**1.Single Responsibility principle**

class Person:

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

self.name = name

def \_\_repr\_\_(self):

return f'Person(name={self.name})'

class PersonDB:

def save(self, person):

print(f'Save the {person} to the database')

if \_\_name\_\_ == '\_\_main\_\_':

p = Person('Ritu')

db = PersonDB()

db.save(p)

**2.Open Closed Principle**

from abc import ABC, abstractmethod

class Person:

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

self.name = name

def \_\_repr\_\_(self):

return f'Person(name={self.name})'

class PersonStorage(ABC):

@abstractmethod

def save(self, person):

pass

class PersonDB(PersonStorage):

def save(self, person):

print(f'Save the {person} to database')

class PersonJSON(PersonStorage):

def save(self, person):

print(f'Save the {person} to a JSON file')

class PersonXML(PersonStorage):

def save(self, person):

print(f'Save the {person} to a XML file')

if \_\_name\_\_ == '\_\_main\_\_':

person = Person('Ritu')

storage = PersonXML()

storage.save(person)

**3.Liskov Substitution Principle**

class kitchenAppliance():

def on():

pass

def off():

pass

class kitchenApplianceWithTemp(kitchenAppliance):

def set\_temp():

pass

class Toaster( kitchenApplianceWithTemp):

def on():

print("Turn on Toaster")

def off():

print("Turn off Toaster")

def set\_temp():

print("set temp on toaster")

class Juicer(kitchenAppliance):

def on():

print("Turn on juicer")

def off():

print("Turn off juicer")

Juicer.on()

**4. Interface Segregation Principle (ISP)**

from abc import ABC,abstractmethod

class Walker(ABC):

@abstractmethod

def walk() -> bool:

return print("Can Walk")

class Swimmer(ABC):

@abstractmethod

def swim() -> bool:

return print("Can Swim")

class Human(Walker, Swimmer):

def walk():

return print("Humans can walk")

def swim():

return print("Humans can swim")

class Whale(Swimmer):

def swim():

return print("Whales can swim")

if \_\_name\_\_ == "\_\_main\_\_":

Human.walk()

Human.swim()

Whale.swim()

Whale.walk()

**5.Dependency Inversion Principle**

class Employee(object):

def Work():

pass

class Manager():

def \_\_init\_\_(self):

self.employees=[]

def addEmployee(self,a):

self.employees.append(a)

class Developer(Employee):

def \_\_init\_\_(self):

print("developer added")

def Work():

print ("turning coffee into code")

class Designer(Employee):

def \_\_init\_\_(self):

print ("designer added")

def Work():

print ("turning lines to wireframes")

class Testers(Employee):

def \_\_init\_\_(self):

print("tester added")

def Work():

print ("testing everything out there")

if \_\_name\_\_ == "\_\_main\_\_":

a=Manager()

a.addEmployee(Developer())

a.addEmployee(Designer())