420-LCU-05 Programming in Python - Lab Exercise 7

October 30, 2017

Goals for this lab:

- Explore some basic concepts of object-oriented programming.

Your submission for this exercise should be either a text or Word file. You do NOT need to copy the questions into the answer file, just write answers and/or copy and paste results from your IDLE session. Any code you include can also be pasted into your submission - there is no need for a separate `.py` file.

You should also show your work by copying and pasting the entire IDLE session at the end of your submission.

1 Class basics

In this part we will just try a few things using the Python shell.

1. Using IDLE or another Python editor, open and read the file `point.py` that we have provided. Notice that the header of the `class` statement specifies that our new class inherits from the builtin class `object`. Notice also that it implements the following five methods:
   (a) `__init__` - the constructor. This is called *implicitly* by Python when you create a new `point` object.
   (b) `__eq__` - a special method that implements comparison of points for equality testing (the '==' operator).
   (c) `__repr__` - a special method that handles conversion of a point to a string format.
   (d) `copy` - this method, like the method in the `list` class, makes an identical copy of a `point` object. Notice how this is done - this method just calls the constructor of the `point` class, with the two attributes provided as arguments.
   (e) `distance` - computes the Euclidean distance between two points.

2. Import the `point` class into your Python shell by typing:

   ```
   >>> import point
   >>> a = point()
   ```

   What happens and why?

3. The `point` class is a simple example of *composition*. We build a single point object using two numeric values.

4. Now we'll try to actually create our first `point` objects. Type the following:

   ```
   >>> a = point.point(1, 2)
   >>> b = point.point(4, 6)
   >>> print(a, b)
   >>> print(a.x, b.x, a.y, b.y)
   ```

   What is printed, and why?

5. Try the following code to use the `distance` method:

   ```
   >>> print(a.distance(b))
   >>> print(b.distance(a))
   >>> print(b.distance(b))
   ```

   What is printed, and why?

6. As we've mentioned, the `distance` method is stored as a name "inside" the `point` class. It is *not* a global. To see this, try the following and describe the results:

   ```
   >>> print(distance(a, b))
   ```

7. By default, the attributes of the objects we create are mutable. Try the following code and show what it prints:

   ```
   >>> a.x = 3
   >>> print(a, b)
   ```

8. Because attributes and objects are mutable, they can produce some of the odd results we see with `list` objects. Try the following:

   ```
   >>> c = b
   >>> c.x = a.x + 3
   >>> print(a, b, c, b is c)
   ```

   Explain what you see here.

9. Now we'll try the `copy` method. Try the following:

   ```
   >>> c = b.copy()
   >>> c.x = 10
   >>> print(a, b, c, b is c)
   ```

   Explain what is different this time. How does the `copy` method of our `point` class relate to the `copy` method of the `list` class?

10. Now we'll try to use the `__eq__` method. We normally rely on Python to call this implicitly for either a `==` or `!=` operator is evaluated, but we *can* choose to call it explicitly if we want to. Try the following code:

    ```
    >>> c = b.copy()
    >>> print(a == b, c == b, a != b, c != b)
    >>> print(a.__eq__(b), c.__eq__(b))
    ```

    What is printed, and why?

11. Lastly, we'll introduce a new builtin function `isinstance(A, B)`. This function answers the question "Is expression A a member of class B?". For example, if we type the code:

    ```
    >>> print(isinstance(1, int), isinstance("a", str), isinstance([], int))
    ```

    it will print `True True False` because 1 is an `int` and `"a"` is a `str`, but the empty list is not an `int`. Now you try the following:

    ```
    >>> from point import point
    >>> a = point(1, 2)
    >>> print(isinstance(a, point), isinstance(a, object), isinstance(a, int))
    >>> print(isinstance(a.x, point), isinstance(a.x, object), isinstance(a.x, int))
    ```

    What does this print, and why?

2 Modifying the class

Now we will implement some changes to the `point` class. Every time you make a change to the class, remember that you will have to:

1. Save the modified file.

2. Re-initialize the shell using the Shell/Restart Shell menu command in IDLE.

3. Re-import the class with an `import` statement.

4. Create a new `point` object.

Otherwise you won't see the effect of your changes. One helpful hint is to remember that, even after restarting the shell, you can recall previous lines of code you've typed using the "up arrow" key. This can save you some typing.

1. Now go back to the file `point.py` and modify the `__repr__` method to look like this:

   ```
       def __repr__(self):
           return "POINT: " + str(self.x) + ", " + str(self.y)
   ```

   Now, following the steps above, execute the following code:

```
>>> import point
>>> a = point.point(1, 2)
>>> b = point.point(4, 6)
>>> print(a, b)
```

What changes and why?

2. By now you may have noticed that, even in a method function, we always refer to the attributes of an object using the notation `self.x`, for example. We *cannot* just write `x` or `y` to refer to the attributes stored in the object associated with `self`. Why do you think this is the case?

3. Implicitly-called functions can often misbehave if we aren't careful - it is up to the programmer to provide Python with a correct implementation. Let's temporarily replace our `__eq__` method with a bogus implementation. You are going to want to restore the correct implementation after you do this part, so you might want to "comment out" the correct implementation by placing a # character in front of the lines of code you are temporarily replacing. This makes it easy to restore those lines when the time is right. Once you're ready, replace the correct method with this code:

   ```
   def __eq__(self, other):
       print("I'm comparing", self, "and", other)
       return True
   ```

   Now follow the steps outlined above and type in the following code:

   ```
   >>> import point
   >>> a = point.point(1, 2)
   >>> b = point.point(3, 4)
   >>> print(a == b)
   ```

   What did we do? Be sure to restore the method once you've tried this.

4. In mathematics, the addition of two points `a` and `b` yields a new point `c` whose whose coordinates are the sum of those of points `a` and `b`:

   $$x_c = x_a + x_b$$

   and

   $$y_c = y_a + y_b.$$

   We can create this behavior in Python with a little work. But first, try to execute the following code:

   ```
   >>> import point
   >>> a = point.point(1, 2)
   >>> b = point.point(3, 4)
   >>> c = a + b
   ```

   What happens?

5. Again let's modify `point.py` to make the previous example work better. We need to add a new special method called `__add__`. It will take two arguments like `__eq__`, but instead of returning a Boolean value, it will return a new `point`.

   ```
   def __add__(self, other):
       return ???
   ```

   What code do you need here? Hint: this method needs to make a *new* point, just like another method we already have.

6. Once you have implemented the `__add__` method correctly, try the following code:

   ```
   >>> import point
   >>> a = point.point(1, 2)
   >>> b = point.point(3, 4)
   >>> c = a + b
   >>> print(a, b, c)
   ```

   What is printed and why?

3 Basics of inheritance

As we mentioned, one of the most powerful ideas in OOP is *inheritance*. When we create a new class, we specify the class whose attributes it should inherit. This creates a hierarchy of classes, when each class inherits from another in a sort of parent/child relationship. The most “basic” class of all, the ancestor of almost all classes in Python, is called `object`.

In this part we’ll explore a little bit about how inheritance works in Python.

1. We’re going to define our own very, very simple class that inherits from the class `list`. All you need to do is type the following code into the shell:

   ```
   >>> class mylist(list):
           pass

   >>>
   ```

2. Now we’ll create one of our `mylist` objects by doing the following:

   ```
   >>> a = mylist([1, 2, 3])
   >>> b = list([1, 2, 3])
   >>> print(type(a), type(b))
   >>> a == b
   ```

   What is printed, and why?

3. Our `mylist` class now duplicates the existing class. We can use all of the `list` methods with our new class. Try this:

   ```
   >>> a.append(0)
   >>> print(a)
   >>> a.sort()
   >>> print(a)
   ```

   What is happening here?

4. We aren’t limited to just making an exact copy. We can actually *extend* the builtin list class. Suppose we wish we had a method that returned the product of the elements on a list. We can accomplish this as follows:

   ```
   >>> class mylist(list):
           def product(self):
               '''Compute the product of all elements in the list.'''
               r = 1
               for x in self:
                   r *= x
               return r

   >>>
   ```

   After typing this into the shell, try the following:

   ```
   >>> a = mylist([1, 2, 3, 4])
   >>> print(a.product())
   >>> b = list([1, 2, 3, 4])
   >>> print(b.product())
   ```

   What happens? Why does this only work in the once case, and not the other?

5. We can do the exact same kind of thing with a class we defined. Try the following:

   ```
   >>> from point import point
   >>> class mypoint(point):
           def distance(self, other):
               return abs(self.x - other.x) + abs(self.y - other.y)
   ```

   Essentially what we’ve done is replace the definition of the `distance` method in our original `point` class with a new version that only applies to *objects in the new class*. Replacing a method is called *overriding* a method.

```
>>> a = mypoint(1, 2)
>>> b = point(4, 6)
>>> print(a.distance(b))
>>> print(b.distance(a))
```

Why don’t these print the same answer?