What Is Inheritance?

Inheritance provides a way to create a new class from an existing class. The new class is a specialized version of the existing class such that it inherits all the non-private fields (variables) and methods of the existing class. The existing class is used as a starting point or as a base to create the new class.

The IS A Relationship

After reading the above definition, the next question that comes to mind is this: when do we use inheritance? Wherever we come across an IS A relationship between objects, we can use inheritance

So, from the above descriptions of inheritance, we can conclude that we can build new classes by extending existing classes.

Existing Class	Derived Class
Shape	Square
Programming Language	Python
Vehicle	Car

The Python Object class

The primary purpose of object-oriented programming is to enable a programmer to model the real-world objects using a programming language.

In Python, whenever we create a class, it is, by default, a subclass of the built-in Python object class. This makes it an excellent example of inheritance in Python. This class has very few properties and methods, but it does provide a strong basis for object-oriented programming in Python.

The Syntax and Terminologies

In inheritance, in order to create a new class based on an existing class, we use the following terminology:

- Parent Class (Super Class or Base Class): This class allows the reuse of its public properties in another class.
- Child Class (Sub Class or Derived Class): This class inherits or extends the superclass.

A child class has all public attributes of the parent class.

Syntax

The name of the parent class is written in brackets after the name of the child class , which is followed by the body of the child class.

Example:

```
class Car:

    def __init__(self, company, engine, wheels):
        self.company = company
        self.engine = engine
        self.wheels = wheels

    def showDetails(self):
        print("Company:", self.company)
        print("Engine:", self.engine)
        print("Wheels", self.wheels)

class FlyingCar(Car):

    def __init__(self, company, engine, wheels, wings):
        self.wings = wings
        Car.__init__(self, company, engine, wheels)

    def showDetails(self):
        print("Company:", self.company)
        print("Engine:", self.engine)
        print("Wheels", self.wheels)
        print("Wings", self.wings)

flyObj = FlyingCar("Tata", "Petrol", "Alloy", "2")
flyObj.showDetails()
```

```
shtlp_0146@SHTLP0146:~/Desktop/Phthon$ /bin/python3 /home/
shtlp_0146/Desktop/Phthon/demo.py
Company: Tata
Engine: Petrol
Wheels Alloy
Wings 2
```

The Super Function

The use of super() comes into play when we implement inheritance. It is used in a child class to refer to the parent class without explicitly naming it. It makes the code more manageable, and there is no need to know the name of the parent class to access its attributes.

Use cases of the super() function

The super function is used in three relevant contexts:

- **Accessing parent class properties**
- **Calling the parent class methods**
- **Using with initializers**

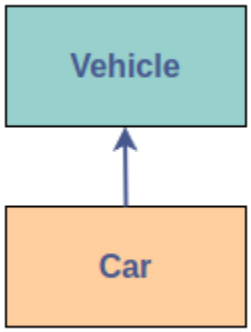
Types of Inheritance

Based upon parent classes and child classes, there exists the following five types of inheritance:

- Single
- Multi-level
- Hierarchical
- Multiple
- Hybrid
-

Single inheritance

In single inheritance, there is only a single class extending from another class. We can take the example of the Vehicle class as the parent class, and the Car class as the child class. Let's implement these classes below:



```
    def setTopSpeed(self, speed):  # defining the set
        self.topSpeed = speed
        print("Top speed is set to", self.topSpeed)


class Car(Vehicle):  # child class
    def openTrunk(self):
        print("Trunk is now open.")


corolla = Car()  # creating an object of the Car class
corolla.setTopSpeed(220)  # accessing methods from the parent clas
corolla.openTrunk()  # accessing method from its own class

```

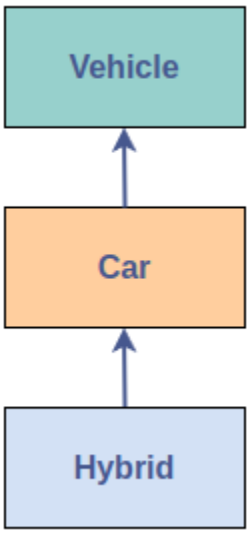
Run Save Reset

Output 0.5s

```
Top speed is set to 220
Trunk is now open.
```

Multi-level inheritance

When a class is derived from a class which itself is derived from another class, it is called multilevel inheritance. We can extend the classes to as many levels as we want to.



Example:

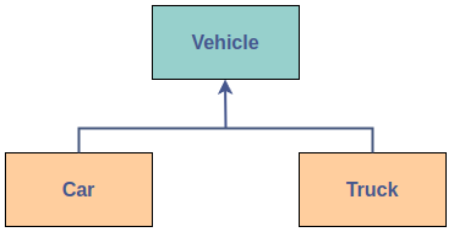
```
class Car:
    def __init__(self, company, engine, wheels):
        self.company = company
        self.engine = engine
        self.wheels = wheels

    def showDetails(self):
        print("Company:", self.company)
        print("Engine:", self.engine)
        print("Wheels", self.wheels)

class FlyingCar(Car):
    def __init__(self, company, engine, wheels, wings):
        self.wings = wings
       # Car.__init__(self, company, engine, wheels)
        super().__init__(company, engine, wheels)

    def showDetails(self):
        super().showDetails()
        print("Wings", self.wings)

class FlyingHydroCar(FlyingCar):
    def __init__(self, company, engine, wheels, wings, pedal):
        super().__init__(company, engine, wheels, wings)
        self.pedal = 4
    def showDetails(self):
        print("Pedals:",self.pedal)
        super().showDetails()


fhc = FlyingHydroCar("Tata", "Petrol", "Alloy", 2, 4)
```

Hierarchical inheritance

In hierarchical inheritance, more than one class extends, as per the requirement of the design, from the same base class. The common attributes of these child classes are implemented inside the base class.

Example:

- A Car IS A Vehicle
- A Truck IS A Vehicle



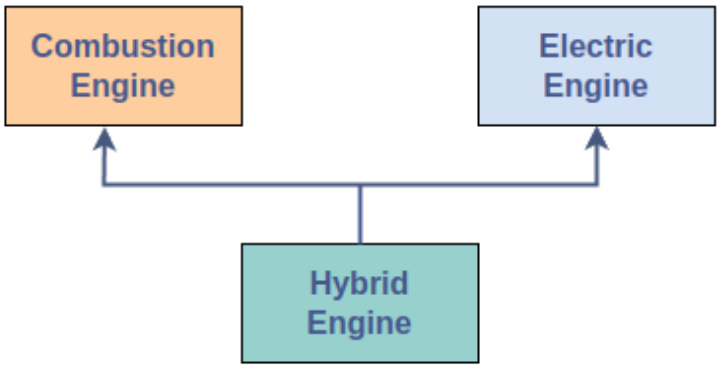
```python
class Vehicle:  # parent class
    def setTopSpeed(self, speed):  # defining the set
        self.topSpeed = speed
        print("Top speed is set to", self.topSpeed)


class Car(Vehicle):  # child class of Vehicle
    pass


class Truck(Vehicle):  # child class of Vehicle
    pass


corolla = Car()  # creating an object of the Car class
corolla.setTopSpeed(220)  # accessing methods from the parent clas

volvo = Truck()  # creating an object of the Truck class
volvo.setTopSpeed(180)  # accessing methods from the parent class

```

Multiple inheritance

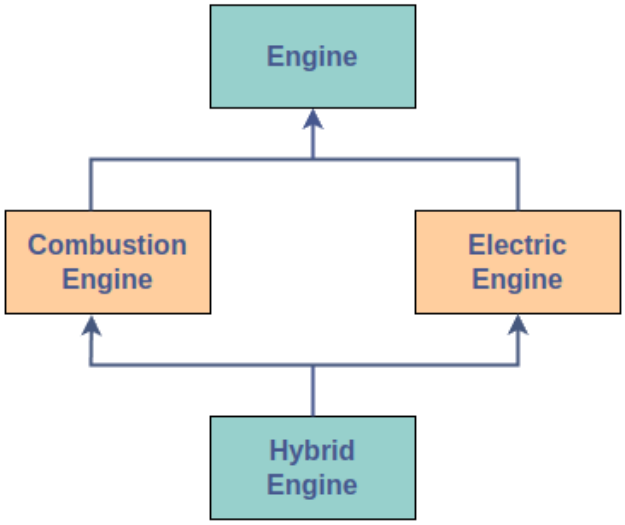
When a class is derived from more than one base class, i.e., when a class has more than one immediate parent class, it is called multiple inheritance.

Example:

- HybridEngine IS A ElectricEngine.
- HybridEngine IS A CombustionEngine as well.



Hybrid inheritance



A type of inheritance which is a combination of Multiple and Multi-level inheritance is called hybrid inheritance.

- CombustionEngine IS A Engine.
- ElectricEngine IS A Engine.
- HybridEngine IS A ElectricEngine and a CombustionEngine.

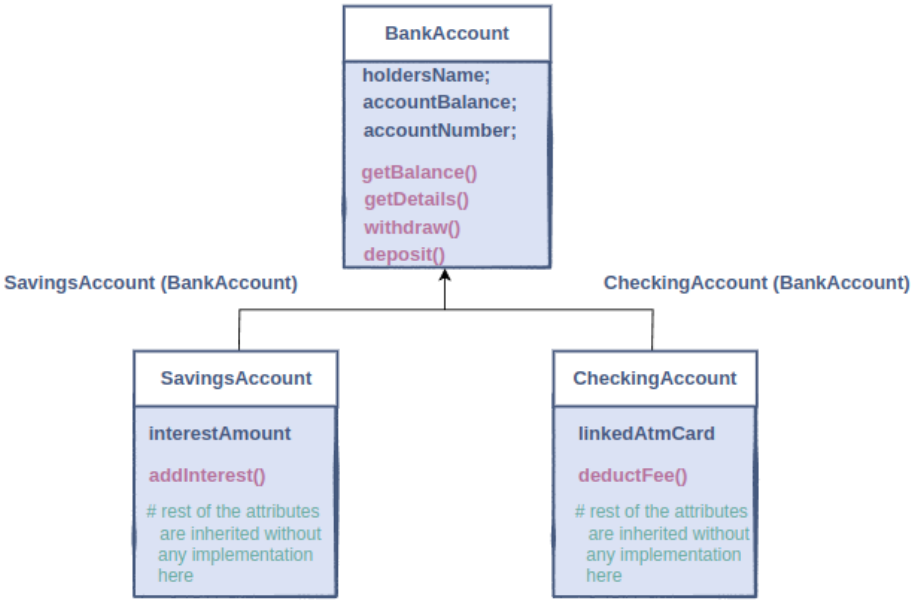
```
class Engine:  # Parent class
    def setPower(self, power):
        self.power = power


class CombustionEngine(Engine):  # Child class inherited from Engi
    def setTankCapacity(self, tankCapacity):
        self.tankCapacity = tankCapacity


class ElectricEngine(Engine):  # Child class inherited from Engine
    def setChargeCapacity(self, chargeCapacity):
        self.chargeCapacity = chargeCapacity

# Child class inherited from CombustionEngine and ElectricEngine


class HybridEngine(CombustionEngine, ElectricEngine):
    def printDetails(self):
        print("Power:", self.power)
        print("Tank Capacity:", self.tankCapacity)
        print("Charge Capacity:", self.chargeCapacity)


car = HybridEngine()
car.setPower("2000 CC")
car.setChargeCapacity("250 W")
car.setTankCapacity("20 Litres")
car.printDetails()

```

Advantages of Inheritance

Reusability

Inheritance makes the code reusable. Consider that you are up for designing a banking system using classes. Your model might have these:

- A parent class: BankAccount
- A child class: SavingsAccount
- Another child class: CheckingAccount



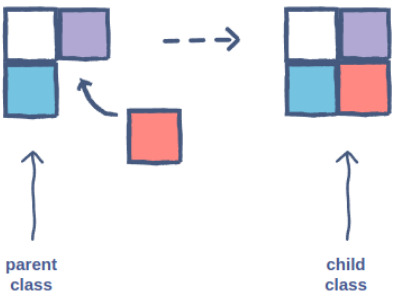
In the above example, you don’t need to duplicate the code for the deposit() and withdraw() methods inside the child classes, namely SavingsAccount and CheckingAccount. In this way, you can avoid the duplication of code.

Code modification

Suppose you put the same code in different classes, but what happens when you have to make changes to a function and in several places? There is a high likelihood that you will forget some places and bugs will be introduced. You can avoid this with inheritance, which will ensure that all changes are localized, and inconsistencies are avoided.

Extensibility

Using inheritance, one can extend the base class as per the requirements of the derived class. It provides an easy way to upgrade or enhance specific parts of a product without changing the core attributes. An existing class can act as a base class from which a new class with upgraded features can be derived.

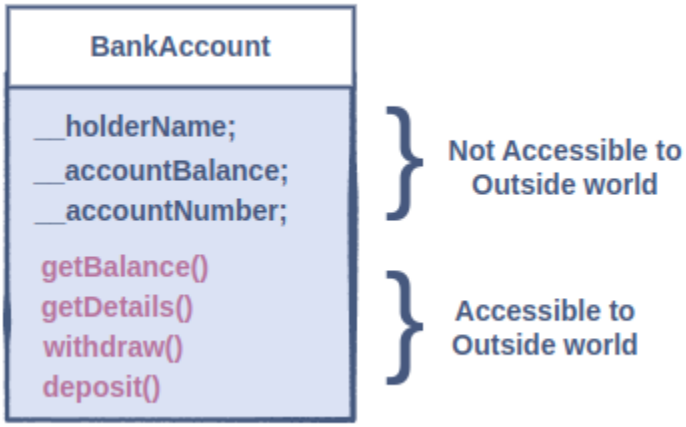


Child class can add on to the Parent class

In the example above, you realize at a later point that you have to diversify this banking application by adding another class for MoneyMarketAccount. So, rather than implementing this class from scratch, you can extend it from the existing BankAccount class as a starting point. You can also reuse its attributes that are common with MoneyMarketAccount.

Data hiding

The base class can keep some data private so that the derived class cannot alter it. This concept is called encapsulation.



Challenge 1: Implement a Banking Account

Problem statement

Implement the basic structure of a parent class, `Account`, and a child class, `SavingsAccount`.

Task 1

Implement properties as **instance variables**, and set them to `None` or 0.

`Account` has the following properties:

- `title`
- `balance`

`SavingsAccount` has the following properties:

- `interestRate`

Task 2

Create an **initializer** for `Account` class. The order of parameters should be the following, where Mark is the title, and 5000 is the account balance:

```
Account("Mark", 5000)
```

Task 3

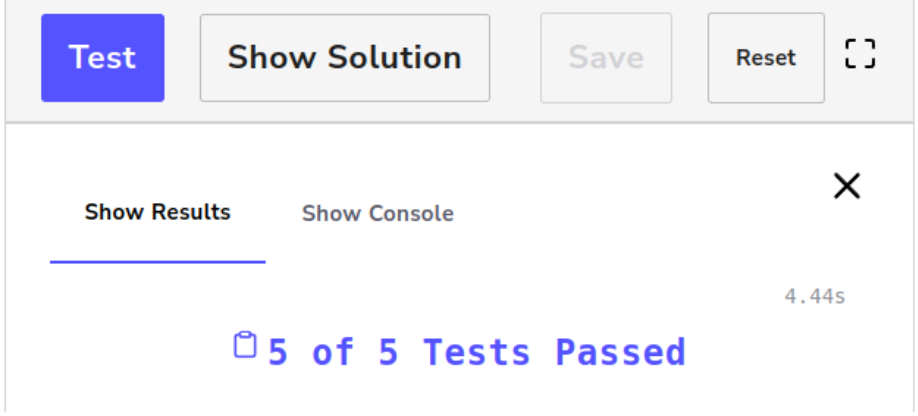
Implement properties as **instance variables**, and set them to `None` or 0.

Create an **initializer** for the `SavingsAccount` class using the initializer of the `Account` class in the order below:

```
Account("Mark", 5000, 5)
```

Here, `Mark` is the `title` and `5000` is the `balance` and `5` is the `interestRate`.

```python
class Account:
    def __init__(self, title=None, balance=0):
        self.title = title
        self.balance = balance


class SavingsAccount(Account):
    def __init__(self, title=None, balance=0, interestRate=0):
        self.interestRate = interestRate
        super().__init__(title, balance)

```



Challenge 2: Handling a Bank Account

Task 3

In the `Account` class, implement the `withdrawal(amount)` *method* that subtracts the `amount` from the `balance`. It **does not** return anything.

Sample input

```
balance = 2000
withdrawal(500)
getbalance()
```

Sample output

```
1500
```

Task 4

In the `SavingsAccount` class, implement an `interestAmount()` *method* that **returns** the interest amount of the **current** balance. Below is the formula for calculating the interest amount:

$$\text{interest amount} = \frac{\text{interest rate x balance}}{100}$$

```python
class Account:
    def __init__(self, title=None, balance=0):
        self.title = title
        self.balance = balance

    def withdrawal(self, amount):
        self.balance -= amount

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

    def getBalance(self):
        return self.balance


class SavingsAccount(Account):
    def __init__(self, title=None, balance=0, interestRate=0):
        super().__init__(title, balance)
        self.interestRate = interestRate

    def interestAmount(self):
        return (self.interestRate*self.balance)/100



# code to test - do not edit this
demo1 = SavingsAccount("Mark", 2000, 5)  # initializing a SavingsA

```