Hyperiondev

TASK

Introduction to Object-Oriented Programming I: Classes

Visit our website

Introduction

WELCOME TO THE INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING TASK!

Object-Oriented Programming (OOP) is a fundamental style of programming for developing larger pieces of software. Up until now, the programs you have written are simple enough to be run from just one file. In the real world of software development, multiple programmers work on large projects that may have hundreds of different files of code that implement the functionality of the project.

Your first step to building more complex programs is understanding OOP. Practice will show you that once you get past the terminology, OOP is very simple.

WHY OOP?

Imagine we want to build a program for a university. This program has a database of students, their information, and their marks. We need to perform computations of this data, such as finding the average grade of a particular student. Here are some observations from the above problem:

- A university will have many students that have the same information stored in the database, for example, age, name, and gender. How can we represent this information in code?
- We need to write code to find the average of a student by simply summing their grades for different subjects, and dividing by the number of subjects taken. How can we only define this code once and reuse it for many students?

OOP is the solution to the above problems, and indeed many real-world implementations of the above systems will use OOP.

In fact, up until this point, you have been using classes and objects even though you may not have been aware of them. In Python, everything is an instance (object) of a class i.e. strings, lists, dictionaries, etc. Everything has a blueprint that is based on what we call a **class**.

Please run the following code in VS Code to see what the output is that is generated:

```
example_list = ["Dave", "Rob", "Stephen"]
example_boolean = True
example_string = "hello world"

print(type(example_list))
print(type(example_boolean))
print(type(example_string))

# even a function is an instance of the function class!

def this_is_a_function(a, b):
    return a * b

print(type(this_is_a_function))
```

The output generated looks as follows:

```
<class 'list'>
<class 'bool'>
<class 'str'>
<class 'function'>
```

We can therefore see that everything in Python is an object built from some particular class.

THE COMPONENTS OF OOP

The Class

The concept of a class may be hard to get your head around at first. A class is a specific Python file that can be thought of as a 'blueprint' for a specific data type.

You can think of a Class as defining your own special data types, with properties you determine.

A class stores properties along with associated functions called ***methods*** which run programming logic to modify or return the class properties.

The String class, for example, has the property which is the value of the string and then there are also methods in the class that can be used such as lower(), upper(), split() etc.

In the example that we discussed earlier (building a program for a university around a database of students), we would use a class called Student to represent a student. This is perfect because we know that the properties of a Student match those stored in the database such as name, age, etc.

Defining a Class in Python

Let us assume that the database stores the age, name, and gender of each student. The code to create a blueprint for a Student, or class, is as follows:

```
class Student():

      def __init__(self, age, name, gender):
            self.age = age
            self.name = name
            self.gender = gender
```

This may look confusing, so let's break it down:

● Line 1:
This is how you define a class. By convention, classes start with a capital letter to differentiate them from variable names which follow the snake_case convention.

● Line 3:
This is called the constructor of the class. A constructor is a special type of function that basically answers the question 'What data does this blueprint for creating a Student need to initialise the Student?'. This is why it uses the term 'init' which is short for initialisation. As you can see, age, name, and gender are passed into the function. The constructor function is called automatically when instantiating a new object of a class and therefore the values of the properties can be automatically assigned for that particular instance of the object by using the constructor function.

● Line 4-6:
We also passed in a parameter called `self`. self is a special variable that is hard to define. It is basically a pointer to this Student object that you are creating with your Student class/blueprint. By saying `self.age = age`, you're saying "I'll take the age passed into the constructor, and set the value of the age parameter of THIS Student object I am creating to have that value". The same logic applies to the name and gender variables.

The above piece of code is more powerful than you may think. All OOP programs you write will have this format to define a class (the blueprint). This class now gives us the ability to create thousands of Student objects which have predefined properties. Let's look at this in more detail.

Creating objects from a class

Now that we have a blueprint for a Student, we can use it to create many Student objects. Objects are basically initialised versions of your blueprint. They each have the properties you have defined in your constructor. Let's look at an example. Say we want to create objects from our object representing two students, namely Philani and Sarah.

This is what it looks like in Python:

```
class Student(object):
      def __init__(self, age, name, gender, grades):
            self.age = age
            self.name = name
            self.gender = gender
            self.grades  = grades
philani  = Student(20, "Philani Sithole", "Male", [64,65])
sarah = Student(19, "Sarah Jones", "Female", [82,58])
```

We now have two objects of the class Student called Philani and Sarah.

Pay careful attention to the syntax for creating a new object. As you can see, the age, name, and gender are passed in when defining a new object of type Student.

These two objects are like complex variables. At the moment they can't do much because the class blueprint for Student just stores data, but let's add some actions to the Student class with methods.

Creating methods for a class

Methods allow us to define functions that are shared by all objects of a class to carry out some core computations. Recall how we may want to compute the average mark of every student. The code below allows us to do exactly that:

```
class Student(object):
      def __init__(self, age, name, gender, grades):
            self.age = age
```

```
            self.name = name
            self.gender = gender
            self.grades  = grades

      def compute_average(self):
            average = sum(self.grades)/len(self.grades)
            print("The average for student " + self.name + " is " + str(average))

philani  = Student(20, "Philani Sithole", "Male", [64,65])
sarah = Student(19, "Sarah Jones", "Female", [82,58])

sarah.compute_average()
```

First, notice that we've added a new property for each student, namely grades, which is a list of ints representing a student's marks on two subjects. In our example of university students, this can most certainly be retrieved from a database.

Secondly, notice a new method called compute_average has been defined under the Student class/blueprint. This method takes in self. This just means that this method has access to the specific Student object properties which can be accessed through **`self.___`**. Notice this method uses self.grades and self.name to access the properties for a particular student average calculation.

The program outputs: The average for student Sarah Jones is 70.

This is the output of the method call on line 17. Note the syntax, especially the () for calling this method from one of our objects. Only an object of type Student can call this method, as it is defined only for the Student class/blueprint.

As you can see, we can call the methods of objects that allow us to carry out present calculations. The code for this program is available in your folder in student.py. Every object we define using this blueprint will be able to run this predefined method, effectively allowing us to define hundreds of Student objects and efficiently find their averages with only 11 lines of code - all thanks to OOP!

Class variables vs instance variables

In the examples covered so far the variables that are used as properties are all examples of what we call instance variables. This means that the values are specific to a particular instance of the class (object). There is another type of property used in classes which are known as class variables. These variables have a value that is shared with every instance of that particular class. In order to adhere to DRY (don't

repeat yourself) principles, when a specific property's value needs to be shared across all instances of that class, we can define that variable at the class level i.e. not in the constructor function.

Let's jump into this concept in a bit more detail using some examples. Firstly, let's look at a class that only has class variables:

```
class Wolf:
    classification = "canine"

new_wolf = Wolf()
print(new_wolf.classification) #the output will be "canine"
```

In the above example, we can see that we have a class named Wolf. A property concerning wolves that will not change is the fact that it is a canine. Therefore, any instance of the wolf class will have the property of classification set to "canine" as you will see if you run this example on your computer. We create a wolf object and print out the classification property for that object (note the use of dot notation for accessing the property value `new_wolf.classification`) which will output the value "canine".

Now let's look at how we can use class and instance variables combined so that objects can have shared properties as well as properties that pertain to the specific objects only.

```
class Wolf:

    # Class variables
    classification = "canine"
    habitat = "forest"

    # Constructor method with instance variables name and age
    def __init__(self, name, age):
        self.name = name
        self.age = age

def main():
    # First object, set up instance variables of constructor method
    silver_tooth = Wolf("Silvertooth", 5)

    # Print out instance variable name
    print(silver_tooth.name)
```

```
    # Print out class variable habitat
    print(silver_tooth.habitat)

    # Second object
    lone_wolf = Wolf("Lone Wolf", 8)

    # Print out instance variable name
    print(lone_wolf.name)

    # Print out class variable classification
    print(lone_wolf.classification)

main()
```

In the above example, both Wolf objects that were created have habitat and classification properties in common, but they each have their own name and age properties that are specific to them. So when creating objects, we eliminate the need to also declare the values of the class variables thereby eliminating repetition.

Changing property values from inside the object

From within an object, it is possible to change the property values when a specific method has been called. The following example shows how we can do this:

```
class Wolf:

    # Class variables
    classification = "canine"
    habitat = "forest"
    is_sleeping = False

    # Constructor method with instance variables name and age
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # method to wake up wolf (self needs to be passed as argument so
    # that all of the properties are available to the method)
    def wake_up(self):
        self.is_sleeping = False

    # method to put wolf to sleep
    def sleep(self):
```

```
        self.is_sleeping = True

    # method that returns the sleep state of the wolf
    def show_sleep_state(self):
        if self.is_sleeping == False:
            return self.name + " is awake"
        else:
            return self.name + " is sleeping"

def main():
    # initialising a wolf object and printing the initial sleep
    # state which is awake
    silver_tooth = Wolf("Silver Tooth", 6)
    print(silver_tooth.show_sleep_state())

    # changing sleep state to sleeping and then printing that state
    silver_tooth.sleep()
    print(silver_tooth.show_sleep_state())

# running main method
main()
```

You can run the above code in your IDE and see what output is generated. You are encouraged to try to add your own properties and find creative ways to change the values of those properties.

Changing property variable values from outside the object:

We can also change the property values from outside of the object without using methods by using dot notation. The following example will show how this can be done:

```
class Wolf:

    # Class variables
    classification = "canine"
    habitat = "forest"
    is_sleeping = False

    # Constructor method with instance variables name and age
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
    # method that returns the sleep state of the wolf
    def show_sleep_state(self):
        if self.is_sleeping == False:
            return self.name + " is awake"
        else:
            return self.name + " is sleeping"

def main():
    # initialising a wolf object and printing the initial sleep
    # state which is awake
    silver_tooth = Wolf("Silver Tooth", 6)
    print(silver_tooth.show_sleep_state())

    # changing sleep state to sleeping using dot notation and then
printing that state
    silver_tooth.is_sleeping = True
    print(silver_tooth.show_sleep_state())

# running main method
main()
```

Instructions

First, read example.py, open it using VS Code.

Read through **example.py** carefully and then run the code and examine the output. When you feel confident that you understand the concept, you can move on to completing the compulsory task below.

Compulsory Task 1

In this task, we're going to be creating an **Email Simulator** using OOP. Follow the instruction and fill in the rest of logic to fulfil the below program requirements.

- Open the file called **email.py**.
- Create an **Email class** and initialise a constructor that takes in three arguments:
    - `email_address` - the email address of the sender
    - `subject_line` - the subject line of the email.
    - `email_content` - the contents of the email.
- The email class should contain the following class **variable** and default value:
    - `has_been_read` - initialised to False.
- The Email class should also contain the following class **method** to edit the values of the email objects:
    - `mark_as_read` which should change `has_been_read` to True.
- Initialise an empty `Inbox` **list** to store, and access, the email objects. **Note**: you can have a list of objects.
- Create the following **functions** to add functionality to your email simulator:
    - `populate_inbox()` - a function which creates an email object with the email address, subject line and contents, and stores it in the *Inbox* list.

      Note: At program start-up, this function should be used to populate your Inbox with three sample email objects for further use in your program. This function does not need to be included as a menu option for the user.
    - `list_emails()` - a function that loops through the Inbox and prints the email's *subject_line*, along with a corresponding number. For example, if there are three emails in the Inbox:

      0 Welcome to HyperionDev!
      1 Great work on the bootcamp!
      2 Your excellent marks!

This function can be used to list the messages when the user chooses to read, mark as spam, and delete an email.

Tip: Use the `enumerate()` function for this function.

- `read_email()` - a function that displays a selected email, together with the *email_address*, *subject_line,* and *email_contents,* and then sets its `has_been_read` instance variable to True.

  For this, allow the user to input an index i.e. `read_email(i)` prints the email stored at position `i` in the list. Following the example above, an index of 0 will print the email with the subject line "Welcome to HyperionDev!".

- Your task is to build out the Class, Methods, Lists, and functions to get everything working! Fill in the rest of the logic for what should happen when the user chooses to:

```
1. Read an email
2. View unread emails
3. Quit application
```

Note: menu option 2 does not require a function. Access the corresponding class variable to retrieve the *subject_line* only.

- Keep the readability of print outputs in mind and take initiative to show the user what is being viewed and what has been executed.

  For example: `print(f"\nEmail from {email.email_address} marked as read.\n")`

Completed the task(s)?

Ask an expert to review your work!

Review work

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.