Python Programming

MODULE - I

Agenda:

- Python Basics,
- Getting started,
- Python Objects,
- Numbers,
- Sequences:
- Strings,
- Lists,
- Tuples,
- Set and Dictionary.
- Conditionals and Loop Structures

Python Basics

- **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 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.
- 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.
- Python makes the development and debugging *fast* because there is no compilation step included in Python development, and edit-test-debug cycle is very fast.

Features of 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
- Open Source Language
- Platform Independent:

- Portability
- Dynamically Typed
- Procedure Oriented and Object Oriented
- Interpreted
- Extensible
- Embeddable
- Extensive Library

Easy to use and learn:

Python is a simple programming language. When we read Python program, we can feel like Reading English statements. The syntaxes are very simple and only 30+ keywords are available. When compared with other languages, we can write programs with very less number of lines. Hence more readability and simplicity.

Open Source Language:

We can use Python software without any licence and it is freeware.Its source code is open,so that we can we can customize based on our requirement.
Eg: Jython is customized version of Python to work with Java Applications.

Platform Independent:
Once we write a Python program, it can run on any platform without rewriting once again. Internally PVM is responsible to convert into machine understandable form.

Portability:
Python programs are portable. ie we can migrate from one platform to another platform very easily. Python programs will provide same results on any paltform.

Dynamically Typed:
In Python we are not required to declare type for variables. Whenever we are assigning the value, based on value, type will be allocated automatically.Hence Python is considered as dynamically typed language.But Java, C etc are Statically Typed Languages because we have to provide type at the beginning only.

Procedure Oriented and Object Oriented:
Python language supports both Procedure oriented (like C, pascal etc) and object oriented (like C++,Java) features. Hence we can get benefits of both like security and reusability etc

Interpreted:

We are not required to compile Python programs explcitly. Internally Python interpreter will take care that compilation. If compilation fails interpreter raised syntax errors. Once compilation success then PVM (Python Virtual Machine) is responsible to execute.

Extensible:

We can use other language programs in Python,The main advantages of this approach are:
1. We can use already existing legacy non-Python code
2. We can improve performance of the application

Embedded:

We can use Python programs in any other language programs. i.e we can embedd Python programs anywhere.

Extensive Library:

Python has a rich inbuilt library.Being a programmer we can use this library directly and we are not responsible to implement the functionality.

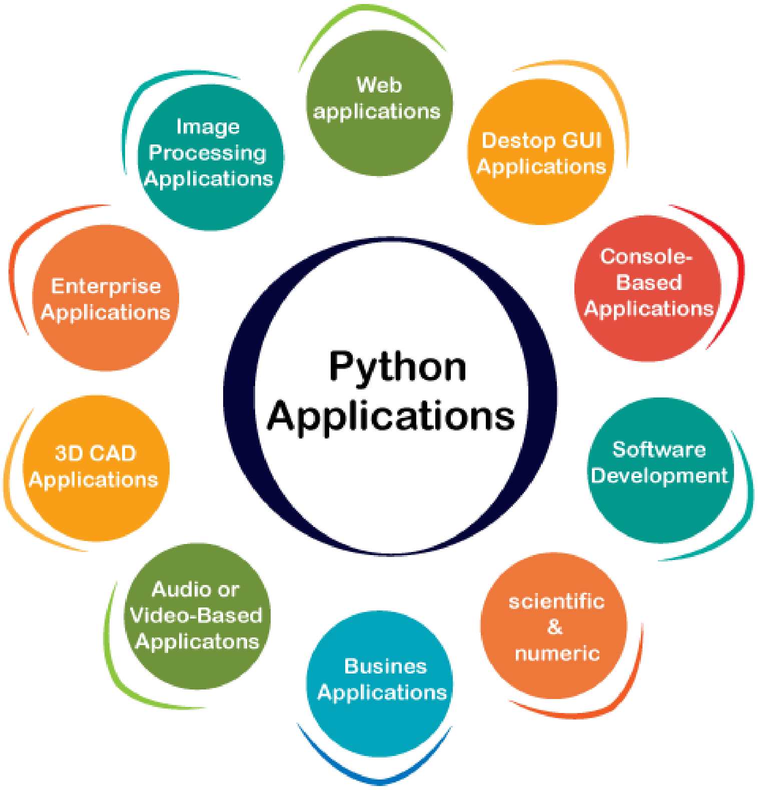
Versions of Python:

Python Version	Released Date
Python 1.0.0	January 1994
Python 1.5.0	December 31, 1997
Python 1.6	September 5, 2000
Python 2.0	October 16, 2000
Python 2.1	April 17, 2001
Python 2.2	December 21, 2001
Python 2.3	July 29, 2003
Python 2.4	November 30, 2004
Python 2.5	September 19, 2006
Python 2.6	October 1, 2008
Python 2.7	July 3, 2010
Python 3.0	December 3, 2008
Python 3.1	June 27, 2009
Python 3.2	February 20, 2011
Python 3.3	September 29, 2012
Python 3.4	March 16, 2014

Python 3.5	September 13, 2015
Python 3.6	December 23, 2016
Python 3.7	June 27, 2018
Python 3.8	October 14, 2019
Python 3.9	October 2020

Python Applications:

- ➢ The following are different area we can use python programming language



Input and output functions

In Python 2 the following 2 functions are available to read dynamic input from the keyboard.

1. raw_input()
2. input()

1. raw_input():

This function always reads the data from the keyboard in the form of String Format. We have to convert that string type to our required type by using the corresponding type casting methods.

Eg:

```
x=raw_input("Enter a Value:")
print(type(x)) It will always print str type only for any input type
```

2. input():

input() function can be used to read data directly in our required format.We are not required to perform type casting.

Eg:

```
x=input("Enter a Value)
type(x)
20 ===> int
"DS"===>str
125.5===>float
True==>bool
```

- In Python 3 we have only input() method and raw_input() method is not available.
- Python3 input() function behaviour exactly same as raw_input() method of Python2. i.e every input value is treated as str type only.

Example:

```
x=input("Enter First Number:")
y=input("Enter Second Number:")
a = int(x)
b = int(y)
print("Sum=",a+b)
```

output:

```
Enter First Number:10
Enter Second Number:20
Sum=30
```

OutPut Function:

We use the print() function or print keyword to output data to the standard output device (screen). This function prints the object/string written in function

Examples:

```
print("Hello World")
We can use escape characters also
 print("Hello \n World")
 print("Hello\tWorld")
 We can use repetetion operator (*) in the string
 print(10*"Hello")
print("Hello"*10)
```

We can use + operator also

```
print("Hello"+"World")
```

Python Comments:

- Python Comment is an essential tool for the programmers.
- Comments are generally used to explain the code. We can easily understand the code if it has a proper explanation.
- A good programmer must use the comments because in the future anyone wants to modify the code as well as implement the new module; then, it can be done easily.
- In the other programming language such as C, It provides the // for single-lined comment and /*.... */ for multiple-lined comment, but Python provides the single-lined Python comment.
- To apply the comment in the code we use the hash(#) at the beginning of the statement or code.

Let's understand the following example.

```
# This is the print statement
print("Hello Python")
```

Here we have written comment over the print statement using the hash(#). It will not affect our print statement.

Docstring in Python

- Python has the documentation strings (or docstrings) feature. It gives programmers an easy way of adding quick notes with every Python module, function, class, and method.
- You can define a docstring by adding it as a string constant. It must be the first statement in the object’s (module, function, class, and method) definition.
- The docstring has a much wider scope than a Python comment. Hence, it should describe what the function does, not how. Also, it is a good practice for all functions of a program to have a docstring.
- The strings defined using triple-quotation mark are docstring in Python. However, it might appear to you as a regular comment

Let's understand the following example.

```
'''
hello good morning
welcome to python
'''
print("Doc Sting")
```

```
“””
hello good morning
welcome to python
“””
print("Doc Sting")
```

Identifiers:

- A name in Python program is called identifier.
- It can be class name or function name or module name or variable name
- The following rules we have to follow while creating an didentifiers

1. Alphabet Symbols (Either Upper case OR Lower case)
2. If Identifier is start with Underscore (_) then it indicates it is private.
3. Identifier should not start with Digits.
4. Identifiers are case sensitive.
5. We cannot use reserved words as identifiers

Eg: def=10

6. There is no length limit for Python identifiers. But not recommended to use too lengthy identifiers.
7. Dollor ($) Symbol is not allowed in Python.

- The following are Examples
- myVar
- var_3
- cse_ds

Reserved Words

- In Python some words are reserved to represent some meaning or functionality. Such type of words are called Reserved words.
- We cannot use a keyword as a variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language.
- In Python, keywords are case sensitive.
- There are 33 keywords in Python 3.7. This number can vary slightly over the course of time.
- All the keywords except True, False and None are in lowercase and they must be written as they are. The list of all the keywords is given below.

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

Data Types or Objects

- Python is an object-oriented programming language, and in Python everything is an object.
- Objects are also called as Data structures.
- All the Data types in python are also called as Data types

- Data Type represents the type of data present inside a variable.
- In Python we are not required to specify the type explicitly. Based on value provided,the type will be assigned automatically.Hence Python is **Dynamically Typed Language.**

Python contains the following inbuilt data types are categorized as follows

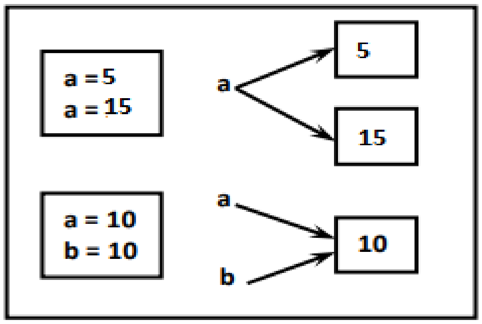
- **Fundamental or Build-in Data types or Data Structures**
- **Composite Data Types or Data Structures**

Object Type	Description	Example
Fundamental or Build-in Data types or Data Structures		
1. int	We can use to represent the whole/integral numbers	26,10,-12,-26
2. float	We can use to represent the decimal/floating point numbers	26.6,-26.2
3. complex	We can use to represent the complex numbers	26+26j
4. bool	We can use to represent the logical values(Only allowed values are True and False)	True,False
5. str	To represent sequence of Characters	“MREC “,”Raj”
Composite Data Types or Data Structures		

6. range	To represent a range of values	r=range(26)
		r1=range(1,26)
		r2=range(1,2,3)

7. list	To represent an ordered collection of objects	L1=[1,2,3,4,5,]
8. tuple	To represent an ordered collections of objects	t=(1,2,3,4,5)
9. set	To represent an unordered collection of unique objects	S={1,2,3,4,5}
10. dict	To represent a group of key value pairs	d={1:'Raj',2:'Sekhar'}
11. None	None means Nothing or No value associated.	a=None

Example:



Python contains several inbuilt functions as follows:

1.type() : to check the type of variable

2. id(): to get address of object

3. print(): to print the value

Example:

```
>>> a=10
>>> type(a)
<class 'int'>
>>> id(a)
2141527304784
>>> print(a)
10
```

Fundamental or Build-in Data types or Data Structures

1. int data type:

- We can use int data type to represent whole numbers (integral values)

Eg: a=10

type(a) #int

We can represent int values in the following ways

1. Decimal form
2. Binary form
3. Octal form
4. Hexa decimal form

1. Decimal form(base-10):

It is the default number system in Python

The allowed digits are: 0 to 9

Eg: a =10

2. Binary form(Base-2):

The allowed digits are : 0 & 1

Literal value should be prefixed with 0b or 0B

Eg: a = 0B1111

a =0B123

a=b111

3. Octal Form(Base-8):

The allowed digits are : 0 to 7

Literal value should be prefixed with 0o or 0O

Eg: a=0o123

a=0o786

4. Hexa Decimal Form(Base-16):

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

Literal value should be prefixed with 0x or 0X

Eg:

```
 a =0XFACE
 a=0XBeef
 a =0XBeer
```

Example:

```
>>> a=10
>>> b=0B0101
>>> c=0o121
>>> d=0xabc
>>> print(a)
10
>>> print(b)
5
>>> print(c)
81
>>> print(d)
2748
```

Base Conversions

Python provide the following in-built functions for base conversions

1.bin():

We can use bin() to convert from any base to binary

Eg:

```
>>> bin(5)
 '0b101'
>>> bin(0o11)
 '0b1001'
 >>> bin(0X10)
 '0b10000'
```

2. oct():

We can use oct() to convert from any base to octal

Eg:

```
>>> oct(10)
 '0o12'
 >>> oct(0B1111)
 '0o17'
```

```
>>> oct(0X123)
'0o443'
```

3. hex():

We can use hex() to convert from any base to hexa decimal

Eg:

```
>>> hex(100)
'0x64'
>>> hex(0B111111)
'0x3f'
>>> hex(0o12345)
'0x14e5'
```

2. float data type:

We can use float data type to represent floating point values (decimal values)

Eg: f=1.234

type(f) float

We can also represent floating point values by using exponential form (scientific notation)

Eg: f=1.2e3

print(f) 1200.0

instead of 'e' we can use 'E'

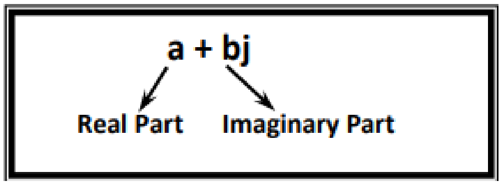
- The main advantage of exponential form is we can represent big values in less memory.
- We can represent int values in decimal, binary, octal and hexa decimal forms. But we can represent float values only by using decimal form.

Eg:

```
 >>> f=0B11.01
 File "<stdin>", line 1
 f=0B11.01
SyntaxError: invalid syntax

 >>> f=0o123.456
 SyntaxError: invalid syntax

 >>> f=0X123.456
 SyntaxError: invalid syntax
```

3. Complex Data Type:

A complex number is of the form



a and b contain intergers or floating point values

Eg:

```
6+3j
9+9.5j
0.5+0.9j
```

In the real part if we use int value or we can specify that either by decimal,octal,binary or hexa decimal form. But imaginary part should be specified only by using decimal form.

```
>>> a=0B011+4j
>>> a
 (3+4j)
 >>> a=3+0B011j
 SyntaxError: invalid syntax
```

Even we can perform operations on complex type values.

```
 >>> a=9+2.5j
 >>> b=4+3.9j
 >>>print(a+b)
(13+6.4j)
>>> a=(20+5j)
>>> type(a)
 <class 'complex'>
```

- Complex data type has some inbuilt attributes to retrieve the real part and imaginary part

```
c=15.4+6.6j
c.real==>15.4
c.imag==>6.6
```

- We can use complex type generally in scientific Applications and electrical engineering Applications

4. bool data type:

- We can use this data type to represent boolean values.
- The only allowed values for this data type are:**True and False**
- Internally Python represents True as 1 and False as 0

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

Eg:

```
a=20
b=30
c=a<b
print(c)==>True
True+True==>2
True-False==>1
```

5. str type:

- str represents String data type.
- A String is a sequence of characters enclosed within single quotes or double quotes.
  s1='MREC'
  s1="MREC"
- By using single quotes or double quotes we cannot represent multi line string literals.
  s1="MREC DS"
- For this requirement we should go for triple single quotes(''') or triple double quotes(""")

```
s1='''MREC
 DS'''
s1="""MREC
 DS"""
```

- We can also use triple quotes to use single quote or double quote in our String.

```
>>> s1="""This is mrec"""
>>> s1
'''This is mrec'''
```

- We can embed one string in another string

```
>>> s1='''This "Python Programming Session" for DS Students'''
>>> s1
'This "Python Programming Session" for DS Students'
```

Slicing of Strings:

- slice means a piece
- [ ] operator is called slice operator,which can be used to retrieve parts of String.
- In Python Strings follows zero based index.
- The index can be either +ve or -ve.
- +ve index means forward direction from Left to Right
- -ve index means backward direction from Right to Left

Eg:

-7 -6 -5 -4 -3 -2 -1

M	R	E	C		D	S
0	1	2	3	4	5	6

```
>>> s="MREC DS"
>>> s[0]
'M'
>>> s[-7]
'M'
>>> s[3]
'C'
>>> s[-4]
'C'
>>> s[-10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> s[50]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range

>>> s[1:4]
'REC'
>>> s[0:4]
'MREC'
>>> s[0:]
'MREC DS'
>>> s[:4]
'MREC'
>>> s[:]
'MREC DS'
>>> len(s)
7
```

Type Casting in Python

We can convert one type value to another type. This conversion is called Typecasting or Type conversion.

The following are various inbuilt functions for type casting.

1. int()
2. float()

3. complex()
4. bool()
5. str()

1.int():

- We can use this function to convert values from other types to int Type.
- We can convert from any type to int except complex type.
- we want to convert str type to int type, compulsary str should contain only integral value and should be specified in base-10

Eg:

```
1) >>> int(13.87)
2) 13
4) >>> int(True)
5) 1
6) >>> int(False)
7) 0
8) >>> int("19")
10) 19
11) >>> int(10+5j)
12) TypeError: can't convert complex to int
13) >>> int("10.5")
14) ValueError: invalid literal for int() with base 10: '10.5'
15) >>> int("ten")
16) ValueError: invalid literal for int() with base 10: 'ten'
17) >>> int("0B1111")
18) ValueError: invalid literal for int() with base 10: '0B1111'
```

2. float():

- We can use float() function to convert other type values to float type.
- We can convert any type value to float type except complex type.
- 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.

Eg:

```
1) >>> float(26)
2) 26.0
3) >>> float(True)
4) 1.0
5) >>> float(False)
6) 0.0
7) >>> float("26")
8) 26.0
```

```
9) >>> float("26.5")
10) 26.5
11) >>> float(26+5j)
12) TypeError: can't convert complex to float
13) >>> float("ten")
14) ValueError: could not convert string to float: 'ten'
15) >>> float("0B1011")
16) ValueError: could not convert string to float: '0B1011'
```

3.complex():

- We can use complex() function to convert other types to complex type.
- We can use this function to convert x into complex number with real part x and imaginary
- 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:

```
1)  complex(26)
      26+0j
2) complex(26.26)
     26.26+0j
3) complex(True)
      1+0j
4) complex(False)
       0j
5) complex("26")
     26+0j
6) complex("26.26")
     26.26+0j
7) complex("MREC")
ValueError: complex() arg is a malformed string
8)complex(26,26)
     26+26j
 9)complex(True,False)
    1+0j
```

4. bool():

- We can use this function to convert other type values to bool type.

Eg:

```
1)  bool(0)
```

```
    False
2)  bool(1)
    True
3)  bool(26)
    True
4)  bool(26.26)
    True
5)  bool(0.26)
    True
6)  bool(0.0)
    False
7)  bool(26-26j)
    True
8)  bool(0+26.26j)
    True
9)  bool(0+0j)
    False
10) bool("True")
    True
11) bool("False")
    True
12) bool("")
    False
```

5. str():

We can use this method to convert other type values to str type

Eg:

```
1) >>> str(26)
        '26'
3) >>> str(26.26)
         '26.26'
5) >>> str(26+5j)
        '(26+5j)'
7) >>> str(True)
        'True'
8)>>>str(False)
        ‘False’
```

Operators in Python

An operator is a symbol that tells the compiler to perform certain mathematical or logical Manipulations. Operators are used in program to manipulate data and variables.
Python language supports the following types of operators.

1. Arithmetic Operators
2. Relational Operators or Comparison Operators
3. Logical operators
4. Bitwise operators
5. Assignment operators
6. Special operators

1. Arithmetic Operators:

Arithmetic operators are used with numeric values to perform common mathematical operations:

- / operator always performs floating point arithmetic. Hence it will always returns float value.
- Floor division (//) can perform both floating point and integral arithmetic. If arguments are int type then result is int type. If at least one argument is float type then result is float type.

Assume variable ‘x’ holds 5 and variable ‘y’ holds 2, then:

Operator	Name	Example
+	Addition - Adds values on either side of the operator	x + y=7
-	Subtraction - Subtracts right hand operand from left hand operand	x – y=3
*	Multiplication - Multiplies values on either side of the operator	x * y=10
/	Division - Divides left hand operand by right hand operand	x / y=2.5
%	Modulus - Divides left hand operand by right hand operand and returns remainder	x % y=1
**	Exponent - Performs exponential (power) calculation on operators	x ** y=25
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	x // y=2

Eg:

```
>>> x=5
>>> y=2
>>> print('x+y=',x+y)
x+y= 7
```

```
>>> print('x-y=',x-y)
x-y= 3
>>> print('x*y=',x*y)
x*y= 10
>>> print('x/y=',x/y)
x/y= 2.5
>>> print('x%y=',x%y)
x%y= 1
>>> print('x**y=',x**y)
x**y= 25
>>> print('x//y=',x//y)
x//y= 2
```

2. Relational Operators or Comparison Operators

Comparison operators are used to compare two values:

Assume variable ‘x’ holds 5 and variable ‘y’ holds 2, then:

Operator	Name	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true	x == y=False
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	x != y=True
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	x > y=True
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	x < y=False
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	x >= y=True
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	x <= y=False

Eg:

```
>>> x=5
>>> y=2
>>> print('x==y=',x==y)
x==y= False
>>> print('x!=y=',x!=y)
x!=y= True
>>> print('x>y=',x>y)
x>y= True
```

```
>>> print('x<y=',x<y)
x<y= False
>>> print('x>=y=',x>=y)
x>=y= True
>>> print('x<=y=',x<=y)
x<=y= False
```

3. Logical operators:

Logical operators are used to combine conditional statements:

X	Y	X AND Y	X OR Y	NOT X
False	False	False	False	True
False	True	False	True	True
Ture	False	False	True	False
True	True	True	True	False

Assume variable ‘x’ holds 5 and variable ‘y’ holds 2, then:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Eg:

```
>>> x=5
>>> y=2
>>> x and y
2
>>> print(x>=5 and y<=5)
True
>>> print(x>=5 or y<=5)
True
>>> print(not x>=5)
False
```

4. Bitwise operators:

- Bitwise operator works on bits and performs bit by bit operation.
- We can apply these operators bitwise on int and boolean types.
- By mistake if we are trying to apply for any other type then we will get Error.

Truth table for bit wise operation

x	Y	x|y	x & y	x ^ y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

Bit wise operators

Operator_symbol	Operator_name
&	Bitwise_AND
|	Bitwise OR
~	Bitwise_NOT
^	XOR
<<	Left Shift
>>	Right Shift

Operator	**Name**	**Description**
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Eg:

```
>>> x=5
>>> y=2
>>> print('x & y=',x&y)
x & y= 0
>>> print('x | y=',x|y)
x | y= 7
>>> print('X ^ y=',x^y)
X ^ y= 7
>>> print('~x=',~x)
~x= -6
>>> print('x>>1=',x>>1)
x>>1= 2
>>> print('y<<1=',y<<1)
y<<1= 4
```

6.Assignment operators:

Assignment operators are used to assign values to variables:

Operator	Example	Equal to
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Eg:

```
>>> x=5
>>> x+=3
>>> print('x=x+3=',x)
x=x+3= 8
>>> x-=3
>>> print('x=x-3=',x)
x=x-3= 5
>>> x*=3
>>> print('x=x*3=',x)
x=x*3= 15
>>> x/=3
>>> print('x=x/3=',x)
x=x/3= 5.0
>>> x%=3
>>> print('x=x%3=',x)
x=x%3= 2.0
>>> x//=3
>>> print('x=x//3=',x)
x=x//3= 0.0
```

```
>>> x**=3
>>> print('x=x**3=',x)
x=x**3= 0.0
>>> x=5
>>> x&=3
>>> print('x=x&3=',x)
x=x&3= 1
>>> x|=3
>>> print('x=x|3=',x)
x=x|3= 3
>>> x^=3
>>> print('x=x^3=',x)
x=x^3= 0
>>> x>>=3
>>> print('x=x>>3=',x)
x=x>>3= 0
>>> x<<=3
>>> print('x=x<<3=',x)
x=x<<3=0
```

5. Special operators:

Python defines the following 2 special operators

1. Identity Operators
2. Membership operators

1. Identity Operators

- Identity Operators in Python are used to compare the memory location of two objects. The two identity operators used in Python are (is, is not).

- Operator is: It returns true if two variables point the same object and false otherwise
- Operator is not: It returns false if two variables point the same object and true otherwise2 identity operators are available.

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

Eg:

```
>>> x=5
>>> y=5
>>> print(x is y)
True
>>> print(id(x))
2265011481008
>>> print(id(y))
2265011481008
>>> print(x is not y)
False
```

2. Membership Operators

- These operators test for membership in a sequence such as lists, strings or tuples. There are two membership operators that are used in Python. (in, not in). It gives the result based on the variable present in specified sequence or string
- For example here we check whether the value of x=4 and value of y=8 is available in list or not, by using in and not in operators.

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

Eg:

```
>>> x="MREC CSE-DS Dept"
>>> print('M' in x)
True
>>> print('-' in x)
True
>>> print('DS' in x)
True
>>> print('1' not in x)
True
>>> print('M' not in x)
False
```

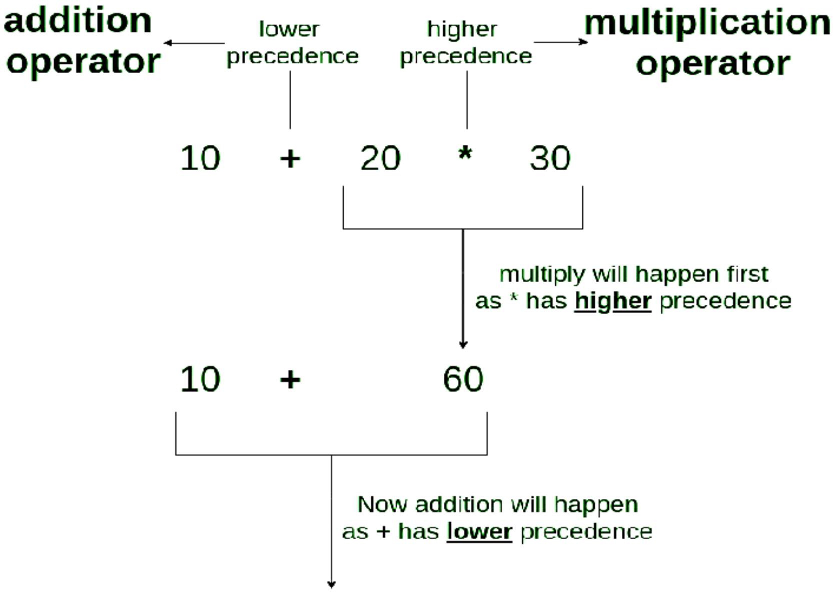
Precedence and Associativity of Operators in Python

- When an expression has more than one operator, then it is the relative priorities of the operators with respect to each other that determine the order in which the expression is evaluated.

Operator Precedence: This is used in an expression with more than one operator with different precedence to determine which operation to perform first.

Eg:10+20*30

10 + 20 * 30 is calculated as 10 + (20 * 30) and not as (10 + 20) * 30



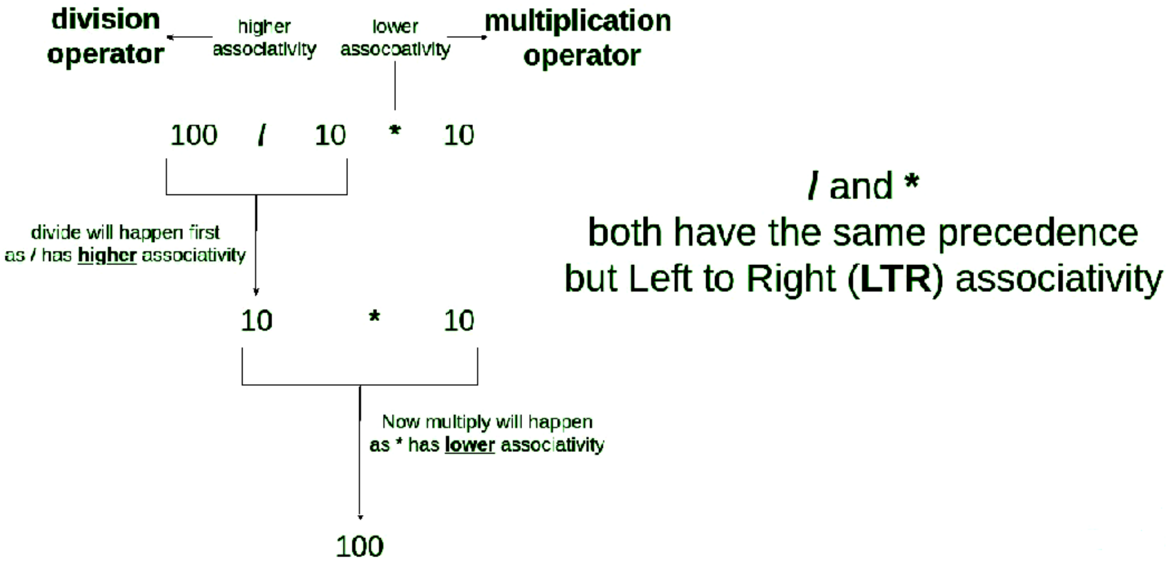
Example:

```
>>> exp=10+20*30
>>> print(exp)
610
```

Operator Associativity:

- When two operators have the same precedence, associativity helps to determine the order of operations.
- Associativity is the order in which an expression is evaluated that has multiple operators of the same precedence. Almost all the operators have left-to-right associativity.
- For example, multiplication and floor division have the same precedence. Hence, if both of them are present in an expression, the left one is evaluated first.

Example: '*' and '/' have the same precedence and their associativity is Left to Right, so the expression "100 / 10 * 10" is treated as "(100 / 10) * 10".


division operator
higher associativity
lower assocoativity
multiplication operator
100 / 10 * 10
divide will happen first as / has higher associativity
10 * 10
Now multiply will happen as * has lower associativity
100
/ and *
both have the same precedence
but Left to Right (LTR) associativity

Example:

```
>>> exp=100/10*10
>>> print(exp)
100.0
```

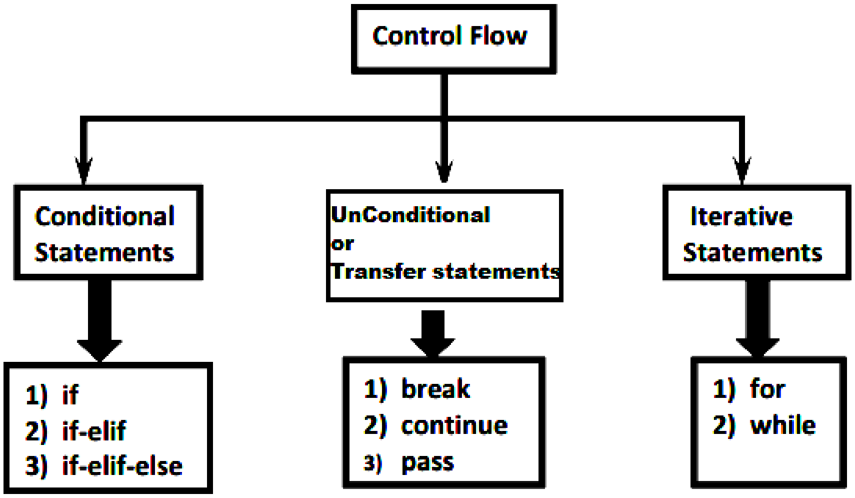
- Please see the following precedence and associativity table for reference. This table lists all operators from the highest precedence to the lowest precedence.

Operator	Description	Associativity
( )	Parentheses	left-to-right
**	Exponent	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ –	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
is, is not in, not in	Identity Membership operators	left-to-right

&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
Not	Logical NOT	right-to-left
And	Logical AND	left-to-right
Or	Logical OR	left-to-right
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left

Conditionals and Loop Structures

A control structure directs the order of execution of the statements in program.The Control statements as categorized as follows.



Conditional statement

- Conditional statements will decide the execution of a block of code based on the expression.
- The conditional statements return either True or False.
- A Program is just a series of instructions to the computer, But the real strength of Programming isn’t just executing one instruction after another. Based on how the expressions evaluate, the program can decide to skip instructions, repeat them, or choose one of several instructions to run. In fact, you almost never want your programs to start from the first line of code and simply execute every line, straight to the end. Flow control statements can decide which Python instructions to execute under which conditions.
- Python supports four types of conditional statements,

1. Simple if or if statement
2. if – else Statement
3. if else if (elif) Statement
4. nested if statement

Indentation: Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

1) Simple if or if statement

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

If condition is true then statements will be executed

Example:

```
>>> a=10
>>> b=5
>>> if(a>b):
            print("a is big")

a is big
>>> if a>b:
            print("a  is big")
a is big
```

2) if else:

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

if condition is true then Statements-1 will be executed otherwise Statements-2 will be executed.

Example:

```
>>> a=10
>>> b=25
>>> if(a>b):
            print("a is big")
else:
            print("b is big")
```

```
b is big
```

3) if elif else:

Syntax:

```
if condition1:
 Statements-1
elif condition2:
Statements -2
elif condition3:
Statements -3
elif condition4:
Statements -4
 …
else:
 Default Action
```

Based condition the corresponding action will be executed.

Example:

```
>>> Option=int(input("Enter a value b/w(1-5)"))
Enter a value b/w(1-5)2
>>> if(Option==1):
        print("you entered one")
elif(Option==2):
        print("You entered Two")
elif(Option==3):
        print("You entered Three")
elif(Option==4):
        print("You entered Four")
elif(Option==5):
        print("You entered Five")
else:
        print("Enter Value b/w (1-5) only")

You entered Two
```

4. nested if statement

We can use if statements inside if statements, this is called nested if statements.

Synatx:

```
if (condition1):
  # Executes when condition1 is true
  if (condition2):
    # Executes when condition2 is true
  # if Block is end here
# if Block is end here
```

Example:

```
>>> username=input("enter user name:")
enter user name:Raj
>>> pwd=input("Enter password")
Enter passwordRaj
>>> if(username=="Raj"):
        if(pwd=="Raj"):
                print("Login successful:")
        else:
                print("Invalid pwd")
else:
        print("Invalid Username")

Login successful:
```

Iterative Statements

If we want to execute a group of statements multiple times then we should go for Iterative statements.

Python supports 2 types of iterative statements.

1. **for loop**
2. **while loop**

1) for loop:

If we want to execute some action for every element present in some sequence(it may be string or collection)then we should go for for loop.

Syntax:

```
for x in sequence :
         body
```

Where sequence can be string or any collection.

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

Eg 1: To print characters present in the given string

```
>>> s="MREC"
>>> for r in s:
        print(r)
M
```

```
R
E
C
```

Eg2: To print characters present in string index wise:

```
>>> i=0
>>> for x in s:
        print('The character present at ',i,'index:',x)
        i+=1

The character present at  0 index: M
The character present at  1 index: R
The character present at  2 index: E
The character present at  3 index: C
```

Eg3: To print Sequence of values:

```
>>> for i in (1,2,3,4,5):
        print(i)

1
2
3
4
5
```

2) while loop:

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

Syntax:

```
while condition :
     body
```

Eg: To print numbers from 1 to 5 by using while loop

```
>>> i=1
>>> while(i<=5):
        print(i)
        i+=1
1
2
3
4
5
```

Eg: To display the sum of first n numbers

```
n=int(input("Enter n value:"))
sum=0
i=1
while i<=n:
   sum=sum+i
   i=i+1
print("sum of ",n," elements are=",sum)
```

OutPut:

```
Enter n value:5
sum of  5  elements are= 15
```

Nested Loops:

- Sometimes we can take a loop inside another loop,which are also known as nested loops
- A nested loop is a loop inside a loop.
- The "inner loop" will be executed one time for each iteration of the "outer loop":

Syntax:

```
while expression:
         while expression:
                  statement(s)
     statement(s)
```

Eg1:

```
r=1
while(r<=3):
   c=1
   while(c<=5):
      print("r=",r,"c=",c)
      c=c+1
   print('\n')
   r=r+1
```

OutPut:

```
r= 1 c= 1       r=2   c=1       r=3   c=1
r= 1 c= 2       r=2   c=2       r=3   c=2
r= 1 c= 3       r=2   c=3       r=3   c=3
r= 1 c= 4       r=2   c=4       r=3   c=4
r= 1 c= 5       r=2   c=5       r=3   c=5
```

Eg2:

```
for r in (1,2,3):
   for c in (1,2,3,4,5):
```

```
            print('r=',r,'c=',c)
          print('\n')
```

OutPut:

```
r= 1 c= 1      r=2   c=1      r=3   c=1
r= 1 c= 2      r=2   c=2      r=3   c=2
r= 1 c= 3      r=2   c=3      r=3   c=3
r= 1 c= 4      r=2   c=4      r=3   c=4
r= 1 c= 5      r=2   c=5      r=3   c=5
```

Transfer Statements

1) break:

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

Eg:

```
for r in (1,2,3,4,5):
   if(r==3):
      print("Break the loop")
      break
   print(r)
```

OutPut:

```
1
2
Break the loop
```

2) continue:

- We can use continue statement to skip current iteration and continue next iteration.

Eg 1: To print even numbers in the range 1 to 10

```
for r in (1,2,3,4,5,6,7,8,9,10):
   if(r%2!=0):
      continue
   print(r)
```

OutPut:

```
2
4
6
8
10
```

Composite Data Types or Data Structures

- The following are different Composite data type in python

6. range Data Type:

- range Data Type represents a sequence of numbers. The elements present in range Data type are not modifiable. i.e range Data type is immutable
- We can access elements present in the range Data Type by using index.

Eg:

1. **range(5)→generate numbers from 0 to 4**

   Eg:

   ```
   r=range(5)
   for i in r :
           print(i)
   OutPut: 0 1 2 3 4 5
   ```

2. **range(5,10)→generate numbers from 5 to 9**

   ```
   r = range(5,10)
   for i in r :
           print(i)
   OutPut:5 6 7 8 9
   ```

3. **range(1,10,2)→2 means increment value**

   ```
   r = range(1,10,2)
   for i in r :
           print(i)
   OutPut: 1 3 5 7 9
   ```

4.
   ```
   r=range(0,5)
   r[0]==>0
   r[15]==>IndexError: range object index out of range
   ```

   We cannot modify the values of range data type

7.list data type:

- If we want to represent a group of values as a single entity where insertion order required to preserve and duplicates are allowed then we should go for list data type.
- An ordered, mutable, heterogeneous collection of elements is nothing but list, where Duplicates also allowed.
- insertion order is preserved
- heterogeneous objects are allowed
- duplicates are allowed
- Growable in nature
- values should be enclosed within square brackets.

  1. **Eg:**

```
 list=[26,26.5,'Raj',True]
 print(list)
output→ [26,26.5,'Raj',True]
```

2. Eg:

```
 list=[10,20,30,40]
 >>> list[0]
 10
 >>> list[-1]
 40
 >>> list[1:3]
 [20, 30]
 >>> list[0]=100
 >>> print(list)
 …
 100
 40
 30
 40
```

- list is growable in nature. i.e based on our requirement we can increase or decrease the size.

```
>>> list=[10,20,30]
>>> list.append("raj")
>>> list
[10, 20, 30, ‘raj’]
>>> list.remove(20)
 >>> list
 [10, 30, ‘raj’]
 >>> list1=list*2
>>> list1
 [10, 30, 'raj', 10, 30, 'raj']
```

Creating list by using range data type:

- We can create a list of values with range data type

  Eg:

```
>>> l = list(range(5))
>>>print(l)
[0, 1, 2, 3, 4]
```

8. tuple data type:

- tuple data type is exactly same as list data type except that it is immutable.i.e we cannot chage values.
- Tuple elements can be represented within parenthesis.
- tuple is the read only version of list

Eg:

```
>>> t1=(1,2,3,4)
 >>>type(t)
 <class 'tuple'>
 >>>t1[0]=26
 TypeError: 'tuple' object does not support item assignment
 >>> t.append("Raj")
 AttributeError: 'tuple' object has no attribute 'append'
 >>> t.remove(2)
 AttributeError: 'tuple' object has no attribute 'remove'
```

9. set Data Type:

- If we want to represent a group of values without duplicates where order is not important then we should go for set Data Type
    - insertion order is not preserved
    - duplicates are not allowed
    - heterogeneous objects are allowed
    - index concept is not applicable
    - It is mutable collection
    - Growable in nature, based on our requirement we can increase or decrease the size

Eg:

```
>>> s={1,2,"raj",True,1,2}
>>> s
{1, 2, 'raj'}
>>> s.remove(2)
>>> s
{1, 'raj'}
>>> s.add(10)
>>> s
{1, 10, 'raj'}
>>> s.add("MREC")
>>> s
{1, 10, 'raj', 'MREC'}
```

10.dict Data Type:

- If we want to represent a group of values as key-value pairs then we should go for dict data type.
- Duplicate keys are not allowed but values can be duplicated. If we are trying to insert an entry with duplicate key then old value will be replaced with new value.

Eg:

```
>>> d={1:"one",2:"Two",3:"Three"}
>>> d[1]
'one'
>>> d
{1: 'one', 2: 'Two', 3: 'Three'}
>>> d[4]="Four"
>>> d
{1: 'one', 2: 'Two', 3: 'Three', 4: 'Four'}
>>> d[5]="error"
>>> d
{1: 'one', 2: 'Two', 3: 'Three', 4: 'Four', 5: 'error'}
>>> d[5]="Five"
>>> d
{1: 'one', 2: 'Two', 3: 'Three', 4: 'Four', 5: 'Five'}
```

11. None Datatype:

- The None Datatype is used to define the null value or no value, the none value means not 0, or False value, and it is a data it's own
- None keyword is an object and is a data type of nonetype class
- None datatype doesn’t contain any value.
- None keyword is used to define a null variable or object.
- None keyword is immutable.

Eg:

Assume a=10, that means a is the reference variable pointing to 10 and if I take a=none then a is not looking to the object 10

```
>>> a=10
>>> type(a)
<class 'int'>
>>> a=None
>>> type(a)
<class 'NoneType'>
```

Python Programming

MODULE - II

Agenda:

- Modules: Modules and Files
- Namespaces
- Importing Modules,
- Importing Module Attributes,
- Module Built-in Functions,
- Packages,
- Other Features of Modules
- Files: File Objects,
- File Built-in Function,
- File Built-in Methods,
- File Built-in Attributes,
- Standard Files,
- Command-line Arguments,
- File System,
- File Execution,
- Persistent Storage Modules.
- Exceptions: Exceptions in Python,
- Detecting and Handling Exceptions,
- Context Management,
- Exceptions as Strings,
- Raising Exceptions,
- Assertions,
- Standard Exceptions,
- Creating Exceptions,
- Why Exceptions,
- Why Exceptions at All?
- Exceptions and the sys Module.

Modules

- Like many other programming languages, Python supports modularity. That is, you can break large code into smaller and more manageable pieces. And through modularity, Python supports code reuse.
- We can import modules in Python into your programs and reuse the code therein as many times as you want.
- Modules provide us with a way to share reusable functions.

A module is simply a "Python file" which contains code we can reuse in multiple Python programs. A module may contain functions, classes, lists, etc.

- Modules in Python can be of two types:
  1. Built-in Modules.
  2. User-defined Modules.

1. Built in Modules in Python

- One of the many superpowers of Python is that it comes with a "rich standard library". This rich standard library contains lots of built-in modules. Hence, it provides a lot of reusable code.
- In Python, modules are accessed by using the import statement
- When our current file is needed to use the code which is already existed in other files then we can import that file (module).
- When Python imports a module called module1 for example, the interpreter will first search for a built-in module called module1. If a built-in module is not found, the Python interpreter will then search for a file named module1.py in a list of directories that it receives from the sys.path variable.
- We can import module in three different ways:
  1. *import <module_name>*
  2. *from <module_name> import <method_name>*
  3. *from <module_name> import **

1.import <module_name>:

- This way of importing module will import all methods which are in that specified module.

  Eg: import math
- Here this import statement will import all methods which are available in math module. We may use all methods or may use required methods as per business requirement.

2.From <module_name> import <method_name>:

- This import statement will import a particular method from that module which is specified in the import statement.
- We can't use other methods which are available in that module as we specified particular method name in the import statement.
- The main advantage of this is we can access members directly without using module name.

```
Eg:from <module_name>import <*>
from math import factorial
from math import*
```

Finding members of module by using dir() function:

- Python provides inbuilt function dir() to list out all members of current module or a Specified module.
- **dir()** ===>To list out all members of current module
- **dir(moduleName)**==>To list out all members of specified module

1.Eg:

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__']
```

2.Eg:

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist',
'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma',
'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod',
'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
```

Some of Standard modules

- **Math module**
- **Calendar module**

Working with math module:

- Python provides inbuilt module math.
- This module defines several functions which can be used for mathematical operations.
- Some main important functions are

1. sqrt(x)
2. ceil(x)
3. floor(x)
4. fabs(x)
5. log(x)
6. sin(x)
7. tan(x)
8.factorial(x)

….

Eg:

```
>>> from math import*
>>> print(sqrt(5))
2.23606797749979
>>> print(ceil(15.25))
16
>>> print(floor(15.25))
15
>>> print(fabs(-15.6))
15.6
>>> print(fabs(15.6))
15.6
>>> print(log(10.5))
2.3513752571634776
>>> print(sin(1))
0.8414709848078965
>>> print(tan(0))
0.0
>>> print(factorial(5))
120
```

Working with Calendar module:

- Python defines an inbuilt module calendar which handles operations related to calendar.

- Calendar module allows output calendars like the program and provides additional useful functions related to the calendar.

calendar.day_name: An array that represents the days of the week in the current locale.

1. Displaying all week names one by one

```
import calendar
for i in calendar.day_name:
 print(i)

output:
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

calendar.month_name:

An array that represents the months of the year in the current locale. This follows normal convention of January being month number 1, so it has a length of 13 and month_name[0] is the empty string.

```
>>> import calendar
>>> for i in calendar.month_name:
        print(i)
January
February
March
April
May
June
July
August
September
October
November
December
```

calendar.monthrange(year, month): Returns weekday of first day of the month and number of days in month, for the specified year and month.

```
>>> import calendar
>>> print(calendar.monthrange(2021,6))
```

```
(1, 30)
>>> print(calendar.monthrange(2021,7))
(3, 31)
>>> print(calendar.monthrange(2022,1))
(5, 31)
>>> print(calendar.monthrange(2021,1))
(4, 31)
```

calendar.isleap(year): Returns True if year is a leap year, otherwise False.

```
>>> import calendar
>>> print(calendar.isleap(2020))
True
>>> print(calendar.isleap(2021))
False
```

calendar.leapdays(y1, y2): Returns the number of leap years in the range from y1 to y2 (exclusive), where y1 and y2 are years.

```
>>> import calendar
>>> print(calendar.leapdays(2000,2020))
5
```

calendar.weekday(year, month, day): Returns the day of the week (0 is Monday) for year (1970–…), month (1–12), day (1–31)

```
>>> import calendar
>>> print(calendar.weekday(2020,5,1))
4
>>> print(calendar.weekday(2021,5,1))
5
```

calendar.weekheader(n): Return a header containing abbreviated weekday names. n specifies the width in characters for one weekday

```
>>> import calendar
>>> print(calendar.weekheader(1))
M T W T F S S
>>> print(calendar.weekheader(3))
Mon Tue Wed Thu Fri Sat Sun
>>> print(calendar.weekheader(10))
  Monday    Tuesday   Wednesday   Thursday    Friday    Saturday    Sunday
```

calendar. calendar(year, w, l, c): Returns a 3-column calendar for an entire year as a multi-line string using the formatyear() of the TextCalendar class.

This function shows the year, width of characters, no. of lines per week and column separations.

```
>>> import calendar
>>> print(calendar.calendar(2021))
Output:prints 2021 full calendar
```

1. *User defined Modules.*

- Another superpower of Python is that it lets you take things in your own hands.
- A python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module. We may have a runnable code inside the python module.
- Modules in Python provides us the flexibility to organize the code in a logical way.
- To use the functionality of one module into another, we must have to import the specific module.

Creating a Module:

Shown below is a Python script containing the definition of sum() function. It is saved as calc.py.

```
#calc.py
def sum(x, y):
   return x + y
def sub(x, y):
   return x - y
def mul(x, y):
   return x * y
def di(x, y):
   return x / y
```

Importing a Module

We can now import this module and execute the any functions which are there in calac.py module in the Python shell.

```
>>> import calc
>>> print(calc.sum(4,5))
9
>>> print(calc.sub(4,5))
-1
>>> print(calc.mul(4,5))
20
```

```
>>> from calc import *
>>> print(sum(4,5))
9
>>> print(sub(4,5))
-1
>>> print(mul(4,5))
20
```

- Every module, either built-in or custom made, is an object of a module class. Verify the type of different modules using the built-in type() function, as shown below.

```
>>> import calc
>>> type(calc)
<class 'module'>
>>> import math
>>> type(math)
<class 'module'>
```

Renaming the Imported Module

Use the as keyword to rename the imported module as shown below.-

```
>>> import calc as c
>>> import math as raj
>>> import calc as c
>>> import math as raj
>>> print(c.sum(4,5))
9
>>> print(raj.factorial(5))
120
```

Namespaces

- Generally speaking, a **namespace** is a naming system for making names unique to avoid ambiguity.
- Everybody knows a namespacing system from daily life, i.e. the naming of people in firstname and familiy name (surname).
- A namespace is a simple system to control the names in a program. It ensures that names are unique and won’t lead to any conflict.
- Some namespaces in Python:
    1. Local Namespace
    2. Global Namespace

3. Built-in Namespace

Local Namespace:

The Variables which are defined in the function are a local scope of the variable. These variables are defined in the function body.

Global Namespace

The Variable which can be read from anywhere in the program is known as a global scope. These variables can be accessed inside and outside the function. When we want to use the same variable in the rest of the program, we declare it as global.

Eg:

```
n=0#global namesapce
def f1():
   n=1#local namespace
   print("local variable n=",n)
f1()
print("Global variable n=",n)
```

OutPut:

```
local variable n= 1
Global variable n= 0
```

Built-in Scope

- If a Variable is not defined in local,or global scope, then python looks for it in the built-in scope.
- In the Following Example, 1 from math module pi is imported, and the value of pi is not defined in global, local and enclosed.
- Python then looks for the pi value in the built-in scope and prints the value. Hence the name which is already present in the built-in scope should not be used as an identifier.

Eg:

```
# Built-in Scope
from math import pi
# pi = 'Not defined in global pi'
def f1():
   print('Not defined in f1() pi')
def f2():
   print('Not defined in f2() pi')
f1()
f2()
print('pi is Built-in scope',pi)
```

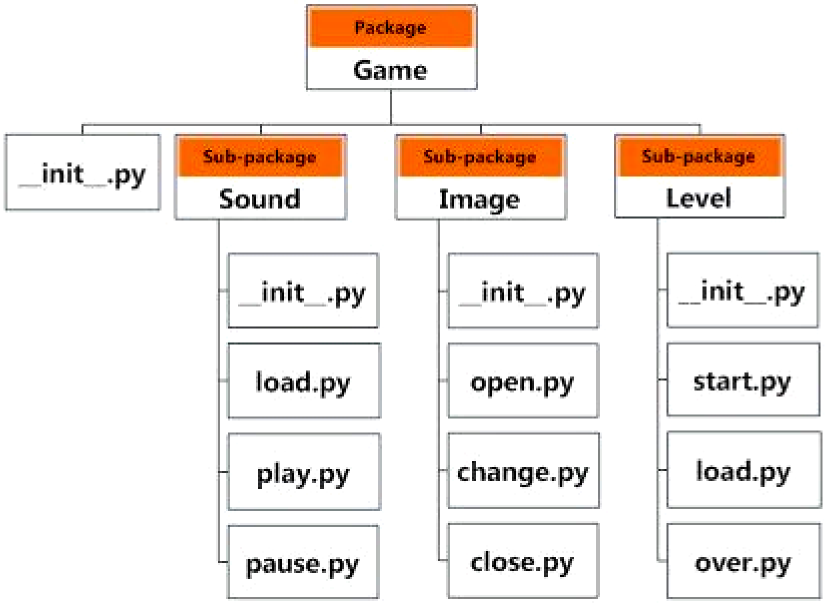
OutPut:

```
Not defined in f1() pi
Not defined in f2() pi
pi is Built-in scope 3.141592653589793
```

Packages in Python

- A Package is nothing but a collection of modules. It is also imported into programs.
- In Package, several modules are present, which you can import in your code.
- Packages are a way of structuring many packages and modules which helps in a well-organized hierarchy of data set, making the directories and modules easy to access.
- Just like there are different drives and folders in an OS to help us store files, similarly packages help us in storing other sub-packages and modules, so that it can be used by the user when necessary.
- Similarly, as a directory can contain subdirectories and files, a Python package can have sub-packages and modules.
- A directory must contain a file named **__init__.py** in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.
- Any folder or directory contains **__init__.py** file,is considered as a Python package.This file can be empty.
- As we discussed, a package may hold other Python packages and modules. But what distinguishes a package from a regular directory? Well, a Python package must have an **__init__.py** file in the directory.
- You may leave it empty, or you may store initialization code in it. But if your directory does not have an **__init__.py** file, it isn't a package; it is just a directory with a bunch of Python scripts. Leaving **__init__.py** empty is indeed good practice.

Example: Suppose we are developing a game. One possible organization of packages and modules could be as shown in the figure below.



The Following Steps to be follow.

Step1: Create a folder or package

Step2: Inside the Folder create a sub folder or package

Step3: Inside the package we have to create _init_.py which indicate its a package

Step4: After that we can create some modules based on requirement

Step5: After that we have to create main module in package folder by importing the created modules in sub package.

Eg 1:

```
F:\>
 |-test.py
 |-python_package
 |-First.py
 |-Second.py
 |-__init__.py
```

test.py

```
from python_package import First,second
First.f1()
second.f2()
```

First.py

```
def f1():
   print("This is First function")
```

Second.py

```
def f2():
   print("This is Second Function")
```

OutPut:

```
This is First function
This is Second Function
```

Files Handling in Python

- Python File Handling Before we move into the topic "Python File Handling", let us try to understand why we need files?
- So far, we have been receiving the input data from the console and writing the output data back to the console.
- The console only displays a limited amount of data. Hence we don't have any issues if the input or output is small. What if the output or input is too large?
- We use files when we have large data as input or output.
- A file is nothing but a named location on disk which stores data.
- Files are also used to store data permanently since it stores data on non-volatile memory.
- Most modern file systems are composed of three main parts:
    1. **Header:** metadata about the contents of the file (file name, size, type, and so on)
    2. **Data:** contents of the file as written by the creator or editor
    3. **End of file (EOF):** special character that indicates the end of the file

Types of Files in Python

- Text File
- Binary File

1. Text File

- Text file store the data in the form of characters.
- Text file are used to store characters or strings.
- Usually we can use text files to store character data

  eg: abc.txt

2. Binary File

- Binary file store entire data in the form of bytes.
- Binary file can be used to store text, image, audio and video.

- Usually we can use binary files to store binary data like images,video files, audio files etc.

File operation on Text Files:

In Python, we can perform the following file operations:

- Open a file
- Read or write a file
- Close a file

Opening a File:

- Before performing any operations like read or write on a file, the first thing we need to do is open a file.
- Python provides an in-built function open() to open a file.
- The open function accepts two parameters: the name of the file and the access mode.
- The access mode specifies what operation we are going to perform on a file whether it is read or write.
- The open() function in turn returns a file object/handle, with which we can perform file operations based on the access mode.

Syntax: file_object=open(filename, access_mode)

- The allowed modes in Python are
- **r :** open an existing file for read operation. The file pointer is positioned at the beginning of the file. If the specified file does not exist then we will get FileNotFoundError. This is default mode.

- **w :** open an existing file for write operation. If the file already contains some data then it will be overridden. If the specified file is not already available then this mode will create that file.

- **a :** open an existing file for append operation. It won't override existing data. If the specified file is not already available then this mode will create a new file.

- **r+ :** To read and write data into the file. The previous data in the file will not be deleted. The file pointer is placed at the beginning of the file.

- **w+ :** To write and read data. It will override existing data.

- **a+ :** To append and read data from the file.It wont override existing data.

- **x :** To open a file in exclusive creation mode for write operation. If the file already exists then we will get FileExistsError.

❖ All the above modes are applicable for text files. If the above modes suffixed with 'b' then these represents for binary files.

- **rb,wb,ab,r+b,w+b,a+b,xb**

Ex:

1. **file_object=open("test.txt") # when file is in the current directory**
2. **file_object=open("C:/User/Desktop/test.txt") # specify full path when file is in different directory**

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()

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)

Eg:1

```
1) f=open("abcd.txt",'w')
2) f.write("MREC \n")
3) f.write("CSE \n")
4) f.write("DS\n")
5)f.write("Dept\")
6) f.close()
abcd.txt:
MREC
CSE
DS
DEPT
```

Eg 2:

```
1) f=open("abcd.txt",'a')
2) list=["\nAI&ML\n","IOT\n","RAJ"]
3) f.writelines(list)
4) f.close()

abcd.txt:
MREC
CSE
DS
DEPT
AI&ML
IOT
RAJ
```

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

Eg 1: To read total data from the file

```
f=open("abc.txt",'r')
data=f.read()
print(data)
f.close()
```

Output

```
MREC
CSE
DS
DEPT
AI&ML
IOT
RAJ
```

Eg 2: To read only first 10 characters:

```
f=open("abc.txt",'r')
data=f.read(10)
print(data)
f.close()
```

Output

```
MREC
CSE
DS
```

Eg 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()
```

Output

```
MREC
 CSE
 DS
```

Eg 4: To read all lines into list:

```
f=open("abc.txt",'r')
lines=f.readlines()
for line in lines:
print(line,end='')
f.close()
```

Output

```
MREC
CSE
DS
DEPTS
AI&ML
IOT
RAJ
```

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.
- The position(index) of first character in files is zero just like string index.

Eg:

```
f=open('F:/abcd.txt','r')
print(f.tell())
print(f.read(2))
print(f.tell())
print(f.read(2))
f.close()
```

Output:

```
0
MR
2
EC
```

seek():

- We can use seek() method to move cursor(file pointer) to specified location.
- **Syntax:f.seek(offset)**

Eg:

```
f=open('F:/abcd.txt','r')
print(f.tell())
print(f.read(2))
print(f.tell())
print(f.read(2))
f.seek(0)
print(f.read(4))
f.seek(4)
print(f.read())
f.close()
```

```
output:     0
            MR
            2
            EC
            MREC
```

File Built in Attributes and Built in Methods

- Once we opened a file and we got file object, we can get various details related to that file by using its properties or attributes and methods on it.
- The Following are some of the attributes.
- **name -->** Name of opened file
- **mode -->**Mode in which the file is opened
- **closed -->**Returns boolean value indicates that file is closed or not

Eg:

```
>>>
f=open("C:/Users/rajas/AppData/Local/Programs/Python/Python39/m2.py",'r')
>>> f.name
'C:/Users/rajas/AppData/Local/Programs/Python/Python39/m2.py'
>>> f.mode
'r'
>>> f.closed
False
```

File Built in Methods

Python has the following set of methods available for the file object.

Method	Description
close()	Closes the file
fileno()	Returns a number that represents the stream, from the operating system's perspective
flush()	Flushes the internal buffer
isatty()	Returns whether the file stream is interactive or not
read()	Returns the file content
readable()	Returns whether the file stream can be read or not
readline()	Returns one line from the file
readlines()	Returns a list of lines from the file
seek()	Change the file position
seekable()	Returns whether the file allows us to change the file position
tell()	Returns the current file position
truncate()	Resizes the file to a specified size
writable()	Returns whether the file can be written to or not
write()	Writes the specified string to the file
writelines()	Writes a list of strings to the file

File close() Method

- Close a file after it has been opened:

```
f = open("raj.txt", "r")
print(f.read())
f.close()
```

File fileno() Method

- Return the file descriptor of the stream:

```
f = open("raj.txt", "r")
print(f.fileno())
```

File flush() Method

- The flush() method cleans out the internal buffer.
- You can clear the buffer when writing to a file:

```
f = open("myfile.txt", "a")
f.write("Now the file has one more line!")
f.flush()
f.write("...and another one!")
```

File isatty() Method

- The isatty() method returns True if the file stream is interactive, example: connected to a terminal device.

```
f = open("raj.txt", "r")
print(f.isatty())
```

File read() Method

- The read() method returns the specified number of bytes from the file. Default is -1 which means the whole file.

```
f = open("raj.txt", "r")
print(f.read())
```

File readable() Method

- The readable() method returns True if the file is readable, False if not.

```
f = open("raj.txt", "r")
print(f.readable())
```

File readline() Method

- The readline() method returns one line from the file.
- You can also specified how many bytes from the line to return, by using the size parameter.

```
f = open("demofile.txt", "r")
print(f.readline())
```

File readlines() Method

- The readlines() method returns a list containing each line in the file as a list item.

```
f = open("raj.txt", "r")
print(f.readlines())

f = open("raj.txt", "r")
print(f.readline())
```

File seek() Method

- The seek() method sets the current file position in a file stream.
- The seek() method also returns the new postion.

```
f = open("raj.txt", "r")
f.seek(4)
print(f.readline())
```

File seekable() Method

- The seekable() method returns True if the file is seekable, False if not.
- A file is seekable if it allows access to the file stream, like the seek() method.

```
f = open("raj.txt", "r")
print(f.seekable())
```

File tell() Method

- The tell() method returns the current file position in a file stream.

```
f = open("raj.txt", "r")
print(f.tell())
```

File truncate() Method

- The truncate() method resizes the file to the given number of bytes.
- If the size is not specified, the current position will be used.

```
f = open("demofile2.txt", "a")
f.truncate(20)
f.close()
```

```
#open and read the file after the truncate:
f = open("demofile2.txt", "r")
print(f.read())
```

File writable() Method

- The writable() method returns True if the file is writable, False if not.
- A file is writable if it is opened using "a" for append or "w" for write.

```
f = open("raj.txt", "a")
print(f.writable())
```

File write() Method

- The write() method writes a specified text to the file.
- Where the specified text will be inserted depends on the file mode and stream position.
- "a": The text will be inserted at the current file stream position, default at the end of the file.
- "w": The file will be emptied before the text will be inserted at the current file stream position, default 0.

```
f = open("demofile2.txt", "a")
f.write("See you soon!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

File writelines() Method

- The writelines() method writes the items of a list to the file.
- Where the texts will be inserted depends on the file mode and stream position.
- "a": The texts will be inserted at the current file stream position, default at the end of the file.
- "w": The file will be emptied before the texts will be inserted at the current file stream position, default 0.

```
f = open("raj.txt", "a")
f.writelines(["See you soon!", "Over and out."])
f.close()

#open and read the file after the appending:
f = open("raj.txt", "r")
print(f.read())
```

File operation on Binary Files:

- Binary file store entire data in the form of bytes.
- Binary file can be used to store text, image, audio and video.
- Usually we can use binary files to store binary data like images,video files, audio files etc.
- In Python, we can perform the following file operations:

- Open a file
- Read or write a file
- Close a file

Eg: program to Read an image and that to another.

```
f1=open('mrec.jpg','rb')
f2=open('mrec1.jpg','wb')
#bytes=f1.read()
f2.write(f1.read())
print("Image copied from f1 to f2:\n")
f1.close()
f2.close()
```

File System in python

- A file system is a process that manages how and where data on storage disk, typically a hard disk drive (HDD), is stored, accessed and managed. It is a logical disk component that manages a disk's internal operations as it relates to a computer and is abstract to a human user.
- A directory simply is a structured list of documents and folders. A directory can have sub-directories and files. When we have too many files, Python directory comes in handy in file management or system with directories and sub-directories.
- Python has os module with multiple methods defined inside for directory and file management or system

Working with Directories:

It is very common requirement to perform operations for directories like

To Know Current Working Directory:

```
import os
print("The cwd=",os.getcwd())
```

OutPut:

```
The cwd= C:\Users\rajas\AppData\Local\Programs\Python\Python39
```

To create a sub directory in the current working directory:

```
import os
```

```
os.mkdir('Raj')
print("The Directory Raj is Created”)
```

OutPut:

```
The Directory Raj is Created
```

To rename a directory in Python:

- Python has rename( ) function to rename a directory.

Syntax: os.rename(old_name,new_name)

```
import os
os.rename('Raj','mrec')
print("The Directory Raj Renamed to mrec")
```

OutPut:

```
The Directory Raj Renamed to mrec
```

To change directories in Python:

- In Python, chdir( ) function defined in module os is used to change the working directories.

Example: Suppose we want to change our working directory to Raj in F: Here is how it is done.

```
>>> import os
>>> os.getcwd()
'C:\\Users\\rajas\\AppData\\Local\\Programs\\Python\\Python39'
>>> os.chdir('F:/')
>>> os.getcwd()
'F:\\'
>>> os.mkdir('Raj')
>>> os.getcwd()
'F:\\'
>>> os.chdir('Raj')
>>> os.getcwd()
'F:\\Raj'
```

To list directories in Python:

- Python has listdir( ) function in module os to list all the directories and files in a particular location.
- listdir( ) returns a list containing the names of the entries in the directory given by path. The list is in arbitrary order, and does not include the special entries '.' and '..' even if they are present in the directory.

Here is an example:

```
>>> import os
>>> os.chdir('F:/')
>>> os.listdir()
['$RECYCLE.BIN', 'abcd.txt', 'add.txt', 'Applicant Details-Cloud.doc', 'c.py', 'cal.csv',
'certifiates', 'copy.txt', 'cse1.txt', 'DCIM', 'Download', 'ds.py', 'ds1.py', 'ds2.txt',
'eee.txt', 'exp2.py', 'filedemo.c', 'filedemo.exe', 'filedemo.o', 'first.py', 'first.txt',
'fwdresearchmethodologynotes.zip', 'Game', 'hello.txt', 'JAVA PROGRAMMING',
'm.c', 'm.exe', 'm.o', 'Machine Learning', 'MarriagePhotos', 'merge.c', 'merge.exe',
'merge.o', 'Meterials', 'Microsoft Office Enterprise 2010 Corporate Final (full
activated)', 'ML', 'myfile.txt', 'myfile1.txt', 'new.csv', 'new.py', 'new.txt', 'old',
'package', 'Packages', 'python', 'r.py', 'R20-python', 'Raj', 'raj.bin', 'raj.txt']
```

To remove a directory:

- To remove or delete a directory path in Python, rmdir( ) is used which is defined in os module.
- rmdir( ) works only when the directory we want to delete is empty, else it raises an OS error.
- So here are the ways to remove or delete empty and non-empty directory paths.

```
>> import os
>>> os.chdir('F:/Raj')
>>> os.mkdir('cse')
>>> os.listdir()
['cse']
>>> os.rmdir('cse')
```

To remove multiple directories in the path:

```
>>> import os
>>> os.chdir('F:/')
>>> os.removedirs('Raj/A')
```

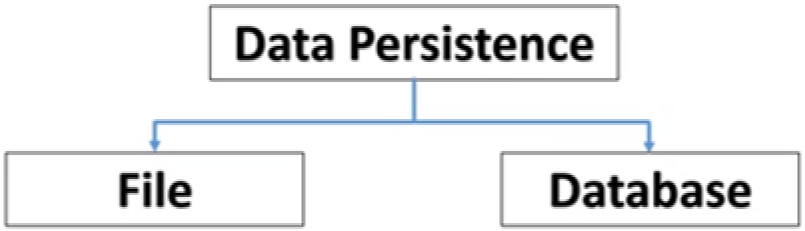
Check if Given Path is File or Directory

- To check if the path you have is a file or directory, import os module and use isfile() method to check if it is a file, and isdir() method to check if it is a directory.

```
>>> import os
>>> os.chdir('F:/')
>>> os.listdir()
['$RECYCLE.BIN', 'abcd.txt', 'add.txt', 'Applicant Details-Cloud.doc', 'c.py',
'cal.csv', 'exp2.py', 'filedemo.c', 'filedemo.exe', 'filedemo.o', 'first.py', 'first.txt',
'Raj']
>>> os.path.isfile('add.txt')
True
>>> os.path.isdir('Raj')
True
```

Persistent Storage Modules

- The word 'persistence' means "the continuance of an effect after its cause is removed".
- The term data persistence means it continues to exist even after the application has ended. Thus, data stored in a non-volatile storage medium such as, a disk file is persistent data storage.
- Data Persistence is the concept of storing data in a persistent form.
- It means that the data should be permanently stored on disk for further manipulation.
- There are two types of system used for data persistence they are



- There are two aspects to preserving data for long-term use: converting the data back and forth between the object in-memory and the storage format, and working with the storage of the converted data.
- The standard library includes a variety of modules that handle both aspects in different situations.

Serialization:

Serialization in Python is a mechanism of translating data structures or object state into a format that can be stored or transmitted and reconstructed later.

De-serialization:

The reverse operation of serialization is called de-serialization

- The type of manual conversion, of an object to string or byte format (and vice versa) is very cumbersome and tedious. It is possible to store the state of a Python object in the form of byte stream directly to a file, or memory stream and retrieve to its original state. This process is called **serialization** and **de-serialization**.
- Python's built in library contains various modules for serialization and de-serialization process. They are as follows.

S.No.	Name of the Module	Description
1	**pickle**	Python specific serialization library
2	**marshal**	Library used internally for serialization
3	**shelve**	Pythonic object persistence
4	**csv**	library for storage and retrieval of Python data to CSV format
5	**json**	Library for serialization to universal JSON format

Eg: Writing data to binary file without pickle module.

```
f=open('bin.bin','wb')
num=[10,20,30,40,50]
arr=bytearray(num)
f.write(arr)
f.close()
f=open('bin.bin','rb')
num=list(f.read())
print(num)
f.close()
```

OutPut:

```
[10,20,30,40,50]
```

- The problem with above program is the binary file requires bytes object only for that we have convert to bytes object only.
- To provide solution for this we have use any above modules

Pickle Module

- Pickling is the process whereby a python object is converted into byte stream.
- Unpickling is the reverse of this whereby a byte stream is converted back into an object.
- We can implement pickling and unpickling by using pickle module of Python.
- pickle module contains dump() function to perform pickling.
- **Syntax:pickle.dump(object,file)**
- pickle module contains load() function to perform unpickling
- **Syntax:obj=pickle.load(file)**

Eg:

```
import pickle
dict={1:"cse",2:"ds"}
f=open('bin.bin','wb')
pickle.dump(dict,f)
f.close()
f=open('bin.bin','rb')
s=pickle.load(f)
print(s)
f.close()
```

OutPut:

```
{1: 'cse', 2: 'ds'}
```

marshal Module

- The marshal module is used to serialize data—that is, convert data to and from character strings, so that they can be stored on file.

- The marshal module uses a simple self-describing data format. For each data item, the marshalled string contains a type code, followed by one or more type-specific fields. Integers are stored in little-endian order, strings are stored as length fields followed by the strings' contents (which can include null bytes), tuples are stored as length fields followed by the objects that make up each tuple, etc.
- Just as pickle module, marshal module also defined load() and dump() functions for reading and writing marshalled objects from / to file.

marshal.dump(value, file[, version]) :
This function is used to write the supported type value on the open writeable binary file. A ValueError exception is raised if the value has an unsupported type.

marshal.load(file) :
This function reads one value from the open readable binary file and returns it. EOF Error, ValueError or TypeError is raised if no value is read.

Example:

```
import marshal
dict={1:"cse",2:"ds"}
f=open('bin.bin','wb')
marshal.dump(dict,f)
f.close()
f=open('bin.bin','rb')
s=marshal.load(f)
print(s)
f.close()
```

OutPut:

```
{1: 'cse', 2: 'ds'}
```

Command-line Arguments

- There are many different ways in which a program can accept inputs from the user. The common way in Python Command-line Arguments is the input() method.
- Another way to pass input to the program is Command-line arguments. Almost every modern programming language support command line arguments.
- In a similar fashion, python does support command line arguments. It's a very important feature as it allows for dynamic inputs from the user.
- In a command-line argument, the input is given to the program through command prompt rather than python script like input() method.
- The Argument which are passing at the time of execution are called **Command Line Arguments.**
- Python supports different modules to handle command-line arguments. one of the popular one of them is **sys module.**

sys module:

- This is the basic and oldest method to handle command-line arguments in python. It has a quite similar approach as the C library argc/argv to access the arguments.
- sys module implements the command line arguments through list structure named sys.argv argv is the internal list structure which holds the arguments passed in command prompt
- argv is not Array it is a List. It is available sys Module.
- argv à list to handle dynamic inputs from the user
  - argv[0] à python filename
  - argv[1] àargument 1
  - argv[2] à argument 2
  - argv[3] à argument 3 and so on.
- Steps to create command line arguments program:
  1. Write a python program
  2. Save the python program as <program name>.py extension
  3. Open a command prompt and change the directory to the python program path
  4. Use the below command to execute the program
  5. py < python file.py > < arg1 > < arg2 > < arg3 >
  6. **Example:** py demo.py 10 20 30 40 50
- The first item in argv list i.e argv[0] is the python file name à in this case demo.py
- argv[1] is the first argument à 10
- argv[2] is the second argument à 20
- argv[3] is the third argument à 30 and so on
- By default, the type of argv is "String" so we have to typecast as per our requirement.

Example1:

```
import sys
print(type(sys.argv))
```

Output:

```
D:\>py c.py
<class 'list'>
```

Example2:

```
from sys import argv
print('The Number of Command Line Arguments:', len(argv))
print('The List of Command Line Arguments:', argv)
print('Command Line Arguments one by one:')
```

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

OutPut:

```
D:\>py c.py Raj cse ds 10
The Number of Command Line Arguments: 5
The List of Command Line Arguments: ['c.py', 'Raj', 'cse', 'ds', '10']
Command Line Arguments one by one:
c.py
Raj
cse
ds
10
```

Example3:Add two values using command line

```
from sys import argv
a=int(argv[1])
b=int(argv[2])
sum=a+b
print("The Sum:",sum)
```

OutPut:

```
D:\>py c.py 1 2
The Sum: 3
```

Example2:Sum of elements

```
from sys import argv
sum=0
args=argv[1:]
for x in args :
   n=int(x)
   sum=sum+n
print("The Sum:",sum)
```

OutPut:

```
D:\>py c.py 1 2 3 4 5
The Sum: 15
```

Exception Handling in Python

Generally any programming language supports two types of errors,

1. Syntax errors
2. Runtime errors

Syntax errors:

- The errors which occur because of invalid syntax are called **syntax errors**.
- Programmer is responsible to correct these syntax errors. Once all syntax errors are corrected then only program execution will be started.

Eg 1:

```
a=10
if a==10
 print("Raj")
SyntaxError: invalid syntax
```

Eg 2:

```
print "Raj"
SyntaxError: Missing parentheses in call to 'print'
```

Runtime errors:

- Runtime errors are also called exceptions.
- When the program is executing, if something goes wrong because of end user input or, programming logic or memory problems etc then we will call them runtime errors.

Exception:

An exception is nothing but an unwanted or unexpected block which disturbs the normal execution flow of program.

- An Exception is a run time error that happens during the execution of program.
- An exception is an error that happens during the execution of a program.
- Python raises an exception whenever it tries to execute invalid code.
- Error handling is generally resolved by saving the state of execution at the moment the error occurred and interrupting the normal flow of the program to execute a special function or piece of code, which is known as the exception handler.
- Depending on the kind of error ("division by zero", "file open error" and so on) which had occurred, the error handler can "fix" the problem and the program can be continued afterwards with the previously saved data.

Eg:

1. print(2/0) ==>ZeroDivisionError: division by zero
2. print(2/"ten") ==>TypeError: unsupported operand type(s) for /: 'int' and 'str'

```
a=int(input("Enter Number:"))
 print(a)
  D:\>py test.py
2
 Enter Number:ten
 ValueError: invalid literal for int() with base 10: 'ten’
```

Types of Exceptions:

Exceptions are divided into two types they are,

1. System defined exceptions
2. User defined exceptions

System defined exceptions:

- These exceptions are defined by system so these are called **system defined or pre-defined exceptions.**
- Every exception in Python is an object. For every exception type the corresponding classes are available.
- Whevever an exception occurs PVM will create the corresponding exception object and will check for handling code. If handling code is not available then Python interpreter terminates the program abnormally and prints corresponding exception information to the console.
- The rest of the program won't be executed
- Some of system defined exceptions are as follows,

S. No	Name of the Built□in Exception	Explanation
1	ZeroDivisionError	It is raised when the denominator in a division operation is zero
2	NameError	It is raised when a local or global variable name is not defined
3	IndexError	It is raised when the index or subscript in a sequence is out of range.
4	TypeError	It is raised when an operator is supplied with a value of incorrect data type.
5	ValueError	It is raised when a built-in method or operation receives an argument that has the right data type but mismatched or inappropriate values.
6	KeyError	**KeyError** exception is what is raised when you try to access a key that isn't in a dictionary (dict).
7	FileNotFoundError	The error **FileNotFoundError** occurs because you either don't know where a file actually is on your computer. Or, even if you do, you don't know how to tell your **Python** program where it is.
8	ModuleNotFoundError	A **ModuleNotFoundError** is raised when Python cannot successfully import a module.

1. **ZeroDivisionError:**

```
>>> a=10
>>> b=0
>>> print(a/b)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    print(a/b)
ZeroDivisionError: division by zero
```

2. NameError:

```
>>> print("a=",a)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print("a=",a)
NameError: name 'a' is not defined
```

3. IndexError:

```
>>> name="MREC"
>>> print(name[10])
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    print(name[10])
IndexError: string index out of range
```

4. ValueError:

```
>>> a=int(input("Enter a value:"))
Enter a value:Raj
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    a=int(input("Enter a value:"))
ValueError: invalid literal for int() with base 10: 'Raj'
```

5. TypeError:

```
>>> a=10
>>> b="raj"
>>> print(a/b)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    print(a/b)
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

6. **KeyError:**

```
>>> D={1:'MREC',2:'CSE',3:'DS',4:'RAJ'}
>>> print(D[1])
MREC
>>> print(D[5])
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    print(D[5])
KeyError: 5
```

7. **FileNotFoundError:**

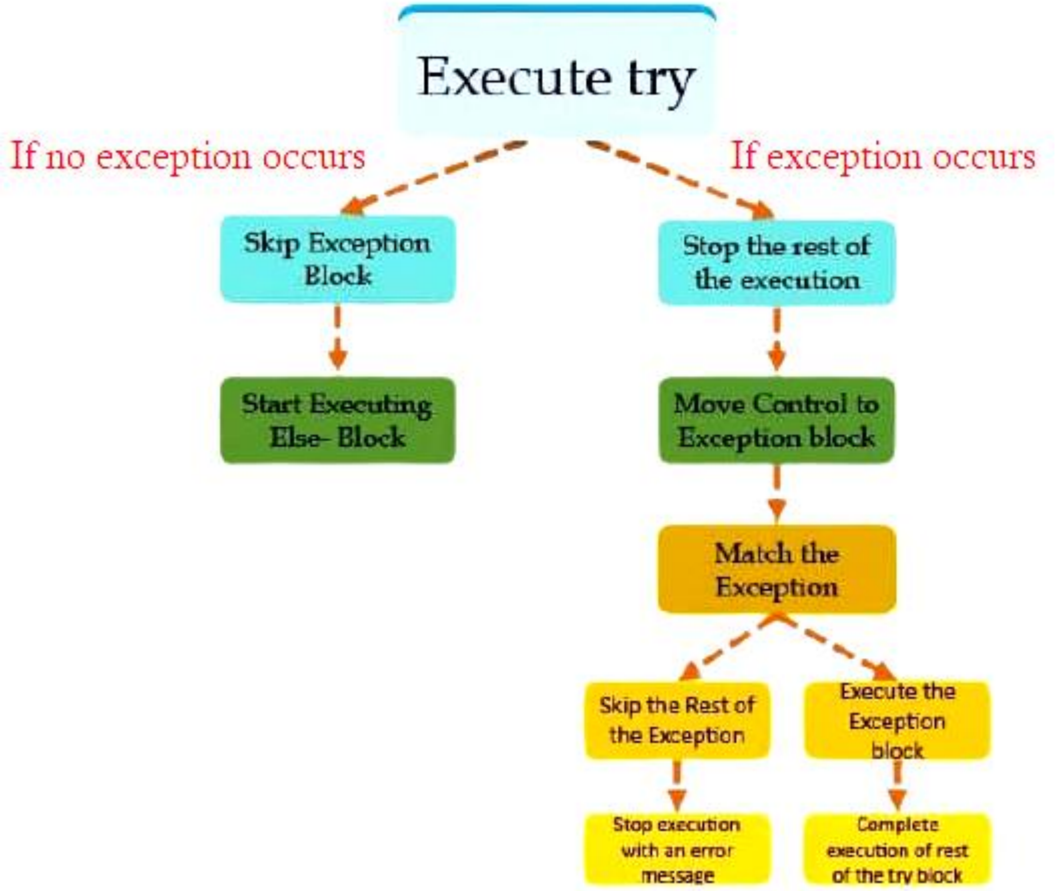
```
>>> f=open('Raj.txt','r')
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    f=open('Raj.txt','r')
FileNotFoundError: [Errno 2] No such file or directory: 'Raj.txt'
```

8. **ModuleNotFoundError:**

```
>>> import cse_ds
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    import cse_ds
ModuleNotFoundError: No module named 'cse_ds'
```

Detecting and Handling Exceptions or Exception Handing in Python

- Exception handling is a concept used in Python to handle the exceptions that occur during the execution of any program. Exceptions are unexpected errors that can occur during code execution.
- Exception handling does not mean repairing exception; we have to define an alternative way to continue rest of the program normally.
- It is highly recommended to handle exceptions. The main objective of exception handling is Graceful Termination of the program.
- Exception can be handled in two ways They are
  1. **Default Exception Handling**
  2. **Customized Exception Handling**
- The flowchart describes the exception handling process.


Execute try
If no exception occurs
If exception occurs
Skip Exception Block
Start Executing Else- Block
Stop the rest of the execution
Move Control to Exception block
Match the Exception
Skip the Rest of the Exception
Execute the Exception block
Stop execution with an error message
Complete execution of rest of the try block

Default Exception Handling

- Every exception in Python is an object. For every exception type the corresponding classes are available.
- Whenever an exception occurs PVM will create the corresponding exception object and will check for handling code.
- If handling code is not available then Python interpreter terminates the program abnormally and prints corresponding exception information to the console.
- The rest of the program won't be executed. This entire process we call it as **Default Exception Handling**
- If an exception raised inside any method then the method is responsible to create Exception object with the following information.
  - ✓ Name of the exception.
  - ✓ Description of the exception.

- ✓ Location of the exception.

- After creating that Exception object the method handovers that object to the PVM.
- PVM checks whether the method contains any exception handling code or not. If method won't contain any handling code then PVM terminates that method abnormally.
- PVM identifies the caller method and checks whether the caller method contain any handling code or not. If the caller method also does not contain handling code then PVM terminates that caller also abnormally
- Then PVM handovers the responsibility of exception handling to the default exception handler.
- Default exception handler just print exception information to the console in the following formats and terminates the program abnormally.
- Name of exception: description
- Location of exception

Example:

```
print("Start:")
print("Default Exception Handling:")
print(15/0)
print("No Exception Block:")
print("Stop")
```

OutPut:

```
Start:
Default Exception Handling:
Traceback (most recent call last):
  File "C:/Users/rajas/AppData/Local/Programs/Python/Python39/test.py", line 3, in <module>
    print(15/0)
ZeroDivisionError: division by zero
```

Customized Exception Handling

- It is highly recommended to handle exceptions.
- The Exceptions can be handled with the help of the following keywords or clauses in python.

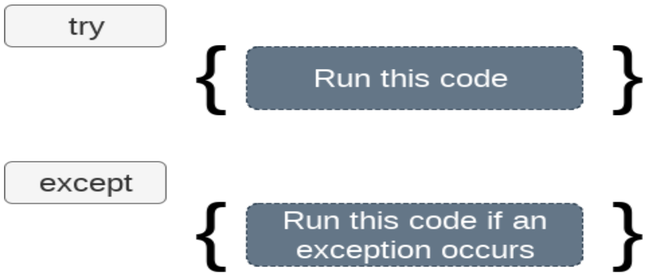
S. No	Name of the Exception Type Keyword	Explanation
1.	try	It will run the code block in which you expect an error to occur.
2.	except	Define the type of exception you expect in the try block

3.	else	If there no exception, then this block of code will be executed
4.	finally	Irrespective of whether there is an exception or not, this block of code will always be executed.
5.	raise	An exception can be raised forcefully by using the **raise** clause in Python.

- The code which may raise exception is called risky code and we have to take risky code inside try block. The corresponding handling code we have to take inside except block.
- We can handle the Exception with following ways.

1. The try-expect statement

- If the Python program contains suspicious or risky code that may throw the exception, we must place that code in the try block.
- The try block must be followed with the except statement, which contains a block of code that will be executed if there is some exception in the try block.
- Within the try block if anywhere exception raised then rest of the try block wont be executed even though we handled that exception. Hence we have to take only risky code inside try block and length of the try block should be as less as possible.
- If any statement which is not part of try block raises an exception then it is always abnormal termination.



Syntax:

```
try :
   #statements in try block
except :
   #executed when error in try block
```

Example: Without Specific error type:

```
print("Start:")
print("Exception Handling without Specific Error Type:")
try:
   print(15/0)
except:
    print("Error occured")
print("Stop")
```

OutPut:

```
Start:
Exception Handling without Specific Error Type:
Error occured
Stop
```

Example: Catch Specific Error Type

```
print("Start:")
print("Exception Handling with Specific Error Type:")
try:
    print(15/0)
except ZeroDivisionError:
    print("we can't divide the value with zero")
print("Stop")
```

OutPut:

```
Start:
Exception Handling with Specific Error Type:
we can't divide the value with zero
Stop
```

try with multiple except blocks:

- The way of handling exception is varied from exception to exception. Hence for every exception type a separate except block we have to provide. i.e try with multiple except blocks is possible and recommended to use.
- As we know, a single try block may have multiple except blocks. The following example uses two except blocks to process two different exception types:

Example:

```
print("Start:")
print("Exception Handling with Specific Error Type:")
try:
    print(15/0)
except TypeError:
    print('Unsupported operation')
except ZeroDivisionError:
    print("we can't divide the value with zero")
print("Stop")
```

OutPut:

```
Start:
Exception Handling with Specific Error Type:
we can't divide the value with zero
Stop
```

Default except block:

- We can use default except block to handle any type of exceptions.
- In default except block generally we can print normal error messages.
- If try with multiple except blocks available then default except block should be last, otherwise we will get Syntax Error.

Syntax:

```
except:
        statements
```

Eg:

```
print("Start:")
print("Default except block:")
try:
   x=int(input("Enter First Number: "))
   y=int(input("Enter Second Number: "))
   print(x/y)
except ZeroDivisionError:
   print("ZeroDivisionError:Can't divide with zero")
except:
   print("Default Except:Plz provide valid input only")
print("Stop")
```

OutPut:

```
Start:
Default except block:
Enter First Number: 5
Enter Second Number: a
Default Except:Plz provide valid input only
Stop
```

except statement using with exception variable:

- We can use the exception variable with the except statement. It is used by using the **as** keyword. this object will return the cause of the exception. Consider the following example:

```
print("Start:")
try:
   x=int(input("Enter First Number: "))
```

```
    y=int(input("Enter Second Number: "))
    print(x/y)
except Exception as e:
    print("ZeroDivisionError:Can't divide with zero")
    print(e)
print("Stop")
```

OutPut:

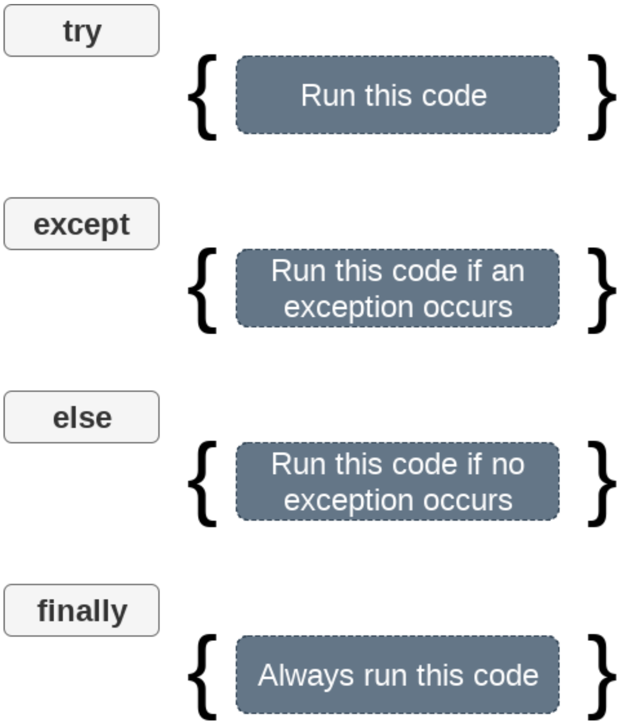
```
Start:
Enter First Number: 5
Enter Second Number: 0
ZeroDivisionError:Can't divide with zero
division by zero
Stop
```

2.else and finally:

- In Python, keywords are else and finally can also be used along with the try and except clauses.
- In python, you can also use else clause on the **try-except block** which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.

Syntax:

```
try:
    #statements in try block
except:
    #executed when error in try block
else:
    #executed if try block is error-free
finally:
    #executed irrespective of exception occured or not
```


try
Run this code
except
Run this code if an exception occurs
else
Run this code if no exception occurs
finally
Always run this code

- The finally block consists of statements which should be processed regardless of an exception occurring in the try block or not. As a consequence, the error-free try block skips the except clause and enters the finally block before going on to execute the rest of the code.
- If, however, there's an exception in the try block, the appropriate except block will be processed, and the statements in the finally block will be processed before proceeding to the rest of the code.
- The example below accepts two numbers from the user and performs their division. It demonstrates the uses of else and finally blocks.

```
print("Start:")
try:
   print('try block')
   x=int(input('Enter a number: '))
   y=int(input('Enter another number: '))
   z=x/y
except ZeroDivisionError:
   print("except ZeroDivisionError block")
   print("Division by 0 not accepted")
else:
   print("else block")
   print("Division = ", z)
finally:
   print("finally block")
   x=0
   y=0
print ("Out of try, except, else and finally blocks." )
print("Stop")
```

OutPut:

```
Start:
try block
Enter a number: 5
Enter another number: 0
except ZeroDivisionError block
Division by 0 not accepted
finally block
Out of try, except, else and finally blocks.
Stop
```

3.Raise an Exception

- An exception can be raised forcefully by using the raise clause in Python. It is useful in in that scenario where we need to raise an exception to stop the execution of the program.
- **Syntax : raise Exception_class,<value>**
- To raise an exception, the raise statement is used. The exception class name follows it.An exception can be provided with a value that can be given in the parenthesis.
- To access the value "as" keyword is used. "e" is used as a reference variable which stores the value of the exception.
- We can pass the value to an exception to specify the exception type

Example 1:

```
print("Start:")
try:
   x=int(input('Enter a number upto 100: '))
   if x > 100:
      raise ValueError(x)
except ValueError:
   print(x, "is out of allowed range")
else:
   print(x, "is within the allowed range")
print("Stop")
```

OutPut:6

```
Start:
Enter a number upto 100: 200
```

```
200 is out of allowed range
Stop
```

Example 2 Raise the exception with user defined message

```
print("Start:")
try:
   x=int(input('Enter a positive integer:'))
   if x <0:
      raise ValueError("You entered negative number")
except ValueError as e:
   print(e)
print("Stop")
```

Output:

```
Start:
Enter a positive integer:-2
You entered negative number
Stop
```

User defined exceptions

- Some time we have to define and raise exceptions explicitly to indicate that something goes wrong, such type of exceptions are called User Defined Exceptions or Customized Exceptions.
- Programmer is responsible to define these exceptions and Python not having any idea about these. Hence we have to raise explicitly based on our requirement by using "raise" Keyword.
- steps to create user defined exceptions

Step 1: Create User Defined Exception Class

- Write a new class for custom exception and inherit it from an in-build Exception class.
- Define function __init__() to initialize the object of the new class.
- You can add as many instance variables as you want, to support your exception. For simplicity, we are creating one instance variable called message.

```
class YourException(Exception):
  def __init__(self, message):
    self.message = message
```

You have created a simple user-defined exception class.

self :

- self represents the instance of the class. By using the "self" keyword we can access the attributes and methods of the class in python.

__init__ :

- "__init__" is a reseved method in python classes. It is known as a constructor in object oriented concepts. This method called when an object is created from the class and it allow the class to initialize the attributes of a class.

Step 2: Raising Exception

- Now you can write a try-except block to catch the user-defined exception in Python.
- For testing, inside the try block we are raising exception using raise keyword.
- **raise YourException("Userdefined Exceptions")**
- It creates the instance of the exception class YourException. You can pass any message to your exception class instance.

Step 3: Catching Exception

- Now you have to catch the user-defined exception using except block.

```
except YourException as err:
  print(err.message)
```

- We are catching user defined exception called YourException.

Step 4: Write a Program for User-Defined Exception in Python

```
class ChildrenException(Exception):
   def __init__(self,arg):
      self.msg=arg
class YouthException(Exception):
   def __init__(self,arg):
      self.msg=arg
class AdultException(Exception):
   def __init__(self,arg):
      self.msg=arg
class SeniorException(Exception):
   def __init__(self,arg):
      self.msg=arg
age=int(input("Enter Age:"))
if (age<18) and (age>0):
   raise ChildrenException("The Person having the age between (0-18)!!!")
elif (age<25) and (age>=19):
   raise YouthException("The Person having the age between (19-24)!!!")
```

```
elif (age<65) and (age>=25):
   raise AdultException("The Person having the age between (25-64)!!!")
elif (age>=65):
   raise SeniorException("The Person having the age between (65
above)!!!")
else:
   print("You have entered invalid age!!!")
```

Output:

```
Enter Age:35
Traceback (most recent call last):
 File
"C:/Users/rajas/AppData/Local/Programs/Python/Python39/user.py",
line 19, in <module>
   raise AdultException("The Person having the age between (25-64)!!!")
AdultException: The Person having the age between (25-64)!!!-
```

Example2:

```
class PassException(Exception):
   def __init__(self,arg):
      self.msg=arg
class FailException(Exception):
   def __init__(self,arg):
      self.msg=arg
class MarksException(Exception):
   def __init__(self,arg):
      self.msg=arg

try:
   marks=int(input("Enter the marks of a subject:"))
   if(marks<35) and (marks>=0):
      raise FailException("Fail")
   elif(marks>=35):
      raise PassException("Pass")
   else:
      raise MarksException("Marks should be positive")
except FailException as e:
   print(e)
except PassException as e:
   print(e)
except MarksException as e:
    print(e)
print("Stop")
```

Output:

```
Enter the marks of a subject:-25
Marks should be positive
Stop
```

ASSERTIONS in python

- Python assert keyword is defined as a debugging tool that tests a condition. The Assertions are mainly the assumption that asserts or state a fact confidently in the program.
- The process of identifying and fixing the bug is called debugging.
- Very common way of debugging is to use print() statement. But the problem with the print() statement is after fixing the bug,compulsory we have to delete the extra added print() statments,otherwise these will be executed at runtime which creates performance problems and disturbs console output.
- To overcome this problem we should go for assert statement. The main advantage of assert statement over print() statement is after fixing bug we are not required to delete assert statements. Based on our requirement we can enable or disable assert statements.
- Hence the main purpose of assertions is to perform debugging. Usully we can perform debugging either in development or in test environments but not in production environment. Hence assertions concept is applicable only for dev and test environments but not for production environment.

Types of assert statements:

There are 2 types of assert statements

1. Simple Version
2. Augmented Version

1. Simple Version:

Syntax: **assert conditional_expression**

2. Augmented Version:

Syntax: **assert conditional_expression,message**

- conditional_expression will be evaluated and if it is true then the program will be continued.If it is false then the program will be terminated by raising AssertionError. By seeing AssertionError, programmer can analyze the code and can fix the problem.

Examples:1

```
assert True
```

```
print("Validation Passed")
```

Output:

```
Validation Passed
```

Examples:2

```
assert False
print("Validation Passed")
```

Output:

```
Traceback (most recent call last):
  File "D://assert1.py", line 1, in <module>    assert False
AssertionError
```

Examples:3

```
assert False ,"Validation Failed"
print("Validation Passed")
```

Output:

```
Traceback (most recent call last):
  File "D://assert1.py",, line 1, in <module>
    assert False ,"Validation Failed"
AssertionError: Validation Failed
```

Example: 4

```
assert "Python" in "Python Programming"
print("Validation Passed")
```

Output:

```
Validation Passed
```

Example:5

```
assert "Python" in "python Programming","Validation Failed"
print("Validation Passed")
```

Output:

```
Traceback (most recent call last):
  File "D: /assert1.py", line 1, in <module>
    assert "Python" in "python Programming","Validation Failed"
AssertionError: Validation Failed
```

Example:6

```
str1="Raj"
str2="Raj"
assert str1==str2,"Strings are not matched"
print("String are matched")
```

Output:

```
String are matched
```

Example:7

```
str1="Raj"
str2="Raj"
assert str1==str2,"Strings are not matched"
print("String are matched")
```

Output:

```
Traceback (most recent call last):
  File "D:/assert1.py", line 3, in <module>
    assert str1!=str2,"Strings are not matched"
AssertionError: Strings are not matched
```

Example:8

```
assert "Raj" in ["MREC","CSE","DS","Raj"],"Validation Failed"
print("Validation passed")
```

Output:

```
Validation passed
```

Example:9

```
assert "raj" in ["MREC","CSE","DS","Raj"],"Validation Failed"
print("Validation passed")
```

Output:

```
Traceback (most recent call last):
  File "D:/assert1.py", line 1, in <module>
    assert "raj" in ["MREC","CSE","DS"],"Validation Failed"
AssertionError: Validation Failed
```

Example:10

```
import math
assert math.factorial(5)==120,"Validation Failed"
print("Validation passed")
```

Output:

```
Validation passed
```

Example:11

```
import math
assert math.factorial(5)!=120,"Validation Failed"
print("Validation passed")
```

Output:

```
Traceback (most recent call last):
  File "D:/assert1.py", line 2, in <module>
    assert math.factorial(5)!=120,"Validation Failed"
AssertionError: Validation Failed
```

Python Programming

MODULE – III(Part-1)

Agenda:

- **Regular Expression (RE):** Introduction,
- Special Symbols and Characters,
- REs and Python.

Regular Expression (RE):

- A regular expression is a series of characters used to search or find a pattern in a string.
- In other words, a regular expression is a special sequence of characters that form a pattern.
- The regular expressions are used to perform a variety of operations like searching a substring in a string, replacing a string with another, splitting a string, etc.
- The Python programming language provides a built-in module re to work with regular expressions.
- There is a built-in module that gives us a variety of built-in methods to work with regular expressions. In Python, the regular expression is known as RegEx in short form.

Special Symbols and Characters

Creating Regular Expression:

The regular expressions are created using the following.

- Metacharacters
- Special Sequences
- Sets

Metacharacters

- Metacharacters are the characters with special meaning in a regular expression. The following table provides a list of metacharacters with their meaning.

Metacharacters	**Meaning**
[ ]	Square brackets specifies a set of characters you wish to match.
\	Backlash \\ is used to escape various characters including all metacharacters.
.	A period matches any single character (except newline '\n')
^	The caret symbol ^ is used to check if a string starts with a certain character.
$	The dollar symbol $ is used to check if a string ends with a certain character.
*****	The star symbol * matches zero or more occurrences of the pattern left to it.
+	The plus symbol + matches one or more occurrences of the pattern left to it.
{ }	Consider this code: {n,m}. This means at least n, and at most m repetitions of the pattern left to it.
|	Vertical bar | is used for alternation (or operator).
()	Parentheses () is used to group sub-patterns.
?	The question mark symbol ? matches zero or one occurrence of the pattern left to it.

[] - Square brackets

- **Square brackets specifies a set of characters you wish to match.**

. - Period

Expression	String	Matched?
[abcd]	Cse	1 match
	cse ds	2 matches
	MREC	No match
	cse ds mrec	3 matches

- **A period matches any single character (except newline '\n').**

Expression	String	Matched?
..	C	No match
	Ds	1 match
	Cse	1 match
	Cseds	2 matches
	cse ds	3 matches(including space)

^ - Caret

- **The caret symbol ^ is used to check if a string starts with a certain character.**

Expression	String	Matched?
^c	C	1 match
	Cse	1 match
	Ds	No match
^ds	cse ds	1 match
	Cse	No match

$ - Dollar

- **The dollar symbol $ is used to check if a string ends with a certain character.**

Expression	String	Matched?
c$	C	1 match
	Mrec	1 match
	Ds	No match

* - Star

- **The star symbol * matches zero or more occurrences of the pattern left to it.**

Expression	String	Matched?
abc*	ab	1 match
	abc	1 match
	abcabc	2 match
	cse	No match
	Dsabc	1 match

+ - Plus

- **The plus symbol + matches one or more occurrences of the pattern left to it.**

Expression	String	Matched?
ab+c	Ac	No match (no a character)
	Abc	1 match
	Abbbc	1 match
	Cse	No match (a is not followed by n)
	Dsabc	1 match

? - Question Mark

- **The question mark symbol ? matches zero or one occurrence of the pattern left to it.**

Expression	String	Matched?
ab?c	ac	1 match
	abc	1 match
	abbbc	No match
	abrc	No match
	cseabc	1 match

{} - Braces

- **Consider this code: {n,m}. This means at least n, and at most m repetitions of the pattern left to it.**

Expression	String	Matched?
a{2,3}	abc dat	No match
	abc data	1 match (at daat)
	aabc daaat	2 matches (at aabc and daaat)
	aabc daaaat	2 matches (at aabc and daaaat)

| - Alternation

- **Vertical bar | is used for alternation (or operator).**

Expression	String	Matched?
a|b	Cde	No match
	Ade	1 match (match at ade)
	acdbea	3 matches (at acdbea)

() - Group

- **Parentheses () is used to group sub-patterns. For example, (a|b|c)xz match any string that matches either a or b or c followed by xz**

Expression	String	Matched?
(a|b|c)xz	ab xz	No match
	Abxz	1 match (match at abxz)
	axz cabxz	2 matches (at axzbc cabxz)

\ - Backslash

- **Backlash \ is used to escape various characters including all metacharacters. For example,**
- **\$a match if a string contains $ followed by a. Here, $ is not interpreted by a RegEx engine in a special way.**
- **If you are unsure if a character has special meaning or not, you can put \ in front of it. This makes sure the character is not treated in a special way.**

```
>>> print(re.findall(r'$a','dscse$a'))
[]
>>> print(re.findall(r'\$a','dscse$a'))
['$a']
```

Special Sequences

- A special sequence is a character prefixed with \, and it has a special meaning. The following table gives a list of special sequences in Python with their meaning.

Special Sequences	Meaning
\A	Matches if the specified characters are at the start of a string.
\b	Matches if the specified characters are at the beginning or end of a word.
\B	Opposite of \b. Matches if the specified characters are not at the beginning or end of a word.
\d	Matches any decimal digit. Equivalent to [0-9]
\D	Matches any non-decimal digit. Equivalent to [^0-9]
\s	Matches where a string contains any whitespace character. Equivalent to [\t\n\r\f\v].
\S	Matches where a string contains any non-whitespace character. Equivalent to [^ \t\n\r\f\v].
\w	Matches any alphanumeric character (digits and alphabets). Equivalent to [a-zA-Z0-9_]. By the way, underscore _ is also considered an alphanumeric character.
\W	Matches any non-alphanumeric character. Equivalent to [^a-zA-Z0-9_]
\Z	Matches if the specified characters are at the end of a string.

- **\A - Matches if the specified characters are at the start of a string.**

Expression	String	Matched?
\Acse	cse ds	Match
	ds cse	No match

- **\b - Matches if the specified characters are at the beginning or end of a word.**

Expression	String	Matched?
\bcse	Csemrec	Match
	ds csemrec	Match
	Dscsemrec	No match
ds\b	cse ds	Match
	cse ds mrec	Match
	cse dsmrec	No match

- **\B - Opposite of \b. Matches if the specified characters are not at the beginning or end of a word.**

Expression	String	Matched?
\Bcse	Csemrec	No match
	ds csemrec	No Match
	Dscsemrec	Match
ds\B	cse ds	No match
	cse ds mrec	No match
	cse dsmrec	Match

- **\d - Matches any decimal digit. Equivalent to [0-9]**

Expression	String	Matched?
\d	12mrec3	3 matches (at 12mrec3)
	cse ds	No match

- **\D - Matches any non-decimal digit. Equivalent to [^0-9]**

Expression	String	Matched?
\D	cseds123	5 matches (at cseds123)
	1345	No match

- **\s - Matches where a string contains any whitespace character. Equivalent to [\t\n\r\f\v].**

Expression	String	Matched?
\s	cse\tds\nmrec	2 match
	Csedsmrec	No match

- **\S - Matches where a string contains any non-whitespace character. Equivalent to [^ \t\n\r\f\v].**

Expression	String	Matched?
\S	a b	2 matches (at a b)
		No match

- **\w - Matches any alphanumeric character (digits and alphabets). Equivalent to [a-zA-Z0-9_]. By the way, underscore _ is also considered an alphanumeric character.**

Expression	String	Matched?
\w	12&": ;c	3 matches (at 12&": ;c)
	%"> !	No match

- **\W - Matches any non-alphanumeric character. Equivalent to [^a-zA-Z0-9_]**

Expression	String	Matched?
\W	1cse@ds	1 match (at 1cse@ds)
	Cseds	No match

- **\Z - Matches if the specified characters are at the end of a string.**

Expression	String	Matched?
ds\Z	cse ds	1 match
	cse ds mrec	No match
	ds cse.	No match

Sets

- A set is a set character enclosed in [ ], and it has a special meaning. The following table gives a list of sets with their meaning.

Set	**Meaning**
[aeiou]	Matches with one of the specified characters are present
[d-s]	Matches with any lower case character from d to s
[^aeiou]	Matches with any character except the specified
[1234]	Matches with any of the specified digit
[3-8]	Matches with any digit from 3 to 8
[a-zA-Z]	Matches with any alphabet, lower or UPPER

- **[aeiou]   Matches with one of the specified characters are present**

```
>>> print(re.findall(r'[aeiou]','cse and ds dept'))
['e', 'a', 'e']
```

- **[d-s]   Matches with any lower case character from d to s**

```
>>> print(re.findall(r'[a-h]','cse and ds dept'))
['c', 'e', 'a', 'd', 'd', 'd', 'e']
```

- **[^aeiou]   Matches with any character except the specified**

```
>>> print(re.findall(r'[^aeiou]','cse and ds dept'))
['c', 's', ' ', 'n', 'd', ' ', 'd', 's', ' ', 'd', 'p', 't']
```

- **[1234] Matches with any of the specified digit**

```
>>> print(re.findall(r'[12345]','cse-1 cse-2 cse-3 cse-4 ds'))
['1', '2', '3', '4']
```

- **[a-zA-Z]	Matches with any alphabet, lower or UPPER**

```
>>> print(re.findall(r'[a-zA-Z]','This is Cse and Ds Dept'))
['T', 'h', 'i', 's', 'i', 's', 'C', 's', 'e', 'a', 'n', 'd', 'D', 's', 'D', 'e', 'p', 't']
```

REs and Python

Built-in methods of re module

The re module provides the following methods to work with regular expressions.

1. match()
2. search()
3. findall()
4. finditer()
5. sub()
6. split()
7. compile()

1. match() in Python:

- We can use match function to check the given pattern at beginning of target string. If the match is available then we will get Match object, otherwise we will get None.
- The re.match() method will start matching a regex pattern from the very first character of the text, and if the match found, it will return a re.Match object. Later we can use the re.Match object to extract the matching string.

Syntax of re.match(): re.match(pattern, string, flags=0)

- The regular expression pattern and target string are the mandatory arguments, and flags are optional.
- **pattern:** The regular expression pattern we want to match at the beginning of the target string. Since we are not defining and compiling this pattern beforehand (like the compile method). The practice is to write the actual pattern using a raw string.
- **string:** The second argument is the variable pointing to the target string (In which we want to look for occurrences of the pattern).
- **flags:** Finally, the third argument is optional and it refers to regex flags by default no flags are applied.

Eg:

```
>>> str='This is MREC'
>>> print(re.match(r'\w{4}',str))
<re.Match object; span=(0, 4), match='This'>
```

- This re.Match object contains the following items.

- A span attribute that shows the locations at which the match starts and ends. i.e., is the tuple object contains the start and end index of a successful match.Save this tuple and use it whenever you want to retrieve a matching string from the target string
- Second, A match attribute contains an actual match value that we can retrieve using a group() method.
- The Match object has several methods and attributes to get the information about the matching string. Let's see those.

Method	**Description**
group()	Return the string matched by the regex
start()	Return the starting position of the match
end()	Return the ending position of the match
span()	Return a tuple containing the (start, end) positions of the match.

```
>>> res=re.match(r'\w{4}',str)
>>> res
<re.Match object; span=(0, 4), match='This'>
>>> res.group()
'This'
>>> res.start()
0
>>> res.end()
4
>>> res.span()
(0, 4)
```

search( ) in Python:

- Python regex re.search() method looks for occurrences of the regex pattern inside the entire target string and returns the corresponding Match Object instance where the match found.
- **Syntax:re.search(pattern, string, flags=0)**

```
>>> import re
>>> print(re.search('cse','cse and ds depts in MREC'))
<re.Match object; span=(0, 3), match='cse'>
>>> print(re.search('ds','cse and ds depts in MREC'))
<re.Match object; span=(8, 10), match='ds'>
```

findall( ) in Python:

- The RE module’s re.findall() method scans the regex pattern through the entire target string and returns all the matches that were found in the form of a Python list.
- **Syntax:re.findall(pattern, string, flags=0)**

```
>>> print(re.findall('abc','abc abcde bchkdhk abc'))
['abc', 'abc', 'abc']
>>> print(re.findall('cse','cse-A cse-B cse-C cse-D'))
['cse', 'cse', 'cse', 'cse']
>>> print(re.findall('ds','This is ds dept'))
['ds']
```

finditer() in python:

- The re.finditer() works exactly the same as the re.findall() method except it returns an iterator yielding match objects matching the regex pattern in a string instead of a list. It scans the string from left-to-right, and matches are returned in the iterator form. Later, we can use this iterator object to extract all matches.
- In simple words, finditer() returns an iterator over MatchObject objects.

```
import re
res=re.finditer(r'\b\w{3}\b','cse ds raj')
for match in res:
   print(match.group())
print(re.findall(r'\b\w{3}\b','cse ds raj'))
```

outPut:

```
cse
raj
['cse', 'raj']
```

sub( ) in Python:

- The sub( ) method of re object replaces the match pattern with specified text in a string.
- The syntax of sub( ) method is **sub(pattern, text, string).**
- The sub( ) method does not modify the actual string instead, it returns the modified string as a new string.

```
>>> print(re.sub('depts','DEPTS','cse and ds depts') )
>>>cse and ds DEPTS
>>> id='rajasekhar.v86@gmail.com'
>>> print(re.sub('.com','.in',id))
rajasekhar.v86@gmail.in
```

split( ) in Python

- The Pythons re module’s re.split() method split the string by the occurrences of the regex pattern, returning a list containing the resulting substrings.

- **Syntax:re.split(pattern, string, maxsplit=0, flags=0)**
- The regular expression pattern and target string are the mandatory arguments. The maxsplit, and flags are optional.
- **pattern:** the regular expression pattern used for splitting the target string.
- **string:** The variable pointing to the target string (i.e., the string we want to split).
- **maxsplit:** The number of splits you wanted to perform. If maxsplit is nonzero, at most maxsplit splits occur, and the remainder of the string is returned as the final element of the list.
- **flags:** By default, no flags are applied.

```
 print(re.split('\.','http://www.google.com/'))
['http://www', 'google', 'com/']

str=" cse and ds depts in mrec'
print(re.split(r'\s+',str))
['cse', 'and', 'ds', 'depts', 'in', 'mrec']

str='123-45-678-9'
print(re.split(r'\D',str,maxsplit=1))
['123', '45-678-9']
```

compile() in python:

- Python’s re.compile() method is used to compile a regular expression pattern provided as a string into a regex pattern object (re.Pattern).
- Later we can use this pattern object to search for a match inside different target strings using regex methods such as a re.match() or re.search().
- In simple terms, We can compile a regular expression into a regex object to look for occurrences of the same pattern inside various target strings without rewriting it.

Syntax of re.compile()

- re.compile(pattern, flags=0)
- **pattern:** regex pattern in string format, which you are trying to match inside the target string.
- **flags:** The expression’s behavior can be modified by specifying regex flag values. This is an optional parameter

```
>>> pattern=re.compile(r'\b\w{4}\b')
>>> result=patten.findall('abcd raaj')
>>> result=pattern.findall('abcd raaj')
>>> result
['abcd', 'raaj']
>>> print(re.findall(pattern,'abcd raaj'))
['abcd', 'raaj']
>>>
```

Examples

Example 1: Write a regular expression to search digit inside a string

```
import re
str="The total no of students are120"
res=re.findall(r'\d',str)
print(res)
```

output:

```
['1', '2', '0']
```

Example2: match 3-letter word anywhere in the string

```
str='MREC civil cse eee ece ds iot ai&ml cs mech'
 print(re.findall(r'\w{3}',str))
```

```
['MRE', 'civ', 'cse', 'eee', 'ece', 'iot', 'mec']
```

Example 3 : Extract all characters from the paragraph using Python Regular Expression.

```
import re
str="The total no of students are120"
print(re.findall(r'.',str))
```

```
['T', 'h', 'e', ' ', 't', 'o', 't', 'a', 'l', ' ', 'n', 'o', ' ', 'o', 'f', ' ', 's', 't', 'u', 'd', 'e', 'n', 't', 's', '
', 'a', 'r', 'e', '1', '2', '0']
```

Example 4: Extract all of the words and numbers

```
import re
str="The total no of students are120"
print(re.findall(r'\w+',str))
```

```
['The', 'total', 'no', 'of', 'students', 'are120']
```

Example 5: Extract only numbers

```
import re
str="The total no of students are120"
print(re.findall(r'\d+',str))
```

```
['120']
```

Example 6: Extract the beginning word

```
import re
```

```
str="The total no of students are120"
print(re.findall(r'^\w+',str))
```

```
['The']
```

Example 7: Extract first two characters from each word (not the numbers)

```
import re
str="The total no of students are120"
print(re.findall(r'\b[a-zA-Z].',str))
```

```
['Th', 'to', 'no', 'of', 'st', 'ar']
```

Example 8: Find out all of the words, which start with a vowel.

```
import re
str="The total no of students are120"
print(re.findall(r'\b[aeiou]\w+',str))
```

```
['of', 'are120']
```

Example 9:Extract date from the string

```
import re
str='Today date is June 09, 2021.'
pattern=r'(\w+)(\s)(\d+)([,]\s)(\d+)'
print(re.findall(pattern,str))
```

```
[('June', ' ', '09', ', ', '2021')]
```

```
import re
str='Today date is 06-09-2021'
pattern=r'(\d+)(.)(\d+)(.)(\d+)'
print(re.findall(pattern,str))
```

```
[('06', '-', '09', '-', '2021')]
```

```
import re
str='Today date is 06-09-2021'
match=re.search(r'\d{2}-\d{2}-\d{4}',str)
```

```
print(match.group())
```

```
06-09-2021
```

Example 10:Extract date from the string

```
import re
str = "Please contact us at rajasekhar.v86@gmail.com for further information."
email = re.findall(r"[a-z0-9\.\-+_]+@[a-z0-9\.\-+_]+\.[a-z]+", str)
print(email)
```

```
['rajasekhar.v86@gmail.com']
```

Example 11:Write a Python program that matches a string that has an a followed by zero or more b's.

```
import re
def match(str):
    pattern = 'ab*?'
    if re.search(pattern,str):
        return 'Found a match!'
    else:
        return('Not matched!')

print(match("ac"))
print(match("abc"))
print(match("abbc"))
print(match("abbbc"))
print(match("$12"))
```

```
Found a match!
Found a match!
Found a match!
Found a match!
Not matched!
```

Example 12:Replace maximum 2 occurrences of space, comma, or dot with a colon

```
import re
text = 'CSE DS, MREC .'
print(re.sub("[ ,.]", ":", text, 2))
```

Example 13:Develop a Python program to match a string that contains only upper and lowercase letters, numbers, and underscores.

```
import re
str = 'Raj_1254'
pattern='^[a-zA-Z0-9_]*$'
print(re.findall(pattern,str))
```

```
['Raj_1254']
```

Python Programming

MODULE – IV

Agenda:

- **Classes and Object-Oriented Programming (OOP): OOP,**
- **Classes, Class Attributes**
- **Instances,Instance Attributes**
- **Binding and Method Invocation**
- **Composition**
- **Subclassing and Derivation**
- **Inheritance**
- **Built-in Functions for Classes, Instances, and Other Objects,**
- **Types vs. Classes/Instances**
- **Customizing Classes with Special Methods,**
- **Privacy,**
- **Delegation and Wrapping**

Object Oriented Programming (OOP)

- In all the programs, we have designed our program around functions i.e. blocks of statements which manipulate the data. This is called the procedure-oriented way of programming.
- There is another way of organizing our program which is to combine data and functionality and wrap it inside something called an object. This is called the object oriented programming paradigm.
- Classes and objects are the two main aspects of object oriented programming.
- A **class** creates a new type where **objects** are instances of the class.
- Object Oriented Programming is a way of computer programming using the idea of "objects" to represents data and methods. It is also, an approach used for creating neat and reusable code instead of a redundant one.
- The program is divided into self-contained objects or several mini-programs. Every Individual object represents a different part of the application having its own logic and data to communicate within themselves.
- Now, to get a more clear picture of why we use oops instead of pop, I have listed down the differences below.

	Procedure Oriented Programming	**Object Oriented Programming**
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP, Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifies	POP does not have any access specifier.	OOP has access specifies named Public, Private, Protected.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Examples of POP are: C, VB, FORTRAN, and Pascal……	Examples of OOP are: C++, JAVA, VB.NET, C#.NET,PYTHON….

- Major principles of object-oriented programming system are given below.

  - Class
  - Object
  - Method
  - Inheritance
  - Polymorphism
  - Data Abstraction
  - Encapsulation

Class:

- A Class in Python is a logical grouping of data and functions. It gives the freedom to create data structures that contains arbitrary content and hence easily accessible.
- A class is a "blueprint" or "prototype" to define an object. Every object has its properties and methods. That means a class contains some properties and methods.
- A class is the blueprint from which the individual objects are created. Class is composed of three things: a name, attributes, and operations
- For example: if you have an employee class, then it should contain an attribute and method, i.e. an email id, name, age, salary, etc.

Syntax

```
class ClassName:
     <statement-1>
     .
     .
    <statement-N>
```

Object

- An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.
- Object is composed of three things: a name, attributes, and operations or Objects are an instance of a class. It is an entity that has state and behavior.

Syntax: **object_name = ClassName(arguments)**

To define class you need to consider following points

Step 1) In Python, classes are defined by the "Class" keyword

```
class myClass():
```

Step 2) Inside classes, you can define functions or methods that are part of this class

```
def ds(self):
  print ("DS Branch")
def cse (self,value):
    print ("CSE Branch" ,value)
```

Here we have defined ds that prints "DS Branch"

Another method we have defined is cse that prints "CSE Branch"+ value . value is the variable supplied by the calling method

Step 3) Everything in a class is indented, just like the code in the function, loop, if statement, etc. Anything not indented is not in the class

Example:

```
class myClass:
   def ds(self):
      print("DS Branch")
   def cse(self,value):
      print("CSE Branch",value)
```

"self" in Python:

- The self-argument refers to the object itself. Hence the use of the word self. So inside this method, self will refer to the specific instance of this object that's being operated on.
- Self is the name preferred by convention by Pythons to indicate the first parameter of instance methods in Python. It is part of the Python syntax to access members of objects

Step 4) To make an object of the class

```
c = myClass()
```

Step 5) To call a method in a class

```
c.ds()
c.cse(5)
```

- Notice that when we call the ds or cse, we don't have to supply the self-keyword. That's automatically handled for us by the Python runtime.
- Python runtime will pass "self" value when you call an instance method on in instance, whether you provide it deliberately or not You just have to care about the non-self arguments

Step 6) Here is the complete code

```
# Example file for working with classes
class myClass:
   def ds(self):
      print("DS Branch")
   def cse(self,value):
      print("CSE Branch",value)
c=myClass()
c.ds()
c.cse(5)
```

output: **DS Branch**

```
CSE Branch 5
```

Python Constructor:

- Python Constructor in object-oriented programming with Python is a special kind of method/function we use to initialize instance members of that class.
- The name of the constructor should be __init__(self)
- Constructor will be executed automatically at the time of object creation.
- The main purpose of constructor is to declare and initialize instance variables.
- Per object constructor will be executed only once.
- Constructor can take atleast one argument(atleast self)
- Constructor is optional and if we are not providing any constructor then python will provide default constructor

Types of Constructors:

We observe three types of Python Constructors, two of which are in our hands. Let's begin with the one that isn't.

- Default Constructor
- Non- Parameterized Constructor
- Parameterized Constructor

Default Constructor:

A constructor that Python lends us when we forget to include one. This one does absolutely nothing but instantiates the object; it is an empty constructor- without a body.

Example:

```
class defaultConstructor:
   def display(self):
      print("This is Default Constructor")
dc=defaultConstructor()
dc.display()
```

OutPut:

```
This is Default Constructor
```

Non- Parameterized Constructor:

- When we want a constructor to do something but none of that is to manipulate values, we can use a non-parameterized constructor.
- As we know that a constructor always has a name init and the name init is prefixed and suffixed with a double underscore(__). We declare a constructor using def keyword, just like methods.

Syntax:

```
def __init__(self):
   # body of the constructor
```

Example:

```
class NonParameterizedConstructor:
   def __init__(self):
      print("Non-Parameterized Constructor")
   def display(self,name):
      print("Name=",name)
npc=NonParameterizedConstructor()
npc.display('Raj')
```

OutPut:

```
         Non- Parameterized Constructor
         Name= Raj
```

Parameterized constructor:

- Constructor with parameters is known as parameterized constructor.The parameterized constructor take its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

Example:

```
class Addition:
         first = 0
         second = 0
         answer = 0
         def __init__(self, f, s):
                  self.first = f
                  self.second = s

         def display(self):
                  print("First number = " + str(self.first))
                  print("Second number = " + str(self.second))
                  print("Addition of two numbers = " + str(self.answer))

         def calculate(self):
                  self.answer = self.first + self.second
obj = Addition(1000, 2000)
obj.calculate()
obj.display()
```

OutPut:

```
First number = 1000
Second number = 2000
Addition of two numbers = 3000
```

Attributes are noting but variables we the following types are there in python.

Types of Class Variables in Python: There are three different types of variables in OOPs in python.

- Instance variables (object level variables)
- Static variables (class level variables)
- Local variables

Instance Variables in Python:

If the value of a variable is changing from object to object then such variables are called as instance variables.

```
class Student:
  def __init__(self, name, id):
     self.name=name
     self.id=id
s1=Student('Srav', 1)
s2=Student('Raj', 2)
print("Studen1 info:")
print("Name: ", s1.name)
print("Id : ", s1.id)
print("Studen2 info:")
print("Name: ",s2.name)
print("Id : ",s2.id)
```

OutPut:

```
Studen1 info:
Name:  Srav
Id :  1
Studen2 info:
Name:  Raj
Id :  2
```

Static variables in Python:

- If the value of a variable is not changing from object to object, such types of variables are called static variables or class level variables. We can access static variables either by class name or by object name. Accessing static variables with class names is highly recommended than object names.

Example:

```
class Student:
  college='MREC'
  def __init__(self, name, id):
    self.name=name
    self.id=id
s1=Student('SRAV', 1)
s2=Student('RAJ', 2)
print("Studen1 info:")
print("Name: ", s1.name)
print("Id : ", s1.id)
print("College name n : ", Student.college)
print("\n")
print("Studen2 info:")
print("Name: ",s2.name)
print("Id : ",s2.id)
print("College name : ", Student.college)
```

OutPut:

```
Studen1 info:
Name:  SRAV
Id :  1
College name n :  MREC

Studen2 info:
Name:  RAJ
Id :  2
College name :  MREC
```

Local Variables in Python:

- The variable which we declare inside of the method is called a local variable. Generally, for temporary usage we create local variables to use within the methods. The scope of these variables is limited to the method in which they are declared. They are not accessible out side of the methods.

Example:

```
class mrec:
  dept="DS"#static variable
  def display(self):
    dept="CSE" #Local Variable
    print(dept)
d=mrec()
d.display()
print(d.dept)
```

OutPut:

```
CSE
DS
```

Binding and Method Invocation:

- There are three main types of methods in Python.
    - Instance methods
    - Static methods
    - Class methods.

Instance methods:

- Instance methods are the most common type of methods in Python classes. These are so called because they can access unique data of their instance.
- And we call it as default method in python.
- If you have two objects each created from a car class, then they each may have different properties. They may have different colors, engine sizes, seats, and so on.
- Instance methods are methods which act upon the instance variables of the class. They are bound with instances or objects, that"s why called as instance methods. The first parameter for instance methods should be self variable which refers to instance. Along with the self variable it can contain other variables as well.
- Any method you create will automatically be created as an instance method, unless you tell Python otherwise.

Example:

```
class Test:
   def __init__(self, a, b):
      self.a = a
      self.b = b
   def avg(self):
      return (self.a + self.b) / 2
s1 = Test(10, 20)
print( s1.avg() )
```

OutPut:

```
15.0
```

Class Methods:

- Class methods are methods which act upon the class variables or static variables of the class. We can go for class methods when we are using only class variables (static variables) within the method.
- Class methods should be declared with @classmethod.
- Just as instance methods have 'self' as the default first variable, class method should have 'cls' as the first variable. Along with the cls variable it can contain other variables as well.
- Class methods are rarely used in python

Example:

```
class Mrec:
  Dept="CSE"
  def Dept_name(self,name):
     print("Instance method=",name)
  @classmethod
  def get_Dept(cls):
     return cls.Dept
m=Mrec()
m.Dept_name("DS")
print("Class method=",Mrec.get_Dept())
```

OutPut:

```
Instance method= DS
Class method= CSE
```

Static methods

- A static method can be called without an object for that class, using the class name directly. If you want to do something extra with a class we use static methods.
- Inside these methods we won't use any instance or class variables. No arguments like cls or self are required at the time of declaration.
- We can declare static method explicitly by using @staticmethod decorator.
- We can access static methods by using class name or object reference

Example:

```
class Demo:
  @staticmethod
  def sum(x, y):
     print(x+y)
  @staticmethod
  def multiply(x, y):
     print(x*y)
Demo.sum(2, 3)
Demo.multiply(2,4)
```

OutPut:

```
5
8
```

Example:

```
class Demo:
    x=10
    y=5
    def __init__(self,x,y):
        self.x=x
        self.y=y
    def add(self):
        print("Sum=",self.x+self.y)
    @classmethod
    def sub(cls):
        print("Sub=", cls.x-cls.y)
    @staticmethod
    def multiply(x,y):
       print("Mul=",x*y)
d=Demo(10,5)
d.add()
Demo.sub()
Demo.multiply(10,5)
```

OutPut:

```
Sum= 15
Sub= 5
Mul= 50
```

Inheritance or Is-A Relation in Python

- The inheritance is the process of acquiring the properties of one class to another class.
- Inheritance in python programming is the concept of deriving a new class from an existing class.
- Using the concept of inheritance we can inherit the properties of the existing class to our new class.
- The new derived class is called the **child class** and the existing class is called the **parent class**.
- The **Parent class** is the class which provides features to another class. The parent class is also known as **Base class** or **Superclass**.
- The **Child class** is the class which receives features from another class. The child class is also known as the **Derived Class** or **Subclass.**

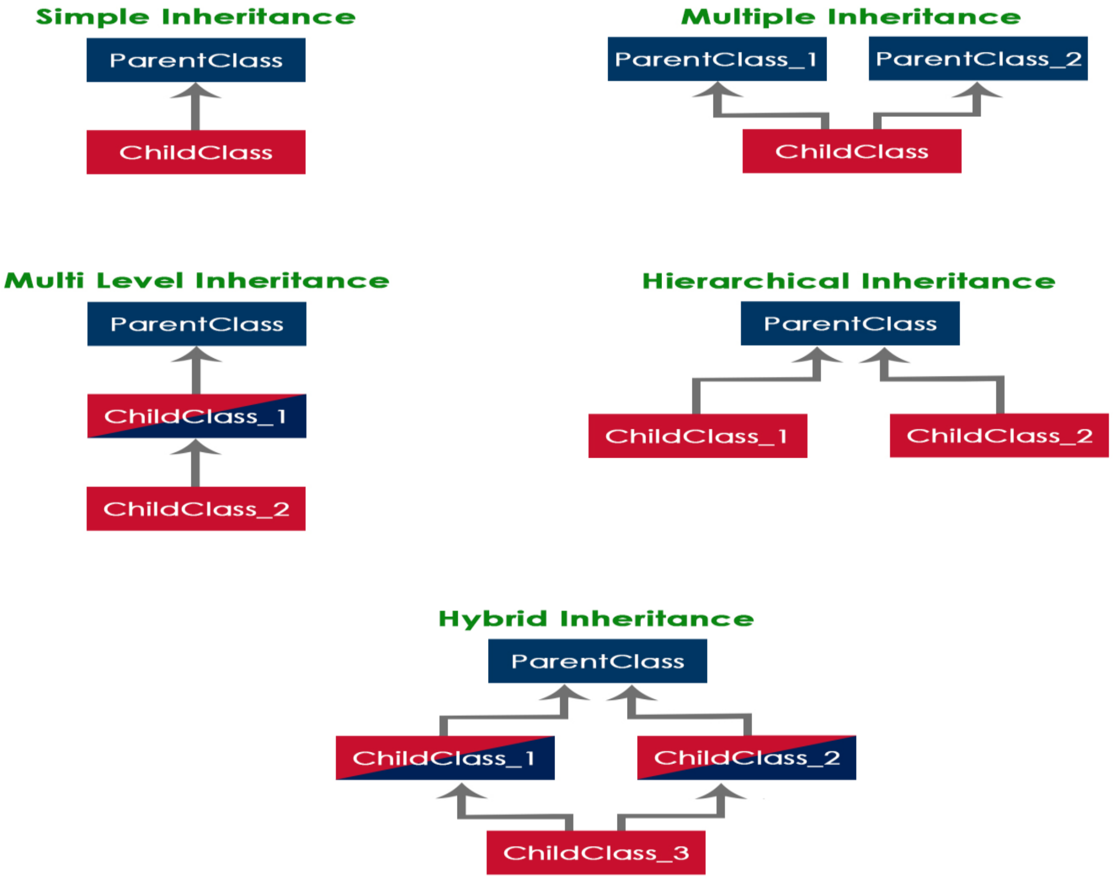
Advantages of Inheritance:

- **Code reusability**- we do not have to write the same code again and again, we can just inherit the properties we need in a child class.

- It represents a real world relationship between parent class and child class.
- It is transitive in nature. If a child class inherits properties from a parent class, then all other sub-classes of the child class will also inherit the properties of the parent class.

- There are five types of inheritances, and they are as follows.
    - Simple Inheritance (or) Single Inheritance
    - Multiple Inheritance
    - Multi-Level Inheritance
    - Hierarchical Inheritance
    - Hybrid Inheritance

The following picture illustrates how various inheritances are implemented.



Creating a Child Class

- In Python, we use the following general structure to create a child class from a parent class.

Syntax:

```
class ChildClassName(ParentClassName):
   ChildClass implementation
```

Simple Inheritance (or) Single Inheritance

In this type of inheritance, one child class derives from one parent class. Look at the following example code.

Example

```
class Parent:
    def func1(self):
        print("This function is in parent class.")

class Child(Parent):
    def func2(self):
        print("This function is in child class.")

object = Child()
object.func1()
object.func2()
```

OutPut:

```
This function is in parent class.
This function is in child class.
```

Multi-Level Inheritance

- In this type of inheritance, the child class derives from a class which already derived from another class. Look at the following example code.

Example:

```
class Parent:
    def func1(self):
        print('this is function 1')
class Child(Parent):
    def func2(self):
        print('this is function 2')
class Child2(Child):
    def func3(self):
        print('this is function 3')
ob = Child2()
ob.func1()
ob.func2()
ob.func3()
```

OutPut:

```
this is function 1
this is function 2
this is function 3
```

Hierarchical inheritance:

- When we derive or inherit more than one child class from one (same) parent class. Then this type of inheritance is called hierarchical inheritance.

Example:

```
class Parent:
    def func1(self):
        print("This function is in parent class.")
class Child1(Parent):
    def func2(self):
        print("This function is in child 1.")
class Child2(Parent):
    def func3(self):
        print("This function is in child 2.")
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

OutPut:

```
This function is in parent class.
This function is in child 1.
This function is in parent class.
This function is in child 2.
```

Multiple Inheritance:

When child class is derived or inherited from the more than one parent classes. This is called multiple inheritance. In multiple inheritance, we have two parent classes/base classes and one child class that inherits both parent classes' properties.

Example:

```
class Father:
    fathername = ""
    def father(self):
        print(self.fathername)
class Mother:
    mothername = ""
    def mother(self):
        print(self.mothername)

class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)
s1 = Son()
s1.fathername = "Raj"
s1.mothername = "Srav"
s1.parents()
```

OutPut:

```
    Father : Raj
    Mother : Srav
```

Hybrid Inheritance:

Hybrid inheritance satisfies more than one form of inheritance ie. It may be consists of all types of inheritance that we have done above. It is not wrong if we say Hybrid Inheritance is the combinations of simple, multiple, multilevel and hierarchical inheritance. This type of inheritance is very helpful if we want to use concepts of inheritance without any limitations according to our requirements.

Example:

```
    class School:
        def func1(self):
            print("This function is in school.")

    class Student1(School):
        def func2(self):
            print("This function is in student 1. ")

    class Student2(School):
        def func3(self):
            print("This function is in student 2.")

    class Student3(Student1, School):
        def func4(self):
            print("This function is in student 3.")
    object = Student3()
    object.func1()
    object.func2()
    object.func4()
```

OutPut:

```
    This function is in school.
    This function is in student 1.
    This function is in student 3.
```

Super() Function in Python:

- super() is a predefined function in python. By using super() function in child class, we can call,
  - Super class constructor.
  - Super class variables.
  - Super class methods.

1. Calling super class constructor from child class constructor using super()

Example:

```
class A:
  def __init__(self):
    print("super class A constructor")
class B(A):
  def __init__(self):
    print("Child class B constructor")
    super().__init__()
    b=B()
```

OutPut:

```
Child class B constructor
super class A constructor
```

2. Calling super class method from child class method using super()

```
class A:
  def m1(self):
    print("Super class A: m1 method")
class B(A):
  def m1(self):
    print("Child class B: m1 method")
    super().m1()
b=B()
b.m1()
```

Output:

```
Child class B: m1 method
Super class A: m1 method
```

3. Calling super class variable from child class method using super()

```
class A:
  x=10
  def m1(self):
    print("Super class A: m1 method")
```

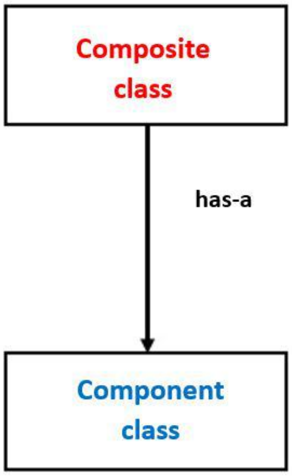
```
class B(A):
  x=20
  def m1(self):
      print('Child class x variable', self.x)
      print('Super class x variable', super().x)
b=B()
b.m1()
```

Output:

```
Child class x variable 20
Super class x variable 10
```

Composition (Has A Relation):

- It is one of the fundamental concepts of Object-Oriented Programming. In this concept, we will describe a class that references to one or more objects of other classes as an Instance variable. Here, by using the class name or by creating the object we can access the members of one class inside another class. It enables creating complex types by combining objects of different classes. It means that a class Composite can contain an object of another class Component. This type of relationship is known as **Has-A Relation**.
- In composition one of the classes is composed of one or more instance of other classes. In other words one class is container and other class is content and if you delete the container object then all of its contents objects are also deleted.



Syntax:

class A :

```
    # variables of class A
    # methods of class A
    …
    …
class B :
    # by using "object" we can access member's of class A.
    object = A()

    # variables of class B
    # methods of class B
    …
    …
```

Example:

```
class Component:
    def __init__(self):
        print('Component class object created…')
    def m1(self):
        print('Component class m1() method executed…')
class Composite:
    def __init__(self):
        self.obj1 = Component()
        print('Composite class object also created…')
    def m2(self):
        print('Composite class m2() method executed…')
        self.obj1.m1()
obj2 = Composite()
obj2.m2()
```

OutPut:

```
Component class object created…
Composite class object also created…
Composite class m2() method executed…
Component class m1() method executed…
```

Privacy or Python Access Modifiers:

- In most of the object-oriented languages access modifiers are used to limit the access to the variables and functions of a class. Most of the languages use three types of access modifiers, they are –
    - Private
    - Public

- Protected.

- Just like any other object oriented programming language, access to variables or functions can also be limited in python using the access modifiers. Python makes the use of underscores to specify the access modifier for a specific data member and member function in a class.
- Access modifiers play an important role to protect the data from unauthorized access as well as protecting it from getting manipulated.
- When inheritance is implemented there is a huge risk for the data to get destroyed(manipulated) due to transfer of unwanted data from the parent class to the child class. Therefore, it is very important to provide the right access modifiers for different data members and member functions depending upon the requirements.

Python: Types of Access Modifiers

- There are 3 types of access modifiers for a class in Python. These access modifiers define how the members of the class can be accessed. Of course, any member of a class is accessible inside any member function of that same class. Moving ahead to the type of access modifiers, they are:

Access Modifier: Public

- The members declared as Public are accessible from outside the Class through an object of the class.

Example:

```
class Student:
    def __init__(self,name,dept):
        self.name=name#Public attribute
        self.dept=dept#Public attribute
class Stud(Student):
    pass
s1=Student("Raj","CSE")
print("Name=",s1.name)
print("Dept=",s1.dept)
d=Stud("Srav","DS")
print("Name=",d.name)
print("Dept=",d.dept)
```

OutPut:

```
Name=Raj
Dept=CSE
Name=Srav
Dept=DS
```

Access Modifier: Private

- These members are only accessible from within the class. No outside Access is allowed.

- It is also not possible to inherit the private members of any class (parent class) to derived class (child class). Any instance variable in a class followed by self keyword and the variable name starting with double underscore ie. self.__varName are the private accessed member of a class.

Example:

```
class Student:
   def __init__(self, name, dept):
      self.__name = name # private
      self.__dept  = dept  # private
s1=Student("Raj","CSE")
print("Name=",s1.__name)
print("Dept=",s1.__dept)
```

OutPut: error

protected Access Modifier:

- Protected variables or we can say protected members of a class are restricted to be used only by the member functions and class members of the same class. And also it can be accessed or inherited by its derived class (child class).
- We can modify the values of protected variables of a class. The syntax we follow to make any variable protected is to write variable name followed by a single underscore (_) ie. _varName.

Example:

```
class Student:
   def __init__(self, name, dept):
      self._name = name # Protected
      self._dept  = dept  #Protected
class Stu(Student):
   pass
s1=Student("Raj","CSE")
print("Name=",s1._name)
print("Dept=",s1._dept)
s2=Stu("Srav","DS")
print("Name=",s2._name)
print("Dept=",s2._dept)
```

OutPut:

```
Name= Raj
Dept= CSE
Name= Srav
Dept= DS
```

Polymorphism in Python

- Polymorphism is taken from the Greek words Poly (many) and morphism (forms). It means that the same function name can be used for different types. This makes programming easier.
- Polymorphism means having vivid or different forms. In the programming world, Polymorphism refers to the ability of the function with the same name to carry different functionality altogether.

Types of Polymorphism :

- **Compile time Polymorphism**
- **Run time Polymorphism**

Compile time Polymorphism or Method Overloading:

- Unlike many other popular object-oriented programming languages such as Java, Python doesn't support compile-time polymorphism or method overloading. If a class or Python script has multiple methods with the same name, the method defined in the last will override the earlier one.
- Python doesn't use function arguments for method signature, that's why method overloading is not supported in Python.

Example:

```
class OverloadDemo:
  def multiply(self,a,b):
    print(a*b)
  def multiply(self,a,b,c):
    print(a*b*c)
m=OverloadDemo()
m.multiply(5,10)
```

OutPut:

```
Traceback (most recent call last):
  File "F:\R20-python\lab\ac.py", line 7, in <module>
    m.multiply(5,10)
TypeError: multiply() missing 1 required positional argument: 'c'
```

Run time Polymorphism or Method Overriding:

- In Python, whenever a method having same name and arguments is used in both derived class as well as in base or super class then the method used in derived class is said to override the method described in base class. Whenever the overridden method is called, it always invokes the method defined in derived class. The method used in base class gets hidden.

Example:

```
class methodOverride1:
   def display(self):
      print("method invoked from base class")

class methodOverride2(methodOverride1):
   def display(self):
      print("method invoked from derived class")

ob=methodOverride2()
ob.display()
```

OutPut:

```
method invoked from derived class
```

Built in Functions for Classes, Instances, and Other Objects,

In Python we have different typed of built in functions in Python.

1. **hasattr() Function**

- The python hasattr() function returns true if an object has given named attribute. Otherwise, it returns false.

Syntax: hasattr(object, attribute)

Parameters

- **object:** It is an object whose named attribute is to be checked.
- **attribute:** It is the name of the attribute that you want to search.
- **Return:**It returns true if an object has given named attribute. Otherwise, it returns false.

Example:

```
class Demo:
   name="raj"
   dept="CSE"
obj=Demo()
print(hasattr(Demo,'name'))
print(hasattr(Demo,'rollno'))
```

OutPut:

```
True
False
```

2. **getattr() Function**

- The python getattr() function returns the value of a named attribute of an object. If it is not found, it returns the default value.

Syntax: getattr(object, attribute, default)

Parameters:

- **object:** An object whose named attribute value is to be returned.
- **attribute:** Name of the attribute of which you want to get the value.
- **default (optional):** It is the value to return if the named attribute does not found.

Return: It returns the value of a named attribute of an object. If it is not found, it returns the default value.

Example:

```
class Demo:
   name="raj"
   dept="CSE"
obj=Demo()
print("name=",getattr(Demo,'name'))
print("college=",getattr(Demo,'college',"MREC"))
```

OutPut:

```
name= raj
college= MREC
```

3. setattr() Function

- Python setattr() function is used to set a value to the object's attribute. It takes three arguments an object, a string, and an arbitrary value, and returns none. It is helpful when we want to add a new attribute to an object and set a value to it. The signature of the function is given below.

Syntax: setattr (object, name, value)

Parameters

- **object:** It is an object which allows its attributes to be changed.
- **name :** A name of the attribute.
- **value :** A value, set to the attribute.

Return: It returns None to the caller function.

Example:

```
class Demo:
   name=""
   dept=""
   id=0
   def __init__(self,name,dept,id):
      self.name=name
      self.dept=dept
      self.id=id
obj=Demo("Raj","CSE",1)
print(obj.name)
print(obj.dept)
print(obj.id)
```

```
setattr(obj,'college','MREC')
print(obj.college)
```

OutPut:

```
Raj
CSE
1
MREC
```

4. delattr() Function

- Python delattr() function is used to delete an attribute from a class. It takes two parameters first is an object of the class and second is an attribute which we want to delete. After deleting the attribute, it no longer available in the class and throws an error if try to call it using the class object.

Syntax: delattr (object, name)

Parameters

- **object:** Object of the class which contains the attribute.
- **name:** The name of the attribute to delete. It must be a string.

Return: It returns a complex number.

Example:

```
class Demo:
   name=""
   dept=""
   id=0
   def __init__(self,name,dept,id):
      self.name=name
      self.dept=dept
      self.id=id
obj=Demo("Raj","CSE",1)
print(obj.name)
print(obj.dept)
print(obj.id)
setattr(obj,'college','MREC')
print(obj.college)
delattr(obj,'college')
print(obj.college)
```

OutPut:

```
Raj
CSE
1
MREC
Traceback (most recent call last):
  File "F:\R20-python\lab\ac.py", line 16, in <module>
```

```
            print(obj.college)
          AttributeError: 'Demo' object has no attribute 'college'
```

5. isinstance() Function

- Python isinstance() function is used to check whether the given object is an instance of that class. If the object belongs to the class, it returns True. Otherwise returns False. It also returns true if the class is a subclass.
- The isinstance() function takes two arguments object and classinfo and returns either True or False. The signature of the function is given below.

Syntax: isinstance(object, classinfo)

Parameters

- **object:** It is an object of string, int, float, long or custom type.
- **classinfo:** Class name.

Return: It returns boolean either True or False.

6. issubclass() function

- The issubclass() function returns True if the specified object is a subclass of the specified object, otherwise False.

Syntax: issubclass(object, subclass)

Parameter

- **object** It is an object of string, int, float, long or custom type.
- **subclass** Name of the subclass

Return: It returns boolean either True or False.

Example:

```
class A:
  pass
class B(A):
  pass
b=B()
print("b is an instance of the class B:",isinstance(b,B))
print("B is sub class of A Class:",issubclass(B,A))
```

OutPut:

```
b is an instance of the class B: True
B is sub class of A Class: True
```

Types vs. Classes/Instances:

Properties	**Types**	**Class**
Origin	Pre-defined data types	User-defined data types
Stored structure	Stored in a stack	Reference variable is stored in stack and the original object is

		stored in heap
When copied	Two different variables is created along with different assignment even though the both variables having the same address if they are pointing the same value	Two reference variable is created but both are pointing to the same object on the heap
When changes are made in the copied variable	Change does not reflect in the original ones.	Changes reflected in the original ones.
Default value	Primitive datatypes having the default value like 0 for int 0.0 for float etc.	No default value for the reference variable which is created for a class

Example	a=15
	print("Type of a=",type(a))
	
	output:
	Type of a= <class 'int'>
		class A:
		def __init__(self,a):
		self.a=a
		obj=A(10)
		print("Type of obj=",type(obj))
		
		output:
		Type of obj= <class '__main__.A'>

Delegation and Wrapping

- Delegation is the mechanism through which an actor assigns a task or part of a task to another actor. This is not new in computer science, as any program can be split into blocks and each block generally depends on the previous ones. Furthermore, code can be isolated in libraries and reused in different parts of a program, implementing this "task assignment". In an OO system the assignee is not just the code of a function, but a full-fledged object, another actor.
- The main concept to retain here is that the reason behind delegation is code reuse. We want to avoid code repetition, as it is often the source of regressions; fixing a bug

in one of the repetitions doesn't automatically fix it in all of them, so keeping one single version of each algorithm is paramount to ensure the consistency of a system.

- Delegation helps us to keep our actors small and specialised, which makes the whole architecture more flexible and easier to maintain (if properly implemented). Changing a very big subsystem to satisfy a new requirement might affect other parts system in bad ways, so the smaller the subsystems the better (up to a certain point, where we incur in the opposite problem, but this shall be discussed in another post).

Example:

```
class Dept:
   def __init__(self,insem,endsem):
      self.insem=insem
      self.endsem=endsem
   def marks(self):
      return self.insem+self.endsem

class student:
   def __init__(self,sname,year,insem,endsem):
      self.sname=sname
      self.year=year
      self.obj_Dept=Dept(insem,endsem)
   def tmarks(self):
      print("The Total Marks of %s"
%self.sname,self.obj_Dept.marks())
s=student('Raj','First',20,65)
s.tmarks()
```

OutPut:

```
The Total Marks of Raj 85
```

Python Programming

MODULE – V

Agenda:

- GUI Programming: Introduction
- Tkinter and Python Programming
- Brief Tour of Other GUIs
- Related Modules and Other GUIs.

Graphical User Interface (GUI)

Graphical User Interface (GUI) is nothing but a desktop application which helps you to interact with the computers. They are used to perform different tasks in the desktops, laptops and other electronic devices.

- **GUI** apps like **Text-Editors** are used to create, read, update and delete different types of files.
- **GUI** apps like Chess and Solitaire are games which you can play.
- **GUI** apps like **Google Chrome, Firefox and Microsoft Edge** are used to browse through the **Internet**.

They are some different types of **GUI** apps which we daily use on the laptops or desktops.

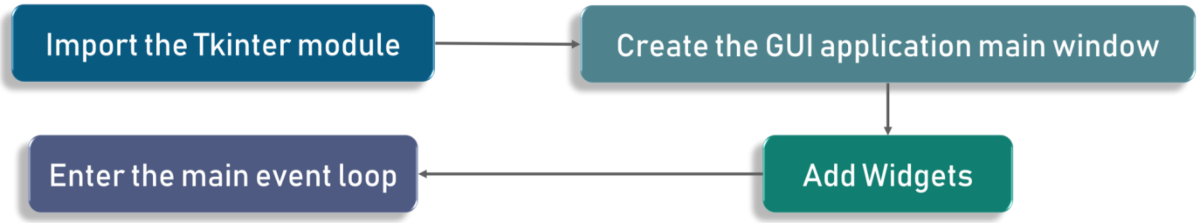
Python Libraries To Create Graphical User Interfaces:
Python has a excess of libraries and these 4 stands out mainly when it comes to GUI. There are as follows:

- Kivy
- Python QT
- wxPython
- Tkinter
- Jpython

Among all of these, Tkinter is preferred by learners and developers just because of how simple and easy it is.

Tkinter

- The primary GUI toolkit we will be using is Tk (Toolkit), Python's default GUI, We'll access Tk from its Python interface called Tkinter (i.e. **Tk interface**). It is originally designed for the **Tool Command Language** (**TCL**). Due to Tk's popularity, it has been ported to a variety of other scripting languages, including Perl (Perl/Tk), Ruby (Ruby/Tk), and Python (Tkinter).
- **Tkinter** is actually an inbuilt **Python** module used to create simple **GUI** apps. It is the most commonly used module for **GUI** apps in the **Python**.
- We no need to worry about installation of the **Tkinter** module as it comes with **Python 3.6 version** by default.
- Consider the following diagram, it shows how an application actually executes in Tkinter:


Import the Tkinter module
Create the GUI application main window
Add Widgets
Enter the main event loop

Developing desktop based applications with python Tkinter is not a complex task. An empty Tkinter top-level window can be created by using the following steps.

1. import the Tkinter module.
2. Create the main application window.
3. Add the **Widgets** like labels, buttons, frames, etc. to the window.
4. Call the **Main event loop** so that the actions can take place on the user's computer screen.

If you noticed, there are 2 keywords here that you might not know at this point. These are the 2 keywords:

- Widgets
- Main Event Loop

An event loop is basically telling the code to keep displaying the window until we manually close it. It runs in an infinite loop in the back-end.

Example

```
from tkinter import *

#creating the application main window.
 top = Tk()

 #Entering the event main loop
top.mainloop()
```

Example

```
import tkinter
window = tkinter.Tk()

# to rename the title of the window window.title("MREC")
 # pack is used to show the object in the window

label = tkinter.Label(window, text = "Hello MREC CSE!").pack()
window.title("GUI Window")
 window.mainloop()
```

Tkinter widgets

Widgets are something like elements in the **HTML**. You will find different types of **widgets** to the different types of elements in the **Tkinter**.

There are various widgets like button, canvas, checkbutton, entry, etc. that are used to build the python GUI applications.

SN	Widget	Description
1	Button	The Button is used to add various kinds of buttons to the python

		application.
2	Canvas	The canvas widget is used to draw the shapes in your GUI.
3	Checkbutton	The Checkbutton is used to display the CheckButton on the window.
4	Entry	The entry widget is used to display the single-line text field to the user. It is commonly used to accept user values (Input Fields)
5	Frame	It can be defined as a container to which, another widget can be added and organized.
6	Label	A label is a text used to display some message or information about the other widgets.
7	ListBox	The ListBox widget is used to display a list of options to the user.
8	Menubutton	The Menubutton is used to display the menu items to the user.
9	Menu	It is used to add menu items to the user.
10	Message	The Message widget is used to display the message-box to the user.
11	Radiobutton	The Radiobutton is different from a checkbutton. Here, the user is provided with various options and the user can select only one option among them.
12	Scale	It is used to provide the slider to the user.
13	Scrollbar	It provides the scrollbar to the user so that the user can scroll the window up and down.
14	Text	It is different from Entry because it provides a multi-line text field to the user so that the user can write the text and edit the text inside it.
14	Toplevel	It is used to create a separate window container.
15	Spinbox	It is an entry widget used to select from options of values.
16	PanedWindow	It is like a container widget that contains horizontal or vertical panes.
17	LabelFrame	A LabelFrame is a container widget that acts as the container
18	MessageBox	This module is used to display the message-box in the desktop based applications.

These widgets are the reason that Tkinter is so popular. It makes it really easy to understand and use practically.

Python Tkinter Geometry Managers

The Tkinter geometry specifies the method by using which; the widgets are represented on display. The python Tkinter provides three geometry methods that help with positioning our widgets.

1. The pack() method
2. The grid() method
3. The place() method

Python Tkinterpack() method

The pack() widget is used to organize widget in the block, which mean it occupies the entire available width. It's a standard method to show the widgets in the window.

The syntax to use the pack() is given below.

Syntax:

widget.pack(options)

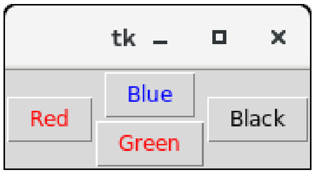
A list of possible options that can be passed in pack() is given below.

- **side:** it represents the side of the parent to which the widget is to be placed on the window.

Example

```
from tkinter import *
parent = Tk()
redbutton = Button(parent, text = "Red", fg = "red")
redbutton.pack( side = LEFT)
greenbutton = Button(parent, text = "Black", fg = "black")
greenbutton.pack( side = RIGHT )
bluebutton = Button(parent, text = "Blue", fg = "blue")
bluebutton.pack( side = TOP )
blackbutton = Button(parent, text = "Green", fg = "red")
blackbutton.pack( side = BOTTOM)
parent.mainloop()
```

Output:



Python Tkintergrid() method

The grid() geometry manager organizes the widgets in the tabular form. We can specify the rows and columns as the options in the method call. We can also specify the column span (width) or rowspan(height) of a widget.

This is a more organized way to place the widgets to the python application. The syntax to use the grid() is given below.

Syntax

widget.grid(options)

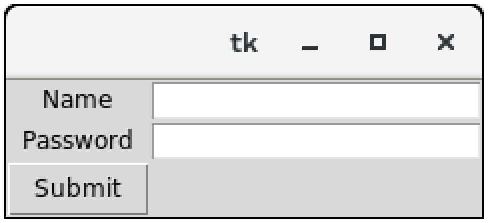
A list of possible options that can be passed inside the grid() method is given below.

- **Column**
  The column number in which the widget is to be placed. The leftmost column is represented by 0.
- **row**
  The row number in which the widget is to be placed. The topmost row is represented by 0.

Example

```
from tkinter import *
parent = Tk()
name = Label(parent,text = "Name").grid(row = 0, column = 0)
e1 = Entry(parent).grid(row = 0, column = 1)
password = Label(parent,text = "Password").grid(row = 1, column = 0)
e2 = Entry(parent).grid(row = 1, column = 1)
submit = Button(parent, text = "Submit").grid(row = 4, column = 0)
parent.mainloop()
```

Output:



Python Tkinterplace() method

The place() geometry manager organizes the widgets to the specific x and y coordinates.

Syntax

widget.place(options)

A list of possible options is given below.

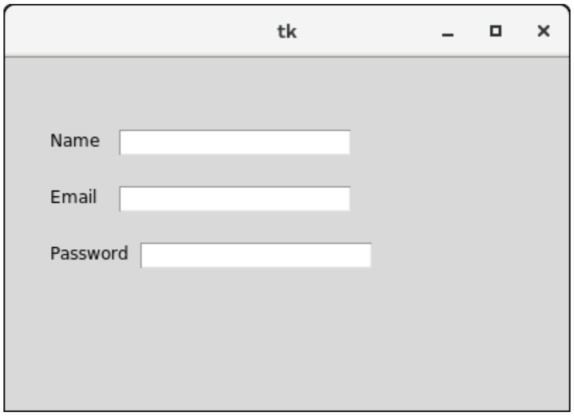
- **x, y:** It refers to the horizontal and vertical offset in the pixels.

Example

```
from tkinter import *
top = Tk()
top.geometry("400x250")
```

```
name = Label(top, text = "Name").place(x = 30,y = 50)
email = Label(top, text = "Email").place(x = 30, y = 90)
password = Label(top, text = "Password").place(x = 30, y = 130)
e1 = Entry(top).place(x = 80, y = 50)
e2 = Entry(top).place(x = 80, y = 90)
e3 = Entry(top).place(x = 95, y = 130)
top.mainloop()
```

Output:



Python Tkinter Button

The button widget is used to add various types of buttons to the python application. Python allows us to configure the look of the button according to our requirements. Various options can be set or reset depending upon the requirements.

We can also associate a method or function with a button which is called when the button is pressed.

The syntax to use the button widget is given below.

Syntax

W = Button(parent, options)

A list of possible options is given below.

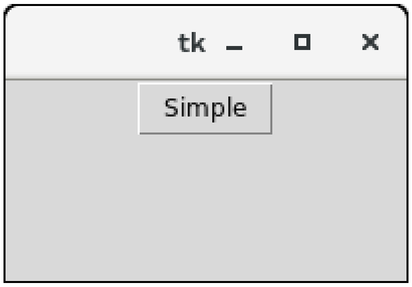
SN	Option	Description
1	activebackground	It represents the background of the button when the mouse hover the button.
2	activeforeground	It represents the font color of the button when the mouse hover the button.
3	Bd	It represents the border width in pixels.
4	Bg	It represents the background color of the button.
5	Command	It is set to the function call which is scheduled when the function is called.

6	Fg	Foreground color of the button.
7	Font	The font of the button text.
8	Height	The height of the button. The height is represented in the number of text lines for the textual lines or the number of pixels for the images.
10	Highlightcolor	The color of the highlight when the button has the focus.
11	Image	It is set to the image displayed on the button.
12	Justify	It illustrates the way by which the multiple text lines are represented. It is set to LEFT for left justification, RIGHT for the right justification, and CENTER for the center.
13	Padx	Additional padding to the button in the horizontal direction.
14	Pady	Additional padding to the button in the vertical direction.
15	Relief	It represents the type of the border. It can be SUNKEN, RAISED, GROOVE, and RIDGE.
17	State	This option is set to DISABLED to make the button unresponsive. The ACTIVE represents the active state of the button.
18	Underline	Set this option to make the button text underlined.
19	Width	The width of the button. It exists as a number of letters for textual buttons or pixels for image buttons.
20	Wraplength	If the value is set to a positive number, the text lines will be wrapped to fit within this length.

Example

```
#python application to create a simple button

from tkinter import *
top = Tk()
top.geometry("200x100")

b = Button(top,text = "Simple")
b.pack()
top.mainaloop()
```

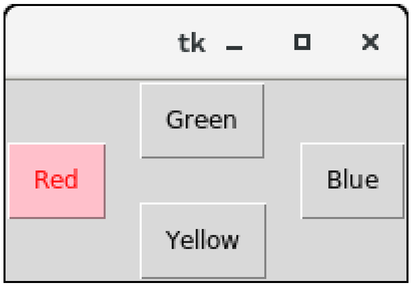
Output:

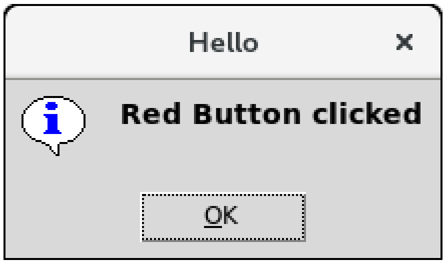


Example

```
from tkinter import *
from tkinter import messagebox

top = Tk()
top.geometry("200x100")

def fun():
   messagebox.showinfo("Hello", "Red Button clicked")

b1 = Button(top,text = "Red",command = fun,activeforeground = "red",activebackg
round = "pink",pady=10)
b2 = Button(top, text = "Blue",activeforeground = "blue",activebackground = "pink
",pady=10)
b3 = Button(top, text = "Green",activeforeground = "green",activebackground = "pi
nk",pady = 10)
b4 = Button(top, text = "Yellow",activeforeground = "yellow",activebackground = "
pink",pady = 10) b1.pack(side = LEFT)
b2.pack(side = RIGHT)
b3.pack(side = TOP)
b4.pack(side = BOTTOM)
top.mainloop()
```

Output:





Red Button clicked

Python Tkinter Canvas

- The canvas widget is used to add the structured graphics to the python application. It is used to draw the graph and plots to the python application. The syntax to use the canvas is given below.

Syntax

w = canvas(parent, options)

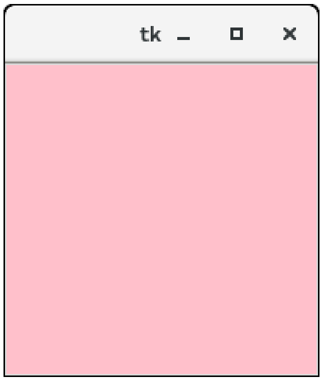
A list of possible options is given below.

SN	Option	Description
1	Bd	The represents the border width. The default width is 2.
2	Bg	It represents the background color of the canvas.
3	confine	It is set to make the canvas unscrollable outside the scroll region.
4	cursor	The cursor is used as the arrow, circle, dot, etc. on the canvas.
5	height	It represents the size of the canvas in the vertical direction.
6	highlightcolor	It represents the highlight color when the widget is focused.
7	relief	It represents the type of the border. The possible values are SUNKEN, RAISED, GROOVE, and RIDGE.
8	scrollregion	It represents the coordinates specified as the tuple containing the area of the canvas.
9	width	It represents the width of the canvas.
10	xscrollincrement	If it is set to a positive value. The canvas is placed only to the multiple of this value.
11	xscrollcommand	If the canvas is scrollable, this attribute should be the .set() method of the horizontal scrollbar.
12	yscrollincrement	Works like xscrollincrement, but governs vertical movement.
13	yscrollcommand	If the canvas is scrollable, this attribute should be the .set() method of the vertical scrollbar.

Example

```
from tkinter import *
top = Tk()
top.geometry("200x200")
#creating a simple canvas
c = Canvas(top,bg = "pink",height = "200")
c.pack()
top.mainloop()
```

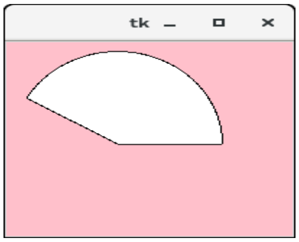
Output:



Example: Creating an arc

```
        from tkinter import *

        top = Tk()
        top.geometry("200x200")

        #creating a simple canvas
        c = Canvas(top,bg = "pink",height = "200",width = 200)
        arc = c.create_arc((5,10,150,200),start = 0,extent = 150, fill= "white")
        c.pack()
        top.mainloop()
```

Output:



Python TkinterCheckbutton

The Checkbutton is used to track the user's choices provided to the application. In other words, we can say that Checkbutton is used to implement the on/off selections.

The Checkbutton can contain the text or images. The Checkbutton is mostly used to provide many choices to the user among which, the user needs to choose the one. It generally implements many of many selections.

The syntax to use the checkbutton is given below.

Syntax

```
w = checkbutton(master, options)
```

A list of possible options is given below.

SN	Option	Description
1	activebackground	It represents the background color when the checkbutton is under the cursor.
2	activeforeground	It represents the foreground color of the checkbutton when the checkbutton is under the cursor.
3	Bg	The background color of the button.
4	Bitmap	It displays an image (monochrome) on the button.
5	Bd	The size of the border around the corner.
6	Command	It is associated with a function to be called when the state of the

		checkbutton is changed.
7	Cursor	The mouse pointer will be changed to the cursor name when it is over the checkbutton.
8	disableforeground	It is the color which is used to represent the text of a disabled checkbutton.
9	font	It represents the font of the checkbutton.
10	fg	The foreground color (text color) of the checkbutton.
11	height	It represents the height of the checkbutton (number of lines). The default height is 1.
12	highlightcolor	The color of the focus highlight when the checkbutton is under focus.
13	image	The image used to represent the checkbutton.
14	justify	This specifies the justification of the text if the text contains multiple lines.
15	offvalue	The associated control variable is set to 0 by default if the button is unchecked. We can change the state of an unchecked variable to some other one.
16	onvalue	The associated control variable is set to 1 by default if the button is checked. We can change the state of the checked variable to some other one.
17	padx	The horizontal padding of the checkbutton
18	pady	The vertical padding of the checkbutton.
19	relief	The type of the border of the checkbutton. By default, it is set to FLAT.
20	selectcolor	The color of the checkbutton when it is set. By default, it is red.
21	selectimage	The image is shown on the checkbutton when it is set.
22	state	It represents the state of the checkbutton. By default, it is set to normal. We can change it to DISABLED to make the checkbutton unresponsive. The state of the checkbutton is ACTIVE when it is under focus.
24	underline	It represents the index of the character in the text which is to be underlined. The indexing starts with zero in the text.
25	variable	It represents the associated variable that tracks the state of the checkbutton.
26	width	It represents the width of the checkbutton. It is represented in the number of characters that are represented in the form of texts.

27	wraplength	If this option is set to an integer number, the text will be broken into the number of pieces.

Methods

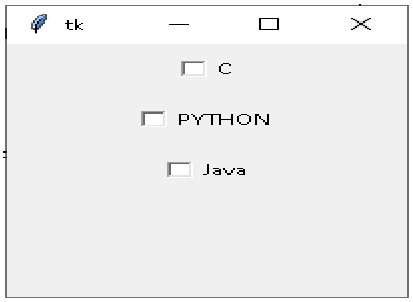
The methods that can be called with the Checkbuttons are described in the following table.

SN	Method	Description
1	deselect()	It is called to turn off the checkbutton.
2	flash()	The checkbutton is flashed between the active and normal colors.
3	invoke()	This will invoke the method associated with the checkbutton.
4	select()	It is called to turn on the checkbutton.
5	toggle()	It is used to toggle between the different Checkbuttons.

Example

```
from tkinter import *

top = Tk()

top.geometry("200x200")

chkbtn1 = Checkbutton(top, text = "C", )

chkbtn2 = Checkbutton(top, text = "PYTHON", )

chkbtn3 = Checkbutton(top, text = "Java", )

chkbtn1.pack()

chkbtn2.pack()

chkbtn3.pack()

top.mainloop()
```

Output:



Python Tkinter Entry

The Entry widget is used to provide the single line text-box to the user to accept a value from the user. We can use the Entry widget to accept the text strings from the user. It can only be used for one line of text from the user. For multiple lines of text, we must use the text widget.

The syntax to use the Entry widget is given below.

Syntax

w = Entry (parent, options)

A list of possible options is given below.

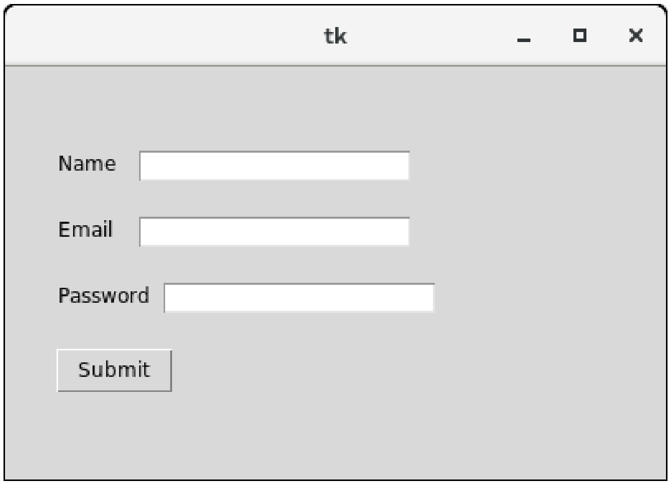
SN	Option	Description
1	Bg	The background color of the widget.
2	Bd	The border width of the widget in pixels.
3	Cursor	The mouse pointer will be changed to the cursor type set to the arrow, dot, etc.
4	Exportselection	The text written inside the entry box will be automatically copied to the clipboard by default. We can set the exportselection to 0 to not copy this.
5	Fg	It represents the color of the text.
6	Font	It represents the font type of the text.
7	Highlightbackground	It represents the color to display in the traversal highlight region when the widget does not have the input focus.
8	Highlightcolor	It represents the color to use for the traversal highlight rectangle that is drawn around the widget when it has the input focus.
9	Highlightthickness	It represents a non-negative value indicating the width of the highlight rectangle to draw around the outside of the widget when it has the input focus.
10	Insertbackground	It represents the color to use as background in the area covered by the insertion cursor. This color will normally override either the normal background for the widget.
11	Insertborderwidth	It represents a non-negative value indicating the width of the 3-D border to draw around the insertion cursor. The value may have any of the forms acceptable to Tk_GetPixels.
12	Insertofftime	It represents a non-negative integer value indicating the number of milliseconds the insertion cursor should remain "off" in each blink cycle. If this option is zero, then the cursor doesn't blink: it is on all the time.

13	Insertontime	Specifies a non-negative integer value indicating the number of milliseconds the insertion cursor should remain "on" in each blink cycle.
14	Insertwidth	It represents the value indicating the total width of the insertion cursor. The value may have any of the forms acceptable to Tk_GetPixels.
15	Justify	It specifies how the text is organized if the text contains multiple lines.
16	Relief	It specifies the type of the border. Its default value is FLAT.
17	Selectbackground	The background color of the selected text.
18	Selectborderwidth	The width of the border to display around the selected task.
19	Selectforeground	The font color of the selected task.
20	Show	It is used to show the entry text of some other type instead of the string. For example, the password is typed using stars (*).
21	Textvariable	It is set to the instance of the StringVar to retrieve the text from the entry.
22	Width	The width of the displayed text or image.
23	Xscrollcommand	The entry widget can be linked to the horizontal scrollbar if we want the user to enter more text then the actual width of the widget.

Example

```
from tkinter import *

top = Tk()
top.geometry("400x250")
name = Label(top, text = "Name").place(x = 30,y = 50)
email = Label(top, text = "Email").place(x = 30, y = 90)
password = Label(top, text = "Password").place(x = 30, y = 130)
sbmitbtn = Button(top, text = "Submit",activebackground = "pink", activeforegroun
d = "blue").place(x = 30, y = 170)
e1 = Entry(top).place(x = 80, y = 50)
e2 = Entry(top).place(x = 80, y = 90)
e3 = Entry(top).place(x = 95, y = 130)
top.mainloop()
```

Output:



Entry widget methods

Python provides various methods to configure the data written inside the widget. There are the following methods provided by the Entry widget.

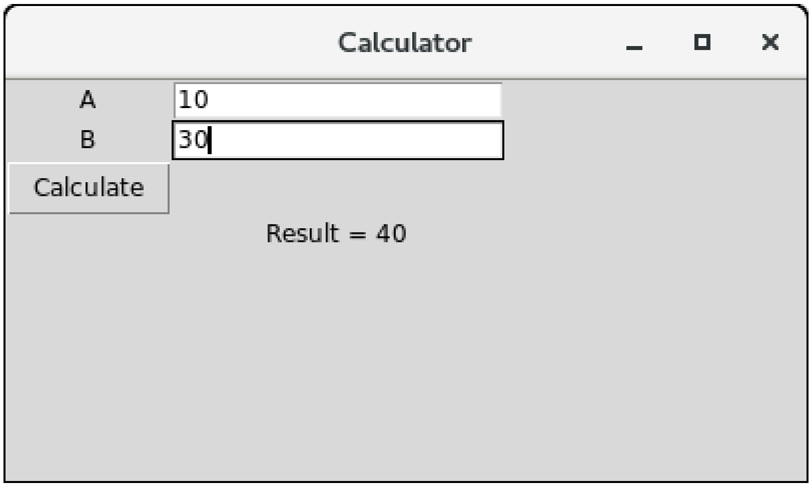
SN	Method	Description
1	delete(first, last = none)	It is used to delete the specified characters inside the widget.
2	get()	It is used to get the text written inside the widget.
3	icursor(index)	It is used to change the insertion cursor position. We can specify the index of the character before which, the cursor to be placed.
4	index(index)	It is used to place the cursor to the left of the character written at the specified index.
5	insert(index,s)	It is used to insert the specified string before the character placed at the specified index.
6	select_adjust(index)	It includes the selection of the character present at the specified index.
7	select_clear()	It clears the selection if some selection has been done.
8	select_form(index)	It sets the anchor index position to the character specified by the index.
9	select_present()	It returns true if some text in the Entry is selected otherwise returns false.
10	select_range(start,end)	It selects the characters to exist between the specified range.
11	select_to(index)	It selects all the characters from the beginning to the specified index.
12	xview(index)	It is used to link the entry widget to a horizontal scrollbar.

13	xview_scroll(number,what)	It is used to make the entry scrollable horizontally.

Example: A simple calculator

```
import tkinter as tk
from functools import partial

def call_result(label_result, n1, n2):
   num1 = (n1.get())
   num2 = (n2.get())
   result = int(num1)+int(num2)
   label_result.config(text="Result = %d" % result)
   return

root = tk.Tk()
root.geometry('400x200+100+200')

root.title('Calculator')

number1 = tk.StringVar()
number2 = tk.StringVar()

labelNum1 = tk.Label(root, text="A").grid(row=1, column=0)
labelNum2 = tk.Label(root, text="B").grid(row=2, column=0)

labelResult = tk.Label(root)

labelResult.grid(row=7, column=2)
entryNum1 = tk.Entry(root, textvariable=number1).grid(row=1, column=2)
entryNum2 = tk.Entry(root, textvariable=number2).grid(row=2, column=2)

call_result = partial(call_result, labelResult, number1, number2)

buttonCal = tk.Button(root, text="Calculate", command=call_result).grid(row=3, c
olumn=0)
root.mainloop()
```

Output:



A 10
B 30

Result = 40

Python Tkinter Frame

Python Tkinter Frame widget is used to organize the group of widgets. It acts like a container which can be used to hold the other widgets. The rectangular areas of the screen are used to organize the widgets to the python application.

The syntax to use the Frame widget is given below.

Syntax

w = Frame(parent, options)

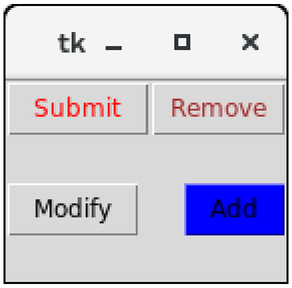
A list of possible options is given below.

SN	Option	Description
1	Bd	It represents the border width.
2	Bg	The background color of the widget.
3	Cursor	The mouse pointer is changed to the cursor type set to different values like an arrow, dot, etc.
4	Height	The height of the frame.
5	highlightbackground	The color of the background color when it is under focus.
6	Highlightcolor	The text color when the widget is under focus.
7	highlightthickness	It specifies the thickness around the border when the widget is under the focus.
8	Relief	It specifies the type of the border. The default value if FLAT.
9	Width	It represents the width of the widget.

Example

```
from tkinter import *

top = Tk()
top.geometry("140x100")
frame = Frame(top)
frame.pack()

leftframe = Frame(top)
leftframe.pack(side = LEFT)

rightframe = Frame(top)
rightframe.pack(side = RIGHT)

btn1 = Button(frame, text="Submit", fg="red",activebackground = "red")
btn1.pack(side = LEFT)

btn2 = Button(frame, text="Remove", fg="brown", activebackground = "brown")
btn2.pack(side = RIGHT)

btn3 = Button(rightframe, text="Add", fg="blue", activebackground = "blue")
btn3.pack(side = LEFT)

btn4 = Button(leftframe, text="Modify", fg="black", activebackground = "white")
btn4.pack(side = RIGHT)

top.mainloop()
```

Output:



Python Tkinter Label

The Label is used to specify the container box where we can place the text or images. This widget is used to provide the message to the user about other widgets used in the python application.

There are the various options which can be specified to configure the text or the part of the text shown in the Label.

The syntax to use the Label is given below.

Syntax

w = Label (master, options)

A list of possible options is given below.

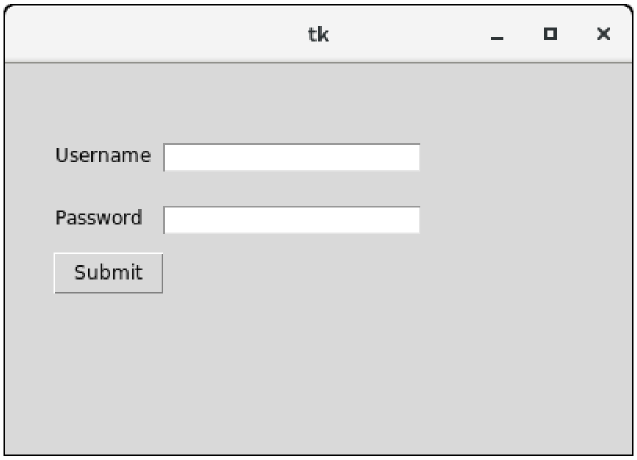
SN	Option	Description
1	Anchor	It specifies the exact position of the text within the size provided to the widget. The default value is CENTER, which is used to center the text within the specified space.
2	Bg	The background color displayed behind the widget.
3	Bitmap	It is used to set the bitmap to the graphical object specified so that, the label can represent the graphics instead of text.
4	Bd	It represents the width of the border. The default is 2 pixels.
5	Cursor	The mouse pointer will be changed to the type of the cursor specified, i.e., arrow, dot, etc.
6	Font	The font type of the text written inside the widget.
7	Fg	The foreground color of the text written inside the widget.
8	Height	The height of the widget.
9	Image	The image that is to be shown as the label.
10	Justify	It is used to represent the orientation of the text if the text contains multiple lines. It can be set to LEFT for left justification, RIGHT for right justification, and CENTER for center justification.
11	Padx	The horizontal padding of the text. The default value is 1.
12	Pady	The vertical padding of the text. The default value is 1.
13	Relief	The type of the border. The default value is FLAT.
14	Text	This is set to the string variable which may contain one or more line of text.
15	textvariable	The text written inside the widget is set to the control variable StringVar so that it can be accessed and changed accordingly.
16	underline	We can display a line under the specified letter of the text. Set this option to the number of the letter under which the line will be displayed.
17	Width	The width of the widget. It is specified as the number of characters.
18	wraplength	Instead of having only one line as the label text, we can break it to the number of lines where each line has the number of characters specified to

		this option.

Example 1

```
from tkinter import *

top = Tk()

top.geometry("400x250")

#creating label
uname = Label(top, text = "Username").place(x = 30,y = 50)

#creating label
password = Label(top, text = "Password").place(x = 30, y = 90)

sbmitbtn = Button(top, text = "Submit",activebackground = "pink", activeforegroun
d = "blue").place(x = 30, y = 120)

e1 = Entry(top,width = 20).place(x = 100, y = 50)
e2 = Entry(top, width = 20).place(x = 100, y = 90)
top.mainloop()
```

Output:



Python TkinterListbox

The Listbox widget is used to display the list items to the user. We can place only text items in the Listbox and all text items contain the same font and color.

The user can choose one or more items from the list depending upon the configuration.

The syntax to use the Listbox is given below.

w = Listbox(parent, options)

A list of possible options is given below.

SN	Option	Description
1	Bg	The background color of the widget.
2	Bd	It represents the size of the border. Default value is 2 pixel.
3	Cursor	The mouse pointer will look like the cursor type like dot, arrow, etc.
4	Font	The font type of the Listbox items.
5	Fg	The color of the text.
6	Height	It represents the count of the lines shown in the Listbox. The default value is 10.
7	highlightcolor	The color of the Listbox items when the widget is under focus.
8	highlightthickness	The thickness of the highlight.
9	Relief	The type of the border. The default is SUNKEN.
10	selectbackground	The background color that is used to display the selected text.
11	selectmode	It is used to determine the number of items that can be selected from the list. It can set to BROWSE, SINGLE, MULTIPLE, EXTENDED.
12	Width	It represents the width of the widget in characters.
13	xscrollcommand	It is used to let the user scroll the Listbox horizontally.
14	yscrollcommand	It is used to let the user scroll the Listbox vertically.

Methods

There are the following methods associated with the Listbox.

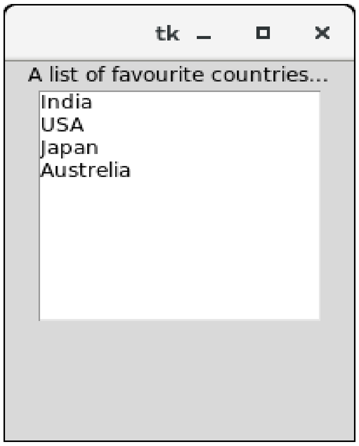
SN	Method	Description
1	activate(index)	It is used to select the lines at the specified index.
2	curselection()	It returns a tuple containing the line numbers of the selected element or elements, counting from 0. If nothing is selected, returns an empty tuple.
3	delete(first, last = None)	It is used to delete the lines which exist in the given range.
4	get(first, last = None)	It is used to get the list items that exist in the given range.
5	index(i)	It is used to place the line with the specified index at the top of the widget.
6	insert(index, *elements)	It is used to insert the new lines with the specified number of elements before the specified index.
7	nearest(y)	It returns the index of the nearest line to the y coordinate of

		the Listbox widget.
8	see(index)	It is used to adjust the position of the listbox to make the lines specified by the index visible.
9	size()	It returns the number of lines that are present in the Listbox widget.
10	xview()	This is used to make the widget horizontally scrollable.
11	xview_moveto(fraction)	It is used to make the listbox horizontally scrollable by the fraction of width of the longest line present in the listbox.
12	xview_scroll(number, what)	It is used to make the listbox horizontally scrollable by the number of characters specified.
13	yview()	It allows the Listbox to be vertically scrollable.
14	yview_moveto(fraction)	It is used to make the listbox vertically scrollable by the fraction of width of the longest line present in the listbox.
15	yview_scroll (number, what)	It is used to make the listbox vertically scrollable by the number of characters specified.

Example 1

```
from tkinter import *

top = Tk()

top.geometry("200x250")

lbl = Label(top,text = "A list of favourite countries...")

listbox = Listbox(top)

listbox.insert(1,"India")
listbox.insert(2, "USA")
listbox.insert(3, "Japan")
listbox.insert(4, "Austrelia")

lbl.pack()
listbox.pack()

top.mainloop()
```

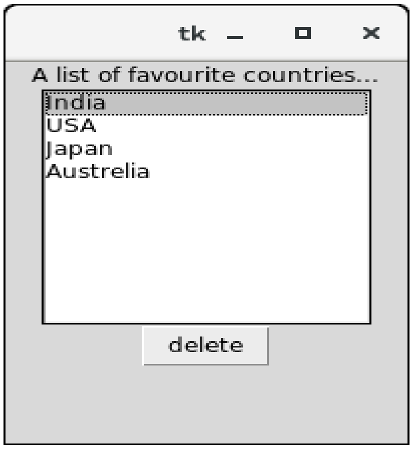
Output:



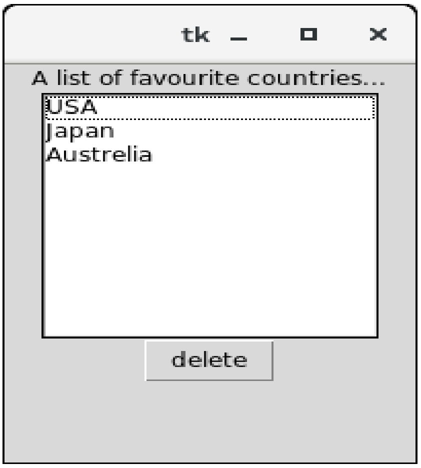
Example 2: Deleting the active items from the list

```
from tkinter import *

top = Tk()
top.geometry("200x250")

lbl = Label(top,text = "A list of favourite countries...")
listbox = Listbox(top)

listbox.insert(1,"India")
listbox.insert(2, "USA")
listbox.insert(3, "Japan")
listbox.insert(4, "Austrelia")

#this button will delete the selected item from the list

btn = Button(top, text = "delete", command = lambda listbox=listbox: listbox.delete
(ANCHOR))

lbl.pack()
listbox.pack()
btn.pack()
top.mainloop()
```

Output:



After pressing the delete button.



Python Tkinter Menu

The Menu widget is used to create various types of menus (top level, pull down, and pop up) in the python application.

The top-level menus are the one which is displayed just under the title bar of the parent window. We need to create a new instance of the Menu widget and add various commands to it by using the add() method.

The syntax to use the Menu widget is given below.

Syntax

w = Menu(top, options)

A list of possible options is given below.

SN	Option	Description
1	activebackground	The background color of the widget when the widget is under the focus.

2	activeborderwidth	The width of the border of the widget when it is under the mouse. The default is 1 pixel.
3	activeforeground	The font color of the widget when the widget has the focus.
4	Bg	The background color of the widget.
5	Bd	The border width of the widget.
6	cursor	The mouse pointer is changed to the cursor type when it hovers the widget. The cursor type can be set to arrow or dot.
7	disabledforeground	The font color of the widget when it is disabled.
8	Font	The font type of the text of the widget.
9	Fg	The foreground color of the widget.
10	postcommand	The postcommand can be set to any of the function which is called when the mourse hovers the menu.
11	relief	The type of the border of the widget. The default type is RAISED.
12	image	It is used to display an image on the menu.
13	selectcolor	The color used to display the checkbutton or radiobutton when they are selected.
14	tearoff	By default, the choices in the menu start taking place from position 1. If we set the tearoff = 1, then it will start taking place from 0th position.
15	Title	Set this option to the title of the window if you want to change the title of the window.

Methods

The Menu widget contains the following methods.

SN	Option	Description
1	add_command(options)	It is used to add the Menu items to the menu.
2	add_radiobutton(options)	This method adds the radiobutton to the menu.
3	add_checkbutton(options)	This method is used to add the checkbuttons to the menu.
4	add_cascade(options)	It is used to create a hierarchical menu to the parent menu by associating the given menu to the parent menu.
5	add_seperator()	It is used to add the seperator line to the menu.
6	add(type, options)	It is used to add the specific menu item to the menu.
7	delete(startindex, endindex)	It is used to delete the menu items exist in the specified range.
8	entryconfig(index,	It is used to configure a menu item identified by the given

	options)	index.
9	index(item)	It is used to get the index of the specified menu item.
10	insert_seperator(index)	It is used to insert a seperator at the specified index.
11	invoke(index)	It is used to invoke the associated with the choice given at the specified index.
12	type(index)	It is used to get the type of the choice specified by the index.

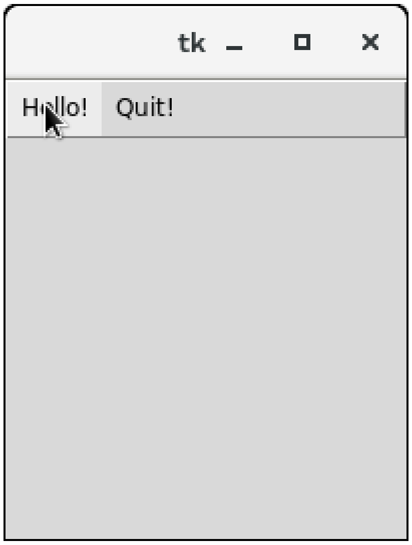
Creating a top level menu

A top-level menu can be created by instantiating the Menu widget and adding the menu items to the menu.

Example 1

```
from tkinter import *

top = Tk()

def hello():
   print("hello!")

# create a toplevel menu
menubar = Menu(top)
menubar.add_command(label="Hello!", command=hello)
menubar.add_command(label="Quit!", command=top.quit)

# display the menu
top.config(menu=menubar)

top.mainloop()
```

Output:



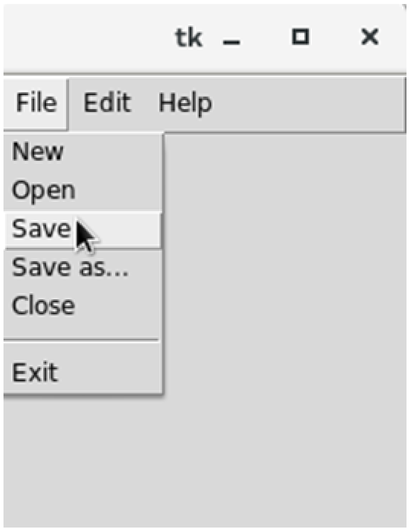
Clicking the hello Menubutton will print the hello on the console while clicking the Quit Menubutton will make an exit from the python application.

Example 2

```
from tkinter import Toplevel, Button, Tk, Menu

top = Tk()
menubar = Menu(top)
file = Menu(menubar, tearoff=0)
file.add_command(label="New")
file.add_command(label="Open")
file.add_command(label="Save")
file.add_command(label="Save as...")
file.add_command(label="Close")

file.add_separator()

file.add_command(label="Exit", command=top.quit)

menubar.add_cascade(label="File", menu=file)
edit = Menu(menubar, tearoff=0)
edit.add_command(label="Undo")

edit.add_separator()

edit.add_command(label="Cut")
edit.add_command(label="Copy")
edit.add_command(label="Paste")
edit.add_command(label="Delete")
edit.add_command(label="Select All")

menubar.add_cascade(label="Edit", menu=edit)
```

```
help = Menu(menubar, tearoff=0)
help.add_command(label="About")
menubar.add_cascade(label="Help", menu=help)

top.config(menu=menubar)
top.mainloop()
```

Output:



Python Tkinter Message

The Message widget is used to show the message to the user regarding the behaviour of the python application. The message widget shows the text messages to the user which can not be edited.

The message text contains more than one line. However, the message can only be shown in the single font.

The syntax to use the Message widget is given below.

Syntax

w = Message(parent, options)

A list of possible options is given below.

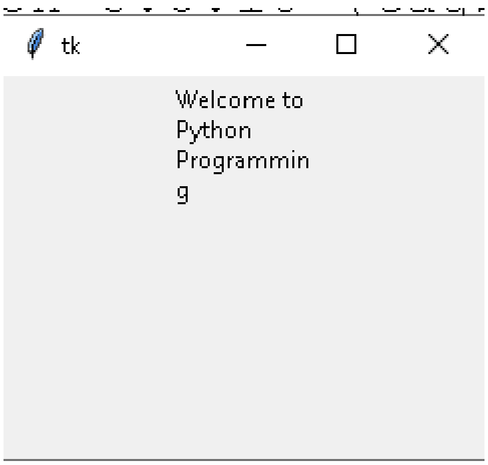
SN	Option	Description
1	Anchor	It is used to decide the exact position of the text within the space provided to the widget if the widget contains more space than the need of the text. The default is CENTER.
2	Bg	The background color of the widget.
3	Bitmap	It is used to display the graphics on the widget. It can be set to any graphical or image object.
4	Bd	It represents the size of the border in the pixel. The default size is 2 pixel.
5	Cursor	The mouse pointer is changed to the specified cursor type. The cursor

		type can be an arrow, dot, etc.
6	Font	The font type of the widget text.
7	Fg	The font color of the widget text.
8	Height	The vertical dimension of the message.
9	Image	We can set this option to a static image to show that onto the widget.
10	Justify	This option is used to specify the alignment of multiple line of code with respect to each other. The possible values can be LEFT (left alignment), CENTER (default), and RIGHT (right alignment).
11	Padx	The horizontal padding of the widget.
12	Pady	The vertical padding of the widget.
13	Relief	It represents the type of the border. The default type is FLAT.
14	Text	We can set this option to the string so that the widget can represent the specified text.
15	Textvariable	This is used to control the text represented by the widget. The textvariable can be set to the text that is shown in the widget.
16	Underline	The default value of this option is -1 that represents no underline. We can set this option to an existing number to specify that nth letter of the string will be underlined.
17	Width	It specifies the horizontal dimension of the widget in the number of characters (not pixel).
18	Wraplength	We can wrap the text to the number of lines by setting this option to the desired number so that each line contains only that number of characters.

Example

```
from tkinter import *

top = Tk()
top.geometry("100x100")

msg = Message( top, text = "Welcome to Python Programming")

msg.pack()
top.mainloop()
```

Output:



```
Welcome to
Python
Programmin
g
```

Python TkinterRadiobutton

The Radiobutton widget is used to implement one-of-many selection in the python application. It shows multiple choices to the user out of which, the user can select only one out of them. We can associate different methods with each of the radiobutton.

We can display the multiple line text or images on the radiobuttons. To keep track the user's selection the radiobutton, it is associated with a single variable. Each button displays a single value for that particular variable.

The syntax to use the Radiobutton is given below.

Syntax

w = Radiobutton(top, options)

SN	Option	Description
1	activebackground	The background color of the widget when it has the focus.
2	activeforeground	The font color of the widget text when it has the focus.
3	anchor	It represents the exact position of the text within the widget if the widget contains more space than the requirement of the text. The default value is CENTER.
4	Bg	The background color of the widget.
5	bitmap	It is used to display the graphics on the widget. It can be set to any graphical or image object.
6	borderwidth	It represents the size of the border.
7	command	This option is set to the procedure which must be called every-time when the state of the radiobutton is changed.
8	cursor	The mouse pointer is changed to the specified cursor type. It can be set to the arrow, dot, etc.
9	Font	It represents the font type of the widget text.
10	Fg	The normal foreground color of the widget text.

11	height	The vertical dimension of the widget. It is specified as the number of lines (not pixel).
12	highlightcolor	It represents the color of the focus highlight when the widget has the focus.
13	highlightbackground	The color of the focus highlight when the widget is not having the focus.
14	image	It can be set to an image object if we want to display an image on the radiobutton instead the text.
15	justify	It represents the justification of the multi-line text. It can be set to CENTER(default), LEFT, or RIGHT.
16	padx	The horizontal padding of the widget.
17	pady	The vertical padding of the widget.
18	relief	The type of the border. The default value is FLAT.
19	selectcolor	The color of the radio button when it is selected.
20	selectimage	The image to be displayed on the radiobutton when it is selected.
21	state	It represents the state of the radio button. The default state of the Radiobutton is NORMAL. However, we can set this to DISABLED to make the radiobutton unresponsive.
22	Text	The text to be displayed on the radiobutton.
23	textvariable	It is of String type that represents the text displayed by the widget.
24	underline	The default value of this option is -1, however, we can set this option to the number of character which is to be underlined.
25	value	The value of each radiobutton is assigned to the control variable when it is turned on by the user.
26	variable	It is the control variable which is used to keep track of the user's choices. It is shared among all the radiobuttons.
27	width	The horizontal dimension of the widget. It is represented as the number of characters.

28	wraplength	We can wrap the text to the number of lines by setting this option to the desired number so that each line contains only that number of characters.

Methods

The radiobutton widget provides the following methods.

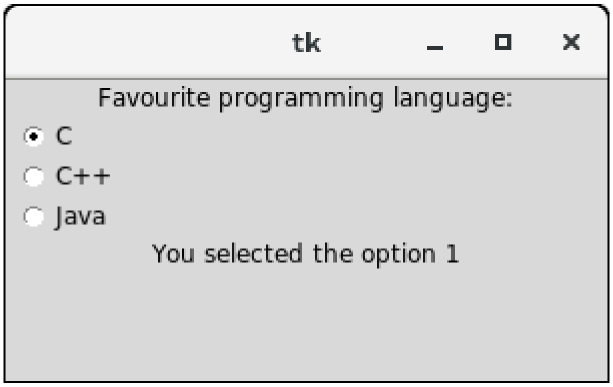
SN	Method	Description
1	deselect()	It is used to turn of the radiobutton.
2	flash()	It is used to flash the radiobutton between its active and normal colors few times.
3	invoke()	It is used to call any procedure associated when the state of a Radiobutton is changed.
4	select()	It is used to select the radiobutton.

Example

```
from tkinter import *

def selection():
   selection = "You selected the option " + str(radio.get())
   label.config(text = selection)

top = Tk()
top.geometry("300x150")
radio = IntVar()
lbl = Label(text = "Favourite programming language:")
lbl.pack()
R1 = Radiobutton(top, text="C", variable=radio, value=1,command=selection)
R1.pack( anchor = W )

R2 = Radiobutton(top, text="C++", variable=radio, value=2, command=selection)

R2.pack( anchor = W )

R3 = Radiobutton(top, text="Java", variable=radio, value=3,  command=selection)

R3.pack( anchor = W)
```

```
label = Label(top)
label.pack()
top.mainloop()
```

Output:



tk

Favourite programming language:

● C

○ C++

○ Java

You selected the option 1

Python Tkinter Scrollbar

The scrollbar widget is used to scroll down the content of the other widgets like listbox, text, and canvas. However, we can also create the horizontal scrollbars to the Entry widget.

The syntax to use the Scrollbar widget is given below.

Syntax

```
w = Scrollbar(top, options)
```

A list of possible options is given below.

SN	Option	Description
1	activebackground	The background color of the widget when it has the focus.
2	Bg	The background color of the widget.
3	Bd	The border width of the widget.
4	command	It can be set to the procedure associated with the list which can be called each time when the scrollbar is moved.
5	cursor	The mouse pointer is changed to the cursor type set to this option which can be an arrow, dot, etc.
6	elementborderwidth	It represents the border width around the arrow heads and slider. The default value is -1.
7	Highlightbackground	The focus highlighcolor when the widget doesn't have the focus.
8	highlighcolor	The focus highlighcolor when the widget has the focus.
9	highlightthickness	It represents the thickness of the focus highlight.
10	jump	It is used to control the behavior of the scroll jump. If it set to 1,

		then the callback is called when the user releases the mouse button.
11	orient	It can be set to HORIZONTAL or VERTICAL depending upon the orientation of the scrollbar.
12	repeatdelay	This option tells the duration up to which the button is to be pressed before the slider starts moving in that direction repeatedly. The default is 300 ms.
13	repeatinterval	The default value of the repeat interval is 100.
14	takefocus	We can tab the focus through this widget by default. We can set this option to 0 if we don't want this behavior.
15	troughcolor	It represents the color of the trough.
16	width	It represents the width of the scrollbar.

Methods

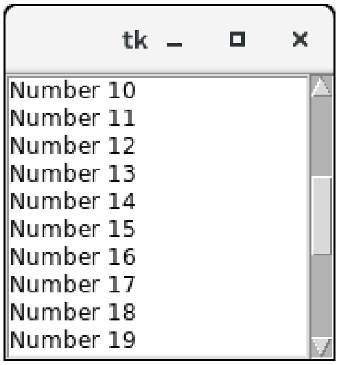
The widget provides the following methods.

SN	Method	Description
1	get()	It returns the two numbers a and b which represents the current position of the scrollbar.
2	set(first, last)	It is used to connect the scrollbar to the other widget w. The yscrollcommand or xscrollcommand of the other widget to this method.

Example

```
from tkinter import *

top = Tk()
sb = Scrollbar(top)
sb.pack(side = RIGHT, fill = Y)

mylist = Listbox(top, yscrollcommand = sb.set )

for line in range(30):
   mylist.insert(END, "Number " + str(line))

mylist.pack( side = LEFT )
sb.config( command = mylist.yview )

mainloop()
```

Output:



```
Number 10
Number 11
Number 12
Number 13
Number 14
Number 15
Number 16
Number 17
Number 18
Number 19
```

Python Tkinter Text

The Text widget is used to show the text data on the Python application. However, Tkinter provides us the Entry widget which is used to implement the single line text box.

The Text widget is used to display the multi-line formatted text with various styles and attributes. The Text widget is mostly used to provide the text editor to the user.

The Text widget also facilitates us to use the marks and tabs to locate the specific sections of the Text. We can also use the windows and images with the Text as it can also be used to display the formatted text.
The syntax to use the Text widget is given below.

Syntax

w = Text(top, options)

A list of possible options that can be used with the Text widget is given below.

SN	Option	Description
1	Bg	The background color of the widget.
2	Bd	It represents the border width of the widget.
3	Cursor	The mouse pointer is changed to the specified cursor type, i.e. arrow, dot, etc.
4	Exportselection	The selected text is exported to the selection in the window manager. We can set this to 0 if we don't want the text to be exported.
5	Font	The font type of the text.
6	Fg	The text color of the widget.
7	Height	The vertical dimension of the widget in lines.
8	highlightbackground	The highlightcolor when the widget doesn't has the focus.
9	highlightthickness	The thickness of the focus highlight. The default value is 1.
10	Highlighcolor	The color of the focus highlight when the widget has the focus.

11	Insertbackground	It represents the color of the insertion cursor.
12	insertborderwidth	It represents the width of the border around the cursor. The default is 0.
13	Insertofftime	The time amount in Milliseconds during which the insertion cursor is off in the blink cycle.
14	Insertontime	The time amount in Milliseconds during which the insertion cursor is on in the blink cycle.
15	Insertwidth	It represents the width of the insertion cursor.
16	Padx	The horizontal padding of the widget.
17	Pady	The vertical padding of the widget.
18	Relief	The type of the border. The default is SUNKEN.
19	Selectbackground	The background color of the selected text.
20	selectborderwidth	The width of the border around the selected text.
21	spacing1	It specifies the amount of vertical space given above each line of the text. The default is 0.
22	spacing2	This option specifies how much extra vertical space to add between displayed lines of text when a logical line wraps. The default is 0.
23	spacing3	It specifies the amount of vertical space to insert below each line of the text.
24	State	It the state is set to DISABLED, the widget becomes unresponsive to the mouse and keyboard unresponsive.
25	Tabs	This option controls how the tab character is used to position the text.
26	Width	It represents the width of the widget in characters.
27	Wrap	This option is used to wrap the wider lines into multiple lines. Set this option to the WORD to wrap the lines after the word that fit into the available space. The default value is CHAR which breaks the line which gets too wider at any character.
28	Xscrollcommand	To make the Text widget horizontally scrollable, we can set this option to the set() method of Scrollbar widget.
29	Yscrollcommand	To make the Text widget vertically scrollable, we can set this option to the set() method of Scrollbar widget.

Methods

We can use the following methods with the Text widget.

SN	Method	Description
1	delete(startindex, endindex)	This method is used to delete the characters of the specified range.
2	get(startindex, endindex)	It returns the characters present in the specified range.
3	index(index)	It is used to get the absolute index of the specified index.
4	insert(index, string)	It is used to insert the specified string at the given index.
5	see(index)	It returns a boolean value true or false depending upon whether the text at the specified index is visible or not.

Mark handling methods

Marks are used to bookmark the specified position between the characters of the associated text.

SN	Method	Description
1	index(mark)	It is used to get the index of the specified mark.
2	mark_gravity(mark, gravity)	It is used to get the gravity of the given mark.
3	mark_names()	It is used to get all the marks present in the Text widget.
4	mark_set(mark, index)	It is used to inform a new position of the given mark.
5	mark_unset(mark)	It is used to remove the given mark from the text.

Tag handling methods

The tags are the names given to the separate areas of the text. The tags are used to configure the different areas of the text separately. The list of tag-handling methods along with the description is given below.

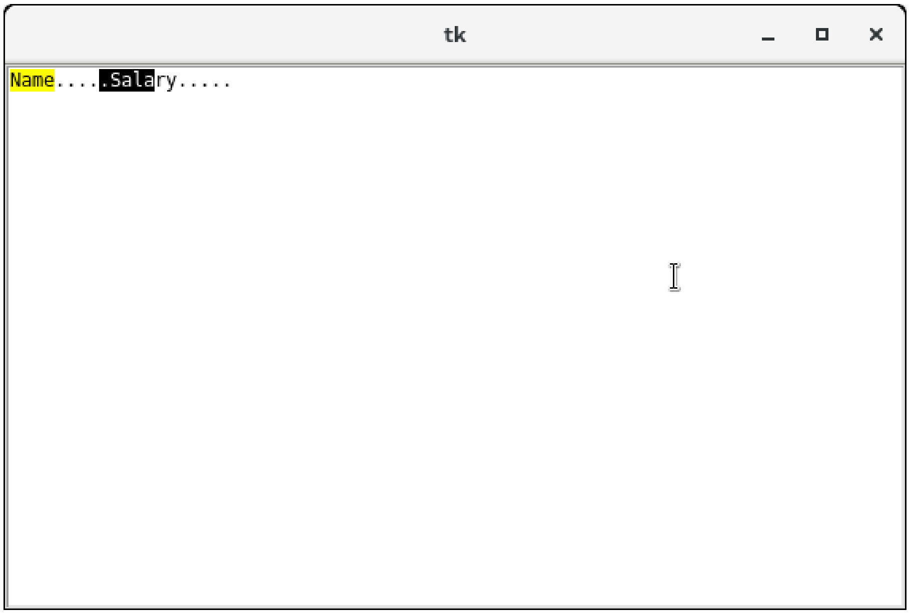
SN	Method	Description
1	tag_add(tagname, startindex, endindex)	This method is used to tag the string present in the specified range.
2	tag_config	This method is used to configure the tag properties.
3	tag_delete(tagname)	This method is used to delete a given tag.
4	tag_remove(tagname, startindex, endindex)	This method is used to remove a tag from the specified range.

Example

```
        from tkinter import *
```

```
top = Tk()
text = Text(top)
text.insert(INSERT, "Name.....")
text.insert(END, "Salary.....")

text.pack()

text.tag_add("Write Here", "1.0", "1.4")
text.tag_add("Click Here", "1.8", "1.13")

text.tag_config("Write Here", background="yellow", foreground="black")
text.tag_config("Click Here", background="black", foreground="white")

top.mainloop()
```

Output:



```
Name.....Salary.....
```

Other GUIs

General GUI development using many of the abundant number of graphical toolkits that exist under Python, but alas, that is for the future. As a proxy, we would like to present a single simple GUI application written using four of the more popular and available toolkits out there: Tix (Tk Interface eXtensions), Pmw (Python MegaWidgetsTkinter extension), wxPython (Python binding to wxWidgets), and PyGTK (Python binding to GTK+).

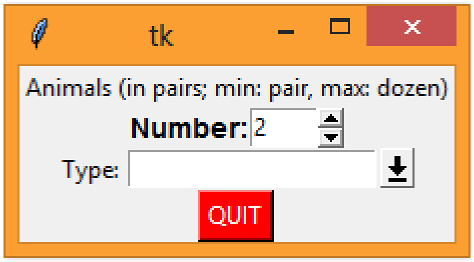
Tix, the Tk Interface eXtension, is a powerful set of user interface components that expands the capabilities of your Tcl/Tk and Python applications. Using Tix together with Tk will greatly enhance the appearance and functionality of your application.

It uses the Tix module. Tix comes with Python!

Example:

```
from tkinter import *
from tkinter.tix import Control, ComboBox

top = Tk()
top.tk.eval('package require Tix')

lb = Label(top,text='Animals (in pairs; min: pair, max: dozen)')
lb.pack()

ct = Control(top, label='Number:',integer=True, max=12, min=2, value=2, step=2)
ct.label.config(font='Helvetica -14 bold')
ct.pack()

cb = ComboBox(top, label='Type:', editable=True)
for animal in ('dog', 'cat', 'hamster', 'python'):
cb.insert(END, animal)
cb.pack()

qb = Button(top, text='QUIT',command=top.quit, bg='red', fg='white')
qb.pack()

top.mainloop()
```

Output:



Animals (in pairs; min: pair, max: dozen)

Number: 2

Type:

Python Mega Widgets

PMW (Python Mega Widgets) is a toolkit for building high -level widgets in Python using the Tkinter module. This toolkit provides a frame work that contains a variety of widgets richer than the one provided by Tkinter.

It basically helps the extend its longevity by adding more modern widgets to the GUI Palette.This package is 100% written in Python, which turns out to be a cross -platform widget library. Being highly configurable allows it to create additional widget collections by extending the basic Tkinter widget core set.

PMW provides many interesting and complex widgets, including: About Dialog, Balloon, Button Box, Combo Box, Combo Box Dialog, Counter, CounterDialog, Dialog, Entry Field, Group, Labeled Widget, MenuBar, Message Bar, Message Dialog, Note BookR, Note BookS, Note Book, Option Menu, Paned Widget, Prompt Dialog, RadioSelect, Scrolled Canvas, Scrolled Field, ScrolledFrame, Scrolled Listbox, ScrolledText, Selection Dialog, Text Dialog, and Time Counter.

Example:

```
from Tkinter import Button, END, Label, W
from Pmw import initialise, ComboBox, Counter

top = initialise()

lb = Label(top,
text='Animals (in pairs; min: pair, max: dozen)')
lb.pack()

ct = Counter(top, labelpos=W, label_text='Number:',
datatype='integer', entryfield_value=2,
increment=2, entryfield_validate={'validator':'integer', 'min': 2, 'max': 12})
ct.pack()
cb = ComboBox(top, labelpos=W, label_text='Type:')
for animal in ('dog', 'cat', 'hamster', 'python'):
cb.insert(end, animal)
cb.pack()

qb = Button(top, text='QUIT',
command=top.quit, bg='red', fg='white')
qb.pack()
```

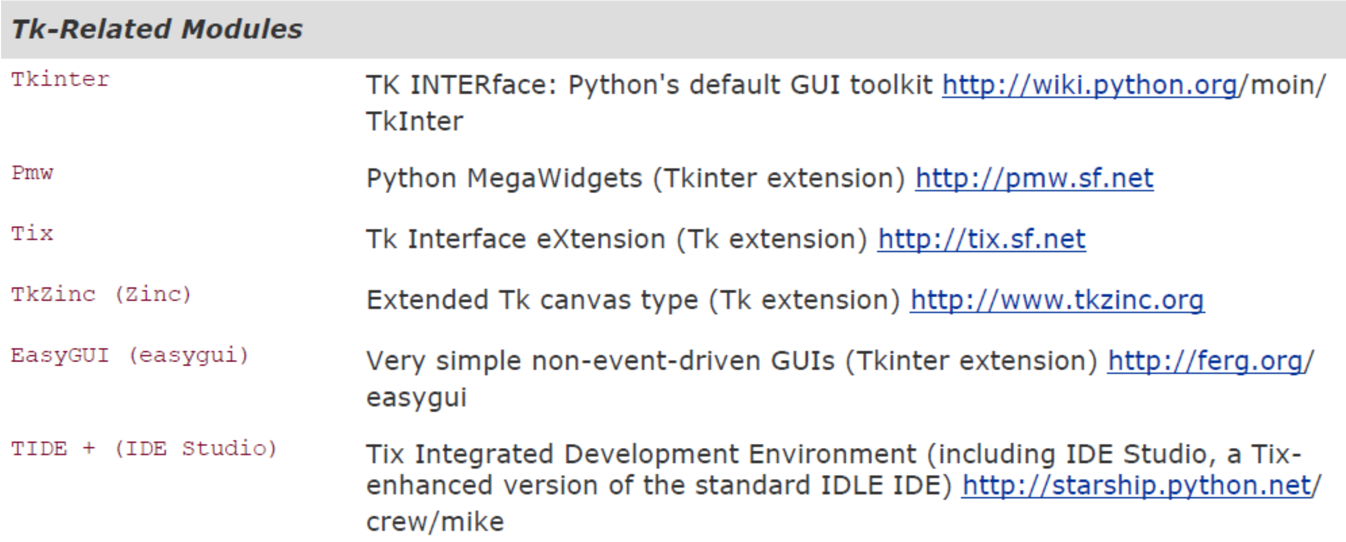
wxWidgets and wxPython

wxWidgets (formerly known as wxWindows) is a cross-platform toolkit used to build graphical user applications. It is implemented using C++ and is available on a wide number of platforms to which wxWidgets defines a consistent and common API. The best part of all is that wxWidgets uses the native GUI on each platform, so your program will have the same ook-and-feel as all the other applications on your desktop. Another feature is that you are not restricted to developing wxWidgets applications in C++. There are interfaces to both Python and Perl.

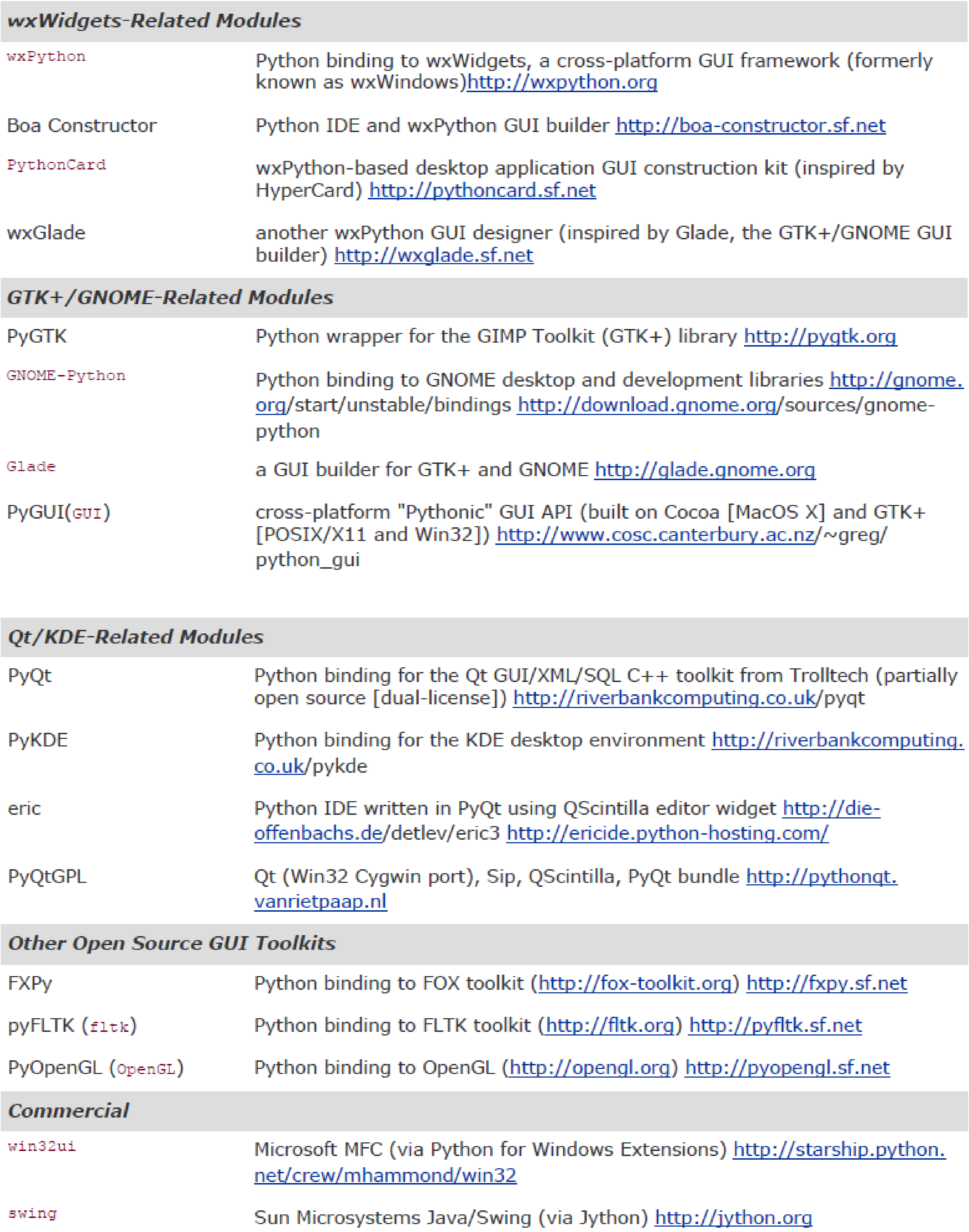
Related Modules and Other GUIs

There are other GUI development systems that can be used with Python. We present the appropriate modules along with their corresponding window systems. Table represents the GUI Systems Available for Python

GUI Module or System Description



Tk-Related Modules	
Tkinter	TK INTERface: Python's default GUI toolkit http://wiki.python.org/moin/TkInter
Pmw	Python MegaWidgets (Tkinter extension) http://pmw.sf.net
Tix	Tk Interface eXtension (Tk extension) http://tix.sf.net
TkZinc (Zinc)	Extended Tk canvas type (Tk extension) http://www.tkzinc.org
EasyGUI (easygui)	Very simple non-event-driven GUIs (Tkinter extension) http://ferg.org/easygui
TIDE + (IDE Studio)	Tix Integrated Development Environment (including IDE Studio, a Tix-enhanced version of the standard IDLE IDE) http://starship.python.net/crew/mike



wxWidgets-Related Modules	
wxPython	Python binding to wxWidgets, a cross-platform GUI framework (formerly known as wxWindows)http://wxpython.org
Boa Constructor	Python IDE and wxPython GUI builder http://boa-constructor.sf.net
PythonCard	wxPython-based desktop application GUI construction kit (inspired by HyperCard) http://pythoncard.sf.net
wxGlade	another wxPython GUI designer (inspired by Glade, the GTK+/GNOME GUI builder) http://wxglade.sf.net
GTK+/GNOME-Related Modules	
PyGTK	Python wrapper for the GIMP Toolkit (GTK+) library http://pygtk.org
GNOME-Python	Python binding to GNOME desktop and development libraries http://gnome.org/start/unstable/bindings http://download.gnome.org/sources/gnome-python
Glade	a GUI builder for GTK+ and GNOME http://glade.gnome.org
PyGUI(GUI)	cross-platform "Pythonic" GUI API (built on Cocoa [MacOS X] and GTK+ [POSIX/X11 and Win32]) http://www.cosc.canterbury.ac.nz/~greg/python_gui
Qt/KDE-Related Modules	
PyQt	Python binding for the Qt GUI/XML/SQL C++ toolkit from Trolltech (partially open source [dual-license]) http://riverbankcomputing.co.uk/pyqt
PyKDE	Python binding for the KDE desktop environment http://riverbankcomputing.co.uk/pykde
eric	Python IDE written in PyQt using QScintilla editor widget http://die-offenbachs.de/detlev/eric3 http://ericide.python-hosting.com/
PyQtGPL	Qt (Win32 Cygwin port), Sip, QScintilla, PyQt bundle http://pythonqt.vanrietpaap.nl
Other Open Source GUI Toolkits	
FXPy	Python binding to FOX toolkit (http://fox-toolkit.org) http://fxpy.sf.net
pyFLTK (fltk)	Python binding to FLTK toolkit (http://fltk.org) http://pyfltk.sf.net
PyOpenGL (OpenGL)	Python binding to OpenGL (http://opengl.org) http://pyopengl.sf.net
Commercial	
win32ui	Microsoft MFC (via Python for Windows Extensions) http://starship.python.net/crew/mhammond/win32
swing	Sun Microsystems Java/Swing (via Jython) http://jython.org

MODULE – V

Agenda:

- **Web Programming: Introduction,**
- **Wed Surfing with Python,**
- **Creating Simple Web Clients,**
- **Advanced Web Clients,**
- **CGI-Helping Servers Process Client Data,**
- **Building CGI Application Advanced CGI,**
- **Web (HTTP) Servers**

Web Surfing:

In Client/Sewer Computing In client/server computing web clients are browsers i.e., the applications that allow the users to look for the documents on worldwide web.

The web servers are the processes which run on information provider host computers. The client's send requests to server which processes those requests and returns the data.

Client

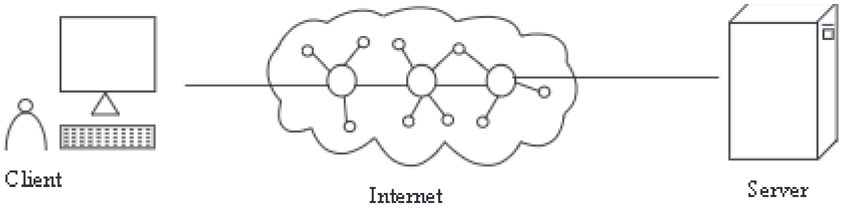


Figure: Web Surfing

The above figure depicts web surfing where a user runs a web client program like browser and establishes connection to web server on internet to obtain information The requests of clients include form submission, page request or information display. These requests are serviced by the server. The communication protocol that is used by web servers and clients is HTTP. It is a stateless protocol that does not keep track of inform ton from one client request to next It handles even/ request as separate service request including the new requests. Internet Internet is simply referred to a collection of different physical networks such as LAN s, MANs and WANs. These networks are connected with each other in order to transmit the data from a computer on one network to computer on another network. In simple terms it is a network of networks i.e., collection of two or m are networks connected to each other with the help of widely available internet working devices such as router; gateway; bridges etc. The architecture of internet completely depends upon standard TCP/ IP and is designed to connect any the network; irrespective of difference in software, hardware and technical design.

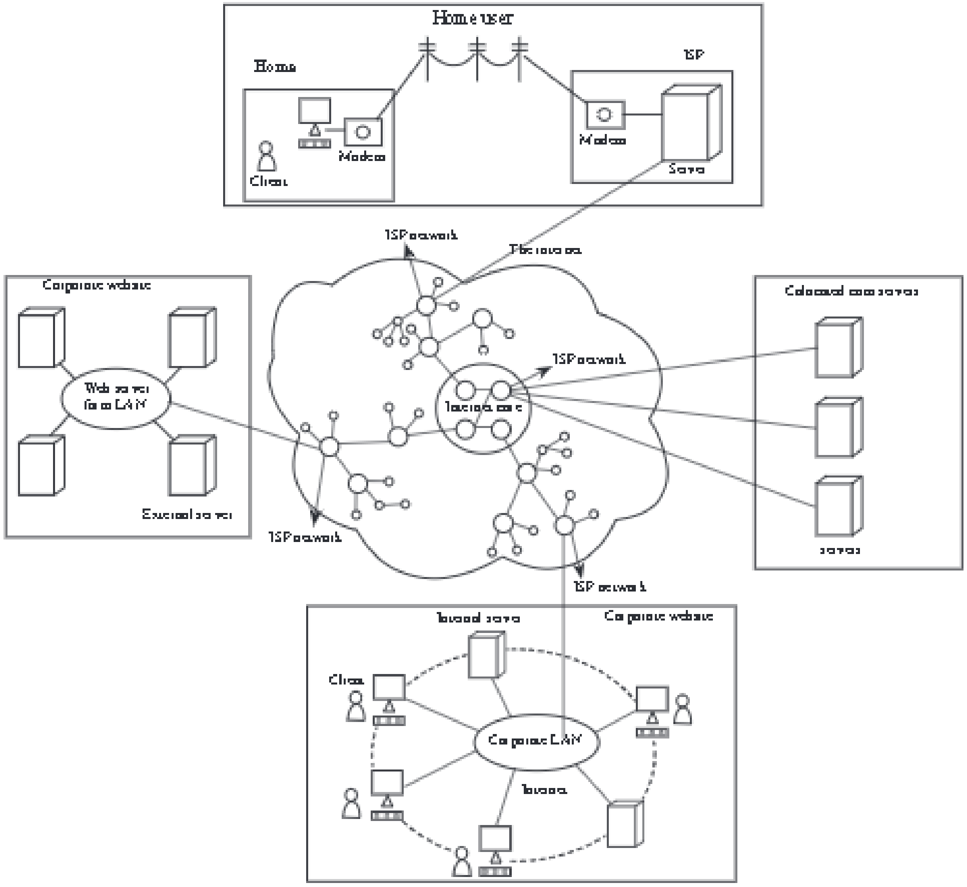


Figure: Detailed View of Internet