UNIT2 INTRODUCTION TO PYTHON

Structure

2.0 INTRODUCTION

In this Unit, we will understand the overview of Python. Python language is a very interactive language. We discuss here the advantages and disadvantages of Python. In Python, there are various ways for executing the command, such as shell prompt, editors, and script mode. We take a brief look into understanding the syntax and semantics of data types, variables, expressions, and statements in python. Programming, functions, and modules are significant parts of any programming language. Break and continue statements are used to make interactive programming control flow statements.

2.1 LEARNING OUTCOMES

After going through this Unit, you will be able to:

- understand basic python concepts
- know the syntax of a programming language;
- explains function & modules;
- define control flow statements;

2.2 BASICS OF PYTHON

Python, an exceedingly resourceful language resembling everyday language, is astonishingly hassle-free to learn, even for novice computer programmers. It has witnessed an enormous surge in its acceptance after the announcement and growth of the Raspberry Pi, for which Python is the legitimately documented programming language.

Python provides a diversity of convenient built-in data structures, like lists, sets, and dictionaries. We can also use it for various system management, text processing, and internet-related tasks. Unlike many languages, its core language is minor and easy to acquire. An actual object-oriented language symbolizes an intuitive method for representing information and actions in a program.

Some of the features of python are dissimilar to other programming languages and should be acknowledged early on so that consequent examples don't seem puzzling. Like C and C++, Python statements do not need to end with a unique character. Python interpreter understands that we are done with a piece of individual information when we enter the 'Return' key from the keyboard. Apart from the sum of the rules that must be followed, we have a lot of independence when composing a program. One of these is using indentation to indicate the loops instead delimiters ({}) or keywords.

2.3 PYTHON'S ASSETS

- Object-oriented structured language: It supports multiple inheritances, polymorphism, and function overloading
- Indentation: The main feature of python is indentation. Due to this, Python knows which statement to execute next or which statement belongs to which block in the code.
- Open source: Downloading, installing, and customizing python is free and easy.
- An elegant syntax structure, which is easy to construct and read.
- An extensive library works in tandem with other programming tasks like modifying files, searching data, and communicating with web servers and data objects.
- Codes can be grouped into packages and modules if desired.
- Very powerful memory management with numerous third-party utilities.
- Portable: It can run virtually on every platform used.
- Platform independent: Python is platform independent.

- An intermediate compiler needs as they are compiled automatically into byte code, which the interpreter reads.
- It's learner-friendly, making it the most popular choice for entry-level coders.
- A vast and active community donates to Python's pool of modules and libraries.
- Interactive Programming: For writing programs, users can interact with the python interpreter.
- Syntax: Python has a straightforward syntax, making it popular among developers.
- Data analysis and machine learning: Python is the primary block in data science. Complex statistical calculations, data visualizations, machine learning algorithms, data manipulation, and research are achievable because python has many libraries.
- Web development: the back end of websites and applications is now being designed in python itself; here, it plays a significant role in communicating with the server, data processing, querying the databases, and URL routing.
- Automation or scripting: Python is very well suited to automate tasks. This can also be further extended in python for error detection across multiple files, removing duplications, doing basic calculations, converting files, etc.

2.4 PYTHON and OTHER LANGUAGES

Table 2.1

	Pros:	**Cons:**
Compiled languages: C, C++, Fortran…	• Very fast. • Can do heavy computations. • Difficult to outperform these languages.	• Distressing usage • No interactivity during development • Mandatory compilation steps.
MATLAB scripting language	• Rich collection of libraries with numerous algorithms. • Fast execution -compiler language. • Pleasant development environment. Integrated editor. • Availability of Commercial support.	• Underprivileged base language and can be restrictive for advanced users. • Not free.
Julia	• Fast code, yet interactive and simple. • Easily connects to Python or C.	• Ecosystem limited to numerical computing. • Still young.
Scripting languages: Scilab, Octave, R, IDL	• Open-source, free, or at least cheaper than MATLAB. • Some features can be very advanced.	• Fewer available algorithms
Python	• Rich scientific computing libraries. • Well thought out language • Free and open-source software. • Widely spread community. • A diversity of prevailing environments to work with.	• Not all the algorithms can be found in specialized software or toolboxes.

2.5 INVOKING PYTHON

2.5.1 Shell Prompt

Enter "python" at the shell command prompt itself. It will bring up the python interpreter with the following message:

```
Python 3.5.2 (default, Nov 23 2021, 16:37:01)
[GCC 5.4.0 20160609] on Linux
Type "help", "copyright", "credits" or "license" for
further information.
>>>
```

The ">>>" sign represents python's prompt; commands are written ahead of this fast, and just enter is pressed for python to run the command. If written with correct syntax, Python has the capability of executing a command typed after prompt immediately. So If we have given an executable command, like the print statement for printing "hello,world!" as input to the fast, we will get a response as shown below.

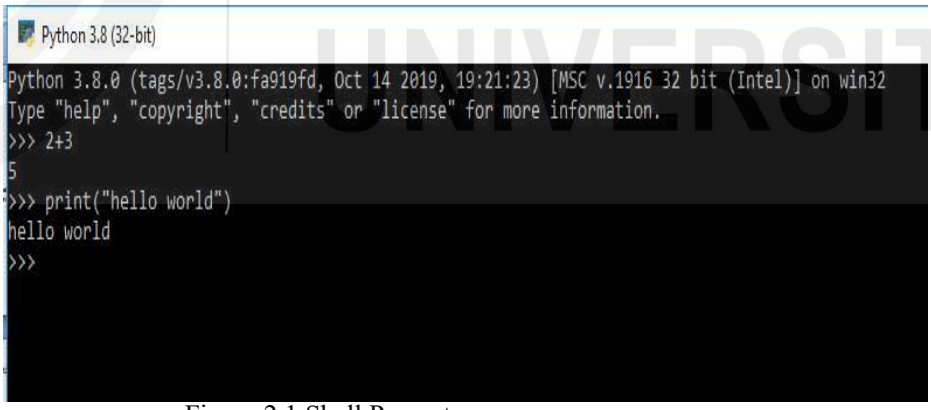

Figure 2.1 Shell Prompt

Regardless of how we invoked python in the first place, a new line is by default added at the finish of the printed string or the output.

Unless some value has been allocated to a variable, the value of the expression is printed as output as soon as it is given in input. This makes python very useful for calculation also.

Let us take a look at some of the examples

```
>>> cost = 27.00
>>> taxrate = .075
>>> cost * taxrate
2.025
>>> newcost=cost + (cost * taxrate)
>>>newcost
29.025 >>> 16+ 25 +92 * 3
317
```

2.5.2 Text Editor

When we compose more extended programs, the use of the python editor is crucial. These programs, irrespective of our OS, can be executed by just entering 'filename' prefixed by "python." The file extension of the files is generally ".py." Programmers write python source code in the file and use the interpreter's services to execute the file's contents directly.

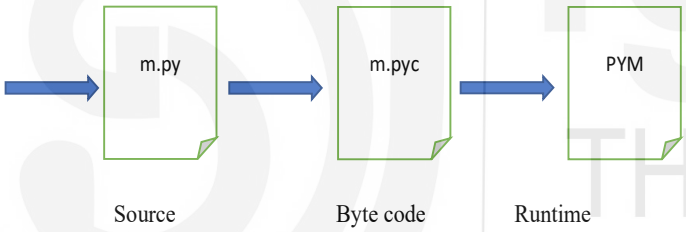

Figure 2.2 Editor

2.5.3 Script Mode

Alternatively, to execute the script through the interpreter, we pass on the file name to the interpreter. If we are working on UNIX and we have a script name File.py, and we run the script using the command."

```
'python file.py'
```

Though working in the interactive mode is far superior if coders work with small bits of code as they can be typed and executed directly if the code has more than 4-5 lines, using a script for coding enables us to modify and use the code even later on.

2.6 PYTHON CURRICULUM

The syntax of any language is a set of rules which generally define how a program will be written and interpreted. A basic Python curriculum can easily be broken down into four vital topics: data types, data structures, variables, loops, and functions.

2.6.1 Data Types

We access written information all the time but perhaps never think about the dissimilarity between numbers and text. But a computer undoubtedly has a big difference between numbers and text. This data is stored in memory and can be of different types. For example, an employee's address is stored in alphanumeric characters, and the phone number will be held in numeric values. Python also offers many standard data types used for defining the operations. These are as follows:

- **Integer**

Int, or integer, either positive or negative, is a whole number without decimals and of unlimited length.

```
>>>print(24687654+2)
24687656
>>>print(70)
70
>>> a=100
>>>print(a)
100
```

Verifying the type of any object:

```
>>>type(11)
<<class 'int'>
>>>a=10
>>>printtype(a))
```

- **Float**

"Float is a number which can be positive or negative and may contain fraction represented in decimal format.

Scientific numbers with an "e" indicate the power of 10.

```
">>> y=7.9
>>> y
7.9
>>> print(type(y))
```

```
<class 'float'>
>>> type(.4)
<class 'float'>
>>> x= 35e3
>>>print(type(x))
<class 'float'>”
```

- **Boolean**

Objects of Boolean contain one of only two values:

```
True and False:
‘>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>’
```

- **String**

Strings are identified as a connecting set of characters characterized in quotation marks. Pairs of single or double quotes can be used in python to denote. ‘hello' is the same as "hello."Names, addresses, and all other kinds of text that we use daily are strings in Python

```
>>>print(‘HOW ARE YOU”)
HOW ARE YOU
```

Following is a table of escape sequences that suppresses the distinct traditional interpretation of a character in a string:

Table 2.2

Escape Sequence	Usual Interpretation	“Escaped” Interpretation
\'	Terminates string with single quote opening delimiter	Literal single quote (') character
\"	Terminates string with double quote opening delimiter	Literal double quote (") character
\newline	Terminates input line	Newline is ignored
\\	Introduces escape sequence	Literal backslash (\) character

- **List**

A collection of data, typically associated with each other. They are kept together as a list as an alternative to storing the data as distinct variables. For example, if we need to store the ages of 8 users. Instead of keeping them as usr1Age, usr2Age, usr3Age, or usr4Age ……, …, it is more sensible to store them as a list.

Declaring a list:

listName = [initial values]

They are most widely used in data structures. They are well-organized and append able, allowing duplicate members

Example:

```
>>>list1=[1,7,9,'p',B'13,8[20,19]]
>>>print(list1)
[1,7,9,'p',B'13,8[20,19]]
```

Adding to list:

```
"myList.append("z")
print(List1)
[1,7,9,'p','B',13,8, [20,191],'z']"
```

- **printing a particular item from a list:**

```
print(list1[2])
```

##Note: the list starts from 0 to n-1(for n number of items),
#print the last item.

```
Print(list[-1])
z
```

#modify the second and print the updated list

```
">>> list1[1] = 6
>>> print(list1)
     [1,6,9,'p','B',13,8, [20,191],'z']"
```

- **Tuple**

These data types are also like lists, but values cannot be changed here. Values provided at first cannot be altered during the program.

For example, we need to store the name of the months of the year, and then we can take the use of tuples. Knowing beforehand there is no need to change the data is a must as selection criteria for using tuples.

To declare a tuple:

```
    >>>tupleName=(initialvalues)
    Example:
    daysofweek                                          =
    (”sun”,”mon”,tue”,”wed”,thur”,”fri”,”sat”)
Print(daysofweek)
```

- **Dictionary**

They are the pool of interrelated data pairs. The values stored are accessible by key. They are an assortment of name-value sets.

Every *key* is associated with some value, and then we have to use the key to access the values linked with that particular key.

The value stored in a key can be anything from a number, string, or maybe even a dictionary.

Dictionaries are wrapped in braces, {}, with a series of key-value pairs inside these braces.

Declaring a dictionary;

```
    dictonaryNmae={dictonaryKey:data}
    ## dictionary keys must be unique and should be an
    immutable type

    >>> phn = {'ace':4098,'pace':4139}
    >>> phn['guide']=4127
    >>> phn
     {'ace':4098,'pace':4139,'guide':4127}
```

If we want to access an individual item, we use the dictionary key:

```
    >>>phn[‘ace’]
    4098
```

Another way to construct a dictionary is using the dict() constructor, which constructs dictionaries straight from the arrangements of the key-value pairs given as input:

```
    >>> dict([('abc', 1000), ('qwe', 2000), ('pqr',
    3000)])
    {'abc': 1000, 'qwe': 2000, 'por': 3000}
```

A dictionary can also be declared even without assigning initial values to it:

dictname = { }
an empty dictionary with no items in it.

Adding items to a dictionary:

```
dictionaryName[dictionarykey] = data".

>>> dictnames"xyz"]=99
>>> dictname
{'xyz': 99}
```

2.6.2 Variables

Variables store values for which memory locations are explicitly reserved and separately. In other words, we give names to the data that has to be manipulated and stored by us further in the programs. The interpreter allocates memory as suggested by the user and then, depending on its depicted data type, allows what kind of data it will store.

Some basic guidelines for the variables:

- Variable name starts with a character or the underscore character.
- Names cannot have initials of a number like 5pie.
- Variable names can only be "alpha-numeric" characters and "underscores."
- Names are generally case-sensitive.

Assigning Values to Variables

No explicit declaration is required to reserve memory. This happens by default whenever we assign a value to any variable. Its name resides on the left side of the operator (=) and the operand to its right is the value that must be stored or assigned to that variable.

For Example −

```
"a= 787            # integer assignment
b = 30.88          # floating point
c = "Sam"          # string
print (a)
print (b)
print (c)
Output:
```

```
787
30.88
Sam”
```
The interpreter automatically determines the data type once the value is assigned to the variable.

Multiple Assignments:

Using commas, we can assign a single value to more than one variable and multiple values to multiple variables simultaneously.

```
>>> a=b=c=d=7001
>>>print(a)
7001
>>> print (c)
 7001
>>> a,b,c=9,8,7
>>>print(a)
9
>>>print(b)
8
```

Output Variables:

The print statement is primarily used to give the output of the variables. It is not necessary to declare with any specific type. And their type can be changed even after it has been set.

```
“x = 12           # x is int
x = "ok "    # now x is a str
print(x)
```

Output: ok”

Text and variable combine

```
‘Python uses the “+” operator:
x = "splendid"
print"Python is " + x)
```

Output

```
Python is splendid.’
```

2.6.3 Expression

Expressions are generally the arrangement of values, variables, and operators. They are assessed using assignment operators.

“Examples:

```
Y=x + 17
>>> x=10
>>> z=x+20
```

```
>>> z
30"
```

Python-defined expressions include identifiers, literals, and operators and are listed below:

Identifiers: Any name used to define a class, function, variable module, or object is an identifier.

Literals: These are language-independent expressions in Python and should exist independently in any programming language. In Python, there are string literals, byte literals, integer literals, floating point literals, and imaginary literals.

Operators: In Python, you can implement the following operations using

Check Your Progress 1

Note: a) Space is given below for writing your answers.

b) Compare your answers with the one given at the end of this Unit.

1) Explain each term with examples:

(i) Data Types

(ii) variables

(iii) Expression

(iv) Statements

(v) Comments

the corresponding tokens".

2.7 FUNCTIONS & MODULES

Reusability is one of the prominent traits of any programming language. This enables the users to reuse specific chunks of code repeatedly, even in various functionalities and programs. We use modules to accumulate multiple functions in one file, later used in various projects.

2.7.1 Functions and their uses

The function is the cluster of related statements which achieve a specific task. They are basically "pre-written codes" that can accomplish a particular task once or again and again. They aid in breaking the program into a smaller and modular portion of codes. When our programs grow into more extensive programs, they build more systematized and manageable ones. This also circumvents replication and thus making the code reusable. Some Functions do require the passing of data to perform their tasks.

For example, the print () command, which is itself a function, is used for that displaying text on the screen.

2.7.2 Built-in Functions

"They are those functions that are already built into the Python language,

Ex:

"bool(), abs (), all(),ascii(),……so on….

```
integer = -50
print('Absolute value of -50 is:', abs(integer))

Output:
Absolute value of -50 is: 50"
```

2.7.3 User Defined Functions

Functions that are defined by the users and are reused again and again in the program are called user-defined functions. We often use them to return some value or pass on the value to another resource.

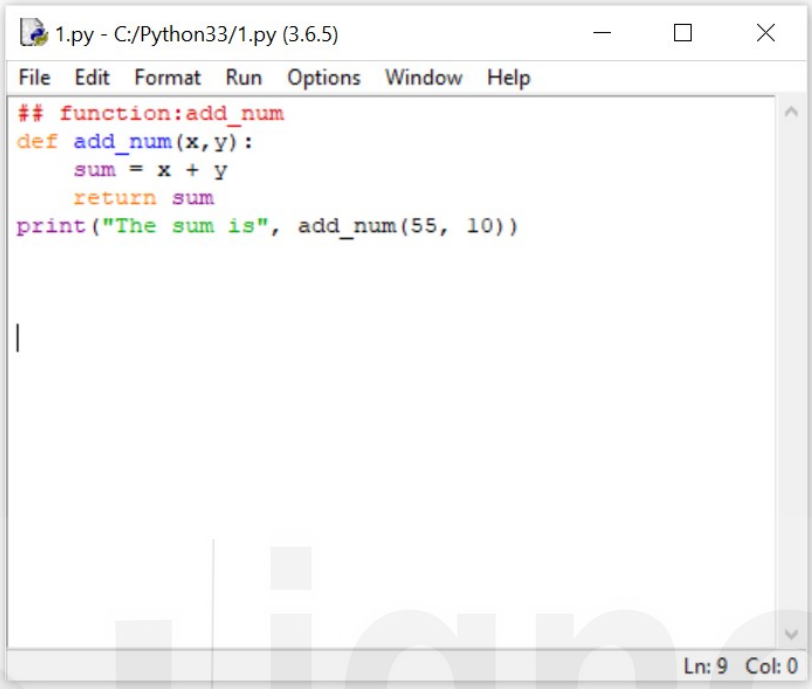

Figure 2.3 Example 1

```
Output:
The sum is '65"
```

Whenever data in the form of variables or another format are passed during the definition of the function, they are labeled as **parameters. But** if we give them when the specific function is passed, they are called **arguments**.

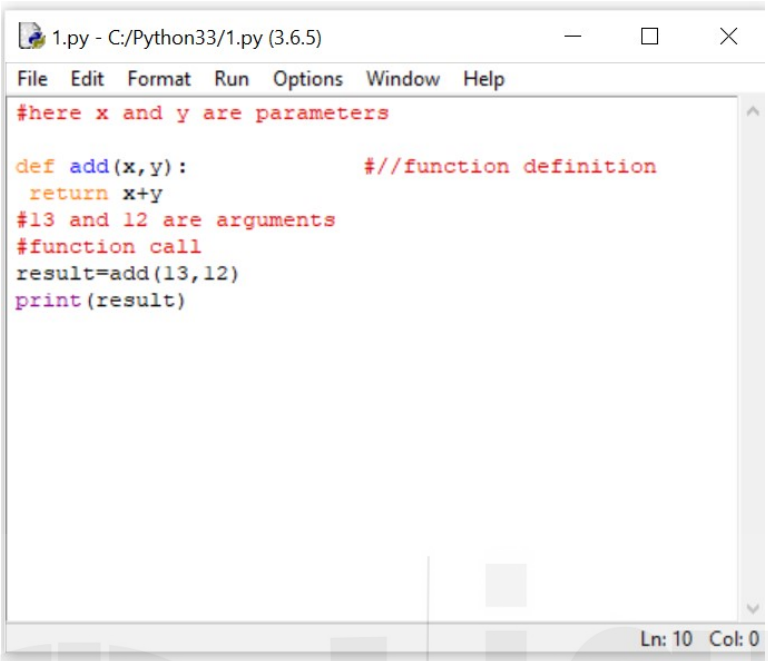

Figure 2.4: Example for passing arguments

```
Output:
25"
```

Further, we can classify these functions arguments into three categories mentioned below:

1. Default Arguments
2. Variable-length Arguments
3. Keyword Arguments

Syntax:

```
"def funct_name():
   statements
     .
     .
     .
funct_name ()""
```

Every function definition consists of the following rules:

1. Function Header, where they Keyword **def** specifies the start of the function. Here the name of the process is selected, with similar rules of writing function as in identifiers.

2. Then we have the option of passing values to the function. Here colon **(:) is used to** end of function header.
3. Then comes the body of the function. We pass some valid python statements in a structured way. This reflects the working of that particular function. Statements are generally given an indentation level of usually four spaces.
4. At last, If the function wishes to return values from the function. Then a statement has to be passed in this regard.

Keyword Arguments

Keyword arguments are associated with function calls. The caller recognizes the statements by the parameter name when used in a function call.

This allows the omission of arguments or positioning them out of order because the Python interpreter can use the keywords to match the values with parameters.

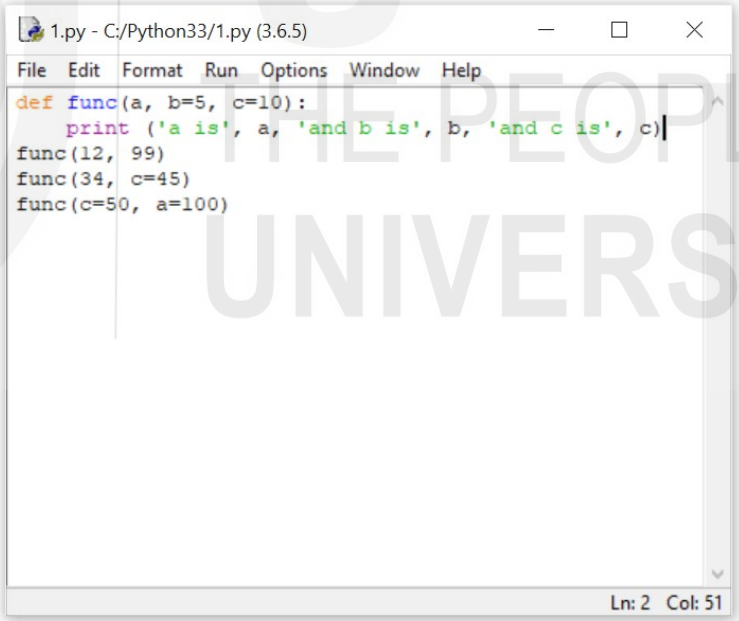

Figure 2.5 Keyword Argument

This also comes in handy when we have many parameters in the function argument and want to input only some of them. So instead

of the position where data has to be filled in, the name (keyword) is used.

This assists in two ways:

1.No doubts regarding the sequencing of the input data.

2. Selective input feeding is also achieved.

Output:

```
a is 12 and b is 99 and c is 10
a is 34 and b is 5 and c is 45
a is 100 and b is 5 and c is '50'
```

#Default Arguments

Some Function arguments can also have default values. This can be done by providing default values to an argument using the assignment operator (=).

```
def greet(name,msg="Good Day!"): """ This greets the
person with the supplied message. If the message is not
supplied, it defaults to "Good Day!" """

print("Hi",name + ', ' + msg)

greet("Mat") greet("Mark", "How do you do?")
```

INDENTATION

In python, we use indentation to describe control and loop. Colon symbols (:) as well as indentation, are employed in python to show where blocks of code begin and end. It is a method of describing to the python interpreter that a particular set of statements fits a specific block of code. This block is a sequence of statements. Block can be regarded as the grouping of statements for a particular purpose.

Most programming languages use braces { } to outline a block of code. Whereas in Python, we have an indentation for highlighting the blocks of code. Generally, all the statements with the same distance to the right belong to the same block of code. If a block gets more intensely nested, then it is merely indented further to the right. Some of the significant rules have also been listed below:

Rule-1:

If a colon sign is used at the end of a line, indentation is needed in the

following line/lines.

Rule-2:

If multiple lines are inside the if block, all the lines should be indented. And with same the indentation as well

Rule-3:

Regardless of the current indentation level, the indent is a must if there is a colon at the end of the line.

Rule-4:

Indentation is determined by the depth/level of indentation, not by the line number.

Indentation Rule-5:

If there is no colon, one should not indent the starting of a line.

Let's recap by looking at some code:

```
phrase = input("Talk to me > ")
if phrase == "hi":
 print("hello")
print("bye")
```

This small program contains one indent: it's before the code print("hello") and it is under if phrase == "hi":

```
if phrase == "hi":
 print("hello")
```

The indentation tells Python that the code print("hello") should only be run if phrase == "hi" is True. Any code indented under the if will form part of that if statement's scope.

```
if phrase == "hi":
 print("hello")
 print("this is also part of the if's scope")
print("not part of the if's scope")
```

Let's look at the program below, where there is another indent: after the else before print("sorry, I don't understand this").

```
phrase = input("Talk to me > ")
if phrase == "hi":
 print("hello")
else:
 print("sorry I dont understand this")
print("bye")
```

Using the same scope rules as the if statement, the print statement indented under the else is part of that else statement's scope.

```
else:
 print("sorry I dont understand this")
```

The final print("bye") is not indented under the if or the else, so it is not part of the if or the else. That makes sense as this line is to run no matter what phrase the user entered.

Without indentation, Python would not be able to determine what code to run as part of the if or the else. This is a common theme throughout Python and one you will use in the coming weeks.

2.7.4 Modules

Modules and python are used side by side. Whenever we use python, modules are there in different ways. A package consists of many modules, and a library contains a lot of packages. In python, using descriptive names for the functions, we can detach blocks of code from the main program. This also helps construct and miniate the main program, an effortless job. When stored separately by their discreet names, these detached are called modules.

When running the main programs, we can then import these modules into our main program. This is what makes python a modular language as code is bunched together. Packaging our functions in separate files allows us to conceal the low-level specifics of the main program's code, enabling us to emphasize the higher-level logic. As being stored separately, these modules can be reused in different programs that need not be connected. They can also be shared with other coders without sharing the whole programs. A key point in making the language open source is that many programmers worldwide can contribute to constructing and enhancing a program through modules without even looking into the main code of the program.

Syntax:

```
import <module-name>
```

There are even some modules that have been integrated into python; let us look into the following example:

```
#importing the whole module.
```

Let's print the value of pi:

```
 >>>print(pi)
NameError: name 'pi' is not defined.
```

pi isn't part of the standard library.

we have to import the math module

```
>>> import math
>>>print(math.pi)
3.141592653589793
```

Let's try print(pi) again:

```
>>>print(pi)
NameError: name 'pi' is not defined
```

Although math library has been called, we need to specify the part we want to use with the name of the module

```
>>>print(math.pi)
3.141592653589793
```

But the best way is to import it as a module from the package:

```
>>> from math import pi
>>>print(pi)
3.141592653589793
```

##no need to import a math package containing many other functions.

```
>>> from math import pi, sqrt
>>> print (sqrt(4))
 Output 2.0
```

If we want to import everything, we use "*" >>> from the math import

Doing our module

Modules are stored as a file with a '.py' extension. The name of the module and the filename are the same. Like any '.py file,' a module contains Python

code. We will save the file as a python file in the same folder as the Python file from which we will import it or call it.

For example, we will create a check prime() function and use it in another Python script. First of all, the following code will be saved as prime.py:

```
"def checkIfPrime (numberToCheck):
    for x in range(2, numberToCheck):
        if (numberToCheck%x == 0):
            return False
        return True"
```

Now we create useCheckIfPrime.py, the other file, and save it in the same folder. It is here we import the module from another file.

```
Code:
"import prime
answer = prime. check prime(13)
          print (answer)"
```

When we run useCheckIfPrime.py, the prime.py is called upon, and we get the output as True".

Check Your Progress II

Note: a) Space is given below for writing your answers.

b) Compare your answers with the one given at the end of this Unit.

1) Write a program to find the given number is even or odd using function.

.....

.....

2.8 INTERACTIVE PROGRAMMING

For interacting programming, the user is asked to provide input. The program, according to the functioning of its code, will work on that code and provide the necessary output. Let's take, for example, the "Hello World" program we worked on earlier. Instead of just saying hello, we will say hello to whomsoever the user wants.

Now, let's save the following program and then run it.

```
Name=input("please enter name")
Print('hello"' Name.)
```

After running, the program will prompt us to our name.

```
Please enter your name:
```

Suppose we entered Mr. X. After pressing enter, we will get the following output:

```
Hello Mr. X
```

Now take a look at another example where we find the simple interest by inputting the amount, rate of interest, and number of years by the user itself:

We will be promoted to give input:

```
Please enter the amount: 1000
Please enter rate: 7
Please enter the number of years: 5

Output:
S.I. = 350.0
```

".

2.9 CONTROL FLOW, LOOPS

Now let's take another aspect of programming where after making a program interactive, we make the programmer smatter also. This means the programs are given the capability to make decisions and choices under certain limits set by the coder itself

To make this happen, we must understand the control flow concept. As the name suggests, they tend to control the flow of the program by invoking various if, if-else statements, loops, and iterations.
These all are a part of conditional statements.

Let's take a look at these conditional statements separately first. And then, we try to generate a program that is both interactive and smatter.

Conditional Statements

Every tool that operates under the preview of control flow entails some decision-making, which evaluates conditional statements and, in most cases, fulfills at least one of them. This fulfillment of any of the conditions changes the direction in which the program will proceed further.

One of the most common ones is the "comparison statement."

Let's say we want to compare two variables to see if they are identical. Then by using the == sign (double =; used as Equals to). We will reach them; if they are equal, the condition has been met, and the statements stand true. Else the statement is said to be false.

For illustration, if we type x == y, then the program is being enquired to examine if both the values of x and y are equal or not. And if equal, the condition has been fulfilled, and the statement is assessed as True. Else, the statement is false.

Let's go through some of them briefly:

Boolean Values and Operators: It is an expression that is either true or false. Following instances, use the operator ==, that will compare two operands and gives the output as True if they are equal or otherwise false:

```
>>>>51==51
True
>>>>15=60
False
```

Both True and False are special values which are not strings and

belong to the type bool".

If Statement

Suppose a statement consists of a logical expression. It considers the data compared to other specified data, and then a decision is made depending upon the result of the comparison. The decision of what to do next is also specified in the program.

"if expression

Depending upon the evaluation of the expression, which if TRUE, statements written inside the 'if 'statement is to be executed. But if FALSE, then code after the if statement is to be completed.

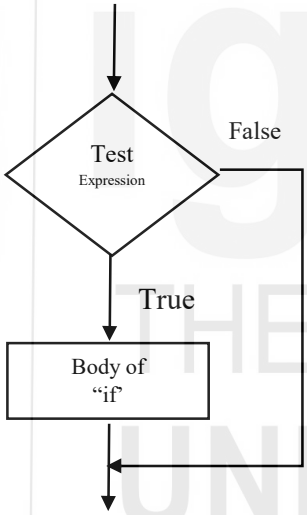

Figure 2.7: Flow Diagram of an expression

Example:

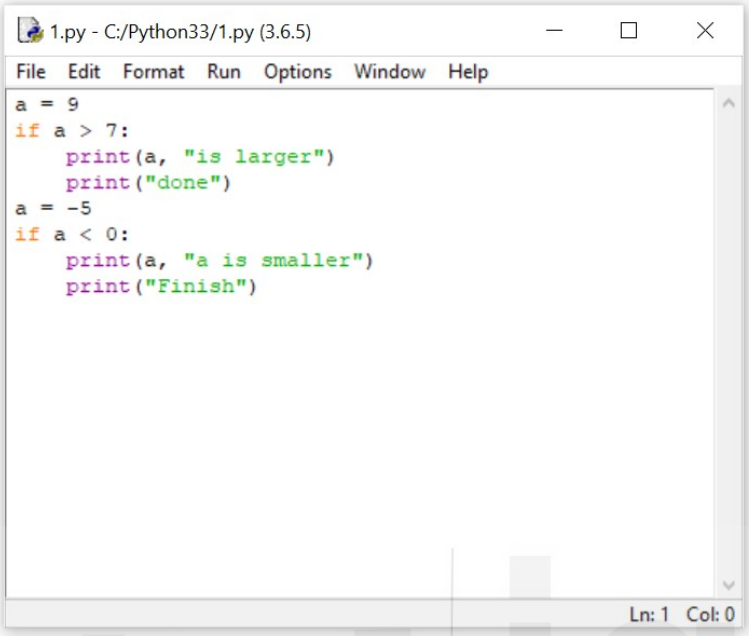

Figure 2.8: Example of if statement

Output:

```
9 is larger
Done
-5 a is smaller
Finish”
```

if - else expression

When we use an if statement, there is no room for decision-making if the statement turns out to be false. But when an else statement is merged with an if statement, we can direct what action/set of actions are to be executed when the condition fails.

But if the conditional expression in the if statement turns out false, then the else statement will be executed.

The else statement is deemed an elective statement, and we have mostly a single else Statement succeeding the if statement.

Syntax of if - else:

```
"if →test expression:
Body of if stmt.
else:
Body of else stmt.
```

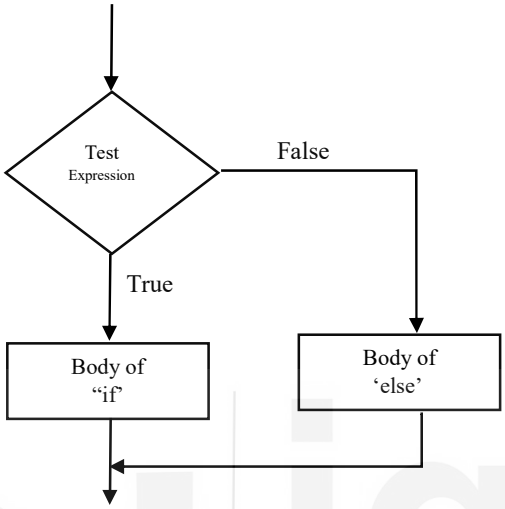

Figure 2.9: Flow Diagram of if-else expression

Example:

```
x=int(input('enter the No.'))
if x>9:
    print("x is greater")
else:
    print("x is smaller than the input given")
```

Output:

```
2
x is smaller than the input given"
```

if – elif - else expression

If we want to check that more than one statement or condition is true, then we use the 'elif' statement. This permits us to test multiple expressions/statements, and as soon as even a single condition turns out to be true, the corresponding block of code or commands is executed. Like, the else command, the elif statement is also optional.

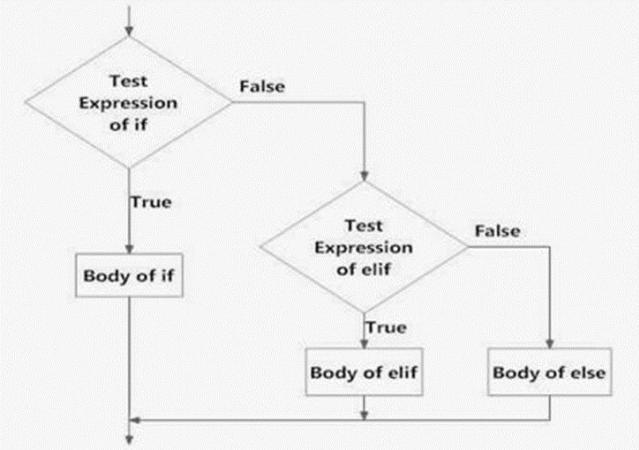

Figure 2.10: Flow Diagram of if – elif - else expression

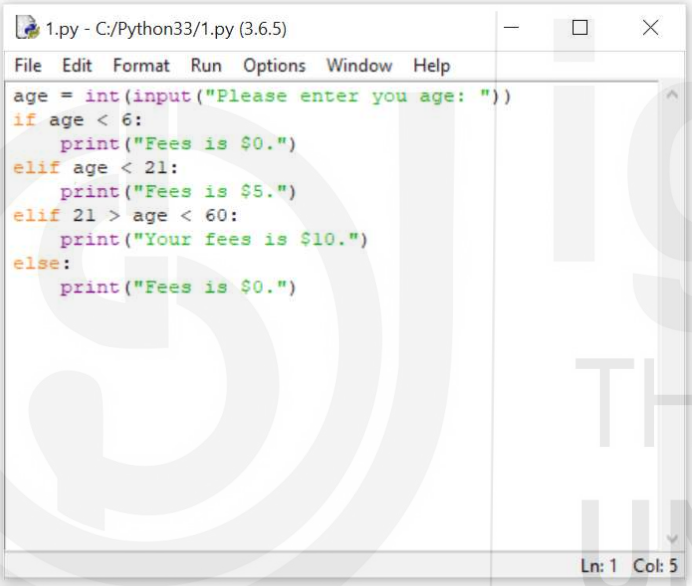

Figure 2.11: if - elif – else

Syntax If test expression:

```
"if stmts
elif test expression:
 elif stmts
else:
else stmts
```

output

```
Please enter your age: 12
The fee is RS. 200."
```

Iteration

A loop statement works on iterable. An iterable denotes whatever can be looped repeatedly, like a tuple, string, list, etc. The loop is executed on a statement or group of statements numerous times until and unless the requirement turns out to be true. Repetitive execution of a set of statements with loop support is called iteration. Once the condition is false, the program comes out of the loop and executes the program's next set of codes (if any).

A basic example can be calculating the respective ages of a class of 50 students through their date of birth, where the date of birth has been stored in a dictionary of a separate list. There are three types: (i) For Loops, (2) While Loop, and (3) Nested for Loop.

For loop:

The **for loop** works on the list items, dictionaries, strings tuples, etc. A set of statements to be executed is passed inside the circle. These statements are then performed for the desired number of times, which id to be fulfilled by some given condition.

In other words, looping is done on an iterable as shown below:

```
for x in iterable:
print (x)
```

```
"”numbers = [1,2,3,4,5,6]
seq=0
for va in numbers:
seq=va*va
print(seq)
```

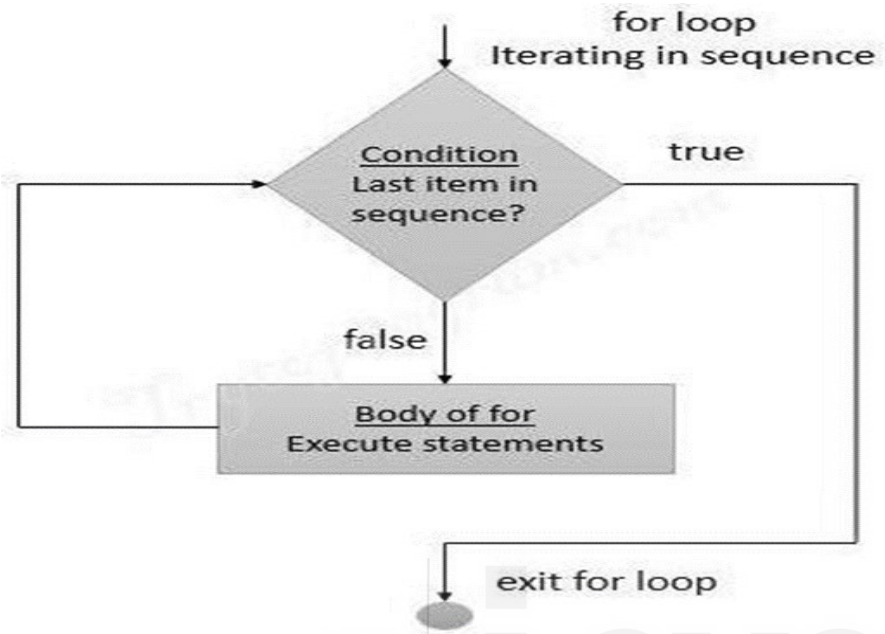

Figure 2.13: Flow Diagram of for loop

Output:

```
1
4
9
16″
```

range()

This function is used if we want to produce a sequence of numbers.

Python range() function for loop is commonly used to iterate sequence type (**Python range() List**, string, etc.) with for and while loop.

range(stop) stop, step)

Iterating over a list:

Example:

```
color = ['red', 'blue', 'green', 'yellow']
 for mycolor in color:
print (mycolor)
```

Output:

```
red
blue
green
yellow
```

Here we have declared a list of colors, with its input, and given it the members 'red', 'blue,' 'yellow,' and 'green.' Next, when we use the for command as **my color in color**, the loop will run through the whole list, and for every member, it is assigned to the variable **mycolor.**

The last statement: print (mycolor), then will print the first member of the list: **red.** Again the programs loop for the second time, and the same as the first time statement: **for mycolor in color,** the second item in the 'color' list will be assigned to 'mycolor' and printed again by the print (mycolor) command".

This loop continues till the whole of the list here has not been traversed, or any condition given by the user is not fulfilled.

"Iterating over a Tuple:

```
"tuple = (2,4,6,8)
print (First four even numbers ')
#Iterating over the tuple
for a in tuple:
    print (a)
```

Output:

```
2
4
6
8"
```

Iterating over a dictionary:

```
"college = {"a":"1","b":"2","c":"3"}
  #creating a dictionary

for keys in college:
    print (keys)

for blocks in college.values():
    print(blocks)
```

Output:

```
a
b
c
1
2
3"
```

Iterating over a String

```
Dictionary = 'abcdxyz'
for lettr in Dictionary:
    print (lettr)
```

Output:

```
a
b
c
d
x
y
z
```

While loop

These Loops are either conditional or infinite. A particular chunk of code will keep repeating until a specific condition is met. While the loop contains the true or false Boolean expression, and until it is negated to a false, this same set of statements in the loop will keep on iterating itself.

There can be one or multiple statements that need to be executed inside the loop.

Syntax:
while(expression):
Statement(s)

```
c=1
total = 1
a = int(input("Enter a number:"))
while c <=a:
total = total * C
c=c+1
print("the total is", total)
```

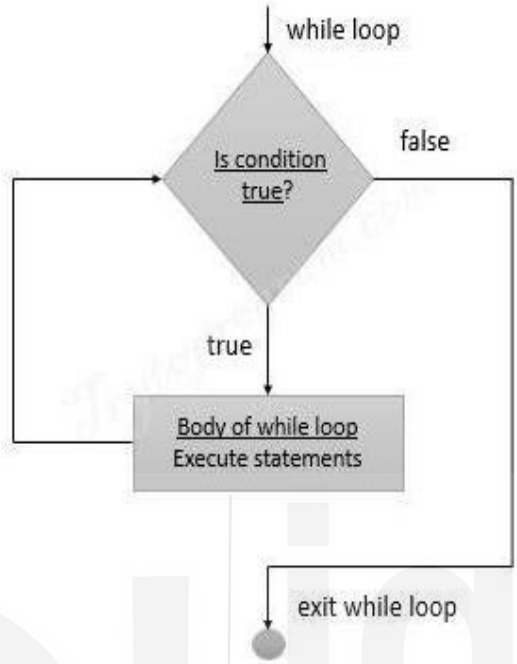

Figure 2.12: Flow Diagram of While loop

output:

```
Enter a number:4
the total is 1
the total is 2
the total is 6
the total is 24
```

Nested For loop

Sometimes we have to define one Loop within another Loop. First, we enter the main loop then the nested loop is executed till its conditions are fulfilled. The nested loop then exits to the outer(initial) loop, executed until its conditions are fulfilled.

Syntax:

```
for value in seq:
for value in seq:
```

Example:

```
"for a in range(1,9):
    for b in range(0,a):
        print(a, end=" ")
print('')
```

```
output
    1 2 2 3 3 3 4 4 4 4 5 5 5 5 5 6 6 6 6 6 6 7 7 7
    7 7 7 7 8 8 8 8 8 8 8 8″
```

Break and Continue Statement

Any loop command is to iterate the code as often as required, but sometimes, the order of a standard loop must be altered. This alteration is induced by applying some conditions.

Let's suppose we want to find a particular number in a list number, so while scanning through the list for a specific number, we will use the break and continue command so that when that number is found, the loop can exit, or we can continue further.

The break command can be operated in both the **while** and **for** loops with their respective method of executing the break command.

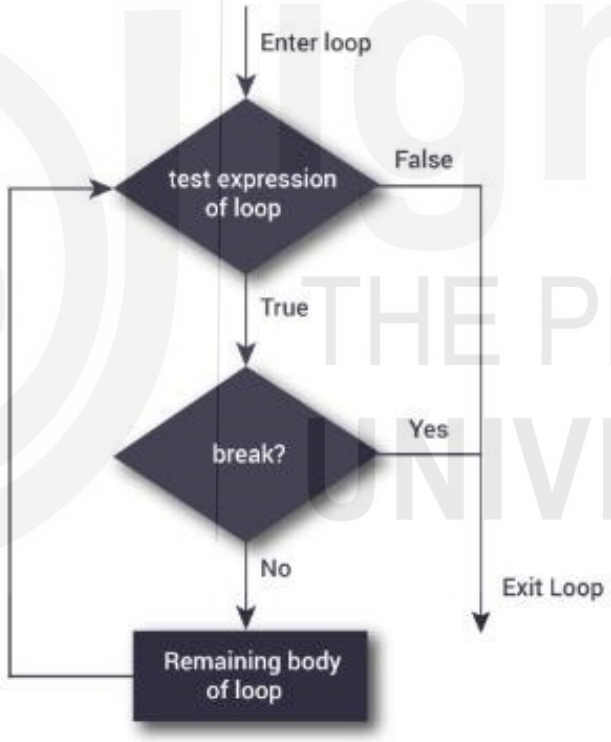

Figure 2.13: Flow Diagram of break loop

For test expression:

"for `var in sequence:`
`If condition:`
`continue (condition satisfied → jump outside the loop)`
`# code inside loop`
`# code outside loop`

while test expression:

```
# code inside while loop
If condition:
continue(break condition satisfied → jump
outside the loop)
# code inside while loop."
```

Continue Statement

When we use the break command, if the condition gets fulfilled and the break is thus invoked, the loop gets terminated. But if we want the loop to continue further with the next iteration, we use the continue command. This command will omit the rest of the code inside the very loop.

For example, the break command is adequate when we want to find a single number in the list. But if we find multiple numbers or strings in a given list, tuple, or dictionary, then we also use the continue command".

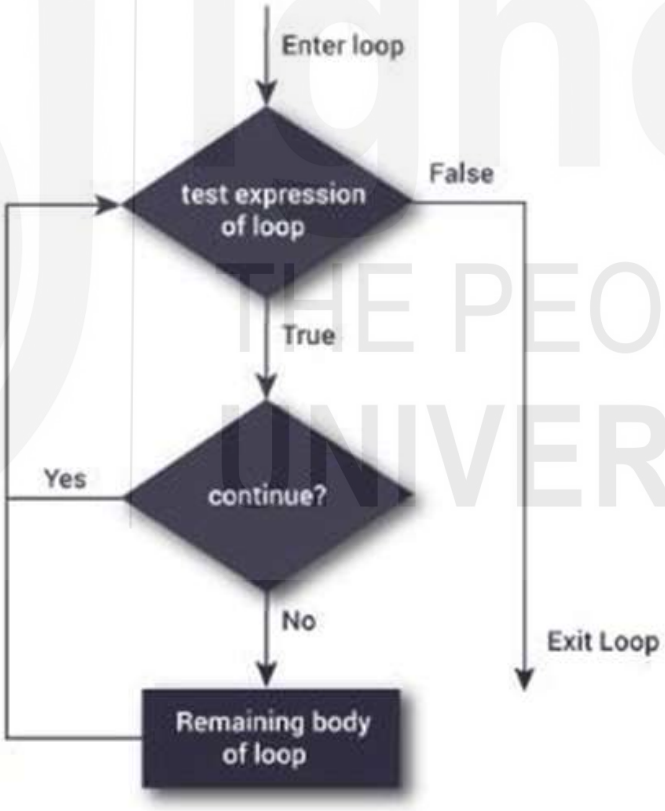

Figure 2.14: Flow Diagram of continue loop

"for var in sequence:

```
If condition:
continue (condition satisfied → jump outside
the loop)
# code inside loop
# code outside loop
```

while test expression:

```
# code inside while loop
If condition:
continue(break condition satisfied  →  jump
outside the loop)
# code inside while loop."
```

Check Your Progress III

Note: a) Space is given below for writing your answers.

b) Compare your answers with the one given at the end of this Unit.

(i) Write a program for determining the largest number of the given three numbers (If Statement)

……

……

(ii) Write down a program to display numbers before a certain number:

……

……

2.10 LET US SUM UP

In this Unit, we have gone through the basics of python programming. At present, python has mainly used language due to its features. Pros and cons for python with other languages are also discussed. You have seen that here python will invoke by three types: shell, editor, and script mode. We also have looked at here syntax and semantics of python fundamentals. Finally, we have functions, modules, and control flow like if, while loop statements are a significant part of any programming language for making the language interactive.

2.11 CHECK YOUR PROGRESS: THE KEY

1) (i) Data Types

Python consists of various standard data types used for defining operations:

-Integer: Either positive or negative is a whole number, short of decimals and unlimited length.

-Float A positive or negative number containing one or more decimals.

-Boolean: Objects in Boolean can either be True or False

-String: we refer to strings as a connecting set of characters denoted in quotation marks.

List: It denotes a collection of data, generally related to each other. We need to store the height of 5 users. We used a list for the same

Tuple: Also, like lists, values cannot be changed.

Dictionary: Collection of related data pairs. The values stored are accessible by key.

(ii) Variables: Variables store values for which memory locations are explicitly reserved and separately.

(iii) **Expression:** It is a grouping of variables, values, and operators. They are assessed via the assignment operator.

(iv) **Statements: These are those** instructions that the Python interpreter can execute. We have two basic statements: the "assignment" and the "print" statements.

(v) Comments: Single-line comments begin with a hash (#) symbol. A **Multi-line comment is** helpful when we need to comment on many lines.

II

```
def odd-even(num)
if num%2 == 0:
print("number is even)
else:
print("number is odd”)
#calling of function :
Find=oddeven(15)

number is odd
```

III

(i)

```
“a=int(input('enter the num'))
b=int(input('enter the num'))
c=int(input('enter the num'))
if a>b:
print("a is greater")
elif b>c:
print("b is greater")
else:
print("c is greater")
```

Output:

```
enter the num 5
enter the num 2
enter the num 9
a is greater”
```

(ii)

```
“for num in [44,45,89,36,17,7,86]:
   print(num)
   if(num==17):
       print("number found")
       print("Dismissing the loop")
       break
```

Output:

```
44
45
89
36”
```