C or Java. A list represents a group of elements. The main difference between a list and an array is that a list can store different types of elements but an array can store only one type of elements. Also, lists can grow dynamically in memory. But the size of arrays is fixed and they cannot grow at runtime. Lists are represented using square brackets [] and the elements are written in [], separated by commas. For example,

```
list = [10, -20, 15.5, 'Vijay', "Mary"]
```

will create a list with different types of elements. The slicing operation like [0: 3] represents elements from 0th to 2nd positions, i.e. 10, 20, 15.5.

Tuple Data type

A tuple is similar to a list. A tuple contains a group of elements which can be of different types. The elements in the tuple are separated by commas and enclosed in parentheses (). Whereas the list elements can be modified, it is not possible to modify the tuple elements. That means a tuple can be treated as a read-only list. Let’s create a tuple as:

tpl = (10, -20, 15.5, 'Vijay', "Mary")

The individual elements of the tuple can be referenced using square braces as tpl[0], tpl[1], tpl[2], … Now, if we try to modify the 0th element as:

tpl[0] = 99

This will result in error. The slicing operations which can be done on lists are also valid in tuples.

Range Data type

The range data type represents a sequence of numbers. The numbers in the range are not modifiable. Generally, range is used for repeating a for loop for a specific number of times. To create a range of numbers, we can simply write:

r = range(10)

Here, the range object is created with the numbers starting from 0 to 9.

Sets

A set is an unordered collection of elements much like a set in Mathematics. The order of elements is not maintained in the sets. It means the elements may not appear in the same order as they are entered into the set. Moreover, a set does not accept duplicate elements. There are two sub types in sets:

- set data type
- frozenset data type

Set Data type

To create a set, we should enter the elements separated by commas inside curly braces {}.

s = {10, 20, 30, 20, 50}

print(s) #may display {50, 10, 20, 30}

Please observe that the set ‘s’ is not maintaining the order of the elements. We entered the elements in the order 10, 20, 30, 20 and 50. But it is showing another order. Also, we repeated the element 20 in the set, but it has stored only one 20. We can use the set() function to create a set as:

ch = set("Hello")

print(ch) #may display {'H', 'e', 'l', 'o'}

Here, a set ‘ch’ is created with the characters H,e,l,o. Since a set does not store duplicate elements, it will not store the second ‘l’.

frozenset Data type

The frozenset data type is same as the set data type. The main difference is that the elements in the set data type can be modified; whereas, the elements of frozenset cannot be modified. We can create a frozenset by passing a set to frozenset() function as:

```
s = {50,60,70,80,90}

print(s) #may display {80, 90, 50, 60, 70}

fs = frozenset(s) #create frozenset fs

print(fs) #may display frozenset({80, 90, 50, 60, 70})
```

Mapping Types

A map represents a group of elements in the form of key value pairs so that when the key is given, we can retrieve the value associated with it. The dict datatype is an example for a map. The 'dict' represents a 'dictionary' that contains pairs of elements such that the first element represents the key and the next one becomes its value. The key and its value should be separated by a colon (:) and every pair should be separated by a comma. All the elements should be enclosed inside curly brackets {}. We can create a dictionary by typing the roll numbers and names of students. Here, roll numbers are keys and names will become values. We write these key value pairs inside curly braces as:

```
d = {10: 'Kamal', 11: 'Pranav', 12: 'Hasini', 13: 'Anup', 14: 'Reethu'}
```

Here, d is the name of the dictionary. 10 is the key and its associated value is 'Kamal'.

The next key is 11 and its value is 'Pranav'. Similarly 12 is the key and 'Hasini' is it value. 13 is the key and 'Anup' is the value and 14 is the key and 'Reethu' is the value. We can create an empty dictionary without any elements as:

```
d = {}
```

Literals in Python

A literal is a constant value that is stored into a variable in a program. Observe the following statement:

```
a = 15
```

Here, 'a' is the variable into which the constant value '15' is stored. Hence, the value 15 is called 'literal'. Since 15 indicates integer value, it is called 'integer literal'. The following are different types of literals in Python.

1. Numeric literals
2. Boolean literals
3. String literals

Numeric Literals

Examples	Literal name
450, -15	Integer literal
3.14286, -10.6, 1.25E4	Float literal

These literals represent numbers. Please observe the different types of numeric literals available in Python.

Boolean Literals

Boolean literals are the True or False values stored into a bool type variable.

String Literals

A group of characters is called a string literal. These string literals are enclosed in single quotes (') or double quotes (") or triple quotes ("'or"""). In Python, there is no difference between single quoted strings and double quoted strings. Single or double quoted strings should end in the same line as:

s1 = 'This is first Indian book'

s2 = "Core Python"

User-defined Data types

The data types which are created by the programmers are called 'user-defined' data types. For example, an array, a class, or a module is user-defined data types.

Constants in Python

A constant is similar to a variable but its value cannot be modified or changed in the course of the program execution. We know that the variable value can be changed whenever required. But that is not possible for a constant. Once defined, a constant cannot allow changing its value. For example, in Mathematics, 'pi' value is 22/7 which never changes and hence it is a constant. In languages like C and Java, defining constants is possible. But in Python, that is not possible. A programmer can indicate a variable as constant by writing its name in all capital letters. For example, MAX_VALUE is a constant. But its value can be changed.

Identifiers and Reserved words

An identifier is a name that is given to a variable or function or class etc. Identifiers can include letters, numbers, and the underscore character (_). They should always start with a nonnumeric character. Special symbols such as?, #, $,%, and @ are not allowed in identifiers. Some examples for identifiers are salary, name11, gross_income, etc. We should also remember that Python is a case sensitive programming language. It means capital letters and small letters are identified separately by Python. For example, the names 'num' and 'Num' are treated as different names and hence represent different variables. Figure 3.12 shows examples of a variable, an operator and a literal:

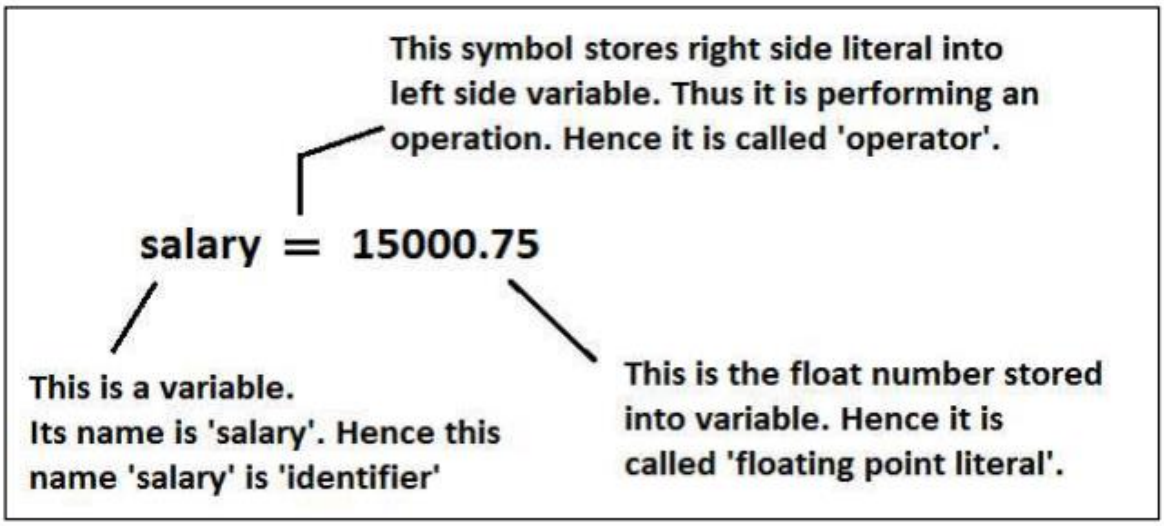


Figure 3.12: Variable, Operator and Literal

Reserved words are the words that are already reserved for some particular purpose in the Python language. The names of these reserved words should not be used as identifiers. The following are the reserved words available in Python:

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

Naming Conventions in Python

Python developers made some suggestions to the programmers regarding how to write names in the programs. The rules related to writing names of packages, modules, classes, variables, etc. are called naming conventions. The following naming conventions should be followed:

1. Packages: Package names should be written in all lower case letters. When multiple words are used for a name, we should separate them using an underscore (_).

2. Modules: Modules names should be written in all lower case letters. When multiple words are used for a name, we should separate them using an underscore (_).

3. Classes: Each word of a class name should start with a capital letter. This rule is applicable for the classes created by us. Python's built-in class names use all lowercase words. When a class represents exception, then its name should end with a word 'Error'.

4. Global variables or Module-level variables: Global variables names should be all lower case letters. When multiple words are used for a name, we should separate them using an underscore (_).

5. Instance variables: Instance variables names should be all lower case letters. When multiple words are used for a name, we should separate them using an underscore (_). Non-public instance variable name should begin with an underscore.

6. Functions: Function names should be all lower case letters. When multiple words are used for a name, we should separate them using an underscore (_).

OPERATORS IN PYTHON

Operator An operator is a symbol that performs an operation. An operator acts on some variables called operands. For example, if we write a + b, the operator ' + ' is acting on two operands 'a' and 'b'. If an operator acts on a single variable, it is called unary operator. If an operator acts on two variables, it is called binary operator. If an operator acts on three variables, then it is called ternary operator. This is one type of classification. We can classify the operators depending upon their nature, as shown below:

1. Arithmetic operators
2. Assignment operators
3. Unary minus operator
4. Relational operators
5. Logical operators
6. Boolean operators
7. Bitwise operators

INPUT AND OUTPUT

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

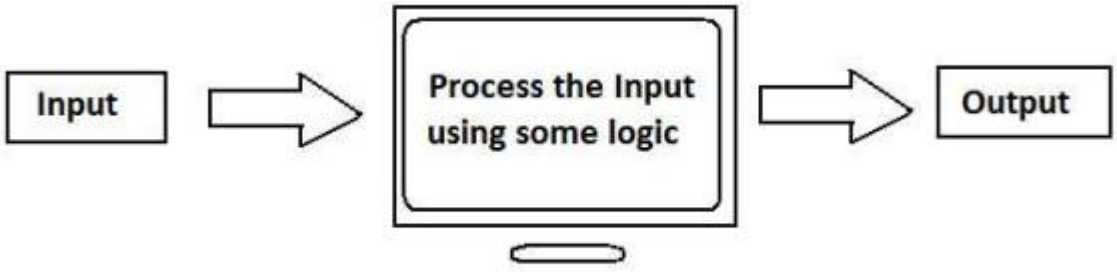


Figure 5.1: Processing Input by the Computer

To provide input to a computer, Python provides some statements which are called Input statements. Similarly, to display the output, there are Output statements available in Python. We should use some logic to convert the input into output. This logic is implemented in the form of several topics in subsequent chapters. We will discuss the input and output statements in this chapter, but in the following order:

- Output statements
- Input statements

Output statements

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

The print() Statement

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

The print(formatted string)

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

print("formatted string"% (variables list))

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

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

value= 10

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

Input Statements

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

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

Enter your name: Raj kumar

```
print(str)
```

Raj kumar

Control Statements

Control statements are statements which control or change the flow of execution. The following are the control statements available in Python:

1. if statement
2. if … else statement
3. if … elif … else statement
4. while loop
5. for loop
6. else suite
7. break statement
8. continue statement
9. pass statement
10. assert statement
11. return statement

The if Statement

This statement is used to execute one or more statement depending on whether a condition is True or not. The syntax or correct format of if statement is given below:

```
if condition:
   statements
```

First, the condition is tested. If the condition is True, then the statements given after colon (:) are executed. We can write one or more statements after colon (:). If the condition is False, then the statements mentioned after colon are not executed.

The if … else Statement

The if … else statement executes a group of statements when a condition is True; otherwise, it will execute another group of statements. The syntax of if … else statement is given below:

```
if condition:
   statements1
else:
   statements2
```

If the condition is True, then it will execute statements1 and if the condition is False, then it will execute statements2. It is advised to use 4 spaces as indentation before statements1 and statements2.

The if … elif … else Statement

Sometimes, the programmer has to test multiple conditions and execute statements depending on those conditions. if … elif … else statement is useful in such situations. Consider the following syntax of if … elif … else statement:

```
if condition1:
    statements1
elif condition2:
    statements2
else:
    statements3
```

When condition1 is True, the statements1 will be executed. If condition1 is False, then condition2 is evaluated. When condition2 is True, the statements2 will be executed. When condition 2 is False, the statements3 will be executed. It means statements3 will be executed only if none of the conditions are True.

The while Loop

A statement is executed only once from top to bottom. For example, 'if' is a statement that is executed by Python interpreter only once. But a loop is useful to execute repeatedly. For example, while and for are loops in Python. They are useful to execute a group of statements repeatedly several times.

The while loop is useful to execute a group of statements several times repeatedly depending on whether a condition is True or False. The syntax or format of while loop is:

```
while condition:
    statements
```

The for Loop

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

```
for var in sequence:
    statements
```

The else Suite

In Python, it is possible to use 'else' statement along with for loop or while loop. The statements written after 'else' are called suite. The else suite will be always executed irrespective of the statements in the loop are executed or not. For example,

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

```
    print("Yes")
else:
    print("No")
```

The break Statement

The break statement can be used inside a for loop or while loop to come out of the loop. When 'break' is executed, the Python interpreter jumps out of the loop to process the next statement in the program. Now, suppose we want to display x values up to 6 and if it is 5 then we want to come out of the loop, we can introduce a statement like this:

```
if x==5: #if x is 5 then come out from while loop
    break
```

The continue Statement

The continue statement is used in a loop to go back to the beginning of the loop. It means, when continue is executed, the next repetition will start. When continue is executed, the subsequent statements in the loop are not executed.

The pass Statement

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

The assert Statement

The assert statement is useful to check if a particular condition is fulfilled or not.

assert expression, message

In the above syntax, the 'message' is not compulsory. Let's take an example. If we want to assure that the user should enter only a number greater than 0, we can use assert statement as:

```
assert x>0, "Wrong input entered"
```

In this case, the Python interpreter checks if x>0 is True or not. If it is True, then the next statements will execute, else it will display AssertionError along with the message "Wrong input entered".

The return Statement

A function represents a group of statements to perform a task. The purpose of a function is to perform some task and in many cases a function returns the result. A function starts with the keyword defthat represents the definition of the function. After 'def', the function should be written. Then we should write variables in the parentheses. For example,

```
def sum(a, b):
    function body
```

After the function name, we should use a colon (:) to separate the name with the body. The body of the statements contains logic to perform the task. For example, to find sum of two numbers, we can write: def

```
sum(a, b):
    print(a+b)
```

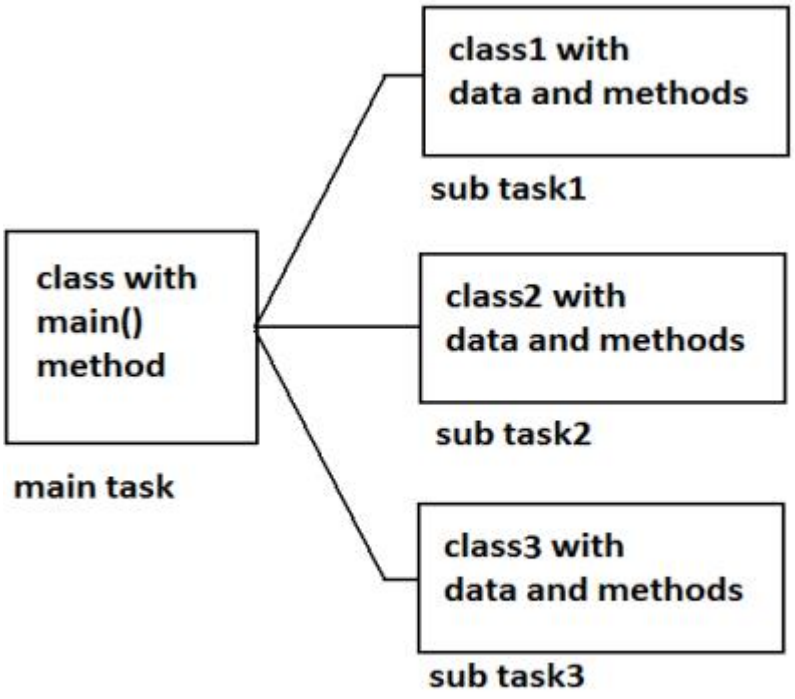
A function is executed only when it is called. At the time of calling the sum() function, we should pass values to variables a and b. So, we can call this function as:

```
sum(5, 10)
```

Now, the values 5 and 10 are passed to a and b respectively and the sum() function displays their sum. In Program 30, we will now write a simple function to perform sum of two numbers.

Introduction to Object Oriented Concepts

Languages like C++, Java and Python use classes and objects in their programs and are called Object Oriented Programming languages. A class is a module which itself contains data and methods (functions) to achieve the task. The main task is divided into several sub tasks, and these are represented as classes. Each class can perform several inter-related tasks for which several methods are written in a class. This approach is called Object Oriented approach.



Object Oriented Approach

Features of Object Oriented Programming System (OOPS)

There are five important features related to Object Oriented Programming System. They are:

1. Classes and objects

2. Encapsulation

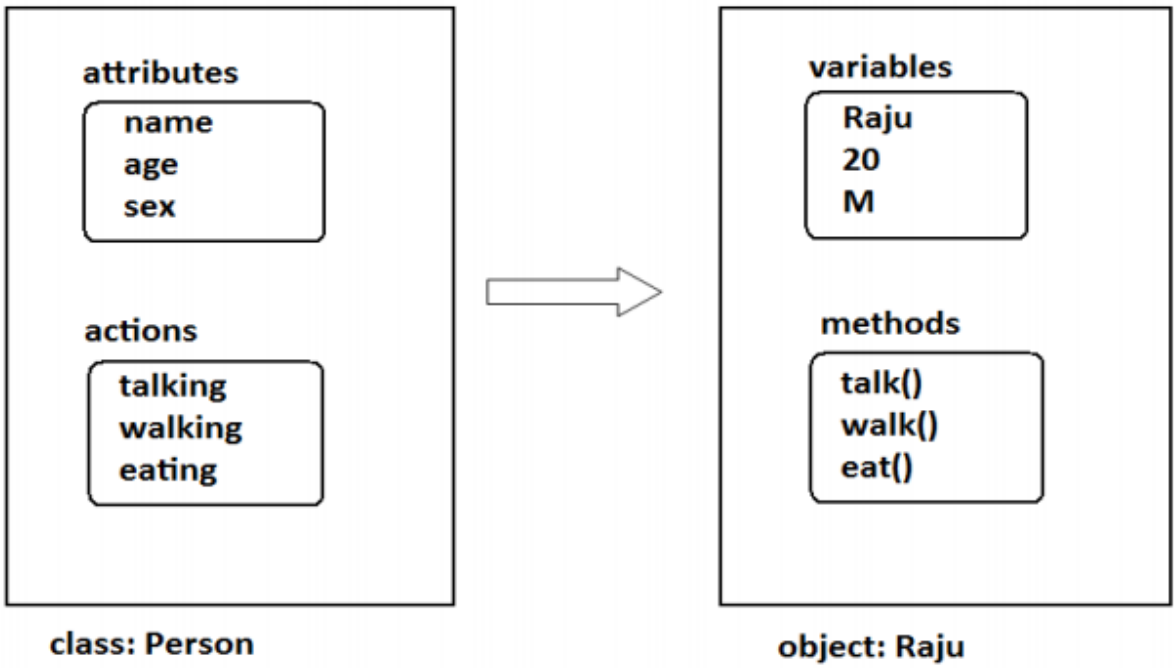
3. Abstraction

4. Inheritance

5. Polymorphism

Classes and Objects

An object is anything that really exists in the world and can be distinguished from others. This definition specifies that everything in this world is an object. For example, a table, a ball, a car, a dog, a person, etc. will come under objects. Then what is not an object? If something does not really exist, then it is not an object. For example, our thoughts, imagination, plans, ideas etc. are not objects, because they do not physically exist.



Encapsulation

Encapsulation is a mechanism where the data (variables) and the code (methods) that act on the data will bind together. For example, if we take a class, we write the variables and methods inside the class. Thus, class is binding them together. So class is an example for encapsulation.

The variables and methods of a class are called ‘members’ of the class. All the members of a class are by default available outside the class. That means they are public by default. Public means available to other programs and classes.

Encapsulation in Python

Encapsulation is nothing but writing attributes (variables) and methods inside a class. The methods process the data available in the variables. Hence data and code are bundled up together in the class. For example, we can write a Student class with ‘id’ and ‘name as attributes along with the display() method that displays this data. This Student class becomes an example for encapsulation.

Abstraction

There may be a lot of data, a class contains and the user does not need the entire data. The user requires only some part of the available data. In this case, we can hide the unnecessary data from the user and expose only that data that is of interest to the user. This is called abstraction.

A good example for abstraction is a car. Any car will have some parts like engine, radiator, battery, mechanical and electrical equipment etc. The user of the car (driver) should know how to drive the car and does not require any knowledge of these parts. For example driver is never bothered about how the engine is designed and the internal parts of the engine. This is why the car manufacturers hide these parts from the driver in a separate panel, generally at the front of the car.

Inheritance

Creating new classes from existing classes, so that the new classes will acquire all the features of the existing classes is called Inheritance. A good example for Inheritance in nature is parents producing the children and children inheriting the qualities of the parents.

Let's take a class A with some members i.e., variables and methods. If we feel another class B wants almost same members, then we can derive or create class B from A as:

```
class B(A):
```

Now, all the features of A are available to B. If an object to B is created, it contains all the members of class A and also its own members. Thus, the programmer can access and use all the members of both the classes A and B. Thus, class B becomes more useful. This is called inheritance. The original class (A) is called the base class or super class and the derived class (B) is called the sub class or derived class.

Polymorphism

The word 'Polymorphism' came from two Greek words 'poly' meaning 'many' and 'morphos' meaning 'forms'. Thus, polymorphism represents the ability to assume several different forms. In programming, if an object or method is exhibiting different behavior in different contexts, it is called polymorphic nature.

Polymorphism provides flexibility in writing programs in such a way that the programmer uses same method call to perform different operations depending on the requirement.

Module – II

Python Classes and Objects

We know that a class is a model or plan to create objects. This means, 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. The same variables and methods are also available in the objects because they are created from the class. These variables are also called 'instance variables' because they are created inside the instance (i.e. object). Please remember the difference between a function and a method. 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

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. Please note that writing 'object' is not compulsory since it is implied.

For example, a student has attributes like name, age, marks, etc. These attributes should be written inside the Student class as variables. Similarly, a student can perform actions like talking, writing, reading, etc. These actions should be represented by methods in the Student class. So, the class Student contains these attributes and actions, as shown here:

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

def talk(self):

print('Hi, I am ', self.name)

print('My age is', self.age)

print('My marks are', self.marks)

See the method talk(). This method also takes the 'self' variable as parameter. This method displays the values of the variables by referring them using 'self'.

The methods that act on instances (or objects) of a class are called instance methods. Instance methods use 'self' as the first parameter that refers to the location of the instance in the memory. Since instance methods know the location of instance, they can act on the instance variables. In the previous code, the two methods __init__(self) and talk(self) are called instance methods. In the Student class, a student is talking to us through talk() method. He is introducing himself to us, as shown here: Hi, I am Vishnu My age is 20 My marks are 900 This is what the talk() method displays. Writing a class like this is not sufficient. It should be used. To use a class, we should create an instance (or object) to the class. Instance creation represents allotting memory necessary to store the actual data of the variables, i.e., Vishnu, 20 and 900. 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()

Here, 's1' is nothing but the instance name. 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'.

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).

Now, 's1' refers to the instance of the Student class. Hence any variables or methods in the instance can be referenced by 's1' using dot operator as:

s1.name #this refers to data in name variable, i.e. Vishnu

s1.age #this refers to data in age variable, i.e. 20

s1.marks #this refers to data in marks variable, i.e. 900

s1.talk() #this calls the talk() method.

The dot operator takes the instance name at its left and the member of the instance at the right hand side. Figure 2.1 shows how ‘s1’ instance of Student class is created in memory:

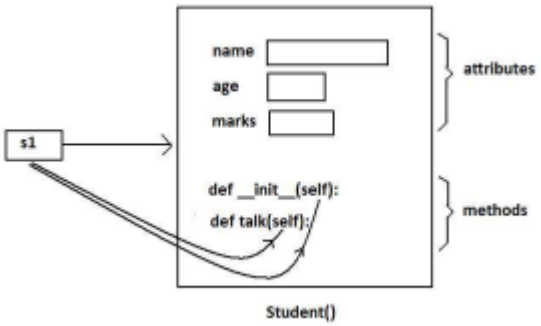


Figure 2.1: Student class instance in memory

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)

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

In Program 1, we used the 'self' variable to refer to the instance of the same class. Also, we used a special method '__init__(self)' that initializes the variables of the instance. Let's have more clarity on these two concepts.

The Self Variable

'self' is a default variable that contains the memory address of the instance of the current class. So, we can use 'self' to refer to all the instance variables and instance methods. When an instance to the class is created, the instance name contains the memory location of the instance. This memory location is internally passed to 'self'. For example, we create an instance to Student class as:

s1 = Student()

Here, 's1' contains the memory address of the instance. This memory address is internally and by default passed to 'self' variable. Since 'self' knows the memory address of the instance, it can refer to all the members of the instance. We use 'self' in two ways:

- 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.

- '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. If this method wants to act on the instance variables, it should know the memory location of the instance variables. That memory location is by default available to the talk() method through 'self'.

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

Here, the constructor has only one parameter, i.e. 'self'. Using 'self.name' and 'self.marks', we can access the instance variables of the class. A constructor is called at the time of creating an instance. So, the above constructor will be called when we create an instance as: s1 = Student()

Here, 's1' is the name of the instance. Observe the empty parentheses after the class name 'Student'. These empty parentheses represent that we are not passing any values to the constructor. Suppose, we want to pass some values to the constructor, then we have to pass them in the parentheses after the class name. Let's take another example. We can write a constructor with some parameters in addition to 'self' as:

```
def __init__(self, n = '', m=0):

self.name = n

self.marks = m
```

Here, the formal arguments are 'n' and 'm' whose default values are given as '' (None) and 0 (zero). Hence, if we do not pass any values to constructor at the time of creating an instance, the default values of these formal arguments are stored into name and marks variables. For example,

```
s1 = Student()
```

Since we are not passing any values to the instance, None and zero are stored into name and marks. Suppose, we create an instance as:

```
s1 = Student('Lakshmi Roy', 880)
```

In this case, we are passing two actual arguments: 'Lakshmi Roy' and 880 to the Student instance. Hence these values are sent to the arguments 'n' and 'm' and from there stored into name and marks variables. We can understand this concept from Program.

Program

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):
```

```
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

------------------
```

We should understand that a constructor does not create an instance. The duty of the constructor is to initialize or store the beginning values into the instance variables. A constructor is called only once at the time of creating an instance. Thus, if 3 instances are created for a class, the constructor will be called once per each instance, thus it is called 3 times.

Types of Variables

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

- Instance variables
- Class variables or Static variables

Instance variables are the variables whose separate copy is created in every instance (or object). For example, if 'x' is an instance variable and if we create 3 instances, there will be 3 copies of 'x' in these 3 instances. When we modify the copy of 'x' in any instance, it will not modify the other two copies. Consider Program.

Program

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()

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
```

Instance variables are defined and initialized using a constructor with 'self' parameter. Also, to access instance variables, we need instance methods with 'self' as first parameter. It is possible that the instance methods may have other parameters in addition to the 'self' parameter. To access the instance variables, we can use self.variable as shown in Program. It is also possible to access the instance variables from outside the class, as: instancename.variable, e.g. s1.x.

Unlike instance variables, class variables are the variables whose single copy is available to all the instances of the class. If we modify the copy of class variable in an instance, it will modify all the copies in the other instances. For example, if 'x' is a class variable and if we create 3 instances, the same copy of 'x' is passed to these 3 instances. When we modify the copy of 'x' in any instance using a class method, the modified copy is sent to the other two instances. This can be easily grasped from Program. Class variables are also called static variables.

Program

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()

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
```

Observe Program. The class variable ‘x’ is defined in the class and initialized with value 10. A method by the name ‘modify’ is used to modify the value of ‘x’. This method is called ‘class method’ since it is acting on the class variable. To mark this method as class method, we should use built-in decorator statement @classmethod. For example,

```
@classmethod      #this is a decorator

def modify(cls):          #cls must be the first parameter

cls.x+=1                  #cls.x refers to class variable x
```

Namespaces

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

n = 10

Here, ‘n’ is the name given to the integer object 10. Please recollect that numbers, strings, lists etc. are all considered as objects in Python. The name ‘n’ is linked to 10 in the namespace. A class maintains its own namespace, called ‘class namespace’. In the class namespace, the names are mapped to class variables. Similarly, every instance will have its own name space, called ‘instance namespace’. In the instance namespace, the names are mapped to instance variables. In the following code, ‘n’ is a class variable in the Student class. So, in the class namespace, the name ‘n’ is mapped or linked to 10 as shown Figure 2. Since it is a class variable, we can access it in the class namespace, using classname.variable, as: Student.n which gives 10.

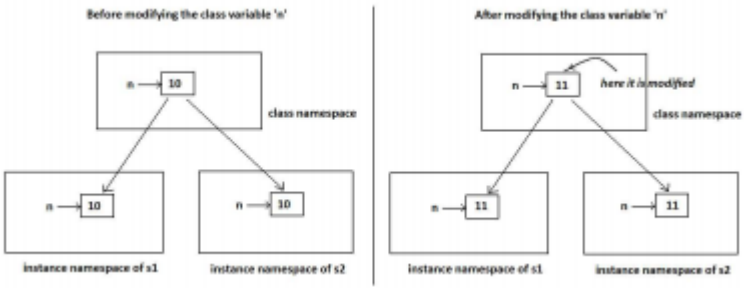


Figure 2: Modifying the class variable in the class namespace

```
#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

We know that a single copy of class variable is shared by all the instances. So, if the class variable is modified in the class namespace, since same copy of the variable is modified, the modified copy is available to all the instances. This is shown in Figure 2.

#modified class var is seen in all instances

s1 = Student() #create s1 instance

print(s1.n) #displays 11

s2 = Student() #create s2 instance

print(s2.n) #displays 11

If the class variable is modified in one instance namespace, it will not affect the variables in the other instance namespaces. This is shown in Figure 3. To access the class variable at the instance level, we have to create instance first and then refer to the variable as instancename.variable.

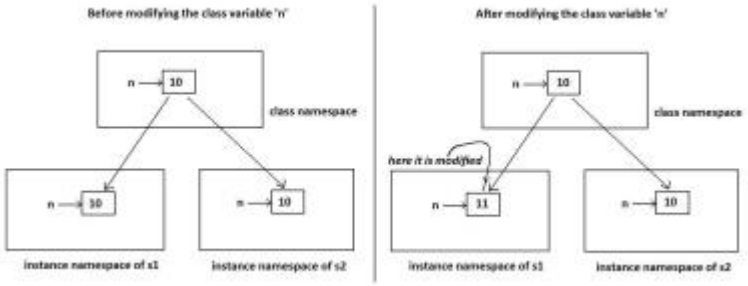


Figure 3: Modifying the class variable in the instance namespace

#understanding instance namespace

class Student:

#this is a class var

n=10

#access class var in the s1 instance namespace

s1 = Student() print(s1.n)

#displays 10

s1.n+=1

#modify it in s1 instance namespace

print(s1.n) #displays 11

As per the above code, we created an instance ‘s1’ and modified the class variable ‘n’ in that instance. So, the modified value of ‘n’ can be seen only in that instance. When we create other instances like ‘s2’, there will be still the original value of ‘n’ available. See the code below:

```
#modified class var is not seen in other instances

s2 = Student()

#this is another instance

print(s2.n) #displays 10, not 11
```

Types of Methods

The purpose of a method is to process the variables provided in the class or in the method. We already know that the variables declared in the class are called class variables (or static variables) and the variables declared in the constructor are called instance variables. We can classify the methods in the following 3 types:

- ❖Instance methods (a) Accessor methods (b) Mutator methods
- ❖Class methods
- ❖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(). Since instance variables are available in the instance, instance methods need to know the memory address of the instance. This is provided through ‘self’ variable by default as first parameter for the instance method. While calling the instance methods, we need not pass any value to the ‘self’ variable.

In this program, we are creating a Student class with a constructor that defines ‘name’ and ‘marks’ as instance variables. An instance method display() will display the values of these variables. We added another instance methods by the name calculate() that calculates the grades of the student depending on the ‘marks’.

Program

Program 5: A Python program using a student class with instance methods to process the data of several students.

```
#instance methods to process data of the objects

class Student:

#this is a constructor.

def __init__(self, n = ‘’, m=0):

self.name = n

self.marks = m
```

```
#this is an instance method.
def display(self):
print('Hi', self.name)
print('Your marks', self.marks)
#to calculate grades based on marks.
def calculate(self):
if(self.marks>=600):
print('You got first grade')
elif(self.marks>=500):
print('You got second grade')
elif(self.marks>=350):
print('You got third grade')
else:
print('You are failed')
#create instances with some data from keyboard
n = int(input('How many students? '))
 i=0
while(i<n):
name = input('Enter name: ')
marks = int(input('Enter marks: '))
#create Student class instance and store data
s = Student(name, marks)
s.display()
s.calculate()
 i+=1
print('----------')
```

Output:

```
C:\>python cl.py
```

How many students? 3

Enter name: Vishnu Vardhan

Enter marks: 800

Hi Vishnu Vardhan

Your marks 800

You got first grade ----------

Enter name: Tilak Prabhu

Enter marks: 360

Hi Tilak Prabhu

Your marks 360

You got third grade ----------

Enter name: Gunasheela

Enter marks: 550

Hi Gunasheela

Your marks 550

You got second grade ----------

Instance methods are of two types: accessor methods and mutator methods. Accessor methods simply access or read data of the variables. They do not modify the data in the variables. Accessor methods are generally written in the form of getXXX() and hence they are also called getter methods. For example,

```
def getName(self):
    return self.name
```

Here, getName() is an accessor method since it is reading and returning the value of 'name' instance variable. It is not modifying the value of the name variable. On the other hand, mutator methods are the methods which not only read the data but also modify them. They are written in the form of setXXX() and hence they are also called setter methods. For example,

```
def setName(self, name):
 self.name = name
```

Here, setName() is a mutator method since it is modifying the value of 'name' variable by storing new name. In the method body, 'self.name' represents the instance variable 'name' and the right hand side 'name' indicates the parameter that receives the new value from outside. In Program, we are redeveloping the Student class using accessor and mutator methods.

Program

Program 6: 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

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())
```

```
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

----------
```

Since mutator methods define the instance variables and store data, we need not write the constructor in the class to initialize the instance variables. This is the reason we did not use constructor in Student class in the above Program.

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. For example, 'cls.var' is the format to refer to the class variable. These methods are generally called using the classname.method(). The processing which is commonly needed by all the instances of the class is handled by the class methods. In Program 7, we are going to develop Bird class. All birds in the Nature have only 2 wings. So, we take 'wings' as a class variable. Now a copy of this class variable is available to all the instances of Bird class. The class method fly() can be called as Bird.fly().

Program

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 information for 2 birds

Bird.fly('Sparrow')

Bird.fly('Pigeon')
```

Output:

```
C:\>python cl.py

Sparrow flies with 2 wings

Pigeon flies with 2 wings
```

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. For example, setting environmental variables, counting the number of instances of the class or changing an attribute in another class, etc. are the tasks related to a class. Such tasks are handled by static methods. Also, static methods can be used to accept some values, process them and return the result. In this case the involvement of neither the class nor the objects is needed. Static methods are written with a decorator @staticmethod above them. Static methods are called in the form of classname.method(). In Program 8, we are creating a static method noObjects() that counts the number of objects or instances created to Myclass. In Myclass, we have written a constructor that increments the class variable 'n' every time an instance is created. This incremented value of 'n' is displayed by the noObjects() method.

Program

Program 8: A Python program to create a static method that counts the number of instances created for a class.

```
 #understanding static methods  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

#this is a static method to display the no. of instances
```

```
@staticmethod  def noObjects():

print('No. of instances created: ', Myclass.n)

#create 3 instances

obj1 = Myclass()

obj2 = Myclass()

obj3 = Myclass()

Myclass.noObjects()
```

Output:

```
C:\>python cl.py

No. of instances created: 3
```

In the next program, we accept a number from the keyboard and return the result of its square root value. Here, there is no need of class or object and hence we can write a static method to perform this task.

Inheritance and Polymorphism

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 9: 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

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
```

When the programmer wants to use this Teacher class that is available in teacher.py file, he can simply import this class into his program and use it as shown here:

Program

Program 10: 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
```

So, the program is working well. There is no problem. Once the Teacher class is completed, the programmer stored teacher.py program in a central database that is available to all the members of the team. So, Teacher class is made available through the module teacher.py, as shown in Figure:

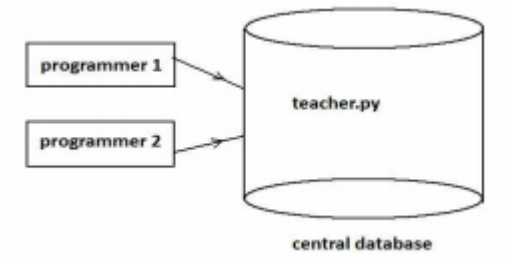


Figure 4: The teacher.py module is created and available in the project database

Now, another programmer in the same team wants to create a Student class. He is planning the Student class without considering the Teacher class as shown in Program 11.

Program

Program 11: A Python program to create Student class and store it into student.py module.

```
#this is Student class –v1.0. save it as student.py

class Student:

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

def setaddress(self, address):

self.address = address

def getaddress(self):

return self.address

def setmarks(self, marks):

self.marks = marks

def getmarks(self):
```

return self.marks

Now, the second programmer who created this Student class and saved it as student.py can use it whenever he needs. Using the Student class is shown in Program 12.

Program

Program 12: A Python program to use the Student class which is already available in student.py

```
#save this code as in h.py

#using Student class from student import Student

#create instance s = Student()

#store data into the instance

s.setid(100)

s.setname('Rakesh')

s.setaddress('HNO-22, Ameerpet, Hyderabad')

s.setmarks(970)

#retrieve data from instance and display

print('id=', s.getid())

print('name=', s.getname())

print('address=', s.getaddress())

print('marks=', s.getmarks())
```

Output:

```
C:\>python inh.py

id= 100

name= Rakesh

address= HNO-22, Ameerpet, Hyderabad

marks= 970
```

So far, so nice! If we compare the Teacher class and the Student classes, we can understand that 75% of the code is same in both the classes. That means most of the code being planned by the second programmer in his Student class is already available in the Teacher class. Then why doesn't he use it for his advantage? Our idea is this: instead of creating a new class altogether, he can reuse the code which is already available. This is shown in Program 13.

Program

Program 13: 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
```

The preceding code will be same as the first version of the Student class. Observe this code. In the first statement we are importing Teacher class from teacher module so that the Teacher class is now available to this program. Then we are creating Student class as: class Student(Teacher): This means the Student class is derived from Teacher class. Once we write like this, all the members of Teacher class are available to the Student class. Hence we can use them without rewriting them in the Student class. In addition, the following two methods are needed by the Student class but not available in the Teacher class:

```
def setmarks(self, marks):

def getmarks(self):
```

Hence, we wrote only these two methods in the Student class. Now, we can use the Student class as we did earlier. Creating the instance to the Student class and calling the methods as:

```
#create instance

s = Student()

#store data into the instance

s.setid(100)

s.setname('Rakesh')

s.setaddress('HNO-22, Ameerpet, Hyderabad')

s.setmarks(970)

 #retrieve data from instance and display

print('id=', s.getid())

print('name=', s.getname())

print('address=', s.getaddress())

print('marks=', s.getmarks())
```

In other words, we can say that we have created Student class from the Teacher class. This is called inheritance. The original class, i.e. Teacher class is called base class or super class and the newly created class, i.e. the Student class is called the sub class or derived class. So, how can we define inheritance? Deriving new classes from the existing classes such that the new classes inherit all the members of the existing classes, is called inheritance. The syntax for inheritance is:

class Subclass(Baseclass):

Then, what is the advantage of inheritance? Please look at Student class version 1 and Student class version 2. Clearly, second version is smaller and easier to develop. By using inheritance, a programmer can develop the classes very easily. Hence programmer's productivity is increased. Productivity is a term that refers to the code developed by the programmer in a given span of time. If the programmer used inheritance, he will be able to develop more code in less time. So, his productivity is increased. This will increase the overall productivity of the organization, which means more profits for the organization and better growth for the programmer.

In inheritance, we always create only the sub class object. Generally, we do not create super class object. The reason is clear. Since all the members of the super class are available to sub class, when we create an object, we can access the members of both the super and sub classes. But if we create an object to super class, we can access only the super class members and not the sub class members.

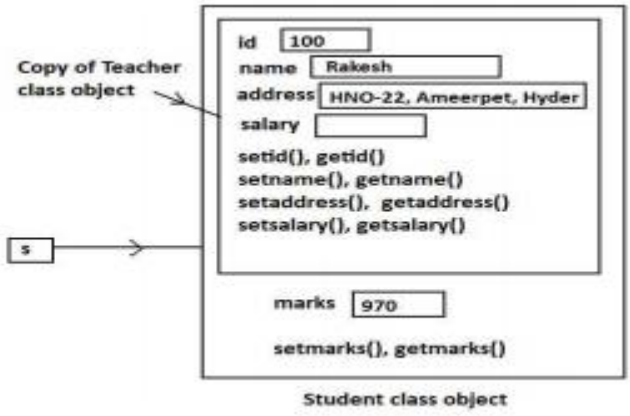


Figure 5: Student class object contains a copy of Teacher class object

Constructors in Inheritance

In the previous programs, we have inherited the Student class from the Teacher class. All the methods and the variables in those methods of the Teacher class (base class) are accessible to the Student class (sub class). Are the constructors of the base class accessible to the sub class or not – is the next question we will answer. In Program 6, we are taking a super class by the name 'Father' and derived a sub class 'Son' from it. The Father class has a constructor where a variable 'property' is declared and initialized with 800000.00. When Son is created from Father, this constructor is by default available to Son class. When we call the method of the super class using sub class object, it will display the value of the 'property' variable.

Program

Program 14: 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
```

The conclusion is this: like the variables and methods, the constructors in the super class are also available to the sub class object by default.

Overriding Super Class Constructors and Methods

When the programmer writes a constructor in the sub class, the super class constructor is not available to the sub class. In this case, only the sub class constructor is accessible from the sub class object. That means the sub class constructor is replacing the super class constructor. This is called constructor overriding. Similarly in the sub class, if we write a method with exactly same name as that of super class method, it will override the super class method. This is called method overriding. Consider Program 15.

Program

Program 15: A Python program to override super class constructor and method in sub class.

```
#overriding the base class constructor and method in sub class

class Father:

def __init__(self):

self.property = 800000.00
```

```
def display_property(self):

print('Father\'s property=', self.property)

class Son(Father):

def __init__(self):

self.property = 200000.00

def display_property(self):

print('Child\'s property=', self.property)

#create sub class instance and display father's property

s = Son()

s.display_property()
```

Output:

```
C:\>python inh.py

Child's property= 200000.00
```

In Program 15, in the sub class, we created a constructor and a method with exactly same names as those of super class. When we refer to them, only the sub class constructor and method are executed. The base class constructor and method are not available to the sub class object. That means they are overridden. Overriding should be done when the programmer wants to modify the existing behavior of a constructor or method in his sub class. In this case, how to call the super class constructor so that we can access the father's property from the Son class? For this purpose, we should call the constructor of the super class from the constructor of the sub class using the super() method.

The super() Method

super() is a built-in method which is useful to call the super class constructor or methods from the sub class. Any constructor written in the super class is not available to the sub class if the sub class has a constructor. Then how can we initialize the super class instance variables and use them in the sub class? This is done by calling the super class constructor using the super() method from inside the sub class constructor. super() is a built-in method in Python that contains the history of super class methods. Hence, we can use super() to refer to super class constructor and methods from a sub class. So super() can be used as:

```
super().__init__()    #call super class constructor

super().__init__(arguments)    # call super class constructor and pass

#arguments

super().method()   #call super class method
```

When there is a constructor with parameters in the super class, we have to create another constructor with parameters in the sub class and call the super class constructor using super() from the sub class

constructor. In the following example, we are calling the super class constructor and passing ‘property’ value to it from the sub class constructor.

```
#this is sub class constructor

def __init__(self, property1=0, property=0):

super().__init__(property)

#send property value to superclass

 #constructor

self.property1= property1

#store property1 value into subclass

#variable
```

As shown in the preceding code, the sub class constructor has 2 parameters. They are ‘property1’ and ‘property’. So, when we create an object (or instance) to sub class, we should pass two values, as:

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

Now, the first value 200000 is stored into ‘property1’ and the second value 800000.00 is stored into ‘property’. Afterwards, this ‘property’ value is sent to super class constructor in the first statement of the sub class constructor. This is shown in Program 8.

Program

Program 16: 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

s = Son(200000.00, 800000.00)

s.display_property()
```

Output:

```
C:\>python inh.py

Total property of child= 1000000.0
```

To understand the use of super() in a better way, let's write another Python program where we want to calculate areas of a square and a rectangle. Here, we are writing a Square class with one instance variable 'x' since to calculate the area of square, we need one value. Another class Rectangle is derived from Square. So, the value of 'x' is inherited by Rectangle class from Square class. To calculate area of rectangle we need two values. So, we take a constructor with two parameters 'x' and 'y' in the sub class. In this program, we are calling the super class constructor and passing 'x' value as:

```
super().__init__(x)
```

We are also calling super class area() method as:

```
super().area()
```

In this way, super() can be used to refer to the constructors and methods of super class.

Types of Inheritance

As we have seen so far, the main advantage of inheritance is code reusability. The members of the super class are reusable in the sub classes. Let's remember that all classes in Python are built from a single super class called 'object'. If a programmer creates his own classes, by default object class will become super class for them internally. This is the reason, sometimes while creating any new class, we mention the object class name in parentheses as:

```
class Myclass(object):
```

Here, we are indicating that object is the super class for Myclass. Of course, writing object class name is not mandatory and hence the preceding code is equivalent to writing:

```
class Myclass:
```

Now, coming to the types of inheritance, there are mainly 2 types of inheritance available. They are:

- Single inheritance
- 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, 'Bank' is a single base class from where we derive 'AndhraBank' and 'StateBank' as sub classes. This is called single inheritance. Consider Figure. It is convention that we should use the arrow head towards the base class (i.e. super class) in the inheritance diagrams.

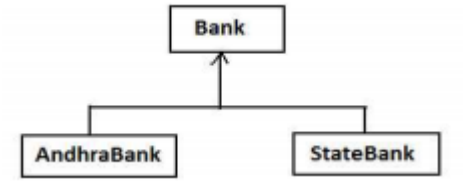


Figure 6: Single Inheritance Example

In Program 17, we are deriving two sub classes AndhraBank and StateBank from the single base class, i.e. Bank. All the members (i.e. variables and methods) of Bank class will be available to the sub classes. In the Bank class, we have some 'cash' variable and a method to display that, as:

```
class Bank(object):

cash = 100000000

@classmethod  def available_cash(cls):

print(cls.cash)
```

Here, the class variable 'cash' is declared in the class and initialized to 10 crores. The available_cash() is a class method that is accessing this variable as 'cls.cash'. When we derive AndhraBank class from Bank class as:

```
class AndhraBank(Bank):
```

The 'cash' variable and available_cash() methods are accessible to AndhraBank class and we can use them inside this sub class. Similarly, we can derive another sub class by the name StateBank from Bank class as:

```
class StateBank(Bank):

cash = 20000000

#class variable in the present sub class

@classmethod

def available_cash(cls):

print(cls.cash + Bank.cash)
```

Here, StateBank has its own class variable 'cash' that contains 2 crores. So, the total cash available to StateBank is 10 crores + 2 crores = 12 crores. Please observe the last line in the preceding code:

print(cls.cash + Bank.cash)

Here, ‘cls.cash’ represents the current class’s ‘cash’ variable and ‘Bank.cash’ represents the Bank base class ‘cash’ variable.

Program

Program 17: A Python program showing single inheritance in which two sub classes are derived from a single base class.

```
#single inheritance  class Bank(object):

cash = 100000000

@classmethod

def available_cash(cls):

print(cls.cash)

class AndhraBank(Bank):

pass

class StateBank(Bank):

cash = 20000000

@classmethod

def available_cash(cls):

print(cls.cash + Bank.cash)

a = AndhraBank()

a.available_cash()

s = StateBank()

s.available_cash()
```

Output:

```
C:\>python inh.py

100000000

120000000
```

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. All the members of the super classes are by default available to sub classes and the sub

classes in turn can have their own members. The syntax for multiple inheritance is shown in the following statement:

class Subclass(Baseclass1, Baseclass2, …):

The best example for multiple inheritance is that parents producing the children and the children inheriting the qualities of the parents. Consider Figure. Suppose, Father and Mother are two base classes and Child is the sub class derived from these two base classes. Now, whatever the members are found in the base classes are available to the sub class. For example, the Father class has a method that displays his height as 6.0 foot and the Mother class has a method that displays her color as brown. To make the Child class acquire both these qualities, we have to make it a sub class for both the Father and Mother class. This is shown in Program 18.

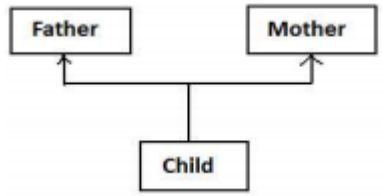


Figure 7: Multiple inheritance example

Program

Program 18: A Python program to implement multiple inheritance using two base classes. #multiple inheritance

```
class Father:

def height(self):

print('Height is 6.0 foot')

class Mother:

def color(self):

print('Color is brown')

class Child(Father, Mother):

pass  c = Child()

print('Child\'s inherited qualities:')

c.height()

c.color()
```

Output:

```
C:\>python inh.py

Child's inherited qualities:
```

Height is 6.0 foot

Color is brown

Problems in Multiple Inheritance

If the sub class has a constructor, it overrides the super class constructor and hence the super class constructor is not available to the sub class. But writing constructor is very common to initialize the instance variables. In multiple inheritance, let's assume that a sub class 'C' is derived from two super classes 'A' and 'B' having their own constructors. Even the sub class 'C' also has its constructor. To derive C from A and B, we write:

class C(A, B):

Also, in class C's constructor, we call the super class super class constructor using super().__init__(). Now, if we create an object of class C, first the class C constructor is called. Then super().__init__() will call the class A's constructor. Consider Program 19.

Program

Program 19: A Python program to prove that only one class constructor is available to sub class in multiple inheritance.

```
#when super classes have constructors

class A(object):

def __init__(self):

self.a = 'a'

print(self.a)

class B(object):

def __init__(self):

self.b = 'b'

print(self.b)

class C(A, B):

def __init__(self):

self.c = 'c'

print(self.c)

super().__init__()

#access the super class instance vars from C

o = C()
```

#o is object of class C

Output:

C:\>python inh.py

c

a

The output of the program indicates that when class C object is created the C's constructor is called. In class C, we used the statement: super().__init__() that calls the class A's constructor only. Hence, we can access only class A's instance variables and not that of class B. In Program 20, we created sub class C, as:

class C(A, B):

This means class C is derived from A and B as shown in the Figure. Since all classes are sub classes of object class internally, we can take classes A and B are sub classes of object class.

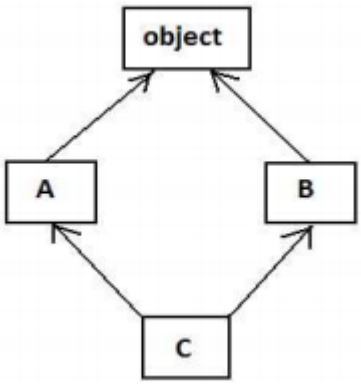


Figure 8: The effect of class C(A, B)

In the above figure, class A is at the left side and class B is at the right side for the class C. The searching of any attribute or method will start from the sub class C. Hence, C's constructor is accessed first. As a result, it will display 'c'. Observe the code in C's constructor:

```
def __init__(self):

self.c = 'c'

print(self.c)

super().__init__()
```

The last line in the preceding code, i.e. super().__init__() will call the constructor of the class which is at the left side. So, class A's constructor is executed and 'a' is displayed. If class A does not have a constructor, then it will call the constructor of the right hand side class, i.e. B. But since class A has a constructor, the search stopped here. If the class C is derived as: class C(B, A):

Then the output will be:

c

b

The problem we should understand is that the class C is unable to access constructors of both the super classes. It means C cannot access all the instance variables of both of its super classes. If C wants to access instance variables of both of its super classes, then the solution is to use super().__init__() in every class. This is shown in Program 20.

Program

Program 20: A Python program to access all the instance variables of both the base classes in multiple inheritance.

```
#when super classes have constructors - v2.0

class A(object):

def __init__(self):

self.a = 'a'

print(self.a)

super().__init__()

class B(object):

def __init__(self):

self.b = 'b'

print(self.b)

super().__init__()

class C(A, B):

def __init__(self):

self.c = 'c'

print(self.c)

super().__init__()

#access the super class instance vars from C

o = C()

#o is object of class C
```

Output:

```
C:\>python inh.py

c
```

a

b

We will apply the diagram given in Figure to Program 20. The search will start from C. As the object of C is created, the constructor of C is called and ‘c’ is displayed. Then super().__init__() will call the constructor of left side class, i.e. of A. So, the constructor of A is called and ‘a’ is displayed. But inside the constructor of A, we again called its super class constructor using super().__init__(). Since ‘object’ is the super class for A, an attempt to execute object class constructor will be done. But object class does not have any constructor. So, the search will continue down to right hand side class of object class. That is class B. Hence B’s constructor is executed and ‘b’ is displayed. After that the statement super().__init__() will attempt to execute constructor of B’s super class. That is nothing but ‘object’ class. Since object class is already visited, the search stops here. As a result the output will be ‘c’, ’a’, ’b’. Searching in this manner for constructors or methods is called Method Resolution Order(MRO).

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:

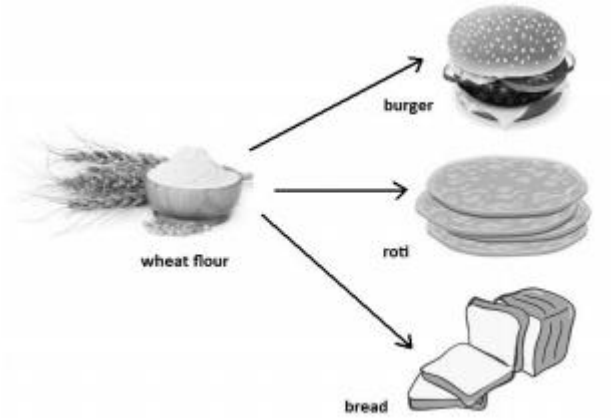


Figure 9: Polymorphism where wheat flour takes different edible forms

In programming, a variable, object or a method will also exhibit the same nature as that of the wheat flour. A variable may store different types of data, an object may exhibit different behaviors in different contexts or a method may perform various tasks in Python. This type of behavior is called polymorphism. So, how can we define polymorphism? If a variable, object or method exhibits different behavior in different contexts, it is called polymorphism. Python has built-in polymorphism. The following topics are examples for polymorphism in Python:

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

- Method overriding

Duck Typing Philosophy of Python

We know that in Python, the data type of the variables is not explicitly declared. This does not mean that Python variables do not have a type. Every variable or object in Python has a type and the type is implicitly assigned depending on the purpose for which the variable is used. In the following examples, 'x' is a variable. If we store integer into that variable, its type is taken as 'int' and if we store a string into that variable, its type is taken as 'str'. To check the type of a variable or object, we can use type() function.

```
x = 5    #store integer into x

print(type(x)) #display type of x

<class 'int'>  x = 'Hello'      #store string into x

print(type(x))     #display type of x

<class 'str'>
```

Python variables are names or tags that point to memory locations where data is stored. They are not worried about which data we are going to store. So, if 'x' is a variable, we can make it refer to an integer or a string as shown in the previous examples. We can conclude two points from this discussion:

1. Python's type system is 'strong' because every variable or object has a type that we can check with the type() function.

2. Python's type system is 'dynamic' since the type of a variable is not explicitly declared, but it changes with the content being stored.

Similarly, if we want to call a method on an object, we do not need to check the type of the object and we do not need to check whether that method really belongs to that object or not. For example, take a method call_talk() that accepts an object (or instance).

```
def call_talk(obj):

obj.talk()
```

The call_talk() method is receiving an object 'obj' from outside and using this object, it is invoking (or calling) talk() method. It is not required to mention the type of the object 'obj' or to check whether the talk() method belongs to that object or not. If the object passed to this method belongs to Duck class, then talk() method of Duck class is called. If the object belongs to Human class, then the talk() method of Human class is called.

Operator Overloading

We know that an operator is a symbol that performs some action. For example, '+' is an operator that performs addition operation when used on numbers. When an operator can perform different actions, it is said to exhibit polymorphism.

Program

Program 21: A Python program to use addition operator to act on different types of objects.

```
#overloading the + operator

#using + on integers to add them

print(10+15)

#using + on strings to concatenate them  s1 = "Red"

s2 = "Fort"

print(s1+s2)  #using + on lists to make a single list

a = [10, 20, 30]

b = [5, 15, -10]

print(a+b)
```

Output:

```
C:\>python op.py

25

RedFort

[10, 20, 30, 5, 15, -10]
```

In Program 18, the '+' operator is first adding two integer numbers. Then the same operator is concatenating two strings. Finally, the same operator is combining two lists of elements and making a single list. In this way, if any operator performs additional actions other than what it is meant for, it is called operator overloading. Operator overloading is an example for polymorphism.

Method Overloading

If a method is written such that it can perform more than one task, it is called method overloading. We see method overloading in the languages like Java. For example, we call a method as:

```
sum(10, 15)

sum(10, 15, 20)
```

In the first call, we are passing two arguments and in the second call, we are passing three arguments. It means, the sum() method is performing two distinct operations: finding sum of two numbers or sum of three numbers. This is called method overloading.

Method overloading is not available in Python. Writing more than one method with the same name is not possible in Python. So, we can achieve method overloading by writing same method with several parameters. The method performs the operation depending on the number of arguments passed in the

method call. For example, we can write a sum() method with default value 'None' for the arguments as:

```
def sum(self, a=None, b=None, c=None):

if a!=None and b!=None and c!=None:

print('Sum of three=', a+b+c)

elif a!=None and b!=None:

print('Sum of two=', a+b)
```

Here, sum() has three arguments 'a','b','c' whose values by default are 'None'. This 'None' indicates nothing or no value and similar to 'null' in languages like Java. While calling this method, if the user enters three values, then the arguments: 'a',' b' and 'c' will not be 'None'. They get the values entered by the user. If the user enters only two values, then the first two arguments 'a' and 'b' only will take those values and the third argument 'c' will become 'None'.

Method Overriding

We already discussed constructor overriding and method overriding under inheritance section. When there is a method in the super class, writing the same method in the sub class so that it replaces the super class method is called 'method overriding'. The programmer overrides the super class methods when he does not want to use them in sub class. Instead, he wants a new functionality to the same method in the sub class.

In inheritance, if we create super class object (or instance), we can access all the members of the super class but not the members of the sub class. But if we create sub class object, then both the super class and sub class members are available since the sub class object contains a copy of the super class. Hence, in inheritance we always create sub class object. In Program 24, we created Square class with the area() method that calculates the area of the square. Circle class is a sub class to Square class that contains the area() method rewritten with code to calculate area of circle. When we create sub class object as:

```
c = Circle() #create sub class object

c.area(15)    #call area() method
```

Then the area() method of Circle class is called but not the area() method of Square class. The reason is that the area() method of sub class has overridden the area() method of the super class.

Program

Program 22: A Python program to override the super class method in sub class.

```
#method overriding  import math  class Square:

def area(self, x):

print('Square area= %.4f}'% x*x)

class Circle(Square):
```

```
def area(self, x):

print('Circle area= %.4f'% (math.pi*x*x))  #call area() using sub class object

c = Circle()  c.area(15)
```

Output:

```
C:\>python over.py

Circle area= 706.8583
```

Abstract Classes and Interfaces

Abstract Method and Abstract Class

An abstract method is a method whose action is redefined in the sub classes as per the requirement of the objects. Generally abstract methods are written without body since their body will be defined in the sub classes anyhow. But it is possible to write an abstract method with body also. To mark a method as abstract, we should use the decorator @abstractmethod. On the other hand, a concrete method is a method with body. An abstract class is a class that generally contains some abstract methods. Since, abstract class contains abstract methods whose implementation (or body) is later defined in the sub classes, it is not possible to estimate the total memory required to create the object for the abstract class. So, PVM cannot create objects to an abstract class. Once an abstract class is written, we should create sub classes and all the abstract methods should be implemented (body should be written) in the sub classes. Then, it is possible to create objects to the sub classes. In Program 23, we create Myclass as an abstract super class with an abstract method calculate(). This method does not have any body within it. The way to create an abstract class is to derive it from a meta class ABC that belongs to abc (abstract base class) module as:

```
class Abstractclass(ABC):
```

Since all abstract classes should be derived from the meta class ABC which belongs to abc (abstract base class) module, we should import this module into our program. A meta class is a class that defines the behavior of other classes. The meta class ABC defines that the class which is derived from it becomes an abstract class. To import abc module's ABC class and abstractmethod decorator we can write as follows: from abc import ABC, abstractmethod or from abc import * Now, our abstract class 'Myclass' should be derived from the ABC class as:

```
classs Myclass(ABC):

@abstractmethod

def calculate(self, x):

pass

#empty body, no code
```

Observe the preceding code. Our Myclass is abstract class since it is derived from ABC meta class. This class has an abstract method calculate() that does not contain any code. We used @abstractmethod decorator to specify that this is an abstract method. We have to write sub classes where this abstract method is written with its body (or implementation). In Program 23, we are going

to write three sub classes: Sub1, Sub2 and Sub3 where this abstract method is implemented as per the requirement of the objects. Since, the same abstract method is implemented differently for different objects, they can perform different tasks.

Program

Program 23: 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):

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

We learned that an abstract class is a class which contains some abstract methods as well as concrete methods also. Imagine there is a class that contains only abstract methods and there are no concrete methods. It becomes an interface. This means an interface is an abstract class but it contains only abstract methods. None of the methods in the interface will have body. Only method headers will be written in the interface. So, 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. In this case, we can create sub classes where we can implement all the methods of the interface. Since the sub classes will have all the methods with body, it is possible to create objects to the sub classes. The flexibility lies in the fact that every sub class can provide its own implementation for the abstract methods of the interface. Since, none of the methods have body in the interface, we may tend to think that writing an interface is mere waste. This is not correct. In fact, an interface is more useful when compared to the class owing to its flexibility of providing necessary implementation needed by the objects. Let's elucidate this point further with an example. We have some rupees in our hands. We can spend in rupees only by going to a shop where billing is done in rupees. Suppose we have gone to a shop where only dollars are accepted, we cannot use our rupees there. This money is like a 'class'. A class satisfies the only requirement intended for it. It is not useful to handle a different situation. Suppose we have an international credit card. Now, we can pay by using our credit card in rupees in a shop. If we go to another shop where they expect us to pay in dollars, we can pay in dollars. The same credit card can be used to pay in pounds also. Here, the credit card is like an interface which performs several tasks. In fact, the credit card is a plastic card and does not hold any money physically. It contains just our name, our bank name and perhaps some number. But how the shop keepers are able to draw the money from the credit card? Behind the credit card, we got our bank account which holds the money from where it is transferred to the shop keepers. This bank account can be taken as a sub class which actually performs the task. See Figure:

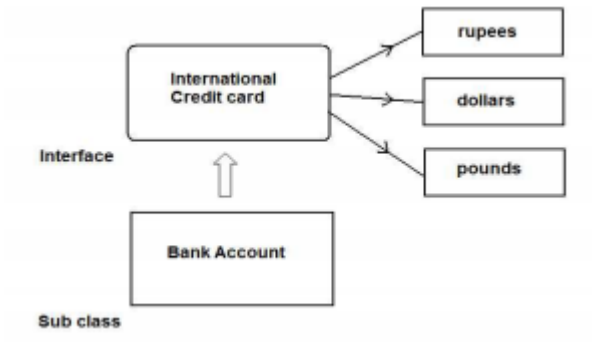


Figure 10: Interface and Sub Classes

Let's see how the interface concept is advantageous in software development. A programmer is asked to write a Python program to connect to a database and retrieve the data, process the data and display the results in the form of some reports. For this purpose, the programmer has written a class to connect to Oracle database, something like this:

#this class works with Oracle database only class Oracle:

def connect(self):

print('Connecting to Oracle database...')

def disconnect(self): print('Disconnected from Oracle.')

This class has a limitation. It can connect only to Oracle database. If a client (user) using any other database (for example, Sybase database) uses this code to connect to his database, this code will not work. So, the programmer is asked to design his code in such a way that it is used to connect to any database in the world. How is it possible?

One way is to write several classes, each to connect to a particular database. Thus, considering all the databases available in the world, the programmer has to write a lot of classes. This takes a lot of time and effort. Even though the programmer spends a lot of time and writes all the classes, by the time the software is released into the market, all the versions of the databases will change and the programmer is supposed to rewrite the classes again to suit the latest versions of databases. This is very cumbersome. Interface helps to solve this problem. The programmer writes an interface 'Myclass' with abstract methods as shown here:

#an interface to connect to any database

class Myclass(ABC):

@abstractmethod

def connect(self):

pass

@abstractmethod

def disconnect(self):

pass

We cannot create objects to the interface. So, we need sub classes where all these methods of the interface are implemented to connect to various databases. This task is left for other companies that are called third party vendors. The third party vendors will provide sub classes to Myclass interface. For example, Oracle Corp people may provide a sub class where the code related to connecting to the Oracle database and disconnecting from the database will be provided as:

```
#this is a sub class to connect to Oracle

class Oracle(Myclass):

def connect(self):

print('Connecting to Oracle database...')

def disconnect(self):

print('Disconnected from Oracle.')
```

Note that 'Oracle' is a sub class of Myclass interface. Similarly, the Sybase company people may provide another implementation class 'Sybase', where code related to connecting to Sybase database and disconnecting from Sybase will be provided as:

```
#this is another sub class to connect to Sybase

class Sybase(Myclass):

def connect(self):

print('Connecting to Sybase database...')

def disconnect(self):

print('Disconnected from Sybase.')
```

Now, it is possible to create objects to the sub classes and call the connect() method and disconnect() methods from a main program. This main program is also written by the same programmer who develops the interface. Let's understand that the programmer who develops the interface does not know the names of the sub classes as they will be developed in future by the other companies. In the main program, the programmer is supposed to create objects to the sub classes without knowing their names. This is done using globals() function. First, the programmer should accept the database name from the user. It may be 'Oracle' or 'Sybase'. This name should be taken in a string, say 'str'. The next step is to convert this string into a class name using the built-in function globals(). The globals() function returns a dictionary containing current global names and globals()[str] returns the name of the class that is in 'str'. Hence, we can get the class name as:

```
classname = globals()[str]
```

Now, create an object to this class and call the methods as:

```
x = classname() #x is object of the class

x.connect()

x.disconnect()
```

The connect() method establishes connection with the particular database and disconnect() method disconnects from the database. The complete program is shown in Program 24.

Program

Program 24: A Python program to develop an interface that connects to any database. #abstract class works like an interface from abc import *

```
class Myclass(ABC):

@abstractmethod

def connect(self):

pass

@abstractmethod

def disconnect(self):

pass

#this is a sub class

class Oracle(Myclass):

def connect(self):

print('Connecting to Oracle database...')

def disconnect(self):

print('Disconnected from Oracle.')

#this is another sub class

class Sybase(Myclass):

def connect(self):

print('Connecting to Sybase database...')

def disconnect(self):  print('Disconnected from Sybase.')

class Database:

#accept database name as a string

str = input('Enter database name:')

#convert the string into classname

classname = globals()[str]

#create an object to that class  x = classname()
```

```
#call the connect() and disconnect() methods

x.connect()

x.disconnect()
```

Output:

```
C:\>python inter.py

Enter database name: Oracle

Connecting to Oracle database...

Disconnected from Oracle.
```

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. Generally, abstract class is written when there are some common features shared by all the objects as they are. 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
```

Let's take Retailer1, a class which represents a retail shop. Retailer1 wants text books of X class and some pens. Similarly, Retailer2 also wants text books of X class and some papers. In this case, we can understand that the text_books() is the common feature shared by both the retailers. But the stationery asked by the retailers is different. This means, the stationery has different implementations for different retailers but there is a common feature, i.e., the text books. So in this case, the programmer designs the WholeSaler class as an abstract class. Retailer1 and Retailer2 are sub classes.

On the other hand, the programmer uses an interface if all the features need to be implemented differently for different objects. Suppose, Retailer1 asks for VII class text books and Retailer2 asks for X class text books, then even the text_books() method of WholeSaler class needs different implementations depending on the retailer. It means, the text_books() method and also stationery() methods should be implemented differently depending on the retailer. So, in this case, the programmer designs the WholeSaler as an interface and Retailer1 and Retailer2 become sub classes. There is a responsibility for the programmer to provide the sub classes whenever he writes an abstract

class. This means the same development team should provide the sub classes for the abstract class. But if an interface is written, any third party vendor will take the responsibility of providing sub classes. This means, the programmer prefers to write an interface when he wants to leave the implementation part to the third party vendors. In case of an interface, every time a method is called, PVM should search for the method in the implementation classes which are installed elsewhere in the system and then execute the method. This takes more time. But when an abstract class is written, since the common methods are defined within the abstract class and the sub classes are generally in the same place along with the software, PVM will not have that much overhead to execute a method. Hence, interfaces are slow when compared to abstract classes.

Module-III

Strings and Functions

Creating strings

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. Sometimes, we can use triple single quotes or triple double quotes to represent strings. These quotation marks are useful when we want to represent a string that occupies several lines as:

str = '''Welcome to Core Python, a book on Python language that discusses all important concepts of Python in a lucid and comprehensive manner.'''

In the preceding statement, the string 'str' is created using triple single quotes. Alternately, the above string can be created using triple double quotes as:

str = """Welcome to Core Python, a book on Python language that discusses all important concepts of Python in a lucid and comprehensive manner."""

Thus, triple single quotes or triple double quotes are useful to create strings which span into several lines. It is possible to display quotation marks to mark a sub string in a string. In that case, we should use one type of quotes for outer string and another type of quotes for inner string as:

s1 = 'Welcome to "Core Python" learning'

print(s1)

The preceding lines of code will display the following output:

Welcome to "Core Python" learning

Here, the string 's1' contains two strings. The outer string is enclosed in single quotes and the inner string, i.e. "Core Python" is enclosed in double quotes. Alternately, we can use double quotes for outer string and single quotes for inner string as:

s1 = "Welcome to 'Core Python' learning"

print(s1)

The preceding lines of code will display the following output:

Welcome to 'Core Python' learning

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. For example,

```
s1 = "Welcome to\tCore Python\nlearning"

print(s1)
```

The preceding lines of code will display the following output:

```
Welcome to Core Python learning
```

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\

To nullify the effect of escape characters, we can create the string as a 'raw' string by adding 'r' before the string as:

```
s1 = r"Welcome to\tCore Python\nlearning"

print(s1)
```

The preceding lines of code will display the following output:

```
Welcome to\tCore Python\nlearning
```

This is not showing the effect of \t or \n. It means we could not see the horizontal tab space or new line. Raw strings take escape characters, like \t, \n, etc., as ordinary characters in a string and hence display them as they are.

Basic operations on strings

1.Length of a 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
```

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.

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
 Core Pyth
```

When 'stepsize' is 2, then it will access every other character from 1st character onwards. Hence it retrieves the 0th, 2nd, 4th, 6th characters and so on.

```
 str[0:9:2]
 Cr yh
```

Some other examples are given below to have a better understanding on slicing. 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.

```
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.

```
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. While comparing the strings, Python interpreter compares them by taking them in English dictionary order. The string which comes first in the dictionary order will have a low value than the string which comes next. It means, 'A' is less than 'B' which is less than 'C' and so on. In the above example, the string 's1' comes before the string 's2' and hence s1 is less than s2. So, if we write:

```
if s1<s2:

print('s1 less than s2')

else:

print('s1 greater than or equal to s2')
```

Then, the preceding statements will display 's1 less than s2'.

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’. In this way, spaces may lead to wrong results. 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.

Consider the following code snippet:

```
name = ' Mukesh Deshmukh ' #observe spaces before and after the name

print(name.rstrip()) #remove spaces at right
```

The output of the preceding statement is as follows:

```
Mukesh Deshmukh
```

Now, if you write:

```
print(name.lstrip()) #remove spaces at left
```

The output of the preceding statement is as follows:

```
Mukesh Deshmukh
```

Now, if you write:

```
print(name.strip()) #remove spaces from both sides
```

The output of the preceding statement is as follows:

```
Mukesh Deshmukh
```

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)

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(‘,’) Observe the comma inside the parentheses. It is called separator that represents where to separate or cut the string. Similarly, the separator will be a space if we want to cut the string at spaces. 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) T
```

he output of the preceding statements is as follows:

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

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
```

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.
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

To understand how to use these methods on strings, let's take an example. In this example, we take a string as:

```
str = 'Delhi999'
```

Now, we want to check if this string 'str' contains only alphabets, i.e. A to Z, a to z and not other characters like digits or spaces. We will use isalpha() method on the string as:

```
str.isalpha()
```

False

Since the string 'Delhi999' contains digits, the isalpha() method returned False.

