Hyperion Development

www.hyperiondev.com

Task 4: Object Oriented Programming

Python Beginner Course

Task 4: Object Oriented Programming

4.1 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 the 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?

Image 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 in code?
- We need to write code to find the average of a student - simply summing their grades for different subjects, and dividing by the number of subject - 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.

4.2 The components of OOP

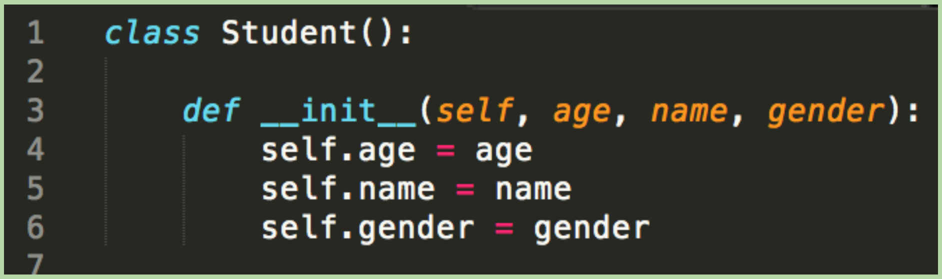
The Class

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 classes **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, but lets break it down:

- Line 1: This is how you define a class. By convention **classes start with a capital letter.**

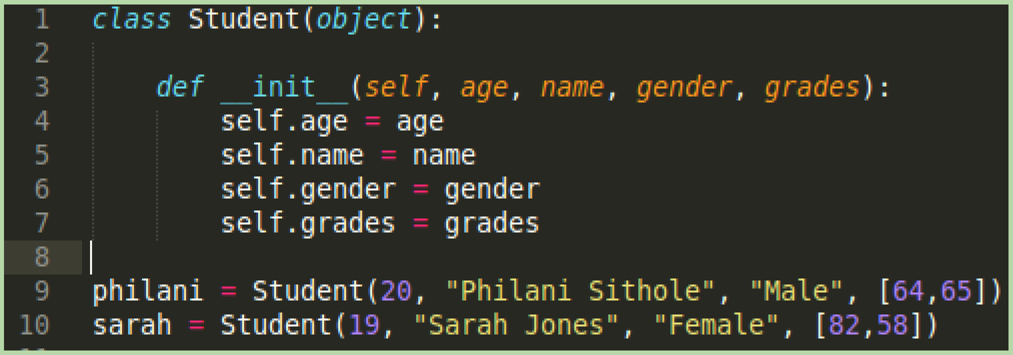
- Line 3: This is called the **constructor** of the class. A constructor is a **special type of function** that basically answers the question - 'What data does this blueprint for creating a Student need to initialise the Student?'. This is why it uses the term **'init'** which is short for initialization. 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, or blueprint. This class now gives us the ability to create thousand 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 - Philani and Sarah - this is what it looks like in Python:



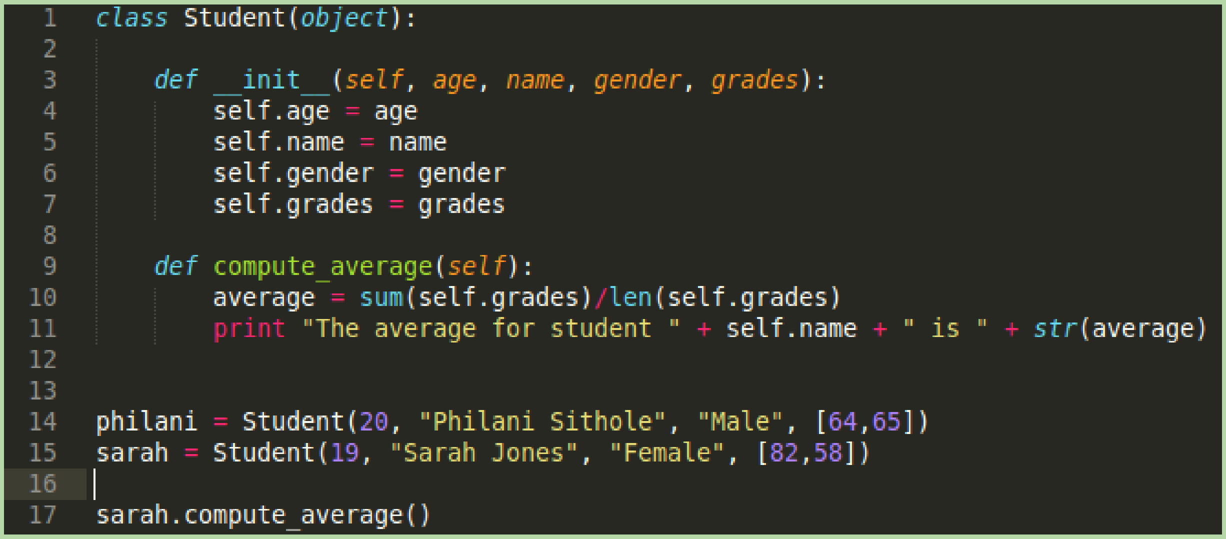
```python
class Student(object):

    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 is passed in when defining a new object of 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 computation. 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 - 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 '**This method has access to the specific Student objects properties which can be accessed through self.___'.** Notice how in this method using **self.grades** and **self.name** to access the properties for a particular average.

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 blueprint.

As you can see, we can call the methods of objects that allow us to carry out present calculations. This code 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!

4.3 Compulsory Task

1.1 Extending student.py

- Create a new student object that matches your name, age, and gender.
- Extend the Student class found in the student.py file in your folder by adding another method called **check_male(self)** that prints out True if a Student is Male, and False if a Student is Female.
- Call the check_male method from one of your objects - does it work?
- Now, place all of three of your Student objects in a list. Loop over this list using a for loop (see previous Tasks), and call both check_male and compute_average on each object. Can you now see the power of OOP?
- Save your student.py file as student_completed.py.

1.2 Creating your own OOP program

Carefully review Part 1 of example.py for another example of an OOP program.

- Think about some real-world entity that may be useful to build a Class of in Python - preferably something that interests you. This could be anything from a book to a football match - consider how almost anything can be represented in OOP.
- Create a file called myClass.py and define the constructor for your class, as well as properties that you think are **common to all objects of that Class in the real world (**for example, all books have an **author).**
- Also create at least one method that would be useful for all objects (this could be as simple as returning one of the properties, eg **return_author** which prints out the author name).

Need help?

Just write your queries in your comments.txt file and your tutor will respond. Alternatively, visit www.hyperiondev.com and login to the Hyperion portal to views all methods of getting help.

4.4 Optional Task

Inheritance

Inheritance is a bit more complicated. Suppose you had written a class with some properties and methods, and you wanted to define another class with most of the same properties and methods plus some additional structure. Inheritance is the mechanism by which you can produce the 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 properties 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, and the class that inherits from the base class is called the "derived" class.

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 which act on base objects may also act on derived objects, a fact which is often useful when writing an object-oriented program. (Note: this distinction is arguably more important in languages with a stronger type system; Python takes everything to an extreme by making all user-defined classes subclasses of a base type confusingly referred to as 'object'. If you are interested in the internals of the Python language (and you should be, because it is interesting!), master the material in this task and then try looking at the relevant material in the Further Reading folder.)

Read Part 2 of example.py for concrete, cow-themed demonstrations of composition, inheritance, and some other Python-specific object-oriented language features.

Now, open exercise.py and fill out the logic for the method definitions. Rename this file to exercise_completed.py when complete. You may leave any questions you have for your tutor in comments.txt.

4.5 Task Statistics

Estimated time for completion: 4 hours self study.
Last update to task: 09/12/2014.
Author: Riaz Moola, Optional Task: Eli Bingham.
Task Feedback link: http://tinyurl.com/hyperionKZN