DAY-3,4

ENCAPSULATION

-->BINDING OF DATA

-->MAKES THE DATA PRIVATE

-->INCS SECURITY

-->IMP:getters & setters

class Student:

def init(self,marks): #default constructor

self.marks=marks

self.\_\_marks=marks #private

def getter(self):

return self.\_\_marks

def setter(self,marks): o/p:79

self.\_\_marks=marks

s1=Student(79)

#set the data

s1.setter(79)

#get the data

ans=s1.getter()

print(ans)

abstraction

->hiding the necessary information

->providing useful onfo only

->abc module,abstract method

->@abstaract method

1.what is meant by decarator? what is the use of it?

2.whta is meant by ABC module? #in this abc is module and ABC is class

PROGRAM:

from abc import ABC,abstractmethod o/p:car is moving

class four\_wheeler(ABC):

@abstractmethod

def engine():

return "hello this is process of making car"

class swift(four\_wheeler):

def car\_start():

return"car is moving"

car\_1=swift

ans=car\_1.car\_start()

print(ans)

#INHERITANCE:

1.SINGLE

2.MULTILEVEL

2.MULTIPLE

4.HIRARCHIAL

5.HYBRID

#SINGLE INHERITANCE

class father: #parent class

def father\_method():

return "This is father method"

#inheriting father class

class child(father);

def child\_method():

return "This is child method"

parent\_object=father

child\_object=child

print(parent\_object.father\_method())

print(child\_object.child\_method())

print(child\_object.father\_method())

#MULTIPLE INHERITANCE

class father:

def father\_method():

return "This is father method"

class mother:

def mother\_method():

return "This is mother method"

class child(father,mother):

def child\_method():

return "I have properties of mother and father" o/p:This is father method

This is mother method

I have properties of mother and father

This is father method

This is mother method

father\_obj=father

mother\_obj=mother

child\_obj=child

print(father\_obj.father\_method())

print(mother\_obj.mother\_method())

print(child\_obj.child\_method())

print(child\_obj.father\_method())

print(child\_obj.mother\_method())

#MULTilevel INHERITANCE

class grandfather:

def grandfather\_method():

return "This is grandfather method"

class father:

def father\_method():

return "This is father method"

class child(grandfather,father):

def child\_method():

return "I have properties of grand father and father"

grandfather\_obj=grandfather

father\_obj=father

child\_obj=child

print(grandfather\_obj.grandfather\_method())

print(father\_obj.father\_method())

print(child\_obj.child\_method())

print(child\_obj.grandfather\_method())

print(child\_obj.father\_method())

o/p:

This is grandfather method

This is father method

I have properties of grand father and father

This is grandfather method

This is father method

#HIRARCHIAL INHERITANCE

class grandfather:

def grandfather\_method():

return "This is grandfather method"

class grandmother:

def grandmother\_method():

return "This is grandmother method"

class father:

def father\_method():

return "This is father method"

class child(grandfather,grandmother,father):

def child\_method():

return "I have properties of grand father,grandmother and father"

grandfather\_obj=grandfather

grandmother\_obj=grandmother

father\_obj=father

child\_obj=child

print(grandfather\_obj.grandfather\_method())

print(grandmother\_obj.grandmother\_method())

print(father\_obj.father\_method())

print(child\_obj.child\_method())

print(child\_obj.grandfather\_method())

print(child\_obj.grandmother\_method())

print(child\_obj.father\_method())

o/p:

This is grandfather method

This is grandmother method

This is father method

I have properties of grand father,grandmother and father

This is grandfather method

This is grandmother method

This is father method

#HYBRID

class grandfather:

def grandfather\_method():

return "This is grandfather method"

class father:

def father\_method():

return "This is father method"

class mother:

def mother\_method():

return "This is mother method"

class child(grandfather, father,mother):

def child\_method():

return "I have properties of grand father,mother and father"

grandfather\_obj=grandfather

father\_obj=father

mother\_obj=mother

child\_obj=child

print(grandfather\_obj.grandfather\_method())

print(father\_obj.father\_method())

print(mother\_obj.mother\_method())

print(child\_obj.child\_method())

print(child\_obj.grandfather\_method())

print(child\_obj.mother\_method())

print(child\_obj.father\_method())

o/p:

This is grandfather method

This is father method

This is mother method

I have properties of grand father,mother and father

This is grandfather method

This is mother method

This is father method

#POLYMORPHISAM

class Animal:

def speak():

return "animal is speaking"

class Bird(Animal):

def speak():

return "bird is speaking"

animal\_object=Animal

bird\_object=Bird

print(animal\_object.speak())

print(bird\_object.speak())

o/p:

animal is speaking

bird is speaking

SUM OF SQUAE OF ANUMBERS

n=3645

sum=0

while(n>0):

digit=n%10

sum=(digit\*digit)+sum

n=n//10

print(sum)

O/P:86

DAY-4

stack: is a linear data structure , perform the O(1) and also insert and delete.

implement by using "list" only.

arr.append()

arr.pop

arr.insert

application:undo/redo,memory management,block chain,recursion,

stack follows the "LIFO".last in first out

\*push

\*pop

\*size

\*is empty

stack=[]

stack.append(10)

stack.append(20)

stack.append(30)

def push():

top=-1

if(top==len(stack)-1):

return"stack is full"

else:

return stack.append(40)

program:

def push(value):

top=-1

if(top==4):

return"stack is full"

else:

top=top+1

return stack.append(value)

stack=[10] #size as 5

push(20)

push(30)

push(40)

push(50)

push(60)

print(stack)

STACK:

stack=[]

top=-1

def push(value):

global top

stack.append(value)

top +=1

def pop():

global top

if top==-1:

priny("Stack is empty.nothing to pop.")

return

else:

stack.pop()

top-=1

def peek():

if top ==-1:

return "Stack is empty.No top element."

else:

return f"top element={stack[top]}"

def display():

if(top==-1):

print("empty")

else:

for i in range(top,-1,-1):

print(stack[i])

top=-1

push(10)

push(30)

push(50)

push(70)

pop()

pop()

print(peek())

display()

O/P:

top element=30

30

10

stack=[]

top=-1

def push(value):

global top

stack.append(value)

top +=1

def pop():

global top

if top==-1:

priny("Stack is empty.nothing to pop.")

return

else:

stack.pop()

top-=1

def peek():

if top ==-1:

return "Stack is empty.No top element."

else:

return f"top element={stack[top]}"

def display():

if(top==-1):

print("empty")

else:

for i in range(top,-1,-1):

print(stack[i])

while True:

print("\n---welcome--")

print("1.push")

print("2.pop")

print("3.peek")

print("4.display all the elements")

print("5.exit")

choise=int(input("Enter your choise"))

if choise==1:

value=int(input("Enter the element"))

push(value)

elif choise==2:

pop()

elif choise==3:

print(peek())

elif choise==4:

display()

else:

print("exit")

break

QUEUE:

is alinear data structure follows the FIFO

APLICATIONS:

1.ROUND ROBIN

2.JOB SCHEDULING

REFERENCE VARIABLES:

1.front

2.rear

TYPES:

1.Enqueue

2.Dequeue

program:

class queue:

def init(self,Q,value):

self.Q=Q

self.value=value

def enqueue(self,Q,value):

return Q.append(value)

QEUE:

class Queue:

def init(self):

self.items = deque()

def is\_empty(self):

return len(self.items) == 0

def enqueue(self, item):

self.items.append(item)

def dequeue(self):

if not self.is\_empty():

return self.items.popleft()

else:

return "Queue is empty"

def peek(self):

if not self.is\_empty():

return self.items[0]

else:

return "Queue is empty"

def display(self):

if self.is\_empty():

print("Queue is empty")

else:

print("Queue elements:")

for item in self.items:

print(item, end=" ")

print()

# Example usage

queue = Queue()

queue.enqueue(1)

queue.enqueue(2)

queue.enqueue(3)

queue.display() # Output: Queue elements: 1 2 3

print("Front element:", queue.peek()) # Output: Front element: 1

queue.dequeue()

queue.display() # Output: Queue elements: 2 3