Another example:

```
str = 'Delhi'

str.isalpha() True
```

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. Similar to these functions, a programmer can also create his own functions which are called 'user-defined' functions.

The following are the advantages of functions:

- ✓ Functions are important in programming because they are used to process data, make calculations or perform any task which is required in the software development.
- ✓ Once a function is written, it can be reused as and when required. So functions are also called reusable code. Because of this reusability, the programmer can avoid code redundancy. It means it is possible to avoid writing the same code again and again.
- ✓ Functions provide modularity for programming. A module represents a part of the program. Usually, a programmer divides the main task into smaller sub tasks called modules. To represent each module, the programmer will develop a separate function. Then these functions are called from a main program to accomplish the complete task. Modular programming makes programming easy.

- ✓ Code maintenance will become easy because of functions. When a new feature has to be added to the existing software, a new function can be written and integrated into the software. Similarly, when a particular feature is no more needed by the user, the corresponding function can be deleted or put into comments.
- ✓ 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 a Function

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. These values are processed in the function body and result is obtained. The values passed to a function are called ‘arguments’. So, 10 and 15 are arguments.

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.

Example:

A Python program to find the sum of two numbers and return the result from the function.

#a function to add two numbers

def sum(a, b):

""" This function finds sum of two numbers """

c = a+b

return c #return result

#call the function

x = sum(10, 15)

```
print('The sum is:', x)

y = sum(1.5, 10.75)

print('The sum is:', y)
```

Output: C:\>python fun.py

```
The sum is: 25

The sum is: 12.25
```

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
```

Here, three values which are in ‘a’, ‘b’, and ‘c’ are returned. These values are returned by the function as a tuple. Please remember a tuple is like a list that contains a group of elements. To grab these values, we can use three variables at the time of calling the function as:

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

Here, the variables ‘x’, ‘y’ and ‘z’ are receiving the three values returned by the function. To understand this practically, we can create a function by the name sum_sub() that takes 2 values and calculates the results of addition and subtraction. These results are stored in the variables ‘c’ and ‘d’ and returned as a tuple by the function.

```
def sum_sub(a, b):

c = a + b

d = a – b

return c, d
```

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. Also, it is possible to return a function from another function. This is similar to returning an object (or value) from a function. The following possibilities are noteworthy:

- ✓ It is possible to assign a function to a variable.
- ✓ It is possible to define one function inside another function.
- ✓ It is possible to pass a function as parameter to another function.
- ✓ It is possible that a function can return another function.

To understand these points, we will take a few simple programs. In Program 9, we have taken a function by the name display() that returns a string. This function is called and the returned string is assigned to a variable ‘x’.

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

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

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?

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
```

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. For example, take a function definition with two arguments as:

def attach(s1, s2)

This function expects two strings that too in that order only. Let's assume that this function attaches the two strings as s1+s2. So, while calling this function, we are supposed to pass only two strings as:

attach('New', 'York')

The preceding statement displays the following output:

NewYork

Suppose, we passed 'York' first and then 'New', then the result will be: 'YorkNew'. Also, if we try to pass more than or less than 2 strings, there will be an error. For example, if we call the function by passing 3 strings as:

attach('New', 'York', City')

Then there will be an error displayed.

Example:

A Python program to understand the positional arguments of a function.

```
#positional arguments demo
def attach(s1, s2):
""" to join s1 and s2 and display total string """
s3 = s1+s2
print('Total string: '+s3)
#call attach() and pass 2 strings
attach('New', 'York') #positional arguments
```

Output: C:\>python fun.py

Total string: NewYork

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):

At the time of calling this function, we have to pass two values and we can mention which value is for what. For example,

grocery(item='Sugar', price=50.75)

Here, we are mentioning a keyword ‘item’ and its value and then another keyword ‘price’ and its value. Please observe these keywords are nothing but the parameter names which receive these values. We can change the order of the arguments as:

grocery(price=88.00, item='Oil')

In this way, even though we change the order of the arguments, there will not be any problem as the parameter names will guide where to store that value.

A Python program to understand the keyword arguments of a function.

#key word arguments demo def grocery(item, price):

""" to display the given arguments """

print('Item = %s'% item)

print('Price = %.2f'% price)

#call grocery() and pass 2 arguments

grocery(item='Sugar', price=50.75) #keyword arguments

grocery(price=88.00, item='Oil') #keyword arguments

Output: C:\>python fun.py

Item = Sugar

Price = 50.75

Item = Oil

Price = 88.00

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):

Here, the first argument is ‘item’ whose default value is not mentioned. But the second argument is ‘price’ and its default value is mentioned to be 40.00. At the time of calling this function, if we do not pass ‘price’ value, then the default value of 40.00 is taken. If we mention the ‘price’ value, then that mentioned value is utilized. So, a default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

A Python program to understand the use of default arguments in a function.

#default arguments demo

def grocery(item, price=40.00):

""" to display the given arguments """

```
print('Item = %s'% item)

print('Price = %.2f'% price)

#call grocery() and pass arguments

grocery(item='Sugar', price=50.75)

#pass 2 arguments grocery(item='Sugar')

#default value for price is used.
```

Output: C:\>python fun.py

```
Item = Sugar

Price = 50.75

Item = Sugar

Price = 40.00
```

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. For example, if the programmer is writing a function to add two numbers, he can write:

```
add(a, b)
```

But, the user who is using this function may want to use this function to find sum of three numbers. In that case, there is a chance that the user may provide 3 arguments to this function as: add(10, 15, 20)

Then the add() function will fail and error will be displayed. If the programmer wants to develop a function that can accept 'n' arguments, that is also possible in Python. For this purpose, a variable length argument is used in the function definition. A variable length argument is an argument that can accept any number of values. The variable length argument is written with a ' * ' symbol before it in the function definition as:

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

Here, 'farg' is the formal argument and '*args' represents variable length argument. We can pass 1 or more values to this '*args' and it will store them all in a tuple. A tuple is like a list where a group of elements can be stored. In Program 19, we are showing how to use variable length argument.

A Python program to show variable length argument and its use.

```
#variable length argument demo

def add(farg, *args):

 #*args can take 0 or more values
```

```
""" to add given numbers """
print('Formal argument=', farg)
sum=0
for i in args:
 sum+=i
print('Sum of all numbers= ',(farg+sum))
#call add() and pass arguments
add(5, 10)
add(5, 10, 20, 30)
```

Output: C:\>python fun.py

Formal argument= 5

Sum of all numbers= 15

Formal argument= 5

Sum of all numbers= 65

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)

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)))
```

Output: C:\>python fun.py

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

A Python program to solve Towers of Hanoi problem.

```
#recursive function to solve Towers of Hanoi
def towers(n, a, c, b):
`if n==1:
#if only 1 disk, then move it from A to C
print('Move disk %i from pole %s to pole %s' %(n, a, c))
```

```
else: #if more than 1 disk
#move first n-1 disks from A to B using C as intermediate pole
 towers(n-1, a, b, c)
 #move remaining 1 disk from A to C
print('Move disk %i from pole %s to pole %s'%(n, a, c))
#move n-1 disks from B to C using A as intermediate pole
 towers(n-1, b, c, a)
 #call the function
n = int(input('Enter number of disks:'))
#we should change n disks from A to C using B as intermediate pole
towers(n, 'A', 'C', 'B')
```

Output: C:\>python fun.py

```
 Enter number of disks: 3
Move disk 1 from pole A to pole C
Move disk 2 from pole A to pole B
Move disk 1 from pole C to pole B
Move disk 3 from pole A to pole C
 Move disk 1 from pole B to pole A
 Move disk 2 from pole B to pole C
Move disk 1 from pole A to pole C
```

Module –IV

Exception Handling

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

        ^

Syntax Error: invalid syntax
```

The most common reason of an error in a Python program is when a certain statement is not in accordance with the prescribed usage.

Such an error is called a syntax error. The Python interpreter immediately reports it, usually along with the reason.

Python (interpreter) raises exceptions when it encounters errors. For example: divided by zero.

There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

Syntax Errors:

When writing a program, we, more often than not, will encounter errors.

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

```
1. >>> if a < 3
2.   File "<interactive input>", line 1
3.     if a < 3
4.           ^
5. SyntaxError: invalid syntax
```

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

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
1. >>> while True print('Hello world')
```

2. File "<stdin>", line 1

3. while True print('Hello world')

    a. ^

4. 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 caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the function print(), since a colon (':') is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

Exceptions

Errors can also occur at runtime and these are called exceptions. They occur, for example, when a file we try to open does not exist (FileNotFoundError), dividing a number by zero (ZeroDivisionError), module we try to import is not found (ImportError) etc.

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: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

>>> 10 * (1/0)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ZeroDivisionError: division by zero

>>> 4 + spam*3

Traceback (most recent call last)

  File "<stdin>", line 1, in <module>

NameError: name 'spam' is not defined

>>> '2' + 2

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

TypeError: Can't convert 'int' object to str implicitly

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are ZeroDivisionError, NameError and TypeError. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be

true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The preceding part of the error message shows the context where the exception happened, in the form of a stack trace back. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

Handling Exceptions:

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using Control-Cor whatever the operating system supports); note that a user-generated interruption is signaled by raising the KeyboardInterrupt exception.

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

Thr **try** statement works as follows.

- First, the *try clause* (the statement(s) between the **try** and **except** keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the **try** statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the **except** keyword, the except clause is executed, and then execution continues after the **try** statement.
- 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 with a message as shown above.

A **try** statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

A class in an **except** clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

```
class B(Exception):
    pass
class C(B):
    pass
class D(C):
    pass
for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Note that if the except clauses were reversed (with except B first), it would have printed B, B, B — the first matching except clause is triggered.

The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

```
import sys
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
```

```
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

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 example:

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

The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the try … except statement.

When an exception occurs, it may have an associated value, also known as the exception's *argument*. The presence and type of the argument depend on the exception type.

The except clause may specify a variable after the exception name. The variable is bound to an exception instance with the arguments stored in instance.args. For convenience, the exception instance defines **str()** the arguments can be printed directly without having to reference .args. One may also instantiate an exception first before raising it and add any attributes to it as desired.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     print(inst.args)     # arguments stored in .args
...     print(inst)          # __str__ allows args to be printed directly,
...                          # but may be overridden in exception subclasses
```

```
...     x, y = inst.args      # unpack args
...     print('x =', x)
...     print('y =', y)
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

If an exception has arguments, they are printed as the last part (‘detail’) of the message for unhandled exceptions.

Exception handlers don’t just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause. For example:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

Raising Exceptions

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

For example:

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

The sole argument to **raise** indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from **Exception**). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

raise ValueError # shorthand for 'raise ValueError()'

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the **raise** statement allows you to re-raise the exception:

>>> 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 <module>

NameError: HiThere

Here is a list all the standard Exceptions available in Python −

Sr.No.	Exception Name & Description
1	**Exception** Base class for all exceptions
2	**StopIteration** Raised when the next() method of an iterator does not point to any object.
3	**SystemExit** Raised by the sys.exit() function.
4	**StandardError** Base class for all built-in exceptions except StopIteration and SystemExit.
5	**ArithmeticError** Base class for all errors that occur for numeric calculation.
6	**OverflowError** Raised when a calculation exceeds maximum limit for a numeric type.
7	**FloatingPointError** Raised when a floating point calculation fails.

8	**ZeroDivisionError** Raised when division or modulo by zero takes place for all numeric types.
9	**AssertionError** Raised in case of failure of the Assert statement.
10	**AttributeError** Raised in case of failure of attribute reference or assignment.
11	**EOFError** Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
12	**ImportError** Raised when an import statement fails.
13	**KeyboardInterrupt** Raised when the user interrupts program execution, usually by pressing Ctrl+c.
14	**LookupError** Base class for all lookup errors.
15	**IndexError** Raised when an index is not found in a sequence.
16	**KeyError** Raised when the specified key is not found in the dictionary.
17	**NameError** Raised when an identifier is not found in the local or global namespace.
18	**UnboundLocalError** Raised when trying to access a local variable in a function or method but no value has been assigned to it.
19	**EnvironmentError** Base class for all exceptions that occur outside the Python environment.
20	**IOError** Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
21	**OSError** Raised for operating system-related errors.
22	**SyntaxError** Raised when there is an error in Python syntax.
23	**IndentationError** Raised when indentation is not specified properly.
24	**SystemError** Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
25	**SystemExit** Raised when Python interpreter is quit by using the sys.exit() function. If not

	handled in the code, causes the interpreter to exit.
26	**TypeError** Raised when an operation or function is attempted that is invalid for the specified data type.
27	**ValueError** Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
28	**RuntimeError** Raised when a generated error does not fall into any category.
29	**NotImplementedError** Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

User-defined Exceptions

In Python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from Exception class. Most of the built-in exceptions are also derived from this class.

```
>>> class CustomError(Exception):
...     pass
...
>>> raise CustomError
Traceback (most recent call last):
...
__main__.CustomError
>>> raise CustomError("An error occurred")
Traceback (most recent call last):
...
__main__.CustomError: An error occurred
```

Here, we have created a user-defined exception called CustomError which is derived from the Exception class. This new exception can be raised, like other exceptions, using the raise statement with an optional error message.

When we are developing a large Python program, it is a good practice to place all the user-defined exceptions that our program raises in a separate file. Many standard modules do this. They define their exceptions separately as exceptions.py or errors.py (generally but not always).

User-defined exception class can implement everything a normal class can do, but we generally make them simple and concise. Most implementations declare a custom base class and derive others exception classes from this base class. This concept is made clearer in the following example.

Example: User-Defined Exception in Python

In this example, we will illustrate how user-defined exceptions can be used in a program to raise and catch errors.

This program will ask the user to enter a number until they guess a stored number correctly. To help them figure it out, hint is provided whether their guess is greater than or less than the stored number.

```
# define Python user-defined exceptions

class Error(Exception):

  """Base class for other exceptions"""

  pass

class ValueTooSmallError(Error):

  """Raised when the input value is too small"""

  pass

class ValueTooLargeError(Error):

  """Raised when the input value is too large"""

  pass

# our main program

# user guesses a number until he/she gets it right

# you need to guess this number

number = 10


while True:

  try:

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

    if i_num < number:

      raise ValueTooSmallError

    elif i_num > number:

      raise ValueTooLargeError
```

```
        break
    except ValueTooSmallError:
        print("This value is too small, try again!")
        print()
    except ValueTooLargeError:
```

Here is a sample run of this program.

1. Enter a number: 12
2. This value is too large, try again!
3.
4. Enter a number: 0
5. This value is too small, try again!
6.
7. Enter a number: 8
8. This value is too small, try again!
9.
10. Enter a number: 10
11. Congratulations! You guessed it correctly.

Defining Clean-up Actions

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

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

A *finally clause* is always 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 (or it has occurred in an except or else clause), it is re-raised after the **finally** clause has been executed. The finallyclause is also executed "on the way out" when any other clause of the try statement is left via a **break, continue or return** statement. A more complicated example:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the **finally** clause is executed in any event. The **TypeError** raised by dividing two strings is not handled by the **except** clause and therefore re-raised after the finally clause has been executed.

In real world applications, the **finally** clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

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. Look at the following example, which tries to open a file and print its contents to the screen.

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

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. 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, end="")
```

After the statement is executed, the file *f* is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

Assertions in Python

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.

- The easiest way to think of an assertion is to liken it to a raise-ifstatement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.
- Assertions are carried out by the assert statement, the newest keyword to Python, introduced in version 1.5.
- Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

The assert Statement

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an AssertionError exception.

The syntax for assert is −

Assert Expression [, Arguments]

If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception, using the try-except statement. If they are not handled, they will terminate the program and produce a traceback.

Example

Here is a function that converts a given temperature from degrees Kelvin to degrees Fahrenheit. Since 0° K is as cold as it gets, the function bails out if it sees a negative temperature −

```
#!/usr/bin/python3

def KelvinToFahrenheit(Temperature):
   assert (Temperature >= 0),"Colder than absolute zero!"
   return ((Temperature-273)*1.8)+32
print (KelvinToFahrenheit(273))
print (int(KelvinToFahrenheit(505.78)))
print (KelvinToFahrenheit(-5))
```

When the above code is executed, it produces the following result −

```
32.0
451
Traceback (most recent call last):
File "test.py", line 9, in <module>
print KelvinToFahrenheit(-5)
File "test.py", line 4, in KelvinToFahrenheit
assert (Temperature >= 0),"Colder than absolute zero!"
AssertionError: Colder than absolute zero!
```

Module-V

Graphical User Interface

A person who interacts with a software or application is called a 'user'. There are two ways for a user to interact with any application. The first way is where the user gives some commands to perform the work. For example, if he wants to print a file's contents, he can type PRINT command. Here the user should know the syntax and correct usage of PRINT command. Only then he can interact with the application. This type of environment where the user uses commands or characters to interact with the application is called CUI (Character User Interface).One example for CUI is MS-DOS operating system. The disadvantage of CUI is that the user has to remember several commands and their usage with correct syntax. A person who does not know anything in computers will find CUI very difficult.

The second way to interact with an application is through graphics, pictures or images. Here the user need not remember any commands. He can perform the task just by clicking on relevant images. For example, to send data to printer, the user will simply click on the Printer image (or picture). Then he has to tell how many copies he wants and the printing will continue. This environment where the user can interact with an application through graphics or images is called GUI (Graphical User Interface). One example for GUI is Windows operating system. 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.
- 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. 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 following are the general steps involved in basic GUI programs:

1. 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.

2. 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.

3. 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'.

4. 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.

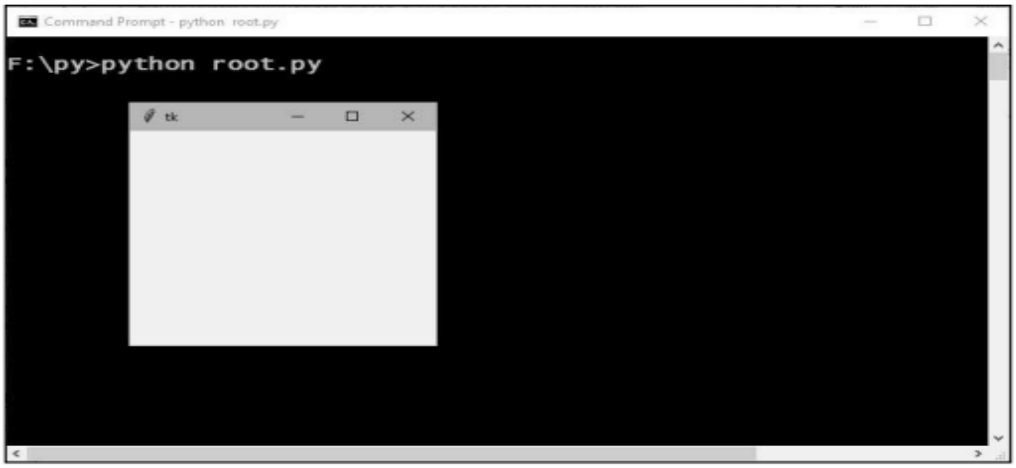
The Root Window

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'. We can say that the root window is the highest level GUI component in any tkinter application. We can reach this root window by creating an object to Tk class. This is shown in Program 1. 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 1: 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:

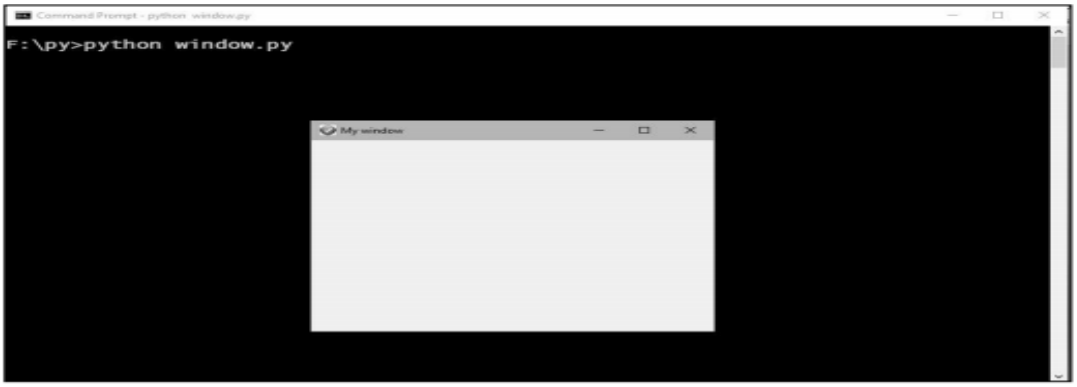


Program 1 can be improved to display our own title in the root window's title bar. We can also set the size of the window to something like 400 pixels X 300 pixels. It is also possible to replace the TK's leaf image with an icon of our choice. For this, we need to use the .ico file that contains an image. These improvements are shown in Program 2.

Program 2: A Python program to create root window with some options.

```
from tkinter import *

# create top level window
root = Tk()

# set window title
root.title("My window")

# set window size
root.geometry("400x300")

# set window icon
root.wm_iconbitmap('image.ico')
# display window and wait for any events
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')
```

Here, the font family name is 'Times' and font size is 40 pixels. If the size is a positive number, it indicates size in points. If the size is a negative number, it indicates size in pixels. The style of the font can be 'bold', 'italic', 'underline', 'overstrike'. We can mention any one or more styles as a string. The following program is useful to know the available font families in your system. All fonts are available in tkinter.font module inside tkinter module.

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)
```

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#00ffffis 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:

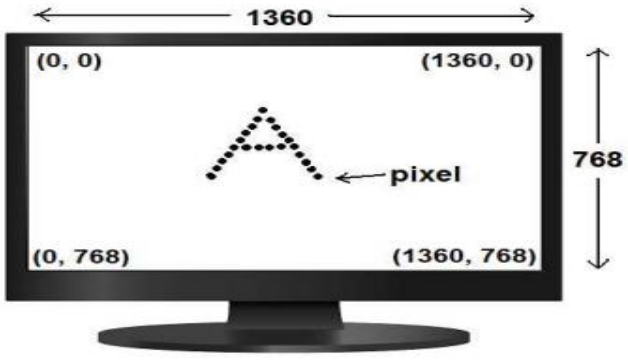
- Canvas: This is a container that is generally used to draw shapes like lines, curves, arcs and circles.
- Frame: This is a container that is generally used to display widgets like buttons, check buttons or menus. After creating the root window, we have to create space, i.e. the container in the root 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. When the monitor screen resolution is 1360 X 768, it indicates that the screen can accommodate 1360 pixels width-wise and 768 pixels height-wise. To understand the pixels and screen coordinates in pixels. The top left corner of the screen will be at (0, 0) pixels as x and y coordinates. When we move from left to right, the x coordinate increases and when we move from top to bottom, the y coordinate increases. Thus the top right corner will be at (1360, 0). The bottom left corner will be at (0, 768) and the bottom right corner will be at (1360, 768).



The pixels and screen coordinates

Colors in the canvas 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 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#00ffffis cyan (green plus blue).

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")
```

This creates a line with the connecting points (50, 50), (200, 50) and (200, 150). 'width' specifies the width of the line. The default width is 1 pixel. 'fill' specifies the color of the line. The create line() method returns an identification number.

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")
```

This creates an oval in the rectangular area defined by the top left coordinates (100,100) and bottom lower coordinates (400, 300). If the rectangle has same width and height, then the oval will become a circle. 'width' represents the width of the oval in pixels. 'fill' represents the color to fill and 'outline' represents the color to be used for the border. The option 'active fill' represents the color to be filled when the mouse is placed on the oval.

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")
```

Here, the polygon is created using the points (10, 10), (200, 200), (300, 200) and then the last point is again connected to the first point, i.e. (10, 10). The option 'smooth' can become 0 or 1. If 0, it indicates a polygon with sharp edges and 1 indicates a polygon with smooth edges.

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:

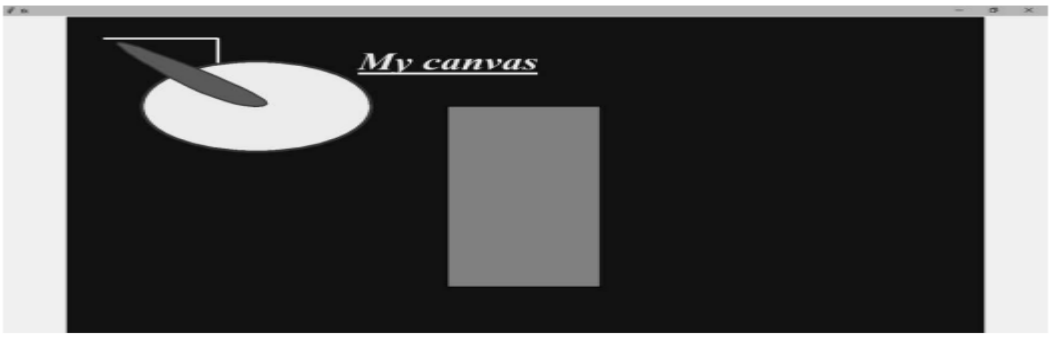
```
fnt =('Times', 40, 'bold')
fnt =('Times', -40, 'bold italic underline')
```

The first option ‘Times’ represents the font family name. This can be ‘Times’, ‘Helvetica’, ‘Courier’, etc. The second option represents a number (40) that indicates the size of the font in points. If we want to mention the size of the font in pixels, we should use minus sign before the size (-40). The third option indicates the style of the font. There are 3 styles: bold, italic and underline. We can mention any one or all the styles. If do not want any style, we need not mention this option.

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()
```

Output:

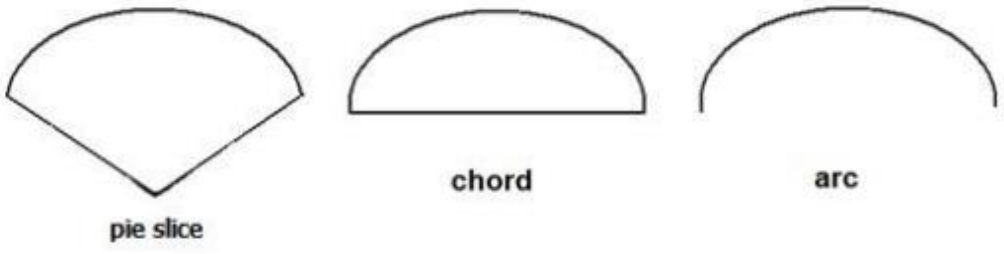


Another important shape that we can draw in the canvas is an arc. An arc represents a part of an ellipse or circle. Arcs can be created using the create arc () method as:

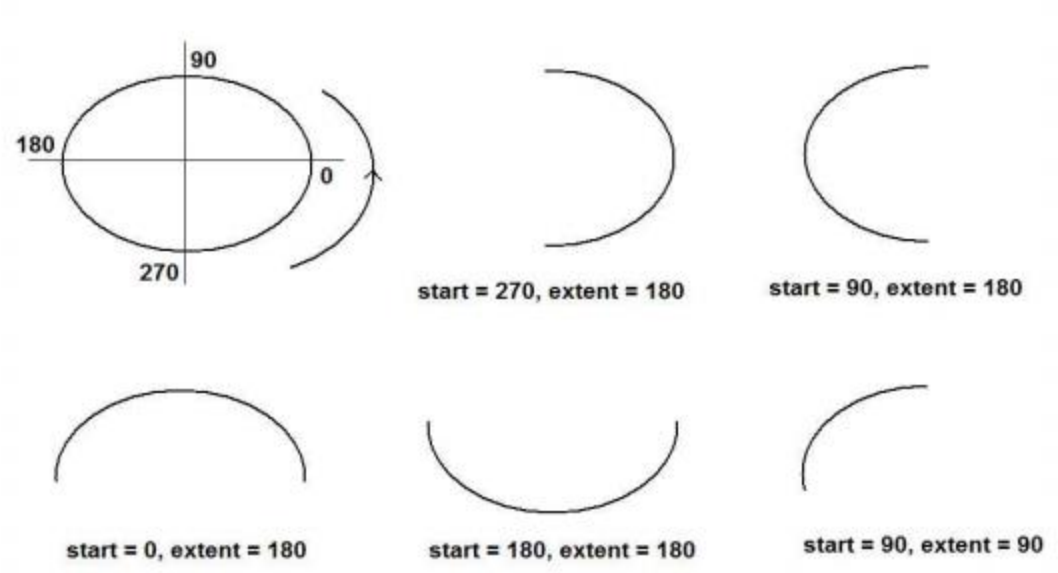
```
id = c.create_arc(100, 100, 400, 300, width=3, start=270, extent=180,
     outline="red", style="arc")
```

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”.



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.



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

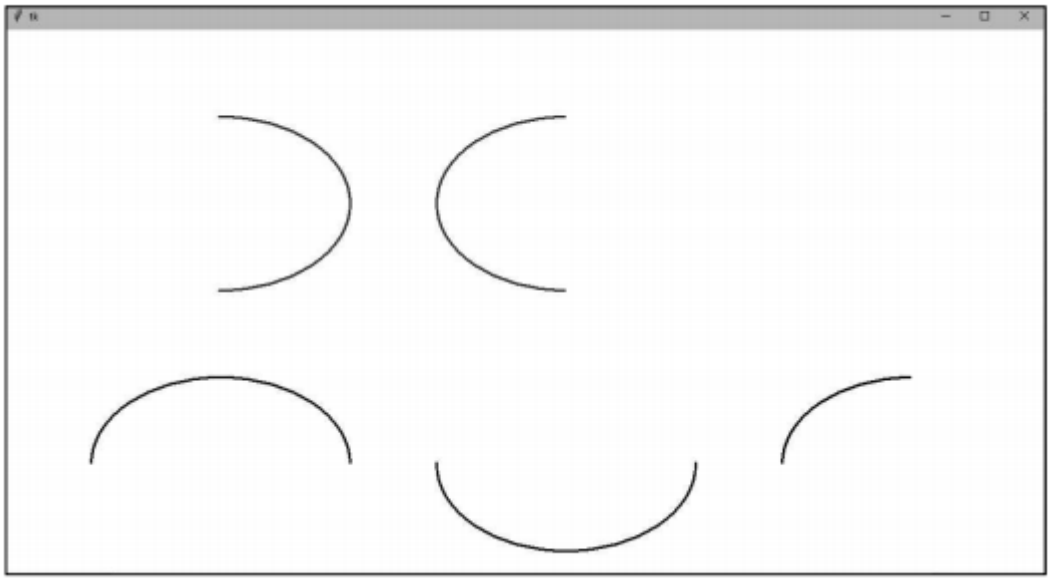
```
id = c.create_arc(500, 100, 800, 300, width=3, start=90, extent=180,
outline="red", style="arc")
```

Here, the arc is created in the rectangular space defined by the coordinates (500, 100) and (800, 300). The arc starts at 90 degrees and extends by 180 degrees. So, the arc’s ending position will be at 90+180 = 270 degrees.

Program: A Python program to create arcs in different shapes.

```
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)

# create arcs in the canvas
id = c.create_arc(100, 100, 400, 300, width=3, start=270, extent=180,
     outline="red", style="arc")
id = c.create_arc(500, 100, 800, 300, width=3, start=90, extent=180,
     outline="red", style="arc")
id = c.create_arc(100, 400, 400, 600, width=3, start=0, extent=180,
     outline="blue", style="arc")
id = c.create_arc(500, 400, 800, 600, width=3, start=180, extent=180,
     outline="blue", style="arc")
id = c.create_arc(900, 400, 1200, 600, width=3, start=90, extent=90,
     outline="black", style="arc")

# add canvas to the root
c.pack()

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

Output:



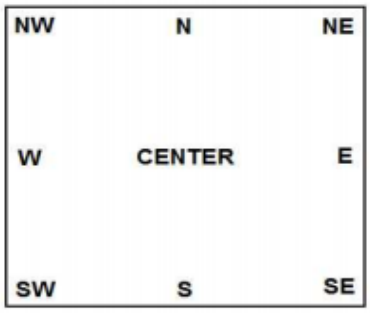
We can display an image in the canvas with the help of create image() method. Using this method, we can display the images with the formats .gif,.pgm,or.ppm. We should first load the image into a file using Photo Image class as:

```
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)
```

Here, the image is displayed relative to the point represented by the coordinates (500, 200). The image can be placed in any direction from this point indicated by 'anchor' option. The directions are represented by actual 8 directions on the earth: NW, N, NE, E, SE, S, SW, W and CENTER. 'activeimage' represents the image file name that should be displayed when the mouse is placed on the image.



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()
```

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.

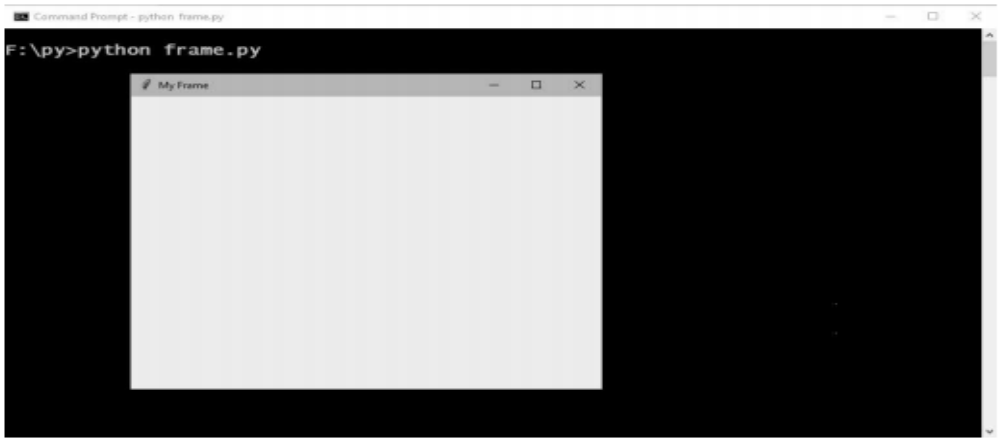
Once the frame is created, it should be added to the root window using the pack () method as follows:

```
f.pack()
```

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 2 Label 3 Message 4 Text 5 Scrollbar 6 Check button 7 Radio button 8 Entry 9 Spin box 10 List box 11 Menu

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')
```

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. ‘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)
```

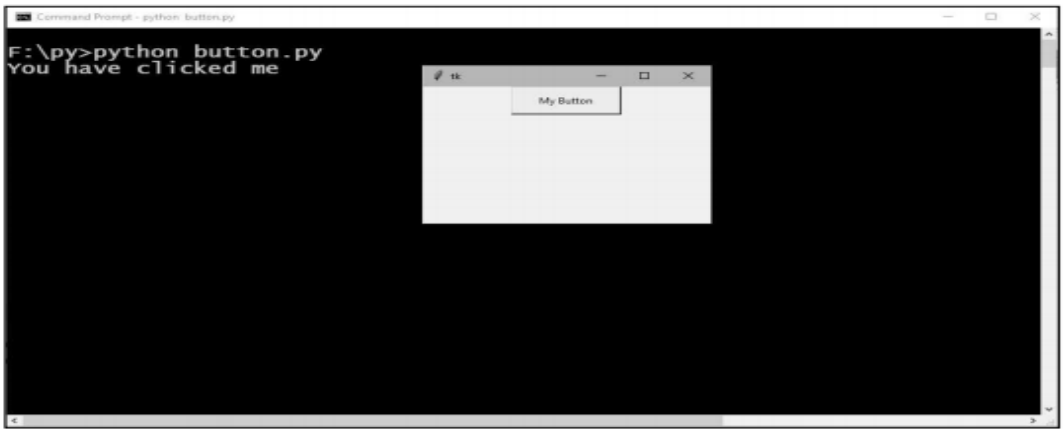
Here, <Button-1> represents the mouse left button that is linked withbuttonClick() method. It means when the mouse left button is clicked, the buttonClick() method is called. This method is called event handler.

In the place of <Button-1>, we can also use <Button-2>. In this case, mouse middle button is linked with the event handler method. The middle button is not found in most of the mouses now-a-days. <Button-3> represents the mouse right button. Similarly, <Enter> represents that the event handler method should be executed when the mouse pointer is placed on the push button.

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:



In the preceding program, we want to make 2 modifications. First, we will eliminate bind() method and we will use 'command' option to link the push button with event handler function as:

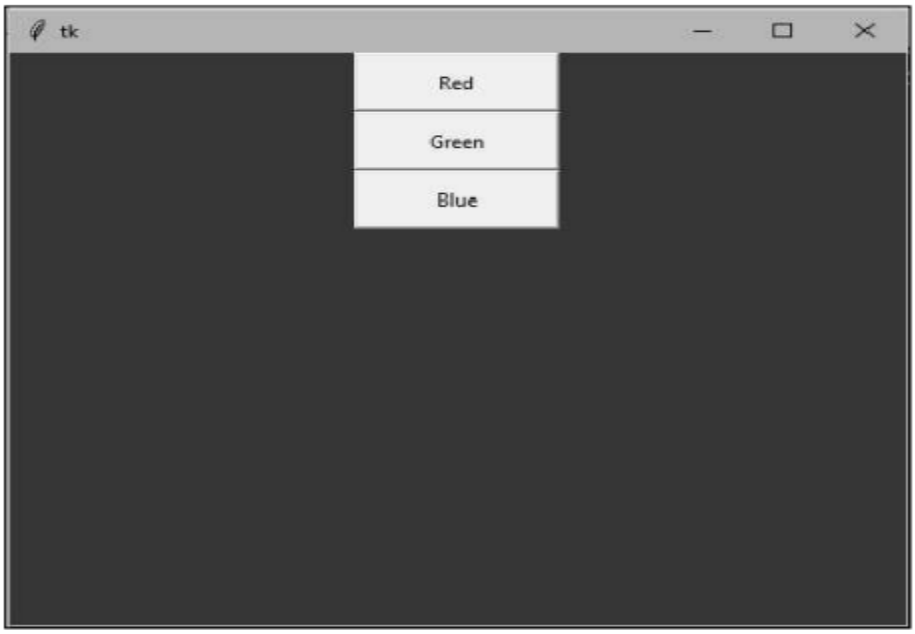
```
b = Button(f, text='My Button', width=15, height=2, bg='yellow',
     fg='blue', activebackground='green', activeforeground='red',
     command=buttonClick)
```

Secondly, we will rewrite the same program using class concept where the entire code will be written inside a class.

Program: A Python program to create three push buttons and change the background of the frame according to the button clicked by the user.

```
from tkinter import *

class MyButton:
    # constructor
    def __init__(self, root):
        # create a frame as child to root window
        self.f = Frame(root, height=400, width=500)

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

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

    # create 3 push buttons and bind them to buttonClick method and
# pass a number
        self.b1 = Button(self.f, text='Red', width=15, height=2,
                  command=lambda: self.buttonClick(1))
        self.b2 = Button(self.f, text='Green', width=15, height=2,
                  command=lambda: self.buttonClick(2))
        self.b3 = Button(self.f, text='Blue', width=15, height=2,
                  command=lambda: self.buttonClick(3))

        # attach buttons to the frame
        self.b1.pack()
        self.b2.pack()
        self.b3.pack()

    # method to be called when the button is clicked
    def buttonClick(self, num):
        # set the background color of frame depending on the button
        # clicked
        if num==1:
        self.f["bg"]= 'red'
        if num==2:
        self.f["bg"]= 'green'
        if num==3:
        self.f["bg"]= 'blue'

    # create root window
    root = Tk()
    # create an object to MyButton class
    mb = MyButton(root)

    # 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.

To creating two push buttons. We display 'Click Me' on the first button. When this button is clicked, we will display a label "Welcome to Python". This label is created in the event handler method buttonClick() that is bound to the first button. We display another button 'Close' that will close the root window upon clicking. The close button can be created as:

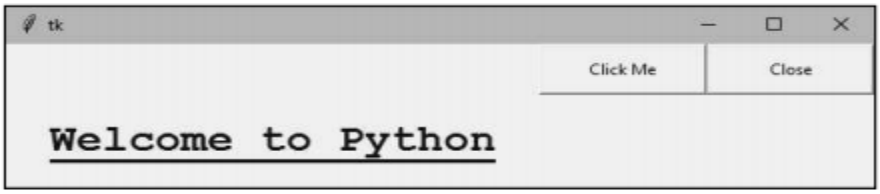
```
b2 = Button(f, text='Close', width=15, height=2, command=quit)
```

Please observe the 'command' option that is set to 'quit'. This represents closing of the root window.

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.

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)
```

Here, 't' represents the object of Text class. 'root' represents an object of root window or frame. 'width' represents the width of the Text widget in characters. 'height' represents the height of the widget in lines. The option 'wrap' specifies where to cut the line. wrap=CHAR represents that any line that is too long will be broken at any character. wrap=WORD will break the line in the widget after the last word that fits in the line. wrap=NONE will not wrap the lines. In this case, it is better to provide a horizontal scroll bar to view the lines properly in the Text widget.

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')
```

Here, the first argument END represents that the text is added at the end of the previous text. We can also use CURRENT to represent that the text is added at the current cursor position.The second argument is the text that is added to the 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'))
```

Here, we are applying 'red' background and 'white' foreground and 'Lucida console' font to the text that is already named as 'start' tag. In this way, we can have several tags in the Text widget. In many cases, it is useful to add scroll bars to the Text widget. A scroll bar is a bar that is useful to scroll the text either horizontally or vertically. For example, we can create a vertical scroll bar by creating an object to Scrollbar class as:

```
s = Scrollbar(root, orient=VERTICAL, command= t.yview)
```

Here, 'orient' indicates whether it is a vertical scroll bar or horizontal scroll bar. The 'command' option specifies to which widget this scroll bar should be connected. 't.yview' represents that the scroll bar is connected to't', i.e. Text widget and 'yview' is for vertical scrolling.

```
t.configure(yscrollcommand=s.set)
```

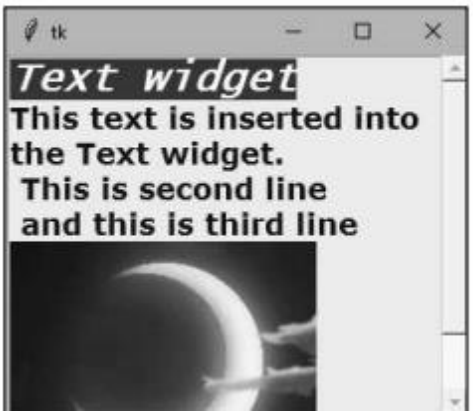
We should set the 'yscrollcommand' option of the Text widget't' to 's.set' method. In this way, the scroll bar's' is connected to Text widget't'.

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()
```

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 Radiobutton 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:

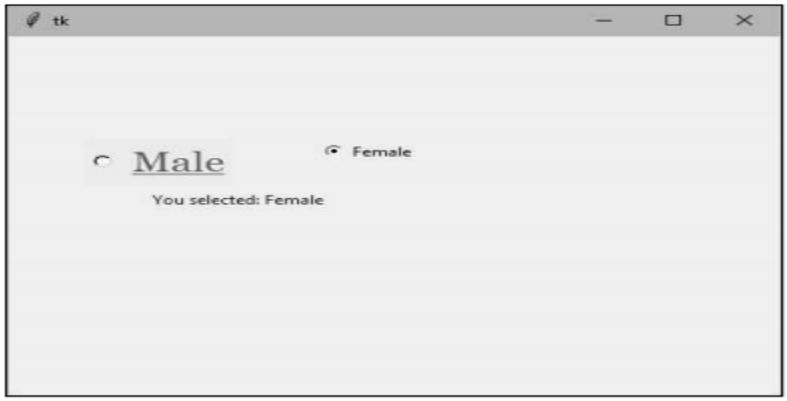
```
var = IntVar()
```

When the user clicks the radio button, the value of this ‘var’ is set to the value given in ‘value’ option, i.e. 1. It means ‘var’ will become 1 if the radio button ‘r1’ is clicked by the user. In this way, it is possible to know which button is clicked by the user.

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()
```

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='*')
```

Here, 'e1' is the Entry class object. 'f' indicates the frame which is the parent component for the Entry widget. 'width' represents the size of the widget in number of characters. 'fg' indicates the fore ground color in which the text in the widget is displayed. 'bg' represents the back ground color in the widget. 'font' represents a tuple that contains font family name, size and style. 'show' represents a character that replaces the originally typed characters in the Entry widget. For example, show='*' is useful when the user wants to hide his password by displaying stars in the place of characters.

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):
```

As seen in the preceding code, we are catching the event through an argument 'event' in the display () method. This argument is never used inside the method. The method consists of the code that is to be executed when the user pressed Enter button.

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()
```

Output: