Hyperion Development

Task: Object Orientated Programming (Part 1)

www.hyperiondev.com

Introduction

What is Object Oriented Programming (OOP)?

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.

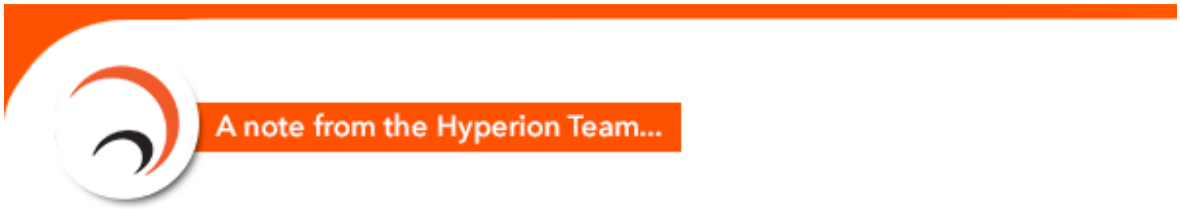
Your first step to building more complex programs is understanding OOP. This may be the first truly abstract concept you encounter in programming but don't worry practice will show you that once you get past the terminology, OOP is very simple.

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.



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. We have discussed different data types in previous tasks, and 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.

We would use a class called Student to represent a student in the above example. 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:

```
1  class Student():
2
3      def __init__(self, age, name, gender):
4          self.age = age
5          self.name = name
6          self.gender = gender
7
```

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 camelBackNotation.

● 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 initialization. As you can see, age, name, and gender are passed into the 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 for 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:

```
1  class Student(object):
2
3      def __init__(self, age, name, gender, grades):
4          self.age = age
5          self.name = name
6          self.gender = gender
7          self.grades = grades
8
9  philani = Student(20, "Philani Sithole", "Male", [64,65])
10 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 extend 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:

```
1  class Student(object):
2
3      def __init__(self, age, name, gender, grades):
4          self.age = age
5          self.name = name
6          self.gender = gender
7          self.grades = grades
8
9      def compute_average(self):
10         average = sum(self.grades)/len(self.grades)
11         print "The average for student " + self.name + " is " + str(average)
12
13
14 philani = Student(20, "Philani Sithole", "Male", [64,65])
15 sarah = Student(19, "Sarah Jones", "Female", [82,58])
16
17 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 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!

Instructions

- Feel free at any point to refer back to previous material if you get stuck. Remember that if you require more assistance our tutors are always more than willing to help you!
- Complete the following compulsory task to progress onto the next task!
- We highly recommend that you do the optional tasks located in **example.py**.

Compulsory Task

In this Task we're going to be simulating an SMS message. Some of the logic has been filled out for you in the **sms.py** file.

- Open the file called **sms.py**.
- Create a class definition for an SMSMessage which has three variables: hasBeenRead, messageText, and fromNumber.
- The constructor should initialise the sender's number.
- The constructor should also initialise hasBeenRead to false.
- Create a method in this class called MarkAsRead which should change hasBeenRead to true.
- Create a list called SMSStore to be used as the inbox.
- Then create the following methods:
    - add_sms - which takes in the text and number from the received sms to make a new SMSMessage object.
    - get_count - returns the number of messages in the store.
    - get_message - returns the text of a message in the list. For this, allow the user to input an index i.e. GetMessage(i) returns the message stored at position i in the list. Once this has been done, hasBeenRead should now be true.
    - get_unread_messages - should return a list of all the messages which haven't been read.
    - remove - removes a message in the SMSStore.

Now that you have these set up, let's get everything working!

- Fill in the rest of the logic for what should happen when the user inputs send/read/quit. Some of it has been done for you.

Still need help?

Just write your queries in your comments.txt file and your tutor will respond.

Task Statistics

Last update to task: 07/01/2016.
Course Content Manager: Riaz Moola.
Task Authors: Riaz Moola and Brandon Haschick
Task Feedback link: Hyperion Development Feedback.