# CS-119 Lab #8

# Expected Learning Objectives

* Classes and objects
* Implementing class models
* Protected access
* Inheritance – superclasses and subclasses
* Class constructors
* Class get() and set() methods
* Abstract methods
* Overriding methods

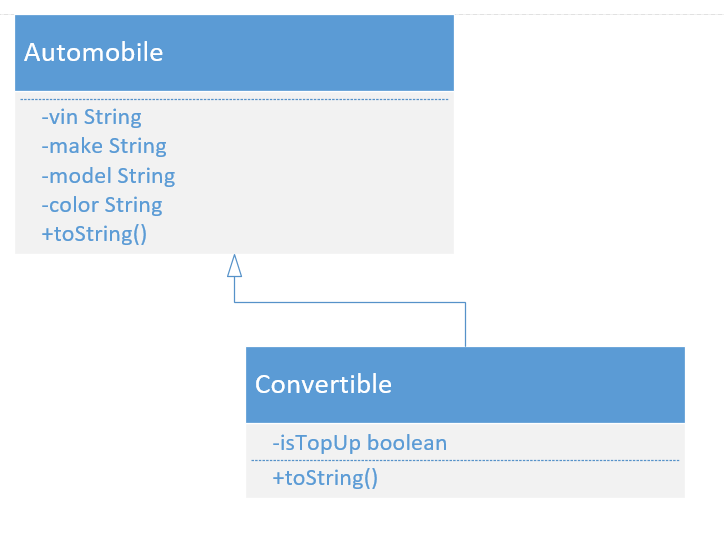
## Overview

This lab will give you some hands-on experience creating user defined classes using inheritance in Python and creating instances in a “test fixture” application. Classes such as these are commonly implemented, tested and debugged in a small scale test environment and then later integrated into a larger scale project.

## Exercise 1 Automobile

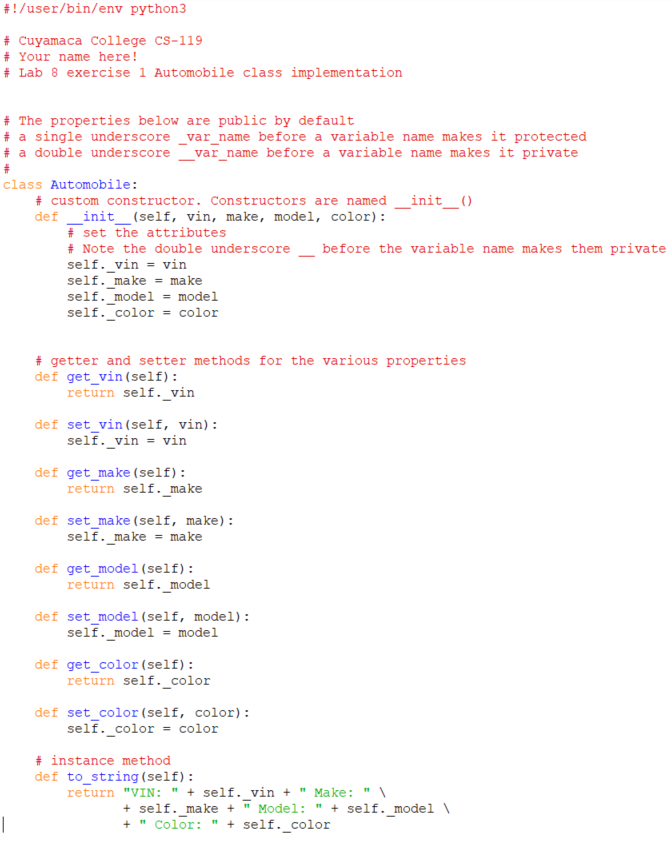
This exercise is based on exercise 1 on page 333 of the textbook.

Open a new UML drawing in Draw.io. Drag and drop 2 classes onto the page. Fill in the class name, attributes and properties. From the exercise, we are given the Automobile class properties are Vehicle Identification Number (VIN), make, model and color. We’ll add a to\_string() method to return a string of all the properties. The Convertible class has the added property to indicate whether the top is up and it will have a to\_string() method we’ll override. When showing an inheritance relationship in a class model, the key thing is to have the inheritance line with the arrow pointing to the parent or base class. The class model should look something like this:



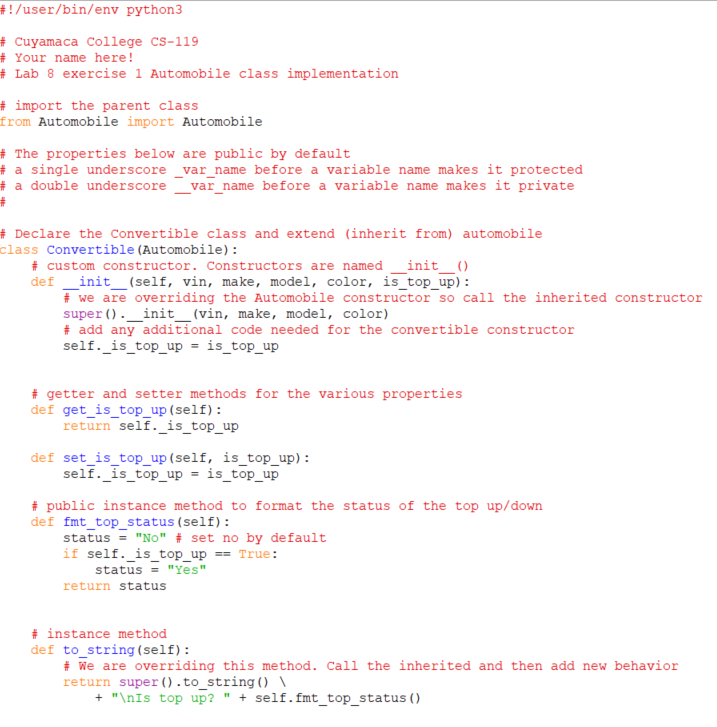
### Automobile Class Python Code

Create a new code file in Python called Automobile and a class called Automobile, code the properties, constructor, getters/setters, and to\_string() method. This class will be coded like the ones we did in the last chapter with the exception that our attributes will be declared with *protected* access. Protected access is just like private except that it allows derived (inherited) classes direct access (see page 316) .



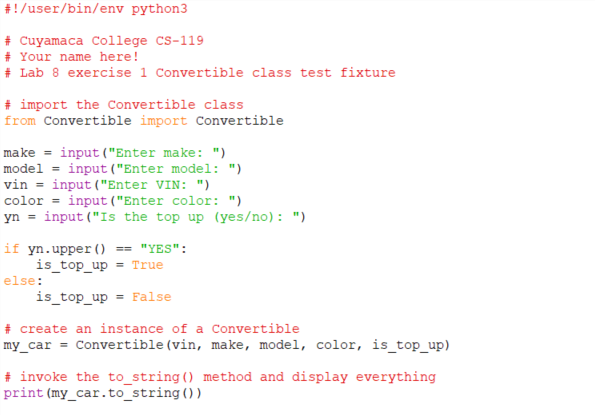
### The Convertible Class

Now we’re going to create a new code file called Convertible in Python. The added twist is in the class header where we have *class Convertible( Automobile ),* and in the constructors where we first call the inherited class constructor by using the super().\_\_init\_\_ statement and then add any new initialization behavior needed.



### Car Test Fixture

Create a new Python code file called CarTest.py. A screen shot of the code is provided below. There shouldn’t be much in the way of surprises for the Python code.



## Exercise 2 Race Car class

Create a Python code file called Racecar.py and a class called RaceCar that inherits from the Automobile class you did in exercise 1. It will have one additional property called number of horse power. Add the class model to the Automobile class model in the last exercise and show an inheritance relationship.

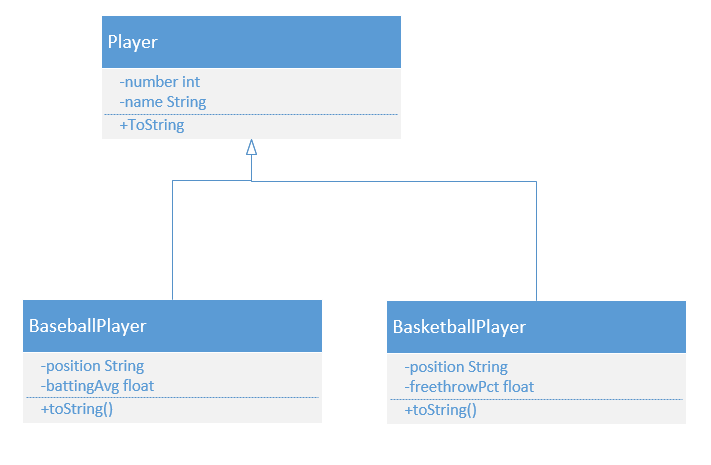
You may add the code needed to prompt the user for the number of horse power, create a race car object and call the to \_string() method to your CarTest test fixture application.

## Exercise 3 Player (Page 334, Exercise 4)

In this exercise, you’ll create a general class called Player and then derive (inherit) 2 classes, BaseballPlayer and BasketballPlayer, from it.

### Player Class Model

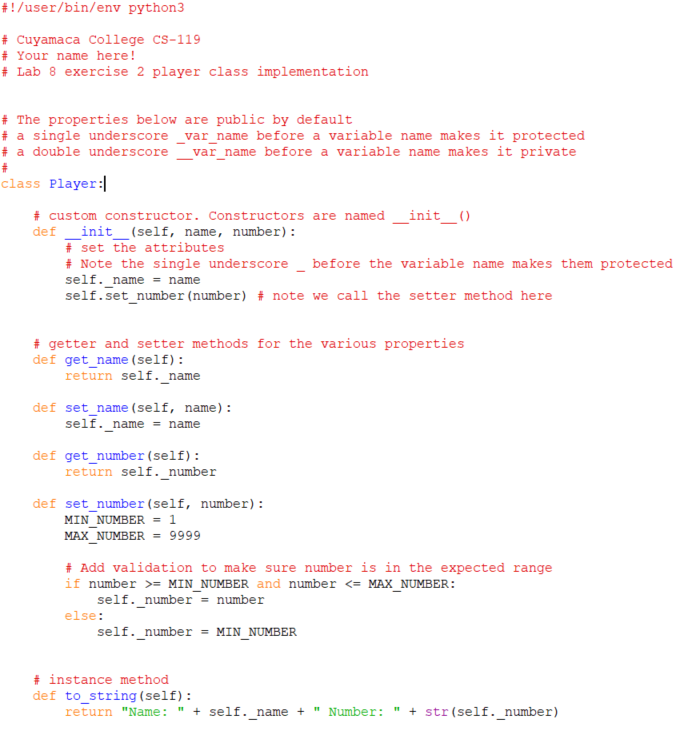
Create a new UML class model in Draw.io. Drag and drop a total of 3 classes onto the page, fill in the class names, attributes for each class, and a toString() method for each class. Show an inheritance relationship where the Player class is the parent class for BaseballPlayer and BasketballPlayer.



### Player Python Code

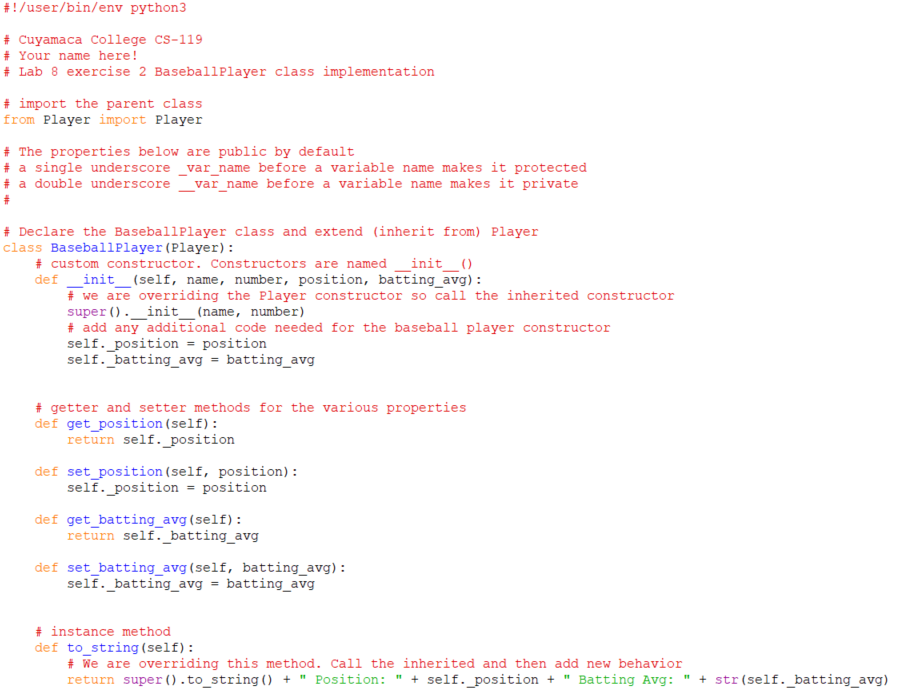
Create a new Python code file called Player.py. Code a new class called Player. Implement the attributes, constructors, getter and setter methods, and the to\_string() method.

One added twist to the code is that we will add some validation to setNumber to make sure a player’s number is within some “reasonable” range. Notice the custom constructor will call setNumber(). The player class should look like the Python code screen shot below.

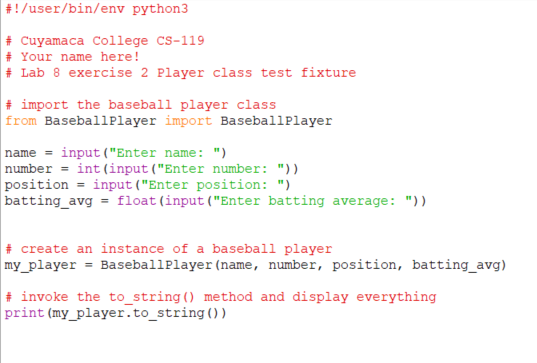


### BaseballPlayer Class

Create a Python code file called BaseballPlayer.py, create a new class called BaseballPlayer, implement the attributes, constructors, getters/setters and to\_string() method. Python screen shots are provided below. In the class header, be sure it extends Player. Notice there’s less code by inheriting from another class. Also note the user of the super() statement to call the inherited class constructor. In the to\_string() method, notice how the statement super().to\_string() calls the to\_string() method in the parent class saving some work.



### Player Test Fixture



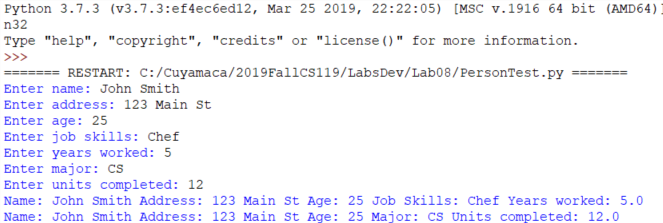
## Exercise 4

Implement the BasketballPlayer class. This inherits from the Player class and has the added properties position and free throw percentage. You may add the code to prompt for free throw percentage, create the basketball player object and call the to\_string() to your PlayerTest test fixture application.

## Exercise 5

In this exercise, we will create a base class (superclass) and two inherited classes (subclasses). To keep things reasonably simple, we’re going to implement just a few properties and a single method. In real world software development, classes like these would contain *numerous* properties and methods. Our goal is to learn the object oriented design/programming concepts, Python syntax for classes and inheritance relationships.

This exercise is very similar to what did in the previous exercises. This time, the instructions aren’t going to be quite as detailed.

1. In Draw.io, create a new drawing using the UML template. Create a class model for a class called *Person* with the following properties: name, address, age. It will have a to\_string() method that will display a formatted string of all the properties.
2. In your class model, add a class called employee. Show an inheritance (often called a *generalization*) relationship from the person class. With inheritance, the arrow points to the superclass or base class. In the employee class, add the following properties: job skills and years worked. The employee class will override the to\_string() method.
3. Add a class called student to your model. Show an inheritance relationship from the person class. In the student class, add the following properties: major and units completed. The student class will also override the to\_string() method..
4. Time for some Python code! Create a code file called Person.py and a class called Person. Implement the properties and methods required.
5. Create a Python code file called Employee.py. Implement the Employee class with the required properties and methods. Be sure to inherit from the Person class!
6. Create a Python code file called Student.py. Implement the Student class with the required properties and methods. Be sure to inherit from the Person class!
7. Create a Python code file called PersonTest.py. Prompt for the inputs needed to test the student and employee classes. Create an instance of each and display the values. Sample program run:  
     
   

Grading Criteria:

| Deliverable | Points | Breakdown |
| --- | --- | --- |
| Car exercise 1 | 10 | Completed, all classes implemented, produces correct output. UML model completed showing all classes and relationships |
| Race car exercise 2 | 5 | All properties and methods implemented per lab handout, integrated into the Car exercise. UML class model added to car exercise class model. |
| Player exercise 3 | 10 | Completed, all classes implemented, produces correct output. UML model completed showing all classes and relationships |
| Basketball player class exercise 4 | 5 | All properties and methods implemented per lab handout, integrated into the Player exercise. UML class model added to car exercise class model |
| Person, Employee, Student exercise 5 | 20 | Completed, all classes implemented, produces correct output. UML model completed showing all classes and relationships |
| **Lab Total** | **50** |  |