Hyperiondev

TASK

OOP – Classes

Visit our website

Introduction

WELCOME TO THE OOP – CLASSES 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 the components of OOP. This task focuses on the concept of classes, the blueprints we use to create multiple objects of the same type.

WHY OOP?

Imagine we want to build a program for a university. This program will use a database of students, their information, and their grades. We need to perform computations with 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 grade 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 OOP components such as 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 a **class**.

If you run the following code you will see that a list, boolean, string, and function are all of the type “class”.

```
# Examples
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 is as follows:

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

Everything in Python is an object built from a particular class.

Reflection

What other classes have you interacted with?

THE CLASS

A **class** is a specific Python file that can be thought of as a 'blueprint' for a specific data type. The concept of a class may be hard to get your head around at first. You can think of a class as defining your own special data types, with properties you determine.

A class stores properties and functions called **methods** which run programming logic to modify or return the class properties. The `String` class, for example, has an attribute which is the value of the string and methods such as `lower()`, `upper()`, `split()` etc. So far, all the classes you have used have been built-in. What we are going to work towards is an understanding of Python classes sufficient to enable you to create your own simple classes and objects created from these classes.

In the example that we discussed earlier (building a program for a university around a database of students), we could use a class called `Student` to represent a student. This is perfect because we can set the properties of a student to 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
```

A lot about this will be new to you, and 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 2:
This is called the **constructor** of the class. A constructor is a special type of function that answers the question 'What attribute data does this blueprint need to initialise a student object?'. The term 'init', short for initialisation, acts as a reminder of the purpose of the constructor. As you can see, age, name, and gender are

passed into the function (as well as something called `self`, which we'll look at in the next point). 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.

● Lines 3-5:
We also passed in a parameter called `self`. This special variable (`self`) is a pointer to the `Student` object that you are creating with your `Student` class. 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 class representing two students called Philani and Sarah.

This is what it looks like in Python:

```
# Create Student class
class Student():
    # Constructor method
    def __init__(self, age, name, gender):
        self.age = age
        self.name = name
        self.gender = gender


# Create Student objects
philani  = Student(20, "Philani Sithole", "Male")
sarah = Student(19, "Sarah Jones", "Female")
```

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 a student just stores data, but we can 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 grade of every student. The code below allows us to do exactly that:

```
# Create  Student class
class Student():
    # Constructor method
    def __init__(self, age, name, gender, grades):
        self.age = age
        self.name = name
        self.gender = gender
        self.grades  = grades

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

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

# Method call
sarah.compute_average()
```

First, notice that we've added a new attribute for each student, namely grades, which is a list of integers representing a student's grades in 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. This method takes `self` as an argument. This 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's average calculation.

The program outputs:

```
The average for student Sarah Jones is 70.
```

This is the output of the method call on the last line. 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.

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 class will be able to run this predefined method, effectively allowing us to define hundreds of `Student` objects and efficiently find their averages - 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 attribute used in classes, namely 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 attribute'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 consider this concept in a bit more detail using some examples. Firstly, let's look at a class that only has class variables:

```
class Wolf:
    # Class variable
    classification = "canine"

# Create Wolf object
new_wolf = Wolf()

# Print classification (class variable) for new_wolf
print(new_wolf.classification)
```

In the above example, we can see that we have a class named `Wolf`. An attribute of all wolves that will not change is the fact that a wolf is a canine. Therefore, any instance of the `Wolf` class will have the attribute of classification set to "canine". When we create the `new_wolf` object and print it out, the classification attribute for that object will be "canine". Take note of the dot notation for accessing the attribute value: `new_wolf.classification`.

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 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

# First object, provide instance variables for the 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)
```

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. When creating objects, we eliminate the need to also declare the values of the class variables thereby eliminating repetition.

Changing attribute values from inside the object

From within an object, it is possible to change the attribute 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 # Defaults to being awake initially

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

    # Method to put wolf to sleep (self needs to be passed as argument
    # so that all of the properties are available to the method)
    def bed_time(self):
        self.is_sleeping = True

    # 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
```

Changing attribute values from outside the object

We can also change the attribute 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 # Defaults to being awake initially

    # 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"


# Initialise a wolf object and print the initial sleep
# State which is awake
silver_tooth = Wolf("Silver Tooth", 6)
print(silver_tooth.show_sleep_state())

# Change sleep state to sleeping using dot notation and then print new
state
silver_tooth.is_sleeping = True
print(silver_tooth.show_sleep_state())
```

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.

Instructions

First, read and run the **example files** provided. Feel free to write and run your own example code before doing the Practical Task, to become more comfortable with the concepts covered in this task.

Practical Task

In this task, we're going to be creating an **email simulator** using OOP. Follow the instructions and complete the logic to fulfil the program requirements below in email.py.

- 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.
- Inside the constructor, initialize the following instance variables:
    - `email_address`
    - `subject_line`
    - `email_content`
    - `has_been_read` (initialised to `False`).
- The `Email` class should also contain the following instance **method** to edit the values of the email objects:
    - Implement an instance method called `mark_as_read()` that sets the `has_been_read` instance variable to `True`.
- Initialise an empty variable called `inbox` of type **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 that creates an email object with the email address, subject line, and contents, and stores it in the `inbox` list.

Note: At program startup, 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 each email's `subject_line` and 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.

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

  For this, allow the user to input an index, such that `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 the initiative to communicate with the user, making it clear to them what is being viewed and what has been executed.

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

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.