UNIT-4

Data Structures

python data structures is classified into two catagories

1. Mutable (changes)
2. Immutable (doesn't change)

Mutable:
- list
- Dictionary
- set

Immutable:
- string
- int
- float
- complex
- bool
- tuple
- frozenset

List:

It's a python data structure. It contains heterogenous elements i.e., seperated by coma. Those elements are enclosed within "[ ]"

In python, the "list" is a class. It contains methods and functions.

Ex: a = [1,2,3,4,5]

a is nothing but list object

```
a[0 1 2 3 4]  ⇒ a[0] = 1
 1,2,3,4,5      a[-1] = 5
-5-4-3-2-1
```

a[1:3]

o/p: [2,3]

type(a)

⇒ It prints class, list

In mutable, we can change the index

UNIT-4

Data Structures

python data structures is classified into two catagories

1. Mutable (changes)
2. Immutable (doesn't change)

Mutable:
- list
- Dictionary
- set

Immutable:
- string
- int
- float
- complex
- bool
- tuple
- frozenset

List:

It's a python data structure. It contains heterogenous elements i.e., seperated by coma. These elements are enclosed within "[ ]"

In python, the "list" is a class. It contains methods and functions.

Ex: a = [1,2,3,4,5]

a is nothing but list object

```
a[0 1 2 3 4]  ⇒ a[0] = 1
 [1,2,3,4,5]     a[-1] = 5
 -5-4-3-2-1
```

a[1:3]

o/p: [2,3]

type(a)

⇒ It prints class, list

In mutable, we can change the index

a[0] = 1

a = [1, 2, 3, 4, 5]

a[0] = 9

here, 9 is assigned to 0th location

∴ a[0] = 9

a = [9, 2, 3, 4, 5]

List of list:

a = [1, 2, 3, [4, 5, 6]]

a[0] = 1 , a[1] = 2, a[2] = 3

a[3][0] = 4, a[3][1] = 5, a[3][2] = 6

a = [1, 2, 3, [4, 5, 6], 44.5, 'Va

a[4] = 44.5

a[5] = Vasavi

a[5][2] = s (∵ s in va

List functions:-

1. len()
2. max()
3. min()
4. list()

len():

It's a list function. It is counting number of elemen in the list

Syntax: len(list_of_objects)

Ex: a = [1, 2, 3, 4, 5]

len(a) = 5

max()

Maximum element should be identified. Either in ordered or inordered.

a = [1, 2, 3, 4, 5]

max(a) = 5

min(): to find the minimum element.

a = [1, 2, 3 4, 5]

min(a) = 1

list()

It is a function that is converted into list object

list(a)

List methods:

* append()
* insert()
* extend()
* sort()
* reverse()
* count()
* remove()
* pop()
* clear()
* copy

append():

Adding the element at the end of list

```
a = [1,2,3,4,5]
a.append(6)
print(a)
```

* Append is a list method. It is adding new element at the end of the list.

class → method in the class

O/P:

1, 2, 3, 4, 5, 6

Syntax: append(element)

↓

either int, float, char

insert():

In this method, we replaces the element in the specific index.

Syntax: insert(index, element)

Ex: a = [1, 2, 3, 4]

0 1 2 3

a.insert(1,5)

O/P (1, 5, 3, 4)

extend():

In this method we used adding group of elements original list.

Syntax: extend(new_elements)

Ex: a = [1, 2, 3, 4]

a.extend(5, 6)

o/p [1, 2, 3, 4, 5, 6)

a.extent([5, 6, 7])

o/p [1, 2, 3, 4, [5, 6, 7]]

Sort():

By following "Quick sort" (mechanism)

* sort is a method i.e., used to arranging the list elements in ascending order by default
* If you arranging descending order, in sort parameter "reverse = True"

Syntax: sort(key = None, reverse = False)

↓

Any type of function

by default it is arrange in ascending order

Ex: a = [7, 6, 3, 2, 1, 5, 4]

a.sort()

o/p: [1, 2, 3, 4, 5, 6, 7]

for descending order;

a.sort(reverse = True)

o/p: [7, 6, 5, 4, 3, 2, 1]

reverse()

Arranging the order of list in reverse order (existing element)

⇒ reverse() used to existing list display in the reverse order

Ex: a = [5, 3, 1, 6, 7, 2]
a.reverse()

o/p: [2, 7, 6, 1, 3, 5]

count():

count is a list method i.e, used to counting frequency of element

Ex: a = [5, 3, 1, 2, 1, 6]

a.count(1) o/p 2

a.count(2) o/p 1

a = ['a', 'b', 'a']
a.count('a')
o/p: 2

remove():

used to remove (delete) the specific element in the list

Syntax: remove(element)

Ex: a = [5, 3, 1, 2, 1]
a.remove(3)
o/p [5, 1, 2, 1]

clear():

clear method is used to delete all the elements in the list.

Ex: a = [5, 6, 1, 3, 2]
a.clear()

pop():

pop is a method i.e, used to delete 'top' element

a = [5, 3, 1, 6, 7, 2, 1]
a.pop()

Ex
o/p: [5, 3, 1, 6, 7, 2] # last element in list

Copy:

Copy method in list contains two methods catagories

1. shallow copy
2. Deepcopy

Both copy methods the lists are stored in different memory locations.

In shallow copy the list is ~~data~~ updated automatically the parent list also updated. That means every element is refference of the parent element.

Whereas, Deepcopy does not change parent list

These copy catagories are implement using copy package i.e import copy

The shallow copy directly implement without using Copy package..

Ex:
```
a=[1,2,3,4,5]
b=a  #shallow copy
b[2]=99
print(b)
print(a)
```

o/p:
```
[1,2,99,4,5]
[1,2,99,4,5]
```

with using package

```
a=[1,2,3,4,5]
b=copy.copy(a)
b[2]=99
print(a)
print(b)
```

o/p
```
[1 2 99 4 5]
[1 2 99 4 5]
```

```
import copy
a = [1,2,3,4,5]
b = copy.deepcopy(a) #deepcopy
b[2] = 100
print(a)
print(b)
```

o/p [1 2 3 4 5]
[1 2 100 4 5]

List comprehension:

List comprehension is a method that is written in a single line and that is enclosed with in square brackets.

Example: Generate 1 to 10 numbers using

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

o/p: [1, 2, 3, 4 . . . . 10]

example 2: Generate squares of 1 to 10 numbers

```
a = [x**2 for x in range(1, 11)]
print(a)
```

o/p: [1, 4, 9, 16, 25 . . . . , 100]

print upto 1-5 numbers

```
a = [x for x in range(1,11) if x<=5]
```

```
a = [1, 2, 3, 4, 5]
b = len(a)
t = a[0]
a[0] = a[b-i]
a[b-i] = t
print(a)
```

```
a = [1, 2, 3, 4, 5]
a[0], a[-i] = a[-i], a[0]
```

o/p: [5, 2, 3, 4, 1]

Write a python program to find the sum of list elements

```
a = [1, 2, 3, 4, 5]
s = 0
for x in a:
    s += x
print(s)
```

o/p: 15

write a python program implement function. for list of even numbers

```
def even(a):
    e = []
    for i in a:
        if a[i] % 2 == 0:
            e.append(a[i])
    return e
print(even[1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

o/p: [2, 4, 6, 8, 10]

write a python program to implement palindrom function in the given list.

```
def newpalindrom(a):
    e=a[]
    for i in a:
        if i == i.reverse():
            e.append(a)
    return e
print(palindrome['vasavi', 'madam', 'TPG', 'krishna',
      malayalam', 'Babu'])
```

o/p:

madam, malayalam

Factory functions: (functional programming)

1. Map:

map is a factory function. The map function is iterable sequence generating that is return map object. This map object is converting to list, touple, set etc

Syntax: map(func, *iterables)

where func is called user defined function or built in function.

* represents multiple elements

iterable returns depends on the function

Ex: ① a=[1,2,3,4,5]
list(map(lambda x: x**2, a))

o/p: [1,4,9,16,25]

② x=['a', 'b', 'c', 'd']
list(map(str.upper, x))

o/p: ['A', 'B', 'C', 'D']

```
a = [1, 2, 3, 4]
b = [5, 6, 7, 8]
List(map( lambda x, y : x+y , a, b))
```

o/p

[6, 8, 10, 12]

(or)

```
def sum(x, y):
    return x+y
list(map(sum, a, b))
```

o/p:

[6, 8, 10, 12]

filter():

filter is a functional programming method it is written true iterable elements

Syntax: filter(func, iterable)

where, function parameter either uses defined function, Anonymous function and built in function

Iterable is a objects of list, touple, set.

Example

```
def myage(x):
    return x<20
age = [15, 25, 35, 19, 16, 44, 32]
List(filter(myage, age))
```

o/p: 15, 19, 16

Print the list of palindrom elements.

```
p = ['rama', 'Babu', 'madam', 'krishna', 'Arun',
                'Malayalam']

list(filter(lambda x: x==x[::-1], p))
```

```
def pal(x):
    return x==x[::-1]
p = ['rama', 'Babu', 'madam', 'krishna', 'Arun', 'Malayalam']
list(filter(pal, p))
```

*

```
a = []
for i in range(1,11):
    ~~a = int(input())~~
    a.append(i)
list(filter(lambda x: x<=5, a))
```

(or)

*

```
a = list(range(1,11)) // dynamic process
list(filter(lambda x: x<=5, a))
```

*

```
a = list(range(1,11))
~~list(filter(lambda x: x<~~
list(map(~~lambda x: x<=5~~ x**2, a))
list(map(lambda x: x**2, filter(lambda x: x<=5, a)))
```

o/p: [1, 4, 9, 16, 25]

```
a = list(range(1, 6))
list(map(lambda x: x**2, a))
```

reduce():

Syntax: reduce(func, iterable, [initial])

Example

```
def mysum(x):
    s = 0
    for i in range(len(x)):
        s += x[i]
    return a + b
a = [1, 2, 3, 4, 5]
import functools (or) from functools import reduce
functools.reduce(mysum, a)
result = reduce(mysum, a)
print(result)
```

reduce() is a function tools method. This function is used to iterable of the sequences is reduced based on function that is given to the some value.

The first parameter func is user defined or built in function where iterable is any sequence (list, touple etc) The third parameter initial is optional this function used before import the functools package.

Ex: import functools
(or)
from functools import reduce

zip():

zip is a function it is return to the touple of list

Syntax: zip(*iterable)

Example:

```
a = [1, 2, 3, 4]
b = ['a', 'b', 'c', 'd', 'e']
list(zip(a, b))
```

o/p: [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

Example:

```
a = [1, 2, 3, 4]
b = ["vasavi", 'rama']
list(zip(a, b))
```

o/p: [(1, 'vasavi'), (2, 'rama')]

Develop a function nearly equal to test whether two strings a, b are nearly equal when 'a' can be generated by a single mutation on 'b' :

dictionary:

dictionary is a mutable data structure. It contains unordered elements Combination of key, value pairs. These elements are enclosed with in curly braces {} the empty dictionary is defined

dictionary_object = { }

Syntax: dict_obj = {key1 : value1 , key2 : value2 ....}

The above syntax keys and values are different type data elements.

The dictionary does not support slicing.

In dictionary extracting values using keys

Ex: d = {'a' : 1, 'b' : 55, 'c' : 3}

print(d['c'])

O/P 3

In dictionary the basic functionalities are

len()

dict()

Methods are

copy()

get()

items()

values()

keys()

update()

sorted()

clear()

keyfrom()

copy()

copy method is used to shallow copy

get()

This method is used to extracting values or keys

Syntax: ~~dict_object(key/value)~~

dict_object.get(key/value)

items()

This method is used to extracting list of touples from the dictionary

Syntax: dict_object.items()

Ex: list(d.items())

o/p: [('a',1),('b',55),('c',3)]

keys():

keys is a dictionary method it is return the list of keys in dictionary.

Ex: d = {'a':1,'b':2,'c':3}

list(d.keys())

['a','b','c']

syntax: dict_ob.keys()

values:

This method return the list of values in dictionary

Syntax: dict_obj.values()

d.values()

[1,2,3]

update():

This method is used to add a new dictionary with updated values of the old dictionary

Syntax: olddict_obj.update(newdict_obj)

Ex: ① d1={'a':1, 'b':2, 'c':3}

d2={'d':4, 'b':100, 'e':5}

d1.update(d2)

o/p: {'a':1, 'b':100, 'c':3, 'd':4, 'e':5}

②

d1.update({'d':4, 'e':5})

o/p: {'a':1, 'b':2, 'c':3, 'd':4, 'e':5}

clear():

This method is used to delete all the elements in dictionary

→ delete dictionary object we use 'del' keyword Ex: del d

pop():

This method is used to delete a specific element in the dictionary using key

Syntax: dict_obj.pop(key)

Example: d.pop('b')

print(d)

o/p: {'a':1, 'c':3}

popItem()
This method is used to delete top element in the dictionary

keyfrom()
This method is used to create a new dictionary using sequence of list

Syntax: dict_obj.keyfrom(sequence, [initial value])

Here initial values are optional

Ex: 1. Mydick.keyfrom([1,2,3])

o/p: {1:None, 2:None, 3:None}

2. mydict.keyfrom([1,2,3], 10)

o/p: {1:10, 2:10, 3:10}

d={'a':1, 'b':5, 'c':[1,2,3,4], 'name':['ram','krishna','kasavi']}

List(d.items())[3][1][1]

o/p: krishna

Modules:

* Module is a python script file.
* It ~~contains~~ implements different type of functions
* This modules access using import keyword

Syntax: import modulename

Packages is a collection of modules. These packages are implemented following steps

Step 1: Create a folder (name as package)

step 2: In this folder create another folder (name as mypack)

Step 3: In mypack folder contains following python script

The first script name as __init__.py file It is empty script file. This is used to import your package to other directories

Another script create module (name as first.py) In this module contains different type of user defined functions.

Step 4: Create a new script name as test.py in this script Contains access your package as following

import Mypack

Note:

This package only access with the package folder

sets:

set is a unordered elements it is access only atomic elements. This set is created within curly braces {" { } "

Concept vice set is immutable but itself it is mutable

Ex: ① s = { } → Here type(s) is <class 'dictionary'>

After initializing values like

s = {1,2,3} → Here type(s) is <class 'set'>

② s = {1,2,3,4,1,2,3} (Here duplicate elements are ignored)
print(s)

O/P: {1,2,3,4}

set functions:

len()
min()
max()
set()

set Methods:

copy():

copy method is used to shallow copy

Ex: s = {1,2,3,4}
s1 = s

add(): This method is used to add a single element into the set

Syntax: set_obj.add(element)

Ex: ① s = {1,2,3,4}
s.add(5)
print(s)

O/P: {1,2,3,4,5}

② s = {1,2,3,4}
s.add(3)
print(s)

O/P: {1,2,3,4}

remove()

This method is used to remove a single element from the set.

Syntax: set_obj.remove(element)

Ex: ① s = {1,2,3,4}
s.remove(3)

o/p: {1,2,4}

② s = {1,2,3,4}
s.remove(5)

o/p: key value error occurs

discard()

discard() is similar to the remove(). If we want to delete a unknown element of the set discard is not respond whereas remove raise an error i.e key value error

Syntax: set_obj.discard(element)

Example: s = {1,2,3,4}
s.discard(33)
print(s)

o/p:
{1,2,3,4}

pop()

This method is used to remove random element in the set

Syntax: set_obj.pop()

Example: s = {1,2,3,4}
print(s.pop())

o/p: 1

clear():

Delete all elements in the set

syntax: set_obj.clear()

```
s.clear()
print(s)
```

o/p: { }

To remove the set object s we use the statement

~~del(s)~~ del s

update():

This method is used to add a list of elements into the set.

syntax: set_obj.update(list_of_elements)

Ex:

```
s={1,2,3,4}
s.update([5,6,7,8])
print(s)
```

o/p:

{1,2,3,4,5,6,7,8}

union():

This method is used to combined all elements in two sets.

Syntax: set_obj1.union(set_obj2)

Example: ① s1={1,2,3,4,5,6}
s2={4,5,6,7,8}

```
print(s1.union(s2))
print(s2)
```

o/p: {1,2,3,4,5,6,7,8}

② print(s1.union(s1))

o/p: {1,2,3,4,5,6,7,8}

It can also be written as

→ print(s1|s2)

o/p: {1,2,3,4,5,6,7,8}

intersection(): Common elements in both the sets

Syntax: set_obj1.intersection(set_obj2)

Example
```
s1 = {1,2,3,4,5}
s2 = {4,5,6,7}
print(s1.intersection(s2))
```
o/p: {4,5}

It can also be written as
```
print(s1&s2)
```
o/p: {4,5}

difference():

Syntax: set_obj1.difference(set_obj2)

Example
```
s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8}
s1.difference(s2)
```
o/p: {1,2,3}

```
s2.difference(s1)
```
o/p: {6,7,8}

It can also represented as s1-s2

issuperset(): It return a boolean value

Syntax: set_obj1.issuperset(set_obj2)

Example
```
s1 = {0,1,2,3,4,5,6,7,8,9}
s2 = {1,2,3,4}
s3 = {4,5,6,7}
s1.issuperset(s2)
s2.issuperset(s1)
s2.issuperset(s3)
```
o/p: True
False
False

issubset()

Syntax: set_obj1.issubset(set_obj2)

example

```
s1 = {1,2,3,4,5,6,7,8,9}
s2 = {1,2,3,4}
s3 = {4,5,6}
s2.issubset(s1)
s3.issubset(s2)
```

o/p: True
False

~~differenceAll~~

difference_update():

This method is used to change the original set

syntax: set_obj1.difference(set_obj2)

example

```
A = {1,2,3,4}
B = {3,4,5,6}
A.difference_update(B)
print(A)
```

o/p: {1,2}

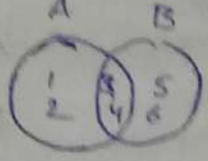
Symmetric():

This method is used to create a new set except common element in both sets.

Syntax: set_obj1.symmetric(set_obj2)

Example

```
A.symmetric(B)
```



o/p: {1,2,5,6}

Symmetric_update():

This method is used to update a specific set

Syntax: set_obj1.symmetric_update(set_obj2)

Example

```
A.symmetric_update(B)
print(A)
```

O/P: {1,2,5,6}

isdisjoint()

This method is return the boolean value

syntax: set_obj1.isdisjoint(set_obj2)

Example

```
A = {1,2,3,4,5,6}
B = {3,4,5,6}
A.isdisjoint(B)
```

O/P: True

intersection_update()

This method is updated the intersection set to the specific set.

Syntax:

set_obj1.intersection_update(set_obj2)

Example

```
A = {1,2,3,4}
B = {3,4,5,6}
A.intersection_update(B)
print(A)
```

O/P {3,4}

* X = [3, 4.25, 'ABC', 0, None, 'DEF', True]

```
S = 0; S1 = ''
for i in X:
    if i>0 or i<=0:
        S += i
    elif i==True or i==False or i==None:
        Continue
    else:
        S1 += i
print(S1, S2)
```

```
X = [3, 4.25, 'ABC', 0, None, 'DEF', True]
S1, S2 = 0, ''
for i in X
    if type(i) == int or type(i) == float:
        S1+i
    if type(i) == str:
        S2 += i
```

O/P: 7.25 ABCDEF

* L = [1, 2, 3, 4, 5]

```
L[0], L[-1] = L[-1], L[0]
print(L)
```

O/P: [5, 2, 3, 4, 1]

* L = [1, 2, 3, 4, 5]

Print the Matrix

```
M = [[1,2,3],[4,5,6],[7,8,9]]

for i in range(len(M)):
    for j in range(len(M[0])):
        print(M[i][j], end=" ")
    print()
```

o/p:

```
1 2 3
4 5 6
7 8 9
```

*

```
L1 = {1,2,3,4,5}
L2 = L1
L3 = L1.copy()
L1[0] = [7]
print(L1, L2, L3)
```

Frozenset :

frozenset is a purely immutable data structure. This frozenset is created by using frozenset function

Example

```
S1 = set([1,2,3,4])
fset = frozenset(S1)
```

where frozenset operation are similar to set operations like copy, difference, isdisjoint, issubset, issuperset, symmetric_difference, union

dictionary list values multiplied by 2 using Dictionary Comprehension:

```
d = {'a':1, 'b':2, 'c':3, 'd':4}
a = list(d.values())
for i in a:
    print(i*2)
```

```
* d1 = {'a':1, 'b':2, 'c':3, 'd':4}
  d2 = {k:v*2 for (k,v) in d.items()}
```

o/p:

```
{'a':2, 'c':6, 'b':4, 'd':8}
```

```
* d1 = {'a':1, 'b':2, 'c':3, 'd':4}
  d2 = {k:v for (k,v) in d.items() if v%2==0}
  print(d2)
```

o/p:

```
{'b':2, 'd':4}
```

```
* d = {'a':1, 'b':2, 'c':3}
  d.update({'d':4})
  print(d)
```

o/p: {'a':1, 'b':2, 'c':3, 'd':4}

```
* print(d.popitem())
* d2 = {k:v for (k,v) in d.items() if k=='e'}
  print(d2)
```

o/p: {'e':6}

* F={'t1':-20, 't2':-10, 't3':-25, 't4':0}

```
d2 = {k:(v-32)*(float(5)/9) for (k,v) in F.items()}
print(d2)
```

* F = {'t1':-20, 't2':-10, 't3':-25, 't4':0}

```
c = list(map(lambda x:(float(5)/9)*(x-32), F.values()))
cel_temp = dict(zip(F.keys(), c))
print(cel_temp)
```

o/p: {'t4': -17.777777, 't2': -23.333336, 't3': -31.666668, 't1': -28.888889}

* ~~t=[1,2,3,1,4,2]~~

~~def test(t) j=~~

~~[]~~

~~i in range~~

* L = [1,2,3,4,1,2]

```
b = []
for i in L:
    if i not in b:
        b.append(i)
```

* L = [1,2,3,4]

```
L.remove(max(L))
print(max(L))
```

* #include<stdio.h>

int main()

Unit - 6

* class - collection of attributes and methods
* object: It is instance of a class (or) entity. Abstraction

Abst
- a = 10 (public)
- -a = 10 (protected)
- --a = 10 (private)

self - instance of class used to access attributes and methods.

self:

It represents the instance of class.

The "self" keyword is especially used for access attributes and methods within the class

Ex:

```
class Test:
    a = 10
    b = 20
    def show(self):
        print(self.a, self.b)
```

⇒

```
class Test(object):
    a = 10
    b = 20
    def show(self):
        print(self.a, self.b)

x = Test()
print(x.a)
>>> 10
print(x.b)
>>> 20
x.show()
>>>{10,20}
```

For private membess: (declared in "--")

```
class Test(object):
        --a = 10
        --b = 20
        def show(self):
            print(self.--a, self.--b)
P = Test()  # P is an obj
print(P.--a)
>>> 10
P.show()
>>> {10,20}
```

without self:

```
class Test(object):
        a = 10
        b = 20
        def show(k):
            print(k.a, k.b)
Y = Test()
Y.show()
>>> {10,20}
```

self - fast performance in pth convention.

In class method declaration, 1st parameteas should be 'self' keyword.

Constructor:

Initialize the membess (or) attributes and allocate the memory internally (attributs of specific class membes)

```
=> class Test():
        def __init__(self, a, b):
            self.a = a
            self.b = b
Obj = Test(4,5)
obj.a
>>> 4
obj.b
>>> 5
```

Constructor:

Constructors is defined in python to magic (or) Dunder methods (Dunder means "__")

These magic methods represented in python two underscores (__) prefix and sufix of the method. These methods also known as operator overloading in python.

The python constructors are defined as

a) __new__()

b) __init__()

__new__() method:

This method creats an object of a class. This method also return instance object.

__new__(cls)

~~__init__(self)~~

__init__()

This method is an initialization of an object there is no return value.

__init__(self)

* The __new__() method contains one parameter cls. cls represents class object.

* In __init__() method contains single parameter self. The self represents the initialization of a class members of the class

Example

for __new__(cls)

```
class Test(object):
    def __new__(cls):
        print('new')

Test()
```

o/p: new

for __init__(self)

① class Test(object):

```
    def __init__(self):
        print('init')

Test()
```

o/p: init

<__main__.Test object at 0x02006210>

② class Test(object):

```
    def __init__(self):
        print('init')

t = Test()
```

o/p: init

return statement in __new__()

```
class Test(object):
    def __new__(cls):
        return 10

Test()
```

o/p: 10

return statement in __init__()

```
class Test(object):
    def __init__(self):
        return 10

t = Test()
```

o/p:

Error: __init__() should return none, not int

__new__() and __init__() in single class

① class Test(object):
    def __new__(cls):
        print('new')
    def __init__(self):
        print('init')

Test() (or) t = Test()

o/p: new

* If we use both __new__() and __init__() methods in a single class the __new__() method will call

② class Test(object):
    def __init__(self):
        print('init')
    def __new__(cls):
        print('new')
        return object.__new__(cls)

Test()

o/p: new
init
<__main__.Test object at 0x01F76180>

t = Test()

o/p: new
init

In python 3.x

```
class Test:
    def __init__(self):
        print('init')
    def __new__(cls, *args, **kwargs):
        print('new')
        return object.__new__(Test, *args, **kwargs)
t=Test()
```

- *args - gives the list of values
- **kwargs - returns the dictionary

```
class Test:
    c=0
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def sum(self):
        self.c = a+b self.a + self.b
        return self.c
a = 1000
b = 2000
t = Test(a,b)
d = t.sum()
print(d)
```

Operator overloading

Operator overloading is a technique of polymorphism. The operator is overloading in between class objects.

Arithematic operator ope overloading

Addition	+	__add__()
subtraction	-	__sub__()
Multiplication	*	__mul__()
Modulo division	/	__truediv__()
floor division	//	__floordiv__()
power	**	__pow__()
Modulo	%	__mod__()

Bitwise operator overloading

&		__and__()
|		__or__()
~		__not__()
<<		__lshift__()
>>		__rshift__()
^		__xor__()

Relational operator overloading

<	__lt__()
<=	__le__()
>	__gt__()
>=	__ge__()
==	__eq__()
!=	__ne__()

Assignment operator overloading

+=	__iadd__()
-=	__isub__()
*=	__imul__()
/=	__idiv__()
//=	__ifloordiv__()
**=	__ipow__()
%=	__imod__()
&=	__iand__()
|=	__ior__()
~=	__inot__()
<<=	__ilshift__()
>>=	__rshift__()
^=	__ixor__()

* class Test:

```
    def __init__(self, a):
        self.a = a
    def __add__(self):
        return self.a + self.a
t1 = Test(5)
t2 = Test(10)
print(t1 + t2)
```

* class Test:

```
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def __add__(self, t):
        return self.a + t.a, self.b + t.b
    def __str__(self):
        return self.a, self.b
t1 = Test(3,2)
t2 = Test(4,3)
t3 = t1 + t2
print(t3)
```

write a program the given 2 numbers greater than and equal to check using operator overloading

```
class Test:
    def __init__(self, a):
        self.a = a
    def __gt__(self, t):
        if(self.a > t.a):
            return self.a print(self.a, "is greater than", t.a)
        else:
            return t.a print(self.a, "is less than", t.a)
t1 = Test(10)
t2 = Test(20)
print(t1 > t2)
```

```
def __eq__(self, t):
    if(self.a == t.a):
        print(self.a, "is equal to", t.a)
    else:
        print(self.a, "is not equal to", t.a)
```

```
t1 = Test(10)
t2 = Test(20)
t1 > t2
t1 == t2
```

Division:

```
class Test:
    def __init__(self, a):
        self.a = a
    def __truediv__(self, t):
        return self.a / t.a
t1 = Test(20)
t2 = Test(10)
print(t1 / t2)
print(t1 // t2)
```

```
def __floordiv__(self, t):
    return self.a // t.a
```

```
class Test:
    def __init__(self,a):
        self.a = a
    def __iadd__(self,t):
        self.a += t.a
        return self.a

t1 = Test(10)
t2 = Test(20)
print(t1 += t2)
```

Add an element into the list using assignment operator

Method overloading

same functions but with different signatures.

Ex:
```
class Test:
    def show(self, name = None):
        if name not in None:
            print('Hello' + name)
        else:
            print('Hello')

t = Test()
t.show()              # Hello
t.show("vasavi")      # Hello vasavi
```

2. class Test:

```
    def area(self, x=None, y=None):
        if (x != None and y != None):
            return x * y
        elif (x != None):
            return x * x
        else:
            return 0

t = Test()
t.area()      # 0
t.area(5)     # 25
t.area(3,3)   # 15
```

3.

```
class Test:
    def __init__(self, a, b):
        self.a = list(a)
        self.b = b
    def __len__(self):
        return (len(self.a))

t = Test(['Apple', 'Banana', 'Organge'], 'Vasavi')
print(len(t))  # 3
```

4.

```
class Test:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def __abs__(self):
        return abs((self.a ** 2 + self.b ** 2) ** 0.5)

t = Test(3, 5)
print(t)
```

class Test

Function overloading:

```
def sum(a):
    return a+5
def sum(a,b):
    return a+b
def sum(a,b,c):
    return a+b+c
print(sum(5)) # 16
print(sum(2,3)) # 5
print(sum(3,5.4,'Vasavi')) # 8.4Vasavi
```

Operator overloading:

1.
```
class Test:
    def __init__(self,a):
        self.a = a
    def __add__(self,t):
        return self.a + t
t = Test(3)
print(t+5)  # 8
print(5+t)  # error
```

```
def __radd__(self,t):
    return self.a+t
t = Test(3)
print(t+5) # error
print(5+t) # 8
```

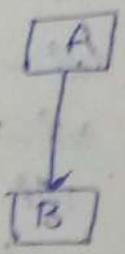
* r means reflection or reverse

Inheritance:

Acquiring properties from super class to derived class

1. Simple
2. Multi level
3. Multiple
4. Hierarichal

simple inheritance:



Syntax:

```
class superclass:
    properties

class Derived_class(Superclass):
    properties
```

Example

```
class A:
    _a = 10        # protected members
    _b = 20
    def show(self):
        print(self._a, self._b)

class B(A):
    def show(self):
        self._a = 100
        self._b = 200
        print(self._a, self._b);

x = B()
x.show()
```

O/P:

100 200

```
class student:
    def getdetails
        def getdata(self):
            self._rno = int(input("enter rno"))
            self._name = input("enter name")
            self._m1 = int(input("enter m1"))
            self._m2 = int(input("enter m2"))
            self._m3 = int(input("enter m3"))
        def putdata(self):
            print(self._rno, self._name, self._m1,
                  self._m2, self._m3)
class Details(student):
    def total(self):
        print(self._m1 + self._m2 + self._m3)

s = Details()
s.getdata()
s.putdata()
s.total()
```

```
100 - 1.5
100-200 - 3.5
>200 - 5
class Details:
    def getdata(self):
        self._name = input("enter name")
        self._Hno = input("enter house number")
        self._units = int(input("enter no of units")
```

```
class ElectricityBill(Details):
    def bill(self):
        if (self.units <= 100):
            self.b = units * 1.5
        elif (self.units >100 and self.units <=200):
        self.b=units * 3.5
        else:
         self.b =units *5
    def putdata(self):
        print( self.-name, self._Hno, self._units,
                self.b)

e = ElectricityBill ()
e.getdata()
e.bill ()
e.putdata()
```

super( ):

super( ) is a built in function in python it is return proxy object. (temporary object). The proxy object is return to the parent. This super( ) function is used in derived and sibling classes.

In python, super ( ) function is used in different formats

Example without super

```
class A:
    def __init__ (self,a):
        self.a =a
    def show(self):
        print(self.a)
class B(A):
    def __init__ (self,a,b):
        A.a = a
        self.b = b
```

```
    def show(self):
        print(A.a, self.b)  # print(self.a, self.b)

obj = B(10, 20)
obj.show()
```

with super()

```
class A:
    def __init__(self, a):
        self.a = a
    def show(self):
        print(self.a)
class B(A):
    def __init__(self, a, b):
        super().__init__(a)
        self.b = b
    def show(self):
        print(self.a, self.b)
ob = B(10, 20)
ob.show()
```

Write a python program to calculate area and perimeter of rectangle and square

```
class Rectange:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def rectangleArea(self):
        print(self.a * self.b)
        print(2 * (self.a + self.b))
class Square(Rectangle):
    def __init__(self, a, b, s):
        super().__init__(a, b)
```

```
        self.s = s
    def squareArea(self):
        print("Area") (self.s * self.s)
        print(s * self.s)

ob = square(2, 3, 5)
ob.rectangleArea()
ob.squareArea()
```

```
class Rect:
    def __init__(self, l, b):
        self.l = l
        self.b = b
    def area(self):
        return self.l * self.b
    def perimeter(self):
        return 2 * self.l + 2 * self.b
class square(Rect):
    def __init__(self, l):
        super().__init__(l, l)

obj = square(4)
print("Area of square is :", obj.area())
obj1 = Rect(3, 4)
print("Area of Rect is :", obj1.area())
```

Multi level inheritance:

Syntax:


```
class Base:
    Bass class properties
class Intermediate(Base):
    Intermediate class properties
class child(Intermediate):
    child class properties.
```

Pass statement.

It is used to required syntactically but you do not want any command or code to execute

The pass statement is a null operation nothing happens in the statement. But the python interpreter is execute the pass statement.

Ex:

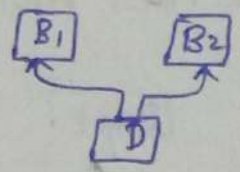
1.
```
class A:
    pass
class B(A):
    pass
class C(B):
    pass
obj = C()
```

Ex:

2.
```
def myfunc():
    pass
```

Multiple inheritance:

Single child class more than one base classes is called multiple inheritance.



Syntax:

```
class B1:
    B1 class properties
class B2:
    B2 class properties
class C(B1, B2):
    child class properties.
```

Example

```
class Father:
    def show(self):
        print('Father class')
class Mother:
    def show(self):
        print('Mother class')
class child(Father, Mother):
    def disp(self):
        self.show()
c = child()
c.disp()
```

o/p:

Father class
Mother class

(MRO - Method Resolution order)

```
for i in c.mro():
    print(i)
```

mro

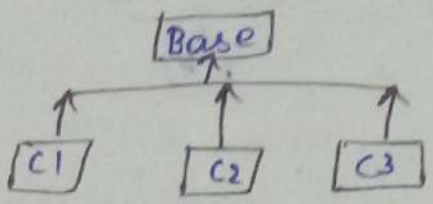
mro (method resolution order).

It is specified hierarchy of classes in multiple inheritance. This is especially used in same function is defined in multiple base classes. that is calling in specific order is called method resolution order

The built in method resolution order is mro

Heivarichal inheritance:

Single base class multiple child classes.



Syntax:

```
class Base:
    Base class Body
class c1 (Base):
    c1 class Body
class c2(Base):
    c2 class Body
class c3(Base):
    c3 class Body
```

Example

```
class Vehicle:
    def __init__(self, model, type)
        self.model = model
        self.type = type
    def show(self):
        print(self.model, self.type)
class Car(vehicle):
    def __init__(self, seat, model, type, seat)
        super().__init__(model, type)
        self.seat = seat
class Bike(vehicle):
    def __init__(self, model, type, seat):
        super.__init__(model, type)
        self.seat = seat
    def __str__(self):
        print(self.model, self.type, self.show)
```

def show(self):
print(self.model, self.type, self.seat)

```
b = Bike(1997, 'petrol', 2)
c = Car(2002, 'Desiel', 4)
b.show()
c.show()
```

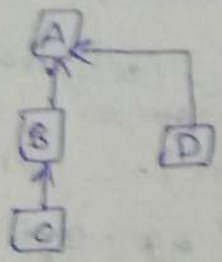
10/10/20

Hybrid inheritance:

This form of inheritance is created more than two forms of inheritance.

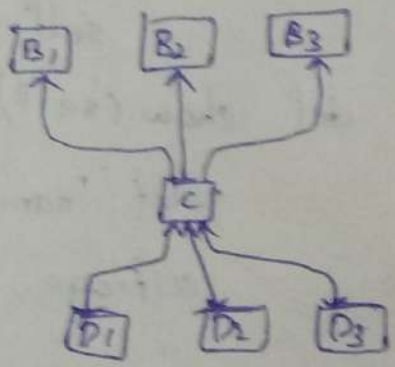
Ex:

```
class A:
    pass
class B(A):
    pass
class C(B):
    pass
class D(A):
    pass
```



Ex:

```
class B1:
    pass
class B2:
    pass
class B3:
    pass
class C(B1, B2, B3):
    pass
class D1(C):
    pass
class D2(C):
    pass
class D3(C):
    pass
```



Multilevel

Base class - person (Constructor, name, id)

intermediate - Employee (Age)

child class - child (salary)

```
class Pearson:
    def __init__(self, name, id):
        self.name = name
        self.id = id

class Employee (pearson):
    def __init__(self, name, id, age):
        super(Employee, self).__init__(name, id)
        self.age = age

class child (Employee):
    def __init__(self, name, id, age, salary):
        super(child, self).__init__(name, id, age)
        self.salary = salary.

    def show (self):
        print ('name', self.name, 'id', self.id, 'age:',
        self.age, 'salary:', self.salary)

c = child('aaa', 123, 25, 25000)
c.show()
```

Method Overriding:

```
class person:
    def __init__(self, name, rno):
        self.name = name
        self.rno = rno
    def show(self):
        print(self.name, self.rno)
class student(person):
    def __init__(self, name, rno, age):
        super().__init__(name, rno)
        self.age = age
    def show(self):
        super().show()
        print(self.age)
s = student('Vasavi', 123, 25)
s.show()
```

isinstance():

This method is used to check the given object (instance) is particular class

syntax:

```
isinstance(object, class_name)
```

Ex:

```
print(isinstance(s, student))
print(isinstance(s, person))
```

issubclass():

This method is used to check whether the given subclass is related to the base class. It returns boolean value

syntax: issubclass(sub_class_name, Base_class_name)

Ex:

```
print(issubclass(student, person))
print(issubclass(person, student))
```

*
```
class A:
    pass
class B(A):
    pass
class C(A):
    pass
c = C()
print(isinstance(c, B)) # False
print(isinstance(c, A)) # True
print(issubclass(c, B)) # False
print(issubclass(C, A)) # True
```

write a python program implement ATM using inheritance

Insert card
Enter pin
withdraw
Deposit
Pin generation

name, Accno, bal, pin

~~class ATM:~~

~~void getdata(self):~~

~~self.name = name~~

```
class ATM:
    self.name = "abc"
    self.acc_no = 20020
    self.bal = 20000
    self.pin = 5314
class verify(ATM):
    void verify_Accno(self, acc_no):
        if self.acno == self.acc:
```

```
class Bank:
    name="abc"
    accno=1234
    bal=25000
    pin=1234
class Verify(Bank):
    def veraifyacc(self,x):
        if x==self.accno:
            return 1
        else:
            return 0
    def verifypin(self,y):
        if y==self.pin:
            return 1
        else:
            return 0
class ATM(Verify):
    def deposit(self):
        print('enter amount for deposit')
        n=int(input())
        self.bal+=n
        print('current balance is:',self.bal)
    def withdraw(self):
        print('enter amount for withdraw')
        n=int(input())
        self if self.bal<n:
                print('insufficient balance')
        else:
            self.bal-=n
            print('current balance is:',self.bal)
    def balenquiry(self):
        print('current balance is',self.bal)
```

```
class Changepin(Bank):                    ATM
    def pinchange(self):
        print('enter new pin')
        pin1 = int(input())
        self.pin = pin1
        print('pin number changed successfully')

a = ATM()
c = Changepin()
while True:
    print('enter acc no:')
    x = int(input())
    if verifyacc(x) == 1:
        print('valid acc number')
        print('enter pin')
        y = int(input())
        if verifypin(y) == 1:
            print('valid pin')
            print('Enter 1. deposit')
            print('2. withdraw')
            print('3. Pin change')
            print('4. Balance enquiry')
            Print('Enter choice')
            Ch = int(input())
            if ch == 1:
                a.deposit()
            elif ch == 2:
                a.withdraw()
            elif ch == 3
                a.pinchange()
            else
                a.bal_enquiry()
        else:
            print('Invalid pin')
```

```
else:
    print('invalid account number')
print('Enter 1 for Exit and any other key for Continue')
op=int(input())
if op==1:
    exit(0)
else:
    pass
```

assert:

Python assert statement is debugging aid if there is no errors execute consecutive statements if an error raises assert AssertionError. This statement especially used for unit Testing

If we want to test any classes using unit test package. Then the developer check the unit test should be inherited the following class.

unittest.TestCase

Ex: ① `assert sum([1,2,3])==5,'Not Equal'`

o/p: AssertionError: Not Equal.

②
```
def my(x):
    return x+1
assert my(4)==6,'Not Equal'
```

o/p: AssertionError: Not Equal

③
```
impat unittest
class Test(unittest.TestCase):
    def test_upper(self):
        self.assertEqual('Vasavi'.upper(),'VASAVI')
    def test_isupper(self):
        self.assertTrue('VASAVI'.isupper())
        self.assertFalse('Vasavi'.isupper())
```

```
    def test_split(self):
        x = 'Vasavi Engineering'
        self.assertEqual(x.split(), ['Vasavi', 'Engineering'])
        with self.assertRaises(ValueError):
            x.split(2)

unittest.main()
```

getter and setter methods: case-1

```
class Test:
    def __init__(self):
        self.__a = 0
    def getA(self):
        print('Call getter Method')
        return self.__a
    def setA(self, a):
        print('Call setter Method')
        self.__a = a
    a = property(getA, setA)

t = Test()
t.a = 10
print(t.a)
```

o/p:

```
Call setter Method
Call getter Method
10
```

Case-2 Overridding of property method

```
class Test(object):
    def __init__(self):
        self.age = __age = 0
    @property
    def age(self):
        print('call getter method')
        return self.__age
    @age.setter
    def age(self,a):
        print('call setter Method')
        self.__age = a
t = Test()
t.age = (Vasavi) 20
print(t.age)
```

o/p: call setter Method
call getter Method
20

UNIT-5

Exception Handling

Exception:

syntax:

```
try:
    Exception Block
except:
    Execute Exception
else:
    Else Block
finally:
    Always Execute Finally Block
```

* Exception is an error which controls abnormal condition of a code

Types of errors

ZeroDivisionError
NameError
ValueError
IOError
ImportError

example

①
```
try:
    x = int(input('Enter x value:'))
    print(x/0)
except ZeroDivisionError:
    print('Division by zero')
```

o/p:

```
Enter x value: 5
Division by zero
```

②
```
try:
    x = int(input('Enter x value:'))
    print(x/0)
except:
    print('Division by zero')
```

o/p:

```
Enter x value: 10
Division by zero
```

3.
```
try:
    x=int(input('Enter x value'))
    print(x/0)
except zeroDivisionError as e:
    print(e)
```

4.
```
try:
    x=int(input('Enter x value'))
    print(x/2)
except:
    print('Division Error')
else:
    print('No Exception Raise')
```

o/p:
```
Enter x value 5
2
No Exception raise
```

5.
```
try:
    x=int(input('Enter x value'))
    print(x/2)
except:
    print('Division Error')
else:
    print('No exception Raise')
finally:
    print('Always execute finally')
```

o/p:
```
Enter x value 5
2
No exception raise
Always execute finally
```

6 try:
    print(x/z)
except NameError:
    print('Invalid name')
except ZeroDivisionError:
    print('Divide by zero')
else:
    print('NO Exception Raise')
finally:
    print('Always Execute Finally')

o/p: Invalid name
     Always Execute finally.

User defined Exception:

class MyException (Exception):
    def __init__(self, a):
        self.a = a
    def __str__(self):
        return repr(self.a)

try:
    raise MyException ("Welcome my Error")
except myException as e:
    print(e.a)

o/p:
    welcome my error

```
else:
    print("You are perfectly Eligible")
```

Files in python:

A file is a container in computer system that stores information. This is identified by file names. The file formats are document, text, video streaming, Audio, picture etc. These files are stored in non-volatile memories such as hard disk, floppy disk, pendrives.

opening a file:

```
file_obj = open(file name, [access_mode])
```

Ex: `file = open("d:\\myfile.txt", "w")`

Create a new file or to open a existed file

closing a file.

```
file_obj.close();
```

Ex: `file.close()`

Access modes:

- r - read mode
- w - write mode
- a - append
- r+ - read and write
- w+ - write and read
- rb - read binary mode
- wb - write binary mode
- ab - append binary mode
- rb+ - read and write binary
- wb+ - write and read binary
- ab+ -

Example

```
file = open('d:\\myfile.txt', 'w')
file.write("This is testing file")
file.close()
file1 = open('d:\\myfile.txt','r')
s=file1.read(10)
print(s)
s1 = file1.read()
print(s1)
file1.close()
```

o/p:

```
This is te
sting file
```

By using os package:

```
Import os
os.mkdir('d:\\vasavi')
file = open('d:\\vasavi\\myfile.txt', 'w')
file.write('This is testing')
file.close()
file1 = open('d:\\vasavi\\myfile.txt', 'r')
s = file1.read()
print(s)
file1.close()
```

o/p: This is testing

A program to count frequency of characters in a given file. Can you use character frequency to tell whether the given files is a python program file, c program file or text file.

```
~~Eoo~~
~~file = open('d:\\myfile.txt', '~~
~~s=input('Enter file name')~~
fil = input('Enter file name with path: ')
 f = open(file, 'r')
 if f.name.endswith('.py'):
       print('This is python file')
 elif f.name.endswith('.c'):
       print('This is c program file')
 elif f.name.endswith('.txt'):
       print('This is text file')
 s = f.read()
 print('Frequency of characters:', len(s))
```

6. A program to compute the number of characters, words and lines in a file.

```
f=open('d:\\csed.txt', 'w')
for i in range(3):
      x=input()
      f.writelines(x + '\n')
f.close()
w=0; L=0; c=0
with open('d:\\csed.txt', 'r') as f:
      s=f.read().strip()
      print(s)
      f.close()
      f=open('d:\\csed.txt', 'r')
```

```
        for line in f:
            word = line.split()
            l+=1
            w+=len(word)
            c+=len(line)
        f.close()
print('no. of characters:', c)
print('No. of words:', w)
print('No. of words:', l)
```

(or)

```
f=open('d:\\csed.txt','r')
for i in f:
    print(i)
    word=i.split()
    l+=1
    w+=len(word)
    c+=len(line)
f.close()
print(c)
print(w)
print(l)
```

Open files using try-except blocks.

①
```
try:
    f = open('d:\\csed.txt', 'r')
    print(f.read())
except:
    print("No file")
finally:
    f.close()
```

②
```
try:
    f=open('d:\\csed.txt', 'r')
    print(f.read())
except IOError as e
    print(e)
finally:
    f.close()
```

Binary files:

GUI - Tkinter

Import statement

2.x - import Tkinter

3.x - import tkinter

object creation

object = Tkinter.Tk()

Ex: win = Tkinter.Tk()

Tk() - is a constructor in Tkinter package

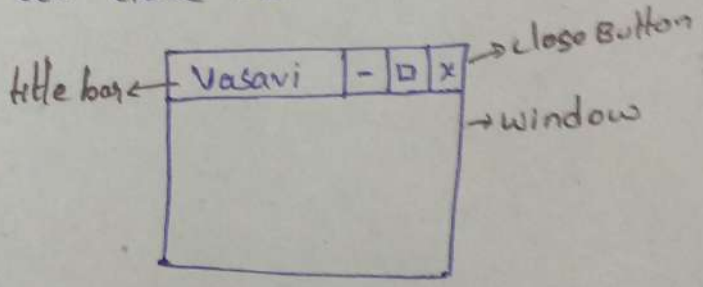
creating title

obj.title(title name)

Ex: win.title('Vasavi')

mainloop:

main loop is used for running the window infinite time until we close the window



geometry:

Geometry is used to change the size of the window the default size of the window is 100x100.

win.geometry('300x300')

Creating label

Syntax:

label_obj = Label(Tkinter_obj, text = 'something to'
bg = 'color', fg = 'color', font = 'font name'
command = ...)

Ex: L1 = Tkinter.Label(win, text='welcome', font='Harrington 16', bg='red', fg='blue')

Grid:

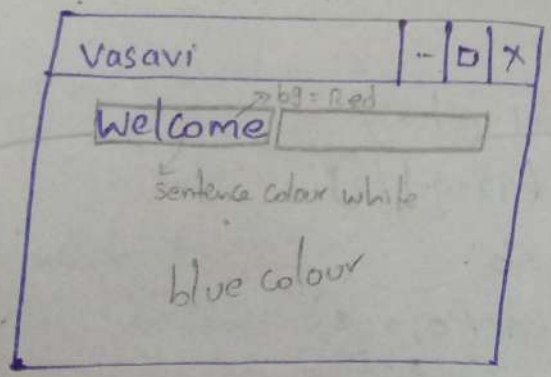
syntax: Label_obj.grid(row=0, colum=0)

grid is used to add Label at the given position

example

```
import Tkinter
win=Tkinter.Tk()
win.title('Vasavi')
win.geometry('300x300')
win.config(bg='blue')
L1=Tkinter.Label(win, text='welcome', font=
    'Harrington 16', bg='red', fg='white'
L1.grid(row=0, column=0)
T1=Tkinter.Entry(win, bd=5, width=20)
T1.grid(row=0, column=1)
win.mainloop()
```

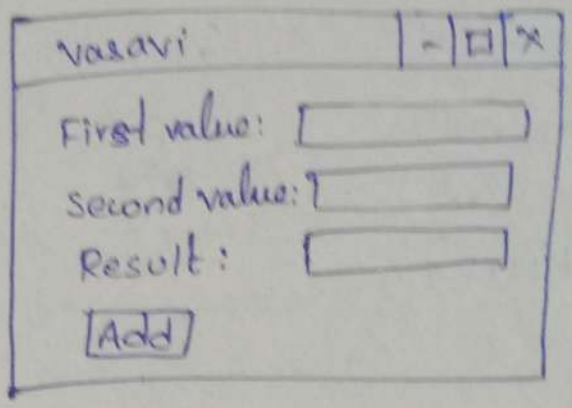
o/p:



Program to add two numbers using GUI

```
import Tkinter
win = Tkinter.Tk()
win.title('Vasavi')
win.geometry('300x300')
L1 = Tkinter.Label(win, text = 'First value:')
L1.grid(row = 0, column = 0)
T1 = Tkinter.Entry(win, bd = 5)
T1.grid(row = 0, column = 1)
L2 = Tkinter.Label(win, text = 'second value:')
L2.grid(row = 1, column = 0)
T2 = Tkinter.Entry(win, bd = 5)
T2.grid(row = 1, column = 1)
L3 = Tkinter.Label(win, text = 'Result:')
L3.grid(row = 2, column = 0)
T3 = Tkinter.Entry(win, bd = 5)
T3.grid(row = 2, column = 1)
def click():
    try:
        a = int(T1.get())
        b = int(T2.get())
        res = a+b
        T3.insert(0, res)
    except:
        T3.insert(0, 'Invalid numbers')
b = Tkinter.Button(win, text = 'Add', command = click)
b.grid(row = 3, column = 0)
win.mainloop()
```

o/p:



turtle:

Example:

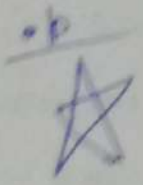
① import turtle

```
turtle.forward
turtle.colour ('green') # gives colour
turtle.pensize (5)  # set pen size
turtle.forward (100)  #
turtle.left (90)  #
turtle.forward (200)  #
turtle.Right (90)  #
turtle.foward (20)  #
turtle.done()
```

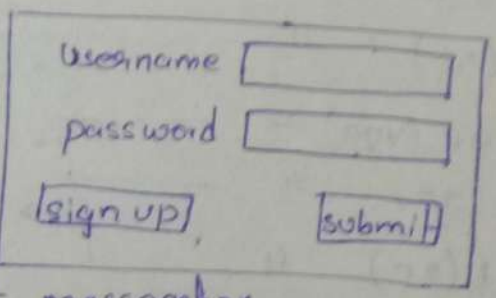
② import turtle

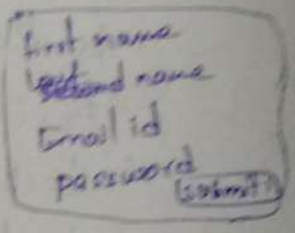
```
turtle.color('red')
while True:
    turtle.forward(200)
    turtle.left(140)
    if abs(turtle.pos()) < 1:
        break
turtle.done()
```

3.
```
import turtle
for i in range(5):
    turtle.colour('red')
    turtle.forward(200)
    turtle.right(145)
turtle.done()
```



4.
```
import turtle
col=['red','yellow','blue','green','brown','yan']
for i in range(40)
    turtle.colour(col[i%6])
    turtle.circle(50)
    turtle.left(20)
turtle.done()
```






```
from tkinter import messagebox
import tkinter as t
win = tkinter.Tk()
dic = {'sai':123, 'Bhaggavi':1234, 'sowmda':345}
win.title('Login form')
win.geometry('200x200')
l1=tLabel('user name')
l1.grid(row=0, column=0)
l2 tLabel('password')
l2.grid(row=1, column=0)
e1 = t.Entry(win, bd=5, width=100)
e1.grid(row=0, column=1)
```

→
```
l3 = t.Label()
l3.grid(row=2, column=1)
```

```
e3 = t.Entry(win, bd=5, width=70e)
e3.grid(row=1, column=0)
e2 = t.Entry(win, bd=5, width=100)
e2.grid(row=1, column=1)

def click():
    a = t1.get()
    b = t1.get()
    if a in dic.key() and dic.get(a) == b:
        l3.insert("Valid Login ID")
    else:
        messagebox.info('Login', 'Invalid username
                                  or ID')

def newwin():
    win1 = tkinker.Toplevel(root)
    win1.title('Register Form')
    win1.geometry('500x500')
    l4 = t.Label('First name')
    l4.grid(row=0, column=0)
    l5 = t.Label('Last name')
    l5.grid(row=1, column=0)
    T
```