Python Inheritance

When you want to extend the functionality of an existing class, you can just modify that class. But there's a good chance you'll make it more complicated or break something that used to work.

Of course, you can write a new class. But that means you have more code to maintain, and the parts of the old class that used to work might drift apart.

The solution is Inheritance.

What Is Inheritance?

Inheritance is the process of creating a new class from an existing one.

A class created through inheritance can use all the code (e.g. attributes and methods) from the old class.

So, you edit only what you need to modify in the new class, and this overrides the behavior of the old class.

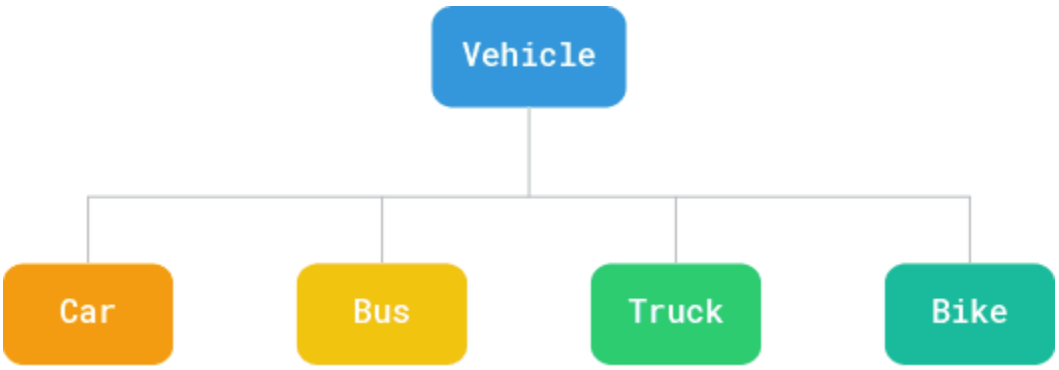
When one class inherits from another, the inheriting class is known as a child, subclass, or derived class, and the class it inherits from is known as its parent, superclass, or base class.

Analogy

There exists a hierarchy relationship between classes. It's similar to relationships that you know from real life.

Think about vehicles, for example. Bikes, cars, buses and trucks, all share the characteristics of vehicles (speed, color, gear). Yet each has additional features that make them different.

Keeping that in mind, you could implement a Vehicle (as a base) class in Python. Whereas, Cars, Buses and Trucks and Bikes can be implemented as subclasses which will inherit from vehicle.



Defining a Base Class

Any class that does not inherit from another class is known as a base class.

The example below defines a base class called Vehicle. It has a method called description that prints the description of the vehicle.

```
# base class
class Vehicle():
    def description(self):
        print("I'm a Vehicle!")
```

Subclassing

The act of basing a new class on an existing class is known as Subclassing. It allows you to add new functionality or override existing functionality.

Now let's create a subclass of Vehicle called Car. You can create a subclass by using the same `class` keyword but with the base class name inside the parentheses.

```
# base class
class Vehicle():
    def description(self):
        print("I'm a Vehicle!")


# subclass
```

```
class Car(Vehicle):
    pass
```

Now let's make one object from each class and call the description method:

```
# base class
class Vehicle():
    def description(self):
        print("I'm a Vehicle!")


# subclass
class Car(Vehicle):
    pass


# create an object from each class
v = Vehicle()
c = Car()


v.description()
# Prints I'm a Vehicle!
c.description()
# Prints I'm a Vehicle!
```

Notice that without doing anything special, the new Car class automatically gained all of the characteristics of Vehicle, such as its `description()` method.

Override a Method

As you just saw, a Car class inherited everything from its base class Vehicle.

However, a subclass should be different from its base class in some way; otherwise, there's no point in defining a new class.

Let's redefine the `description()` method inside of the Car:

```
# base class
```

```
class Vehicle():
    def description(self):
        print("I'm a Vehicle!")

# subclass
class Car(Vehicle):
    def description(self):
        print("I'm a Car!")

# create an object from each class
v = Vehicle()
c = Car()

v.description()
# Prints I'm a Vehicle!
c.description()
# Prints I'm a Car!
```

Here, when you call `description()` method, the Car subclass version of the method is called.

This is because when you add a method in the subclass with the same name as the base class, it gets overridden.

When a subclass provides its own custom implementation of a method that is already provided by its base class, it is known as overriding.

You can not only override an instance method, but any method, including `__init__()`.

Add a Method

The subclass can also add a method that was not present in its base class.

Let's define the new method `setSpeed()` for Car class.

```
# a parent class
class Vehicle():
    def description(self):
        print("I'm a", self.color, "Vehicle")


# subclass
class Car(Vehicle):
    def description(self):
        print("I'm a", self.color, self.style)
    def setSpeed(self, speed):
        print("Now traveling at", speed,"miles per hour")


# create an object from each class
v = Vehicle()
c = Car()


c.setSpeed(25)
# Prints Now traveling at 25 miles per hour
v.setSpeed(25)
# Triggers AttributeError: 'Vehicle' object has no attribute 'setSpeed'
```

A Car object can react to a `setSpeed()` method call, but a Vehicle object cannot.

The super() Function

When you override a method, you sometimes want to reuse the method of the base class and add some new stuff.

You can achieve this by using the super() Function.

To demonstrate this, let's redefine our Vehicle and Car class, but this time with the `__init__()` method.

Notice that the `__init__()` call in the Vehicle class has only 'color' parameter while the Car class has an additional 'style' parameter.

```
# base class
class Vehicle():
    def __init__(self, color):
        self.color = color
    def description(self):
        print("I'm a", self.color, "Vehicle")

# subclass
class Car(Vehicle):
    def __init__(self, color, style):
        super().__init__(color)    # invoke Vehicle's __init__() method
        self.style = style
    def description(self):
        print("I'm a", self.color, self.style)

# create an object from each class
v = Vehicle('Red')
c = Car('Black', 'SUV')

v.description()
# Prints I'm a Red Vehicle
c.description()
# Prints I'm a Black SUV
```

Here, the super() invokes the superclass's `__init__()` method. That's why you are able to access both the color and style attributes.

We could have defined our new class as follows:

```
class Car(Vehicle):
    def __init__(self, color, style):
```

```
        self.color = color
        self.style = style


    def description(self):
        print("I'm a", self.color, self.style)
```

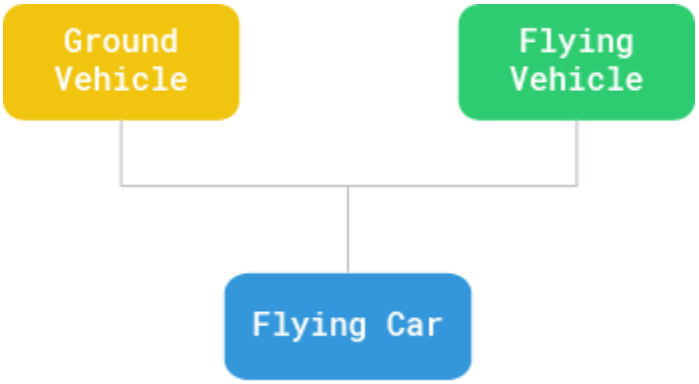
But it would have defeated the purpose of code-reuse. We used `super()` to make sure the car can do everything a vehicle can, plus more.

There's another benefit; if the definition of Vehicle changes in the future, the `super()` will ensure that all the changes will be reflected.

Multiple Inheritance

Python also supports multiple inheritance, where a subclass can inherit from multiple superclasses.

In multiple inheritance, the characteristics of all the superclasses are inherited into the subclass.



Let's define a FlyingCar subclass that inherits from a GroundVehicle class and a FlyingVehicle class.

```
# base class 1
class GroundVehicle():
    def drive(self):
        print("Drive me on the road!")
```

```
# base class 2
class FlyingVehicle():
    def fly(self):
        print("Fly me to the sky!")


# subclass
class FlyingCar(GroundVehicle, FlyingVehicle):
    pass


# create an object of a subclass
fc = FlyingCar()


fc.drive()
# Prints Drive me on the road!
fc.fly()
# Prints Fly me to the sky!
```

Notice that methods from both superclasses were effectively used in the subclass.