```
data type -> class
variable -> object
```

In [ ]:

In [4]:
```
# class
```

```
Class is a blueprint using which we can create objects
```

In [ ]:
```
# objects
```

```
objects are the instance of class.
At every instance the value of the obj contains different values
```

In [ ]:

In [ ]:
```
# Object Literal
```

```
Python also provides objecy literal through which we can create the obj
of the class in thos way also for built in data types
```

In [3]:
```
l1 = list([1,2,3])
l1
```

Out[3]:
```
[1, 2, 3]
```

In [4]:
```
l2 = list()
l2
```

Out[4]:
```
[]
```

In [ ]:

In [5]:
```
#Using Object Literal for built in data types
```

In [6]:
```
l3 = [1,2,3]
l3
```

Out[6]:
```
[1, 2, 3]
```

In [7]:
```
l4 = []
l4
```

Out[7]:
```
[]
```

In [ ]:

In [8]:
```
# function
```

```
A function is a block of code which only runs when it is called. You can
pass data, known as parameters, into a function
```

In [9]:
```
# Method
```

Method is a special fucntn/ fuctn defined inside a class

In [ ]:

In [12]:
```
# Constructor / special / magic / dunder methods
```

Constructor is a special kind of method that is called when an object is created.

The code written inside the constructor is run automatically when an object of the class is created

In [ ]:

In [13]:
```
# creating an Atm
```

In [7]:
```python
class Atm:
    def __init__(self):
        self.pin=''
        self.bal=0

        print('ID of self ',id(self))
        self.menu()

    def menu(self):
        while True:
            user_inp = input('''
Welcome to WesternNewton Bank

press 1: to create pin
press 2: to deposit money
press 3: to withdraw
press 4: to check balance
press 5: to quit
''')

            if user_inp=='1':
                #print('Creating Pin')
                self.create_pin()
            elif user_inp=='2':
                #print('Depositing Money')
                self.deposit()
            elif user_inp=='3':
                #print('Withdrawing Money')
                self.withdraw()
            elif user_inp=='4':
                #print('Checking Balance')
                self.check_bal()
            elif user_inp=='5':
                #print('Quitting Now')
                break

    def create_pin(self):
        self.pin = int(input('\nenter your pin : '))
        print('PIN Set Successful !!')

    def deposit(self):
        #print('current pin ',self.pin)
        tmp = int(input('\nEnter the pin : '))
        if tmp==self.pin:
            amount = int(input('Enter the Deposit amount : '))
            self.bal+=amount
            print('Balance Updated Successfully')
            print('Balance : ',self.bal)
        else:
            print('Incorrect Pin entered')

    def withdraw(self):
        tmp = int(input('\nEnter the pin : '))
        if tmp==self.pin:
            amount = int(input('Enter the Withdrawal amount : '))

            if amount<self.bal and amount>0:
                self.bal-=amount
                print('Balance Updated Successfully')
                print('Balance : ',self.bal)
            else:
                print('Low Balance !!')
                print('Balance : ',self.bal)
        else:
            print('Incorrect Pin entered')

    def check_bal(self):
        tmp = int(input('\nEnter the pin : '))
        if tmp==self.pin:
            print('Balance : ',self.bal)
        else:
            print('Incorrect Pin entered')
```

In [3]:
```
harsh = Atm()
```

```


Welcome to WesternNewton Bank

press 1: to create pin
press 2: to deposit money
press 3: to withdraw
press 4: to check balance
press 5: to quit
1

enter your pin : 1234
PIN Set Successful !!

Welcome to WesternNewton Bank

press 1: to create pin
press 2: to deposit money
press 3: to withdraw
press 4: to check balance
```

In [ ]:
```

```

In [ ]:
```
Self : Current Obj
```

```
JIS obj ke sath abhi kaam kr rhe hoto hoo wohi self hota hai
```

```
the id of the object gets passed in self in the form of default
parameter to methods inside the class
```

In [8]:
```
sbi_atm = Atm()
```

```
ID of self  139790454110288

Welcome to WesternNewton Bank

press 1: to create pin
press 2: to deposit money
press 3: to withdraw
press 4: to check balance
press 5: to quit
5
```

In [9]:
```
print(id(sbi_atm))
```
```
139790454110288
```

In [ ]:

In [10]:
```
hdfc_atm = Atm()
```
```
ID of self  139790448605744

Welcome to WesternNewton Bank

press 1: to create pin
press 2: to deposit money
press 3: to withdraw
press 4: to check balance
press 5: to quit
5
```

In [11]:
```
print(id(hdfc_atm))
```
```
139790448605744
```

In [ ]:

Fraction code

 1 -> Numerator
---
 2 -> Denominator

In [19]:
```
class Fraction:
    def __init__(self,n,d):
        self.n = n
        self.d = d
```

In [20]:
```
f1 = Fraction(1,2)
print(f1)
```
```
<__main__.Fraction object at 0x7f2378686640>
```

In [ ]:

In [21]:
```
class Fraction:
    def __init__(self,n,d):
        self.n = n
        self.d = d

    def __str__(self):
        return (f'{self.n}/{self.d}')
```

In [23]:
```
frac1 = Fraction(2,3)
print(frac1)
```
```
2/3
```

In [24]:
```python
frac2 = Fraction(4,5)
print(frac2)
```
```
4/5
```

In [25]:
```python
# trying to add 2 fraction obj
print(frac1+frac2)
```
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [25], in <module>
      1 # trying to add 2 fraction obj
----> 2 print(frac1+frac2)

