**DAY1:BUILT IN PYTHON FUNCTIONS,OOPS,DECORATORS**

**Basic python inbuilt functions:**

**abs() function**

**all()**

**any()**

**ascii()**

Python ascii() function returns a string containing a printable representation of an object and escapes the non-ASCII characters in the string using \x, \u or \U escapes.

**int()**

**len()**

**max()**

**min()**

**bool()**

**chr()**

**dict()**

**divmod()**

**enumerate()**

**filter()**

**codes:**

#absolute value

var = float(input("Enter a number :"))

print('Absolute value of floating point number is:', abs(var))

#all function in lists

l1=[]

print(all(l1))

l2=[1,9,0,-3]

print(all(ele>0 for ele in l2))

l3=[1,2,3,4]

print(all(l3))

#any function

l4=[False,True,False]

print(any(l4))

l5=[1,0,-9,8]

print(any(ele>0 for ele in l5))

#ascii

print(ascii("#"))

s = "S a n j a n a"

print(ascii(s))

string='''Sanjana

Raghunath'''

print(ascii(string))

#bool()

x=bool(1)

print(x)

y=bool()

print(y)

bool(-1)

bool(0.0)

bool("string")

#chr()

num=32

print("Character value of Number is:",chr(num))

#dict()

dictionary=dict(flower="rose",fruit="banana")

print(dictionary)

#divmod()

print("5,10",divmod(5,10))

print("10,3",divmod(10,3))

#enumerate()

list1=["my","name","is"]

for i,name in list1:

    print(f"index of {i}",{name})

print(list(enumerate(list1)))

#filter()

def odd(n):

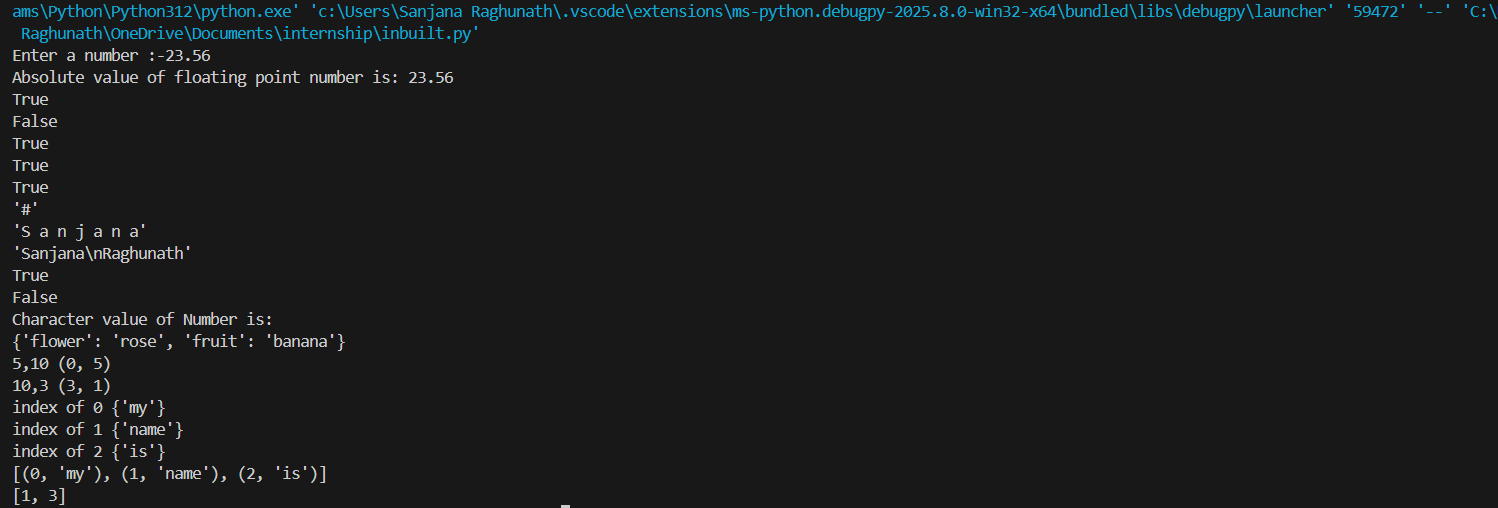
    return n%2!=0

list1=[1,2,3,4]

list2=filter(odd,list1)

print(list(list2))

**OUTPUT:**

****

**OOPS CONCEPT BASIC CODES:**

class Animal:

    species="Human"

    def \_\_init\_\_(self,name,age):

        self.name=name

        self.age=age

animal=Animal("Priya",30)

animalnew=Animal("Ravi",32)

print(animal.name)

print(animal.age)

print(animal.species)

print(animalnew.age)

print(animalnew.name)

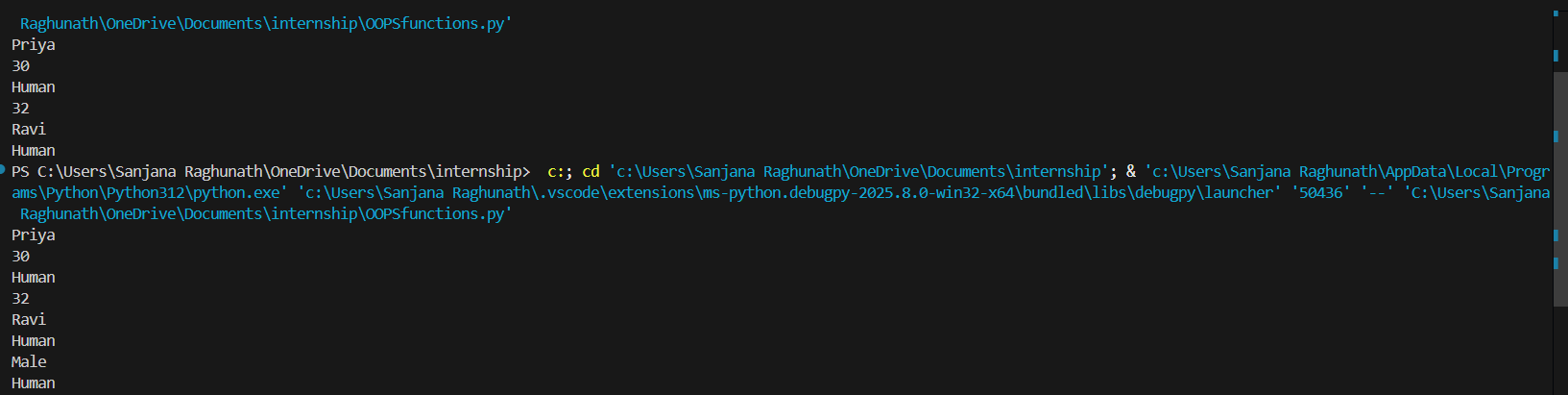
print(animalnew.species)

animalnew.species="Male"

print(animalnew.species)

print(animal.species)

OUTPUT:



**INHERITANCE:**

class Human:

    species="Human"

    def \_\_init\_\_(self,name,age,name1):

        self.name=name

        self.name1=name1

        self.age=age

    def display(self):

        print("The name of the person is:",{self.name})

class Men(Human):

    def play(self):

        print("Man's name:",self.name)

        print("Men pay cricket.")

class Women(Human):

    def playtennis(self):

        print("Woman's age and name:",self.age,self.name1)

        print("Women play tennis")

class children(Men,Women):

    def playhop(self):

        print("Child's name:",self.name)

        print("Children play hopscotch")

men=Men("antony","Jessi",29)

men.display()

men.play()

women=Women("mark","jessica",30)

women.display()

women.playtennis()

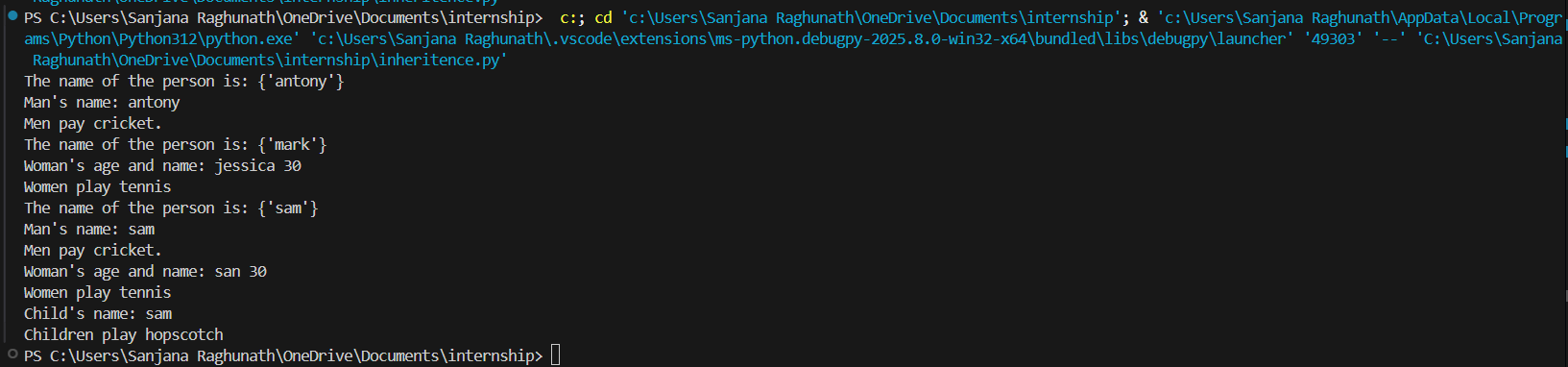
child=children("sam","san",30)

child.display()

child.play()

child.playtennis()

child.playhop()



**POLYMORPHISM:**

class Dog:

def speak(self):

return "Woof!"

class Cat:

def speak(self):

return "Meow!"

def animal\_sound(animal):

print(animal.speak())

dog = Dog()

cat = Cat()

animal\_sound(dog) # Output: Woof!

animal\_sound(cat) # Output: Meow!

2)METHOD OVERRODING(USING INHERITANCE)

class Animal:

def speak(self):

return "Some sound"

class Dog(Animal):

def speak(self):

return "Bark"

class Cat(Animal):

def speak(self):

return "Meow"

animals = [Dog(), Cat(), Animal()]

for animal in animals:

print(animal.speak())

3)OPERATOR OVERLOADING

class Vector:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_add\_\_(self, other):

return Vector(self.x + other.x, self.y + other.y)

def \_\_str\_\_(self):

return f"({self.x}, {self.y})"

v1 = Vector(2, 4)

v2 = Vector(3, 6)

v3 = v1 + v2

print(v3) # Output: (5, 10)

ABSTRACTION:

1)with one abstract method:

from abc import ABC, abstractmethod

class Animal(ABC):

@abstractmethod

def make\_sound(self):

pass

class Dog(Animal):

def make\_sound(self):

print("Bark")

class Cat(Animal):

def make\_sound(self):

print("Meow")

dog = Dog()

cat = Cat()

dog.make\_sound() # Output: Bark

cat.make\_sound() # Output: Meow

with both abstract and concrete methods

from abc import ABC, abstractmethod

class Vehicle(ABC):

@abstractmethod

def start(self):

pass

def fuel\_type(self):

print("Petrol or Diesel")

class Car(Vehicle):

def start(self):

print("Car engine started")

car = Car()

car.start() # Output: Car engine started

car.fuel\_type() # Output: Petrol or Diesel

DECORATORS:

BASIC DECORATRS:

def decorator\_function(original\_function):

def wrapper\_function():

print("Wrapper executed before", original\_function.\_\_name\_\_)

original\_function()

print("Wrapper executed after", original\_function.\_\_name\_\_)

return wrapper\_function

@decorator\_function # This is the decorator

def say\_hello():

print("Hello!")

say\_hello()

WITH ARGUMENTS:

def decorator\_func(func):

def wrapper(\*args, \*\*kwargs):

print("Function is being called with:", args, kwargs)

return func(\*args, \*\*kwargs)

return wrapper

@decorator\_func

def add(a, b):

return a + b

print(add(5, 10))

Using functools.wraps()

from functools import wraps

def my\_decorator(func):

@wraps(func)

def wrapper(\*args, \*\*kwargs):

print("Calling", func.\_\_name\_\_)

return func(\*args, \*\*kwargs)

return wrapper

@my\_decorator

def greet():

print("Hi!")

greet()