PYTHON

LECTURE-I

INTRODUCTION TO PYTHON

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:

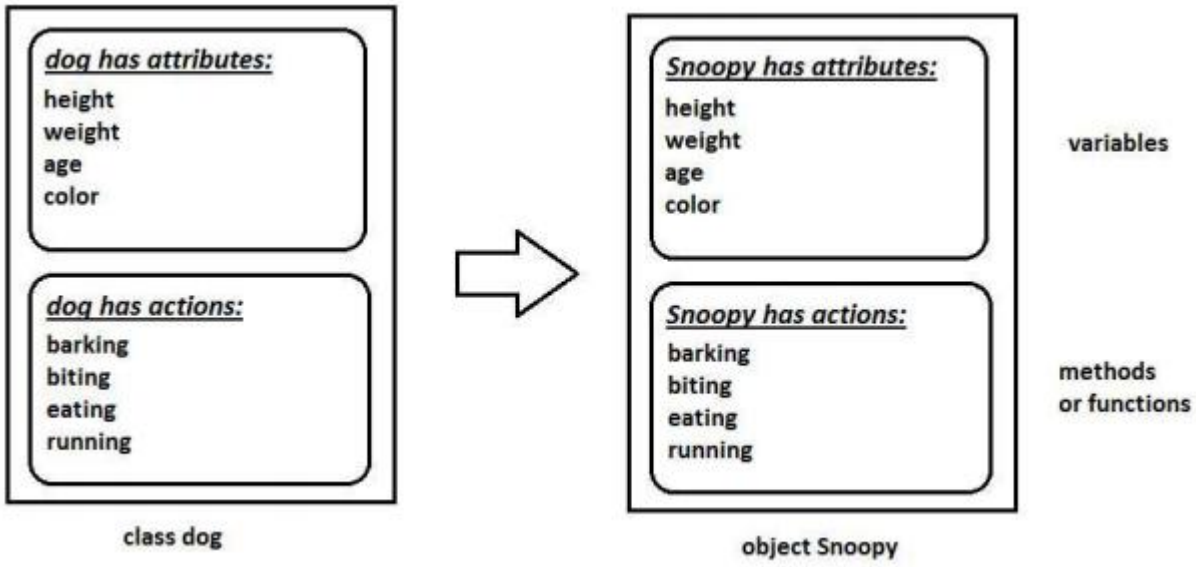


Fig:A Class and its object

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:

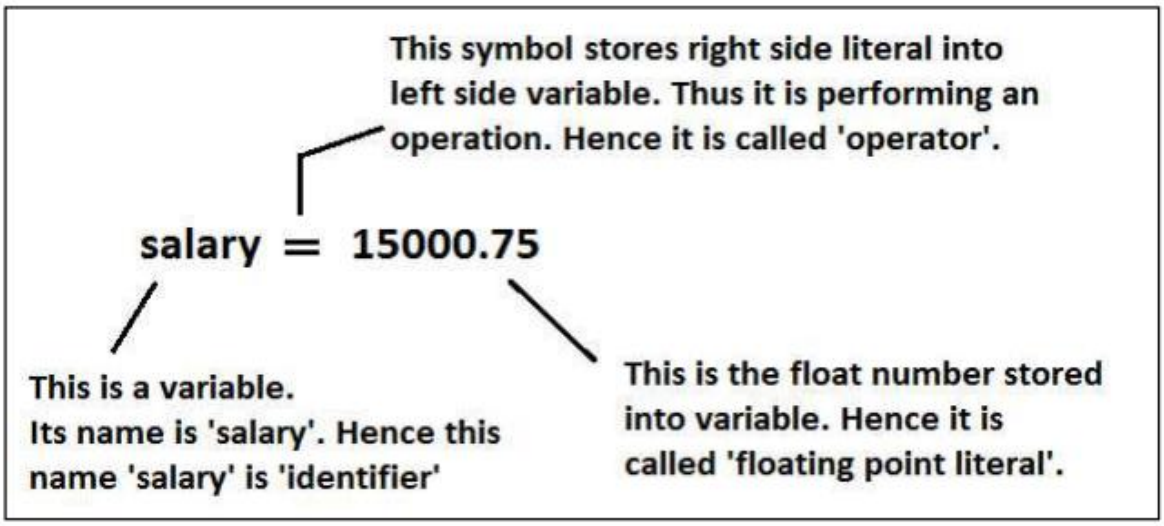


Figure 3.12: Variable, Operator and Literal

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 (_).

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.