TypeError: unsupported operand type(s) for +: 'Fraction' and 'Fraction'
```

In [ ]:

In [41]:
```python
class Fraction:
    def __init__(self,n,d):
        self.n = n
        self.d = d

    def __str__(self):
        return (f'{self.n}/{self.d}')

    def __add__(self,other):
        num = (self.n * other.d) + (self.d * other.n)
        den = self.d * other.d
        return f'{num}/{den}'

    def __sub__(self,other):
        num = (self.n * other.d) - (self.d * other.n)
        den = self.d * other.d
        return f'{num}/{den}'

    def __mul__(self,other):
        num = self.n * other.n
        den = self.d * other.d
        return f'{num}/{den}'

    def __truediv__(self,other):
        num = self.n * other.d
        den = self.d * other.n
        return f'{num}/{den}'
```

In [42]:
```python
frac1 = Fraction(3,4)
print(frac1)
```
```
3/4
```

In [43]:
```python
frac2 = Fraction(5,6)
print(frac2)
```
```
5/6
```

In [44]:
```
print(frac1+frac2)
```
```
38/24
```

In [45]:
```
print(frac1-frac2)
```
```
-2/24
```

In [46]:
```
print(frac1*frac2)
```
```
15/24
```

In [47]:
```
print(frac1/frac2)
```
```
18/20
```

In [ ]:
```
```

Instance Variable : Is a kind of variable for which the value of the variable is different for differenet obj

We can access the instance variable using the object and dot (.) operator.

In Python, to work with an instance variable and method, we use the self keyword. We use the self keyword as the first parameter to a method. The self refers to the current object.

Every object has its own copy of instance variables

In [ ]:
```
```

Making the data private so no one can access it Outside of the class

In [57]:
```
class Atmx:
    def __init__(self):
        self.__pin=''
        self.__bal=0
        self.secret = 1212

        print('ID of self ',id(self))
        print('Secret ID is ',self.secret)
        self.__menu()

    def __menu(self):
        while True:
            user_inp = input('''
Welcome to WesternNewton Bank

press 1: to create pin
press 2: to deposit money
press 3: to withdraw
press 4: to check balance
press 5: to quit
''')

            if user_inp=='1':
                #print('Creating Pin')
                self.create_pin()
            elif user_inp=='2':
                #print('Depositing Money')
                self.deposit()
            elif user_inp=='3':
                #print('Withdrawing Money')
                self.withdraw()
            elif user_inp=='4':
                #print('Checking Balance')
                self.check_bal()
            elif user_inp=='5':
                #print('Quitting Now')
                break

    def create_pin(self):
        self.__pin = int(input('\nenter your pin : '))
        print('PIN Set Successful !!')

    def deposit(self):
        #print('current pin ',self.pin)
        tmp = int(input('\nEnter the pin : '))
        if tmp==self.__pin:
            amount = int(input('Enter the Deposit amount : '))
            self.__bal+=amount
            print('Balance Updated Successfully')
            print('Balance : ',self.__bal)
        else:
            print('Incorrect Pin entered')

    def withdraw(self):
        tmp = int(input('\nEnter the pin : '))
        if tmp==self.__pin:
            amount = int(input('Enter the Withdrawal amount : '))

            if amount<self.__bal and amount>0:
                self.__bal-=amount
                print('Balance Updated Successfully')
                print('Balance : ',self.__bal)
            else:
                print('Low Balance !!')
                print('Balance : ',self.__bal)
        else:
            print('Incorrect Pin entered')

    def check_bal(self):
        tmp = int(input('\nEnter the pin : '))
        if tmp==self.__pin:
            print('Balance : ',self.__bal)
        else:
            print('Incorrect Pin entered')
```

In [65]:
```
hdfc = Atmx()
```

```
ID of self  139789956464496
Secret ID is  1212

Welcome to WesternNewton Bank

press 1: to create pin
press 2: to deposit money
press 3: to withdraw
press 4: to check balance
press 5: to quit
1

enter your pin : 1234
PIN Set Successful !!

Welcome to WesternNewton Bank

press 1: to create pin
press 2: to deposit money
press 3: to withdraw
press 4: to check balance
press 5: to quit
5
```

In [66]:
```
hdfc.secret
```

Out[66]:
```
1212
```

In [67]:
```
hdfc.secret=999
```

In [68]:
```
hdfc.secret
```

Out[68]:
```
999
```

```
@ as we were easily able to modify secret as it was not set as private
```

In [71]:
```
hdfc.__pin = 9999
```

In [73]:
```
hdfc.pin = 8888
```

In [76]:
```
hdfc.check_bal()
```

```
Enter the pin : 8888
Incorrect Pin entered
```

In [77]:
```
hdfc.check_bal()
```

```
Enter the pin : 9999
Incorrect Pin entered
```

@ But the pin is not been changes as it was set private

In [80]:
```
hdfc.deposit()
```

```
Enter the pin : 1234
Enter the Deposit amount : 100
Balance Updated Successfully
Balance :  100
```

In [ ]:

NOTE ::

- When creating a private data/ variable we use
__ brefore the var name

- and when we do this it gets changed to _classname_varname

- Ex:
__pin -> _Atmx__pin

__balance -> _Atmx__balance

Nothing in python is truly private

In [ ]:

In [ ]:

In [2]:
```python
class Atmx:
    def __init__(self):
        self.__pin=''
        self.__bal=0
        self.secret = 1212

        print('ID of self ',id(self))
        print('Secret ID is ',self.secret)
        self.__menu()

    def __menu(self):
        while True:
            user_inp = input('''
Welcome to WesternNewton Bank

press 1: to create pin
press 2: to deposit money
press 3: to withdraw
press 4: to check balance
press 5: to quit
''')

            if user_inp=='1':
                #print('Creating Pin')
                self.create_pin()
            elif user_inp=='2':
                #print('Depositing Money')
                self.deposit()
            elif user_inp=='3':
                #print('Withdrawing Money')
                self.withdraw()
            elif user_inp=='4':
                #print('Checking Balance')
                self.check_bal()
            elif user_inp=='5':
                #print('Quitting Now')
                break


    def create_pin(self):
        self.__pin = int(input('\nenter your pin : '))
        print('PIN Set Successful !!')

    def deposit(self):
        #print('current pin ',self.pin)
        tmp = int(input('\nEnter the pin : '))
        if tmp==self.__pin:
            amount = int(input('Enter the Deposit amount : '))
            self.__bal+=amount
            print('Balance Updated Successfully')
            print('Balance : ',self.__bal)
        else:
            print('Incorrect Pin entered')

    # getter method
    def get_pin(self):
        return self.__pin

    # setter method
    def set_pin(self,new_pin):
        if type(new_pin)==str:
                self.__pin=new_pin
                print('PIN Changed')
            else:
                print('PIN not allowed. Use a new one !!')

    def withdraw(self):
        tmp = int(input('\nEnter the pin : '))
        if tmp==self.__pin:
            amount = int(input('Enter the Withdrawal amount : '))

            if amount<self.__bal and amount>0:
                self.__bal-=amount
                print('Balance Updated Successfully')
                print('Balance : ',self.__bal)
            else:
                print('Low Balance !!')
                print('Balance : ',self.__bal)
        else:
            print('Incorrect Pin entered')

    def check_bal(self):
        tmp = int(input('\nEnter the pin : '))
        if tmp==self.__pin:
            print('Balance : ',self.__bal)
        else:
            print('Incorrect Pin entered')
```

In [3]:
```
idfc = Atmx()
```

```
ID of self  139863734085760
Secret ID is  1212

Welcome to WesternNewton Bank

press 1: to create pin
press 2: to deposit money
press 3: to withdraw
press 4: to check balance
press 5: to quit
1

enter your pin : 1234
PIN Set Successful !!

Welcome to WesternNewton Bank

press 1: to create pin
press 2: to deposit money
press 3: to withdraw
press 4: to check balance
press 5: to quit
5
```

In [4]:
```
idfc.get_pin()
```

Out[4]:
```
1234
```

In [5]:
```
idfc.set_pin(1111)
```
```
PIN not allowed. Use a new one !!
```

In [6]:
```
idfc.set_pin('4444')
```
```
PIN Changed
```

In [7]:
```
idfc.get_pin()
```
Out[7]:
```
'4444'
```

In [ ]:
```

```

Encapsulation :

data = self.__balance

methods = get_pin, set_pin

[data member + fuctn 2 methods] Encapsulate

It describes the idea of wrapping data and the methods that work on data within one unit.

This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data

In [ ]:
```

```

Getter & Setter Methods :

The primary use of getters and setters is to ensure data encapsulation in object-oriented programs.

We use getters & setters to add validation logic around getting and setting a value.

Setter: The setter is a method that is used to set the value of private attributes in a class.

Getters: These are the methods used in Object-Oriented Programming (OOPS) which helps to access the private attributes from a class.

In [ ]:
```

```

Pass By Reference :
Pass by reference means that you have to pass the reference to a variable which refers that the variable already exists in memory consisting adress locn of the obj

- Pass By Reference mai mutable data type ko bhejoge toh original data types mai permanent changes ho jayenge & immutable data type bhejoge toh changes nhi honge

- To aboid this apan ko hamesa clone bhejna chaiya ex. l = [1,2,3] send l[:] instead of l in function

In [25]:
```
class Customer:
    def __init__(self,name):
        self.name = name

def greet_customer(Customer):
    print('hey ',Customer.name)


c1 = Customer('Nitish')
print(c1)
```

```
<__main__.Customer object at 0x7f346bf9c1c0>
```

In [26]:
```
c1.name
```

Out[26]:
```
'Nitish'
```

In [28]:
```
# obj pass as argument

greet_customer(c1)
```

```
hey  Nitish
```

In [ ]:

In [ ]:

In [41]:
```
class Customer:
    def __init__(self,name,gender):
        self.name = name
        self.gender = gender

def greet_customer(Customer):
    if Customer.gender=='Male':
        print('Hey Mr',Customer.name)
    else:
        print('Hi Miss ',Customer.name)


c1 = Customer('Nitish','Male')
print(c1.name)
print(c1.gender)
```

```
Nitish
Male
```

In [42]:
```
greet_customer(c1)
```

```
Hey Mr Nitish
```

In [ ]:

In [39]:
```
c2 = Customer('Nishi','Female')
print(c2.name)
print(c2.gender)
```

```
Nishi
Female
```

In [40]:
```
greet_customer(c2)
```

```
Hi Miss  Nishi
```

In [ ]:

```
Returning an object from the fucntiona nd storing it into a variable
```

In [2]:
```
class Customer:
    def __init__(self,name):
        self.name = name

def greet(cust):
    print('Hello ',cust.name)

    c2 = Customer('Rajni')
    return c2


c1 = Customer('Harsh')
x = greet(c1)
```

```
Hello  Harsh
```

In [3]:
```
print(greet(x))
```

```
Hello  Rajni
<__main__.Customer object at 0x7f3670495240>
```

In [ ]:

In [ ]:

```
Passing the id of the obj inside the fuctn
```

In [47]:
```
class Customer:
    def __init__(self,name):
        self.name = name

def greet(cust):
    print(id(cust))
```

In [49]:
```
c1 = Customer('akash')
print(id(c1))
```

```
139863126330912
```

In [51]:
```
greet(c1)
```

```
139863126330912
```

In [ ]:

In [6]:
```python
a = 3
b = 10-7
print(id(a),id(b))
```

```
94025525593184 94025525593184
```

In [ ]:

In [52]:
```python
# pass by reference ::
```

```
Its working like aliasing
```

In [57]:
```python
a = 10
b = a

print(f'a = {a}  Id : {id(a)}')
print(f'b = {b}  Id : {id(b)}')
```

```
a = 10  Id : 9789248
b = 10  Id : 9789248
```

In [ ]:

```
If an obj is passed to the functn & if the functn made some changes to
the (object attributes / data members). Then the changes will also be
made to the orginal obj
```

In [67]:
```python
class Customer:
    def __init__(self,name):
        self.name = name

def greet(cust):
    cust.name = 'new name'
    print(cust.name)
```

In [68]:
```python
c1 = Customer('Nitish')
print(c1.name)
```

```
Nitish
```

In [69]:
```python
greet(c1)
```

```
new name
```

In [ ]:

```
Note ::
Objects of the class are also mutable like list, dict, sets
```

In [ ]:

```
How elements in  List is being changed ?
- List is mutable so changes would take place
- But in the case of tuple the changes will not take place
```

In [14]:
```
def change(l,item):
    print('Before : ',id(l),l)
    l.append(item)
    print('After : ',id(l),l)

l1 = [1,2,3,4]
print('Id of l1 : ',id(l1))
change(l1,5)

print('Original List ',l1)
```

```
Id of l1 :  139871789604424
Before :  139871789604424 [1, 2, 3, 4]
After :  139871789604424 [1, 2, 3, 4, 5]
Original List  [1, 2, 3, 4, 5]
```

In [ ]:

Sending a clone of the list inside the fuctn

In [15]:
```
def change(l,item):
    print('Before : ',id(l),l)
    l.append(item)
    print('After : ',id(l),l)

l1 = [1,2,3,4]
print('Id of l1 : ',id(l1))
change(l1[:],5)

print('Original List ',l1)
```

```
Id of l1 :  139871789317768
Before :  139871789604424 [1, 2, 3, 4]
After :  139871789604424 [1, 2, 3, 4, 5]
Original List  [1, 2, 3, 4]
```

In [ ]:

In [17]:
```
def change(l,item):
    print('Before : ',id(l),l)
    l+=(item,)
    print('After : ',id(l),l)

l1 = (1,2,3,4)
print('Id of l1 : ',id(l1))
change(l1,5)

print('Original Tuple ',l1)
```

```
Id of l1 :  139871788457240
Before :  139871788457240 (1, 2, 3, 4)
After :  139871866990928 (1, 2, 3, 4, 5)
Original Tuple  (1, 2, 3, 4)
```

In [ ]:

Looping through objects
- Accessing obj data types/ attributes using loop

In [ ]:

In [71]:
```
class Customer:
    def __init__(self,name):
        self.name = name

def greet(cust):
    print(id(cust))
```

In [73]:
```
c1 = Customer('ram')
c2 = Customer('mohan')
c3 = Customer('roy')
```

In [76]:
```
l = [c1,c2,c3]
for i in l:
    print(f'{i.name}  {id(i)}')
```

```
ram  139863115396576
mohan  139863115396960
roy  139863115396720
```

In [ ]:

In [ ]:

In [80]:
```
class Customer:
    def __init__(self,name,age):
        self.name = name
        self.age = age

    def intro(self):
        print(f'My name is {self.name} & i am {self.age} years old.')

def greet(cust):
    print(id(cust))
```

In [81]:
```
c1 = Customer('ram',21)
c2 = Customer('mohan',22)
c3 = Customer('roy',23)

l = [c1,c2,c3]
```

In [84]:
```
for i in l:
    i.intro()
```

```
My name is ram & i am 21 years old.
My name is mohan & i am 22 years old.
My name is roy & i am 23 years old.
```

In [85]:
```
# This things only works with Mutable data types
```

In [ ]:

```
Types of Variables :

1] Instance Variable : variable value jiske har obj ke lia alag hai
- are present inside constructor
-> access using self.
```

2] Static/Class Variable : variable jiski value, sare obj ke lia same hoou
-> access using classname.class/static var name

In [ ]:

Ex. Static/class variable

In [114]:
```
class Customer:
    customer_no = 1
    def __init__(self,name,age):
        self.name = name
        self.age = age
        self.cno = Customer.customer_no
        Customer.customer_no+=1

    def intro(self):
        print(f'\nCustomer no : {self.cno}')
        print(f'Name: {self.name}')
        print(f'Age: {self.age}')
```

In [115]:
```
c1 = Customer('ram',21)
c2 = Customer('mohan',22)
c3 = Customer('roy',23)
```

In [116]:
```
l = [c1,c2,c3]

for i in l:
    i.intro()
```

```

Customer no : 1
Name: ram
Age: 21

Customer no : 2
Name: mohan
Age: 22

Customer no : 3
Name: roy
Age: 23
```

In [ ]:

In [118]:
```
print(c1.customer_no)
print(c2.customer_no)
print(c3.customer_no)
```

```
4
4
4
```

In [ ]:

we dont want to give any user the permissin to change the value of the counter.

So to solve this issue we will be creating a get and set method & make our counter variable (static/class variable) private

In [1]:
```
class Customer:

    # static variable/class
    __customer_no = 1
    def __init__(self,name,age):
        self.name = name
        self.age = age
        Customer.__customer_no+=1

    def intro(self):
        print(f'Name: {self.name}')
        print(f'Age: {self.age}')

    @staticmethod
    def get_counter():
        print('Counter no : ',Customer.__customer_no)

    @staticmethod
    def update_counter(no):
        if type(no)==int:
            Customer.__customer_no=no
            print('counter updated')
        else:
            print('Invalid Input')
```

In [2]:
```
c1 = Customer('ram',21)
c2 = Customer('mohan',22)
c3 = Customer('roy',23)

l = [c1,c2,c3]

for i in l:
    i.intro()
```

```
Name:  ram
Age:  21
Name:  mohan
Age:  22
Name:  roy
Age:  23
```

In [3]:
```
# getting the value of counter using Getter method
```

In [4]:
```
(c1.get_counter())
(c2.get_counter())
(c3.get_counter())
```

```
Counter no :  4
Counter no :  4
Counter no :  4
```

In [166]:
```
# setting the value of counter using Setter method
```

In [6]:
```
c1.update_counter(11)

(c1.get_counter())
```
```
counter updated
Counter no :  11
```

In [7]:
```
# as we can see the value of counter has been updated


(c1.get_counter())
(c2.get_counter())
(c3.get_counter())
```
```
Counter no :  11
Counter no :  11
Counter no :  11
```

In [ ]:

```
Static Method
The @staticmethod is a built-in decorator that defines a static method
in the class in Python.

- access without obj
- use when dealing with static varibale

The static method cannot access the class attributes or the instance
attributes.

The static method can be called using ClassName.MethodName() and also
using object.MethodName().

It can return an object of the class
```

In [ ]:

In [ ]:

https://pythonguides.com/python-pass-by-reference-or-value/ (https://pythonguides.com/python-pass-by-reference-or-value/)

In [ ]:

```
Types of Relationship In Python:

1] Aggregation {Has-A}
2] Inheritance {Is-A}

2:32
```

In [ ]:

In [ ]:
```
Aggregation ex
```

In [180]:
```
class Customer:
    def __init__(self,name,gender,address):
        self.name = name
        self.gender = gender
        self.address = address

    def intro(self):
        print(f'Name = {self.name}')
        print(f'Gender = {self.gender}')
        print(f'Address = {self.address}')

class Address:
    def __init__(self,city,state,pincode):
        self.city = city
        self.state = state
        self.pincode = pincode

```

In [181]:
```
add = Address('Ahmedabad','Gujarat',234234)
c1 = Customer('Harsh','Male',add)
```

In [179]:
```
c1.intro()
```

```
Name = Harsh
Gender = Male
Address = <__main__.Address object at 0x7f346bbf6940>
```

In [183]:
```
c1.address.city
```

Out[183]:
```
'Ahmedabad'
```

In [184]:
```
c1.address.pincode
```

Out[184]:
```
234234
```

In [185]:
```
c1.address.state
```

Out[185]:
```
'Gujarat'
```

In [ ]:

In [ ]:

In [198]:
```
class Customer:
    def __init__(self,name,gender,address):
        self.name = name
        self.gender = gender
        self.address = address

    def change_profile(self,new_name,new_gender,new_city,new_state,new_pi
        self.name = new_name
        self.gender = new_gender
        self.address.change_add(new_city,new_state,new_pincode)

    def intro(self):
        print(f'Name = {self.name}')
        print(f'Gender = {self.gender}')
        print(f'Address = {self.address}')

class Address:
    def __init__(self,city,state,pincode):
        self.city = city
        self.state = state
        self.pincode = pincode

    def change_add(self,new_city,new_state,new_pincode):
        self.city = new_city
        self.state = new_state
        self.pincode = new_pincode
```

In [199]:
```
add = Address('Ahmedabad','Gujarat',234234)
c1 = Customer('Harsh','Male',add)
```

In [200]:
```
c1.intro()
```

```
Name = Harsh
Gender = Male
Address = <__main__.Address object at 0x7f346bc8cb20>
```

In [201]:
```
c1.name
```

Out[201]:
```
'Harsh'
```

In [202]:
```
c1.address
```

Out[202]:
```
<__main__.Address at 0x7f346bc8cb20>
```

In [203]:
```
c1.address.city
```

Out[203]:
```
'Ahmedabad'
```

In [ ]:

In [205]:
```
# updating the profile details

c1.change_profile('Rin','female','pune','rishi',123123)
```

In [207]:
```
c1.intro()
```

```
Name = Rin
Gender = female
Address = <__main__.Address object at 0x7f346bc8cb20>
```

In [208]:
```
c1.address
```

Out[208]:
```
<__main__.Address at 0x7f346bc8cb20>
```

In [209]:
```
c1.address.city
```

Out[209]:
```
'pune'
```

In [210]:
```
c1.address.pincode
```

Out[210]:
```
123123
```

2:43

In [ ]:

Inheritance 2:44, 2:50
- DRY [Dont Repeat Yourself] concept
- Biggest advantage : code reusability
- Inherit -> data members, member functn : Methods, constructor
- Private methods are not inherited

In [9]:
```
class User:
    def login(self):
        print('login')

    def register(self):
        print('register')

class Student(User):
    def enroll(self):
        print('enroll')

    def review(self):
        print('review')
```

As we can see Student class can access his Methods as well as the User class methods

- Parent can't access the methods of the child class but the student can do it viceversa

In [10]:
```
s1 = Student()
```

In [12]:
```
s1.enroll()
```

```
enroll
```

In [13]:
```
s1.review()
```

```
review
```

In [14]:
```
s1.login()
```
```
login
```

In [15]:
```
s1.register()
```
```
register
```

In [ ]:

```
2:55 : class diagram for inheritance
```

In [ ]:

```
=> If the child class does not have any construcutor then the parent
class constructor will be called
```

In [17]:
```
class Phone:
    def __init__(self,price,brand,camera):
        print('Phone class constructor')
        self.price = price
        self.brand = brand
        self.camera = camera

class Smartphone(Phone):
    pass
```

In [18]:
```
realmex = Smartphone(12000,'realme','12mp')
```
```
Phone class constructor
```

In [19]:
```
realmex.price
```
Out[19]:
```
12000
```

In [20]:
```
realmex.brand
```
Out[20]:
```
'realme'
```

In [21]:
```
realmex.camera
```
Out[21]:
```
'12mp'
```

In [ ]:

```
=> Inheriting private Members
- obj of child class cannot access the hidden/Private data members of
the parent class
```

In [24]:
```
class Phone:
    def __init__(self,price,brand,camera):
        print('Phone class constructor called')
        self.price = price
        self.__brand = brand
        self.camera = camera

class Smartphone(Phone):
    pass
```

```
here the brand data member of the parent class is set as private we will
try to access it
```

In [25]:
```
p1 = Smartphone(1000,'Elephone','23mp')
```

```
Phone class constructor called
```

In [26]:
```
p1.price
```

Out[26]:
```
1000
```

In [27]:
```
p1.__brand
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [27], in <module>
----> 1 p1.__brand

AttributeError: 'Smartphone' object has no attribute '__brand'
```

In [28]:
```
p1.brand
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [28], in <module>
----> 1 p1.brand

AttributeError: 'Smartphone' object has no attribute 'brand'
```

In [ ]:

In [33]:
```
class Parent:
    def __init__(self,secret):
        self.__secret = secret

    def show_secret(self):
        return self.__secret

class Child(Parent):
    def show(self):
        print('This is a child class')
```

```
In [36]: c1 = Child(123)
         c1.show_secret()

Out[36]: 123
```

```
In [38]: c1.show()

         This is a child class
```

```
In [ ]:
```

If the child has a contructor then the parent constructor is not called & if the child does not have a constructor then the constructor of the parent class is called automatically

```
In [39]: class Parent:
             def __init__(self,secret):
                 self.__secret = secret

             def get_secret(self):
                 return self.__secret

         class Child(Parent):
             def __init__(self,name,secret):
                 self.__name = name

             def get_name(self):
                 return self.__name
```

```
In [ ]:
```

```
In [40]: c1 = Child('harsh',123)
```

```
In [42]: c1.get_name()

Out[42]: 'harsh'
```

```
In [43]: c1.get_secret()

         ----------------------------------------------------------------------
         ----
         AttributeError                            Traceback (most recent call l
         ast)
         Input In [43], in <module>
         ----> 1 c1.get_secret()

         Input In [39], in Parent.get_secret(self)
               5 def get_secret(self):
         ----> 6     return self.__secret

         AttributeError: 'Child' object has no attribute '_Parent__secret'
```

```
In [ ]:
```

In [4]:
```
class A:
    def __init__(self):
        self.var1 = 100

    def display1(self,var1):
        print('class A ',self.var1)

class B(A):
    def display2(self,var1):
        print('class B ',self.var1)
```

In [5]:
```
c1 = B()
```

In [6]:
```
c1.display1(100)
```

```
class A  100
```

In [ ]:
```

```

```
Super Keyword :
- super().classMethod()
- super(), this should be ur first statement after constructor/ Method

- using this keyword parents method, parent constructor invoke
- we cant access even attribute using super
- we can only use super inside the class not outside the class
```

In [13]:
```
class Phone:
    def __init__(self,price,brand,camera):
        print('Inside phone constructor')
        self.__price = price
        self.brand = brand
        self.camera = camera

    def buy(self):
        print('Buying a phone : Phone class')

class Smartphone(Phone):
    def buy(self):
        print('Buying a smart-phone : child class')
        super().buy()
```

In [14]:
```
s = Smartphone(20000,'INOX','12 mp')
```

```
Inside phone constructor
```

In [15]:
```
s.buy()
```

```
Buying a smart-phone : child class
Buying a phone : Phone class
```

In [ ]:
```

```

In [16]:
```
# u can't use constructor outside the class
s.super().buy()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [16], in <module>
      1 # u can't use constructor outside the class
----> 2 s.super().buy()

AttributeError: 'Smartphone' object has no attribute 'super'
```

In [ ]:

In [17]:
```
class Phone:
    def __init__(self,price,brand,camera):
        print('Inside phone constructor')
        self.price = price
        self.brand = brand
        self.camera = camera

    def get_phone_details(self):
        print('Price ',self.price)
        print('Brand ',self.brand)
        print('Camera ',self.camera)

class Smartphone(Phone):
    def __init__(self,os,ram,price,brand,camera):
        print('Pehle yaha')
        super().__init__(price,brand,camera)
        print('Inside Smartphone constructor')
        self.os = os
        self.ram = ram

    def get_smarthphone_details(self):
        print('Os ',self.os)
        print('Ram ',self.ram)
```

In [18]:
```
s1 = Smartphone('nogut','12 gb',12000,'NIOX','32mp')
```

```
Pehle yaha
Inside phone constructor
Inside Smartphone constructor
```

In [19]:
```
s1.get_phone_details()
```

```
Price  12000
Brand  NIOX
Camera  32mp
```

In [20]:
```
s1.get_smarthphone_details()
```

```
Os  nogut
Ram  12 gb
```

In [ ]:

Examples on super() keyword

In [21]:
```
class Parent:
    def __init__(self,secretno):
        self.__secretno = secretno

    def show_no(self):
        return self.__secretno

class Child(Parent):
    def __init__(self,secretno,secretword):
        super().__init__(secretno)
        self.__secretword = secretword

    def show_word(self):
        return self.__secretword
```

In [22]:
```
c1 = Child(121,'wow')
```

In [23]:
```
c1.show_word()
```

Out[23]:
```
'wow'
```

In [25]:
```
c1.show_no()
```

Out[25]:
```
121
```

In [ ]:

The self : keyword is used to represent an instance (object) of the given class.

In [ ]:

We can access the attribute of the parent class inside the method of child class. as self keyword is used to represent an instance (object) of the given class.

In [31]:
```
class Parent:
    def __init__(self):
        self.no = 101

class Child(Parent):
    def __init__(self):
        super().__init__()
        self.msg = 'Hii'

    def show(self):
        print(self.msg,
              self.no)
```

In [32]:
```
c1=Child()
```

In [33]:
```
c1.show()
```

```
Hii 101
```

In [ ]:

In [37]:
```
class Parent:
    def __init__(self):
        self.__now = 100

    def show(self):
        print('Parent data member : ',self.__now)

class Child:
    def __init__(self):
        super().__init__()
        self.__time = 200

    def show(self):
        print('child data member : ',self.__time)
```

In [41]:
```
p1 = Parent()
p1.show()
```

```
Parent data member :  100
```

In [42]:
```
c1 = Child()
c1.show()
```

```
child data member :  200
```

In [38]:

In [ ]:

Polymorphism
- Method Overriding
- Method Overloading
- Operator Overloading

=> Method Overriding
- If the same method is present in both the parent class and the child class and the child class is inheriting from the parent class.

- If the method is called from the child class then the - method present in the child class will be executed and given priority

In [ ]:

Dynamically Add Instance Variable to a Object :

We can add instance variables from the outside of class to a particular object. Use the following syntax to add the new instance variable to the object.

object_referance.variable_name = value

In [ ]:

Inheritance :
Iheritance is the capability of one class to derive or inherit the properties from another class.

```
Benefits of inheritance are:

- It represents real-world relationships well.
- It provides the reusability of a code. We don’t have to write the same
code again and again. Also, it allows us to add more features to a class
without modifying it.
- It is transitive in nature, which means that if class B inherits from
another class A, then all the subclasses of B would automatically
inherit from class A.
```

In [ ]:

```
Types of inheritance :

    Single inheritance.
    Multi-level inheritance.
    Multiple inheritance.
    Hierarchical Inheritance.
    Hybrid Inheritance.
```

In [1]:

```python
# Single level inheritancs

class Parent:
    def __init__(self,fname,lname):
        self.fname = fname
        self.lname = lname

    def get_details(self):
        print(f'My name is {self.fname} {self.lname}')

class Child(Parent):
    pass
```

In [2]:

```python
c1 = Child('Happy','Harsh')
```

In [4]:

```python
c1.get_details()
```

```
My name is Happy Harsh
```

In [ ]:

In [5]:
```
# Multi-level Inheritance

class Product:
    def review(self):
        print('review on the way')

class Phone(Product):
    def __init__(self,name,brand,price):
        self.name = name
        self.brand = brand
        self.__price = price

    def get_details(self):
        print('Model ',self.name)
        print('Brand ',self.brand)
        print('price ',self.__price)

class Smartphone(Phone):
    pass
```

In [6]:
```
s1 = Smartphone('x1','Nothing','12000')
```

In [7]:
```
s1.get_details()
```

```
Model  x1
Brand  Nothing
price  12000
```

In [8]:
```
s1.review()
```

```
review on the way
```

In [ ]:

In [9]:
```
p1 = Phone('s7 edge','Samsung',32000)
```

In [10]:
```
p1.get_details()
```

```
Model  s7 edge
Brand  Samsung
price  32000
```

In [11]:
```
p1.review()
```

```
review on the way
```

In [ ]:

In [12]:
```
#  Hierarchical Inheritance

class Product:
    def review(self):
        print('review on the way')

class Phone(Product):
    def __init__(self,name,brand,price):
        self.name = name
        self.brand = brand
        self.__price = price

    def get_details(self):
        print('Model ',self.name)
        print('Brand ',self.brand)
        print('price ',self.__price)

class Smartphone(Phone):
    pass

class FeaturePhone(Phone):
    pass
```

In [13]:
```
s1 = Smartphone('s7 edge','samsung',32000)
```

In [14]:
```
s1.get_details()
```

```
Model  s7 edge
Brand  samsung
price  32000
```

In [15]:
```
s1.review()
```

```
review on the way
```

In [ ]:
```

```

In [16]:
```
f1 = FeaturePhone('3310','NOkia',5000)
```

In [17]:
```
f1.get_details()
```

```
Model  3310
Brand  NOkia
price  5000
```

In [18]:
```
f1.review()
```

```
review on the way
```

In [ ]:
```

```

In [23]:
```
# Mulitple Inheritance

class Phone:
    def __init__(self,name,brand,price):
        self.name = name
        self.brand = brand
        self.__price = price

    def get_details(self):
        print('Model ',self.name)
        print('Brand ',self.brand)
        print('price ',self.__price)

class Product:
    def review(self):
        print('\nCustomer review')
        print(f'For model {self.name}')
# using multiple inheritance here
class SmartPhone(Phone,Product):
    pass
```

In [24]:
```
s1 = SmartPhone('Iphone XR','Apple',45000)
```

In [25]:
```
s1.review()
```

```

Customer review
For model Iphone XR
```

In [26]:
```
s1.get_details()
```

```
Model  Iphone XR
Brand  Apple
price  45000
```

In [ ]:
```
```

Method Resolution Order ::

- In python, method resolution order defines the order in which the base classes are searched when executing a method.

First, the method or attribute is searched within a class and then it follows the order we specified while inheriting.

This order is also called Linearization of a class and set of rules are called MRO(Method Resolution Order)

In [38]:
```
# Mulitple Inheritance explaning MRO

class Phone:
    def __init__(self,name,brand,price):
        self.name = name
        self.brand = brand
        self.__price = price

    def buy(self):
        print('-- Phone Class --')
        print('Model ',self.name)
        print('Brand ',self.brand)
        print('price ',self.__price)

class Product:
    def review(self):
        print('\nCustomer review')
        print(f'For model {self.name}')

    def buy(self):
        print('-- Product Class --')
        print('Model ',self.name)
        print('Brand ',self.brand)
        print('price ',self.__price)

# using multiple inheritance here MRO
class SmartPhone(Phone,Product):
    pass
```

In [39]:
```
s1 = SmartPhone('Reno x5','OPPO',23000)
```

In [40]:
```
s1.buy()
```

```
-- Phone Class --
Model  Reno x5
Brand  OPPO
price  23000
```

In [ ]:

In [44]:
```
# Mulitple Inheritance explaning MRO

class Phone:
    def __init__(self,name,brand,price):
        self.name = name
        self.brand = brand
        self.__price = price

    def buy(self):
        print('-- Phone Class --')
        print('Model ',self.name)
        print('Brand ',self.brand)

class Product:
    def review(self):
        print('\nCustomer review')
        print(f'For model {self.name}')

    def buy(self):
        print('-- Product Class --')
        print('Model ',self.name)
        print('Brand ',self.brand)

# using multiple inheritance here MRO
class SmartPhone(Product,Phone):
    pass
```

In [45]:
```
s2 = SmartPhone('Reno x5','OPPO',23000)
```

In [46]:
```
s2.buy()
```

```
-- Product Class --
Model  Reno x5
Brand  OPPO
```

In [ ]:

In [47]:
```
# Multi-level Inheritance

class A:
    def m1(self):
        return 20

class B(A):
    def m1(self):
        return 30

    def m2(self):
        return 40

class C(B):
    def m2(self):
        return 20
```

In [49]:
```
obj1 = A()
obj2 = B()
obj3 = C()
```

In [50]:
```
# 20 + 30 + 20
obj1.m1() + obj2.m1() + obj3.m2()
```

Out[50]:
```
70
```

In [ ]:

In [55]:
```
class A:
    def m1(self):
        return 20

class B(A):
    def m1(self):
        val = self.m1() + 10
        return val
```

In [56]:
```
obj1 = B()
```

In [57]:
```
obj1.m1()
```

```
------------------------------------------------------------------------
----
RecursionError                            Traceback (most recent call l
ast)
Input In [57], in <module>
----> 1 obj1.m1()

Input In [55], in B.m1(self)
      6 def m1(self):
----> 7     val = self.m1() + 10
      8     return val

Input In [55], in B.m1(self)
      6 def m1(self):
----> 7     val = self.m1() + 10
      8     return val

    [... skipping similar frames: B.m1 at line 7 (2970 times)]

Input In [55], in B.m1(self)
      6 def m1(self):
----> 7     val = self.m1() + 10
      8     return val

RecursionError: maximum recursion depth exceeded
```

In [ ]:

Polymorphism
- Method Overriding
- Method Overloading
- Operator Overloading

2] Method Overloading

Methods in Python can be called with zero, one, or more parameters. This process of calling the same method in different ways is called method overloading.

- one method behave in diffrent manner when different no of arguments are passed.

```
In [62]: class GetArea:
             def area(self,r):
                 self.r = r
                 print(f'Area of circle is {3.14*self.r *self.r }')

             def area(self,h,b):
                 self.h = h # height
                 self.b = b # breadth
                 print(f'Area of Rectangle is {self.h*self.b}')
```

```
In [63]: q1 = GetArea()
```

```
In [64]: q1.area(22)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [64], in <module>
----> 1 q1.area(22)

TypeError: area() missing 1 required positional argument: 'b'
```

Note :: As we can see the method has been over-rided by the same area method written below. Python does not support method-Over riding

```
In [ ]:
```

But we can also use Over-riding in a smarter way

```
In [68]: class GetArea:
             def area(self,a,b=0):
                 if b==0:
                     self.r = a
                     print(f'Area of circle is {3.14*self.r *self.r }')
                 else:
                     self.h = a # height
                     self.b = b # breadth
                     print(f'Area of Rectangle is {self.h*self.b}')
```

```
In [69]: q1 = GetArea()
```

```
In [70]: q1.area(4)
```

```
Area of circle is 50.24
```

```
In [71]: q1.area(4,2)
```

```
Area of Rectangle is 8
```

In [ ]:

Polymorphism
- Method Overriding
- Method Overloading
- Operator Overloading

3] Operator Over-loading

Operator Overloading means giving extended meaning beyond their predefined operational meaning.

or.

Python that allows the same operator to have different meaning according to the context is called operator overloading

For example operator + is used to add two integers as well as join two strings and merge two lists.

It is achievable because ‘+’ operator is overloaded by int class and str class.

You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called Operator Overloading.

In [77]:
```python
from fractions import Fraction
x = Fraction(1,2)
y = Fraction(3,4)
print(x+y) # this is fraction addn not mathematical addn
```

```
5/4
```

In [ ]:

In [ ]: