CS 30 Lab 9 — Introduction to Object-Oriented Programming

This lab gives you some practice with simple object-oriented programming. Don't hesitate to talk things over with your lab mates, or call me or Dan over for help or answers to questions.

---

1. Copy the files for this lab to your own directory using the following Linux commands:

```
cp -r /common/cs/cs30/lab9/ .
cd lab9
ls
```

2. Create a new file called **Car.py** and implement a class called Car with the following properties. A Car has a make, model, and year, a certain fuel efficiency (measured in miles per gallon), and a certain amount of fuel in the gas tank (measured in gallons). First, we need a constructor method `__init__` that takes the make, model, year, and miles-per-gallon and initializes the Car's internal variables accordingly:

```
class Car:

   def __init__(self, make, model, year, mpg):
      self.carMake = make
      self.carModel = model
      self.carYear = year
      self.efficiency = mpg
      self.fuelLevel = 0.0
```

   We can then create a new car object like this: `mycar = Car("Honda", "Accord", 2004, 28)`

3. Next, let's add a `__str__` method so that we can print out `Car` objects in a nice way. We'll have `__str__` return a string such as "2004 Honda Accord: 28 MPG, 6.1 gallons of fuel left":

```
   def __str__(self):
      s = "%d %s %s: %d MPG, %.1f gallons of fuel left" % \
          (self.carYear, self.carMake, self.carModel, self.efficiency,
           self.fuelLevel)
      return s
```

   Now test your code by creating a few `Car` objects and printing them:

```
mycar = Car("Honda", "Accord", 2004, 28)
myothercar = Car("Rolls Royce", "Phantom", 1968, 15)
print mycar
print myothercar
```

4. Next, adds methods called `addGas` and `drive` to your `Car` class. The `addGas` method should take an amount of gas (in gallons) as input and add it to the car's internal fuel level. The `drive` method should take a distance (in miles) as input and reduce the car's fuel level by the amount needed to drive the given distance. If the fuel needed is greater than the amount in the tank, it should be set to zero and a message "Ran out of gas" should be printed. Your methods should behave as shown below:

```
>>> mycar = Car("Honda", "Accord", 2004, 28)
>>> mycar.addGas(10)
Added 10 gallons of gas to the tank
>>> mycar.drive(150)
Drove 150 miles.  4.6 gallons of gas left
>>> mycar.drive(200)
Ran out of gas after 130 miles
```

   Hint: the amount of fuel needed to drive d miles is $d \div mpg$, and the distance traveled on g gallons of gas is $g \times mpg$.

5. The file **Student.py** defines the `Student` class. Take a look at this code and try out the `main()` test program. Each `Student` object currently keeps track of a student's name and graduation year. Add a new instance variable called `self.gpa` to keep track of a student's GPA. You'll need to modify `__init__` so that it takes an initial GPA value in addition to the student's name and graduation year. Add a method called `getGPA` that takes no input parameters and just returns the value of `self.gpa`. Also, change the `__str__` method so that it shows the GPA along with the graduation date. Test your code.

6. Since Pomona uses a somewhat unusual 12.0 scale for the GPA, it would be nice to have a method that returns a student's GPA using the more traditional 4.0 scale. Add a method called `getGPA4` to your `Student` class that returns the student's GPA on this scale. Test your code to make sure it works.

7. Implement a `BeerCan` class (in a separate file **BeerCan.py**) with methods `getSurfaceArea` and `getVolume`. The `__init__` method for `BeerCan` objects should take the height and radius of the can as input (in inches). The formulas for calculating the surface area and volume of a cylinder are given below. Test your code.

$$SurfaceArea = 2\,\pi\,r\,(r + h)$$
$$Volume = \pi\,r^2\,h$$

8. Now modify `BeerCan` objects so that they can hold beer. Add a `fill` method that takes an integer and fills the can with that many ounces of beer, and a `drain` method that does the opposite. Also, modify your `__init__` constructor to initialize each new can with some amount of beer. Next, add a `drink` method to your `Student` class that accepts a can of beer as input and empties it. Finally, add an instance variable called `self.count` to the `Student` class that keeps track of how many cans of beer a student drinks. Change `getGPA` and `getGPA4` so that they return a student's GPA divided by the number of beers the student has had. If you like, you could also add a `soberUp` method that resets the beer count back to zero.

   In order for `Student` objects to be able to use `BeerCan` objects, you'll need to put the following statement at the top of your **Student.py** file:

   ```
   from BeerCan import *
   ```

   In the `main()` test program, create a few cans of beer for your students to drink and test out your code to make sure everything works.

9. Study the code in **Neuron.py**, which models a simple neuron. Add a method called `test()` that takes no input parameters (besides `self`) and reports the neuron's output for each of the input patterns `[0, 0]`, `[0, 1]`, `[1, 0]`, and `[1, 1]`. Since the neuron's weights are initially zero, you should see an output of zero for all four patterns.

10. Try changing the neuron's weights from zeros to something else and then call `test` to see the effect. Does the neuron's output change for some of the patterns? You can change the weights interactively by directly assigning a new list of values to the neuron's `weights` variable. For example:

    ```
    >>> n.weights = [0.3, 0.4]
    >>> n.test()
    ```

    Can you find sets of weights that implement the logical functions NAND and NOR? The outputs for NAND on the four patterns would be 1 1 1 0, respectively, and 1 0 0 0 for NOR.

11. Add another method called `randomize()` that sets all of the neuron's weights to small random values in the range -0.1 to +0.1. What is the neuron's performance on the patterns using random weights?