In [27]:
```
# Encapsulation --> public, private(can't be accessible outside class)
class ATM:
    __original_username = 'vaishnavi' # to change a property to private, add 2 _
    __original_pin_number = '1234'

    def __init__(self, username, pin_number):
        self.username = username
        self.pin_number = pin_number

    # getters
    def get_original_username(self):
        if(self.validate()):
            return self.__original_username
        return 'Unauthorized'

    def get_original_pin_number(self):
        if(self.validate()):
            return self.__original_pin_number
        return 'Unauthorized'

    # setters
    def set_original_username(self, new_username):
        if(self.validate()):
            self.__original_username = new_username
            return f'username changed successfully- {self.__original_username}'
        return 'Unauthorized'

    def validate(self):
        if(self.username == ATM.__original_username and self.pin_number == ATM.__origi
            return True
        return False

Meena = ATM('meena', '6789')
# print(ATM.__original_username, ATM.__original_pin_number)
Vaishnavi = ATM('vaishnavi', '1234')
# print(Vaishnavi.__original_username, Vaishnavi.__original_pin_number)

print(Vaishnavi.get_original_username())
# print(Meena.get_original_username())

# print(Vaishnavi.get_original_pin_number())
# print(Meena.get_original_pin_number())

# print(Meena.set_original_username('meena'))
print(Vaishnavi.set_original_username('Vasanth'))

print(Vaishnavi.get_original_username())
```

```
vaishnavi
username changed successfully- Vasanth
Vasanth
```

In [32]:
```
# Inheritance
class Item:
    # class property - constant for all objects
    discount = 0.2 # 20% discount

    def __init__(self, name, price = 0, quantity = 1, is_case_available = False):
```

```
        assert price >= 0, f'Invalid price -> {price}. It should be greater than or eq
        assert quantity > 0, f'Invalid quantity -> {quantity}. It should be greater th
        # object / self properties
        self.name = name
        self.price = price
        self.quantity = quantity
        self.is_case_available = is_case_available

    def calculate_total_price(self):
        return self.price * self.quantity

    def apply_discount(self):
        self.price = self.price - (self.price * self.discount) # by default it will ta

Phone = Item('Iphone 14', 70000, 1)
print(Phone.__dict__)

Google_pixel = Item('Pixel 6', 40000, 1, True)
print(Google_pixel.__dict__)

tomato = Item("tomato", 30, 2)
print(tomato.__dict__)
```

```
{'name': 'Iphone 14', 'price': 70000, 'quantity': 1, 'is_case_available': False}
{'name': 'Pixel 6', 'price': 40000, 'quantity': 1, 'is_case_available': True}
{'name': 'tomato', 'price': 30, 'quantity': 2, 'is_case_available': False}
```

In [43]:

```
# Inheritance
class Item:
    # class property - constant for all objects
    discount = 0.2 # 20% discount

    def __init__(self, name, price = 0, quantity = 1):
#         print('item init ----')

        assert price >= 0, f'Invalid price -> {price}. It should be greater than or eq
        assert quantity > 0, f'Invalid quantity -> {quantity}. It should be greater th
        # object / self properties
        self.name = name
        self.price = price
        self.quantity = quantity

    def calculate_total_price(self):
        return self.price * self.quantity

    def apply_discount(self):
        self.price = self.price - (self.price * self.discount) # by default it will ta

# inherit Item to Phone
class Phone(Item):

    def __init__(self, name, price = 0, quantity = 1, is_case_available = False):
#         print('phone init ----')
        # super - parent --> Item
        super().__init__(name, price, quantity)
        self.is_case_available = is_case_available

iphone = Phone('Iphone 14', 60000, 2)
```

```
print(iphone.__dict__)
print(iphone.calculate_total_price())
```

```
{'name': 'Iphone 14', 'price': 60000, 'quantity': 2, 'is_case_available': False}
```

In [65]:
```
# types of inheritance -> single, multilevel, multiple, heirarchical

# single level inheritance
class Parent:
    def __init__(self):
        self.name = 'Parent'

    def display_name(self):
        print(self.name)

class Child(Parent):
    def __init__(self):
        super().__init__()
        self.name = 'child'


father = Parent()
# father.display_name()

ankur = Child()
# ankur.display_name()
```

In [78]:
```
# multi level inheritance
class GrandParent:
    def __init__(self):
        self.house = 'Own house'

class Parent(GrandParent):
    def __init__(self):
        super().__init__()
        self.car = 'Maruti'

class Child(Parent):
    def __init__(self):
        super().__init__()
        self.bike = 'Yamaha'

gp = GrandParent()
parent = Parent()
child = Child()
# print(gp.house)
# print(parent.house)

print(child.house)
print(child.car)
print(child.bike)
```

```
Own house
Maruti
Yamaha
```

In [99]:
```
# multiple inheritance
class Father:
    def __init__(self):
        self.name = 'Father'
```

```
    def play(self):
        print('Playing with father')

class Mother:
    def __init__(self):
        self.name = 'Mother'
        self.love = "mother's love"

    def play(self):
        print('playing with mother')

class Child(Mother, Father):
    def __init__(self):
#         super().__init__()
        Father.__init__(self)
        Mother.__init__(self)

    def display(self):
        print(self.name, self.love)

child = Child()
# child.display()
child.play()
```

```
playing with mother
```

In [106…

```
# heirarchical inheritance
class Parent:
    def __init__(self):
        self.name = 'Parent'

    def display_parent_name(self):
        print(self.name)

class Child1(Parent):
    def __init__(self):
        super().__init__()
        self.country = 'UK'

class Child2(Parent):
    def __init__(self):
        super().__init__()
        self.country = 'India'

parent = Parent()
# print(parent.name)

child1 = Child1()
print(child1.country)
child1.display_parent_name()


child2 = Child2()
print(child2.country)
child2.display_parent_name()
```

```
UK
Parent
India
Parent
```

In [115…

```python
# Polymorphism
class Parent:
    def play(self):
        print('parent playing')

class Child(Parent):
    # method overriding
    def play(self):
        print('child playing')

child = Child()
child.play()

# method overloading is not supported in python
```

```
child playing
```

In [149…

```python
# Abstraction - not present in python by default
# Abstract Base Class(ABC) - cannot be instantiated(creating object)
# for a class to become Abstract class, it should have atleast 1 abstract method

from abc import ABC, abstractmethod

class Computer(ABC):

    @abstractmethod # decorator
    def process(self):
        pass

class Laptop(Computer):
    def play_games(self):
        print('playing games')

    def process(self):
        print('some process is running on laptop')

class Mobile:
    def play_games(self):
        print('playing games')

#     def process(self):
#         print('some process is running on mobile')

class Programmer:
    def work(self, device):
        print('creating application...')
        device.process()

macbook = Laptop()
# macbook.play_games()
iphone = Mobile()

iphone.play_games()

vasi = Programmer()
```

```
vasi.work(macbook)
# vasi.work(iphone)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [149], in <cell line: 32>()
     29         print('creating application...')
     30         device.process()
---> 32 macbook = Laptop()
     33 # macbook.play_games()
     34 iphone = Mobile()

TypeError: Can't instantiate abstract class Laptop with abstract method process
```

In [153…

```
# hands on - 2

# 1. Create a function named ‘factor’ that can only accept 1 argument. The function sh
# return the factorial of that number.

def factor(num):
    fact = 1
    if (num < 0): return 'Invalid'
    elif (num < 2): return 1
    else:
        for i in range(1, num+1):
            fact = fact * i

        return fact

print(factor(5))
print(factor(1))
print(factor(0))
print(factor(-5))
```

```
120
1
1
Invalid
```

In [160…

```
name = 'vasi'
# for char in name:
#     print(char)

# iter
iterable = iter(name)
print(next(iterable))
print(next(iterable))
print(next(iterable))
print(next(iterable))
print(next(iterable))
```

```
v
a
s
i
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Input In [160], in <cell line: 11>()
      9 print(next(iterable))
     10 print(next(iterable))
---> 11 print(next(iterable))

StopIteration:
```

In [168…

```
# 2. Create a function named 'check_string', the function should accept a string data
# user and the function should check if the user input contains the letter 's' in it.
# the letter 's' then print- 'The string is containing the letter 's'', if not then pr
# doesn't contain the letter 's''.

def check_string():
    word = input('Enter the word to be checked: ')
    word = word.lower()
    is_present = False # default
    for char in word:
        if (char == 's'):
            is_present = True
            break

    if(is_present == True):
        print('word contains letter S')
    else:
        print('word does not contains letter S')

check_string()
```

```
Enter the word to be checked: VASI
word contains letter S
```

In [171…

```
# 3. Create a class named 'student' and inside the class, create a function named 'fur
# method should accept the user defined input and return that value.
# a. Create another method named- message() and that method should print the user
# defined input that we have defined in 'fun1'.

class Student:
    def __init__(self):
        self.word = None

    def fun1(self):
        word = input('Enter the input: ')
        self.word = word
        return self.word

    def message(self):
        print('printing message', self.word)

manorath = Student()
manorath.fun1()
manorath.message()
```

```
Enter the input: piyush
printing message piyush
```

In [173…

```
# 4. Create a lambda function that should double or multiply the number (that we will
# in the lambda function) by 2. Store the lambda function in a variable named 'double_
```

```
double_num = lambda num: num * 2
print(double_num(5))
print(double_num(15))
```

```
10
30
```

In [175…

```
# 5. Take the user input string and check whether that string is palindrome or not.
# MAM, MADAM, RACECAR

# forward string == reverse string
word = input('Enter the string: ')
if (word == word[::-1]):
    print('It is palindrome')
else:
    print('Not a palindrome')
```

```
Enter the string: vasanth
Not a palindrome
```

In [176…

```
# 6. Create a class named 'Super' and inside that class define a user-defined functior
# fun1.
# a. Inside the 'fun1' function, pass the message "This is function 1 in the Super cla
# in the print statement.

class Super:
    def fun1(self):
        print('This is function 1 in the Super class.')

s = Super()
s.fun1()
```

```
This is function 1 in the Super class.
```

In [180…

```
# 7. Create another class named 'Modified_Super' and inherit this class from the Super
# a. Inside the Modified_Super class, create a function named 'fun1' and pass the
# following message inside the print statement: 'This is function 1 in the Modified
# Super class.'
# b. Create another user-defined function named 'fun2' and pass the message: 'This is
# the 2nd function from the Modified Super class' in the print statement.
# c. After that, now create an object for the Modified_Super class and call the fun1()

class Super:
    def fun1(self):
        print('This is function 1 in the Super class.')

class ModifiedSuperClass(Super):
    def fun1(self):
        print('This is function 1 in the modified super class.')

    def fun2(self):
        print('This is function 2 in the modified super class.')

msc = ModifiedSuperClass()
msc.fun1()
msc.fun2()
```

```
This is function 1 in the modified super class.
This is function 2 in the modified super class.
```

In [184…

```
# 8. Create 2 methods named ‘Hello’. In the 1st Hello method, pass only one argument c
# this message: ‘This function only has 1 argument’. And in the 2nd Hello method, pass
# two arguments and pass this message: ‘This function has 2 arguments’.
# a. Try to call both the methods and analyze the output of both the methods.

# eg: method overloading
class Hello:

    def hello(self, one, two):
        print('This function only has 2 arguments')

    def hello(self, one):
        print('This function only has 1 argument')

h = Hello()
# h.hello('vasi', 'kumar')
h.hello('vasi')
# order of execution matters
```

```
This function only has 1 argument
```

In [186…

```
# 9. Create a method named ‘Sum’ that can accept multiple user inputs. Now add those u
# defined input values using for loop and the function should return the addition of t

class Sum:
    def get_sum(self):
        total = 0
        no_of_inputs = int(input("enter the no of inputs: "))
        for i in range(no_of_inputs):
            user_input = int(input("enter the no to get added: "))
            total = total + user_input
        print(total)

s = Sum()
s.get_sum()
```

```
enter the no of inputs: 5
enter the no to get added: 1
enter the no to get added: 2
enter the no to get added: 3
enter the no to get added:  4
enter the no to get added: 5
15
```

In [188…

```
# 10. Create a class named ‘Encapsulation’:
# a. Inside the class, first create a constructor. Inside the constructor, initialize
# originalValue variable as 10.
# b. After creating the constructor, define a function named ‘Value’ and this functior
# should return the variable that we have initialized in the constructor.
# c. Now create 2nd function named setValue, and pass an argument named
# ‘newValue’. The task of this function will be to replace the value of the
# originalValue variable by the value of the newValue variable.

class Encapsulation:
    def __init__(self):
        self.original_value = 10

    def value(self):
        return self.original_value
```

```
    def set_value(self, new_value):
        self.original_value = new_value

e = Encapsulation()
print(e.value())
e.set_value(20)
print(e.value())
```

```
10
20
```

In [ ]: