Hyperiondev

TASK

OOP – Inheritance

Visit our website

Introduction

WELCOME TO THE OOP - INHERITANCE TASK!

Inheritance is one of the core pillars of object-oriented programming (OOP). In this task, you will learn how inheritance allows us to reuse code from existing classes in new classes and include additional logic or variations on the logic.

WHAT IS INHERITANCE?

Let's begin by examining inheritance from the natural world. A look at a child in comparison to a parent quickly reveals that the child inherits certain traits from the parents. The attributes that they may inherit include eye colour, nose size, or height. They could also inherit certain abilities such as athletic abilities.

Suppose you had written a class with some attributes and methods, and you wanted to define another class with most of the same attributes and methods plus some additional attributes and methods. Inheritance is the mechanism by which you can produce a second class from the first without redefining the second class completely from scratch or altering the definition of the first class itself.

Alternatively, suppose you wish to write two related classes that share a significant subset of their respective attributes and methods. Rather than writing two completely separate classes from scratch, you could encapsulate the shared information in a single class and write two more classes that inherit all of that information from the first class, saving space and improving code clarity in the long run. A class that is inherited from is called the **base class** (also referred to as the **parent class**), and the class that inherits from the base class is called the **derived class** (also referred to as the **subclass**).

In most major object-oriented languages, objects of the derived class are also objects of the base class, but not vice-versa. This means that functions that act on base objects may also act on derived objects; this is often useful when writing an object-oriented program.

PYTHON INHERITANCE IN ACTION

To demonstrate how inheritance works, first, we need to create a parent class. We will use a “car” class as an example:

```
# Parent class for a car which we can extend to a subclass
class Car:
    # Class variable for whether the engine is running or not
    is_running = False

    # Constructor that allows us to set the make and model
    # as instance variables
    def __init__(self, make, model):
        self.make = make
        self.model = model

    # Method to start the engine
    def start_car(self):
        self.is_running = True

    # Method to turn off the engine
    def turn_off_car(self):
        self.is_running = False

    # Method to print the make and model
    def show_make_and_model(self):
        print(f"This vehicle is a {self.make} {self.model}")
```

The parent class above has attributes for the make and model of the car and some actions or methods that can turn the car on and off. We can extend this class to a pickup truck class (the subclass) which will inherit the attributes and methods from the car class, but we can add additional attributes and methods.

```
# We are inheriting all of the attributes and methods
# from the Car class by passing it as an argument to
# the PickupTruck class
class PickupTruck(Car):
    # This is an additional class variable that is specific
    # to the PickupTruck class
    is_loaded = False
```

```
# Method to load the truck
    def load(self):
        self.is_loaded = True

    # Method to remove the load from the truck
    def unload(self):
        self.is_loaded = False
```

We can then create an instance of the **`PickupTruck`** class and demonstrate how the attributes and methods are inherited from the **`Car`** class as well as new attributes and methods that we created in the subclass:

```
# Create a pickup truck object
pickup_truck_1 = PickupTruck("Toyota", "Hilux")

# Call the load method that we created in the subclass
# This changes the variable from False to True
pickup_truck_1.load()

# Call the start_car method inherited from the parent class
pickup_truck_1.start_car()

# Print out to values so that we can see that both of
# the above methods worked
print(pickup_truck_1.is_running)
print(pickup_truck_1.is_loaded)

# Call another method that was inherited from the parent class
pickup_truck_1.show_make_and_model()
```

The above code will generate the following output which demonstrates that all the attributes and methods are working in the subclass:

```
True
True
This vehicle is a Toyota Hilux
```

THE "SUPER" FUNCTION AND ITS USE

Python's `super()` function gives you access to methods in a parent class from the subclass that inherits from it. The `super()` function returns a temporary object of the parent class that allows you to call that parent class's methods. The goal of the `super()` function is to provide a much more abstract and portable solution for initialising classes.

When referring to the parent class from the subclass, we don't need to write the name of the parent class explicitly. Calling the methods created in the parent class with `super()` saves you from needing to rewrite those methods in your subclass and enables you to swap out parent classes with minimal code changes.

Let's look at an example of the `super()` function in Python.

```
# Define parent class
class Computer():
    def __init__(self, computer, ram, ssd):
        self.computer = computer
        self.ram = ram
        self.ssd = ssd

# Define subclass
class Laptop(Computer):
    def __init__(self, computer, ram, ssd, model):
        super().__init__(computer, ram, ssd)
        self.model = model

# Create a laptop object
vivobook = Laptop('Asus', 8, 512, 'Vivobook')

# Print laptop's features
print('Computer make:', vivobook.computer)
print('Computer model:', vivobook.model)
print(f"This computer has {vivobook.ram} GB of RAM.")
print(f"This computer has {vivobook.ssd} GB of SSD storage.")
```

In the above example, the `Laptop` class inherits from the `Computer` class. Using the `super()` function, we are able to use the constructor function of the parent class in the constructor function of the subclass so that the three attributes from the parent class (computer, ram, and ssd) can be initialised together with the additional attribute of the subclass (model). Now, if we create a subclass object, we have access to the parent class's attributes because of the `super()` function.

METHOD OVERRIDING

Overriding is the ability of a class to change the implementation of a method provided by a parent class.

Overriding is a very important part of OOP. By using method overriding a class can make a copy of another class, but at the same time enhance or customise part of its behaviour. Method overriding is thus a part of the inheritance mechanism.

In Python, method overriding occurs by defining a method in the subclass with the same name as a method in the parent class. When you define a method in the subclass that has the same name as the method in the parent class, an instance of the subclass will execute the logic in the subclass when that method is called. If this method is not defined in the subclass, then the method in the parent class is executed.

Here is an example of a method being inherited from the parent class without method overriding. When this method is executed from an instance of the subclass, the method in the parent class is executed:

```
# Parent class
class Father():
    def transport(self):
        print("The transport used is a car")
# Subclass
class Son(Father):
    pass

son_1 = Son()
# This will output "The transport used is a car"
# because it is using the inherited method from the Father class
son_1.transport()
```

Now let's override the `transport()` method in the subclass so that the transport used by the `Son` subclass will be a bicycle.

```
# Parent class
class Father():
    def transport(self):
        print("The transport used is a car")
```

```python
# Subclass
class Son(Father):
    def transport(self):
        print("The transport used is a bicycle")

son_1 = Son()
# This will output "The transport used is a bicycle"
# because the inherited method is being overridden
# by the Son subclass
son_1.transport()
```

Take note:

This task covers single inheritance. Multiple inheritance is also possible in Python. We encourage you to do some research on multiple inheritance.

Practical Task 1

In this task, you will demonstrate your understanding of inheritance. Make a copy of the **task1_instructions.py** file and name it **practical_task_1.py**. Then, follow the instructions below.

- Add another method in the `Course` class that prints the head office location: *Cape Town*.
- Create a subclass of the `Course` class named `OOPCourse.`
- Create a constructor that initialises the following attributes with default values:
    - `description = “OOP Fundamentals”`
    - `trainer = "Mr Anon A. Mouse"`
- Create a method in the `OOPCourse` subclass named `trainer_details` that prints what the course is about and the name of the trainer by using the `description` and `trainer` attributes.
- Create a method in the `OOPCourse` subclass named `show_course_id` that prints the ID number of the course: *#12345*

- Create an object of the `OOPCourse` subclass called `course_1` and call the following methods
    - `contact_details`
    - `trainer_details`
    - `show_course_id`
- These methods should all print out the correct information to the terminal.

Practical Task 2

Create a file named **method_override.py** and follow the instructions below:

- Take user inputs that ask for the name, age, hair colour, and eye colour of a person.
- Create an `Adult` class with the following attributes and method:
    - Attributes: `name`, `age`, `eye_colour`, and `hair_colour`
    - A method called `can_drive()` which prints the name of the person and that they are old enough to drive.
- Create a subclass of the `Adult` class named `Child` that has the same attributes, but overrides the `can_drive()` method to print the person's name and that they are too young to drive.
- Create some logic that determines if the person is 18 or older and create an instance of the `Adult` class if this is true. Otherwise, create an instance of the `Child` class. Once the object has been created, call the `can_drive()` method to print out whether the person is old enough to drive or not.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.