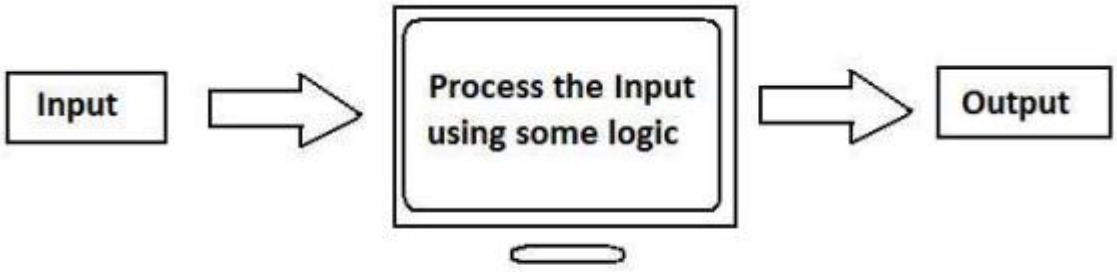


Figure 5.1: Processing Input by the Computer

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

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. Relational operators
4. Logical operators
5. Bitwise operators
6. Membership operators
7. Identity operators

Arithmetic operators

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

Arithmetic operators in Python

Operator	Meaning	Example
+	Add two operands or unary plus	x + y +2
-	Subtract right operand from the left or unary minus	x - y -2
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y (remainder of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y (x to the power y)

Assignment operators

Assignment operators are used to assign values to the variables.

Assignment operators in Python

Operator	Example	Equivatent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

Relational operators

Relational operators are used to compare two values

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Logical operators

Logical operators are used to combine conditional statements

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is false.

Bitwise operators

Bitwise operators are used to compare (binary) numbers

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Membership operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators

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

Identity operators

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

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

LECTURE – II

CONTROL STRUCTURES

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. 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
```

range() function

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

Syntax

```
range(start, stop, step)
```

Nested for Loop

Python programming language allows to use one loop inside another loop.

```
for iterator_var in sequence:
        for iterator_var in sequence:
          statements(s)
          statements(s)
```

example

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry“]
for x in adj:
  for y in fruits:
    print(x, y)
```

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
```

Syntax:

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

Example

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

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.

Syntax:

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

Example

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

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.

Syntax:

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

Example

```
for i in range(6):
   if i == 4:
      continue
   else:
      pass
   print(i)
```

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.

LECTURE-III

LIST, TUPLES, DICTIONARY AND ARRAYS

The most basic data structure in Python is the sequence. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth. The most common ones are lists and tuples. Certain operations on them are indexing, slicing, adding, multiplying, checking for membership, finding the length of a sequence and finding its largest and smallest elements.

List

A list is a collection which is ordered and changeable. In Python lists are written with square brackets.

Create a List:

```
list = ["aaa", "bbb", "ccc"]
```

Access Items

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index.

List Index

```
my_list = ['p','r','o','b','e']

print(my_list[0])        # Output: p

print(my_list[2])        # Output: o
```

Negative indexing

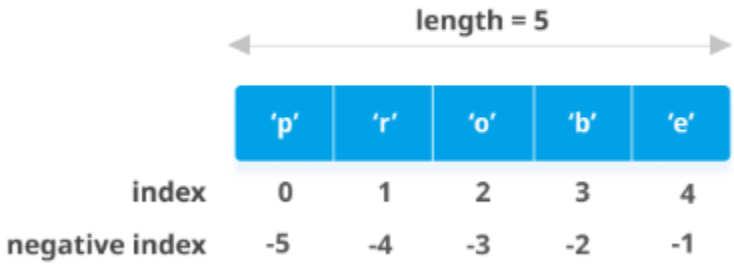
Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

```
my_list = ['p','r','o','b','e']

print(my_list[-1])       # Output: e

print(my_list[-5])       # Output: p
```

How to slice lists in Python?




```
my_list = ['p','r','o','g','r','a','m','i','z']
print(my_list[2:5])      # elements 3rd to 5th
print(my_list[:-5])      # elements beginning to 4th
print(my_list[5:])       # elements 6th to end
print(my_list[:])        # elements beginning to end
```

List Manipulations

Methods that are available with list object in Python programming are

FUNCTION	DESCRIPTION
Append()	Add an element to the end of the list
Extend()	Add all elements of a list to the another list
Insert()	Insert an item at the defined index
Remove()	Removes an item from the list
Pop()	Removes and returns an element at the given index
Clear()	Removes all items from the list
Index()	Returns the index of the first matched item
Count()	Returns the count of number of items passed as an argument
Sort()	Sort items in a list in ascending order
Reverse()	Reverse the order of items in the list
copy()	Returns a copy of the list

index()

In simple terms, the index() method finds the given element in a list and returns its position. If the same element is present more than once, the method returns the index of the first occurrence of the element.

The syntax of the index() method is:
list.index(element)

Example

```
# vowels list
vowels = ['a', 'e', 'i', 'o', 'i', 'u']
# index of 'e'
index = vowels.index('e')
print('The index of e:', index)
# index of the first 'i'
index = vowels.index('i')
print('The index of i:', index)
```

append()

To add an item to the end of the list, use the append() method.

The syntax of the append() method is:
list.append(item)

Example
```
list = ["aaa", "bbb", "ccc"]
list.append("ooo")
print(list)
```

insert()

The insert() method inserts an element to the list at a given index.

The syntax of insert() method is
list.insert(index, element)

Example
```
fruits = ['aaa', 'bbb', 'ccc']
fruits.insert(1, "ooo")
```

copy()

The copy() method returns a shallow copy of the list.

The syntax of copy() method is:
new_list = list.copy()

Example
```
fruits = ['aaa', 'bbb', 'ccc', 'ooo']
x = fruits.copy()
```

extend()
The extend() method adds the specified list elements (or any iterable) to the end of the current list.

The syntax for extend() method
list.extend(seq)

Example
```
b1 = ['cse', 'ece', 'eee']
b2 = ['mec', 'civ', 'aero']
b1.extend(b2)
```

count()

Python list method count() returns count of how many times *obj* occurs in list.

Syntax
The syntax for count() method

list.count(obj)

Example
List = [123, 'xyz', 'zara', 'abc', 123];
print "Count for 123 : ", List.count(123)
print "Count for zara : ", List.count('zara')

remove()

The remove() method removes the first occurrence of the element with the specified value.

Syntax
list.remove(elmnt)

Example
fruits = ['aaa', 'bbb', 'ccc']
fruits.remove("bbb")

pop()

The pop() method removes the element at the specified position.

Syntax
list.pop(pos)

Example
languages = ['Python', 'Java', 'C++', 'French', 'C']
return_value = languages.pop(3)
print('Return Value:', return_value)
print('Updated List:', languages)

reverse()

The reverse() method reverses the sorting order of the elements.

Syntax
list.reverse()

Example
os = ['Windows', 'macOS', 'Linux']
print('Original List:', os)
os.reverse()
print('Updated List:', os)

sort()

The sort() method sorts the elements of a given list in a specific order - Ascending or Descending.

The syntax of sort() method is:
list.sort(key=..., reverse=...)
Alternatively, you can also use Python's in-built function sorted() for the same purpose.
sorted(list, key=..., reverse=...)

Example

```
vowels = ['e', 'a', 'u', 'o', 'i']
vowels.sort()
print('Sorted list:', vowels)
```

len()

Python list function len() returns the number of elements in the list.

Syntax
The syntax for len() method
len(list)

Example

```
list1, list2 = [123, 'xyz', 'zara'], [456, 'abc']
print( "First list length : ", len(list1))
print ("Second list length : ", len(list2))
```

Nested List

A list can contain any sort object, even another list (sublist), which in turn can contain sublists themselves, and so on. This is known as nested list. A nested list is created by placing a comma-separated sequence of sublists.

Example1

```
L = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', 'h']
```

Example2

```
L = ['a', 'b', ['cc', 'dd', ['eee', 'fff']], 'g', 'h']
print(L[2])        # ['cc', 'dd', ['eee', 'fff']]
print(L[2][2])     # ['eee', 'fff']
print(L[2][2][0])  # eee
```

Tuples

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Create a Tuple:

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5 )
```

Accessing Values in Tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index.

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );
print("tup1[0]: ", tup1[0])
print("tup2[1:5]: ", tup2[1:5])
```

Basic Tuples Operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple.

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Tuples are sequences, indexing and slicing work the same way for tuples as they do for strings.

```
L = ('cse', 'ece', 'eee!')
print(L[2])
print(L[-2])
print(L[1:])
```

Built-in Tuple Functions

Python includes the following tuple functions

SN	Function	Description
1	cmp(tuple1, tuple2)	It compares two tuples and returns true if tuple1 is greater than tuple2 otherwise false.
2	len(tuple)	It calculates the length of the tuple.
3	max(tuple)	It returns the maximum element of the tuple.
4	min(tuple)	It returns the minimum element of the tuple.
5	tuple(seq)	It converts the specified sequence to the tuple.

len()

Python tuple method len() returns the number of elements in the tuple.

Syntax
The syntax for len() method
len(tuple)

```
tuple = ("aaa", "bbb", "ccc")
print(len(tuple))
```

min()

Python tuple method min() returns the elements from the tuple with minimum value.

Syntax
The syntax for min() method
min(tuple)

```
tuple1, tuple2 = (456, 123, 999), (111, 222, 333)
print("min value element in tuple1 : ", min(tuple1))
print("min value element in tuple2: ", min(tuple2))
```

max()

Python tuple method max() returns the elements from the tuple with maximum value.

Syntax
The syntax for max() method
max(tuple)

```
tuple1, tuple2 = (456, 123, 999), (111, 222, 333)
print("max value element in tuple1 : ", max(tuple1))
print("max value element in tuple2: ", max(tuple2))
```

count()

The count() method returns the number of times a specified value appears in the tuple.

Syntax
tuple.count(*value*)

```
tuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = tuple.count(5)
print(x)
```

index()

The index() method finds the first occurrence of the specified value. The index() method raises an exception if the value is not found.

Syntax
tuple.index(*value*)

```
tuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = tuple.index(8)
print(x)
```

Dictionary

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

Create a dictionary
Creating a dictionary is as simple as placing items inside curly braces {} separated by comma. An item has a key and the corresponding value expressed as a pair, key: value

```
# empty dictionary
my_dict = {}
# dictionary with integer keys
my_dict = {1: 'apple', 2: 'ball'}
# dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}
# using dict()
my_dict = dict({1:'apple', 2:'ball'})
# from sequence having each item as a pair
my_dict = dict([(1,'apple'), (2,'ball')])
```

Access elements from a dictionary
The items of a dictionary can be accessed by referring to its key name, inside square brackets.

```
my_dict = {'name':'Jack', 'age': 26}
```

```
print(my_dict['name'])   # Output: Jack
print(my_dict.get('age'))# Output: 26
```

Dictionary Length

To determine how many items (key-value pairs) a dictionary has, use the len() method.

Example

```
print(len(dict))
```

Adding Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

Example

```
dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
dict["color"] = "red"
print(dict)
```

Removing Items

There are several methods to remove items from a dictionary:

Example

The pop() method removes the item with the specified key name:

```
dict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
dict.pop("model")
print(dict)
```

Dictionary Methods

Python has a set of built-in methods that you can use on dictionaries.

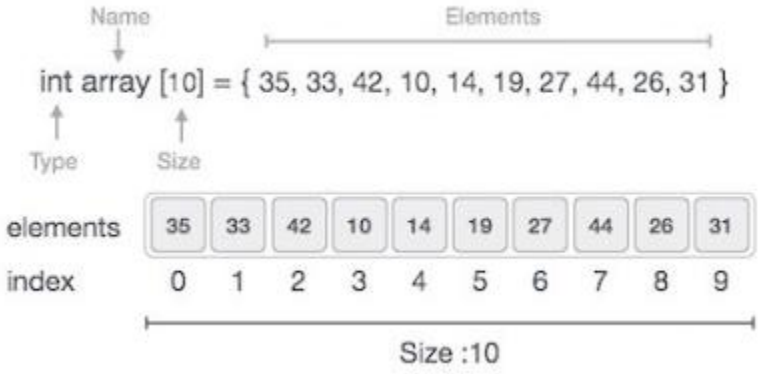
Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and values
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Arrays

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together.

Array in Python can be created by importing array module. array(data_type, value_list) is used to create an array with data type and value list specified in its arguments.

Array Representation



Basic Operations

Following are the basic operations supported by an array.

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.

```
import array as arr
a = arr.array('i', [2, 4, 6, 8])
print("First element:", a[0])
print("Second element:", a[1])
print("Last element:", a[-1])
```

Slice arrays

Access a range of items in an array by using the slicing operator :

```
import array as arr
numbers_list = [2, 5, 62, 5, 42, 52, 48, 5]
numbers_array = arr.array('i', numbers_list)
print(numbers_array[2:5]) # 3rd to 5th
print(numbers_array[:-5]) # beginning to 4th
print(numbers_array[5:])  # 6th to end
print(numbers_array[:])   # beginning to end
```

NumPy

NumPy is a general-purpose array-processing package. It provides a high-performance multidimensional array object, and tools for working with these arrays. It is the fundamental package for scientific computing with Python.

```
# Python program to demonstrate
# basic array characteristics
import numpy as np

# Creating array object
arr = np.array( [[ 1, 2, 3],
                 [ 4, 2, 5]] )

# Printing type of arr object
print("Array is of type: ", type(arr))

# Printing array dimensions (axes)
print("No. of dimensions: ", arr.ndim)

# Printing shape of array
print("Shape of array: ", arr.shape)

# Printing size (total number of elements) of array
print("Size of array: ", arr.size)

# Printing type of elements in array
print("Array stores elements of type: ", arr.dtype)
```

LECTURE-IV

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')
```

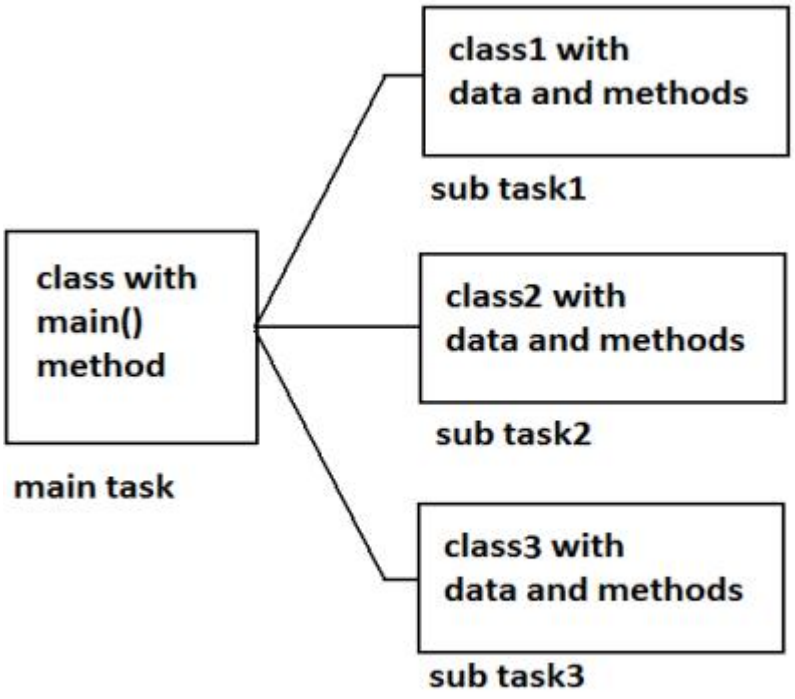
Output: C:\>python fun.py

```
 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
```

LECTURE –V

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)

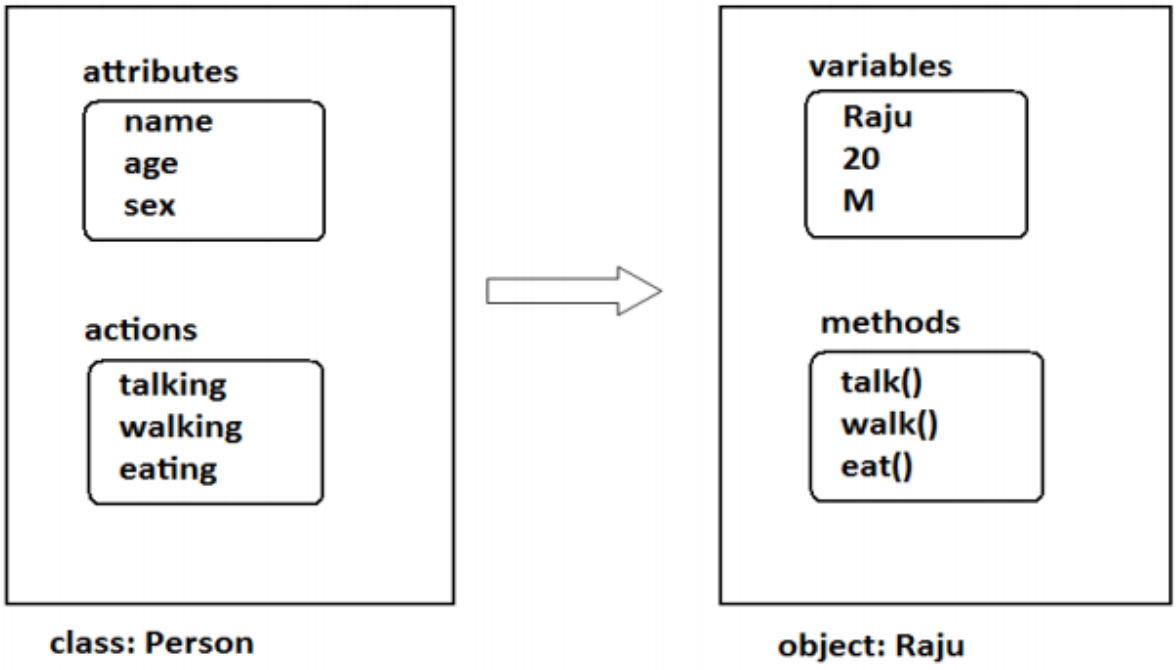
There are five important features related to Object Oriented Programming System. They are:

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.

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:

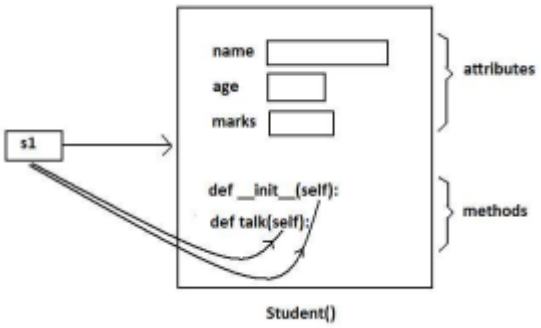


Figure 2.1: Student class instance in memory

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.

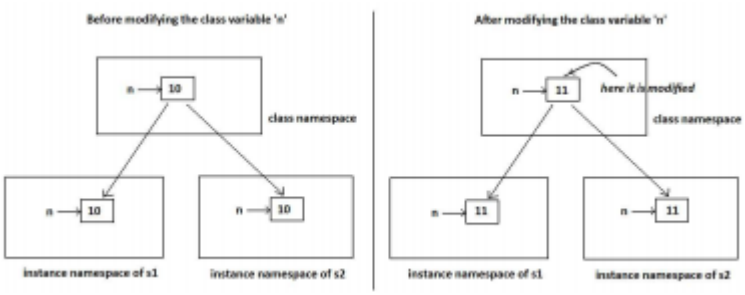


Figure 2: Modifying the class variable in the class namespace

#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.

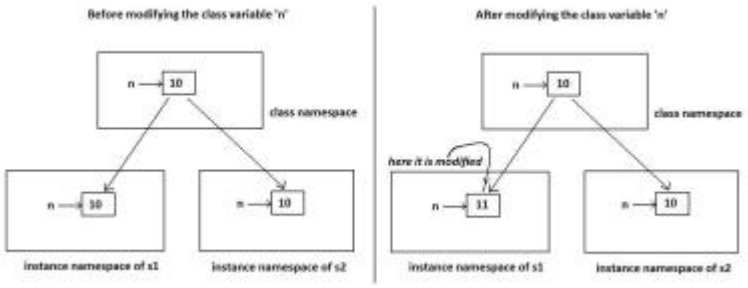


Figure 3: Modifying the class variable in the instance namespace

#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:

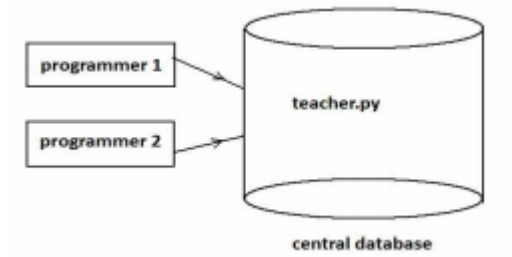


Figure 4: The teacher.py module is created and available in the project database

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.

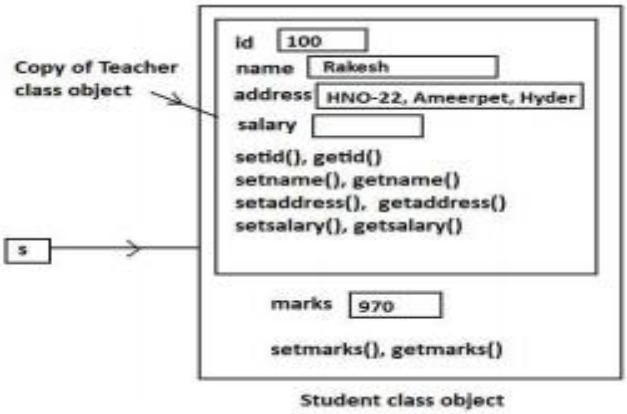


Figure 5: Student class object contains a copy of Teacher class object