In [2]:
```
#1  five key concepts of Object-Oriented Programming (OOP)
```

```
Class
Object
Encapsulation
Inheritance
Polymorphism
```

In [3]:
```
#2. Write a Python class for a `Car` with attributes for `make`, `model`, and `year`. Include a method to display
```

In [4]:
```
class Car:
    Amount=100000
    def make(self,car_make):
        print('car is successfully made')
    def model(self,car_model):
        print('This car is model no 209')
    def year(self,car_year):
        print('this car is of year 2024')
```

In [5]:
```
c1=Car()
c1.Amount
```

Out[5]:
```
100000
```

In [6]:
```
#3. Explain the difference between instance methods and class methods. Provide an example of each.
```

```
Method Overloading:

Two or more methods have the same name but different numbers of parameters or different types of parameters,
or both. These methods are called overloaded methods and this is called method overloading.
```

In [7]:
```
def product(a, b):
    p = a * b
    print(p)

def product(a, b, c):
    p = a * b*c
    print(p)

product(4, 5, 5)
```

```
100
```

In [8]:
```
#4. What are the three types of access modifiers in Python? How are they denoted?
```

```
A Class in Python has three types of access modifiers:

Public Access Modifier: Theoretically, public methods and fields can be accessed directly by any class.
Protected Access Modifier: Theoretically, protected methods and fields can be accessed within the same class
it is declared and its subclass.
Private Access Modifier: Theoretically, private methods and fields can be only accessed within the same class
it is declared.
```

In [10]:
```
#example of public access modifier
class Customer:
    def __init__(self, name, email):
        self.name = name
        self.email = email

john = Customer("John Doe", "john@example.com")
print(john.name)
john.email = "johndoe@example.com"
print(john.email)
```

```
John Doe
johndoe@example.com
```

In [11]:

```
#example of protected access modifier
class BankAccount:
    def __init__(self, account_number, balance):
        self._account_number = account_number
        self._balance = balance

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        if amount <= self._balance:
            self._balance -= amount
        else:
            print("Insufficient funds!")

account = BankAccount("1234567890", 1000)
print(account._balance)
account.deposit(500)
account.withdraw(200)
print(account._balance)
```

```
1000
1300
```

In [13]:

```
#private access modifier
class User:
    def __init__(self, username, password):
        self.__username = username
        self.__password = password

    def login(self, username, password):
        if self.__username == username and self.__password == password:
            print("Login successful!")
        else:
            print("Invalid credentials!")

jenny = User("jenny123", "password123")
jenny.login("jenny123", "password123")
```

```
Login successful!
```

In [14]:

```
#5. Explain the difference between instance methods and class methods. Provide an example of each.
```

Class methods- are associated with the class rather than instances. They are defined using the @classmethod decorator and take the class itself as the first parameter, usually named cls. Class methods are useful for tasks that involve the class rather than the instance, such as creating class-specific behaviors or modifying class-level attributes.

Instance methods- are the most common type of methods in Python classes. They are associated with instances of a class and operate on the instance's data. When defining an instance method, the method's first parameter is typically named self, which refers to the instance calling the method. This allows the method to access and manipulate the instance's attributes.

In [16]:

```
#example class method
class MyClass:
    class_variable = 0

    def __init__(self, value):
        self.instance_variable = value

    @classmethod
    def class_method(cls, x):
        cls.class_variable += x
        return cls.class_variable

# Creating instances of the class
obj1 = MyClass(5)
obj2 = MyClass(10)
print(MyClass.class_method(3))
print(MyClass.class_method(7))
```

```
3
10
```

In [17]:
```
#example instance method
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduce(self):
        return f"Hi, I'm {self.name} and I'm {self.age} years old."
person1 = Person("Kishan", 20)

print(person1.introduce())
```

```
Hi, I'm Kishan and I'm 20 years old.
```

In [18]:
```
#6 6. Describe the five types of inheritance in Python. Provide a simple example of multiple inheritance.
```

```
There are four types of inheritance in Python:
Single Inheritance
Multiple Inheritance
Multilevel Inheritance
Hierarchical Inheritance
```

In [20]:
```
#Example of single inheritance
# Python program to demonstrate
# single inheritance

# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")

# Derived class


class Child(Parent):
    def func2(self):
        print("This function is in child class.")


# Driver's code
object = Child()
object.func1()
object.func2()
```

```
This function is in parent class.
This function is in child class.
```

In [21]:

```
#example of multiple inheritance
# Python program to demonstrate
# multiple inheritance

# Base class1
class Mother:
    mothername = ""

    def mother(self):
        print(self.mothername)

# Base class2


class Father:
    fathername = ""

    def father(self):
        print(self.fathername)

# Derived class


class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)


# Driver's code
s1 = Son()
s1.fathername = "RAM"
s1.mothername = "SITA"
s1.parents()
```

```
Father : RAM
Mother : SITA
```

In [23]:

```
#example of multilevel inheritance
# Python program to demonstrate
# multilevel inheritance

# Base class


class Grandfather:

    def __init__(self, grandfathername):
        self.grandfathername = grandfathername

# Intermediate class


class Father(Grandfather):
    def __init__(self, fathername, grandfathername):
        self.fathername = fathername

        # invoking constructor of Grandfather class
        Grandfather.__init__(self, grandfathername)

# Derived class


class Son(Father):
    def __init__(self, sonname, fathername, grandfathername):
        self.sonname = sonname

        # invoking constructor of Father class
        Father.__init__(self, fathername, grandfathername)

    def print_name(self):
        print('Grandfather name :', self.grandfathername)
        print("Father name :", self.fathername)
        print("Son name :", self.sonname)


# Driver code
s1 = Son('Prince', 'Rampal', 'Lal mani')
print(s1.grandfathername)
s1.print_name()
```

```
Lal mani
Grandfather name : Lal mani
Father name : Rampal
Son name : Prince
```

In [24]:

```
#example of Hierachial
# Python program to demonstrate
# Hierarchical inheritance


# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")

# Derived class1


class Child1(Parent):
    def func2(self):
        print("This function is in child 1.")

# Derivied class2


class Child2(Parent):
    def func3(self):
        print("This function is in child 2.")


# Driver's code
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

```
This function is in parent class.
This function is in child 1.
This function is in parent class.
This function is in child 2.
```

In [25]:

```
#7. What is the Method Resolution Order (MRO) in Python? How can you retrieve it programmatically?
```

```
Method Resolution Order(MRO) it denotes the way a programming language resolves a method or attribute. Python
supports classes inheriting from other classes. The class being inherited is called the Parent or Superclass,
while the class that inherits is called the Child or Subclass.
```

In [28]:

```
#example
# Python program showing
# how MRO works
class A:
    def rk(self):
        print(&quot; In class A&quot;)
class B(A):
    def rk(self):
        print(&quot; In class B&quot;)
class C(A):
    def rk(self):
        print(&quot;In class C&quot;)

# classes ordering
class D(B, C):
    pass

r = D()
r.rk()
```

```
  File "<ipython-input-28-fa8ff323d3ef>", line 6
    print(&quot; In class A&quot;)
          ^
SyntaxError: invalid syntax
```

In [29]:

```
#8 Demonstrate polymorphism by creating a function that can work with different shape objects to calculate and pr
```

The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function.

In [30]:

```
#example of polymorphism
print(len("geeks"))

# len() being used for a list
print(len([10, 20, 30]))
```

```
5
3
```

In [31]:

```
#9 Create a decorator that measures and prints the execution time of a function.
```

Everything in Python is an object. Functions in Python also object. Hence, like any other object they can be referenced by variables, stored in data structures like dictionary or list, passed as an argument to another function, and returned as a value from another function. In this article, we are going to see the timing function with decorators.

Decorator: A decorator is used to supercharge or modify a function. A decorator is a higher-order function that wraps another function and enhances it or changes it.

In [33]:

```
#example
def my_decorator(func):
    def wrapper_function(*args, **kwargs):
        print("*"*10)
        func(*args, **kwargs)
        print("*"*10)
    return wrapper_function


def say_hello():
    print("Hello Sanjay!")

@my_decorator
def say_bye():
    print("Bye Sanjay!")


say_hello = my_decorator(say_hello)
say_hello()
say_bye()
```

```
**********
Hello Sanjay!
**********
**********
Bye Sanjay!
**********
```

In [34]:

```
#10 Explain the concept of the Diamond Problem in multiple inheritance. How does Python resolve it?
```

The diamond problem occurs when two classes have a common parent class, and another class has both those classes as base classes.this kind of implementation of inheritance, if class B and class C override (any) same method of class A and instance of class D calls that method, it becomes ambiguous to programming languages, whether we want to call the overridden method from class B or the one from class C.

In [35]:

```python
#example
class A:
    def display(self):
        print("This is class A")

class B(A):
    def display(self):
        print("This is class B")

class C(A):
    def display(self):
        print("This is class C")

class D(B, C):
    pass

obj = D()
obj.display()
```

```
This is class B
```

In [ ]: