Table of Contents

- Random Module
- Platform Module
- OS Module
- Date & Time Module
- Calendar Module
- Copy Module

History of Python

Python was invented by **Guido van Rossum** in 1991 at CWI in Netherland. The idea of Python programming language has taken from the ABC programming language or we can say that ABC is a predecessor of Python language.

There is also a fact behind the choosing name Python. Guido van Rossum was a fan of the popular BBC comedy show of that time, **"Monty Python's Flying Circus"**. So he decided to pick the name **Python** for his newly created programming language.

Python has the vast community across the world and releases its version within the short period.

What is Python

Python is a general purpose, dynamic, high-level, and interpreted programming language. It supports Object Oriented programming approach to develop applications. It is simple and easy to learn and provides lots of high-level data structures.

Python is *easy to learn* yet powerful and versatile scripting language, which makes it attractive for Application Development.

Python's syntax and *dynamic typing* with its interpreted nature make it an ideal language for scripting and rapid application development.

Python supports *multiple programming pattern*, including object-oriented, imperative, and functional or procedural programming styles.

Python is not intended to work in a particular area, such as web programming. That is why it is known as *multipurpose* programming language because it can be used with web, enterprise, 3D CAD, etc.

We don't need to use data types to declare variable because it is *dynamically typed* so we can write a=10 to assign an integer value in an integer variable.

Why Python

Python provides many useful features to the programmer. These features make it most popular and widely used language. We have listed below few-essential feature of Python.

- Easy to use and Learn
- Interpreted Language
- Object-Oriented Language
- Open Source Language
- Extensible
- Learn Standard Library
- GUI Programming Support
- Integrated
- Embeddable
- Dynamic Memory Allocation
- Wide Range of Libraries and Frameworks

Where Python Can be Used

The various areas of Python use are given below.

- Data Science
- Date Mining
- Desktop Applications
- Console-based Applications
- Mobile Applications
- Software Development
- Artificial Intelligence
- Web Applications
- Enterprise Applications
- 3D CAD Applications
- Machine Learning
- Computer Vision or Image Processing Applications.
- Speech Recognitions

Who Uses Python?

1. YouTube
2. Google
3. DropBox
4. RospBerryPI
5. BitTorrent
6. NASA
7. NSA
8. NETFLIX
9. Yahoo
10. Honeywell , HP , Philips and United Space Alliance

Input and out function of Python

input():

Python has a built-in function named input () to read input data from the keyboard. Its syntax is as follows:

```
var = input(prompt)
```

This function first takes the input from the user and then evaluates the expression, which means Python automatically identifies whether user entered a string or a number or list. If the input provided is not correct then either syntax error or exception is raised by python.

```
# eg: Python program showing a use of input()

val = input ("Enter your value: ")
print(val)
```

How the input function works in Python :

1. When input() function executes program flow will be stopped until the user has given an input.
2. The text or message display on the output screen to ask a user to enter input value is optional i.e. the prompt, will be printed on the screen is optional.

3. Whatever you enter as input, input function convert it into a string. if you enter an integer value still input() function convert it into a string. You need to explicitly convert it into an integer in your code using typecasting.

Ex: Program to check input type in Python

```
num = input ("Enter number :")
print(num)
name1 = input("Enter name : ")
print(name1)
# Printing type of input value
print ("type of number", type(num))  print ("type of
name", type(name1))
```

print()

The print() function prints the specified message to the screen, or other standard output device. The message can be a string, or any other object, the object will be converted into a string before written to the screen.

Syntax:

```
print(value(s),sep=' ',end='\n', file=file, flush=flush)
```

Parameters:

value(s): Any value, and as many as you like. Will be converted to string before printed sep='separator': (Optional) Specify how to separate the objects, if there is more than one. Default:' ' end='end': (Optional) Specify what to print at the end. Default: '\n' file: (Optional) An object with a write method. Default: sys. stdout

flush: (Optional) A Boolean, specifying if the output is flushed (True) or buffered (False). Default: False

Python | sep parameter in print ()

The separator between the arguments to print () function in Python is space by default (softspace feature), which can be modified and can be made to any character, integer or

string as per our choice. The ‘sep’ parameter is used to achieve the same, it is found only in python 3.x or later. It is also used for formatting the output strings.

eg:

```
#code for disabling the softspace feature print ('k','i','r','a',‘n’, sep='')

#for formatting a date
print ('13','04,'2021', sep='-')

#another example
print ('KIRAN','VCUBE SOLUTIONS TECH JNTU', sep='@')
```

The sep parameter when used with end parameter it produces awesome results. Some examples by combining the sep and end parameter.

```
#n provides new line after printing the year print ('22','12', DEC='-',
end='-2018n')

print ('KIRANA','VCUBE SOLUTIONS', sep='', end='@') print('PYTHON')
```

Python | end parameter in print ()

By default, python’s print () function ends with a newline. A programmer with C/C++ background may wonder how to print without newline.

Python’s print () function comes with a parameter called ‘end’. By default, the value of this parameter is ‘\n’, i.e. the new line character. You can end a print statement with any character/string using this parameter.

```
# This Python program must be run with # ends the
output with a <space>  print ("Welcome to”, end = ' ')
print ("VCUBE SOLUTIONS TECH", end = ' ')
print ("Python”, end = '@')
print ("VCUBE SOLUTIONS TECH")
```

Python Comments:

Writing comments for our code is useful for programmers to add **notes** or **explanations** of the code. Commentes are just for reading and python interpreter ignores whatever we write in comments.

Python supports two types of comments.

1 Single lined comment.
2 Multi lined Comment.

Single line comment:

In Python, we use the hash (#) symbol to start writing a comment. If developer want to only specify one line comment than use single line comment, then comment must start with #. The symbol(#) indicates the **starting** of the comment and everything written after the # considered as **comment**.

```
# This is single line comment. (hash or pound)
#print("vcube solutions ")

#This is a comment
#print out the name of the cityo print('Hyderabad')
```

Multi-line comments:

If we have comments that extend multiple lines, one way of doing it is to use hash (#) in the beginning of each line. For example:

```
#This is a vcube solutions
#near JNTU
#HYDERABAD
```

Another way of doing this is to use triple quotes

These triple quotes are generally used for multi-line strings. But they can be used as multiline comment as well. Unless they are not docstrings, they do not generate any extra code.

```
"""This is program was developed
 kiran kumar, senior python trainer in
Vcube solutions, near jntu , hyderabad"""

print("python")
print("Vcube solutions")
print("near jntu hyd")"""
```

Data Types in Python

In programming, data type is an important concept.

Variables can store data of different types, and different types can do different things.

Python has the following data types built-in by default, in these categories:

Text Type:	Str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	Dict
Set Types:	set, frozenset
Boolean Type:	Bool
Binary Types:	bytes, bytearray, memoryview

1. int datatype:

The int datatype represents an integer number. an integer number is a number without any decimal point.

Ex-1: 10,20,0,-34...etc are treated as integer number.

Ex-2: a=25

Here, a is called as integer variable, since it is storing 25 value.

In python, there is no limit for the size of an integer datatype. It can store very large integer numbers conveniently.

We can represent int values in the following ways

```
```

- Decimal form

```
````````````
```

- Binary form
- Octal form
- Hexa decimal form
```

1. Decimal form(base-10):

It is the default number system in Python.

The allowed digits are: 0 to 9.

The total number of digits are 10. So, its base value is '10'

Ex: a =10

2. Binary form(Base-2):

The allowed digits are : 0 & 1

Binary number start with either 0b or 0B.

The total number of digits are 2. So its base value is '2'

Ex:

x = 0B1111

y = 0B10110101 z = 0b1110

3. Octal Form(Base-8):

The allowed digits are : 0 to 7 octal number

start with either with 0o or 0O.

The total number of digits are 8. So its base value is '8'

Ex:

x=0o123 y=0o736

4. Hexa-decimal Form(Base-16):

The allowed digits are : 0 to 9, a-f (both lower and upper cases are allowed)

Hexa decimal number Literal value should be prefixed with 0x or 0X

The total number of digits are 16. So its base value is '16'

Ex:

a=0XFACE B=0XBeef

c=0XBeer

Note: Being a programmer we can specify literal values in decimal, binary, octal and hexa-decimal forms. But PVM will always provide values only in decimal form.

a=10

b=0o10

c=0X10

```
d=0B10
print(a)        #10
print(b)        #8
print(c)        #16
print(d)        #2
```

2. float data type:

The float datatype represents floating point numbers. The floating point number is a number that contains a decimal point.

Ex: 10.0, 20,0, 45.7,-34.8...etc.

3. Complex Datatype:

Complex number is a number that is written in the form of a+bj form.

Here 'a' represents real part and 'b' represent imaginary part.

Complex numbers are much in use in Python programming.

Ex:

```
a=4+8j
print(a)        print(type(a)
```

Even we can perform operations on complex type values.

Ex:

```
x=6+2j
y=3+8j          z=x+y
print(z)
print(type(z))
```

4. boolean:

The bool datatype in python represents boolean values. There are mainly two boolean values True and False. Python internally True represented by 1 and False represented by 0. The type of the built-in values True and False. Useful in conditional expressions, and anywhere else you want to represent the truth or false of some condition. Mostly interchangeable with the integers 1 and 0. A blank string like "" is also represented as False.

Ex-1:

```
b=True          type(b) =>bool
```

Ex-2:

```
a=10
b=20          c=a<b
print(c)==>True
```

Ex-3:

```
True+True==>2
True-False==>1
```

Ex 4:

```
x=True          y=False
z=x+y
print(z)
```

5. Strings:

Strings, one of the core data types in Python are used to record textual information as well as arbitrary collections of bytes.

Python allows strings to be enclosed in single or double quote characters and it also has a multiline string literal form enclosed in triple quotes (single or double). When this form is used, all the lines are concatenated together, and end-of-line characters are added where line breaks appear.

Ex:

```
s1= "python is a high level language"          print(s1)
print(type(s1))
```

6. **Bytes:**

bytes data type represents a group of byte numbers just like an array.

Ex:

```
x=[11,22,33,44]        b=bytes(x)
print(b)         print(type(b))
for i in b:
    print(i)
```

Note 1 : bytes object does not supprot item assignment.

```
b[0]=100  # error , we cannot change its content
```

7. **bytearray:**

bytearray is exactyly same as bytes data type except that its elements can be modified.

Ex:

```
x=[11,22,33,44]      b=bytes(x)
print(b)
print(type(b))
for i in b:
  print(i)
b[0]=999
for i in b:
  print(i)
```

Type Casting Functions

To convert a variable value from one type to another type. This conversion is called Typecasting or Type conversion.

Function	Description
int(x)	Converts x to an integer.
float(x)	Converts x to a floating-point number.

str(x)	Converts object x to a string representation.
complex(l)	Creates a complex number
bool( )	Converts othe type to bool type
list(s)	Converts s to a list.
tuple(s)	Converts s to a tuple.
set(s)	Converts s to a set.
dict(d)	Creates a dictionary. d must be a sequence of (key,value) tuples.
ord(x)	Converts a single character to its integer value.
hex(x)	Converts an integer to a hexadecimal string.
oct(x	Converts an integer to an octal string.
bin(x)	Converts an integer to binary type

• **int( ):** We can use this function to convert values from other types to int

Ex:
int(123.7891234)#123
int(10+5j)
TypeError: can't convert complex to int int(True) #
1
int(False) # 0
int("10") # 10 int("10.5")
ValueError: invalid literal for int() with base 10: '10.5' int("ten")
ValueError: invalid literal for int() with base 10: 'ten' int("0B1111")
ValueError: invalid literal for int() with base 10: '0B1111' Note:

1. We can convert from any type to int except complex type.
2. If we want to convert str type to int type, compulsary str should contain only integral value and should be specified in base-10

• **float ( ):** We can use float() function to convert other type values to float type.

Ex:
float(10) 10.0
float(10+5j)

TypeError: can't convert complex to float float(True)

```
#    1.0
float(False)     #     0.0
float("10")      #  10.0
float("10.5")    #    10.5
float("ten")
```

ValueError: could not convert string to float: 'ten' float("0B1111")

ValueError: could not convert string to float: '0B1111'

Note:

1. We can convert any type value to float type except complex type.
2. Whenever we are trying to convert str type to float type compulsary str should be either integral or floating point literal and should be specified only in base-10.

- **Complex ( ):**

We can use complex() function to convert other types to complex type.

Form-1: complex(x)

We can use this function to convert x into complex number with real part x and imaginary part 0.

Ex:

```
complex(10)==>10+0j
complex(10.5)===>10.5+0j
complex(True)==>1+0j
complex(False)==>0j
complex("10")==>10+0j
complex("10.5")=>10.5+0j
complex("ten")
```

ValueError: complex() arg is a malformed string

Form-2: complex(x,y)

We can use this method to convert x and y into complex number such that x will be real part and y will be imaginary part.

Eg: complex(10,-2)==>10-2j

complex(True,False)==>1+0j

- **bool( ):** We can use this function to convert other type values to bool type.

Ex:

1) bool(0)==>False

2) bool(1)==>True
3) bool(10)===>True
4) bool(10.5)===>True
5) bool(0.178)==>True
6) bool(0.0)==>False
7) bool(10-2j)==>True
8) bool(0+1.5j)==>True
9) bool(0+0j)==>False
10) bool("True")==>True
11) bool("False")==>True
12) bool("")==>False

- **str( ):** We can use this method to convert other type values to str type

Ex:

1) str(10)
2) '10'
3) str(10.5)
4) '10.5'
5) str(10+5j)
6) '(10+5j)'
7) str(True) 8) 'True' type conversion in python by exanple:

```
v1 = int(2.7) # 2 v2 = int(-3.9) # -
3 v3 = int("2") # 2
v4 = int("11", 16) # 17, base 16 v5 = long(2) v6 =
float(2) # 2.0 v7 = float("2.7") # 2.7 v8 =
float("2.7E-2") # 0.027 v9 = float(False) # 0.0 vA =
float(True) # 1.0 vB = str(4.5) # "4.5" vC = str([1, 3,
5]) # "[1, 3, 5]"
vD = bool(0) # False;bool fn since Python 2.2.1 vE = bool(3) # True
vF = bool([]) # False - empty list vG = bool([False]) # True - non-empty
list vH = bool({})#False-empty dict;same for empty
tuple vI = bool("") # False - empty string vJ = bool(" ") # True -
non-empty string vK = bool(None) # False vL = bool(len) #
True vM = set([1, 2]) vN = list(vM)
vO= list({1: "a", 2: "b"})#dict -> list of keys vP = tuple(vN)
vQ = list("abc") # ['a', 'b', 'c']
print v1, v2, v3, type(v1), type(v2), type(v3)
```

Operators in Python

An operator is a mathematical symbol, that represents an action.

Operators are the constructs, which can manipulate the value of operands.

Consider the expression 4 + 5 = 9.

Here, 4 and 5 are called the operands and + is called the operator.

Python Supports 6 types of Operators:

1. Arithmetical Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Special operators
    a). Identity opertors

    b). Membership operators

- **Arithmetical Operators:**

Arithmetic operators are used to perform arithmentic operations such as addition, minus,muliplication....and so on.

Operator	Meaning
+	Addition operator
-	Subtraction operator
*	Multiplication operator
/	Division operator
%	Modulus or remainder operator
**	Exponent or power operator

// Floor division operator

Ex:

```
x = 10
y = 20
print('x + y =',x+y)
print('x - y =',x-y)
print('x * y =',x*y)
print('x / y =',x/y)
print('x // y ',x//y)
print('x ** y =',x**y)
```

- **Relational Operators:**

These operators are used to compare the values, and decide the relation among them. They are also called Relational operators.

Operator Description

== If the values of two operands are equal, then the condition becomes true.

!= If values of two operands are not equal, then condition becomes true.

> If the value of left operand is greater than the value of right operand, then condition becomes true.

< If the value of left operand is less than the value of right operand, then condition becomes true.

>= If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.

<= If the value of left operand is less than or equal to the value of right operand, then condition becomes true.

Ex:

```
x = 10
y = 20
print('x > y  is',x>y)
print('x < y  is',x<y)
print('x == y is',x==y)
print('x != y is',x!=y)
print('x >= y is',x>=y)
print('x <= y is',x<=y)
```

- **Logical(Boolean) Operators:**

These operators are used to combine two or more relational expressions.
Python supports the following logical operators.

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

Ex:

```
x = True
y = False
print('x and y is',x and y)
print('x or y is',x or y)
print('not x is',not x)
```

- **Bitwise Operators:**

These are used to perform bit operations. All the decimal values will be converted into binary values and bitwise operators will work on these bits such as shifting them left to right or converting bit value from 0 to 1 etc.

OPERATOR	MEANING

&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows −

a = 0011 1100

b = 0000 1101

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

<< Binary Left Shift: The left operand's value is moved left by the number of bits specified by the right operand.

Ex: a << 2 = 240 (means 1111 0000)

>> Binary Right Shift: The left operand's value is moved right by the number of bits specified by the right operand.

Ex: a >> 2 = 15 (means 0000 1111)

- **Assignment Operators:**

Assignment operators are used in Python to assign values to variables.

a = 5 is a simple assignment operator that assigns the value 5 on the right to the variable 'a' on the left.

We can combine asignment operator with some other operator to form compound

here are various compound operators in Python like a += 5 that adds to the variable and later assigns the same. It is equivalent to a = a + 5. assignment operators in python are:

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

Write a program to demonstrate arithmetic assignment operator

```
m=8
print(m)
m+=35
print(m)
m-=3
print(m)
m*=100
print(m)
m/=3
print(m)
m%=5
print(m)
m//=5
print(m)
m**=3
print(m)
```

write a program to demonstrate bitwise assignment operator m=10

```
print(m)
m&=5
print(m)
m|=3
print(m)
m^=5
print(m)
m<<=3
print(m)
m>>=5
print(m)
```

- **Special Operators:**

Python Language offers some special type of operators:

1. Membership operators
2. Identity operators

1. **Membership Operators:**

They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

Operator	**Meaning**
in	True if value/variable is found in the sequence not
in	True if value/variable is not found in the sequence **Eg:**

Ex:

```
x = 'Hello world'
y = {1:'a',2:'b'} print('H' in x)
print('hello' not in x) print(1 in y)
print('a' in y)
```

2. **Identity Operators:**

They are used to check if two values (or variables) are located on the same part of the memory. Identity operators compare the memory locations of two objects.

Operator	Meaning
Is	True if the operands are identical
is not	True if the operands are not identical

Ex:

```
x = "Hello"
y = "hello" x1=512
y1=215
print('x is y is',x is y)
print('x is not y is',x is not y) print('x1 is y1',x1 is y1)
print('x1 is not y1',x1 is not y1)
```

Walrus operator :

1. In python 3.8 introduces a new operator := is called a walrus operator.
2. This operator allows us to assign a vlaue to a variable inside the python expression.
3. It is a convenient operator which makes our code more compact.
4. It is a new way to assign the varaibles within an expression using the := notation
5. The syntax is :

```
variablename:=expression
```

Ex-1:

```
print(a:=10)
```

to assign value 10 to variable 'a' and print it.
without walrus operator , we have to create two lines:

```
a=10
        print(a)
```

Ex-2: walrus operator with if statement:

```
if x:=10 > 5:
   print('x value is above 5')
else:
   print('x value is below 5')
```

Ex-3: walrus operator with while loop:

```
words=[]
while (word:=input('enter any word'))!='quit':
   words.append(word)
print(words)
```

Ex-4: walrus operator with for loop using walrus operator:

```
languages=['python','cpp','pascal','java','fortran'] for i in languages:
if (n:=len(i))<5:
      print('warning,the language{}has{}\ characters'.format(i,n))
```

without using walrus operator:

```
languages=['python','cpp','pascal','java','fortran']
for i in languages:
n=len(i)        if n<5:
      print('warning, the language {} has {} characters'.format(i,n))
```

Order of Operations:

When an expression contains more than one operator, the order of evaluation depends on the order of operations. For mathematical operators, Python follows mathematical convention. The acronym PEMDAS is a useful way.

PENDAS BADMAS

Parentheses Exponentiation Multiplication Division Addition Subtraction
Parentheses

2 * (3-1) ==> 4
(1+1)**(5-2) ==> 8
Exponentiation

1 + 2**3 ==> 9, not 27, 2 * 3**2 ==>
18, not 36.
Multiplication and Division have higher precedence than Addition and Subtraction.

2*3-1 ==> 5, not 4, 6+4/2 ==> 8, not 5.

Decision Making Statements

- **Conditional control Statements**

Decision making is required when we want to execute a code only if a certain condition is satisfied. The if…elif…else statement is used in Python for decision making.

The conditional statements are

a. if statement
b. if-else statement
c. elif-ladder statement
d. nested if statement

- **if statement:**

It is also called simple if statement. The syntax is:

```
if  condition:
        Statement
```

(or)

```
if condition:
        statement-1
        statement-2
        statement-3
```

The given condition is True then statements will be executed. Otherwise statements will not be executed.

Ex-1:

```
if (a<b):
        print("a is smallest")
```

Ex-2:

```
if (course=="python"):
        print("Dear, u are joined python course")
```

- **if-else:**

The synatx is as follows:

```
if condition :
statement-1
else :
statement-2
```

The given condition is True then statement-1 will be executed and statement-2 will not be executed.

The given condition is False, then statement-1 is not executed and statement-2 is executed.

Ex-1:

```
if (a<b):
    print ("a is smallest")
else:
    print ("b is smallest")
```

Ex-2:

```
if (course=="python"):
    print ("u are joined python course")
else:
    print("u are joined other course")
```

- **if-elif statement:**

Syntax:

```
if condition-1:
    statement-1
elif condition-2:
    statement-2
elif condition-3:
    statement-3
```

```
        elif condition-4:
                statement-4
.               - -  - -
              - - - - -
        elif codition-n
                statement-n
```

Based on the condition the corresponding statements are executed

- **if-elif-else:**

The elif statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

Similar to the else, the elif statement is optional. However, unlike else, for which there can be at most one statement, there can be an arbitrary number of elif statements following an if.

Syntax:

```
        if  condition-1:
                statement-1
        elif condition-2:
                statement-2
        elif condition-3:
                statement-3
        elif condition-4:
                statement-4
.               - -  - -
              - - - - -
        elif codition-n
```

```
        statement-n
else:
        else statements.
```

Ex:

```
var = 100
if var == 200:
        print (var)
elif var == 150:
        print (var)

elif var == 100:
        print( var)
else:
        print (var)
print("Good")
```

based on the condition the corresponding statements are executed. if all the conditions are false, then only the else is executed.

- **Python Nested if statements:**

We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming.

Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. This can get confusing, so must be avoided if we can.

```
num = int(input("Enter a number: "))
if num >= 0:
        if num == 0:
            print("Zero")
        else:
             print("Positive number")
else:
        print("Negative  number")
```

Loop Control Structures

Loops are used in programming to repeat iterations. A loop is a sequence of instructions that is continually repeated until a certain condition is reached.

Why Loop?

In a loop structure, the loop asks a question. If the answer requires an action, it is executed. The same question is asked again and again until no further action is required. Each time the question is asked is called an iteration.

Repetative execution of the same block of code over and over is referred to as iteration. There are two types of iteration.

1. **Definite Iteration** : In which the number of repetitions is specified explicityly in advance. In python , definite iteration is performed with a *for* loop.

2. **Indefinite Iteration**: In which the code block executes until some condition is met. In python , indefinite iteration is performed with a *whie* loop.

Looping statements are used to execute a set of statements repeatedly. Python supports 2 types of iterative statements.

a) while loop
b) for loop

a) <u>while loop</u>:

If we want to execute a group of statements iteratively until some condition false, then we should go for while loop.

The while loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed as long as the given condition is true.

The while loop is mostly used in the case where the number of iterations is not known in advance. The while loop implements indefinite iteration, where the number of times the loop will be executed is not specified explicitly in advance.

Syntax: while (condition) :
    body of the while loop

Ex-1: To print numbers from 1 to 10 by using while loop

```
x=1
```

```
while x <= 10:                    print(x)
    x=x+1
```

Ex-2: To display the sum of first n numbers

```
n=int(input("Enter number:"))
sum=0  i=1
whilei<=n:
    sum=sum+i
    ii=i+1
print("The sum of first",n,"numbers is :",sum)
```

Infinite while loop

If the condition given in the while loop never becomes false then the while loop will never terminate and result into the infinite while loop.

Any non-zero value in the while loop indicates an always-true condition whereas 0 indicates the always-false condition. This type of approach is useful if we want our program to run continuously in the loop without any disturbance.

Ex:

```
while (1):
    print("Hi! we are inside the infinite while loop")
```

***while loop with else*:**

python allows us an optional ***else*** clause at the end of a while loop. This is a unique feature of python, not found in other programming languages.

The syntax is shown below:

```
while  condition :
    body of the while loop

else:

    body of the else block
```

Python enables us to use the while loop with the while loop also. The else block is executed when the condition given in the while statement becomes false. Like for loop, if the while loop is broken using break statement, then the else block will not be executed and the statement present after else block will be executed.

Ex:

```
i=1;        while i<=5:
                print(i)        i=i+1
else:
                        print("The while loop exhausted")
```

b) <u>for loop</u>:

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

Syntax:

```
for var in sequence:

    Body of for loop
```

Here, var is the variable that takes the value of the item inside the sequence on each iteration.

Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

Body will be executed for every element present in the sequence.

Ex-1: To print Hello 10 times

```
for x in range(10) :
        print("Hello")
```

Ex-2: To display numbers from 0 to 10

```
for x  in range(0,11,1):
    print(x)
```

In python, in for loop the default starting value is 0 and ending value is ' stop-1'.

Ex- 3: To display numbers from 0 to 10 for x in range(11)
:

```
        print(x)
```

Ex-4: To display odd numbers from 0 to 20 for x in range(21)
:

```
        if (x%2!=0):
            print(x)
```

<u>for loop with else</u>

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts.

The syntax is shown below:

```
for var in sequence:
    body of the for loop
else:
    body of the else block
```

break statement can be used to stop a for loop. In such case, the else part is ignored. Hence, a for loop's else part runs if no break occurs.

Eg:

```
digits = [0, 1, 5]
 for i in digits:     print(i)
else:
    print ("No items left.")
```

Nested Loops:

Python programming language allows to use one loop inside another loop. It is called nested loops.

1. nested for loop
2. nested while loop

nested for loop:

Syntax:

```
for iterator in sequence:

        for iter in sequence:

    statementes

statements(s)
```

Ex:

```
for i in range(1, 5):                    for j in range(1,5):
    print(i, end=' ')
print()
```

1) **nested while loop:**

The syntax for a nested while loop statement in Python programming language is as follows:

```
while condition:
    while expression:
        statements
```

statement(s)

A final note on loop nesting is that we can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

Ex:

```
i=1
while i<=5:
        j=1
        while j<=i:
            print(i,j)
                j=j+1
        print()
        i=i+1
```

Control Statements

break statement:

1. It is a keyword.
2. The break statement in python is used to exit from the loop based on some condition.
3. This statement can be used in while loop or for loop 4. The syntax of break statement is as follows: break.

we can use break statement inside loops to break loop execution based on some condition.

Ex-1:

```
for i in range(10):
    if i==7:
        print("processing is enough..plz break")
        break
    print(i)
```

Ex-2:

```
x='python'
for i in x:
    if i=='h:
        break
print(i)
```

Ex-3:

```
x=[11,22,33,44,55]
for i in x:
    if i==44:
        break
print(i)
```

2 Continue statement:

1. It is a keyword.
2. The continue statement in python is used to bring the program control to the beginning of the loop.
3. The continue statement skips the remaining lines of code inside the loop and start with the next iteration.

4. It is mainly used for a particular condition inside the loop so that we can skip some specific code...
5. This statement can be used in while loop or for loop
6. The syntax of continue statement is as follows:

```
continue
```

Ex-1: To print odd numbers in the range 0 to 9

```
for i in range(10):
    if i%2==0:
        continue
    print(i)
```

Ex-2:

```
x='python'
for i in x:
    if i=='h':
        continue
    print(i)
```

Ex-3:

```
x=[11,22,33,44,55]      for i in x:
   if i==44:
      continue
   print(i)
```

3 pass statement:

pass is a keyword in Python.

In our programming syntactically if block is required which won't do anything then we can define that empty block with pass keyword. pass

|- It is an empty statement

|- It is null statement

|- It won't do anything

Ex-1:

```
def display():
    pass
display()
```

Ex-2:

```
While True:
    Pass
```

Ex-3:

```
if True:
    pass
```

Sequential Data Types

LIST Data Structure

The important characterstics of python lists are as follows:

1) List objects are created by using square brackets ([ ]), or by using list() Function. Within the list each element is separated by commas.
2) Lists are ordered.
3) A list is an instrumental data structure in python. We can put multiple data types in a list. The list can have float values, integer values, and string values in it.
4) We can also put multiple lists in a list(Nested lists).
5) Insertion order is preserved.
6) Duplicate elements are allowed.
7) Heterogeneous elements are allowed.
8) List is a dynamic because based on our requirement we can increase the size and decrease the size.
9) Every element in the list is represented by with unique index number.
10) Hence index will play very important role.
11) Python supports both positive and negative indexes.
    a. where as
        i. positive index means from left to right.
        ii. negative index means right to left iii. [10,"A","B",20, 30, 10]
12) List objects are mutable. i.e., we can change the content.
13) Insertions are allowed.
14) Deletion are allowed.

Creation of List Objects:

1. We can create empty list object as follows...
   A list without any element is called an empty list

```
list= [ ]
print(list)
print(type(list))
```

2. If we know elements already then, we can create list as follows

```
list= [10,20,30,40]          print(list)
print(type(list))
```

3. With dynamic input:

```
list=eval(input("Enter List:"))
print(list)
print(type(list))
```

4. With list( ) function:

```
l=list(range(0,10,2))          print(l)
print(type(l))
```

5. with split() function:

```
l="Learning Python is very very easy !!!"
l1=l.split()                   print((l1)
print(type(l1))
```

Accessing elements of List:

We can access elements of the list either by using index or by using slice operator (:)

- **By using index:**

  List follows zero based index. ie index of first element is zero.

  List supports both +ve and -ve indexes.

  +ve index meant for Left to Right

  -ve index meant for Right to Left

```
list=[10,20,30,40]
print(list[0]) ==>10
print(list[-1]) ==>40
print(list[10]) ==>IndexError: list index out of range
```

- **By using slice operator:**

  The **' : '** operator is called a slice operator**.**

  This operator is used to access morethan one element at a time.

  Syntax:

```
list2= list1[start:stop:step]
```

start : It indicates the starting index number.
where slice has to start, default value is '0'.
stop : It indicates the Ending index number.
where slice has to end default value is max allowed index of list ie length of the list
step : Increment value, default value is 1.

Ex:

```
n=[1,2,3,4,5,6,7,8,9,10]
print(n[2:7:2])
print(n[4::2])
print(n[3:7])
print(n[8:2:-2])
print(n[4:100])
```

List vs mutability:

Once we creates a List object,we can modify its content.

Hence List objects are mutable.

Ex:

```
n=[10,20,30,40] print(n)
n[1]=888 print(n)
```

Traversing the elements of List:

The sequential access of each element in the list is called traversal.

1. By using while loop:

```
n=[0,1,2,3,4,5,6,7,8,9,10]  i=0
while i<len(n):
    print(n[i])
    i=i+1
```

2. By using for loop:

```
n=[0,1,2,3,4,5,6,7,8,9,10]
for n1 in n:
    print(n1)
```

3. To display only even numbers:

```
n=[0,1,2,3,4,5,6,7,8,9,10]  for n1 in n:
  if n1%2==0:
```

```
print(n1)
```

4. To display elements by index wise:

```
l=["A","B","C"]

x=len(l)
for i in range(x):
    print(l[i],"is available at positive indeg: ",i,"and at negative indeg: ",i-x)
```

Built-in methods in lists:

1. **len( ):** This function returns the number of elements present in the given list. Ex:

```
x=[10,20,30,40]
print(len(x))==>4
```

2. **max( ):** This function returns the highest element present in the given list Ex:

```
x=[10,20,30,40]
print(max(x))==>40
```

3. **min( ):** This function returns the lowest element present in the given list eg:

```
x=[10,20,30,40]
print(min(x))==>10
```

4. **sum( ):**This function returns the sum of all elements present in the given list Ex:

```
x=[10,20,30,40]
print(sum(x))==>100
```

5. **count( ):**It returns the number of times present a given element in a list.

Ex:

```
n=[1,2,2,2,2,3,3]
print(n.count(1))
print(n.count(2))
print(n.count(3))
print(n.count(4))
```

6. **Index( ):** This function returns the index of first occurrence of the specified item.

Ex:

```
n=[1,2,2,2,2,3,3]
```

print(n.index(1)) ==>0 print(n.index(2)) ==>1
print(n.index(3)) ==>5
print(n.index(4)) ==>ValueError: 4 is not in list

Note: If the specified element not present in the list then we will get ValueError. Hence before index () method we have to check whether item present in the list or not by using in operator. print(4 in n)==> False

Inserting Elements into a list:

7.append( ): This function is used to add new element at the end of the list.

Ex:

```
list=[ ]
list.append("A")  list.append("B")
list.append("C")
print(list) ff
```

Ex: To add all elements to list upto 100 which are divisible by 10

```
list=[]
for i in range(101):
    if i%10==0:
        list.append(i)
print(list)
```

8.insert( ) : This function is used to insert new element at specified index position

Ex:

```
n=[1,2,3,4,5]
n.insert(1,888) print(n)
```

Ex:

```
n=[1,2,3,4,5]
n.insert(10,777)
n.insert(-10,999) print(n)
```

Note: If the specified index is greater than max index then element will be inserted at last position. If the specified index is smaller than min index then element will be inserted at first position.

9.extend( ) :

This function is used to add all items of one list to another list**.**

Syntax:

```
l1.extend(l2)  # all items present in l2 will be added to l1
```

Ex:

```
x=["Chiru","Nag","Balaiah"]
y=["ramcharan","chaitu",,"ntr","allu arjun"]
x.extend(y)
print(order1)
```

Deleting elements from a list:

10.remove() : This function is used to remove specified item from the list.

If the item present multiple times then only first occurrence will be removed.

Ex**:**

```
n=[10,20,10,30]
n.remove(10)
print(n)
```

If the specified item not present in list then we will get ValueError.

Ex:

```
n=[10,20,10,30]
n.remove(40) # ValueErrir 40 not in list
print(n)
```

Note: Hence before using remove () method first we have to check specified element present in the list or not by using in operator.

11. pop( ) : It removes and returns the last element of the list. This is only function which manipulates list and returns some element.

Ex:

```
n[10,20,30,40] print(n.pop()) print(n.pop())
print(n)
```

If the list is empty then pop() function raises IndexError

Ex:

```
n=[ ]
print(n.pop()) ==> IndexError: pop from
```

empty list

Note:

1. pop () is the only function which manipulates the list and returns some value 2. In general we can use append () and pop () functions to implement stack data structure by using list, which follows LIFO(Last In First Out) order.

In general we can use pop () function to remove last element of the list. But we can use to remove elements based on index.

n.pop(index):To remove and return element present at specified index.

n.pop() :To remove and return last element of the list :

```
n=[10,20,30,40,50,60] print(n.pop()) #60
print(n.pop(1)) #20


print(n.pop(10)) ==>IndexError: pop index out of range
```

Differences between remove() and pop()

Note: List objects are dynamic. i.e. based on our requirement we can increase and decrease the size.

append (), insert() ,extend() ===>for increasing the size/growable nature
remove (), pop() =======>for decreasing the size /shrinking nature
remove() pop( )

1) The remove ( ) is used to remove special element from the List.
   But the pop () is used to remove last element from the List.

2) the remove function can't return any value. but the pop () returned removed element.

3) In remove ( ) , If special element not available then we get VALUE ERROR.
but in pop () , If List is empty then we get Error.

12.<u>clear( )</u> : This function is used to remove all elements of List.

Ex:

```
n=[10,20,30,40] print(n)
n.clear()
print(n)
```

13.reverse(): This function is used to arrange the elements in reverse order in a given list.

Ex:

```
n=[10,20,30,40]
n.reverse()
print(n)
```

14. sort( ) : In list by default insertion order is preserved. If want to sort the elements of list according. to default natural sorting order then we should go for sort() method.

For numbers ==>default natural sorting order is Ascending Order

For Strings ==> default natural sorting order is Alphabetical Order.

Ex-1:

```
n=[20,5,15,10,0]
n.sort()
```

print(n) #[0,5,10,15,20]

Ex-2:

```
s=["Grapes","Banana","Orange","Apple"]
s.sort()
print(s)
```

Note: To use sort() function, compulsory list should contain only homogeneous elements. otherwise we will get TypeError.

Ex-3:

```
n=[20,10,"A","B"]
n.sort()
print(n)#TypeError: '<' not supported between
                #instances of 'str' and 'int'
```

To sort in reverse of default natural sorting order:

We can sort according to reverse of default natural sorting order by using reverse=True argument.

Ex-4:

```
n=[40,10,30,20]
n.sort()
print(n) ==>[10,20,30,40]
n.sort(reverse=True)
print(n) ===>[40,30,20,10]
n.sort(reverse=False)
```

print(n) ==>[10,20,30,40]

Aliasing and Cloning of List objects:

The process of giving another reference variable to the existing list is called aliasing.

Ex:

```
x=[10,20,30,40] y=x
print(id(x)) print(id(y))
```

The problem in this approach is by using one reference variable if we are changing content, then those changes will be reflected to the other reference variable.

```
x=[10,20,30,40]
y=x
y[1]=777
print(x) =>[10,777,30,40]
```

To overcome this problem we should go for cloning.

The process of creating exactly duplicate independent object is called cloning.
We can implement cloning by using slice operator or by using copy() function

- **By using slice operator:**

```
x=[10,20,30,40]  y=x[:]
y[1]=777
print(x) ==>[10,20,30,40]
print(y) ==>[10,777,30,40]
```

- **By using copy() function:**

```
 x=[10,20,30,40]      y=x.copy()
y[1]=777
print(x) ==>[10,20,30,40]
print(y) ==>[10,777,30,40]
```

Difference between = operator and copy function is
'=' operator meant for aliasing .
copy() function meant for cloning.

Mathematical operators for List Objects:

1. Concatenation operator(+):

The + operator is used to concatenate two or more lists into a single list.

Ex:

```
a=[10,20,30] b=[40,50,60]
c=a+b
print(c) ==>[10,20,30,40,50,60]
```

Note: To use + operator compulsory both arguments should be list objects,otherwise we will get TypeError.

Ex:

```
c=a+40 ==>TypeError: can only concatenate list
```

(not "int") to list

```
c=a+[40] ==>valid
```

2. Repetition Operator (*):

The repetition operator * used to repeat elements of list specified number of times.

Ex:

```
x=[10,20,30]

y=x*3
print(y)==>[10,20,30,10,20,30,10,20,30]
```

3.Comparing List objects :

We can use comparison operators for List objects.

Ex:

```
x=[11,22,33,44] y=[55,66,77,88]
z=[11,22,33,44] print(x==y) ➔False
print(x==z) ➔True
print(x != z)➔ False
print(y != z)➔ True
```

Note: Whenever we are using comparison operators(==,!=) for List objects then the following should be considered

1. The number of elements

2. The order of elements
3. The content of elements (case sensitive)

Note: When ever we are using relatational operators(<,<=,>,>=) between List objects,only first element comparison will be performed.

Ex:

```
x=[50,20,30] y=[40,50,60,100,200]
print(x>y)➔ True
print(x>=y)➔ True
print(x<y) ➔False print(x<=y) ➔False
```

Ex:

```
x=["Dog","Cat","Rat"] y=["Rat","Cat","Dog"]
print(x>y)➔ False
print(x>=y) False
 print(x<y)➔ True
 print(x<=y)➔ True
```

Membership operators in and not in operators in lists:

We can check whether element is a member of the list or not by using memebership operators.

・in operator ・not in operator

Ex:

```
n=[10,20,30,40]
print (10 in n)
print (10 not in n) print (50 in n)
print (50 not in n)
```

Nested Lists:

Sometimes we can take one list inside another list.
Such type of lists are called nested lists.

Ex:

```
n=[10,20,[30,40]] print(n)
print(n[0]) print(n[2])
print(n[2][0]) print(n[2][1])
```

Note: We can access nested list elements by using index just like accessing multi-dimensional array elements.

Nested List as Matrix: In Python we can represent matrix by using nested lists.

Ex:

```
n= [[10,20,30], [40,50,60], [70,80,90]]  print(n)
print ("Elements by Row wise:")
for r in n:
        print(r)
print ("Elements by Matrix style:")  for i in range(len(n)):
        for j in range(len(n[i])):
print(n[i][j], end=' ')                    print ()
```

List Comprehensions:

It is very easy and compact way of creating list objects from any iterable objects(like list,tuple,dictionary,range etc) based on some condition.

Syntax:

list=[expression for item in list if condition] Examples:

Ex-1 :

```
s=[ x*x for x in range(1,11)]
print(s)
```

Ex-2**:**

```
v=[2**x for x in range(1,6)]
print(v)
```

Ex- 3:

```
m=[x for x in s if x%2==0]
print(m)
```

Ex- 4 :

```
words=["Balaiah","Nag","Venkatesh","Chiran”]
l=[w[0] for w in words]          print(l)
```

Ex-5:

```
num1=[10,20,30,40]
```

num2=[30,40,50,60]
num3=[i for i in num1 if i not in num2] print(num3) # [10,20]

Ex- 6: common elements present in num1 and num2

num4=[i for i in num1 if i in num2] print(num) # [30, 40]

del command:

A list item can be deleted with the del command:

Syntax:

Del list object(index)

```
>>> a=[11,22,33,44]
>>> del a[3]  # index number 3 will be deleted
>>> del a    #entire list a will be deleted
```

TUPLE Data Structure

Properties:

1) Tuple objects can be created by using Parenthesis or by using tuple( ) function or by assigning multiple values to a single variable. Tuple elements are separated by comma separator.
2) In python tuple is a collection, that cannot be modified. Tuple is exactly same as List except that it is immutable. i.e. once we create Tuple object, we cannot perform any changes in that object.
3) Hence Tuple is Read Only version of List.
4) If our data is fixed and never changes then we should go for Tuple.
5) Insertion Order is preserved
6) Duplicates are allowed
7) Heterogeneous objects are allowed.
8) We can preserve insertion order and we can differentiate duplicate objects by using index. Hence index will play very important role in Tuple also.
9) Tuple support both +ve and -ve index. +ve index means forward direction(from left to right) and -ve index means backward direction(from right to left) 10) Parenthesis are optional but recommended to use.
11) Insertions are not allowed.
12) Deletions are not allowed.

Ex:

```
t=10,20,30,40
print(t)        print(type(t))
```

Tuple creation:

```
t=( )
```

creation of empty tuple t=(10,)

```
t=10,
```

Note: creation of single valued tuple, parenthesis is optional, should ends with comma

creating a tuple with multi values

```
t=10,20,30
```

```
t=(10,20,30)
```

Note: creation of multi values tuples & parenthesis are optional

creating a tuple by using tuple( ) function: list=[10,20,30]

```
t=tuple(list)
print(t)
```

Accessing elements of Tuple:

We can access either by index or by slice operator

1. By using index:

```
t=(10,20,30,40,50,60)
print(t[0]) #10
print(t[-1]) #60
print(t[100]) IndexError: tuple index out of range
```

2. By using slice operator:

```
t=(10,20,30,40,50,60) print(t[2:5])
print(t[2:100])
print(t[::2])
```

Tuple vs immutability:

Once we create tuple, we cannot change its content. Hence tuple objects are immutable.

Ex:

```
t=(10,20,30,40)
t[1]=70 TypeError: 'tuple' object does not support item assignment
```

Note: Mathematical operators for tuple: We can apply + and * operators for tuple

Concatenation Operator(+):

```
t1=(10,20,30)
t2=(40,50,60)
t3=t1+t2
print(t3) # 10,20,30,40,50,60)
```

Multiplication operator or repetition operator(*)

```
t1=(10,20,30)
t2=t1*3
print(t2) #(10,20,30,10,20,30,10,20,30)
```

Important functions/Built=in Methods of Tuple:

1.len(): To return number of elements present in the tuple

Ex:

```
t=(10,20,30,40)
print(len(t)) #4
```

2.max(): It return highest value present in the tuple

Ex:

```
t=(10,20,30,40)
print(max(t)) #40
```

3.min(): It return lowest value present in the tuple

Ex:

```
t=(10,20,30,40)
```

```
print(min(t)) #10
```

4.sum(): It returns sum of all elements present in the tuple. Ex:

```
t=(10,20,30,40)
print(sum(t)) #100
```

5.count( ):

This function returns number of times present a given element in a tuple.
If the element not present in a tuple it returns zero.

Ex:

```
t=(10,20,10,10,20)
print(t.count(10)) #3
```

6.index(): It returns index of first occurrence of the given element.

If the specified element is not available then we will get ValueError.

Ex:

```
t=(10,20,10,10,20)
print(t.index(10)) #0
print(t.index(30))
# ValueError: tuple.index(x): x not in tuple
```

7.sorted(): To sort elements based on default natural sorting order.

Ex:

```
t=(40,10,30,20) t1=sorted(t)
print(t1)
print(t)
```

We can sort according to reverse of default natural sorting order as follows

```
t1=sorted(t,reverse=True)
print(t1) [40, 30, 20, 10]
```

Tuple Packing and Unpacking: We can create a tuple by packing a group of variables. Ex:

```
a=10
b=20
c=30
```

```
d=40 t=a,b,c,d
print(t) #(10, 20, 30, 40)
```

Here a,b,c,d are packed into a tuple t. This is nothing but tuple packing.

Tuple unpacking is the reverse process of tuple packing.We can unpack a tuple and assign its values to different variables

Eg:

```
t=(10,20,30,40)
a,b,c,d=t
print("a=",a,"b=",b,"c=",c,"d=",d)
```

Note: At the time of tuple unpacking the number of variables and number of values should be same., otherwise we will get ValueError.

Ex:

```
t=(10,20,30,40)
a,b,c=t
#ValueError: too many values to unpack  (expected 3)
```

Tuple Comprehension:

Tuple Comprehension is not supported by Python.

```
t= ( x**2 for x in range(1,6))
```

Here we are not getting tuple object and we are getting generator object.

```
t= ( x**2 for x in range(1,6))
print(type(t))
for x in t:
        print(x)
```

Write a program to take a tuple of numbers from the keyboard and print its sum and average?

```
t=eval(input("Enter Tuple of Numbers:"))
l=len(t)
sum=0
for x in t:
   sum=sum+x
```

```
print("The Sum=",sum)
print("The Average=",sum/l)
```

Differences between List and Tuple:

1. List and Tuple are exactly same except small difference: List objects are mutable where as Tuple objects are immutable.
2. In both cases insertion order is preserved, duplicate objects are allowed, heterogenous objects are allowed, index and slicing are supported.
3. List is a Group of Comma separated Values within Square Brackets and Square Brackets are mandatory.

   Ex:

   i = [10, 20, 30, 40]

   Tuple is a Group of Comma separated Values within Parenthesis and Parenthesis are optional.

   Ex:

   t = (10, 20, 30, 40)

   t = 10, 20, 30, 40
4. List Objects are Mutable i.e. once we create List Object, we can perform any changes in that Object. eg: l [3] = 888

   whereas Tuple Objects are Immutable i.e. once we create Tuple Object, we cannot change its content.

   t [3] = 8888..... ValueError: tuple object does not support item assignment.
5. If the Content is not fixed and keep on changing then we should go for List. If the content is fixed and never changes then we should go for Tuple.

SET Data Structure

Properties:

Python's built-in set type has the following characteristics:

1. Set objects can be created by using curly brackets { } or by using set() function.
2. Set elements are separated by comma. we want to represent a group of unique values as a single entity then we should go for set.
3. Duplicates are not allowed.
4. The set doesn't maintain the elements in any particular oder.
   i.e. set is a unordered collection of elements.
5. Indexing and slicing not allowed for the set.
6. Heterogeneous elements are allowed in sets.
7. Set objects are mutable i.e once we create set object, we can perform any changes in that object based on our requirement.
8. We can represent set elements within curly braces and with comma separation
9. We can apply mathematical operations like union, intersection, difference etc on set objects.
10. A set itself may be modified, but the elements contained in the set Must be of an immutable type.
11. Insertions are allowed.
11. Deletions are allowed.

Creation of Set objects:

To create set objects by using curly braces

Ex:

```
s={10,20,30,40}
print(s)
print(type(s))
```

We can create set objects by using set () function.

Syntax:

s=set (any sequence)

Ex-1:

```
l = [10,20,30,40,10,20,10]
```

```
s=set(l)
print(s) # {40, 10, 20, 30}
```

Ex-2:

```
s=set(range(5))
```

print(s) #{0, 1, 2, 3, 4}

Note: While creating empty set we have to take special care.

Compulsory we should use set() function.

s={} #It is treated as dictionary but not empty set.

Ex:

```
s={} print(s)
print(type(s))
```

Ex:

```
s=set () print(s)
print(type(s))
```

Built-in methods in sets:

1. add(x): Adds item x to the set.

Ex:

```
s={10,20,30}
s.add(40)
print(s) #{40, 10, 20, 30}
```

2. update(x,y,z): To add multiple items to the set.

Arguments are not individual elements and these are Iterable objects like List, range etc. All elements present in the given Iterable objects will be added to the set.

Ex:

```
s= {10,20,30}
l= [40,50,60,10]
update(l,range(5))
print(s)
```

Q. What is the difference between add () and update () functions in set?

We can use add() to add individual item to the Set, where as we can use update() function to add multiple items to Set. add() function can take only one argument where as update() function can take any number of arguments but all arguments should be iterable objects.

3. copy(): Returns copy of the set. It is cloned object.

Ex:

```
s={10,20,30} s1=s.copy()
print(s1)
```

4. pop(): It removes and returns some random element from the set.

Ex:

```
s={40,10,30,20} print(s) print(s.pop())
print(s)
```

5. remove(x): It removes specified element from the set. If the specified element not present in the Set then we will get KeyError

Ex:

```
s={40,10,30,20}
s.remove(30)
print(s)
```

s.remove(50) #KeyError: 50

6. discard(x): It removes the specified element from the set. If the specified element not present in the set then we won't get any error.

Ex:

```
s={10,20,30}
s.discard(10) print(s) ===>{20, 30}
s.discard(50)
print(s) ==>{20, 30}
```

7. clear (): To remove all elements from the Set.

Ex:

```
s={10,20,30} print(s)
s.clear()
print(s)
```

8. len( ): To return number of elements present in the set

Ex:

```
s={10,20,30,40}
print(len(s)) #4
```

9. max( ): It return highest value present in the tuple

Ex:

```
s=[10,20,30,40}
print(max(s)) #40
```

10.min( ) : It return lowest value present in the tuple

Ex:

```
s=(10,20,30,40)
print(min(s)) #10
```

11.Sum( ) : It returns sum of all elements present in the tuple

Ex:

```
s=(10,20,30,40)
print(sum(s)) #100
```

Mathematical operations on the Set:

1.union():

We can use this function to return all elements present in both sets.

x.union(y) or x|y

Ex:

```
x={10,20,30,40}
y={30,40,50,60}
print(x.union(y))#{10, 20, 30, 40, 50, 60}
print(x|y) #{10, 20, 30, 40, 50, 60}
```

2. intersection():

This function _Returns common elements present in both x and y.

x.intersection(y) or x&y

Ex:

```
x={10,20,30,40} y={30,40,50,60}
print(x.intersection(y)) #{40, 30} print(x&y) #{40, 30}
```

3. difference():

This function returns the elements present in x but not in y.

x.difference(y) or x-y**.**

Ex:

```
x={10,20,30,40} y={30,40,50,60}
print(x.difference(y)) #{10, 20}
print(x-y) #{10, 20}
print(y-x) #{50, 60}
```

4.symmetric_difference():

Returns elements present in either x or y but not in both.

x.symmetric_difference(y) or x^y

Ex:

```
x={10,20,30,40}
y={30,40,50,60}
        print(x.symmetric_difference(y))
                #{10,50,20, 60}
print(x^y) #{10, 50, 20, 60}
```

5. isdisjoint( ) :

This method returns True if two sets have a null intersection.

Ex:

```
x={1,2,3,4}
y={11,22,334,74}
print(x.isdisjoint(y))        #True
```

6.issubset( ):

x.issubset(y)

Returns True, if x is a subset of y. "<=" is an abbreviation for

"Subset of" and ">=" for "superset of"

"<" is used to check if a set is a proper subset of a set.

```
x = {"a","b","c","d","e"}
y = {"c","d"}
x.issubset(y)
y.issubset(x)
```

Membership operators: (in , not in):

If the given element present in a set it returns True , otherwise it returns False.

Ex-1:

```
s=set("kusu") print(s)
print('u' in s) print('z' in s)
```

Ex-2:

```
s={11,22,33,44} print(10 in s) print(22
in s) print(44 not in s)
print(99 in s)
```

for loop in sets:

Using for loop to access set elements one by one.

Ex:

```
s={11,22,33,44}
for i in s:
    print(i)
```

Set Comprehension:

Set comprehension is possible.

Ex-1:

```
s={x*x for x in range(5)}
print(s) #{0, 1, 4, 9, 16}
```

Ex-2:

```
s={2**x for x in range(2,10,2)} print(s) #{16, 256, 64, 4}
```

set objects won't support indexing and slicing:

Ex-3:

```
s={10,20,30,40}
```

```
print(s[0])
#TypeError:'set' object does not support indexing
print(s[1:3])
#TypeError: 'set' object is not sub scriptable
```

Write a program to eliminate duplicates present in the list?

Approach-1:

```
l=eval(input("Enter List of values: "))
s=set(l)
print(s)
```

Approach-2:

```
l=eval(input("Enter List of values: "))  l1=[]
for x in l:
    if x not in l1:
        l1.append(x)
print(l1)
```

Write a program to print different vowels present in the given word?

```
w=input("Enter word to search for vowels: ") s=set(w)
v={'a','e','i','o','u'}
d=s.intersection(v)
print("The  different  vowel  present in",w,"are",d)
```

DICTIONARIES

Properties:

1. A collection of key-value pairs, each pair separated by commas, and enclosed by curly braces. It is called a dictionary.
2. Dictionaries are created by using the dict( )
3. Insertion order is preserved.
4. Insertions are allowed.
5. Deletions are allowed.
6. Heterogeneous elements are allowed.
   i.e. different data types
7. Duplicate keys are not allowed but, duplicate values are allowed.
8. Dictionaries are mutable. i.e to change its content.
9. Dictionary is dynamic i.e. based on our requirement the size can be increased or decreased.
10. Dictionaries are not supporting indexing and slicing in place of indexing dictionaries support keys.
11. Dictionary declaration syntax is:

    d = { key-1: value-1, key-2: value-2 .....key-n: value-n }

12. A colon separates each key from its associated value.

ex-1: To create a empty dictionary

```
d={ }
print(d)
print(type(d))
```

ex-2: To create a empty dictionary using dict()

```
d=dict() print(d)
print(type(d))
```

ex-3: To create a dictionary with elements

```
d={0:11,1:22,2:33,3:44 }
print(d)
print(type(d))
```

ex-4: To create a dictionary with elements

```
d={1:'python',2:'cpp',3:'java',4:'data science'} print(d)
print(type(d))
```

ex-5: To create a dictionary with values at runtime d={ }

```
k=int(input('enter key:')) v=input('enter
value:') d[k]=v print(d)
```

ex-6: To create a dictionary with more than one #values at runtime

```
d={ }
n=int(input('enter how many elements in a dictionary'))
for i in range(1,n+1):
    k=int(input('enter key:'))    v=input('enter value:')
    d[k]=v
print(d)
```

ex-7: To create a dictionary with duplicate keys

```
    d={1:'python',2:'cpp',3:'java',4:'datascience',2:'django'}
print(d)
print(type(d))
```

ex-8: To create a dictionary with duplicate values

```
    d={1:'python',2:'cpp',3:'java',4:'data \
science',5:'cpp'}
    print(d)
    print(type(d))
```

A dictionary is a data type similar to arrays, but works with keys and values instead of indexes. Each value stored in a dictionary can be accessed using a key, which is any type of object (a string , a number, a list etc.) instread of using its index to address it.

Ex: A database of phone number could be stored using a dictionary like this:

```
phonebook = { }
phonebook['Jhon']=9948123456
phonebook['Laden']=9956781234
phonebook['Modi']=9912345678
print(phonebook)
```

Built-in methods in dictionaries:

1. **len( )**: It returns no. of elements present in a dictionary.
2. **max( )**: It returns highest key in a dictionary
3. **miin( )**: It returns lowest key in a dictionary
4. **sum( )**: It returns sum of keys in a dictionary.

Ex:

```
d={101:'python',102:'ML',103:'DL',104:'AI'}
print('no. of elements in a dictionary = ',len(d)) print('highest key in a
dictionary = ',max(d)) print('lowest key in a dictionary = ',min(d)) print('sum of
elements in a dictionary = ',sum(d))
```

5. **keys( )**: This function returns only keys.

Ex:

```
d={101:'python',102:'ML',103:'DL',104:'AI'}
print(d)
print(d.keys())
for i in d.keys():
        print(i)
```

6. **values()**: This function returns only values.

Ex:

```
d={101:'python',102:'ML',103:'DL',104:'AI'}
print(d)
print(d.values())
for i in d.values():
    print(i)
```

7. **items( )**: This function returns both keys and values.

Ex:

```
d={101:'python',102:'ML',103:'DL',104:'AI'}
print(d)
  print(d.items())
for i in d.items():
        print(i)
for i,j in d.items():
 print(i,'--->',j)
```

8. **setdefault( )**: This function is used to insert new elements in a dictionary.

Ex:

```
d={101:'python',102:'ML',103:'DL',104:'AI'}
print(d)
d.setdefault(108,'Django')
d.setdefault(109,'Flask')
print(d)
```

9. **popitem():** This function removes last element from a dictionary

Ex:

```
d={101:'python',102:'ML',103:'DL',104:'AI'}
print(d)
d.popitem()
print(d)
```

10. **pop()**: This function removes specified key from a dictionary.

Ex:

```
d={101:'python',102:'cpp',103:'java',104: \
        'data science'}
print(d)
d.pop(102)
d.pop(103)
print(d)
```

11. **clear()**: This function removes all elements from a dictionary.

Ex:

```
d={101:'python',102:'cpp',103:'java',104: \
'data science'}
print(d)
d.clear()
print(d)
```

12. **get( ): T**his function returns the value of the given key.

Ex:

```
d={101:'python',102:'cpp',103:'java',104: \
        'data science'}
print(d.get(102))
print(d.get(104))
print(d.get(101))
print(d[101])
```

13. copy( ): This funciton copy the elements from one dictionary to another dictionary. i.e cloned copy.

Ex:

```
d={101:'python',102:'cpp',103:'java',104:\
        'data science'}
d1=d.copy()
print(d)
print(d1)
```

14. update( ): This funciton is used to add multiple values in a dictionary.

Ex:

```
x={1:'chiru',2:'venky',3:'nag'}      d={101:'python',102:'cpp',103:'java',104: \
        'data science'}
d.update(x)
print(d)
```

15. fromkeys( ):

The fromkeys( ) method creates a new dictionary from the given sequence of elements with a value provided by the user.

The syntax of fromkeys() method is:

```
dictionary.fromkeys(sequence[, value])
```

sequence - sequence of elements which is to be used as keys for the new dictionary

value (Optional) - value which is set to each each element of the dictionary

Ex-1:

```
seq = { 'a', 'b', 'c', 'd', 'e' }
res_dict = dict.fromkeys(seq)
print ("The newly created dict with None values : " + str(res_dict))
```

Ex-2:

```
seq = { 'a', 'b', 'c', 'd', 'e' } res_dict2 = dict.fromkeys(seq, 1)
```

```
print ("The newly created dict with 1 as value : " + str(res_dict2))
```

16. sorted( ) :

This method sorts the elements of a given iterable in a specific order either

Ascending order or Descending order.

Syntax :

```
sorted(iterable[, reverse])
```

Ex:

```
pyDict = {'e': 1, 'a': 2, 'u': 3, 'o': 4, 'i': 5}     print(sorted(pyDict))
pyFSet = frozenset(('e', 'a', 'u', 'o', 'i'))
print (sorted(pyFSet, reverse=True))
```

Dictionary comprehension:

To create dictionary with key-value pairs using comprehension:

Syntax:

```
dictionaryname= {exp-1: exp-2 for loop if condition}
```

Here, exp-1 represents key

exp-2 represents value

Ex-1 : To create a dictionary 1 to 5 numbers:

```
d={i:i for i in range(1,6)}
print(d)
```

Ex-2: To create a dictionary even numbers between 1 to 10 numbers:

```
d={i:i for i in range(1,11) if i%2==0}
print(d)
```

Ex-3: To create a dictionary cube of each value

```
d={i:i*i*i for i in range(1,6)}
print(d)
```

Ex-4:

```
Fruits=  ['apple', 'mango','banana','cherry']
# dict comprehension to create dict with fruit name as keys
d={f:len(f) for f in fruits}
print(d)
```

Ex-5:

```
sdict = {x.upper(): x*3 for x in 'python'}  print (sdict)
```

EX-5: using comprehension

```
keys = ['a','b','c','d','e']
values = [1,2,3,4,5]
d = { k:v for (k,v) in zip(keys, values)}    print(d)
```

without comrehension:

```
keys = 'a','b','c','d','e']  values = [1,2,3,4,5]
d = dict(zip(keys, values))    print (d)
```

Dictionaries and lists share the following characteristics:

- Both are mutable.
- Both are dynamic. They can grow and shrink as needed.
- Both can be nested. A list can contain another list.
  A dictionary can contain another dictionary.
  A dictionary can also contain a list, and vice versa.

Dictionaries differ from lists primarily in how elements are accessed:

- List elements are accessed by their position in the list, via indexing.
- Dictionary elements are accessed via keys.

STRINGS

1. In python, A single character or group of characters is called a string.
2. Strings are sequences of character data. The string type in python is called **str**.
3. A character is simply a symbol. For example, the English language has 26 characters.
4. Strings are enclosed by single quotes or double quotes or triple single quotes or triple double quotes.
5. Strings are two types:

    a) Single line strings: A string written in only one line it is called single line strings. These strings are enclosed by single quotes or double quotes or triple single quotes or triple double quotes.

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

    b). Multi-line strings: A string written in more than one line it is called multi-line strings. These strings are enclosed by triple single quotes or triple double quotes.

```
var3 = ' ' ' python is  a high level programming language
  python is  a interpreter programming language
   it is scripting programming language ' ' '
```

Accessing Values in Strings:

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring. For example :

```
var1 = 'Hello World!'
var2 = "Pythonprogramming"
print ("var1[0]: ", var1[0])
print ("var2[1:5]: ", var2[1:5])
output:
var1[0]:  H
var2[1:5]: ytho
```

Indexing:

Python supports both positive and negative index. Here two indices are used separated by a colon (:). A slice 3:7 means indices characters of 3rd, 4th, 5th and 6th positions. The second integer index i.e. 7 is not included. You can use negative indices for slicing.

string1 ="PYTHON"

Character	P	Y	T	H	O	N	
Index (from left)	0		1	2	3	4	5
Index (from right)	-6		-5	-4	-3	-2	-1

eg:

```
str1="python is a high level programming language" print(str1[0])
print(str1[-1]) print(str1[1:4])
print(str1[-4])
```

Updating Strings:

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether. For example −

```
var1 = 'Hello World!'
print ("Updated String: - ", var1[:6] + 'Python') When the above code is
executed, it produces the following result −

Updated String: - Hello Python
```

For loop in strings :

Using for loop we can iterate through a string. Here is an example to count the number of 'l' in a string.

```
count = 0
for letter in 'Hello World':
    if letter =='l':
       count = count + 1

print(count,'letters found')
```

special characters in strings

In Python The backslash (\) character is used to introduce a special character.

Escape Sequence	Meaning
\n	Newline
\t	Horizontal Tab
\\	Backslash
\'	Single Quote
\"	Double Quote

Ex**:**

```
print("This is Back Slash (\\)Mark.")
print("This is Back Slash \t Mark.")
print("This is Back Slash \'Single Quotrs\' Mark.") print("This is Back Slash \n
Mark.")
```

String Formatting Operator:

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf() family.

Format Symbol	**Conversion**
%c	character

%s	string conversion via str() prior to formatting
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)

Mathematical operators in strings:

1. concatenate operator(+) :

This operator is used to Concatenates (joins) two or more strings.

syntax: string1 + string2

```
Ex:  a = "kosmik  " + "Technologies"
  print(a)
```

2. repetition operator(*) :

This operator is used Repeats the string for number of 'n' times specified

```
  Ex:    a = "Python " * 3
      print(a)
output: python python python
```

Slice operator [ ] : Returns the character from the index provided at x.

```
string[x]  a = "Sea"
print(a[1])
output: e
```

Range Slice[:] : Returns the characters from the range provided at x:y.

```
Syntax:        string[x:y]
Ex:
               a = "Mushroom" print(a[4:8])
               output:   room
```

in Membership: Returns True if x exists in the string. Can be multiple characters.

Syntax: x in string

Ex:

```
a = "hyderabad" print ("m" in
a) print ("b" in a) print ("bad"
in a)
output: False
        True
        True
```

not in Membership: Returns True if x does not exist in the string. Can be multiple characters.

Syntax: x not in string

Ex:

```
a = "Mushroom" print("m" not in a)

print("b" not in a)

print("shroo" not in a)

output: False
                True
                False
```

Suppresses an escape sequence (\x) so that it is actually rendered. In other words, it prevents the escape character from being an escape character. r"\x"

```
a = "1" + "\t" + "Bee" b = "2" + r"\t"
+ "Tea" print(a) print(b) Output:
        1               Bee
        2\tTea
```

% operator: Performs string formatting. It can be used as a placeholder for another value to be inserted into the string. The % symbol is a prefix to another character (x) which defines the type of value to be inserted. The value to be inserted is listed at the end of the string after another % character.

Format symbol(character)	Description
%c	Character.
%s	String conversion via str() prior to formatting.

%i	Signed decimal integer.
%d	Signed decimal integer.
%u	Unsigned decimal integer.
%o	Octal integer.
%x	Hexadecimal integer using lowercase letters.
%X	Hexadecimal integer using uppercase letters.
%e	Exponential notation with lowercase e.
%E	Exponential notation with uppercase e.
%f	Floating point real number.
%g	The shorter of %f and %e. %G The shorter of %f and %E.

Ex1:

```
a = "Hi %s" % ("Homer")
print(a)
```

Ex2:

```
print("%s having %s Years Experience in IT." %("Kusu srinivas","20+"))
print("%s is one of the best IT training center %d Years Experience in IT."
%("KOSMIK",10))
```

Ex3:

```
name = "Guido van rossum"
print("Hello, %s!" % name)
```

Ex4:

```
name = "python " age = 30
print("%s is %d years old programming language." % (name, age))
```

String methods:

1. len(): This function returns no. of characters present in a string.

Ex-1:

```
x='python is a high level language'
print(len(x))
```

2. capitalize (): This function is used to convert the first charcter to uppercase

Ex-1:

```
x='python is a high level language'
print(x.capitalize())
```

Ex-2:

```
s='level'
print(s.capitalize())
```

3. upper (): This function is used to convert the string into uppercase.

Ex-1:

```
x='python is a high-level language'
print(x.upper())
```

Ex-2:

```
y='level'
print(y.upper())
```

4. lower(): This function is used to convert the string into lowercase

Ex:

```
x='KKRCODERS'
print(x.lower())
```

5. swapcase(): This function swap cases, lower case becomes upper case and vice versa.

Ex:

```
x='KOSMik'
print(x.swapcase())
```

6 . split(): This function splits a string into a list you can specify the separator, default separator is any whitespace.

syntax:

```
string.split(separator)
```

Ex-1:

```
x= 'hello friends good morning'
print(x.split())
```

Ex-2:

```
x= ' hello friends good morning'
print(x.split('e'))
```

Ex-3:

```
s='python is a high level language'          print(s.split('h'))
```

7. replace(): This function replaces a new string with old you can specify the separator, default separator is any whitespace.

syntax:

```
string.split(oldvalue,new value)
```

Ex-1:

```
x='hello friends good morning'
print(x.replace('hello','good'))
```

Ex-2:

```
x='hello friends good morning'
print(x.replace('morning','afternoon'))
```

Ex-3: replace the first two occurence of the word 'python'

```
x='python is a high level language python is a scripting language,
python is' print(x.replace('python','kosmik',2) )
```

8. **count():** This function returns the number of times a specified value appears in the string.

Ex:

x='python is a high level language python is a scripting language, python is' print(x.count('python')) print(x.count('h'))

9. **index():** This function finds the index number of the specified value.

Ex:

```
x='hello, welcome to hyderabad'
print(x.index('welcome'))
print(x.index('h'))
```

10. **islower():** If all the characters in the string are lower case, it returns True. otherwise False.

Ex-1:

```
x='pyThon'
print(x.islower())
```

Ex-2:

```
y='kosmik'
print(y.islower())
```

Ex-3:

```
z='HELLO'
print(z.islower())
```

11. **isupper():** If all the characters in the string are upper case, it returns True. otherwise False.

Ex-1:

```
x='PYTHON'
print(x.isupper())
```

Ex-2:

```
y='kosmik'
print(y.isupper())
```

Ex-3:

```
z='HELLO'
```

```
print(z.isupper())
```

12. **isdigit():** If all the characters in the string are digits, then it returns True. otherwise returns False

Ex-1:

```
x='python'
print(x.isdigit())
```

Ex-2:

```
y='12345'
print(y.isdigit())
```

Ex-3:

```
z='python 3.8.0'  print(z.isdigit())
```

13. **isalnum ( ) :** It checks whether the string consists of alphanumeric characters.

Ex:

```
str = "this2009";
print(str.isalnum( ))
str = "this is string example....wow!!!"; print(str.isalnum ())
```

14. **max( ):** This method returns the max alphabetical character from the string str.

Ex**:**

```
str1="abcd" print(max(str1))
str2="abcddcba" print(max(str2))
str3="Maximum" print(max(str3))
str4="12344321" print(max(str4))
```

15. **title( ):** It returns a copy of the string in which first characters of all the words are capitalized.

Ex:

```
str = "this is powerful python scripting !"; print(str.title())
```

16. **join( ):** This methods is used to joining list of strings into one string.

Ex-1:

```
items = ["dell ", "hp ", "acer"]        message = ""
print(message.join(items))
```

Ex-2:

```
s = "****";
seq = ("98", "66", "11")
print(s.join( seq ))
```

17. **isdecimal( ):** It checks whether the string consists of only decimal characters. This method are present only on Unicode objects.

Ex-1:

```
str = "this2020"
print(str.isdecimal( ))
```

Ex-2:

```
str = "23443434";
print(str.isdecimal( ))
```

18. **zfill( ):** The method zfill() pads string on the left with zeros to fill width.

Ex:

```
str="Python Rocks"
print(str.zfill(20))
```

19. **format():**This is one of the string formatting methods in Python3.0 above versions. which allows multiple substitutions and value formatting.

Ex:

```
a=10
b=20
print('the value of a  is {} and the value of b is {}\ '.format(a, b))
```

FUNCTIONS

A collection of statements together into a single unit. This unit is called a function. In Python, function is a group of related statements that perform a specific task.

If a group of statements is repeatedly required then it is not recommended to write these statements every time separately. We have to define these statements as a single unit and we can call that unit any number of times based on our requirement without rewriting. This unit is nothing but function.

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Advantages of Python Functions:

---

1. Code reusability because we can call the same function multiple times.
2. Modular code since we can define different functions for different tasks.
3. Improves maintainability of the code.
4. Abstraction as the caller doesn't need to know the function implementation.
5. To avoid repetition.

There are three types of functions in python:

1. Built-in functions
2. User-defined functions
3. Anonymous functions

1. Built-in functions:

These functions provided by the Python language. These function are coming from along with python software internally .

Ex:

print(), len( ),str( ),list( ),input( ),type( ),id( ) max(), min( ), int( ),float( ) bool( ),complex( )

append(), remove( ), get( ),clear( ), copy( ),.....etc.

2. user-defined functions:

The functions defined by us in a Python program.

These functions are defined by programmer as per the requirements, are called user defined functions.

In Python a function is defined using the def keyword

syntax:

```
def functionname(argumentslist):
    '''docstring'''
        st-1          st-2
        st-3      ---      -
        --      st-n
    [return expression]
```

Here,

1. def is a keyword. To define the function header.
2. functionname is a name of the function
3. argumentlist are formal arguments . which we pass values to a function. They are optional.
4. A colon (:) to mark end of the function header and is a beginning of the function body.
5. st-1,st-2,st-3.....st-n are called body of the function
6. An optional return statement to return value from function.

A docstring is a special comment at the beginning of a user-defined function that documents the function's behavior. Docstrings are typically specified as triple-quotes stirng comments.

How 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.

Ex- 1: Write a function to find the addition of two numbers

```
def addition(x,y):
    z=x+y
    return z
print(addition(10,20)) print(addition(6,9))
print(addition(4,36))
```

Ex-2: write a function to display a message

```
def wish():
    print('welcome to hyderabad')
wish( )
```

Ex-3: write multiple functions in a single program

```
def mng():
    print('good morning')
def aft():
    print('good afternoon')
def evg():
    print('good evening')
def ngt():
    print('good night')

mng()
aft()
evg()
ngt()
```

Types of user-defined functions:

1. Functions with arguments and with return values
2. Functions with arguments and no return values
3. Functions no arguments and with return values
4. Functions no arguments and no return values

Write a program to find the factorial of a given number:

Functions with arguments and with return value:

```
def factorial(x):
        fact=1
        for i in range(1,x+1):
            fact = fact * i
        return fact
n=int(input('enter any number'))
f=factorial(n)
print('Factorial value=',f)
```

Write a program to find the area of a rectangle.

Functions with arguments and no return values

```
def rectangle(x,y):
        a=x*y
        print('area of a rectngle=',a)
l=int(input('enter length value')) b=int(input('enter breadth
value')) rectangle(l,b)
```

write a function to find the cube value

Functions no arguments and with return values

```
def cube():
        n=int(input('enter any number'))     return n*n*n
c=cube()
print('cube = ',c)
```

write a function no arguments and no return value

```
def myfunction():
        print('hai this is my function')
 myfunction()
```

<u>return statement:</u>

1. return is a keyword.
2. return statement is optional statement.
3. It is used to return one value or more than one values.

4. The return statement is used to exit a function and go back to the place from where it was called.
5. Whenever control reach the return statement of a function then without executing the remaining part of the function control will comes out from function execution by returning some value.
6. The return value of the function can be stored into a variable. and we can use that variable in the remaining part of the function.

7 If we are not storing the return value of a function in any variable then that data will become as a garbage collection.

8. We can define 'n' number of return statements in a function.

Syntax:

return [expression_list]

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.

Ex-1: write a function no return values:

```
def sample():
    print('goodmorning')
    return
sample()
sample()
```

Ex-2: write a function to return only one value

```
def biggest(x,y):
    if x>y:
        return x
    else:
        return y
print('biggest number = ',biggest(10,20)) print('biggest number
= ',biggest(100,20)) print('biggest number = ',biggest(9,45))
```

Ex-3: write a function return more then one value

```
def calculate(x,y):
    return x+y,x-y,x*y,x/y
a,b,c,d=calculate(10,20) print(a)
```

```
        print(b)
        print(c)
        print(d)
```

Ex-4: write a function multiple return values

```
        def func():
            return 1,2,3
        ont,two,three=func()
        print(one,two,three)
```

Nested Functions:

A function defined inside another function is called a nested function.

syntax:

```
def outerfunction(arugmentslist):
        def innerfunction1(arugmentlist):
            body ofthe innerfunction1
        def innerfunction2(arugmentlist):
            body of the innerfunction2
        body of the outerfunctions
```

Ex-1:

```
def f1():
        def f2():
            print('I am f2...')
        def f3():
            print('I am f3...')
        f2()
        f3()
f1()
```

Ex-2:

```
def display():
        def welcome():
            print('welcome to hyderabad')    print('welcome to
            kosmik')        print('hai')
        welcome()
```

```
print('Main program')
    display()
```

nonlocal:

The nonlocal keyword is used to access the variables defined outside the scope of the block . This keyword is always used in the nested functions to access variables defined outside.

Ex:

```
def outer():
        v = 'outer'
        def inner():
            nonlocal v = 'inner'
            inner()
                print(v)
outer()
```

Types of varialbes or scope of the variables:

It is important to understand the concept of local and global variables and how they interact with functions

1. local variables
2. global variables
3. enclosed variables
4. built-in variables

1. Local variables:

a). The variable which are declared inside the current function.
These variables are called local variables.

b). The scope of the variable also only inside of the functions.

c). Local variables are created when the function has started execution and are lost when the function terminates(forever).

Ex-1: To declare local variables inside a functions.

```
def f1():
```

```
        x=10
        print(x)
    def f2():
        y=20
        print(y)
    def f3():
        z=30
        print(z)

    f1()
f2()
f3()
```

Note: x,y,z are called local variables.

Ex-2**:** Write a program to find number of local variables in a function

We can use the co_nlocals() function which returns the number of local variables used by the function to get the desired result.

Implementation of above approach A function containing 3 variables

```
        def fun():
            a = 1
            b=2.1
            str = 'python' x=True
            y=9+3j
    print(fun.__code__.co_nlocals)
```

Ex-3: Write a Python program to find number of local variables in a function.

A function containing no variables

```
        def displa():
            pass
        def fun():
            a, b, c = 1, 2.25, 333
```

```
        str = 'hyderabad'

    print(displa.__code__.co_nlocals)
    print(fun.__code__.co_nlocals)  )
```

2. global variables:

a). The variable which are declared top level of the program or script or module or outside of all functions or outside of main program.

b). These variables are called global variables.

c). The scope of the variables the entire program.

d). Global variables are declared outside the function.

e). Global variables are created as execution starts and are lost when the program ends.

f). They are alos accessible within a function(inside blocks).

Ex:

```
m=55
n=44
k=99
def f1():
    x=10
    print(x)    print(m)    print(n)
    print(k)
def f2():
    y=20
    print(y)    print(m)    print(n)
    print(k)
f1()
f2()
print(m)
print(n)
print(k)
```

Here, m, n ,k are called global variables. x, y are called local variables

3.Enclosed variables or nonlocal variables:

Enclosed variables are declared using the nonlocal keyword. The scope of the nonlocal variables inner function only.

Ex:

```
a=99
def f1():
    b=10
    def innerf1():
        c=5
        nonlocal b
        b=77
        print(a)
        print(b)
        print(c)
    print(b)
    innerf1()
    print(b)
f1()
print(a)
```

4. built-in variables or built-in scope:

The built-in scope has all the names that are loaded into python variable scope when we start the interpreter.

For example, we never need to import any module to access functions like print() and id().

```
def f1():
        def f2():
                print('i am f2')
        f2()
        print('i am f1')
f1()
print(' i am main')
```

<u>global keyword</u> :

The global is a keyword. It is used to declare a global variable in a function. It allows a user to modify a variable outside of the current scope. It is used to create global variables from a non-global scope i.e inside a function. Global keyword is used inside a function only when we want to do assignments or when we want to change a variable. Global is not needed for printing and accessing.

Rules of global keyword:

1. If a variable is assigned a value anywhere within the function's body, it's assumed to be a local unless explicitly declared as global.

2. Variables that are only referenced inside a function are implicitly global.
3. We Use global keyword to use a global variable inside a function.

4. There is no need to use global keyword outside a function.

Ex-1

```
def f1():
        x=10
        global m
        m=99      #....here, f1 fucntion treats as m global     print(x)
        print(m)
def f2():
        y=20
        print(y)
        print(m)
 f1()
 f2()
print(m)
```

Types of Arguments:

Formal Arguments : Which are defined in a function definition.

Actual Arguments : Which are defined in a function calling.

Python supports 5 types of actual arguments.

- Positional arguments
- Keyword arguments
- Default arguments

- Variable length arguments
- Keyword-variable length arguments

1. Positional arguments:

a). The number of actual arguments must be same as number of formal arguments

b). The position is also important

Ex-1: write a function to find addition of two numbers.

```
def addition(x,y):
        z=x+y
        return z

a=int(input('enter first number')) b=int(input('enter second
number')) c=addition(a,b)
print('sum=',c)
```

Ex-2: write a function to display name and age using function

```
def display(y,x):
        print('Name of the person=',x)
        print('Age of the person=',y)

display('ramkumar',40)
```

2. keyword arguments:

a). The number of actual arguments must be same as number of formal arguments

b). The position is not important

c). In function, the values passed through arguments are assigned to parameters, irrespective of its position while calling the funtion to supply the values.

d). We use keywords if we don't know the sequence.

Ex-1:

```
def display(x,y):
        print('Name of the person=',x)
        print('Age of the person=',y)
display(y=40,x='ramkumar')
```

Ex-2:

```
def printnames(m,n):
        print(m + n + 'are best friends')

printnames(n='chandrababu naidu ',m='kcr  ')
```

3. <u>Default arguments:</u>

a). The number of arguments same or not, is not important

b). The position is also not important

c). sometimes we may want to use parameters in a function that takes default values in case the user does not want to provide a value for them.

d). For this, we can use default arguments which assumes a default value if a value is not supplied as an argument while calling the function.

e). An assignment operator '=' is used to give a default value to an argument.

f). In parameters list, we can give default values to one or more parameters.

Ex-1:

```
def display(x='Gudio Van Rossum',y=70):       print('Name of the
        person=',x)       print('Age of the person=',y)

display('ramkumar',40)
display('venkat',55)
display()
```

Ex-2:

```
 def country(name='INDIA'):
        print('current country name = ',name)

country( )
country('NEWYORK')
country('CHINA')
country()
```

Ex-3:

```
def printlist(upperlimit=4):
        print('upper limit = ',upperlimit)     list1=list(range(upperlimit))
        print('list = ',list1)

printlist ()
```

```
printlist(5)
printlist(3)
printlist( )
```

Ex-4: one argument is already deined in a formal argument,
and only one argument has to pass.

```
def person(name,age=45):
    print(name)
    print(age)

person("kusu Srinivas")
```

4. <u>Variable length arugments</u>:

a). The number of arguments is not important

b). The position is also not important

c). Variable length arguments are also known as arbitrary arguments.

d). Sometimes, we do not know in advance the number of arguments that will be passed into a function.

e). python allows us to handle this kind of situation through function calls with arbitrary number of arguments or variable length argument.

f). In the function definition we use an asterisk (*) before the parameter name to denote this kind of argument.

g). This type of argument is used to pass multiple values.

Ex-1:

```
def calculate(*x):
    sum=0
    print(x)
    print(type(x))
    for i in x:
    sum=sum+i
    print('sum of elements=',sum)
```

```
calculate(11,22,33,44)
calculate(5,8)
calculate(1,2,3,4,5,6,7,8,9,10)
calculate()
calculate(11)
calculate(5,8,9)
```

Ex-2:

```
def printnames(*x):
        for i in x:
    print(i)

printnames('ram') printnames('laxman','anji')
printnames()
printnames('ravana','bharat','laxman','anji')
```

Ex-3:

```
def largest(*x):
    return max(x)

print(largest(20,30))
print(largest(2,5,3))
print(largest(9,45,12,18,6))
print(largest(10,40,80,50))
print(largest(16,3,12,44,40))
```

Ex-4:

```
def sum(x,*y):
        a=x+y
    print(a)


sum(5,6,7,2,8)
```

Here, the tip is that when we pass multiple values, then the first value goes to the first formal argument, and the star used with the second formal argument is to accept all other values in it.

5.<u>Keyword-variable length arguments:</u>

a). The special syntax **kwargs in function definitions in python is used to pass a keyworded, variable-length argument list.

b). We use the name kwargs with the double star. The reason is because the double star allows us to pass through keyword arguments (and any number of them).

c). A keyword argument is where you provide a name to the variable as you pass it into the function.

d). One can think of the kwargs as being a dictionary that maps each keyword to the value that we pass alongside it. That is why when we iterate over the kwargs there doesn't seem to be any order in which they were printed out.

Ex-1: usage of **kwargs:

```
def f1(**x):
        print(x)
        print(type(x))
        print(x.keys())
        print(x.values())
        for k, v in x.items():
                print ("%s == %s" %(k, v))
f1(f ='ram', s ='laxman', t='sita')
```

Ex-2:

```
def employeeinfo(**data):
        for i,j in data.items():
                print('thevalue{} is {}'.format(i,j))

employeeinfo(name='ram',age=40,sal=56000) employeeinfo(name='venkat',age=67)
employeeinfo(name='siva',sal=99000)
```

Recursion Functions:

We know that in Python, a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions. a). A function that calls itself. It is called a recursion.

b). Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

c). The base condition in the form of if-else statement.

Ex-1:

```
        def wish():
                print('good morning')    wish()
```

```
wish()
```

Note: the above function calls itself .

Ex-2:

```
def wish(x):
    if x<12:
        print('good morning at time',x)       wish(x+1)
    else:
        return
wish(6)
```

Ex-3: write a recursive function to find the factorial of a number

```
def  calc_factorial(x):

    """This is a recursive function     to find the factorial of an integer"""
    if x == 1:
        return 1
    else:
        return x * calc_factorial(x-1)
num = 4
print("The factorial of", num, "is", calc_factorial(num))
```

In the above example, calc_factorial() is a recursive functions as it calls itself.

When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function call multiples the number with the factorial of number 1 until the number is equal to one. This recursive call can be explained in the following steps.

```
calc_factorial(4)               # 1st call with 4 4 *
calc_factorial(3)               # 2nd call with 3

4 * 3 * calc_factorial(2)       # 3rd call with 2 4 * 3
* 2 * calc_factorial(1)   # 4th call with 1

4 * 3 * 2 * 1                   # return from 4th call as number=1
```

```
4 * 3 * 2                    # return from 3rd call
4 * 6                        # return from 2nd call
24                           # return from 1st call
```

Our recursion ends when the number reduces to 1. This is called the base condition.

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.

Lambda Functions:

Anonymous functions,which are also called lambda functions because they are not declared with the standard def keyword.

1. lambda functions are declared by using the lambda keyword.

2. labmda functions are also called anonymous functions or name less functions.

3. In Python, anonymous function is a function that is defined without a name.

4. A lambda function can take any number of arguments , but can only one statement(expression).

5. The syntax is :

```
lambda argumentslist: expression
```

Here,

lambda is a keyword
argumentslist is a list of arguments
expression is only one statement

Ex-1: write a lambda function find the addition of two numbers:

```
c=lambda x,y: x+y
```

```
a=int(input('enter first number')) b=int(input('entersecond
        number'))
        print('addition of two numbers=',c(a,b))
```

Ex-2: write a lambda function to find the cube of a given number:

```
cube= lambda x: x*x*x
a=int(input('enter any  number'))
print('cube of a given number=',cube(a))
```

Ex-3: write a lambda function find the biggest of two numbers:

```
big= lambda x,y: x if x>y  else y
a=int(input('enter first number')) b=int(input('enter second number'))
print('biggest of two numbers=',big(a,b))
```

Ex-4: write a lambda function to print given string

```
x="some kind of a useless lambda"
(lambda x : print(x))(x)
```

Ex-5: write a lambda function to print squres of numbers between 1 to 10 numbers

```
s= lambda n:n*n
for i in range(1,11):
print('The square of {} is:{}'.format(i,s(i)))
```

Uses of Lambda Function

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

filter( ):

1. This function is used to select some particular elements from a sequence of elements.
2. The sequence can be any iterator like lists, sets, tuples, etc.
3. The elements which will be selected is based on some pre-defined constraint. It takes 2 parameters:

syntax: filter (function, sequence)

a). A function that defines the filtering constraint

b). A sequence (any iterator like lists, tuples, etc.)

Ex:

```
x = [10,2,8,7,5,4,3,11,0, 1]
fs = filter (lambda i: i > 4, x)
print(list(fs))
```

1. In the first statement, we define a list called sequences which contains some numbers.
2. Here, we declare a variable called filtered result, which will store the filtered values returned by the filter() function.
3. A lambda function which runs on each element of the list and returns true if it is greater than 4.
4. Print the result returned by the filter function.

map( ):

1. This map function is used to apply a particular operation to every element in a sequence.
2. A function that defines the op to perform on the elements One or more sequences
3. Like filter(), it also takes 2 parameters 4. syntax: map (function, sequence)

a). function is a lambda function

b). sequence may be list, tuple, set

Ex: To print the squares of numbers in a given List:

```
x= [10,2,8,7,5,4,3,11,0, 1]
fr = map (lambda i: i*i, x)  print(list(fr))
```

Here, we define a list called sequences which contains some numbers.

We declare a variable called filtered result which will store the mapped values

A lambda function which runs on each element of the list and returns the square of that number.

Print the result returned by the map function.

reduce():

The reduce function, like map (), is used to apply an operation to every element in a sequence. However, it differs from the map in its working. These are the steps followed by the reduce () function to compute an output:

Step 1) Perform the defined operation on the first 2 elements of the sequence.

Step 2) Save this result

Step 3) Perform the operation with the saved result and the next element in the sequence.

Step 4) Repeat until no more elements are left.

MODULES

1. In python, Module is nothing but a file.
2. A module contains variables, constants, functions, classes and objects.
3. python modules are 2 types:
   a). Built-in modules

   b). User-defined modules

a). Built-in modules:

These modules are coming from python software internally.

Ex: math, copy, os, platform, random,

keyword, datatime, date, time, calendar

tkinter, mysql,sqlite3

numpy, scipy, pandas, matplotlib, matrix,pillow

math module:

It is a built-in module. To perform mathematical operations:

Examples:

```
import math
print(math.pi
print(math.sqrt(25))
print(math.sqrt(100)
print(math.ceil(5.6)
print(math.floor(5.6)
print(math.factorial(5)
print(math.pow(4,3)
print(math.prod([1,2,3,4])
print(math.prod(range(1,11))
```

math.pi - an approximation of the ratio of the circumference to the diameter of a circle.

math.e - an approximation of the base of the natural logarithm.

math.ceil(x) - Returns the smallest integer greater than or equal to it, as a float.
math.fabs(x) - Returns the absolute value of x.

math.factorial(x) - Returns x factorial, which is 1 * 2 * 3 * ... *x.

math.floor(x) - Returns the greatest integer less than or equal to x as a float.

math.exp(x) - Returns exp vaue.

math.log(x) - Returns ln x.

math.log(x, base) - Returns log base x.

math.sqrt(x) - Returns the square root of x.

math.cos(x) - Returns the cosine of x radians.

math.sin(x) - Returns the sine of x radians.

math.tan(x) - Returns the tangent of x radians.

Keyword module:

It is a built-in module. It contains following functions.

kwlist: It displays the list of keywords.

iskeyword(): It is a function. the given argument is a keyword or not.

It returns either True/False.

If it is a keyword it returns True.otherwise it returns False.

Ex-1:Write a program to display all python keywords.

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

Ex-2: Write a program to find given word is keyword or not.

```
import keyword
w=input('enter any word')
if keyword.iskeyword(w):
```

```
    print('given word is keyword')
else:
    print('given word is not a keyword')
```

Calendar module:

This module displays a calendar information.

write a program to display a calendar of a given month.

```
import calendar
m=int(input('enter month number')) y=int(input('enter year:'))
print(calendar.month(y,m))
```

write a program to display a year calendar.

```
import calendar
y=int(input('enter year:'))
print(calendar.calendar(y))
```

2. User-defined modules:

These modules developed by programmer as per the requirements.

Ex-1**:** To create user defined module with variables and functions.

```
a=88 b=99
def f1()
   print('i am f1 of first module')
def f2():
   print('i am f2 of first odule')
def f3():
   print('i am f3 of first module')
```

run the module:

```
import first  first.a  first.b
first.f1()  first.f2()
first.f3()
```

<u>Various forms of import statement:</u>

syntaxes

1. import modulename
2. import modulename as aliasname

3. import modulename-1, modulename-2,.....modulename-n
4. import modulename-1 as alias-1, modulename-2 as alias-2, modulenam-3 as alias-3....modulename-n as alias-n
5. from modulename import all
6. from modulename import specified items

examples:

1. import math
2. import math as m
3. import math, keyword,random,calendar
4. import math as m, keyword as k,random as r,calendar as c
5. from math import *
6. from math import pi,e,sqrt

Ex: write a program to create a user-defined module: first.py:

```
m=10
n=20
a=100
b=200
def add():
    print(m+n+a+b)
def sub():
    print(m-n)
def mul():
    print(m*n)
```

Datetime module:

This module is used to perform the operations on date and time.

ex-1: To display present date and time

```
import datetime dt=datetime.datetime.now()
print(dt)
```

ex-2: To display present date and time

```
import datetime dt=datetime.datetime.today()
print(dt)
```

ex-3:To display day,weekday.

```
import datetime dt=datetime.datetime.today()
print(dt.day)
print(dt.weekday())
print(dt.isoweekday())
```

ex-4:To display day,month and year

```
import datetime
dt=datetime.datetime.today() print(dt.day)
print(dt.month) print(dt.year)
```

strftime(): We used the "strftime function" for formatting.

This function uses different control code to give an output.

Each control code resembles different parameters like year,month, weekday and date [(%y/%Y – Year), (%a/%A- weekday), (%b/%B- month), (%d - day of month)] .

In our case, it is ("%Y") which resembles year, it prints out the full year with the century (e.g., 2018)

ex-5: to display date and time different formats:

```
import datetime x=datetime.datetime.now() print(x)
print(x.strftime("%H")) #24 hours format print(x.strftime("%a"))   #week day short version
print(x.strftime("%A"))   #week day long version print(x.strftime("%w"))   #week day as a
number print(x.strftime("%d")) #day of month print(x.strftime("%b"))   #month name
short version print(x.strftime("%B"))   #month name long version print(x.strftime("%m"))
#month has number print(x.strftime("%y"))   #year short version print(x.strftime("%Y"))
#year long version print(x.strftime("%I"))   #12 hours format print(x.strftime("%p"))
#am/pm print(x.strftime("%M"))   #to display minutes.
print(x.strftime("%S"))   #to display seconds print(x.strftime("%f"))   #micro seconds
print(x.strftime("%j"))   #number of days completed print(x.strftime("%U"))   #week
number in the year print(x.strftime("%x"))   #local version of date print(x.strftime("%X"))
#local version of time
```

EXCEPTION HANDLING

An **exception** is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a **Python** script encounters a situation that it cannot cope with, it raises an **exception**. An **exception** is a **Python** object that represents an **error**.

Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.

In python programming language, there are three types of errors. These are

1. Syntax Errors
2. Logical Errors or Semantic Errors
3. RunTime Errors

1. Syntax Errors

These are the most basic type of errors. Syntax errors are almost always fatal. In IDLE, it will highlight where the syntax error is. Most syntax errors are typos, incorrect indentation/incorrect arguments.

Example:

```
print(Hello, World!) print("Hello')
print('Hello")
Name="Kusu srinivas
```

Here are some ways to avoid the most common syntax errors:

1). Never use a Python keyword for a variable name

2). Any strings in the code should have matching quotation marks

3). Always check unclosed opening operators - (,{,(

4). .Always check the indentation

5). Don't give your module the same name as one of the standard Python modules.

==> syntax errors will prevent execution of the program.

In this example, the if statement is missing a colon to end the statement.

```
if x>9
    print(' x is above 9')
```

2. Logical Errors:

Your program might run without crashing (no syntax or run-time errors), but still do the wrong thing.

These errors are also semantic errors.

a). semantic errors are errors that happen as a result of one or more

problems in logic.

b). These errors can be more complex, because no error is generated.

c). The code runs but generates unexpected and/or incorrect output, or no output.

d) . A classic example of this , would be an infinite loop,

that most new programmers experience at least once.

Example:

```
x = 3
y = 4
avg = x + y / 2
print(avg)
```

but the program prints 5.0 instead! x+ y / 2, this has the same mathematical meaning as x + (y / 2) = 3 + (4 / 2) = 3 + 2 = 5. To fix the problem

Example

```
a = 1,000,000
print()
```

The output is :(1, 0, 0)

Python interprets 1,000,000 as a comma separated sequence of integers.This is the first example we have seen of a semantic error: the code runs without producing an error message.

3. RunTime Errors:

Python is able to understand what the program says, but runs into problems when actually performing the instructions. Like spelling mistakes or invalid Methods etc...!!

Example:

```
x=500 printf(x)

CallMe="KSR"
print(callme)
```

Common RunTime Errors in PYTHON:

NameError: Occurs when programmer uses a variable that doesn't exist in the current environment

TypeError: Occurs when programmer uses a value improperly

KeyError: Occurs when programmer is accessing an element of a dictionary using a non-existent key

AttributeError: Occurs when programmer is accessing an attribute or method that does not

exist

IndexError: The index being used to access a list, string, or tuple is greater than its length minus one

Eg:Abnormal termination: The concept of terminating the program or script in the middle of its execution without executing last statement(s) of the program is known as abnormal termination.

```
a=input("Enter any Number") b=input("Enter any
Number") c=a/b
print("The Value is: " ,c)
print("Bye")
```

NOTE: TypeError: unsupported operand type(s) for /: 'str' and 'str'

Abnormal termiation is the undesirable situation in any Programming language.

How to handle Errors:

Python provides two important features to handle any unexpected error in your Python programs:

1 Exception Handling:
2 Assertions

What is an Exception?

An exception is an error that happens during execution of a program. When that error occurs, Python generate an exception that can be handled, which avoids your program to crash.

Common Exception Errors

1 except IOError:
print('An error occurred trying to read the file.')

2 except ValueError:
print('Non-numeric data found in the file.')

3 except ImportError:
print("NO module found")
except EOFError:
print('End Of File No Data')

4 except KeyboardInterrupt:
print('You cancelled the operation.')

6. except DivisionByZeroError:

print('An error occurred.')

Example:

number=int(input("Enter Any Number between 1-10: ")) print("You Are Entered: ", number)

Note: This program works perfectly as long as the user enters a number, If not It terminates?

Python Built-in Exceptions	
Exception	**Cause of Error**
AssertionError	Raised when assert statement fails.
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when the input() functions hits end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raise when a generator's close() method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits interrupt key (Ctrl+c or delete).
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in local or global scope.
NotImplementedError	Raised by abstract methods.
OSError	Raised when system operation causes system related error.
OverflowError	Raised when result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access a

	garbage collected referent.
RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Raised by next() function to indicate that there is no further item to be returned by iterator.
SyntaxError	Raised by parser when syntax error is encountered.
IndentationError	Raised when there is incorrect indentation.
TabError	Raised when indentation consists of inconsistent tabs and spaces.
SystemError	Raised when interpreter detects internal error.
SystemExit	Raised by sys.exit() function.
TypeError	Raised when a function or operation is applied to an object of incorrect type.
UnboundLocalError	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
UnicodeError	Raised when a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError	Raised when a Unicode-related error occurs during encoding.
UnicodeDecodeError	Raised when a Unicode-related error occurs during decoding.
UnicodeTranslateError	Raised when a Unicode-related error occurs during translating.
ValueError	Raised when a function gets argument of correct type but improper value.
ZeroDivisionError	Raised when second operand of division or modulo operation is zero.

Why use Exceptions?

Exceptions are convenient in many ways for handling errors and special conditions in a program. When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates.

Syntax-1:

```
try:
        Some Statements Here
except:
        Exception Handling
```

Ex-1:

```
try:
        x=int(input("Enter a value: "))
        print("Try Block:Enter Number is: ",x)
except ValueError:
        print("ExceptBlock:Invalid Input Enter \    Only Numbers");
```

Ex-2:

```
try:
        x=int(input("Enter Required Number: "))      y=int(input("Enter
        Required Number: "))
        z=x/y
        print("The Result is: ",int(z))
except ZeroDivisionError:
        print('you cannot divide by zero.')
        print("ExceptDisplayed")
```

Ex-3:

```
try:
            x=int(input("Enter Required Number: "))
            y=int(input("Enter Required Number: "))   print("The Result
            is: ",x/y)
except Exception as msg:
            print("You can't divide by zero.")
            print('Error: ', msg)
```

Ex-4:

```
try:
```

```
        p=int(input("Enter Any Number: "))     q=int(input("Enter Any
        Number: "))
        r=p/q
        print("The Value is: ",int(r))     print("TryBlockIsExecuted:")
    except ValueError:
        print("ExceptBlock")
        print("Sorry User Numbers Only Valid")
    except Exception as arg:
        print("You can't divide by zero.")  print('Error: ', arg)
```

Syntax-2:

```
        try:
          You do your operations here;
          ..... except Exception-I:
          If there is ExceptionI, then execute this block.
          .....
        else:
          If there is no exception then execute this block.
```

Ex:

```
        try:
            fi=open("Hai1.txt",'r')
            print(fi.read(6))
        except IOError:
            print("File Not Existed Please Create")
        else:
            print("ContentReadSuccessfully")     fi.close()
```

NOTE: If file Not Existed It returns except IOError

The except Clause with No Exceptions:

You can also use the except statement with no exceptions:

Syntax-3:

```
        try:
          Do operations here;
        except:
```

If any exception, then it executes this block.

else:

If no exception then it executes this block.

Example:

```
try:
 x=int(input("Enter Required Number: "))   y=int(input("Enter
Required Number: "))   print("The Result is: ",int(x/y)) except:
   print("Arithmetic Exception Raised.")
else:
   print("SuccessfullyDone")
```

The try-finally Clause:

You can use a finally: block along with a try: block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not.

```
try:
  Do operations here;
finally:
  always be executed.
```

NOTE: You cannot use else clause, along with a finally clause.

Example:

```
try:
     fp = open("MyFile.txt",mode='r')
     print(fp.read(4))
finally:
  fp.close()
  print("This Block Always get Executed")
```

Nested Try Block:

Syntax:

```
try:
  Do operations here;   try:
   Do operations here; finally:
   Executed Always
```

Eg:

```
try:
  fp = open("MyFile.txt", mode="w")    try:
    fp.write("This is my test file for exception handling!!")
    print("FileCreated&WrittenSuccessfully")    finally:
    print("This Block Always get Executed")      fp.close() except
IOError:
  print("Error: can't find file or read data")
```

Assertions in Python:

(Automatic Error Detection) assert is the PYTHON Keyword. Python's assert statement helps you find bugs more quickly and with less pain. When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an AssertionError exception.

Syntax

```
assert expression, argument
 expression
Required. Expression to evaluate.
 argument
Optional. Argument passed to the exception raise
```

Asserts VS Try...Except:

Software Errors are Two Catetories:

1. Recoverable Errors (try ... except)
   - ⇨ User can take corrective action(Try Again or Choose Another Option)

2. Un-Recoverable Errors(assert)
   - ⇨ Not Enough information to fix or no alternative action is possible

Places to consider putting assertions:

1. Checking parameter types, classes, or values
2. Checking data structure invariants (never changed)
3. After calling a function, to make sure that its return is reasonable

TIP: Assertions are like airbags for your software. Always there, work automatically when you need them.

Use Assertions to write the following strategies: 1.
Pre-Conditions and Post-Conditions

2. The most powerful debugging tool.
3. Automating Debugging
4. It is powerful to catch all bugs or defects

Example:

```
assert 2 + 2 == 4 assert 2 + 2 == 3
assert 2 + 2 == 3,"That can't be right."
```

Example:

```
def power(x,y):
    assert x>0,"x Must be Positive Number not {0}"    assert y>0,"y Must
    be Positive Number not {0}"   return x**y
print(power(1,-2))
```

Example:

```
def GetAge(age):
    assert age>18,"Age Must not Be less than 18Years"
  print("You are Allow to Access: ",age)
GetAge(19)
```

Assertions are not a substitute for unit tests or system tests, but rather a complement. Because assertions are a clean way to examine the internal state of an object or function, they provide "for free" a clear-box assistance to a black-box test that examines the external behaviour.

FILES

File is a collection of related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk). RAM is volatile which loses its data when computer is turned off, we use files for future use of the data.

Files are very common permanent storage areas to store our data Types of files:

Thera are two types of files.

1. Text files: These files are used to store character data.
2. Binary files: These files are used to store binary data , like images , video files , audio files.

File operations: In Python, a file operation takes place in the following order.

1. Open a file
2. Read or write a file (perform operation)
3. Append a file
4. Close the file

Opening a File:

Python has an in-built function called open() to open a file. It takes a minimum of one argument as mentioned in the below syntax. The open method returns a file object which is used to access the write, read and other in-built methods.

Syntax:

```
file_object = open(file_name, mode)
```

Here, file_name is the name of the file or the location of the file that you want to open, and file_name should have the file extension included as well. Which means in test.txt – the term test is the name of the file and .txt is the extension of the file.

The mode in the open function syntax will tell Python as what operation you want to do on a file.

The allowed modes are

'r' : Read Mode: Read mode is used only to read data from the file.

'w' : Write Mode: This mode is used when you want to write data into the file or modify it. Remember write mode overwrites the data present in the file.

'a' : Append Mode: Append mode is used to append data to the file. Remember data will be appended at the end of the file pointer.

'r+' : Read or Write Mode: This mode is used when we want to write or read the data from the same file.

'w+' : To open a file for write and read data. It will override existing data.

'a+' : Append or Read Mode: This mode is used when we want to read data from the file or append the data into the same file.

'x' : To open a file in exclusive creation mode for write operation. If the file already exists then we will get FileExistsError.

Note: The above-mentioned modes are for opening, reading or writing text files only.

ex-1:

```
fo = open("C:/Documents/Python/test.txt", "r+")
```

In the above example, we are opening the file named 'test.txt' present at the location 'C:/Documents/Python/' and we are opening the same file in a read-write mode which gives us more flexibility.

ex-2:

```
f = open("abc.txt","w")
```

We are opening abc.txt file for writing data.

Closing a File:

After completing our operations on the file,it is highly recommended to close the file. For this we have to use close() function.

f.close()

The file Object Attributes

Once a file is opened and you have one file object, you can get various information related to that file.

Attribute	**Description**
file.closed	Returns True if file is closed, False otherwise.
file.mode	Mode in which the file is opened.
file.name	Name of the opened file.
file.readable( )	Returns boolean value indicates that whether file is readable or not
file.writable( )	Returns boolean value indicates that whether file is writable or not

eg:

```
# Open a file Method-I
fi = open("python.txt", mode="r") print("Name of the file: ",
fi.name) print("Closed or not : ", fi.closed) print("Opening mode : ",
fi.mode) print("file is readable: ",fi.readable( )      print("file is
writeable: ",fi.writable( )
fi.close( )
```

Writing data to text files:

We can write character data to the text files by using the following 2 methods.

```
write(str)
writelines(list of lines)
```

Ex:

```
f=open("abcd.txt",'w')
f.write("python")
f.write("apex\n")
```

```
f.write("java\n")
print("Data written to the file successfully")
f.close()
```

Note: In the above program, data present in the file will be overridden everytime if we run the program. Instead of overriding if we want append operation then we should open the file as follows.

Ex:

```
f=open("abcd.txt",'w')
list=["sunny\n","bunny\n","vinny\n","chinny"]
f.writelines(list)
print("List of lines written to the file successfully")
f.close()
Output: abcd.txt:
sunny bunny vinny
chinny
```

Note: while writing data by using write() methods, compulsory we have to provide line seperator(\n),otherwise total data should be written to a single line.

Reading Character Data from text files:

We can read character data from text file by using the following read methods.

```
read()          : To read total data from the file
read(n)         : To read 'n' characters from the file
readline()      : To read only one line
readlines()     : To read all lines into a list
```

E x-1: **To read total data from the file**

```
f=open("abc.txt",'r') data=f.read()
print(data)
f.close()
```

Ex-2: To read only first 10 characters

```
f=open("abc.txt",'r') data=f.read(10)
print(data)
f.close()
```

Ex-3: To read data line by line

```
f=open("abc.txt",'r') line1=f.readline()
print(line1,end='')
line2=f.readline()
print(line2,end='')
line3=f.readline()
print(line3,end='')
f.close()
```

Ex-4: To read all lines into list

```
f=open("abc.txt",'r')  lines=f.readlines()
for line in lines:  print(line,end='')
f.close()
```

Ex-5: To read all lines

```
f=open("abc.txt","r") print(f.read(3))
print(f.readline()) print(f.read(4))
print("Remaining data")
print(f.read())
```

Ex-6: write a prgoram read 'n' number of lines from a text file

```
fp=open("sample.txt","r") n=int(input("enter how manuy lines u
want") for i in range(1,n+1):
        l=fp.readline()
    print(l) fp.close()
```

The with statement:

The with statement can be used while opening a file.We can use this to group file operation statements within a block.

The advantage of with statement is it will take care closing of file,after completing all operations automatically even in the case of exceptions also, and we are not required to close explicitly.

Ex:

```
with open("abc.txt","w") as f:    f.write("kosmik\n")
        f.write("Techonologies\n")
    f.write("Near Jntu Hyderabad\n")    print("Is File Closed:
",f.closed)
print("Is File Closed: ",f.closed)
```

The seek( ) and tell( ) methods:

tell(): We can use tell() method to return current position of the cursor(file pointer) from beginning of the file. [can you plese tell current cursor position]

The position(index) of first character in files is zero just like string index.

Ex:

```
f=open("abc.txt","r") print(f.tell())
print(f.read(2)) print(f.tell())
print(f.read(3)) print(f.tell())
```

seek( ): We can use seek() method to move cursor(file pointer) to specified location.
[Can you please seek the cursor to a particular location]

f.seek(offset, fromwhere)

offset represents the number of positions

The allowed values for second attribute(from where) are

0: From beginning of file(default value)

1: From current position

2: From end of the file

Note: Python 2 supports all 3 values but Python 3 supports only zero.

Eg:

```
data="All Students are STUPIDS" f=open("abc.txt","w")
f.write(data) with open("abc.txt","r+") as f:
text=f.read() print(text)
print("The Current Cursor Position: ",f.tell())
f.seek(17)
print("The Current Cursor Position: ",f.tell())
f.write("GEMS!!!")
f.seek(0) text=f.read()
print("Data After Modification:") print(text)
```

Write a program to check whether the given file exists or not. If it is available then print its content?

```
import os,sys
    fname=input("Enter File Name: ")    if
    os.path.isfile(fname):
        print("File xists:",fname)
        fP=open(fname,"r")
    else:
        print("File does not exist:",fname)    sys.exit(0)
      print("The content of file is:")    data=f.read()
  print(data)
```

Q: Program to print the number of lines,words and characters present in the given file?

```
import os,sys
fname=input("Enter File Name: ")        if os.path.isfile(fname):
    print("File exists:",fname)                f=open(fname,"r")
else:
    print("File does not exist:",fname)
    sys.exit(0)        lcount=wcount=ccount=0
for line in f:            lcount=lcount+1
        ccount=ccount+len(line)    words=line.split()
    wcount=wcount+len(words)
print("The number of Lines:",lcount)    print("The number of Words:",wcount)
print("The number of Characters:",ccount)
```

Handling Binary Data:

It is very common requirement to read or write binary data like images,video files,audio files etc.

While using binary files, we have to use the same modes with the letter 'b' at the end. So that Python can understand that we are interacting with binary files.

'wb' : Open a file for write only mode in the binary format.

'rb' : Open a file for the read-only mode in the binary format.

'ab' : Open a file for appending only mode in the binary format.

'rb+' : Open a file for read and write only mode in the binary format.

'ab+' : Open a file for appending and read-only mode in the binary format.

Write a Program to read image file and write to a new image file?

```
f1=open("rossum.jpg","rb")
f2=open("newpic.jpg","wb") bytes=f1.read()
f2.write(bytes)
print("New Image is available with the name: newpic.jpg")
```

Handling csv files:

CSV means Comma seperated values

As the part of programming,it is very common requirement to write and read data wrt csv files. Python provides csv module to handle csv files.

Writing data to csv file:

```
import csv with open("emp.csv","w",newline='') as f:
w=csv.writer(f) # returns csv writer object
w.writerow(["ENO","ENAME","ESAL","EADDR"])  n=int(input("Enter Number of Employees:"))
for i in range(n):
    eno=input("Enter Employee No:")
    ename=input("Enter Employee Name:")    esal=input("Enter Employee Salary:")
        eaddr=input("Enter Employee Address:")
    w.writerow([eno,ename,esal,eaddr])
print("Total Employees data written to csv file successfully")
```

Reading Data from csv file:

```
import csv f=open("emp.csv",'r')
r=csv.reader(f) #returns csv reader object data=list(r) #print(data)
for line in data:
    fo
```

```
    r word in line:
        print(word,"\t",end='')
    print()
```

OOPS

Procedure-oriented vs Object-oriented Programming languages

Procedural Programming:

1. Procedural programming uses a list of instructions to do computation step by step.
2. In procedural programming, It is not easy to maintain the codes when project becomes lengthy.
3. It doesn't simulate the real world. It works on step by step instructions divided in small parts called functions.
4. Procedural language doesn't provide any proper way for data binding so it is less secure.

Ex. of procedural languages are: C, Fortran, Pascal, VB,COBOL ...etc.

Object-oriented Programming:

1. Object-oriented programming is an approach to problem solving where computation is done by using objects.
2. It makes development and maintenance easier.
3. It simulates the real world entity. So real world problems can be easily solved through oops.
4. It provides data hiding. so it is more secure than procedural languages. You cannot access private data from anywhere.

Eg: C++, Java, .Net, Python, C#, R, APEX..... etc.

PYTHON Object Oriented Programming System:(OOPS)

1. Python is an object oriented programming language.
2. In Python everything is an Object
   like:classes, objects, functions, methods, modules etc..
3. These can passed as arguments to functions or assign to variables.

Overview of OOP Terminology:

- **Class** : A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable** : A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Data member** : A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading** : The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.
- **Instance variable** : A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance** : The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance** : An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation :** The creation of an instance of a class.
- **Method** : A special kind of function that is defined in a class definition.
- **Object** : A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.

- **Operator overloading** : The assignment of more than one function to a particular operator.

Built-In Class Attributes:

Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute −

- **__dict__** : Dictionary containing the class's namespace.
- **__doc__** : Class documentation string or none, if undefined.
- **__name__** : Class name.
- **__module__** : Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- **__bases__** : A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

You can use **issubclass()** or **isinstance()** functions to check a relationships of two classes and instances.

- The **issubclass(sub, sup)** boolean function returns true if the given subclass **sub** is indeed a subclass of the superclass **sup**.
- The **isinstance(obj, Class)** boolean function returns true if *obj* is an instance of class *Class* or is an instance of a subclass of Class

OOPS concepts supported in python:

1. Classes and objects
2. Data encapsulation
3. Data abstraction
4. Constructors and destructors
5. Inheritance
6. Polymorphism

CLASSES AND OBJECTS

Creating Classes: A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object.

Some points on Python class:

1. classes are created by keyword class.
2. Attributes are the variables that belong to class.
3. Attributes are always public and can be accessed using dot (.) operator.

   Eg.: Myclass.Myattribute

4. A collection of variables(data members) and functions(methods) is called a class.

   Syntax:

   class classname:

   list of data mbers list of methods

Ex-1: To create a class only variables

```
class test:
    a=10
        b=20
print(test.a+test.b)
```

Ex-2: To create a class only methods

```
class wish:
        def mng():
           print('good morning')
        def aft():
                print('good afternoon')     def evng():

                print('good evening')          def ngt():
                  print('good night')
wish.mng()
wish.aft()
wish.evng()
wish.ngt()
```

#ex-3:To create a class with variables and methods . To find area of rectangle:

```
class rectangle:
        l=10
        b=5
        def find():
              rectangle.area=rectangle.l*rectangle.b
        def display():
                print('length=',rectangle.l)
                print('breadth=',rectangle.b)
                print('area=',rectangle.area)

rectangle.find()
rectangle.display()
```

Object:

1. Object is simply a collection of data (variables) and methods (functions) that act on those data. And, class is a blueprint for the object.

2. We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.
3. As, many houses can be made from a description, we can create many objects from a class.
4. An object is also called an instance of a class and the process of creating this object is called instantiation.
5. object creation syntax:

   object name = classname()

   Ex: r= rectangle() s=student()

Attributes in classes:

In computing, an attribute is a specification that defines a property of an object, element, or file. It may also refer to or set the specific value for a given instance of such.

There are two types of Attributes:

- Built- in Class Attributes
- Attributes defined by Users

Built-In Class Attributes:

Every Python class has five built-in attributes and they can be accessed using dot operator like any other attribute −

- **__dict__** : Dictionary containing the class's namespace.
- **__doc__** : Class documentation string or none, if undefined.
- **__name__** : Class name.
- **__module__** : Module name in which the class is defined. This attribute is

  "__main__" in interactive mode.

- **__bases__** : A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

For the above class let us try to access all these attributes

Ex:

```
class Employee:
    'Common base class for all employees'    empCount = 0
    def __init__(self, name, salary):
        self.name = name        self.salary = salary
        Employee.empCount += 1
    def displayCount(self):
        print("Total Employee %d" % Employee.empCount)    def
    displayEmployee(self):
        print("Name : ", self.name)
        print("Salary: ", self.salary)
    print("Employee.__doc__:", Employee.__doc__) print("Employee.__name__:",
    Employee.__name__) print("Employee.__module__:", Employee.__module__)
    print("Employee.__bases__:", Employee.__bases__) print("Employee.__dict__:",
    Employee.__dict__)
```

Attribute Defined by Users:

1. Attributes are created inside the class definition.
2. We can dynamically create new attributes for existing instances of a class.
3. Attributes can be bound to class names as well.

Ex:

```
class Employee():
    empCount=0        print(Employee.empCount)
obj=Employee()
print(obj.empCount)
obj.newCount=30
print(obj.newCount)
```

Access Modifiers:

Access modifiers are used to restrict or control the accessibility of class resources, by declaring them as public, protected and private.

- Private
- protected
- public

Private:

1. Private attributes can be accessed only within the class i.e. private attributes cannot access outside of the class.
2. By convention, We can declare member as private type, we write double underscore(_ _) symbol before the identifiers.

Protected:

1. Protected attributes can be accessed within the class anywhere but from outside of the class only in child classes.
2. By convention, We can declare member as protected type, we write single underscore(_) symbol before the identifiers.
3. These attributes should only be used under certain conditions

Public:

In python, By default, every member of class is public.

Public members are available publicly. Means can be accessed from anywhere.

These attributeswe can access from anywhere eithter inside the class or from outside of the class.

Ex:

```
class sample:
    a=10
    _b=20
    __c=30
    def display(self):
        print("inside the class c=",sample.__c)
```

```
s1=sample()
print(s1.a)
print(s1._b)
s1.display()
print(s1.__c)   #Error
```

Data Hiding in Python(Abstraction):

1. Data hiding means making the data private, so that it will not be accessible to the other class members. It can be accessed only in the class where it is declared.
2. In python, if we want to hide the variable, then we need to write double underscore (__) before the variable name.
3. It is the concept of hiding unnecessary details of objects and shows essential features to commmunicate.
4. Abstraction explains what an object can do instead of how it does it.
5. We can implement abstraction in python program using the pass statement.
6. The partial definition of an object is called abstrct class.
7. we need to override abstract methods in the child class.

Ex-1:

```
Class myClass:
        __num = 10
        def add(self, a):
                sum = self.__num + a        print(sum)
obj = MyClass()
obj.add(20)
#print(obj.__num)
```

Note: The above statement gives an error because we are trying to access private variable outside the class

E**x-2:**

```
class vehicle:

        def fuel(self):

                return True

        def ac(self):

              pass
```

```
class bike(vehicle):
        def ac(self): #override
                return False
class car(vehicle):
        def ac(self):  #override
                return True
class abstraction:
        def main():
        c=car()
        print('car fuel:',c.fuel())         print('car ac:',c.ac())
        b=bike()
        print('bike fuel:',b.fuel())         print('bike ac:',b.ac())
        return()
abstraction.main()
```

The self Parameter:

1. The self parameter is a reference to the class itself .
2. And is used to access variables that belongs to the class.
3. It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class.
4. Self is not a keyword.

Ex:

```
        class person:
           def __init__(mysillyobject, name, age):
              mysillyobject.name = name
                 mysillyobject.age = age
              def myfunc(abc):
              print("Hello my name is " + abc.name)
        p1 = person("John", 36)
        p1.myfunc()
```

Encapsulation

1. It is the concept of writing data and code into a single unit.
2. you can implement of encapsulation using class.
3. it is the concept of protecting the information from outside of access.
4. The communication between the objects must be allowed through the functionaly only.

Ex: encapsulation class

```
{       data:variables
code:methods
}
```

Rules to implement encapsulation:

1. class is public . it is visible to all the other objects in the communication world.
2. properties are private
3. Initialization using constructor
4. setter() and getter() methods for every property need to define eg:

```
class employee:
    def _init__(self,x,y,z):
        self.__empno=x
        self.__ename=y
        self.__sal=z
    def display(self):
        print(self.__empno)        print(self.__ename)
        print(self.__sal)
e1=employee(101,'ram',78900.00) e1.display()
```

__init__():

1. The __init__() function is a built-in function.
2. This function is exectued automatically whenever the object is created.
3. No need of explicit calling.
4. The main purpose the init() function is to initialize the varaible values.

eg: class person:

```
def __init__(self,name):

    self.name=name

def display(self):              print("Hello",self.name)
  return
```

```
p1=person('ramarao')
p1.display()
```

Constructors:

1. A constructor is a class function that instantiates an object to predefined values.

2. A consturctor is a special method. It begins with a double underscore (_ _).
3. In python constructors are implemented by using the __init__()
4. The init() method is called the constructor and is always called when creating an object.
5. Contructors are 2 types.
   a) Default constructor
   b) parameterized contructor

a). **Default constructor**:

To create a object without parameters is called a default constructor.

Ex-1:

```
class test:
        def __init__(self):
                print('i am test class initfunction')
t1=test()
```

Ex-2:

```
class sample:
        def __init__(self):
                print('i am default constructor')
s1=sample()
```

Ex-3: write a program to count no. of obejcts create a class

```
class test:
        count=0
        def __init__(self):
                test.count=test.count+1
        def display():
                print('no. of objects=',test.count)
t1=test()
t2=test()
t3=test()
t4=test()
t5=test()
t6=test()
test.display()
```

b) parametrized contructor :

To create a obect with parameters is called a parametrized constructor.

Ex-1:

```
class User:
        name = ""
        def __init__(self, name):
                self.name = name     def sayHello(self):
                print("Welcome to KKRCoders, " + self.name)

        User1 = User("ramkumar")
        User1.sayHello()
```

Ex-2:

```
class car:
        def __init__(self,x,y,z):
                self.color=x        self.companyname=y
           self.model=z

        def display(self):
                print('car color = ',self.color)        print('car company name =
                ',self.companyname)        print('car model name = ',self.model)

c1=car('red','maruth','swift')
c1.display()
```

Destructors in Python:

1. Destructors are called when an object gets destroyed.
2. In Python, destructors are not needed as much needed, in C++ or java , because Python has a garbage collector that handles memory management automatically.
3. The __del__() method is a known as a destructor method in Python.

Syntax of destructor declaration :

```
        def __del__(self):

                body of destructor
```

eg: Python program to demonstrate destructor

```
        class employee:
           def __init__(self):
```

```
                print('Employee created.')          def __del__(self):
            print('Destructor called,Employee  deleted.')
        e1 = employee()
        del e1
```

Difference between Function and Method in PYTHON:

Function:

1. It is always independent, not related to any Class
2. Outside the class a piece of code
3. No Self Parameter required
4. No Instance required to call

Method:

1. It is always related to a class
2. Inside the class a piece of code
3. Self Parameter required
4. Always instance required to call

INHERITANCE

It is the process of inheriting the class members from one class to another class is called Inheritance.

OR

The concept of using properties of one class into another class without creating object of that class explicitly is known as inheritance.

1. A class which is extended by another class is known as 'super class'.
2. Both super class property and sub class property can be accessed through sublcass reference variable

Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class.

Terminology:

Parent class ==> Base class ==>Super class
Child class ==> Derived class==> Sub class

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. syntax to inherit a base class into the derived class:

Syntax:

```
class derived-classname(base classname):
        list of variables
        list of methods
```

Types of Inheritance:

1. Single Level Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

Single Level Inheritance:

The concept of inheriting properties from only one class into another class is known as a single inheritance.

Syntax :

```
class BaseClassname:
        Body of base class class
DerivedClassname(BaseClassname):        Body of derived
class
```

Ex: program to implement single inheritance

```
 class animal:
            def speak(self):
                print("Animal Speaking")   class
dog(animal):
        def bark(self):
        print("dog barking")
d = dog()
d.bark()
d.speak()
```

Multi-Level inheritance:

Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

Python Inheritance

The syntax of multi-level inheritance is given below.

Syntax:

```
class class1:
```

```
        <class-suite>
class class2(class1):
        <class suite>
class class3(class2):
        <class suite>
                        .
                        .
class classn(classn-1):
        <class suite>
```

Ex: program to implement multi level inheritance

```
class animal:
        def speak(self):
                print("Animal Speaking")
class dog(animal):
        def bark(self):
                print("dog barking")
class dogchild(dog):
        def eat(self):
                print("Eating bread...")
d = dogchild()
d.bark()
d.speak()
d.eat()
```

Multiple inheritance:

The class which inherits the properties of multiple classes is called Multiple Inheritance.

When we have one child class and more than one parent classes then it is called multiple inheritance. i.e. when a child class inherits from morethan one parent class.

Syntax:

```
class Base1:
        body of the base-1 class
```

```
class Base2:
        body of the base-2 class
                ------
                        -------
class BaseN:
        body of the base-n class

class Derived(Base1, Base2, ..... BaseN):
        body of the derived class
```

Ex: program to implement multiple inheritance

```
class Calculation1:
    def Summation(self,a,b):
        return a+b




class Calculation2:
    def Multiplication(self,a,b):
        return a*b
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b

d = Derived()
print(d.Summation(10,20))   print(d.Multiplication(10,20))
print(d.Divide(10,20))
```

Hirarchical Inheritance:-

The concept of inheriting properties from one class into multiple classes is known as a hierarical inheritance.

Syntax:

```
class A:
        body of base class
```

```
class B(A):
        body of derieved-1 class
class C(A):
    body of derived-2 class
```

Ex:program to implement hierachical inheritance

```
class hello():
        def myone(self):
                print("I am in Hello Class")
class hai(hello):
        def mytwo(self):
                print("I am in Hai Class")
class bye(hello):
        def mythree(self):
                print("I am in Bye Class")
h=hai( )
h.MyOne( )
h.mytwo( )
b=bye( )
b.myone( )
b.mythree( )
```

Hybrid Inheritance:

Hybrid Inheritance is that type in which we combine two or more types of inheritance.

OR

It is the combination of sinlge+Hirarchical+Multiple+Multilevel Inheritances are called Hybrid Inheritance

Syntax:

```
class A:
        body of class A
class B(A):
        body of class B
class C(A):
        body of class C:
class D(B, C):
                body of class D
```

Ex:program to implement HYBRID inheritance

```
class A:
        def m1(self):
```

```
                print("method from Class A....")
class B(A):
        def m2(self):
          print("method from Class B....")

class C(A):
        def m3(self):
          print("method from Class C....")
class D(B, C):
        def m4(self):
        print('method class')

d=D()
d.m1()
d.m2()
d.m3()
d.m4()
```

Method Overloading in Python

Multiple methods with the same name but with a different type of parameter or a different number of parameters is called Method overloading.

Ex:

```
def product(a, b):
        p = a*b
        print(p)
def product(a, b, c):
        p = a*b*c

        print(p

 product(2, 3)

product(2, 3, 5)
```

Method overloading is not supported in Python, because if we see in the above example we have defined two functions with the same name 'product' but with a different number of parameters.

But in Python, the latest defined will get updated, hence the function product(a,b) will become useless.

Method Overriding in Python

If a subclass method has the same name which is declared in the superclass method then it is called Method overriding

To achieve method overriding we must use inheritance.

Ex:

```
class A:
        def sayHi():
                print("I am in A")
class B(A):
        def sayHi():
                print("I am in B")


ob = B()
ob.sayHi()
```

POLYMORPHISM

1. Poly means many and morphism means forms
2. Forms means functionalities or logics.
3. The concept of defining multiple logics to perform same operation is known as a polymorphism.
4. Polymorphim can be implemented in python by using mehtod overriding.

5. Python is implicitly polymorphic

Eg:

Volkswagen	Sports Car
Beetle	Van
start()	start()

Driver

<== Driver can call start() method in either class even though they are unrelated.

Ex:

```
+ Operator:
A=5, B=6
print(A+B)
A="Hello"
B="PYTHON" print(A+B)
```

Polymorphism different forms:

1. Compile-time polymorphism
2. Runtime polymorphism

1. Static or Complile time polymorphism

Method Overloading:

NOTE: PYTHON does not supports Method Overloading, It is dynamically typed language.

Ex:

```
class CalCulate():
        def add(self,a,b):
                return a+b
        def add(self,a,b,c):
            return a+b+c
obj=CalCulate() print(obj.add(3,1,3))
print(obj.add(3,1))
```

NOTE: TypeError: add() missing 1 required positional argument: 'c'

NOTE: In Python it always calls latest implemention of the method

Ex:

```
class dog():
        def sound(self):
                print("Barking")
class cat:
        def sound(self):
                print("meow")
        def makesound(animaltype):
                animaltype.sound()
catobj=cat()
dogobj=dog()
makesound(catobj)
makesound(dogobj)
```

2. Dynamic Polymorphism: Method overriding :

1. what ever members avaialable in the parent class are by default available to the child class through inheritance. if the child class not satified with parent class implementation then child class is allowed to redefine that method in the child class based on its requirement. this concept is called overriding.

2. overriding concept applicable for both methods and constructors.

3. in the above program always executed child class constructor.

if we want to execute parent class constructor, we can implement by using super().

Ex-1: to demonstrate constructor overriding:

```
class person:
        def __init__(self):
                print('i am base class constructor')
class child(person):
        def __init__(self):
                print('i am child class constructor')        super().__init__()
c1=child()
```

Ex-2: To demonstrate method overriding

```
class A():
        def display(self):
                print("Method belongs to Class A")
class B(A):
        def display(self):
                print("Method belongs to Class B")

b1=B()
b1.display()
```

write a program to demonstrate overriding with person employee classes

```
class person:
        def __init__(self,x,y):
                self.name=x
                self.age=y
class employee(person):
        def __init__(self,a,b,c,d,e):
             super().__init__(a,b)
                 self.empno=c
                 self.sal=d
                self.city=e
        def display(self):
                print("employee information is:")        print("employee number=",self.empno)
                print("employee name=",self.name)        print("employee salary =",self.sal)
                print("employee age =",self.age)         print("empoyee city=",self.city)

e1=employee('ramarao',25,1005,78000,'hyd')
e1.display()
```

Operator Overloading

Assigning extra work to operators is called operators overloading

OR

You can change the meaning of an operator in Python depending upon the operands used. That allows same operator to have different meaning according to the context is called operator overloading.

OR

we can use the same operator for multiple purposes. which is nothing but operator overloading. python supports operator overloading.

ex-1: + acts as concatenation of two strings and arithmetic addition

```
x='python'

y='kosmik

z=x+y

print(z)


a=10

b=20

c=a+b

print(c)
```

ex-2: * acts as repetition and arithmetic multiplicaiton

```
x='python'

print(x*10)


a=9
```

```
b=3
print(a*b)
```

magic methods:

1. Internally + operator is implemented by using __add__ () method.
   This method is called magic method for + operator.

2 We have to override this method of our class.

3. For every operator magic methods are available .

+	:	__add(self,other)
-.	:	__sub__(self,other)
*.	:	__mul__(self,other)
/	:	__div__(self,other)
//	:	__floordiv__(self,other)
%.	:	__mod__(self,other)
**	:	__pow__(self,other)
+=	:	__iadd__(self,other)
-=	:	__isub__(self,other)
*=	:	__imul__(self,other)
/=	:	__idiv__(self,other)
//=	:	__ifloordiv__(self,other)
%=	:	__imod__(self,other)
**=	:	__ipow__(self,other)
<.	:	__lt__(self,other)
>.	:	__gt__(self,other)
<=	:	__le__(self,other)

>= : __ge__(self.other)

== : __eq__(self.other)

!= : __ne__(self.other)

Overloading the + Operator in Python:

To overload the + sign, we will need to implement __add__() function in the class. With great power comes great responsibility. We can do whatever we like, inside this function. But it is sensible to return a Point object of the coordinate sum.

Ex:

```
class Point:
        def __init__(self, x = 0, y = 0):
            self.x = x
                self.y = y
        def __str__(self):
                return "({0},{1})".format(self.x,self.y)
        def __add__(self,other):
                x = self.x + other.x        y = self.y + other.y
                return Point(x,y)

p1 = Point(2,3)
p2 = Point(-1,2)
print(p1 + p2)
```

ITERATORS

1. Iterators are everywhere in Python. They are elegantly implemented within for loops, comprehensions, generators etc. but hidden in plain sight.

2. Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.
3. An object is called iterable if we can get an iterator from it. Most of built-in containers in Python like: list, tuple, string etc. are iterables.

4. Technically speaking, Python iterator object must implement two special methods, __iter__() and __next__(), collectively called the iterator protocol.

5. The iter() function (which in turn calls the __iter__() method) returns an iterator from them.

6. if we want to get one by one element from iterator object by calling next()

7. if no more elements in our iterator object, still we can call next() function to raise StopIeration Exception.

Ex-1: to print list values normally using for loop

```
x=[11,22,33,44]
for i in x:
        print(i)
```

Ex-2: to print list values normally using iter() and next()

```
x=[11,22,33,44]
print(x)
print(type(x))
y=iter(x)
print(y)
print(type(y))
print(next(y))
print(next(y))
print(next(y))
print(next(y))
print(next(y))    #StopIteration.
```

Building Your Own Iterator in Python:

Building an iterator from scratch is easy in Python. We just have to implement the methods __iter__() and __next__().

The __iter__() method returns the iterator object itself. If required, some initialization can be performed.

The __next__() method must return the next item in the sequence. On reaching the end, and in subsequent calls, it must raise StopIteration.

Here, we show an example that will give us next power of 2 in each iteration. Power exponent starts from zero up to a user set number.

```
class powtwo:
"""class to implement an iterator of powers of two"""
        def __init__(self,max=0):
                self.max=max
        def __iter__(self):
                self.n=0
                return self



        def __next__(self):
                if self.n<=self.max:
                   result=s**self.n
                   self.n+=1
                   return result
                else:
                        raise StopIteration
```

Now we can create an iterator and iterate through it as follows.

```
        a = PowTwo(4)
```

```
i = iter(a)
next(i) 1
next(i) 2
next(i) 4
next(i) 8
next(i) 16
next(i)
                Traceback (most recent call last):
                ...
                StopIteration
```

We can also use a for loop to iterate over our iterator class.

```
for i in PowTwo(5): ...
    print(i)
                ...
                1
                2
                4
                8
                16
                32
```

Python Infinite Iterators:

It is not necessary that the item in an iterator object has to exhaust. There can be infinite iterators (which never ends). We must be careful when handling such iterator.

Here is a simple example to demonstrate infinite iterators.

The built-in function iter() can be called with two arguments where the first argument must be a callable object (function) and second is the sentinel. The iterator calls this function until the returned value is equal to the sentinel.

```
 int()
0

 inf = iter(int,1)  next(inf) 0
next(inf)
0
```

We can see that the int() function always returns 0. So passing it as iter(int,1) will return an iterator that calls int() until the returned value equals 1. This never happens and we get an infinite iterator.

We can also built our own infinite iterators. The following iterator will, theoretically, return all the odd numbers.

```
class infiter:
"""infinite iterator to return all    odd number"""
        def __iter__(self):
                self.num=1
                return self
        def __next__(self):       num=self.num
        self.num+=2
                                return num
```

A sample run would be as follows.

```
a = iter(InfIter())
next(a) 1
next(a) 3
next(a) 5
next(a)
                        7
                        And so on...
```

Be careful to include a terminating condition, when iterating over these type of infinite iterators.

The advantage of using iterators is that they save resources. Like shown above, we could get all the odd numbers without storing the entire number system in memory. We can have infinite items (theoretically) in finite memory.

StopIteration

The example above would continue forever if you had enough next() statements, or if it was used in a for loop.

To prevent the iteration to go on forever, we can use the StopIteration statement.

In the __next__() method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

Example: Stop after 20 iterations:

```
class MyNumbers:
        def __iter__(self):
                        self.a = 1     return self
        def __next__(self):
                if self.a <= 20:
                        x = self.a
                        self.a += 1       return x
                else:
                        raise StopIteration
myclass = MyNumbers()
myiter = iter(myclass)
for x in myiter:
        print(x)
```

GENERATORS

Python generators are a simple way of creating iterators. All the overhead we mentioned above are automatically handled by generators in Python. Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

How to create a generator in Python:

1. It is fairly simple to create a generator in Python.
2. It is as easy as defining a normal function with yield statement instead of a return statement.
3. If a function contains at least one yield statement (it may contain other yield or return statements), it becomes a generator function. Both yield and return will return some value from a function.

The difference is that, while a return statement terminates a function entirely, yield statement pauses the function saving all its states and later continues from there on successive calls.

Differences between Generator function and a Normal function:

1. Generator function contains one or more yield statement.
2. When called, it returns an object (iterator) but does not start execution immediately.
3. Methods like __iter__() and __next__() are implemented automatically. So we can iterate through the items using next().
4. Once the function yields, the function is paused and the control is transferred to the caller.
5. Local variables and their states are remembered between successive calls.
6. Finally, when the function terminates, StopIteration is raised automatically on further calls.

Eg:

```
def my_gen():
    n=1
    print('This is printed first')
    yield n
    n+=1
    print('This is printed second')
    yield n
    n+=1
    print('This is printed at last')
    yield n
```

An interactive run in the interpreter is given below.

Run these in the Python shell to see the output.

```
# It returns an object but does not start execution immediately.
    a = my_gen()

# We can iterate through the items using next().
    >>> next(a)
```

```
        This is printed first
        1
        >>> next(a)
        This is printed second
        2
        >>> next(a)
        This is printed at last
        3
```

Finally, when the function terminates, StopIteration is raised automatically on further calls.

```
        >>> next(a)
                        Traceback (most recent call last):
                        ...
                        StopIteration  next(a)
                        Traceback (most recent call last):
                        ...
                        StopIteration
                         or
```

using for loop:

```
def my_gen():
        n=1
        print('This is printed first')
        yield n
        n+=1
        print('This is printed second')
        yield n
        n+=1
        print('This is printed at last')
        yield n

for item in my_gen():
        print(item)
```

Python Generator Expression:

1. Simple generators can be easily created on the fly using generator expressions. It makes building generators easy.

2. Same as lambda function creates an anonymous function, generator expression creates an anonymous generator function.

3. The syntax for generator expression is similar to that of a list comprehension in Python. But the square brackets are replaced with round parentheses.

4. The major difference between a list comprehension and a generator expression is that while list comprehension produces the entire list, generator expression produces one item at a time.

5. They are kind of lazy, producing items only when asked for.For this reason, a generator expression is much more memory efficient than an equivalent list comprehension.

```
my_list=[1,3,6,10] a=(x**2 for x in
my_list) print(next(a)) print(next(a))
print(next(a))
print(next(a))
```

Generator expression can be used inside functions. When used in such a way, the round parentheses can be dropped.

```
>>sum(x**2 for x in my_list)
>>max(x**2 for x in my_list)
```

Why generators are used in Python

There are several reasons which make generators an attractive implementation to go for.

1. Easy to Implement:

Generators can be implemented in a clear and concise way as compared to their iterator class counterpart. Following is an example to implement a sequence of power of 2's using iterator class.

```
class powtwo:
        def __init__(self,max=):
                self.max=max
        def __iter__(self):
                self.n=0
                return self
        def __next__(self):
                if self.n>self.max:
                raise StopIteration        result=2**self.n
                self.n+=1
                return result
```

This was lengthy. Now lets do the same using a generator function.

```
        def PowTwoGen(max = 0):
    n = 0     while n < max:
yield 2 ** n
                n += 1
```

Since, generators keep track of details automatically,it was concise and much cleaner in implementation.

2. Memory Efficient:

A normal function to return a sequence will create the entire sequence in memory before returning the result. This is an overkill if the number of items in the sequence is very large.

Generator implementation of such sequence is memory friendly and is preferred since it only produces one item at a time.

3. Represent Infinite Stream:

Generators are excellent medium to represent an infinite stream of data. Infinite streams cannot be stored in memory and since generators produce only one item at a time, it can represent infinite stream of data.

The following example can generate all the even numbers

```
def all_even():
        n = 0
        while True:
                yield n
                n += 2
```

4. <u>Pipelining Generators</u>

Generators can be used to pipeline a series of operations. Suppose we have a log file from a famous fast food chain. The log file has a column (4th column) that keeps track of the number of pizza sold every hour and we want to sum it to find the total pizzas sold in 5 years. Assume everything is in string and numbers that are not available are marked as 'N/A'. A generator implementation of this could be as follows.

```
        with open('sells.log') as file:
           pizza_col = (line[3] for line in file)     per_hour = (int(x) for x in pizza_col if x !=
        'N/A')     print("Total pizzas sold = ",sum(per_hour))
```

This pipelining is efficient and easy to read (and yes, a lot cooler!).

DATA BASE COMMUNICATION

The Python standard for database interfaces is the Python DB-API. Most Python database interfaces adhere to this standard. You can choose the right database for your application. Python Database API supports a wide range of database servers:

- MySQL
- PostgreSQL
- Microsoft SQL Server 2000
- Informix
- Interbase
- Oracle
- Sybase

The DB API provides a minimal standard for working with databases using Python structures and syntax wherever possible.

- Importing the API module.
- Acquiring a connection with the database.

- Issuing SQL statements and stored procedures.
- Closing the connection

What is a Database?

A database is a separate application that stores a collection of data. Each database has one or more distinct APIs for creating, accessing, managing, searching and replicating the data it holds. Relational Database Management Systems to store and manage huge volume of data.

RDBMS Terminology

- Database
- Table
- Column
- Row
- Redundancy
- Primary Key
- Foreign Key
- Compound Key
- Index
- Referential Integrity

MySQL Database

I. MySQL is a fast, easy-to-use RDBMS being used for many small and big businesses.
II. MySQL is released under an open-source license. So you have nothing to pay to use it.
III. MySQL uses a standard form of the well-known SQL data language. IV. MySQL works on many operating systems and with many languages V. MySQL works very quickly and works well even with large data sets. VI. MySQL is customizable.

MySQL is a leading open source database management system. It is a multi user, multithreaded database management system. MySQL is especially popular on the web. It is one of the parts of the very popular LAMP platform. Linux, Apache, MySQL, PHP. Currently MySQL is owned by Oracle. MySQL database is available on most important OS platforms.

It runs under BSD Unix, Linux, Windows or Mac. Wikipedia and YouTube use MySQL. These sites manage millions of queries each day. MySQL comes in two versions. MySQL server system and MySQL embedded system.

Mysql commnands:

DDL Commands

DDL is short name of Data Definition Language, which deals with database schemas and descriptions, of how the data should reside in the database.

CREATE	:	To create a database and its objects like (table, index, views, store procedure, function, and triggers)
ALTER	:	Alters the structure of the existing database
DROP	:	Delete objects from the database
TRUNCATE	:	Remove all records from a table, including all spaces allocated for the records are removed
COMMENT	:	Add comments to the data dictionary
RENAME	:	Rename an object

DML Commands:

DML is short name of Data Manipulation Language which deals with data manipulation and includes most common SQL statements such SELECT, INSERT, UPDATE, DELETE, etc., and it is used to store, modify, retrieve, delete and update data in a database.

SELECT	:	Retrieve data from a database
INSERT	:	Insert data into a table
UPDATE	:	Updates existing data within a table
DELETE	:	Delete all records from a database

		table
MERGE	:	UPSERT operation (insert or update)
CALL	:	Call a PL/SQL or Java subprogram
EXPLAIN PLAN	:	Interpretation of the data access path
LOCK TABLE	:	Concurrency Control

DCL Commands:

DCL is short name of Data Control Language which includes commands such as GRANT and mostly concerned with rights, permissions and other controls of the database system.

GRANT	:	allow users access privileges to the database
REVOKE	:	withdraw users access privileges given by using the GRANT command

TCL Commands:

TCL is short name of Transaction Control Language which deals with a transaction within a database.

COMMIT	:	Commits a Transaction
ROLLBACK	:	Rollback a transaction in case of any error occurs
SAVEPOINT	:	To rollback the transaction making points within groups
SET TRANSACTION	:	Specify characteristics of the transaction

CREATE COMMAND:

CREATE DATABASE statement is used to create a database in MySQL.

Syntax: CREATE DATABASE database;

Eg: CREATE DATABASE TEST;
Eg: CREATE DATABASE SAMPLE;
CREATE DATABASE Students;

mysql> USE Students
CREATE TABLE statement is used to create a table in MySQL database. syntax:
EG: CREATE TABLE EMP(EMPNO INT, ENAME VARCHAR(20));
EG: CREATE TABLE STUDENT(HTNO INT. SNAME VARCHAR(20),
M1 INT, M2 INT,M3 INT);

mysql>show tables;
mysql>describe EMP;

Adding a Column to the Table:

At this point, the emp table does not have a unique identifier for every record, called a primary key . We can make changes to the table and add another column to it that will always take in only unique values. This can be done with multiple ALTER commands as follows:

mysql> ALTER TABLE emp ADD COLUMN deptno INT(10);
mysql> ALTER TABLE emp ADD PRIMARY KEY(empno)

Insert data to the table:

The INSERT statement is used to add values to a table. Below is the command to insert 3 employee records into the table emp:

mysql>
insert into emp values (1111, "ram");
insert into emp values (1111, "ram"); insert into emp values
(2222, "sita"); insert into emp values (3333, "laxman");

Viewing the Table:

Now that you are done inserting values, how do you check to see if the records are in the table now. This is done with the SELECT command as follows:

```
mysql> select * from emp
mysql> select empno from emp mysql>select
empno,ename from emp mysql>select ename,sal
from emp
```

Updating selected records:

To update information in a specific column for a specific record, the UPDATE command is used as follows:

```
mysql> UPDATE emp SET ename= 'venkatesh'  WHERE empno=2222;
```

This updates the ename of the record whose empno is 2222.

```
mysql>update student  set m1= m1+10  where result='fail'; mysql>update student  set
m2=m2+10 where result='fail';
```

Delete data from the Table:

if you want to delete a record or mulptiple records from a table , use DELETE command:

```
mysql>delete from emp where sal>75000;
mysql>delete from student  where result='fail';
mysql>delete from emp where deptname='mechanical';
```

Connecting Python with the database

A table resides within a database. This is particularly true for MySQL. To create a table, a database must be created first, or at least a database must be present. So to retrieve data from a table, a connection to the database must be established. This is done by using the connect() method. In other words, connect is the constructor of the phpMyAdmin. The parameters are as follows:

- **Host** : is the name of the system where the MySQL server is running. It can be a name or an IP address. If no value is passed, then the default value used is localhost.
- **User :** is the user id, which must be authenticated. In other words, this is the authentic id for using the services of the Server. The default value is the current effective user. Most of the time it is either 'nobody' or 'root'.
- **passwd** : It is by using a combination of the user id and a password that MySQL server (or for that matter any server) authenticates a user. The default value is no passwords. That means a null string for this parameter.
- **Db :** is the database that must be used once the connection has been established with the server. However, if the database to be used is not selected, the connection established is of no use. There is no default value for this parameter.

Creation of the cursor:

In the terminology of databases, cursor is that area in the memory where the data fetched from the data tables are kept once the query is executed. In essence it is the scratch area for the database.

MySQL does not support cursors. But it is easy to emulate the functionality of cursors. That's what the phpMyAdmin does. To get a cursor, the cursor() method of connection object has to be used. There is only one parameter to this method -- the class that implements cursor behavior. This parameter is optional. If no value is given, then it defaults to the standard Cursor class. If more control is required, then custom Cursor class can be provided. To obtain a cursor object the statement would be:

```
cursor= db.cursor()
```

Once the above statement is executed, the cursor variable would have a cursor object.

Execution of the SQL statement:

The steps enumerated until now have done the job of connecting the application with the database and providing an object that simulates the functionality of cursors. The stage has been set for execution of SQL statements. Any SQL statement supported by MySQL can be executed using the execute() method of the Cursor class. The SQL statement is passed as a string to it. Once the statement is executed successfully, the Cursor object will contain the result set of the retrieved values. For example, to retrieve all the rows of a table named STUDENT the statement would be:

```
cursor.execute("select * from STUDENT")
```

Once the above statement is executed, the cursor object would contain all the retrieved. This brings us to the fourth step, fetching of the resultset. Before moving on to the next step, there is one point you must understand. The execute() function accepts and executes any valid SQL statement, including DDL statements such as delete table, alter table, and so on

Fetching the result set:

The flexibility of Python comes to the fore in this step also. In the real world, fetching all the rows at once may not be feasible. MySQLdb answers this situation by providing different versions of the fetch() function of Cursor class.

The two most commonly used versions are:

- fetchone(): This fetches one row in the form of a Python tuple. All the data types are mapped to the Python data types except one -- unsigned integer. To avoid any overflow problem, it is mapped to the long. eg:

```
numrows = int(cursor.rowcount)
                #get the count of total rows in the  resultset
    # get and display one row at a time      for x in
range(0,numrows):            row = cursor.fetchone()       print(
row[0], "-->", row[1])
```

- fetchall(): This fetches all the rows as tuple of tuples. While fetchone() increments the cursor position by one, fetchall() does nothing of that kind. Everything else is similar. eg:

```
result = cursor.fetchall( )
for record in result:
    print( record[0] , "-->", record[1])
```

Q. Write a Program to create student table with htno and name ,insert data and display data by using mysql database.

```
import mysql.connector
try:
con=mysql.connector.connect(host='localhost',user='root',passw
ord='root',database='xydb')
cursor=con.cursor()
cursor.execute("create table student(htno int(5) primary key,sname varchar(20)))")
print("Table Created...")
cursor.execute("insert into student values(1111,'RAM')") cursor.execute("insert into
student values(2222,'LAXMAN')") cursor.execute("insert into student
values(3333,'SITA')") print("Records Inserted Successfully...") cursor.execute("select *
from employees") d=cursor.fetchall( )
print(d)
```

GUI Programming

Python GUI – tkinter

Python offers multiple options for developing GUI (Graphical User Interface). Out of all the GUI methods, tkinter is most commonly used method. It is a standard Python interface to the Tk GUI toolkit shipped with Python. Python with tkinter outputs the fastest and easiest way to create the GUI applications. Creating a GUI using tkinter is an easy task.

To create a tkinter:

1. Importing the module – tkinter
2. Create the main window (container)
3. Add any number of widgets to the main window
4. Apply the event Trigger on the widgets.

Importing tkinter is same as importing any other module in the python code.

```
import tkinter
```

There are two main methods used you the user need to remember while creating the Python application with GUI.

1. **Tk(screenName=None, baseName=None, className='Tk', useTk=1):** To create a main window, tkinter offers a method 'Tk(screenName=None, baseName=None, className='Tk', useTk=1)'. To change the name of the window, you can change the className to the desired one. The basic code used to create the main window of the application is: m=tkinter.Tk( )
where m is the name of the main window object

2. **mainloop( ):** There is a method known by the name mainloop() is used when you are ready for the application to run. mainloop( ) is an infinite loop used to run the application, wait for an event to occur and process the event till the window is not closed. m.mainloop( )

```
import tkinter
 m = tkinter.Tk()
'''
        widgets are added here
'''
m.mainloop()
```

tkinter also offers access to the geometric configuration of the widgets which can organize the widgets in the parent windows. There are mainly three geometry manager classes class.

1. **pack( ) method** :It organizes the widgets in blocks before placing in the parent widget.
2. **grid( ) method** :It organizes the widgets in grid (table-like structure) before placing in the parent widget.
3**. place( ) method** :It organizes the widgets by placing them on specific positions directed by the programmer.

There are a number of widgets which you can put in your tkinter application.

Tkinter Widgets:

There are currently 16 types of widgets in Tkinter.

1. **Button**
2. **Canvas**
3. **Checkbutton**
4. **Entry**
5. **Frame**
6. **Label**
7. **Listbox**
8. **Menubutton**
9. **Menu**
10. **Message**
11. **Radiobutton**
12. **Scale**
13. **Scrollbar**
14. **Text**
15. **Spinbox**
16. **tkMessageBox**
17. **pannedWindow**

1. Button:To add a button in your application, this widget is used.
The general syntax is:

w=Button(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the Buttons. Number of options can be passed as parameters separated by commas. Some of them are listed below.

Activeforeground : to set the foreground color when button is under the cursor.

Bg : to set he normal background color.

Command **:** to call a function.

Font : to set the font on the button label.

Image : to set the image on the button.

Width : to set the width of the button.

Height : to set the height of the button.

Ex:

```
import tkinter as tk
window= tk.Tk()
window.title('Counting Seconds')
b1=tk.Button(window,text='Stop',width=25
,command=window.destroy)
b1.pack( )
window.mainloop( )
```

2. Canvas: It is used to draw pictures and other complex layout like graphics, text and widgets.

The general syntax is:

```
w = Canvas(master, option=value)
```

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

Bd : to set the border width in pixels.

Bg : to set the normal background color.

Cursor : to set the cursor used in the canvas.

Highlightcolor : to set the color shown in the focus highlight.

Width : to set the width of the widget.

Height : to set the height of the widget.

Ex:

```
from tkinter import *
window = Tk()
c1 = Canvas(window, width=40, height=60)  c1.pack()
canvas_height=20
canvas_width=200
y = int(canvas_height / 2)  c1.create_line(0, y, canvas_width,
y )  window.mainloop( )
```

3. CheckButton: To select any number of options by displaying a number of options to a user as toggle buttons. The general syntax is:

w = CheckButton(master, option=value)

There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

Title	**:**	To set the title of the widget.
Activebackground	**:**	To set the background color when widget is under the cursor.
Activeforeground	**:**	To set the foreground color when widget is under the cursor.
Bg	**:**	To set he normal backgrouSteganography
Command	**:**	T call a function.
Font	**:**	To set the font on the button label.
Image	**:**	To set the image on the widget.

Ex:

```
from tkinter import *
window = Tk( )
var1 = IntVar( )
  Checkbutton(window, text='male', \
variable=var1).grid(row=0, sticky=W)
var2 = IntVar( )
   Checkbutton(window, text='female', \
variable=var2).grid(row=1, sticky=W)  window.mainloop( )
```

4. Entry:It is used to input the single line text entry from the user.. For multi-line text input, Text widget is used. The general syntax is:

w=Entry(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

Bd : to set the border width in pixels.

Bg **:** to set the normal background color.

Cursor : to set the cursor used.

Command : to call a function.

Highlightcolor : to set the color shown in the focus highlight.

Width : to set the width of the button.

Height : to set the height of the button.

Ex:

```
from tkinter import *
window = Tk()
Label(window, text='First Name').grid(row=0)
Label(window, text='Lastname').grid(row=1)
e1 = Entry(window)
e2 = Entry(window)
e1.grid(row=0, column=1)
e2.grid(row=1, column=1)
window.mainloop( )
```

5. Frame:

It acts as a container to hold the widgets. It is used for grouping and organizing the widgets. The general syntax is:

```
w = Frame(master, option=value)
```

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

Highlightcolor **:** To set the color of the focus highlight when widget has to be focused.

Bd : to set the border width in pixels.

Bg : to set the normal background color.

Cursor : to set the cursor used.

Width : to set the width of the widget.

Height : to set the height of the widget.

Ex:

```
from tkinter import * window =
Tk()
f1 = Frame(window)  f1.pack()
bottomframe = Frame(window)
bottomframe.pack( side = BOTTOM )
redbutton = Button(f1, text = 'Red', fg ='red')  redbutton.pack( side = LEFT)
greenbutton = Button(f1, text = 'Brown', fg='brown')  greenbutton.pack( side
= LEFT )
bluebutton = Button(f1, text ='Blue', fg ='blue')  bluebutton.pack( side = LEFT
)
blackbutton = Button(bottomframe, text ='Black', fg
='black')
blackbutton.pack( side = BOTTOM)
window.mainloop()
```

6. Label:

It refers to the display box where you can put any text or image which can be updated any time as per the code. The general syntax is:

```
w=Label(master, option=value)
```

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

Bg : to set he normal background color.

Command : to call a function.

Font : to set the font on the button label.

Image : to set the image on the button.

Width : to set the width of the button.

Height : to set the height of the button.

Ex:

```
from tkinter import *
window = Tk()
l1 = Label(window, text='KKRCoders')  l1.pack()
window.mainloop()
```

7. Listbox:

It offers a list to the user from which the user can accept any number of options.

The general syntax is:

```
w = Listbox(master, option=value)
```

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

Highlightcolor : To set the color of the focus highlight when widget has to be focused.

Bg : to set he normal background color.

Bd : to set the border width in pixels.

Font : to set the font on the button label.

Image : to set the image on the widget.

Width : to set the width of the widget.

Height : to set the height of the widget.

Ex:

```
from tkinter import *
window = Tk()
```

```
Lb = Listbox(window)
Lb.insert(1, 'Python')
Lb.insert(2, 'Java')
Lb.insert(3, 'C++')
Lb.insert(4, 'Any other')
Lb.pack()
window.mainloop()
```

8. MenuButton:

It is a part of top-down menu which stays on the window all the time. Every menubutton has its own functionality. The general syntax is:

```
w = MenuButton(master, option=value)
```

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

Activeforeground	:	To set the foreground when mouse is over the widget.
Bg	:	to set he normal background color.
Bd	:	to set the size of border around the indicator.
Cursor	:	To appear the cursor when the mouse over the menubutton.
Image	:	to set the image on the widget.
Width	:	to set the width of the widget.
Height	:	to set the height of the widget.
Highlightcolor	:	To set the color of the focus highlight when widget has to be focused.

Ex:

```
from tkinter import *
window = Tk( )
mb =  Menubutton ( window, text = &quot;GfG&quot;)  mb.grid()
mb.menu  =  Menu ( mb, tearoff = 0)
mb[&quot;menu&quot;]  =  mb.menu  cVar  = IntVar()
```

```
aVar = IntVar()
mb.menu.add_checkbutton ( label ='Contact', variable = cVar )
mb.menu.add_checkbutton ( label = 'About', variable = aVar )  mb.pack()
window.mainloop()
```

9. Menu:

It is used to create all kinds of menus used by the application. The general syntax is:

```
w = Menu(master, option=value)
```

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

Title : To set the title of the widget.

Activebackground : To set the background color when widget is under the cursor.

Activeforeground : To set the foreground color when widget is under the cursor.

Bg : To set he normal background color.

Command : To call a function.

Font : To set the font on the button label.

Image : To set the image on the widget.

Ex:

```
from tkinter import * window = Tk()
menu = Menu(window)
window.config(menu=menu)
filemenu = Menu(menu)
menu.add_cascade(label='File', menu=filemenu)
filemenu.add_command(label='New')  filemenu.add_command(label='Open...')
filemenu.add_separator()
filemenu.add_command(label='Exit', command=window.quit)  helpmenu =
Menu(menu)
menu.add_cascade(label='Help', menu=helpmenu)
helpmenu.add_command(label='About')  window.mainloop()
```

10.Message:
It refers to the multi-line and non-editable text. It works same as that of Label.

The general syntax is:

w = Message(master, option=value)

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

Bd	:	to set the border around the indicator.
Bg	:	to set he normal background color.
Font	:	to set the font on the button label.
Image	:	to set the image on the widget.
Width	:	to set the width of the widget.
Height	:	to set the height of the widget.

Ex:

```
from tkinter import *
window = Tk()
ourMessage ='This is our Message'
messageVar = Message(window, text = ourMessage)
messageVar.config(bg='lightgreen')  messageVar.pack( )
window.mainloop( )
```

11. RadioButton:

It is used to offer multi-choice option to the user. It offers several options to the user and the user has to choose one option. The general syntax is:

w = RadioButton(master, option=value)

There are number of options which are used to change the format of this widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

Activebackground : to set the background color when widget is under the cursor.

Activeforeground : to set the foreground color when widget is under the cursor.

Bg : to set he normal background color.

Command : to call a function.

Font : to set the font on the button label.

Image : to set the image on the widget.

Width : to set the width of the label in characters.

Height : to set the height of the label in characters.

Ex:

```
from tkinter import *
window = Tk( )
v = IntVar()
Radiobutton(window, text='python', variable=v,\ value=1).pack(anchor=W)
Radiobutton(window,  text='java',  variable=v, \ value=2).pack(anchor=W)
window.mainloop ( )
```

12. Scale:

It is used to provide a graphical slider that allows to select any value from that scale.

The general syntax is:

```
w = Scale(master, option=value)
```

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

Cursor : To change the cursor pattern when the mouse is over the widget.

Activebackground : To set the background of the widget when mouse is over the widget.

Bg : to set he normal background color.

Orient : Set it to HORIZONTAL or VERTICAL according to the requirement.

From : To set the value of one end of the scale range.

To : To set the value of the other end of the scale range.

Image : To set the image on the widget.

Width : To set the width of the widget.

Ex:

```
from tkinter import *
window = Tk()
s1 = Scale(window, from_=0, to=50)
s1.pack()
s1 = Scale(window, from_=0, to=200,
orient=HORIZONTAL)
s1.pack()
window.mainloop()
```

13. Scrollbar:

It refers to the slide controller which will be used to implement listed widgets.

The general syntax is:

```
w = Scrollbar(master, option=value)
```

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

Width : to set the width of the widget.

Activebackground : To set the background when mouse is over the widget.

Bg : to set he normal background color.

Bd : to set the size of border around the indicator.

Cursor : To appear the cursor when the mouse over the menubutton.

Ex:

```
from tkinter import *
window = Tk()
scrollbar = Scrollbar(window)  scrollbar.pack( side = RIGHT, fill = Y )
mylist = Listbox(window, yscrollcommand = scrollbar.set )  for line in range(100):
  mylist.insert(END, 'This is line number' + str(line))  mylist.pack( side = LEFT, fill =
BOTH )  scrollbar.config( command = mylist.yview )  window.mainloop()
```

14. Text:

To edit a multi-line text and format the way it has to be displayed.

The general syntax is:

```
w    =Text(master, option=value)
```

There are number of options which are used to change the format of the text. Number of options can be passed as parameters separated by commas. Some of them are listed below.

Highlightcolor : To set the color of the focus highlight when widget has to be focused.

Insertbackground : To set the background of the widget.

Bg : to set he normal background color.

Font : to set the font on the button label.

Image : to set the image on the widget.

Width : to set the width of the widget.

Height : to set the height of the widget.

Eg:

```
from tkinter import *
window = Tk()
t1 = Text(window, height=2, width=30)
t1.pack()
t1.insert(END, 'kosmiktechnologies\nHyderabad\n')  window.mainloop( )
```

15. TopLevel:

This widget is directly controlled by the window manager. It don't need any parent window to work on.The general syntax is:

```
w = TopLevel(master, option=value)
```

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

Bg : to set he normal background color.

Bd **:** to set the size of border around the indicator.

Cursor : To appear the cursor when the mouse over the menubutton.

Width : to set the width of the widget.

Height : to set the height of the widget.

Ex:

```
from tkinter import *
window = Tk()
window.title('digital nest')
top = Toplevel()  top.title('Python')  top.mainloop()
```

16. SpinBox:

It is an entry of 'Entry' widget. Here, value can be input by selecting a fixed value of numbers.The general syntax is:

```
w = SpinBox(master, option=value)
```

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

Bg	:	to set he normal background color.
Bd	:	to set the size of border around the indicator.
Cursor	:	To appear the cursor when the mouse over the menubutton.
Command	:	To call a function.
Width	:	to set the width of the widget.
Activebackground	:	To set the background when mouse is over the widget.
Disabledbackground	:	To disable the background when mouse is over the widget.
from_	:	To set the value of one end of the range.
To	:	To set the value of the other end of the range.

Ex:

```
from tkinter import *
window = Tk()
sb = Spinbox(window, from_ = 0, to = 10)  sb.pack( )
window.mainloop()
```

17. PannedWindow

It is a container widget which is used to handle number of panes arranged in it. The general syntax is:

```
w   = PannedWindow(master, option=value)
```

master is the parameter used to represent the parent window.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

Bg : to set he normal background color.

Bd : to set the size of border around the indicator.

Cursor : To appear the cursor when the mouse over the menubutton.

Width **:** to set the width of the widget.

Height : to set the height of the widget.

Ex:

```
from tkinter import *
m1 = PanedWindow()  m1.pack(fill =
BOTH,expand = 1)
left = Entry(m1, bd = 5)  m1.add(left)
m2 = PanedWindow(m1, orient = VERTICAL)  m1.add(m2)
top = Scale( m2, orient = HORIZONTAL)  m2.add(top)
mainloop()
```

ARRAY module in python

1. An array is a special variable, which can hold more than one value at a time.
2. All the values must be same data type. i.e. Python arrays are homogenous data structure.
3. Arrays are available in Python by importing the 'array' module.
4. This module is useful when we have to manipulate the elements
5. arrays are not fundamental type, but lists are internal to Python.
6. An array accepts values of one kind while lists are independent of the data type.

Array element –

Every value in an array represents an element.

Array index –

a). Every element has some position in the array known as the index.

b). The array is made up of multiple parts. And each section of the array is an element.

c). We can access all the values by specifying the corresponding integer index.

d). python supports positive and negative indexing.
positive index starts from left to right i.e. 0,1,2,3,....size-1
negative index starts from right to left i.e. -1,-2,-3,.....-n

e). The first element starts at index 0 and so on. At 9th index, the 10th item would appear.

Declare Array in Python:

You have first to import the array module in your Python script.
After that, declare the array variable as per the below syntax.

Syntax:

```
  array_var = array(TypeCode, [Initializers]
```

here,
array_var.....> is the name of the array variable.
the array() function which takes two parameters.
TypeCode .....>is the type of array whereas
Initializers ....>are the values to set in the array.

The argument "TypeCode" can be any value from the below chart:

TYPE CODE	C TYPE	PYTHON TYPE	MINIMUM SIZE IN BYTES
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

Ex-1: To create an array of 10 integers.

```
import array as a
a = a.array('i', range(10))
print("Type of array_var is:", type(a))

# Print the values generated by range() function
print("The array will include: ", list(range(10)))
```

Array Operations
Indexing an array
We can use indices to retrieve elements of an array.

```
import array as ar

# Create an array of 10 integers using range()
a = ar.array('i', range(10))

# Print array values using indices
print("1st array element is {} at index 0.".format(a[0]))
print("2nd array element is {} at index 1.".format(a[1]))
print("Last array element is {} at index 9.".format(a[9]))
print("Second array element from the tail end is {}."   \
      .format(array_var[-2]))
```

Slicing arrays:

The slice operator ":" is commonly used to slice strings and lists.
However, it does work for the arrays also.

```
from array import *

# Create an array from a list of integers
intger_list = [10, 14, 8, 34, 23, 67, 47, 22]
intger_array = array('i', intger_list)

# Slice the given array in different situations
print("Slice array from 2nd to 6th index: {}".format(intger_array[2:6]))
print("Slice last 3 elements of array: {}".format(intger_array[:-3]))
print("Slice first 3 elements from array: {}".format(intger_array[3:]))
print("Slice a copy of entire array: {}".format(intger_array[:]))
```

Python Operator

```
from array import *

# Declare two arrays using Python range()
# One contains -ve integers and 2nd +ve values.
num_array1 = array('i', range(-5, 0))
num_array2 = array('i', range(0, 5))
```

```
# Printing arrays before joining
print("num_array1 before joining: {}".format(num_array1))
print("num_array2 before joining: {}".format(num_array2))

# Now, concatenate the two arrays
num_array = num_array1 + num_array2

print("num_array after joining num_array1 and num_array2: {}".format(num_array))
```

Built-in functions or Built-in methods in array module:

1. array():

This function is used to create an array with data type and value listspecified in its arguments.

syntax:

array(data type, value list)

Ex:

```
import array
a = array.array('i', [1, 2, 3])
# printing original array
print ("The new created array is : ",end=" ")
for i in range (0, 3):
    print (a[i], end=" ")
```

2. append() :-

This function is used to add the value mentioned in its arguments at the end of the array.

Ex:

```
a.append(4)
a.append(5)
```

3. insert():

This function is used to add the value at the position specified in its argument.

syntax:

```
insert(i,x)
```

Ex:

```
a.insert(1,10)
```

```
a.insert(3,20)
```

4. pop() :-

This function removes the element at the position mentioned in its argument, and returns it.

5. remove() :-

This function is used to remove the first occurrence of the value mentioned in its arguments.

```
import array
a= array.array('i',[1, 2, 3, 1, 5])

# printing original array
print ("The new created array is : ",end="")
for i in range (0,5):
    print (a[i],end=" ")

print ("\r")

# using pop() to remove element at 2nd position
print ("The popped element is : ",end="")
print (a.pop(2));

# printing array after popping
print ("The array after popping is : ",end="")
for i in range (0,4):
    print (a[i],end=" ")

print("\r")

# using remove() to remove 1st occurrence of 1
a.remove(1)

# printing array after removing
print ("The array after removing is : ",end="")
for i in range (0,3):
    print (a[i],end=" ")
```

6. index():

This function returns the index of the first occurrence of value mentioned in arguments.

7. reverse():

This function reverses the array.

Ex:

```
from array import *
a=array('i',range(11,19))
r=reversed(a)
print('original array is :',end=' ' )
for i in a:
    print(i,end=' ')
print()
print('reversed array is :',end=' ')
for i in r:
    print(i,end=' ')
```

8. typecode :

This function returns the data type by which array is initialised.

Ex:

```
import array
a= array.array('i',[1, 2, 3, 1, 2, 5])
print ("The datatype of array is : ", a.typecode)
```

9. itemsize:

This function returns size in bytes of a single array element.

Ex:

```
import array
a= array.array('i',[1, 2, 3, 1, 2, 5])
print ("The itemsize of array is : ",a.itemsize)
```

10. buffer_info():

Returns a tuple representing the address in which array is stored and number of elements in it.

Ex:

```
import array
a= array.array('i',[1, 2, 3, 1, 2, 5])
print ("The buffer info. of array is : ",a.buffer_info())
```

11. count():

This function counts the number of occurrences of argument mentioned in array.

ex:

```
import array
a = array.array('i',[1, 2, 3, 1, 2, 5])
print(a.count(1))
print(a.count(2))
```

```
print(a.count(99))
```

12. extend()

This function appends a whole array mentioned in its arguments to the specified array.

syntax: extend(array)

Ex:

```
import array
a1 = array.array('i',[1, 2, 3, 1, 2, 5])
a2 = array.array('i',[1, 2, 3])
a1.extend(a2)
print ("The modified array is : ",end="")
for i in range (0,9):
    print (a1[i],end=" ")
```

13. fromlist(list) :-

This function is used to append a list mentioned in its argument to end of array.

syntax:

```
fromlist(list)
```

Ex:

```
import array
a = array.array('i',[1, 2, 3, 1, 2, 5])
x = [1, 2, 3]
a.fromlist(x)
print ("The modified array is : ",end="")
for i in range (0,9):
    print (a[i],end=" ")
```

14. tolist():

This function is used to convert an array into list

Ex:

```
import array
a = array.array('i',[1, 2, 3, 1, 2, 5])
x =a.tolist()
print ("The new list created is : ", x)
#how to fetch 3 digit value in list using list comprehension:
n=[5,567,12,4789,874,344,24,2]

li=[x for x in n if len(str(x))==3]
print('Three digit values in a given list=',end=' ')
print(li)
```

RANDOM MODULE

You can generate random numbers in Python by using random module.

1. **random( ):** This function is used to generates a random float numbers between 0.0 to 1.0. This function does not need any arguments.

   Ex:

   ```
   import random print(random.random())
   ```

2. **randint( ):** It returns a random integer between the specified integers.

   Ex:

   ```
   import random
   print(random.randint(1,10))
   ```

3. **randrange( )**: This function returns a radomly selected element from the range created by the start,stop,step arguments.

   Start: starting number. by default '0'.

   Stop: ending number.

   Step: step value. by default increased by '1'.

   Ex-1:

   ```
   import random
   print(random.randrange(1,10))
   ```

   Ex-2:

   ```
   import random
   print(random.randrange(10,101,10))
   ```

   Ex-3:

   ```
   import random
   print(random.randrange(0,100,20))
   ```

4. **shuffle( ):** This functions randomly reorders the elements in a list.

Ex-1:

```
import random x=[1,2,3,4,5] random.shuffle(x)
print(x)
```

Ex-2:

```
    import random
b=['python', 'is', 'dynamic', 'language']      random.shuffle(b)
print(b)
```

5. **choice( ):** This function returns a randomly selected element from a non-empty sequence . An empty sequence is an argument then it returns IndexError.

Ex-1:

```
import random
print(random.choice('python'))
```

Ex-2:

```
    import random
print(random.choice([11,22,33,44]))
```

#write a program to generate 4 digit OTP.

```
import random
print(random.randint(0,9),random.randint(0,9),    \
random.randint(0,9),random.randint(0,9))
```

#write a program to generate 6 digit OTP.

```
from random import *
print(randint(0,9),randint(0,9), randint(0,9), \
randint(0,9), randint(0,9),randint(0,9))
```

#write a program to generate 6 digit OTP using for loop.

```
import random
for i in ange(1,7):
    print(random.randint(0,9),end=' ' )
```

PLATFORM MODULE

This module is used to access underlaying platform data such as

a. Hardware

b. Os
c. Interpreter version information i.e.where python program is running

a) <u>Hardware Related functions</u>:

1. machine() : This function returns machine type.
2. node() : Tthis function returns network name
3. processor() : This function returns processors name.
4. release() : This function returns systems version release.
5. system() : It returns os name
6. version() : It returns systems versio release
7. system-alias : It returns all the three values in order. (system,release,version)
8. 8. uname() : It returns sys,node,release,version,machine and processor details

Ex:

```
import platform as p
print('System Details:')
print('machine type = ',p.machine()) print('network name =
',p.node()) print('processors name =',p.processor()) print('system
version release = p.release())
print('operating system ame=',p.system())
print('system version release =,p.version())          print('all
details of hardware = ',p.uname())
```

b) <u>OS Related Functions</u>:

1. win32_edition() : This function returns windows edition type.

2. win32_is_iot() : This function returns True/False. it returns True then the system supported IoT. otherwise False.

3. win32_ver() : It returns os version(return type tuple).

Ex:

```
import platform as p
print('In platform module os details')
print('windows edition type = ',p.win32_edition()) print('This suprroted IoT or
not =',p.win32_is_iot())
print('os version details = ',p.win32_ver())
```

c) Interpreter Related Functions:

1. python_branch() : it returns branch of the python(build no) with tags
2. python_build() : it returns build no and date
3. python_compiler() : it returns the type of compiler
4. python_implementation() : it returnsthe type of python implementation.

The python implementations are –

cpython,ironpython,jpython,jython,pypy.

5. python_version() : it returns python version
6. python_version_tuple() : it returns major,minor,path level

Ex:

```
import platform as p
print('Interpreter Details :') print(p.python_branch())
print(p.python_build()) print(p.python_compiler())
print(p.python_implementation()) print(p.python_version())
print(p.python_version_tuple())
```

OS MODULE

1. If there is a large number of file handles to handle in your python program , we can arrange our code within different directories to make things more manageble and reachable.
2. A directory or folder is a collectionof or set of files and subdirectories.
3. python has os module which provides us with the many useful functions to work with directories directory management in python means creating a directory, remaining it, listing all directories and working with them functions.

1. getcwd(): This function returns the current working directory.

Ex:

```
import os
x=os.getcwd()
print(x)
```

2. mkdir( ): This function is used to create a new sub directory.

Ex-1:

```
import os
os.mkdir('kusu')
print('new sub directory is created in presentdirectory')
```

Ex-2:

```
import os
os.mkdir('c:/users/admin/desktop/kusu')
print('new sub directory is created in desktop')
```

note: if we create the same name of the directory when the directory already exist we get 'FileExistsError'.

3.makedirs( ): This funciton is used to create a directory and multiple subdirectories at the same time cwd.

Ex:

```
import os
os.makedirs('c:/users/admin/desktop/kusu/python/admin) print('main and sub directories are created')
```

4.removedirs( ): This function is used remove the main directory and its sub directories

Ex:

```
import os
os.removedirs('c:/users/admin/desktop/kusu/python/admin) print('main and sub directories are deleted')
```

5. rmdir( ): This function is used remove the subdirectory.

Ex:

```
import os os.rmdir('kusu')
print('within current directory kusu sub directory deleted')
```

6. rename( ): This function is used to rename a existing file

```
# syntax: rename('oldfilename','newfilename')
```

Ex:

```
import os
os.rename('hello.txt','surya.txt')
print('old name is changed')
```

7. system( ): This method execute the command (a string) in a subshell. This method is implemented by calling the Standard C function system(), and has the same limitations. If command generates any output, it is sent to the interpreter standard output stream

Ex-1: To run python file

```
import os print('hai')
x='py batch9am.py'
os.system(x)
print('finish')
```

Ex-2: To open a notepad

```
  import os
cmd = 'notepad'
os.system(cmd)
```

Ex-3: write a program to change present system date

```
import os
cmd = 'date'
os.system(cmd)
```

DATE & TIME MODULE

In Python, date, time and datetime classes provides a number of functions to deal with dates, times and time intervals. Date and datetime are an objects in Python, so when you manipulate them, you are actually manipulating objects and not string or timestamps.

The datetime classes in Python are categorized into main 5 classes.

date	:	Manipulate just date (Month, day, year)
time	:	Time independent of the day (Hour, minute, second, microsecond)

datetime	:	Combination of time and date (Month, day, year, hour, second, microsecond)
timedelta	:	A duration of time used for manipulating dates
tzinfo	:	An abstract class for dealing with time zones

What is Tick?

The floating-point numbers in units of seconds for time interval are indicated by Tick in python. Particular instants in time are expressed in seconds since 12:00am, January 1, 1970(epoch). The function time.time() returns the current system time in ticks since 12:00am, January 1, 1970(epoch).

Ex:

```
import time ticks = time.time()
print("Number of ticks since 12:00am, January 1, 1970:", ticks)
```

What is TimeTuple?

Python's time functions handle time as a tuple of 9 numbers

Index	Field	Values
0	4-digit year	2021
1	Month	1 to 12
2	Day	1 to 31
3	Hour	0 to 23
4	Minute	0 to 59
5	Second	0 to 60
6	Day of Week	0 to 6 (0 is Monday)
7	Day of year	1 to 366 (Julian day)
8	Daylight Savings	-1, 0, 1

United States begins Daylight Saving Time at 2:00 a.m. on the second Sunday in March and reverts to standard time on the first Sunday in November. Daylight Saving Time- Benjamin Franklin.

In the European Union, Summer Time begins and ends at 1:00 a.m. Universal Time (Greenwich Mean Time). It begins the last Sunday in March and ends the last Sunday in October. **Struct_Time Structure**

Index	Attributes	Values
0	tm_year	2018
1	tm_mon	1 to 12
2	tm_mday	1 to 31
3	tm_hour	0 to 23
4	tm_min	0 to 59
5	tm_sec	0 to 61 (60 or 61 are leap-seconds)
6	tm_wday	0 to 6 (0 is Monday)
7	tm_yday	1 to 366 (Julian day)
8	tm_isdst	-1, 0, 1, -1 means library determines DST

Getting current time:

To translate a time instant from a seconds since the epoch floating-point value into a timetuple, pass the floating-point value to a function (e.g., localtime) that returns a time-tuple with all nine items valid.

Ex:

```
#!/usr/bin/python import time;
localtime = time.localtime(time.time())
print("Local current time :", localtime)
```

How to Use Date & DateTime Class:

Before you run the code for datetime, it is important that you import the date time modules **Example:**

```
from datetime import date
td=date.today()
print("Today is: ",td)
day=td.day
print("Day is: ",day)
```

Example:

```
from datetime import date td=date.today()
print(td.month)
print(td.year) print(td.weekday())
print(date.weekday(td))
```

How to Format Time Output:

1. 1 We used the "strf time function" for formatting.
2. 2 This function uses different control code to give an output.
3. Each control code resembles different parameters like year,month, weekday and date [(%y/%Y – Year), (%a/%A- weekday), (%b/%B- month), (%d - day of month)] .

Example:

```
from datetime import datetime x=datetime.now()
print(x.strftime("%y")) print(x.strftime("%Y"))
print(x.strftime("%a")) print(x.strftime("%A"))
print(x.strftime("%b")) print(x.strftime("%B"))
print(x.strftime("%A %d %B,%y"))
print(x.strftime("%A %D %B,%Y"))
```

With the help of "Strftime" function we can also retrieve local system time, date or both.

```
%c- indicates the local date and time
%x- indicates the local date
%X- indicates the local time
```

Example:

```
from datetime import datetime x=datetime.now()
print(x.strftime("%c")) print(x.strftime("%x"))
print(x.strftime("%X"))
```

The "strftime function" allows you to call the time in any format 24 hours or 12 hours.

Example:

```
from datetime import datetime x=datetime.now()
print(x.strftime("%I:%M:%S %p"))
print(x.strftime("%H:%M %p"))
```

How to use Timedelta Objects:

With timedelta objects, you can estimate the time for both future and the past. In other words, it is a timespan to predict any special day, date or time. **Example:**

```
from datetime import datetime
from datetime import timedelta
print(timedelta(days=365,hours=8, minutes=15)) print("ToDay is: ",
datetime.now())
```

The time Module:

There is a popular time module available in Python which provides functions for working with times and for converting between representations. Here is the list of all available methods:

Getting formatted time:

You can format any time as per your requirement, but simple method to get time in readable format is asctime().

Example:

```
import time
lt=time.asctime(time.localtime(time.time())) print(lt)
dst=time.asctime(time.localtime(time.daylight)) print(dst)
```

Python time clock() Method:

It returns the current processor time as a floating point number expressed in seconds on Unix.

Syntax:

```
time.clock( )
```

Ex:

```
print(time.clock( ))
```

CALENDAR MODULE

It allows you to output calendars like the Unix cal program and provides additional useful functions related to the calendar. By default, these calendars have Monday as the first day of the week, and Sunday as the last.

iterweekdays( ) :

The iterweekdays() method returns an iterator for the weekday numbers that will be used for one week. The first number from the iterator will be the same as the number returned by firstweekday().

Syntax:

```
iterweekdays()
```

Ex:

```
import calendar
cal= calendar.Calendar(firstweekday=0)
for x in cal.iterweekdays():
        print(x)
```

itermonthdates( ) :

The itermonthdates() method returns an iterator for the month (1-12) in the year. This iterator will return all days (as datetime.date objects) for the month and all days before the start of the month or after the end of the month that are required to get a complete week.

Syntax

```
itermonthdates(year, month)
```

Ex:

```
import calendar
cal= calendar.Calendar()
for x in cal.itermonthdates(2020,3):
   print(x))
```

itermonthdays2() :

The itermonthdays2() method is used to get an iterator for the month in the year similar to itermonthdates( ). Days returned will be tuples consisting of a day number and a week day number.

Syntax:

itermonthdays2(year, month)

Ex:

```
import calendar
cal= calendar.Calendar()
for x in cal.itermonthdays2(2020, 3):
    print(x)
```

itermonthdays( ) :

The itermonthdays( ) method returns an iterator of a specified month and a year. Days returned will simply be day numbers. The method is similar to itermonthdates( ).

Syntax:

itermonthdays(year, month)

Ex:

```
import calendar
cal= calendar.Calendar()
for x in cal.itermonthdays(2020, 3):
    print(x)
```

monthdatescalendar( ) :

The monthdatescalendar() method is used to get a list of the weeks in the month of the year as full weeks. Weeks are lists of seven datetime.date objects.

Syntax

monthdatescalendar(year, month)

Ex:

```
import calendar
cal= calendar.Calendar()
print(cal.monthdatescalendar(2020,3))
```

month( ):

The calendar module gives a wide range of methods to play with yearly and monthly calendars. Here, we print a calendar for a given month (March 2020)

Syntax:

calendar.month(theyear, w=2, l=1, c=6, m=3)

year : Year for which the calendar should be generated.

W : The width between two columns. Default value is 2.

L : Blank line between two rows. Default value is 1.

c : Space between two months (Column wise). Default value is 6. **M** : Number of months in a row. Default value is 3.

Ex:

```
import calendar
cal = calendar.month(2020,3) print( cal)
```

formatmonth( ) :

The formatmonth() method is used to get a month's calendar in a multiline string.

Syntax:

formatmonth(theyear, themonth, w=0, l=0)

Ex:

```
import calendar
tc= calendar.TextCalendar(firstweekday=0)   print(tc.formatmonth(2020, 3))
```

formatyear( ) :

The formatyear() method is used to get a m-column calendar for an entire year as a multi-line string.

Syntax:

formatyear(theyear, w=2, l=1, c=6, m=3)

Ex:

```
import calendar
```

tc= calendar.TextCalendar(firstweekday=0) print(tc.formatyear(2020, 3))

Python HTMLCalendar Class:

formatmonth( ) :

The formatmonth( ) method is used to get a month's calendar as an HTML table.

Syntax:

formatmonth(theyear, themonth, withyear=True)

Ex:

```
import calendar
htm= calendar.HTMLCalendar(firstweekday=0)  print(htm.formatmonth(2016, 5))
```

isleap( ):

The isleap() method returns True if the year is a leap year, otherwise False.

Syntax:

isleap(year)

Ex:

```
  import calendar
print(calendar.isleap(2020))
```

leapdays( ) :

The leapdays() method is used to get the number of leap years in a specified range of years. This function works for ranges spanning a century change.

Syntax:

leapdays(y1, y2)

Ex:

```
import calendar
print(calendar.leapdays(2015, 2020))
```

weekheader( ):

The weekheader() method is used to get a header containing abbreviated weekday names.

Syntax:

```
weekheader(n)
```

Ex:

```
import calendar
print(calendar.weekheader(3))
```

calendar( ) :

The calendar() method is used to get a 3-column calendar for an entire year as a multi-line string using the formatyear() of the TextCalendar class.

Syntax:

```
calendar(year, w=2, l=1, c=6, m=3)
```

Ex:

```
import calendar
print(calendar.calendar(2020))
```

month( ):

The month() method is used to get a month's calendar in a multi-line string using the formatmonth() of the TextCalendar class.

Syntax:

```
month(theyear, themonth, w=0, l=0)
```

Example:

```
import calendar
print(calendar.month(2021,3))
```

Ex:

```
while True:
    print("Options: ")
    print("Enter 'Yes' to Display Calendar: ")
    print("Enter Quit to End Program: ")
    user_input=input(":")
    if user_input=="Quit":
        break
    elif user_input=="Yes":
```

```
import calendar
Y=int(input("Enter Year: "))
M=int(input("Enter Month: "))
print(calendar.month(Y.M))
```

COPY MODULE

This module is used to copy the values from one object to another object.

Three ways

1. using assignment operator
2. copy module
    a). copy()
    b). deepcopy()
3. using slice operator

1. assignment operator: To copy the values from one object to another object.

if any changes in one object it will automatically effect another object.

Ex:

```
list1=[1,2,3,4]
list2=list1
list1.append(10)
print(list1)
print(list2)
list1.remove(3)
print(list1)
print(list2)
```

using copy module copy() in lists: In case of lists if any changes in one object it will not effect in another object.

Ex:

```
import copy
list1=[[1,2],[3,4]] list2=copy.copy(list1)
list1[0][0]='aa'
```

```
print(list1)
print(list2)
```

using copy module copy( ) in nested list: In case of nested list if any changes in one object it will automatically effect in another object.

Ex:

```
import copy
list1=[[1,2],[3,4]] list2=copy.copy(list1)
list1[0][0]='aa'
print(list1)
print(list2)
```

deepcopy(): It is used in nested lists. If any changes in one object it will not effect in another object.

Ex:

```
import copy
list1=[[1,2],[3,4]] list2=copy.deepcopy(list1) list1[0][0]='aa'
print(list1)
print(list2)
```

using slicing operator: Using slicing operator , to copy the elements from one object to another object. if any changes in one object it will not effect in anohter object.

Ex-1: To copy the elements

```
list1=[1,2,3,4]
list2=list1[:]
print(list1)
print(list2)
list2[0]=99
print(list1)
```

```
print(list2)
```

Ex-2: To copy the elements in a given range

```
list1=[1,2,3,4,5,6,7,8,9,10] list2=list1[2:10:3]
print(list1)
print(list2)
list2[0]=99
print(list1)
print(list2)
```

********** ALL THE BEST **********