# OOPs in Python

## What is Object-Oriented Programming (OOPs)?

Object-Oriented Programming (OOPs) is a programming paradigm in Python where code is organized around **objects** and **classes**. These objects represent real-world entities by combining **data (attributes)** and **behavior (methods)** into one unit.

## Why is OOPs Used in Python?

- Helps model real-world systems easily
- Improves code **reusability** through inheritance
- Enhances **modularity**, **readability**, and **maintenance**
- Makes it easier to manage and extend large codebases
- Provides structure to the code by bundling data and operations together

---

## Evolution of Programming Paradigms

| Paradigm | Description | Examples |
|---|---|---|
| **Procedural Programming** | Code is written as sequences of instructions or procedures (functions). | C, Pascal |
| **Functional Programming** | Emphasizes use of pure functions, immutability, and avoids state changes. | Haskell, early Python |
| **Object-Oriented Programming** | Organizes code using objects and classes, modeling real-world systems. | Python, Java, C++ |

---

# Four Pillars of OOPs in Python

## 1. Encapsulation

Encapsulation is the concept of **wrapping data and methods** into a single unit (class), while restricting direct access to some of the object's components.

```python
class Person:
    def __init__(self, name):
        self.__name = name  # Private variable

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

---

## 2. Abstraction

Abstraction means **hiding complex implementation details** and exposing only the essential features of an object.

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def start_engine(self):
        pass
```

---

## 3. Inheritance

Inheritance allows a class to **inherit properties and methods** from another class. It promotes **code reuse**.

```
class Animal:
    def speak(self):
        print("Animal sound")

class Dog(Animal):
    def speak(self):
        print("Bark")
```

---

## 4. Polymorphism

Polymorphism allows **different classes to use the same interface** or method name in different ways.

```
def make_sound(animal):
    animal.speak()

make_sound(Dog())     # Output: Bark
make_sound(Animal())  # Output: Animal sound
```

In [49]:
```
# proceedural programming
# data store
name1='Vipul'
course1='Data Analytics'

name2='Ram'
course2='Data science'
```

In [50]:
```
# functional programming
# function-->
```

```
def info(**kwargs):
    return kwargs
```

In [51]:
```
info(name='Anshum',course='Data analytics')
```

Out[51]:
```
{'name': 'Anshum', 'course': 'Data analytics'}
```

In [52]:
```
#OPP
#class
#object
#feature-->attributes\
#function-->method
class Skillcircle:
    name='Vipul'
    course='Data Analyst'
    age=19
    def info(self):
        print('name\t',self.name)
        print('course\t',self.course)
        print('age\t',self.age)
```

In [53]:
```
#obj
stud1=Skillcircle()
stud2=Skillcircle()
```

In [54]:
```
stud1.name
```

Out[54]:
```
'Vipul'
```

In [55]:
```
stud1.course
```

Out[55]:
```
'Data Analyst'
```

In [56]:
```
stud1.age
```

Out[56]:
```
19
```

In [57]:
```
stud1.info()
```

```
name         Vipul
course         Data Analyst
age         19
```

In [58]:
```
stud2.name='Ram'
stud2.age=23
stud2.info()
```

```
name         Ram
course         Data Analyst
age         23
```

In [59]:
```
# railway --> classes
```

```
class Railway():
    fname='Vipul'
    lname='Pandey'
    dept='Delhi'
    to='Goa'
    def info(self):
        print('Name\t',self.fname+' '+self.lname)
        print('dept\t',self.dept)
        print('to\t',self.to)
```

In [60]:
```
obj1=Railway()
obj2=Railway()

obj1.info()
```

```
Name          Vipul Pandey
dept          Delhi
to          Goa
```

In [61]:
```
obj2.fname='Anshu'
obj2.lname='Kumar'
obj2.to='Dubai'
obj2.info()
```

```
Name          Anshu Kumar
dept          Delhi
to          Dubai
```

In [62]:
```
#class
class Employee:
    pass
```

In [63]:
```
emp1=Employee()
emp1.name = 'Gaurav'
emp1.desig='Data Analyst'
```

In [64]:
```
emp1.name
```

Out[64]:
```
'Gaurav'
```

In [65]:
```
# init method --> constructors
class Employee:
    amt=1.20
    def __init__(self,fname,lname,desig='',sal=0):
        self.fname=fname
        self.lname=lname
        self.desig=desig
        self.sal=sal
        self.email=self.fname.lower()+self.lname.lower()+'@zh.du.ac.in'
    def info(self):
        print('Name of Emp\t',self.fname+' '+self.lname)
        print('Designation \t',self.desig)
        print('Email \t\t',self.email)
        print('Salary \t\t',self.sal)
    def apply_raise(self):
        self.sal=self.sal*self.amt
        print('Salary after appraisal\t',self.sal)
```

In [66]:
```
emp1 = Employee('Vipul','Pandey','Data Analyst',100000)
emp2=Employee(lname='Kumar',fname='Anshu')
```

In [67]:
```
emp1.info()
print('-'*60)
emp2.info()
```

```
Name of Emp          Vipul Pandey
Designation           Data Analyst
Email                   vipulpandey@zh.du.ac.in
Salary                   100000
------------------------------------------------------------
Name of Emp          Anshu Kumar
Designation
Email                   anshukumar@zh.du.ac.in
Salary                   0
```

In [68]:
```
emp1.email
```

Out[68]:
```
'vipulpandey@zh.du.ac.in'
```

In [69]:
```
emp1.info()
```

```
Name of Emp          Vipul Pandey
Designation           Data Analyst
Email                   vipulpandey@zh.du.ac.in
Salary                   100000
```

In [70]:
```
emp1.apply_raise()
```

```
Salary after appraisal         120000.0
```

In [71]:
```
emp1.info()
```

```
Name of Emp          Vipul Pandey
Designation           Data Analyst
Email                   vipulpandey@zh.du.ac.in
Salary                   120000.0
```

In [72]:
```
from sklearn.linear_model import Lasso
```

In [73]:
```
lr1=Lasso()
lr1
```

Out[73]:
```
▾ Lasso
Lasso()
```

In [74]:
```
lr2=Lasso(alpha=50,max_iter=100,tol=0.1)
lr2
```

Out[74]:
```
▾ Lasso
Lasso(alpha=50, max_iter=100, tol=0.1)
```

In [ ]: