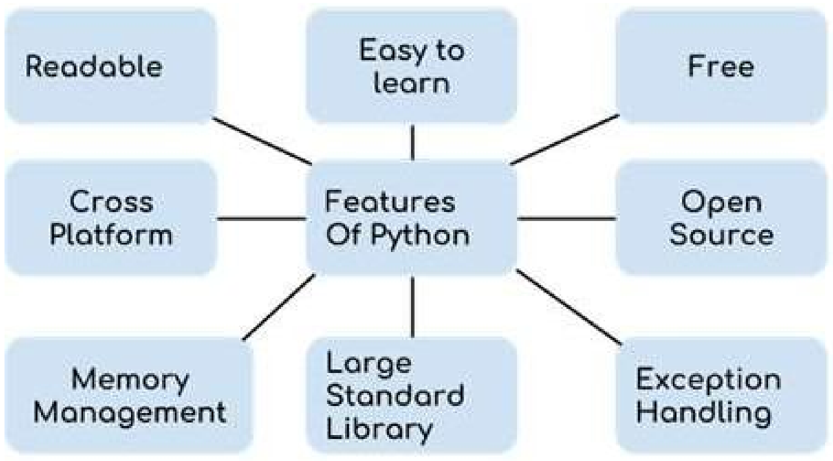
Introduction: Introduction to Python, Program Development Cycle, Input ,Processing, and Output, Displaying Output with the Print Function, Comments, Variables, Reading Input from the Keyboard, Performing Calculations, Operators. Type conversions, Expressions, More about Data Output. Data Types and Expression: Strings Assignment, and Comment, Numeric Data Types and Character Sets, Using functions and Modules. Decision Structures and Boolean Logic: if, if-else, if-elif-else Statements, Nested Decision Structures, Comparing Strings, Logical Operators,Boolean Variables. Repetition Structures: Introduction, while loop, for loop, Calculating a Running Total, Input Validation Loops, Nested Loops.

Introduction to Python:

- Python is a widely used general-purpose, high level object oriented programming language.
- It was developed by **Guido van Rossum** in the year 1989 and was released in 1991.
- It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code.
- Python is a programming language that lets you work quickly and integrate systems more efficiently.
- There are two major Python versions: Python 2 and Python 3.Both are quite different.

Features of Python:

1. Simple and easy to learn.
2. Portable (or) Cross Platform
3. Free and Open source.
4. Large Standard library
5. Both procedure oriented and object oriented.
6. Interpreted programming language.
7. Memory Management
8. Extensible.
9. Embedded.
10. Supports exception Handling.



1. **Simple and Easy to Learn:** Learning python is easy as this is an expressive and high level programming language, which means it is easy to understand the language and thus easy to learn.

2. **Portable (or) Cross platform:** Python is available and can run on various operating systems such as Mac, Windows, Linux, Unix etc .This makes it across platform and portable language. It enables programmers to develop the software for several competing platforms by writing the program only once.

3. **Free and Open Source:** Python is a open source programming language. Python is free to download and use. This means you can download it for free and use it in your application. Python is an example of a FLOSS (Free/Libre Open Source Software), which means you can freely distribute copies of this software, read its source code and modify it.

4. **Large standard library:** Python comes with a large standard library that has some handy codes and functions which we can use while writing code in Python.

5. **Both procedure-Oriented and Object-Oriented language:** Python supports procedure-oriented programming as well as object oriented programming. In Procedure-oriented languages, the program is built around procedures or functions which are nothing but reusable pieces of programs. In object oriented languages, the program is built around objects which combine data and functionality.

6. **Interpreted:** Python is an interpreted language; i.e program written in python does not need compilation to binary. You can just execute the program directly from the source code. Internally python converts the source code into an intermediate form called byte code and then translates this is not the native language of your computer and then runs it.

7. **Automatic memory management:** Python supports automatic memory management which means the memory is cleared and freed automatically. You do not have to bother clearing the memory and you don’t need to assign the data type of the variable, when you assign the value to the variable, it automatically allocates the memory to the variable at runtime.

8. **Extensible:** Python is extensible, that some part of the program written in c, c++ or java can use them in your python program. Some of the greatest open source library for numeric computing like Tensor flow which is written and used by Google is made in this format.

9. **Embeddable:** You can embed python within your c, c++ programs to give ‘scripting’ capabilities for your program’s users.

10. **Supports exception handling:** If you are new, you may wonder what is an exception? An exception is an event that can occur during program exception and can disrupt the normal flow of program. Python supports exception handling which means we can write less error prone code and can test various scenarios that can cause an exception later on.

Uses of Python:

- Google makes extensive use of python in its web search systems.
- Python is used for developing desktop applications and web applications too.
- Python is widely used in Game development.
- It is popularly used in the field of data science, Machine Learning and Artificial Intelligence to analyze data, build predictive models and make business decisions.

Applications:

1. **Web development** – Web framework like Django and Flask are based on Python. They help you write server side code which helps you manage database, write backend programming logic, mapping urls etc.

2. **Machine learning** – There are many machine learning applications written in Python. Machine learning is a way to write logic so that a machine can learn and solve a particular problem on its own. For example, products recommendation in websites likes Amazon, Flip kart, eBay etc .is a machine learning algorithm that recognizes user's interest. Face recognition and Voice recognition in your phone is another example of machine learning.

3. **Data Analysis** – Data analysis and data visualization in form of charts can also be develop educing Python.

4. S**cripting** – Scripting is writing small programs to automate simple tasks such as sending automated response emails etc. Such type of applications can also be written in Python programming language.

5. **Game development**– You can develop game using Python.

6. You can **develop embedded applications** in Python.

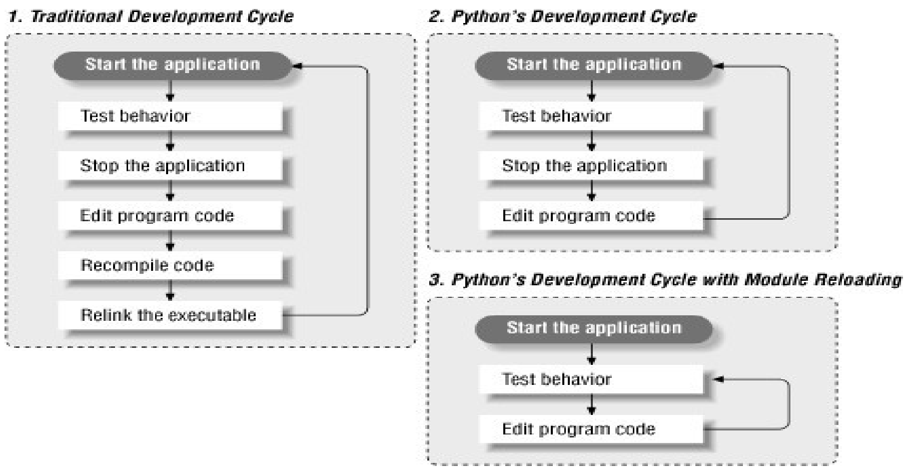
7. **Desktop applications** – You can develop desktop application in Python using library like TKinter or QT.

Organizations using Python:

- Google(Components of Google spider and Search Engine)
- Yahoo(Maps)
- YouTube
- Mozilla
- Drop box
- Microsoft
- Cisco
- Spotify

Program Development Life cycle

- Program Development Life Cycle (PDLC) is a systematic way of developing quality software. It provides an organized plan for breaking down the task of program development into manageable chunks, each of which must be successfully completed before moving onto the next phase.
- Python's development cycle is dramatically shorter than that of traditional tools as shown in figure



- In Python, there are no compile or link steps -- Python programs simply import modules at run time and use the objects they contain. Because of this, Python programs run immediately after changes are made and in cases where **dynamic** module reloading can be used, it's evenpossibletochangeandreloadpartsofarunningprogramwithoutstoppingitatall.

Phases of program Development Cycle

- **Problem Definition:** Here the formal definition of the problem is stated. This stage gives thorough understanding of the problem, and all the related factors such as input, output, processing requirements and memory requirements etc.,
- **Program Design:** Once the problem and its requirements have been identified, then the design of the program will be carried out with algorithms and flowcharts. A

  An **<u>Algorithm</u>** is a step-by-step process to be followed to solve the problem. It is formally written using the English language

  The **<u>Flowchart</u>** is a visual representation of the steps mentioned in the algorithm. This flowchart has some set of symbols that are connected to perform the intended task. The symbols such as square, diamond, lines and circles etc., are used.

Example:

Algorithm for adding two numbers:

Step 1: Start

Step 2: Declare variables num1, num2 and sum.

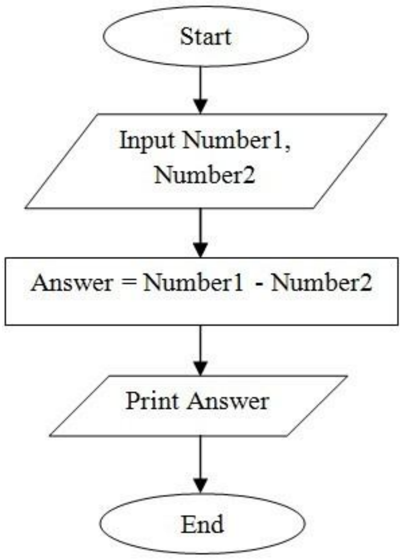
Step 3: Read values for num1, num2.

Step 4: Add num1 and num2 and assign the result to a variable sum.

Step 5: Display sum

Step 6: Stop

Flow chat



❖ **Coding:** Once the design is completed, the program is written using any programming language such as c, c++, python and java etc. The coding usually takes very less time, and syntax rules of the language are used to write the program.

<u># adding two integer numbers in python</u>

```
a=int(input('enter  a'))
b=int(input('enter  b'))
c=a+b
print('The sum is',c)
```

- **Debugging:** At this stage the errors such as syntax errors in the programs are detected are corrected. This stage of program development is an important process. Debugging is also known as program validation.

- **Testing:** The program is tested on a number of suitable test cases. The most insignificant and the most special cases should be identified and tested.

- **Documentation:** Documentation is a very essential step in the program development. Documentation helps the users and the who actually maintain the software.

- **Maintenance:** Even after the software is completed, it needs to be maintained and evaluates regularly. In software maintenance, the programming team fixes the program errors and updates the software when new features are introduced.

Variables

A **variable is a reserved memory area (memory address) to store value**. For example, we want to store an employee's salary. In such a case, we can create a variable and store salary using it. Using that variable name, you can read or modify the salary amount.

In other words, a variable is a value that varies according to the condition or input pass to the program. Everything in Python is treated as an object so every variable is nothing but an object in Python.

A variable can be either **mutable** or **immutable**. If the variable's value can change, the object is called mutable, while if the value cannot change, the object is called immutable.

→Rules for creating variables in python:

A name in a Python program is called an identifier. An identifier can be a variable name, class name, function name, and module name.

- A Variable name must start with a letter or the underscore character.
- A variable name cannot start with a number
- A variable may can only contain alpha-numeric characters and underscores(A-Z,a-z,0-9 and _)
- Variable names are case –sensitive (name, Name and NAME are three different variables)
- The reserved words (Keywords) cannot be used for naming the variable.

→Creating a Variable and Assigning Values to Variable

- Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.
- The operand to the left of the = operator is the name of the variable and the operand to the right of the= operator is the values to red in the variable.

Syntax:
<Variable_name>=<Variable_value>

For example :

```
counter=100         #An integer assignment
miles=1000.0        #A floating point
name                ="John" #A string
print(counter)
print(miles)
print(name)
```

Output:

```
100
1000.0
‘John’
```

→Changing the value of a variable

Many programming languages are statically typed languages where the variable is initially declared with a specific type, and during its lifetime, it must always have that type.
But in Python, variables are **dynamically typed** and not subject to the **data type** restriction. A variable may be assigned to a value of **one type**, and then later, we can also re-assign a value of a **different type**.

→Get the data type of variable

No matter what is stored in a variable (object), a variable can be any type like int, float, str, list, tuple, dict, etc. There is a built-in function called type() to get the data type of any variable.
The type() function has a simple and straight forward syntax.

Syntax of type() **:**

type(<Variable_name>)

```
var = 10
print(var)  # 10
# print its type
print(type(var))  # <class 'int'>
# assign different integer value to var
var = 55
print(var)  # 55
# change var to string
var = "Now I'm a string"
print(var) # Now I'm a string
# print its type
print(type(var))  # <class 'str'>
# change var to float
var = 35.69
print(var)  # 35.69
# print its type
print(type(var))  # <class 'float'>
```

→Multiple Assignments:

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

For example: a= b =c = 1

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

For example: a, b, c =1, 2,"john"

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

→ Variable scope

Scope: The scope of a variable refers to the places where we can access a variable.
Depending on the scope, the variable can categorize into two types **local variable** and the **global variable.**

Local variable

A local variable is a variable that is accessible inside a block of code only where it is declared. That means, if we declare a variable inside a method, the scope of the local variable is limited to the method only. So it is not accessible from outside of the method. If we try to access it, we will get an error.

```
def test1(): # defining 1st function
    price = 900 # local variable
    print("Value of price in test1 function :", price)

def test2(): # defining 2nd function
    # NameError: name 'price' is not defined
    print("Value price in test2 function:", price)

# call functions
test1()
test2()
```

In the above example, we created a function with the name test1. Inside it, we created a local variable price. Similarly, we created another function with the name test2 and tried to access price, but we got an error "price is not defined" because its scope is limited to function test1 (). This error occurs because we cannot access the local variable from outside the code block.

Global variable

A Global variable is a variable that is defined outside of the method (block of code). That is accessible anywhere in the code file.

Example

```
price =900# Global variable
def test1(): # defining 1st function
    print("price in 1st function :", price) # 900
def test2(): # defining 2nd function
    print("price in 2nd function :", price) # 900

# call functions
test1()
test2()
```

In the above example, we created a global variable price and tried to access it in test1 and test2. In return, we got the same value because the global variable is accessible in the entire file.

Note: You must declare the global variable outside function.

Delete a variable

Use the **del** keyword to delete the variable. Once we delete the variable, it will not be longer accessible and eligible for the garbage collector.

Example

```
var1 = 100
print(var1) # 100
```

Now, let’s delete var1 and try to access it again.

```
var1 = 100
del var1
print(var1)
```

Output:

```
Name Error: name 'var1' is not defined
```

Python Comments

Comments are descriptions that help programmers better understand the intent and functionality of the program. It is completely ignored by the Python interpreter.

Single-Line Comments in Python: In Python, we use the hash symbol # to write a single-line comment.

Example1: Writing Single-Line Comments

```
# printing a string
print ('Hello world')
```

Output: Hello world

Here, the comment is:

#printing a string.
This line is ignored by the Python interpreter. Everything that comes after isignored

#.So, we can also write the above program in a single line as:

```
print('Hello world') #printing a string
```

The output of this program will be the same as in Example 1.

The interpreter ignores all the text after #.

Multi-Line Comments in Python: We can use # at the beginning of each line of comment on multiple lines.

Example2:

```
# Using multiple
# it is a multiline comment
```

Here, each line is treated as a single comment and all of them are ignored.

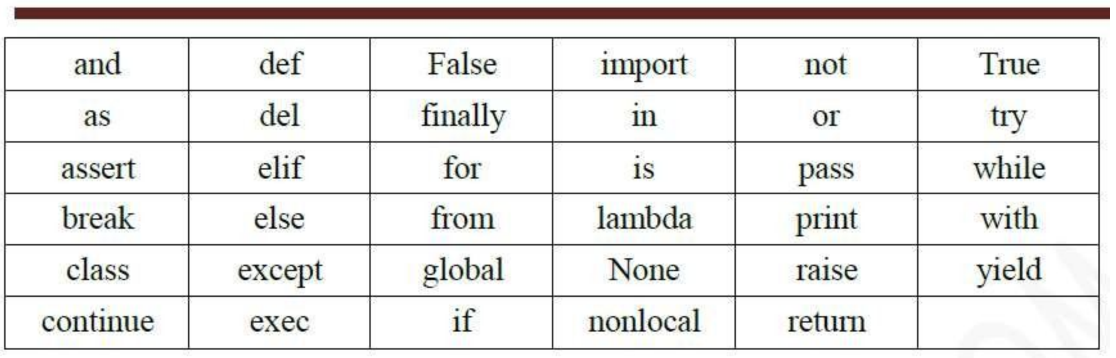
In a similar way, we can use multiline strings (triple quotes) to write multiline comments as shown below. The quotation character can either be ' or ".

```
'''
I am a
multiline comment! '''
print ("Hello World")
```

Keywords

Keywords are predefined, reserved words that are used in programming and they have a special meaning. Keywords are part of the syntax and they cannot be used as an identifier.
Identifier refers to name given to entities such as variables, functions, classes, lists, etc. The identifier must be unique.

Python has 33 Keywords:



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

If you want to print these keywords in the python interpreter, type the following code as

```
import keyword
print (keyword. kwlist)
```

<u>Input, Processing, and Output</u>

❖ Most useful programs accept inputs from some source, process these inputs, and then final l you put results to some destination.

❖ In terminal- based interactive programs, the input source is the keyboard, and the output destination is the terminal display.

❖ The Python shell itself takes inputs as Python expressions or statements. Its processing evaluates these items. Its outputs are the results displayed in the shell.

❖ Python provides numerous built-in functions that are readily available to us at the Python prompt.

❖ Some of the functions like input()and print() are widely used for standard input and output Operations respectively.

Let us see the output section first.: Python Output Using print() function

- ❖ We use the print () function to output data to the standard output device (screen). We can also output data to a file

print() function: The print() function prints the given object to the standard output device (screen) or to the text stream file.

Syntax of print()is:

```
print(*objects, sep='' ,end='\n' ,file=sys .stdout ,flush=False)
```

- **objects**-object to be printed .***** indicates that there may be more than one object
- **sep**-objects are separated by sep .**Default value**:''
- **end**- end is printed at last by **default it has\n**
- **file**- must be an object with write (string) method. If omitted it, sys.stdout will be used which prints objects on thescreen.
- **flush**-A Boolean, specifying if the output is flushed(True) or buffered(False). Default: False

Example1:

```
print( 'This sentence is output to the screen')
```

Output: This sentence is output to the screen

Example2:

```
a=5
print('The value of a is' ,a)
```

Output: The value of a is 5

Example3:

```
print("Python is fun.")
a=5
#Two objects are passed
print("a=",a)
b=a
#Three objects are passed
print('a=',a,'=b')
```

Output:

```
Python is fun.
a=5
a=5=b
```

Case1:print with out any arguments

print(‘SRKR Engineering College’)
print()#we have not passed any argument by default it takes new line and create one line blank space
print(‘Information Technology)

Output:

```
SRKR Engineering College

Information Technology
```

Case2: print function with string operations(with arguments)

1.Print(‘hello world’) **Output:** hello world

2.print(‘hello\n world’) **Output:** hello

```
hello
 world
```

3.print(‘hello\t world’) **Output :**hello world

4. #string concatenation both objects are string only

print(‘it’+‘cse’) **Output:** itcse

print(‘it’,‘ cse’) **Output:** it cse

5**.#repeat string into number of times**

print(5*’it’) **Output:** itititititit

print(5*'it\n') **Output**: it

```
it
it
it
it
it
```

print(5*'it\t') **Output:** it it it it it

Case3: print function with any number of arguments

1. (a)print(‘values are:’,10,20,30) **Output:** values are:10 20 30

(b)a ,b, c=10,20,30
print(‘values are: ’,a, b, c)
Output: values are:10 20 30

2. **Print function with“ sep” attribute: This is used to separate objects**

   By default ,value of sep is empty space(sep=’’)

   print ('values are:',10,20,30,sep=":")

   Output: values are::10:20:30

   print ('values are:',10,20,30,sep="--")

   Output: values are:--10--20--30

Case4: print statement with end attribute

- The **end** key of print function will set the string that needs to be appended when printing is done.
- By default, the **end** key is set by newline character (by default, attribute end =’\n’ in print function). So after finishing printing all the variables, a newline character is appended. Hence, we get the output of each print statement in different line. But we will now overwrite the new line character by any character at the end of the print statement.

Example-1:

- print('Aditya') #in this statement by default end=’\n ’so it takes new line
- print('raja') #in this statement by default end=’\n ’so it takes newline
- print('devi') #in this statement by default end=’\n’ so it takes newline

Output:

```
Aditya
Raja
devi
```

Note: when you observe output 1st print statement prints output Aditya and immediately takes newline character and execute 2nd print statement and followed.

Example-2:

- print('Aditya’ ,end='$')
- print('raja', end='*')
- print('devi')

Output: Aditya $raja*devi

Case5: print function with sep and end attribute Example

- print(19,20,30,sep=':',end='$$$')
- print(40,50,sep=':')
- print(70,80,sep=':',end='&&&')
- print(90,100)

Output:19:20:30$$$40:50

70:80&&&90100

<u>Python |Output Formatting</u>

- There are several ways to present the output of a program, data can be printed in a human-readable form, or written to a file for future use. Sometimes user often wants more control the formatting of output than simply printing space-separated values. There are several ways to format output.

1. **Print function with replacement operator {}**or **format function**

str. format() is one of the *string formatting methods* in Python3, which allows multiple substitutions and value formatting. This method lets us **concatenate elements within a string.**

Syntax: {}.format (value)

→ The format() method formats the specified values and insert them inside the string's placeholder. The format() method returns the formatted string.

➔ The placeholder is defined using curly bracket { }. The placeholder can be identified using names indexes{ prices}, numbered indexes{0}, or even empty placeholders{ }

Example:1(using empty placeholders)
Name= 'John'
Salary='1000'
print('hello my name is{}and my salary is{}'.format(Name, Salary))

Output: hello my name is John and my salary is1000

Example:2
name='John'
salary=1000
print('hello my name is"{}"and my salary is"{}"'.format(name ,salary))

Output: : hello my name is" John" and my salary is"1000"

***********we can also use index values and print the output***********

Example:3

name='John'

salary=1000

print('hello my name is"{0}"and my salary is"{1}"'.format(name ,salary))

Output: hello my name is "John" and my salary is "1000"

Example: 4

print('hello my name is"{1}" and my salary is"{0}"'.format(name,salary))

Output :hello my name is "1000"andmy salary is "john"

******* ****we can also use variables in the reference operator***********

Example:5

print('hello my name is"{n}" and my salary is"{s}"‘.format(n=name,s=salary))

Output: hello my name is" john" and mysalaryis"1000"

Example:6

print('hello my name is"{n}"and my salary is{ s}'.format(s=salary, n=name))

Output: hello my name is" john" and mysalaryis"1000"

2. **Formatting output using String modulo operator(%):**

   Syntax : print(‘formatted string’%(variable list)

   The % operator can also be used for string formatting. String modulo operator (%) is still available in Python(3.x)and user is using it widely.

Example 1

(a) a=6

print(‘a value is=%i ’%a)

Output: a value is =6

(b)a=6;b=7;c=8

print('a value is=% i and b=%f and c=%i'%(a ,b ,c))

Output: a value is=6andb=7.000000and c=8

3. Using f string

Formatted strings or f-strings were introduced in python 3.6. A f-string is a string literal that is prefixed with “f”. These strings may contain replacement fields, which are expressions enclosed within curly braces { }. The expressions are replaced with their values.
An f at the beginning of the string tells python to allow any at the valid variable names within the string.

Example

```
 Name=”sai”
Age=20
print(f” Hello, my name is{name} and I’m {age} years old”)
```

<u>Reading Input from the Keyboard:</u> In python input() is used to read the input from the keyboard dynamically . By default ,the value of the input function will be stored as a string

Syntax: Variable name=input(‘prompt’)

Example:

```
name=input("Enter Employee Name")
salary=input("Enter salary")
company=input("Enter Company name")
print("\n")
print("Printing  Employee  Details")
print("Name", "Salary"," Company")
print(name, salary, company)
```

Output:

```
Enter Employee Name Jon
Enter salary12000
Enter Company name Google
Printing Employee Details
Name Salary Company
Jon12000Google
```

Accept a numeric input from User: To accept an integer value from a user in Python. We need to convert an input string value into an integer using a int () function.

Example: first _number=int (input("Enter first number"))
We need to convert an input string value into an integer using a float()function.
Example: first_ number=float(input("Enter first number"))

#program to calculate addition of two input numbers

```
First _number=int(input("Enter first number "))
second_ number=int (input("Enter second number"))
print("First Number:" ,first _number)
print("Second Number:",second_ number)
sum1 =first_ number+ second _number
print("Addition of two number is:",sum1)
```

Output:

```
Enter first number 20
Entersecondnumber40
FirstNumber:20
SecondNumber:40
Additionoftwonumberis:60
```

Get multiple input values from a user in one line: In Python, we can accept two or three values from the user in one input()call.

Taking Multiple inputs from the user using split() method in python

split() method: Is a library method in python, it accepts multiple inputs and brakes the given input based on the provided separator, if we don’t provide any separator the white space is considered as the separator.
Syntax: input().split(<separator>)

Example:
In a single execution of the input () function, we can ask the use his/her name, age, and phone number and store it in three different variables.

```
name, age, phone=input("Enter your name, Age, Percentage separated byspace").split()
print("\n")
print("User Details:",name, age, phone)
```

Output:

```
Enter your name Age, Percentage separated by space John 2675.50
User Details: John 26 75.50
```

<u>Performing Calculations</u>

Computers are great at math problems! How can we tell Python to solve a math problem for us? In this we use numbers in Python and the special symbols we use to tell it what kind of **calculation** to do.

Example-1:

```
1.print(2+2)
2.print("2"+"2")
```

Output:

```
4
22
```

Question: Why does line two give the wrong Answer?
Answer: When we do math in Python, we can’t use strings. We have to use numbers. The first line Uses two numbers. Both of them are **integers** (called int in Python).

Example-2:

Subtraction:

print(2 - 2)

Multiplication:

print(2*2)

Division:
print(2/2)

Question: Why did the last statement output 1.0?

Answer: When Python does division, it uses a different kind of number called a **float**. Floats always have a decimal point. Integers are always whole numbers and do not have decimal points.

Run: print(7/2)

Calculations in Python follow the Order of Operations, which is sometimes called **PEMDAS**.

Run:print((6-2)*5)
print(6-2*5)

Subtraction:
print(2 - 2)
Multiplication:
print(2*2)
Division:
print(2/2)

Example-3:

The first statement is **evaluated** by Python like this:

1.(6–2)*5 *Parentheses first*
2.4*5
3.20

The second statement is evaluated like this:

1.6–2*5 *Multiplication first*
2.6–10
3.-4

Example-4:

The **modulo** operator(%)finds the remainder of the first number divided by the second number.

Run:

```
print(12%10)
```

12/10=1with a remainder of 2.

Integer division (//)is like normal division, but it rounds down if there is a decimal point.

Run:

```
print(5//2)
```

5/2=2.5, which is rounded down to2

Exponentiation(**) raises the first number to the power of the second number. Run:

```
print(3**2)
```

This is the same as 3^2

Example-5:

Remember, input()**returns** a string, and we can't do math with strings. Fortunately, we can change Strings into ints like so:

```
two= "2"
two = int (two)
print(two+ two)
```

If you need to work with a decimal point, you can change it to a float instead:

```
two=float(two)
```

Example-6:

Activity1:

Do you know anyone who tends to one-up you in conversation? In this activity, we'll make a simple chatbot that asks a series of questions, explaining to you why it's superior after each one. The robot will have a variable level of one-upmanship.

We'll use print, input, and math operators and variables to accomplish this.

Example-7:

```
one_up_level=1
a1=input("How many seconds does it take you to run the100meterdash?")
a1= int(a1)
print("That's cool. I can do it in",a1-one_up_level,"seconds though. And I don't even have legs, sooooo…")

a2=input("But what about your GPA? I'm sure that's pretty good, eh?(Enter your GPA)")
a2=float(a2)
print("Alright.Minewas",a2+one_up_level)
print("Not that it matters, lol")
```

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

1. Python Arithmetic Operators.
2. Python Comparison/relational Operators
3. Python Logical Operators
4. Python Assignment Operators
5. Python Identity Operators
6. Python Membership Operators
7. Python Bitwise Operators

➔ Arithmetic operators

Arithmetic operators are the most commonly used. The Python programming language provides arithmetic operators that perform addition, subtraction, multiplication, and division. It works the same as basic mathematics.

There are seven arithmetic operators we can use to perform different mathematical operations, such as:

1. + (Addition)
2. - (Subtraction)
3. * (Multiplication)
4. / (Division)
5. // Floor division)
6. % (Modulus)
7. ** (Exponentiation)

Addition operator +

It adds two or more operands and gives their sum as a result. It works the same as a unary plus. In simple terms, it performs the addition of two or more than two values and gives their sum as a result.

Example

```
x = 10
y = 40
print(x + y)# Output 50
```

Also, we can use the addition operator with strings, and it will become string concatenation.

Example

```
name = "Kelly"
surname = "Ault"
print(surname + " " + name) # Output Ault Kelly
```

Subtraction –

Use to subtract the second value from the first value and gives the difference between them. It works the same as a unary minus. The subtraction operator is denoted by - symbol.

Example

```
x = 10
y = 40
print(y - x)# Output 30
```

Multiplication *

Multiply two operands. In simple terms, it is used to multiply two or more values and gives their product as a result. The multiplication operator is denoted by a * symbol.

Example

```
x = 2
 y = 4
 z = 5
print(x * y) # Output 8 (2*4)
print(x * y * z)# Output 40 (2*4*5)
```

You can also use the multiplication operator with string. When used with string, it works as a repetition.

Example

```
name = "Jessa"
print(name * 3) # Output Jessa Jessa Jessa
```

Division /

Divide the left operand (dividend) by the right one (divisor) and provide the result (quotient) in a float value. The division operator is denoted by a / symbol.

Note:

- The division operator performs floating-point arithmetic. Hence it always returns a float value.
- Don’t divide any number by zero. You will get a *Zero Division Error: Division by zero*

Example

```
x = 2
y = 4
z = 8
print(y / x) # Output 2.0
print(z / y / x)# Output 1.0
# print(z / 0) # error
```

Floor division //

Floor division returns the quotient (the result of division) in which the digits after the decimal point are removed. In simple terms, It is used to divide one value by a second value and gives a quotient as a round figure value to the next smallest whole value. It works the same as a division operator, except it returns a possible integer. The // symbol denotes a floor division operator.

Note:

- Floor division can perform both floating-point and integer arithmetic.
- If both operands are int type, then the result types. If at least one operand type, then the result is a float type.

Example

```
x = 2
y = 4
z = 2.2
# normal division
print(y / x)# Output 2.0
# floor division to get result as integer
print(y // x)# Output 2
# normal division
print(y / z) # 1.81
 # floor division.
 # Result as float because one argument is float
 print(y // z) # 1.0
```

Modulus %

The remainder of the division of left operand by the right. The modulus operator is denoted by a % symbol. In simple terms, the Modulus operator divides one value by a second and gives the remainder as a result.

Example

```
x = 15
y = 4
print(x % y)# Output 3
```

Exponent **

Using exponent operator left operand raised to the power of right. The exponentiation operator is denoted by a double asterisk ** symbol. You can use it as a shortcut to calculate the exponential value.

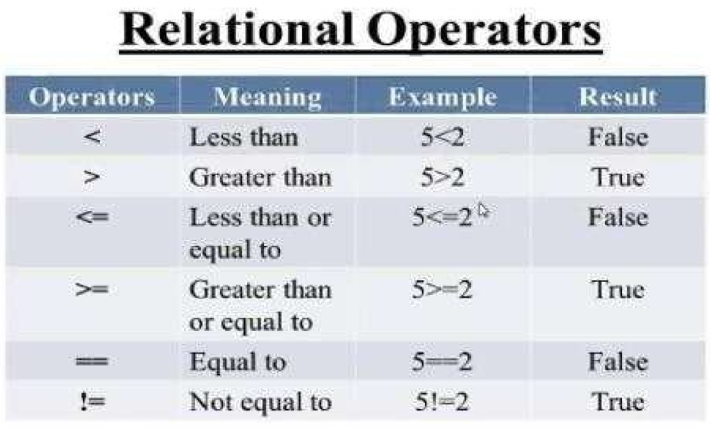
For example, 2**3 Here 2 is multiplied by itself 3 times, i.e., 2*2*2. Here the 2 is the base, and 3 is an exponent.

Example

```
num = 2
# 2*2
print(num ** 2)
# Output 4
# 2*2*2
print(num ** 3)
# Output 8
```

Relational (comparison) operators

Relational operators are also called comparison operators. It performs a comparison between two values. It returns a boolean True or False depending upon the result of the comparison.
Python has the following six relational operators.



Relational Operators

Operators	Meaning	Example	Result
<	Less than	5<2	False
>	Greater than	5>2	True
<=	Less than or equal to	5<=2	False
>=	Greater than or equal to	5>=2	True
==	Equal to	5==2	False
!=	Not equal to	5!=2	True

You can compare more than two values also. Assume variable x holds 10, variable y holds 5, and variable z holds 2.

So print(x > y > z) will return True because x is greater than y, and y is greater than z, so it makes x is greater than z.

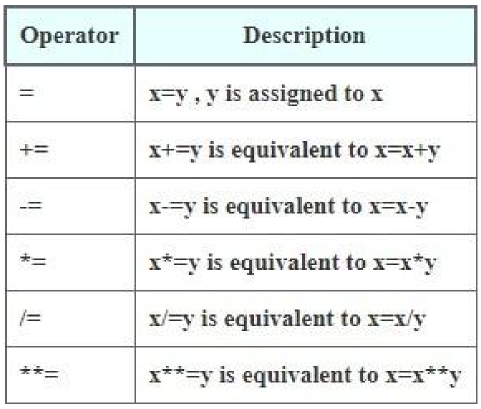
Example:

```
x =5
y =2

print('x>y is',x>y)
# Output: x > y is False
print('x<y is',x<y)
# Output: x < y is True
print('x==y is',x==y)
# Output: x == y is False
print('x!=yis',x!=y)
# Output: x != y is True
print('x>=y is',x>=y)
# Output: x >= y is False
print('x<=yis',x<=y)
# Output: x <= y is True
```

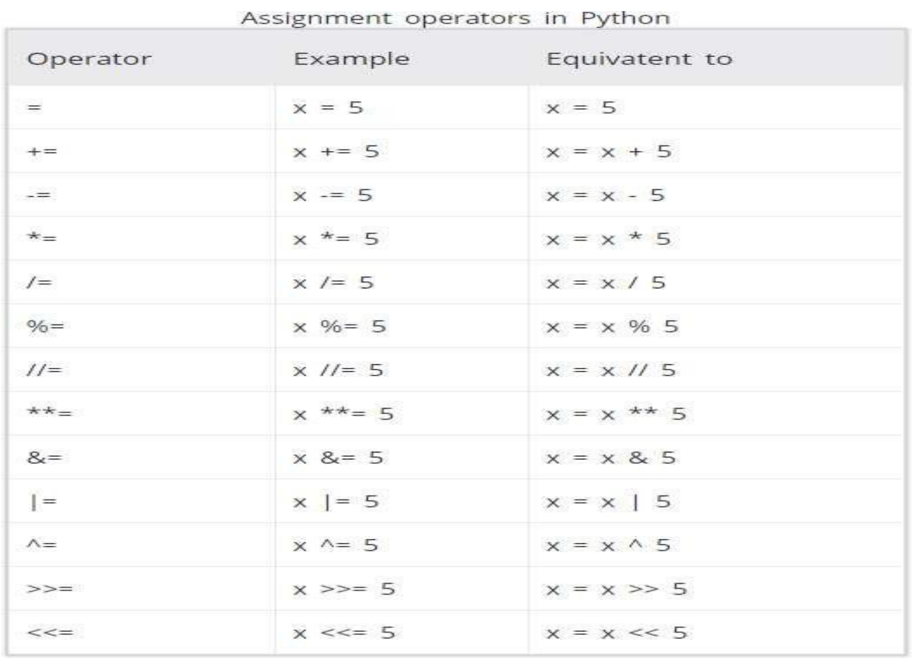
<u>**Assignment operators**</u>

In Python, Assignment operators are used to assigning value to the variable. Assign operator is denoted by = symbol. For example, name = "Jessa" here, we have assigned the string literal ‘Jessa’ to a variable name.

Also, there are shorthand assignment operators in Python. For example, a+=2 which is equivalent to a = a+2.



Operator	Description
=	x=y , y is assigned to x
+=	x+=y is equivalent to x=x+y
-=	x-=y is equivalent to x=x-y
*=	x*=y is equivalent to x=x*y
/=	x/=y is equivalent to x=x/y
**=	x**=y is equivalent to x=x**y



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

Logical operators

Logical operators are useful when checking a condition is true or not. Python has three logical operators. All logical operator returns a boolean value True or False depending on the condition in which it is used.

Operator	Description	Example
and (Logical and)	True if both the operands are True	a and b
or (Logical or)	True if either of the operands is True	a or b
not (Logical not)	True if the operand is False	not a

and (Logical and)

The logical and operator returns True if both expressions are True. Otherwise, it will return. False

Example

```
print(True and False) # False
print(True and True) # True
print(False and False) # False
print(False and True)  # false
# actual use in code
a = 2
b = 4
# Logical and
if a > 0 and b > 0:
        # both conditions are true
        print(a * b)
else:   print("Do nothing")
```

In the case of **arithmetic values**, Logical and always returns the **second value**; as a result

Example

```
print(10 and 20) # 20
```

or (Logical or)

The **logical** or the operator returns a boolean True if one expression is true, and it returns **False** if both values are false.

Example:

```
print(True or False) # True
print(True or True) # True
print(False or False) # false
print(False or True) # True
# actual use in code
a = 2
b = 4
# Logical and
if a > 0 or b < 0:
    # at least one expression is true so conditions is true
    print(a + b) # 6
else:
    print("Do nothing")
```

In the case of **arithmetic values**, Logical or it always returns the first value; as a result

Example

```
print(10 or 20) # 10
```

not (Logical not)

The logical not operator returns boolean True if the expression is false.

Example

```
print(not False) # True return complements result
print(not True) # True return complements result

# actual use in code
a = True

# Logical not
if not a:
  # a is True so expression is False
  print(a)
else:
    print("Do nothing")
```

In the case of **arithmetic values**, Logical not always return False for **nonzero** value.

Example

```
print(not 10) # False. Non-zero value
print(not 0) # True. zero value
```

→Membership operators

Python’s membership operators are used to check for membership of objects in sequence, such as string, list, tuple. It checks whether the given value or variable is present in a given sequence. If present, it will return True else False.

In Python, there are two membership operator **in** and **not in**

In operator

It returns a result as True if it finds a given object in the sequence. Otherwise, it returns False**.**

Example

```
My _list = [11, 15, 21, 29, 50, 70]
number = 15
if number in my_ list:
        print("number is present")
else:
       print("number is not present")
```

Not in operator

It returns True if the object is not present in a given sequence. Otherwise, it returns False

Example

```
my_ tuple = (11, 15, 21, 29, 50, 70)
number = 35
if number not in my_ tuple:
        print("number is not present")
else:
        print("number is present")
```

→Identity operators

Use the Identity operator to check whether the value of two variables is the same or not. This operator is known as a **reference-quality operator** because the identity operator compares values according to two variables’ memory addresses.

Python has 2 identity operators *is* and *is not*.

is operator

The *is* operator returns Boolean *True* or *False*. It Returns *True* if the memory address first value is equal to the second value. Otherwise, it returns *False.*

Example

```
x = 10
y = 11
z = 10
print(x is y) # it compare memory address of x and y
print(x is z) # it compare memory address of x and z
```

is not operator

The is not the operator returns boolean values either True or False. It Return True if the first value is
not equal to the second value. Otherwise, it returns False.

Example

```
x = 10
y = 11
z = 10
print(x is not y) # it campare memory address of x and y
 print(x is not z) # it campare memory address of x and z
```

Example

```
x1 , y1 = 5 ,5
x2 , y2 = "cse", ”cse”  x3 , y3= [1,2,3],[1,2,3]
```

➔ **Bitwise Operators**

In Python, bitwise operators are used to performing bitwise operations on integers. To perform bitwise, we first need to convert integer value to binary (0 and 1) value.

The bitwise operator operates on values bit by bit, so it’s called **bitwise**. It always returns the result in integer format. Python has 6 bitwise operators listed below.

1. & Bitwise and
2. | Bitwise or
3. ^ Bitwise xor
4. ~ Bitwise 1’s complement
5. << Bitwise left-shift
6. >> Bitwise right-shift

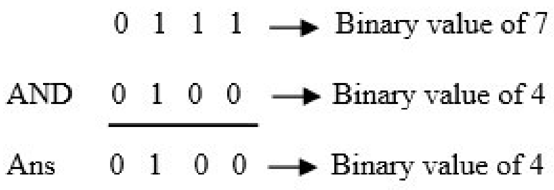
Bitwise and (&)

It performs **logical AND** operation on the integer value after converting an integer to a binary value and gives the result as a decimal value. It returns True only if both operands are True. Otherwise, it returns False.

Example

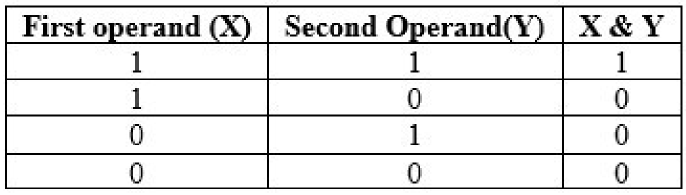
```
 a = 7
b = 4
c = 5
print(a & b) #4
print(a & c) #5
print(b & c) #4
```

Here, every integer value is converted into a binary value. For example, a =7, its binary value is 0111, and b=4, its binary value is 0100. Next we performed logical AND, and got 0100 as a result, similarly for a and c, b and c



```
     0 1 1 1  → Binary value of 7
AND  0 1 0 0  → Binary value of 4
     -------
Ans  0 1 0 0  → Binary value of 4
```

AND operators Truth table (True = 1, False = 0)



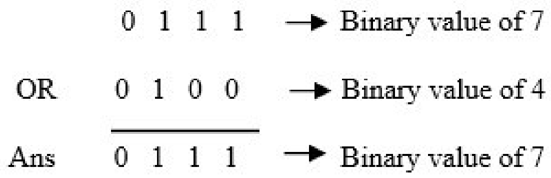
First operand (X)	Second Operand(Y)	X & Y
1	1	1
1	0	0
0	1	0
0	0	0

<u>Bitwise or(|)</u>

It performs **logical OR** operation on the integer value after converting integer value to binary value and gives the result a decimal value. It returns False only if both operands are True. Otherwise, it returns True.

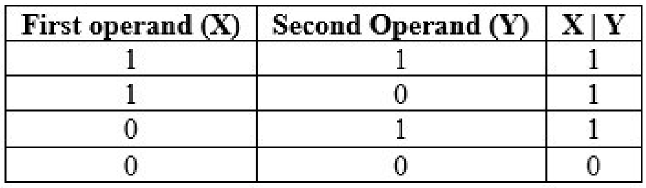
Example

```
a = 7
 b = 4
c = 5
print(a | b)
print(a | c)
print(b | c)
```



```
     0 1 1 1  → Binary value of 7
OR   0 1 0 0  → Binary value of 4
     -------
Ans  0 1 1 1  → Binary value of 7
```

OR operators Truth table (True = 1, False = 0)



First operand (X)	Second Operand (Y)	X | Y
1	1	1
1	0	1
0	1	1
0	0	0

Here, every integer value is converted into binary. For example, a =7 its binary value is 0111, and b=4, Its binary value is 0100, after logical OR, we got 0111 as a result. Similarly for *a* and *c*, *b* and *c*.

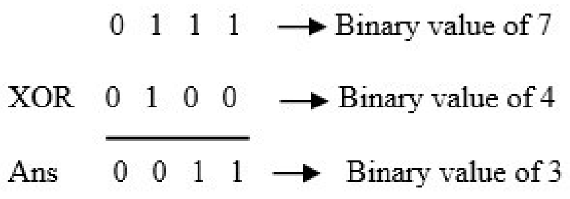
Bitwise xor ^

It performs Logical XOR ^ operation on the binary value of a integer and gives the result as a decimal value.

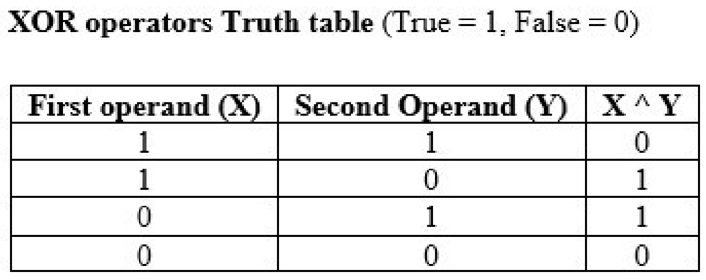
Example: –

```
a = 7
b = 4
c = 5
print(a ^ c)
print(b ^ c)
```

Here, again every integer value is converted into binary. For example, a =7 its binary value is 0111 and b=4, and its binary value is 0100, after logical XOR we got 0011 as a result. Similarly for a and c, b and c.



```
      0 1 1 1  → Binary value of 7

XOR   0 1 0 0  → Binary value of 4
      -------
Ans   0 0 1 1  → Binary value of 3
```



XOR operators Truth table (True = 1, False = 0)

First operand (X)	Second Operand (Y)	X ^ Y
1	1	0
1	0	1
0	1	1
0	0	0

Bitwise 1’s complement ~

It performs 1’s complement operation. It invert each bit of binary value and returns the bitwise negation of a value as a result.

Example

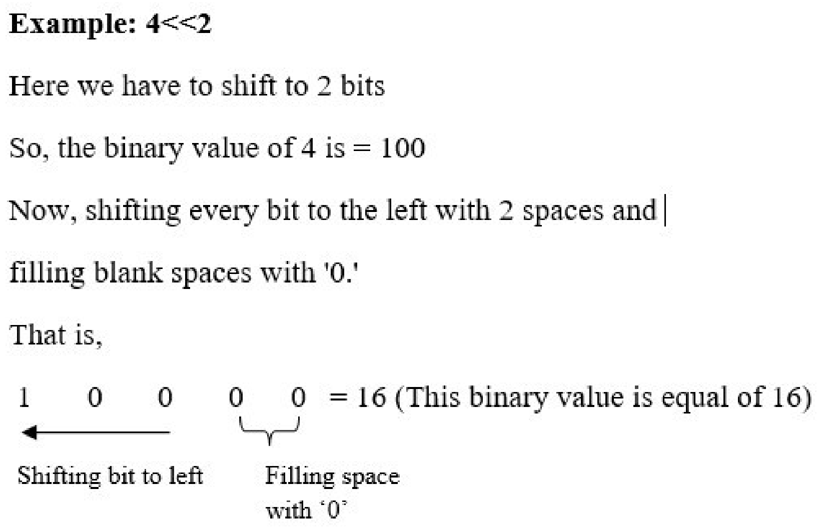
```
a = 7
b = 4
c = 3
print(~a, ~b, ~c)
 # Output -8 -5 -4
```

Bitwise left-shift <<

The left-shift << operator performs a shifting bit of value by a given number of the place and fills 0's to new positions.

Example: –

```
print(4 << 2) # Output 16
print(5 << 3)# Output 40
```



Example: 4<<2

Here we have to shift to 2 bits

So, the binary value of 4 is = 100

Now, shifting every bit to the left with 2 spaces and

filling blank spaces with '0.'

That is,

1 0 0 0 0 = 16 (This binary value is equal of 16)

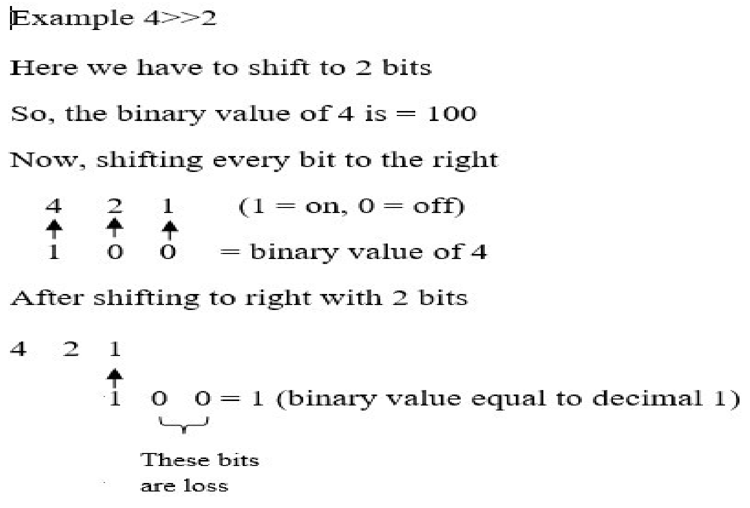
Shifting bit to left

Filling space with '0'

Bitwise right-shift >>

The left-shift >> operator performs shifting a bit of value to the right by a given number of places. Here some bits are lost.

```
print(4 >> 2)# Output 1
```



Example 4>>2

Here we have to shift to 2 bits

So, the binary value of 4 is = 100

Now, shifting every bit to the right

4 2 1 (1 = on, 0 = off)

1 0 0 = binary value of 4

After shifting to right with 2 bits

4 2 1

1 0 0 = 1 (binary value equal to decimal 1)

These bits are loss

Type conversions

You can explicitly cast, or convert, a variable from one type to another type.

- To explicitly convert a float number or a string to an integer, cast the number using **int()** function. Ex: f=3.2, i=int(f), then variable i contains 3, decimal part is truncated.
- The **float()** function returns a floating point number constructed from a number or string. Ex: a=3, **f=float(a),** then f contains 3.0
- The **str()** function returns a string which is fairly human readable. Ex: a=3, **s=str(a),** then s contains ‘3’
- Convert an integer to a string of one character whose ASCII code is same as the integer using **chr()** function. The integer value should be in the range of 0–255.
- char_A=**chr(65),** here char_A contains ascii character capital A
- char_a=**chr(97),** here char_a contains ascii character small a
- Use **complex()** function to print a complex number with the value **real + imag*j** or convert a string or number to a complex number. c1=complex(3,4), then if we print the value of c1, it gives out output as follow: (3+4j)
- The **ord ()** function returns an integer representing Unicode code point for the given Unicode character.
- alpha_Z=**ord('Z’),** where alpha contains ascii value 90.
- Convert an integer number (of any size) to a lowercase hexadecimal string prefixed with “0x” using hex() function. For example: i_ to_ h=**hex(255),**I_ to _h contains ‘0xff’ and i_ to_ h=hex(16), i_ to _h contains ‘0x10’
- Convert an integer number (of any size) to an octal string prefixed with “0o” using **oct()** function. For example, o=oct(8), where o contains ‘0o10’ and o=oct(16), where o contains ‘0o20’
- The **type()** function returns the data type of the given object. If we pass 20 to the type() function as follow type(20) then its type will be displayed as <class 'int'>

Note: Python is a dynamically typed, high level programming language.

Expressions

An Expression is a combination of operators and operands that computes a value when executed by the Python interpreter. In python, an expression is formed using the mathematical operators and operands (sometimes can be values also).

Precedence of operator determines the way in which operators are parsed with respect to each other. Operators with higher precedence will be considered first for execution than the operators with lower precedence. **Associativity** determines the way in which operators of the same precedence are parsed. Almost all the operators have left-to-right associativity. The exponent and assignment operators have right-to-left associativity. The acronym PEMDAS is a useful way to remember the order of operations:

Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2*(3-1) is 4, and (1+1)**(5-2) is 8.

Exponentiation has the next highest precedence, so 2 **1+1 is 3 and not 4, and 3*1** 3 is 3 and not 27.

Multiplication and Division have the same precedence, which is higher than Addition and Subtraction, which also have the same precedence. So 2 *3-1 yields 5 rather than 4, and 2/3-1 is -1, not 1 (re-member that in integer division, 2/3=0).

Precedence of the Operators

Precedence level	Operator	Meaning
1 (Highest)	()	Parenthesis
2	**	Exponent
3	+x, -x ,~x	Unary plus, Unary Minus, Bitwise negation
4	*, /, //, %	Multiplication, Division, Floor division, Modulus
5	+, -	Addition, Subtraction
6	<<, >>	Bitwise shift operator
7	&	Bitwise AND
8	^	Bitwise XOR
9	|	Bitwise OR
10	==, !=, >, >=, <, <=	Comparison
11	is, is not, in, not in	Identity, Membership
12	not	Logical NOT
13	and	Logical AND
14 (Lowest)	or	Logical OR

Data types in Python

Numbers

•Int / float/complex : type
•Describes the numeric value & decimal value
•These are immutable modifications are not allowed.

Boolean

• bool : type
•represent True/False values.
•0 =False & 1 =True
•Logical operators and or not return value is Boolean

Strings

•str : type
•Represent group of characters
•Declared with in single or double or triple quotes
•It is immutable modifications are not allowed.

Lists

•list : type
•group of heterogeneous objects in sequence.
•This is mutable modifications are allowed
•Declared with in the square brackets [ ]

Tuples

•tuple : type
•group of heterogeneous objects in sequence
•this is immutable modifications are not allowed.
•Declared within the parenthesis ( )

Dictionaries

•dict : type
•it stores the data in key value pairs format.
•Keys must be unique & value
•It is mutable modifications are allowed.
•Declared within the curly brasses {key: value}

Boolean data type : (bool)

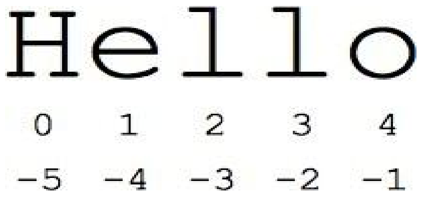
- true& false are result values of comparison operation or logical operation in python.
- true & false in python is same as 1 & 0   1=true 0=false
- except zero it is always True.
- while writing true & false first letter should be capital otherwise error message will be generated.
- Comparison operations are return Boolean values.

Example:

```
print (1 == 1)            #true

print (5 > 3)             #true

print  (True  or  False)  #true

print (3> 7)              #false


print (True and False)   #false
```

Strings Data type : (str)

- A string is a list of characters in order enclosed by single quote or double quote.
- Python string is immutable modifications are not allowed once it is created.
- In java String data combine within data it will become String but not in python.
- String index starts from 0,trying to access character out will generate Index Error.
- In python it is not possible to add any two different datatypes,possible to add only same data



Slice Notation

- <string_name>[startIndex:endIndex],
- <string_name>[:endIndex],
- <string_name>[startIndex:]

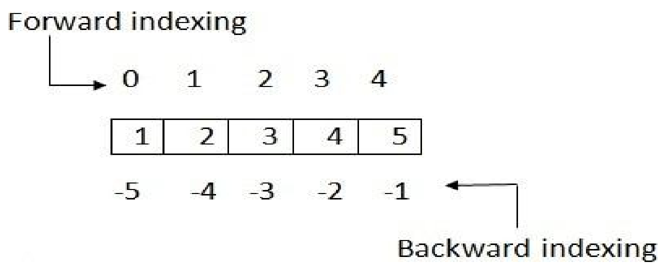
- □ s[1:4]is'ell' --chars starting at index1andextending up to but not including index4s[1:] is'ello'—omitting either index defaults to the start or end of the string
- □ s[:]is 'Hello'—omitting both always gives us a copy of the whole thing

- □ s[1:100] is 'ello' -- an index that is too big is truncated down to the string length s[-1] is 'o' – last char(1st from the end)
- □ s[-4]is 'e' --4th from the end
- □ s[:-3]is'He'—going up to but not including the last 3 chars.
- □ s[-3:]is'llo'—starting with the 3rd char from the end and extending to the end of the string.

List data type:(list)

- List is used to store the group of values& we can manipulate them, in list the values are stores in index format starts with 0;
- List is mutable object so we can do the manipulations.
- A python list is enclosed between square([])brackets.
- In list insertion order is preserve means in which order we inserted element same order output is printed.
- A list can be composed by storing a sequence of different type of values separated by commas.

  <list_ name>=[value1,value2,value3,..., value n];
- The list contains forward indexing &back word indexing.



Example: List data

data1=[1,2,3,4]	#list of integers
data2=['x','y','z']	#list of String
data3=[12.5,11.6]	#list of floats
data4=[]	# empty listdata5=['ramu',10,56.4,'a']#list with mixed data types

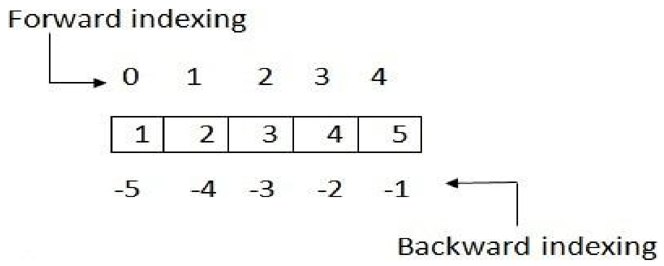
Practice Examples:

Accessing List data

```
data1=[1,2,3,4]
data2=['srkr' ,'anu',' deva']
print(data1[0])
print (data1[0:3])
print(data2[-3:-1])
print(data1[0:])
print(data2[:2])
print(data2[:])
```

Tuple data type:(tuple)

- tuple:type
- group of heterogeneous objects in sequence
- this simmutable modificationsare not allowed.
- Declared with in the parenthesis()
- Insertion order is preserve dit means in which order we inserted the objects same order output is printed.
- The tuple contains forward indexing &back ward indexing.



Example:

```
tup1 = ('ratan',  'anu', 'durga')

tup2=(1,2, 3,4,5)
tup3="a", "b", "c", "d"                    # valid not recommended
tup4=()
tup5=(10)
tup6 = (10,)
tup7=(1,2,3,"ratan",10.5)
print(tup1)
print(tup2)
print(tup3)
print(tup4)
print(type(tup5)) #<class 'int'>
print(type(tup6))#<class 'tuple'>
print(tup7)
```

Syntactically, a tuple is a comma-separated list of values:

t= 'a',' b' ,'c', 'd', 'e'

Although it is not necessary, it is common to enclose tuples in parentheses to help us quickly identify tuples when we look at Python code:

t= ('a' ,'b', 'c', 'd',' e')

Dictionary data type:

- ▢ List, tupple, set data types are used to represent individual objects as a single entity.
- ▢ To store the group of objects as a key-value pairs use dictionary.
- ▢ Adictionary is a data type similar to arrays ,but works with keys and values instead of indexes.
- ▢ Eachvaluestoredinadictionarycanbeaccessedusingakey,whichisanytypeofobject(astring,a number ,a list, etc.)instead of using its index to address it.
- ▢ The keys must be unique keys but values can be duplicated.

Example:

```
phonebook = {}
phonebook["ramu"]=935577566
phonebook["anu"]=936677884
phonebook["devi"] = 9476655551
print(phonebook)
```

Example:

Alternatively, a dictionary can e initialized with the same values in the following notation:

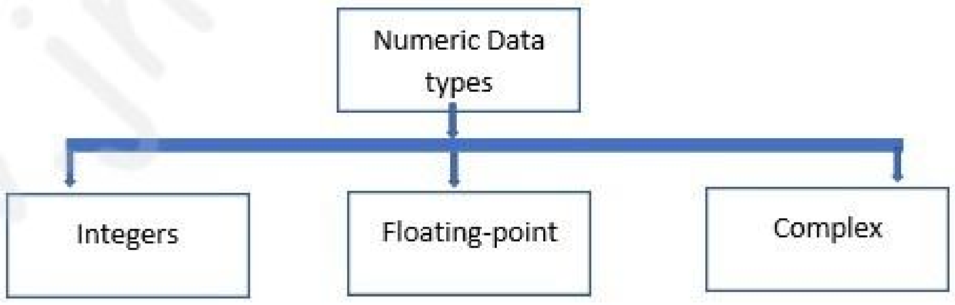
```
phonebook = { "ramu" : 935577566, "anu" : 936677884, "devi" : 9476655551}
print(phonebook)
```

- ▢ Dictionaries can be created using pair of curly braces ({}). Each item in the dictionary consist of key , followed by a colon, which is followed by value .And each it emis separated using commas(,).
- ▢ An item has a key and the corresponding value express edas a pair ,key :value.
- ▢ In dictionary values can be o fany data type and can repeat,.
- ▢ Keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

Example:

```
#empty dictionary
        my_ dict = {}print(my_dict)
#dictionary with integer keys
        my_dict = {1: 'apple', 2: 'ball'}print(my_dict)
#dictionary with mixed keys
        my_dict={'name':'John',1:[2,4,3]}print(my_dict)
```

Numeric Data Types

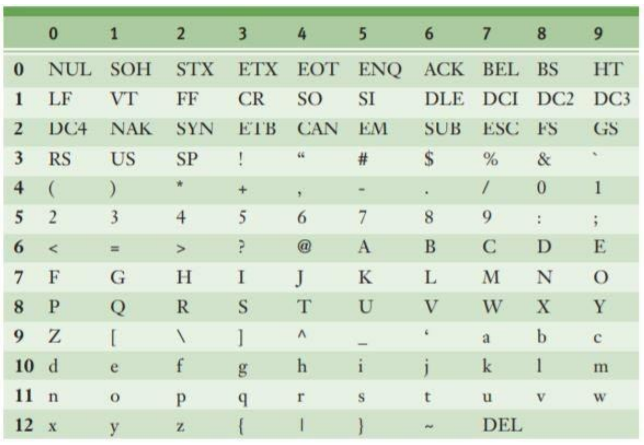


Under the numeric data types python has three different types: integers, floating-point, complex numbers. These are defined as int, float, complex in python. Integers can be of any length; it is only limited by the memory available. Python uses floating-point.

numbers to represent real numbers. A **floating-point** number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. 1 is an integer, 1.0 is floating point number. A floating-point number can be written using either ordinary decimal notation or scientific notation. Example, 37.8 can be represented in scientific notation as 3.78e1. **Complex numbers** are written in the form, x + yj, where x is the real part and y is the imaginary part. Example, (3+4j).

Character Sets

Some programming languages use different data types for strings and individual characters. In Python, character literals look just like string literals and are of the string type. But they also belong to several different character sets, among them the ASCII set and the Unicode set.



	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DCI	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	SP	!	“	#	$	%	&	`
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	‘	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

Decision Structures and Boolean Logic

In all most all programming languages the control flow statements have been classified as Selection/Decision Statements, Loop/Repetition/Iterative Statements, and Jump Statements. Under the Decision statements in Python we have ***if, elif and else*** statement. Under the Repetition statements we have ***for*** and ***while*** statements. Under the Jump statements we have break, continue and ***pass*** statements.

The if Decision statement:

The simplest form of decision/selection is the if statement. This type of control statement is also called a ***one-way selection*** statement, because it consists of a condition and just a single sequence of statements. If the condition is True, the sequence of statements is run. Otherwise ,control proceeds to then ext statement following the entire selection statement.

The Syntax of the if statement will be as follow:

```
If Boolean_Expression:
        Statement 1
        Statement2
        :

        Statement N
```

The if decision control flow statement starts with **if** keyword and ends with a colon. The expression in an if statement should be a Boolean expression which will be evaluated to True or False. The Boolean _Expression is evaluated to True, then if block will be executed, otherwise the first statement after the if block will be executed. The statements inside the if block must be properly indented with spaces or tab.

Write python program to test whether a given number is Even or not using if decision statement.

```
#Even or Odd
n=int(input("Enter the number"))
if n%2==0:
    print(n,' is even number')

    print('End of ifblock')
print('End
```

The if- else decision statement

An **if** statement can also be followed by an **else** statement which is optional. An else statement does not have any condition. The statements in the if block are executed if the Boolean_Expressionis **True**, otherwise the else block will beexecuted.

```
If Boolean_Expression:
         statements1
else:
       statements1
```

The if… else statement used for two-way decision, that means when we have only

Two alternatives .The syntax of if- else will be as follow:

Here, Statements 1 and Statement 2 can be single statement or multiple statements. The statements inside if block and else block must be properly indented. Both the indentation need not be same.

if- else decision statements.

```
#Even or Odd
n=int(input("Enter the number"))
if n%2==0:
        print(n,' is even number')
        print('End of ifblock')
else:
        print(n,' isodd')
        print('else block')
 print('End of the program')
 print('Execution is  completed')
```

The if-elif-else decision statement

The if… elif…else is also called multi- way decision statement. This multi-way decision statement is preferred whenever we need to select one choice among multiple alternatives. The keyword '**elif**' is short for 'else if'. The else statement will be written at the end and will be executed when no if or elif blocks are executed. The syntax of will be as follow:

```
If Boolean_expression1:
        Statements
elif Boolean_expression2:
        Statements
elif Boolean_exxpression3:
        Statements
else:
        Statements
```

Write a Program to Prompt for a Score between 0.0 and 1.0. If the Score Is Out of Range, Print an Error. If the Score Is between 0.0 and 1.0, Print a Grade Using the Following Table.

Score	>=0.9	>=0.8	>=0.7	>=0.6	<0.6
Grade	A	B	C	D	F

```
 score=float(input('Enter your  score:'))
 if score<0orscore>1:
    print('Wrong input is given')
elif  score>=0.9:
    print('Your Grade is A')
elif score>=0.8:
    print('Your Grade is B')
 elif  score>=0.7:
    print('Your  Grad e is C')
 elif score>=0.6:
    print('Your Grade is D')
else:
    print('Your Grade is F')
```

Nested if statements

Sometimes it may be need to write an if statement inside another if block then such if statements are called nested if statements. The syntax would be as follow:

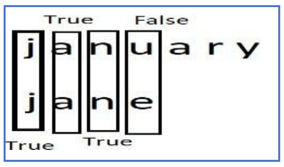
```
ifBoolean_expression1:
        ifBoolean_expression2:
                ifBoolean_expression3:
                        Statements
                else:
                        Statements
```

Program to Check If a Given Year Is a Leap Year

```
year=int(input('Enter  year:'))
if year%4==0:
    if year%100==0:
        if year%400==0:#nested if statements
            print(f'{year}is a leap year')
        else:
            print(f'{year}is not leap year')

    else:
        print(f'{year}is a leap year')
else:
        print(f'{year}is not a leap year')
```

Comparing Strings

We can use comparison operators such as >, <, <=,>=, ==, and! =to compare two strings. This expression can return a Boolean value either True or False. Python compares strings using ASCII value of the characters. For example,



Print(january==june)
Flase

String equality is compared using == (double equal sign). This comparison process is carried out as follow:

- First two characters (j and j) from the both the strings are compared using the ASCII values.
- Since both ASCII values are same then next characters (a and a) are compared. Here they are alsoequal, and hence next characters (n and n) from both strings are compared.
- This comparison also returns True, and comparison is continued with next characters (u and e).

Since the ASCII value of the ‘u’ is greater than the ASCII value of ‘e’ this time it returns False. Finally, the comparison operation returns False.

Boolean Variables

The variables that store Boolean value either True or False called Boolean variables. Ift he expression is returning a Boolean value then it is called Boolean expression.

Example:

```
>>>X=True
>>>Y=False
>>> a>5 and a<10 #Boolean expression
```

I. Repetition Structures or Control Structures

Introduction

Whenever if we want to execute a block of statements repeatedly for some finite number of times or until some condition is satisfied then repetition structures are used. The repetition structures also known as loops, which repeat an action. Each repetition of the action is known as a ***pass*** or iteration. There are two types of loops—those that repeat an action a predefined number of times (***definite iteration***) and those that perform the action until the program determines tha it need to stop (***indefinite iteration***). There are two loop statements in Python, for and while.

The while loop In many situations, however, the number of times that the block should execute is not Known in advance. The program’s loop continues to execute as long as valid input is entered and terminates at special input. This type of process is called conditional iteration .In this section, we explore the use of the ***while loop*** to describe conditional iteration

While loop repeats as long as a certain Boolean condition is met. The block of statements is repeatedly executed as long as the condition is evaluated to True. The general form of while will be as follow:

```
While condition:#Loop Header
        statement1
         statement2
        ………….
        Statements
```

The first line is called loop header which contains a keyword while, and condition which return a Boolean value and colon at the end. The body of the loop contains statement1, statements2, and so on. Thus is repeated until the condition is evaluated to True otherwise loop will be terminated.

Write a python program to demonstrate wile loop for computing the sum of numbers entered by the user. Terminates when user enters special character ‘space’.

```
data=input('Enter any number')sum=0
while data!=" ":
    sum=sum +int(data)
   data=input('Enter any number or space to
quit')print('The sum is:',sum)
```

The count control with while loop

We can also use while loop for count-controlled loops. The body of the while is Repeatedly executed until the condition which returns a Boolean value True. Otherwise body of the loop is terminated and the first statement after the while loop will be executed.

Write a program that asks the user for their name and how many times to print it. The program should print out the user’s name the specified number of times.

```
n=int(input('Enter the number that you want to display your name:'))
name=input('Enter name:')
while n>=0:
    print(name)
    n=n-1
print('End of the program')
```

Output:

```
Enter the number that you want to display your name:3

Enter name:Guido
Guido Guido Guido
End of the program
```

The for loop

For loop iterates over a given sequencer list. It is help in running a loop on eachitem in the list. The general form of "for" loop in Python will be as follow:

```
For variable in[value1,value2,etc.]:#Loop Header
    statement1

    statement2
    ………..
    Statement N
```

Here variable is the name of the variable. And **for** and **in** are the keywords .Inside the square brackets a sequence of values is separated by comma. In Python, a comma-separated sequence of data items that are enclosed in a set of square brackets is called a ***list***. The list is created with help of [] square brackets. The list also can be created with help of tuple. We can also use range () function to create the list.

The general form of the range() function will be as follow:

- range(number)–ex: **range(10)**–Ittakesallthevaluesfrom0to9
- range (start ,stop, interval _size) –ex: **range(2,10,2)**-It lists all the numbers such as 2,4,6,8.
- range(start ,stop)-ex: **range(1,6)**, lists all the numbers from 1to 5, but not 6. Here, by default the interval size is 1.

Write a Python Program to find the sum of all the items in the list using forloop.

fortest.py	**Output**

#sum of all items in the list
s=0
for x in[1,2,3,4,5]: **# list**
    s=s+ x
print("The sum of all items in the list is:",s)
	The sum of all items in the list is:15

Write a program that uses a *for* loop to print the numbers 8, 11, 14, 17, 20, . . . ,83,86,89.(LabPrg3)

```
for i in range(8,90,3):
    Print(i)
```

Use a for loop to print a triangle like the one below. Allow the user to specify how high the triangle should be.(labPrg5)

```
For i in range(0,4):
  For j in range(0,i+1):
      print('*',end="")
  print("\n")
```

Calculating a Running Total

We can calculate the sum of input numbers while entering from the keyboard as demonstrated in the following example.

```
n=int(input('Enter n:'))

sum=0

for i in range(n):

    data=float(input('enter value'))

    sum=sum+ data

#display sum

print('Sum is:',sum)
```

Output:

```
Enter n:4

Enter  value12

Enter  value13

Enter  value21

Enter  value22

Sumis:68.0
```

Input Validation Loops

Loops can be used to validate user input. For instance, a program may require the user to enter a positive integer . Many of us have seen a “yes/no” prompt at some point, although probably in the form of a dialog box with buttons rather than text.

```
Import random

 action="Y"

While action=="Y":
   print("Generating random number...")
   randomNumber = random.randint(1,10)
   print("Random numberi s",randomNumber)
    action=input("Another(Y/N)?")
   while action!="Y" induction!="N":
      action=input("Invalid input! Enter Yor N:")
 print("All done!")
```

Nested loops

Writing a loop statement inside another r is called Nested loops. The" ***inner loop***" will be executed one time for each iteration of the "***outer loop***". We can put any type of loop inside any other type of loop. For example, a for loop can be inside a while loop or vice versa.

Syntax of nested loops:

```
# Nested for loops

For iterating_ var in sequence: #outer for

   loopfor iterating _var in sequence: #inner

   for loop

      statements(s)

     statements(s)
```

```
#Nested loop with for &while
for iterating_var in sequence: #outer for loop
 while expression: #inner for loop
          statement(s)

     statements
```

Use a for loop to print a triangle like the one below. Allow the user to specify how much high the triangle should be.

Using for nested loops	Using for and while nested loops

for i in range(4):
    for j in range(0,i+1):
        print('*',end=' ')
    print('\n')
	for i in range(4):
	      j=0
	    While j<(i+1):
	        print('*',end=")
	        j=j+1
	    print('\n')

* ** *** ****	

Else with loops

Loop statements may have an else clause

- It is executed when the loop terminates through exhaustion of the list (with for loop).
- It is executed when the condition becomes false (with while loop), but not when the loop is terminated by a break statement.

Example: Printing all primes numbers up to 100

```
print('2')

for I in range(3,101,2):


    for  j in range(2,i):

        if I %j==0:

            break

    else:

        print(i)
```

Using the while inner loop

```
print('2')
for i in
range(3,101,2):
j=2
   while j<i:
      ifi%j==0:
         break
      j=j+1
   else:
      print(i)
```

Jump Statements

- We have three jump statements: **break, continue and pass.**
  - Break statement:
    - It terminates the current loop and resumes execution at the next statement, just like the traditional break statement in C.
    - The break statement can be used in both while and for loops.

Write a python program to search for a given number whether it is present in the list or not.

```
n=int(input('Enter number to  search  :'))

for x in range(1,11):

#[1,2,3,4,5,6,7,8,9,10]

    if x==n:

        print(n,' is found')

        break #terminates the loop

print('end of program')
```

The continue statement

- It returns the control to the beginning of the loop.
- The continue statement skips all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.
- The continue statement can be used in both while and for loops.

```
 n=int(input('Enter number to  search  :'))
for x in range(1,11):
#[1,2,3,4,5,6,7,8,9,10]
    if x==n:
        continue
else:
        print(n,' is found')
print('end of program')
```

The pass statement

- The pass statement does nothing
- It can be used when a statement is required syntactically but the program requires no action
- Example: creating an infinite loop that does nothing

```
While True:
            pass
```

Example program to demonstrate the pass statement

```
f=True
while f:
    pass
    print('This line will be printed')

    f=False
print('End of the program')
```

Output:

```
        This line will be printed
        End of the program
```

Built-in functions and Modules in Python

- The Python interpreter has a number of built-in functions. They are loaded automatically as the interpreter starts and are always available.
- For example, print**()**and **input( )**for I/O.
- Number conversion functions **int(),float(),complex()**,data type conversions **list(),tuple(), set(),**etc.
- In addition to built- in functions ,a large number of **pre-defined functions** are also available as a part of libraries bundled with Python distributions. These functions are defined as **modules**.
- A module is a file containing definitions of **functions, classes, variables ,constants or any other Python objects** .Contents of this file can be made available to any other program.
- Built-in modules are written in C and integrated with the Python interpreter.
- Each built-in module contains resources for certain system-specific functionalities such as OS management, disk IO, etc .The standard library also contains many Python scripts (with the.py extension) containing useful utilities.
- To display a list of all available modules, use the following command in the Python console:

  ```
  >>>help('modules')
  ```

  Which displays all modules that are supported in the python3

How to import modules

The *import* statement is used to import the whole module. Also, we can import specific classes and functions from a module.

For example, import module name.

With the help of the import keyword, both the **built-in** and **user-defined** modules are imported.

```
        import math
        # use math module functions
        print(math. sqrt(5))
# Output 2.23606797749979
```

Import multiple modules

If we want to use more than one module, then we can import multiple modules. This is the simplest form of import a statement that we already use in the above example.

Syntax of import statement:

import module1[,module2[,.. module N]

```
# Import two modules
import math, random

print(math. factorial(5))
print(random .randint(10, 20))
```

Import only specific classes or functions from a module

To import particular classes or functions, we can use the form...import statement. It is an alternate way to import. By using this form, we can import individual attributes and methods directly into the program.

Syntax of from...import statement:

from <module_name> import <name(s)>

Example

# import only factorial function from math module

```
from math import factorial

print(factorial(5))
```

Import with renaming a module

If we want to use the module with a different name, we can use from..import…as statement.

It is also possible to import a particular method and use that method with a different name. It is called **aliasing**. Afterward, we can use that name in the entire program

Syntax of from..import ..as keyword:

from <module_ name> import <name> as <alternative _name>

Example 1: Import a module by renaming it

```
                    import random as rand

print(rand.rand range(10, 20, 2))
```

.**Example 2**: import a method by renaming it

```
# rename randint as random _number

          from random import  randint as random_ number

          # Gives any random number from range(10, 50)

          print(random_number(10, 50))
```

Import all names

If we need to import all functions and attributes of a specific module, then instead of writing all function names and attribute names, we can import all using an **asterisk** *.

Syntax of import * statement:

import *

Example

```
from math import *
print(pow(4,2))
print(factorial(5))
print(pi*3)
print(sqrt(100))
```

Create Module

In Python, to create a module, write Python code in the file, and save that file with the.py extension. Here our module is created.

Example

```
def my_ func():
 print("Learn Python easily")
```

Variables in Module

In Python, the module contains Python code like classes, functions, methods, but it also has variables. A variable can list, tuple, dict, etc.

Let’s see this with an example:

First, create a Python module with the name test_module.py and write the below code in that file.

Now, create a Python file with the name test_file.py, write the below code and import the above module test_module.py in that file. See the following code.

Example

```
cities_ list = ['Mumbai', 'Delhi', 'Bangalore' ,import test _module

# access first city
city = test_ module.  Cities _list[1]
print("Accessing 1st city:", city)

# Get all cities
cities = test_ module .cities_ list
print("Accessing All cities :", cities) 'Karnataka', 'Hyderabad']
```

When we execute this test_file.py, the variable of test_module.py is accessible using the dot(.)operator.

(i) Python –Math Module:

Some of the most popular mathematical functions are defined in the math module. These include trigonometric functions ,representation functions, garithmic functions, angle conversion functions, etc. In addition, two mathematical constants are also defined in this module.

- ✓ Pie (π)is a well-known mathematical constant ,which is defined as the ratio of the circumference to the diameter circle and its value is 3.141592653589793.

```
>>>import math
>>>math.pi3.141592653589793
```

- ✓ Another well-known mathematical constant defined in the math module is **e**. It is called **Euler' s number** and it is abase of the natural logarithm. Its value is2.718281828459045.

```
>>>math.e2.718281828459045
```

- ✓ The math module contains functions for calculating various trigonometric ratios for a given angle.
- ✓ The functions(sin , cos ,tan ,etc.)need the angle in radian sa san argument. We ,on the other hand, are used to express the angle in degrees.

The math module presents two angle conversion functions: degrees() and radians(), to convert the angle from degrees t oradians and vice versa.

- ✓ For example ,the following statements convert the angle of 30 degrees to radians and back (Note: π radians is equivalent to180 degrees).

```
>>>math .radians(30)0.5235987755982988
>>>math. degrees(math .pi/6)29.999999999999996
```

- ✓ The following statements show **sin, cos and tan** ratios for the angle of 30 degrees (0.5235987755982988 radians):

```
>>math.sin(0.5235987755982988)0.49999999999999994
>>>math.cos(0.5235987755982988)0.8660254037844387
>>>math.tan(0.5235987755982988)0.5773502691896257
```

- ✓ You may recall that sin(30) =0.5, cos(30)=32 (which is 0.8660254037844387) and tan(30)=13 (which is 0.5773502691896257).

math.log():

- ✓ The math.log() method returns the natural logarithm of a given number. The natural logarithm is calculated to the base e.

```
>>>math.log(10)2.302585092994046
```

math.log10():

- ✓ Themath.log10()method returns the base-10 logarithm of the given number. It is called the standard logarithm.

```
>>>math.log10(10)1.0
```

math.exp():

- ✓ The math.exp() method returns a float number after raising e(math.e) to given number.In other words,exp(x) gives e**x.

```
>>>math.exp(10)1.0
```

This can be verified by the exponent operator.

```
>>>math .e**1022026.465794806703
```

math.pow():

- ✓ The math.pow() method receives two float arguments ,raises the first to the second and returns the result .In other words, pow (4,4)is equivalentto4**4.

```
>>>math.pow(2,4)16.0
>>>2**416
```

math. sqrt():

- ✓ The math .sqrt()method returns the square root of a given number.

```
>>>math.  sqrt(100)10.0
>>>math . sqrt(3)1.7320508075688772
```

Representation functions:

- ✓ The **ceil()** function approximates the given number to the smallest integer, greater than or equal to the given floating point number. The **floor()** function returns the largest integer less than or equal to the given number.

```
>>>math .ceil(4.5867)5
>>>math .floor(4.5687)4
```

(ii) Python-Statistics Module:

- ✓ The statistics module provides functions to mathematical statistics of numeric data. The following popular statistical functions are defined in this module.
- ✓ The **mean()** method calculates the arithmetic mean of the number sinalist.

```
>>>import statistics
>>>statistics .mean([2,5,6,9])5.5
```

- ✓ The **median()**method returns the middle value of numeric dataina list.

```
>>>import statistics
>>>statistics .median([1,2,3,8,9])3
>>>statistics .median([1,2,3,7,8,9])5.0
```

- ✓ The **mode()** method returns the most common data point in the list.

```
>>>import statistics
>>>statistics. mode([2,5,3,2,8,3,9,4,2,5,6])2
```

- ✓ The **stdev()** method calculates the standard deviation on a given sample in theformof a list.

```
>>>import statistics
>>>statistics.stdev([1,1.5,2,2.5,3,3.5,4,4.5,5])1.3693063937629153
```

(iii) Python-Random Module

- ✓ Functions in the random module depend on a pseudo-random number generator function random(),which generates random float numberbetween0.0and1.0.
- ✓ **random.random():**Generates a random float number between 0.0 to 1.0. The function doesn't need any arguments.

```
>>>import random
>>>random .random() 0.645173684807533
```

- ✓ **random.randint():** Returns a random integer between the specified integers.

```
>>>import random
>>>random.randint(1,100)  95
>>>random .randint(1,100 ) 49
```

- ✓ **random. randrange():** Returns a randomly selected element from the range created by the start, stop and step arguments.The value of start is 0 by default. Similarly, the value of step is1by default.

```
>>>random. randrange(1,10)2
>>>random .randrange(1,10,2) 5
>>>random .randrange(0,101,10) 80
```

- ✓ **random. choice():** Returns a randomly selected element from a non-empty sequence. An empty sequence as argument raises an Index Error.

```
>>>import random
>>>random. choice('computer') 't'
>>>random.   choice([12,23,45,67,65,43])45
>>>random. choice((12,23,45,67,65,43))67
```

- ✓ **random.shuffle():**This functions randomly reorders the elements in a list.

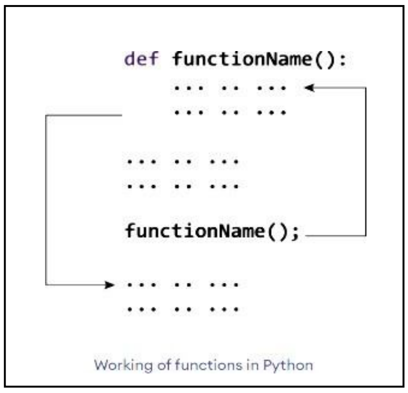
```
>>>numbers=[12,23,45,67,65,43]
>>>random .shuffle(numbers)
>>>numbers
[23,12,43,65, 67,45]
>>>random. shuffle(numbers)
>>>numbers
[23,43,65,45, 12,67]
```

- ✓ **random. sample() :** function for randomly picking more than one element from the list without repeating elements. It returns a list of unique item chosen randomly form a list , sequence or set

```
sample list=[20,40, 50, 60 ,70,      200]
ramdom. sample(samplelist,4)
```

Python Functions

- In Python, a function is a group of related statements that performs a specific task.
- Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.
- It avoids repetition and makes the code reusable.



Syntax of Function:

```
def function _name(parameters):
"""docstring""" statement(s)
return
```

Above shown is a function definition that consists of the following components.

- Keyword ***def*** that marks the start of the function header.
- A ***function name*** to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python.
- ***Parameters (arguments)*** through which we pass values to a function. They are optional.
- A ***colon (:)*** to mark the end of the function header.
- documentation string (***docstring***) to describe what the function does which is optional.
- One or more valid python ***statements*** that make up the function body. Statements must have the same indentation level (usually 4 spaces).
- ***return*** statement to return a value from the function which is optional.

Example of a function:

```
def greet(name): """
This function greets to the person passed in as a parameter
"""
print("Hello, " + name + ". Good morning!")
```

To call a function in python:

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
>>> greet('Tirumala')
Hello, Tirumala. Good morning!
```

<u>Note:</u> Try running the above code in the Python program with the function definition to see the output.

<u>Docstrings:</u>

- The first string after the function header is called the docstring and is short for documentation string. It is briefly used to explain what a function does.
- Python docstrings are the string literals that appear after the definition of a method, class, or module also.
- In the above example, we have a docstring immediately below the function header. We generally use triple quotes so that docstring can extend up to multiple lines.
- We can access these doc strings using the __**doc**____attribute.

<u>Example: 1</u>

```
>>> print(greet._doc__)

This function greets to the person passed in as a parameter
```

<u>Example 2:</u>

```
def square(n):
‘’’Takes in a number n, returns the square of n’’’ return n**2
print(square._doc_)
```

<u>Output:</u>

```
Takes in a number n, returns the square of n
```

<u>Example 3:</u> Doc strings for the built-in print() function

```
print(print._doc_)
```

<u>Output:</u>

```
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
Prints the values to a stream, or to sys.stdout by default.
```

Optional keyword arguments:
file: a file-like object (stream); defaults to the current sys .std out. sep: string inserted between values, default a space.
end: string appended after the last value, default a newline. flush: whether to forcibly flush the stream.

Doc strings for Python Modules:

- The doc strings for Python Modules should list all the available classes, functions, objects and exceptions that are imported when the module is imported.
- They should also have a one-line summary for each item.
- They are written at the beginning of the Python file.

Example 4: doc strings for the built in module in Python called pickle.

```
import pickle print(pickle._doc_)
```

Doc strings for Python Classes:

- The doc strings for classes should summarize its behavior and list the public methods and instance variables.
- The subclasses, constructors, and methods should each have their own doc strings.

Example 5: Doc strings for Python class. class Person:

```
""""
A class to represent a person.
...
Attributes
--------
name : str
first name of the person age : int
age of the person """
Methods
-------
info(additional=""):
Prints the person's name and age. """
```

def info(self, additional=""):
"""
Prints the person's name and age.
If the argument 'additional' is passed, then it is appended after the main info.

Output:
>>> print(Person._doc_)

Using the help() Function for Doc strings

- We can also use the help() function to read the doc strings associated with various objects.
- Here, we can see that the help() function retrieves the doc strings of the Person class along with the methods associated with that class.

Example 6: Read Doc strings with the help() function
>>> help(Person)

The return statement:

- The return statement is used to exit a function and go back to the place from where it was called.
- This statement can contain an expression that gets evaluated and the value is returned.
- If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.

Syntax of return: return [expression _list] Example:1
>>> print(greet("May")) Hello, May. Good morning! None
Here, None is the returned value since greet() directly prints the name and no return statement is used.
Example: 2
def absolute_ value(num):
"""This function returns the absolute value of the entered number""" if num >= 0:
return num else:
return -num print(absolute_ value(2)) print(absolute_ value(-4))
Output:
2
4

Types of Functions:

Basically, we can divide functions into the following two types:

1. ***Built-in functions*** - Python has several functions that are readily available for use. These functions are called built-in functions.

   Examples:
   abs(),any(),all(),ascii(),bin(),bool(),callable(),chr(),compile(),classmethod(),delattr(),dir(), divmod(),stat
   icmethod(),filter(),getattr(),globals(),exec(),hasattr(),hash(),isinstance(),issubclass(),iter( ),locals(),
   map(), next(),memoryview(),object(),property(),repr(),reversed(),vars(), import (),super()

2. ***User-defined functions -***
   - Functions that we define ourselves to do certain specific task are referred as user-defined functions.
   - If we use functions written by others in the form of library, it can be termed as library functions.

   - All the other functions that we write on our own fall under user-defined functions. So, our user- defined function could be a library function to someone else.

Advantages of user-defined functions:

- User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
- If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
- Programmers working on large project can divide the workload by making different functions. Example:

```
def add_ numbers (x ,y):
     sum = x + y return sum
     num1 = 5
      num2 = 6
print("The sum is", add _numbers(num1, num2))
```

Output:

```
The sum is 11
```

Python Function Arguments:

- In Python, you can define a function that takes variable number of arguments.

Example:

```
def greet(name, msg):
        """This function greets to
        the person with the provided message"""
        print("Hello", name + ', ' + msg)
greet("Titumala", "Good morning!")
```

Output:

```
Hello Tirumala, Good morning!
```

- Here, the function greet() has two parameters. Since we have called this function with two arguments, it runs smoothly and we do not get any error.

- If we call it with a different number of arguments, the interpreter will show an error message.
- If we call to this function with one and no arguments along with their respective error messages.

```
>>> greet("Monica")          # only one argument
Type Error: greet() missing 1 required positional argument: 'msg'

>>> greet()    # no arguments
Type Error: greet() missing 2 required positional arguments: 'name' and 'msg'
```

- Three different forms of Function Arguments are there. they are below.

1. Default Arguments.
2. Keyword Arguments.
3. Arbitrary Arguments/Variable Length Arguments.

1. Default Arguments:

Function arguments can have default values. We can provide a default value to an argument by using the assignment operator (=).

Example:

```
def greet(name, msg="Good morning!"):
        print("Hello", name + ', ' + msg)

greet("Tirumala")
greet("CSE", "Welcome to Technical Wing.")
```

Output:

```
Hello Tirumala, Good morning!
Hello CSE, Welcome to Technical Wing.
```

Explanation:

- The parameter **name** does not have a default value and is required (mandatory) during a call.
- The parameter **msg** has a default value of **"Good morning!".** So, it is optional during a call. If a value is provided, it will overwrite the default value.
- Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.
- This means to say, non-default arguments cannot follow default arguments. Example:

```
def greet(msg = "Good morning!", name):
```

Output: Syntax Error: non-default argument follows default argument.

2. Keyword Arguments:

- Python allows functions to be called using keyword arguments. When we call functions in this way, the order (position) of the arguments can be changed.
- Following calls to the above function are all valid and produce the same result.

```
# 2 keyword arguments
greet(name = "TEC" ,msg = "Good Morning")

# 2 keyword arguments (out of order)

greet(msg = "Good Morning" ,name = "TEC")

1 positional, 1 keyword argument greet("TEC", msg = "Good Morning")
```

Output: Hello TEC, Good Morning

- Having a positional argument after keyword arguments will result in errors.

  Example: greet(name="TEC","Good Morning”)

  Output: Syntax Error: non-keyword arg after keyword arg.

3. Arbitrary Arguments / Variable Length Arguments:

- Sometimes, we do not know in advance the number of arguments that will be passed into a function.
- Python allows us to handle this kind of situation through function calls with an arbitrary number of arguments.
- In the function definition, we use an asterisk (*) before the parameter name to denote this kind of argument.

Example:

```
def greet(*names):
# names is a tuple with arguments for name in names:
print("Hello", name)
greet("Monica", "Lakshmi" , "Sravan" , "Jasmin")
```

Output:

```
Hello Monica Hello Lakshmi Hello Sravan Hello Jasmin
```

Explanation: we have called the function with multiple arguments. These arguments get wrapped up into a tuple before being passed into the function. Inside the function, we use a for loop to retrieve all the arguments back.

variable length of keyword arguments

To accept variable length of keyword arguments i.e to create functions that take n number of keyword arguments we use **kwargs.

The **k wargs collects the password arguments into anew dictionary, where the argument names are the keys and their values are their key values

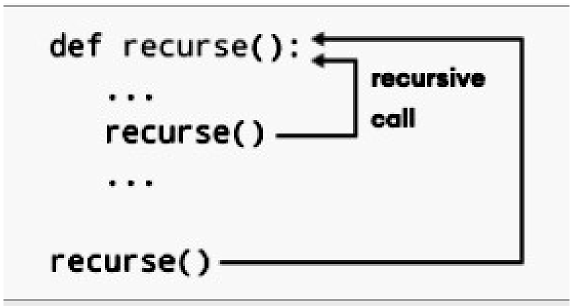
```
def  display(**k warg s):
        for i in kwargs:
              print(i)
   display(name=”raj”, age=20)
```

output

```
name
age
```

Recursive Function:

- A function that calls itself is known as Recursive Function.
- A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.
- The following image shows the working of a recursive function called ***recursive.***



```
def recurse():
    ...
    recurse()
    ...

recurse()
```

Recursive Function in Python

Example:
- Factorial of a number is the product of all the integers from 1 to that number.
- the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.

```
def factorial(x): if x == 1:
return 1 else:
return (x * factorial(x-1))

num = 3
print("The factorial of", num, "is", factorial(num))
```

Output:

```
The factorial of 3 is 6
```

This recursive call can be explained in the following steps.

factorial(3)	# 1st call with 3
3 * factorial(2)	# 2nd call with 2
3 * 2 * factorial(1)	# 3rd call with 1
3 * 2 * 1	# return from 3rd call as number=1
3 * 2	# return from 2nd call
6	# return from 1st call

Advantages of Recursion:

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion:

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

Python Anonymous/Lambda Function:

- In Python, an anonymous function is a function that is defined without a name.
- While normal functions are defined using the def keyword in Python, anonymous functions are defined using the lambda keyword.
- Hence, anonymous functions are also called lambda functions.
- Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

Syntax of Lambda Function: lambda arguments: expression

Example:

```
# Program to show the use of lambda functions
double = lambda x: x * 2
print(double(5))
```

Output:

```
10
```

Use of Lambda Function in python:

- We use lambda functions when we require a nameless function for a short period of time.
- In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like filter(), map() etc.

Example use with filter(): The filter() function in Python takes in a function and a list as arguments.

```
# Program to filter out only the even items from a list
my_ list = [1, 5, 4, 6, 8, 11, 3, 12]
New _list = list(filter(lambda x: (x%2 == 0) , my_ list))
 print(new _list)
```

Output:

```
[4, 6, 8, 12]
```

Example use with map(): The map() function in Python takes in a function and a list.

```
# Program to double each item in a list using map()
my _list = [1, 5, 4, 6, 8, 11, 3, 12]
new_ list = list(map(lambda x: x * 2 , my_ list))
print(new _list)
```

Output: [2, 10, 8, 12, 16, 22, 6, 24]

Scope and Lifetime of variables:

- Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.
- The lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.
- They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Example:

```
def my _fun():
x = 10
print("Value inside function:",x)

x = 20
my _fun()
print("Value outside function:",x)
```

Output:

```
Value inside function: 10 Value outside function: 20
```

Explanation:

- Here, we can see that the value of x is 20 initially. Even though the function my_ fun() changed the value of x to 10, it did not affect the value outside the function.
- This is because the variable x inside the function is different (local to the function) from the one outside. Although they have the same names, they are two different variables with different scopes.

Python Global, Local and Nonlocal variables

1. **Global Variables:** In Python, a variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.

   Example 1: Create a Global Variable

```
 x = "global"
def fun():
        print("x inside:", x)
fun()
print("x outside:", x)
```

   Output:

```
x inside: global x outside: global
```

➢ **What if you want to change the value of x inside a function?**

   Example 2:

```
x = "global"
def fun():
        x = x * 2
        print(x)
fun()
```

   Output:

```
Unbound Local Error: local variable 'x' referenced before assignment
```

   Explanation: The output shows an error because Python treats x as a local variable and x is also not defined inside fun(). To make this work, we use the global keyword.

   Example 3:   Changing Global Variable from inside a Function using

```
global c = 0   # global variable
def add():
         global c
        c = c + 2        # increment by 2 print("Inside add():", c)

print("Before In main:", c)
add()
print("In main:", c)
```

   Output:

```
Before In main: 0
Inside add(): 2
In main: 2
```

2. **Local Variables:** A variable declared inside the function's body or in the local scope is known as a local variable.

   Example 1: Accessing local variable outside the scope

```
def fun():
        y = "local"
 fun()
print(y)
```

   Output:

```
Name Error: name 'y' is not defined
```

   Explanation: The output shows an error because we are trying to access a local variable y in a global scope whereas the local variable only works inside fun() or local scope.

Advantages of functions:

- User-defined functions are reusable code blocks; they
- Only need to be written once ,then they can be used multiple times.
- These functions are very useful, from writing common utilities to specific business logic.
- Thecodeisusuallywellorganized,easytomaintain,anddeveloper-friendly.
- A function can haved if type so arguments& return value.

Example:1

```
Def disp():
        print("hi sri vidya")
        print("hi students")
disp()          #function calling
```

Example:2

To specify no body of the function use pass statement.

```
Def d isp():
  Pass disp()
```

Example:3

One function is able to call more than one function.

```
>>>def happy Birthday(person):print("Happy Birthday dear" ,person)

>>>def Mohan():
         Happy Birthday('CSE')happy Birthday('IT')
>>>Mohan()
```

Inner functions: A function can be created as an inner function in order to protect it from everything that is happening outside of the function. In that case, the function will be hidden from the global scope.

Example:1

```
def function1(): # outer function
print("Hello from outer function")
```

```
def function2():# inner function
        print("Hello from inner function")
        function2()

>>>function1()
```

Output:

```
Hello from outer function
Hello from inner function
```

Explanation:

In the above example, function2 () has been defined inside function1(), making it an inner function. To call function2(), we must first call function1(). The function1() will then go ahead and call function2()as it has been defined inside it.

It is important to mention that the outer function has to be called in order for the inner function to execute if the outer function is not called, the inner function will never execute.

Example:

- Inside the inner function to represent outer function variable use **non local keyword**.

```
def outer():
         var_ outer = 'TEC '
         print (var _outer)
         def inner():
            nonlocal var_ outer
            var_ outer="CSE&IT"
         inner()                    #calling of inner function print(var_ outer)

>>>outer()
```

Output:

```
TEC CSE&IT
```

Example:

- Inside the function to represent the global value use global keyword.

```
name='Tirumala'
def outer():
    var_ outer=' Eamcet  Code'
    def inner():
        non local var_outer
        var_outer="EamcetCode:TMLN"
    global name
    print(name)
    name="TEC"
    print(name)
    print(var_outer)
    inner()  #calling of inner function
    print(var_outer)


>>>outer()
```

Output:

```
Tirumala TEC
Eamcet Code Eamcet Code: TMLN
```