6.00 Introduction to Computer Science and Programming
Fall 2008

6.00: Introduction to Computer Science and Programming

Problem Set 9: As If You Needed More Classes

Handed out: Thursday, October 30, 2008
Due: Monday, November 3, 2008

Introduction

In this problem set, you will be creating and using classes and methods. Along the way, you should become familiar with concepts such as inheritance and overriding methods.

Workload

Please let us know how long you spend on each problem. We want to be careful not to overload you by giving out problems that take longer than we anticipated.

Collaboration

You may work with other students. However, each student should write up and hand in his or her assignment separately. Be sure to indicate with whom you have worked. For further details, please review the collaboration policy as stated in the syllabus.

Getting Started

Add the code you write for this problem set to the following template: ps9.py

For problem 4, use the following sample input file: shapes.txt

Problem 1

In this problem set, we will be using classes to encapsulate shapes.

We have implemented two different shape subclasses for you, namely `Square` and `Circle`. For each of these shapes, we are interested in getting the area (note how the respective `area` methods override the `area` method of the `Shape` superclass), checking for equality, and specifying a string representation of the object.

Implement the `Triangle` class, which also extends `Shape`. `Triangle` should have the same methods as the other subclasses of `Shape`. Other things to note:

- A `Triangle` is initialized with a `base` and `height`.
- The string representation for a triangle with base 3.0 and height 4.0 should be:

```
Triangle with base 3.0 and height 4.0
```

Note: Remember that a double underscore (__) around a method name indicates that the method has special meaning to Python, even though it can be used and overridden like any other method. Python has some special conventions for calling such methods; for example, `len(x)` makes Python call `x.__len__()`. You can learn more about these special methods from the Python Language Reference.

Problem 2

It may be handy to work with a set of shapes. In this problem we define a class, `ShapeSet`, which manages a set of shape objects. Create a method, `addShape`, to add a shape to the set, with the stipulation that no two shapes in the set may be equal (that is, when being compared with the == operator). The `__iter__` method should return an iterator that allows you to iterate over the set of shapes, one shape at a time. You may assume that the ShapeSet is not mutated while any iterator is active. More information on iterator types can be found in the Python Library Reference.

Fill in the following code skeleton according to the specifications.

```
class ShapeSet:
    def __init__(self):
        """
        Initialize any needed variables
        """
        # TO DO
    def addShape(self, sh):
        """
        Add shape sh to the set; no two shapes in the set may
        be equal
        sh: shape to be added
        """
        # TO DO
    def __iter__(self):
        """
        Return an iterator that allows you to iterate over the
        set of shapes, one shape at a time
        """
        # TO DO
    def __str__(self):
        """
        Return the string representation for a set, which
        consists of the string representation of each shape,
        categorized by type.
        """
        # TO DO
```

The string representation of `ShapeSet` should include the string representation of each shape (each of which would be on its own line, if printed). Categorize the shapes in the set by their types.

If our set consisted of a square with side length 4, another square with side length 1, a circle with radius 2, and a triangle with base 1 and height 1, the string representation of this `ShapeSet` could be:

```
Circle with radius 2.0
Square with side 4.0
Square with side 1.0
Triangle with base 1.0 and height 1.0
```

Problem 3

Now that we have a working representation of individual shapes and groups of shapes, we can utilize these in outside code. For example, suppose we want to find the shape(s) with the largest area in a `ShapeSet`. This function will be outside of the classes we've defined.

Write a function `findLargest` that returns a tuple containing all items in a `ShapeSet` with the largest area.

```
    def findLargest(shapes):
        """
        Returns a tuple containing the elements of ShapeSet with the largest area.
        shapes: ShapeSet
        """
```

To test your code, try finding the largest shape in several different `ShapeSet`s. For example:

```
>>> ss = ShapeSet()
>>> ss.addShape(Triangle(1.2,2.5))
>>> ss.addShape(Circle(4))
>>> ss.addShape(Square(3.6))
>>> ss.addShape(Triangle(1.6,6.4))
>>> ss.addShape(Circle(2.2))
>>> largest = findLargest(ss)
>>> largest
(<__main__.Circle object at xxxxx>,)
>>> for e in largest: print e
Circle with radius 4.0
```

Note that `largest` is a tuple containing an instance of `Circle`, because that is the shape with the largest area in the set.

In some cases, there may be multiple shapes with the largest area. For example, a triangle with base 4 and height 6 had the same area as a triangle with base 3 and height 8. In this case, the tuple should contain both shapes.

```
>>> ss = ShapeSet()
>>> ss.addShape(Triangle(3,8))
>>> ss.addShape(Circle(1))
>>> ss.addShape(Triangle(4,6))
>>> largest = findLargest(ss)
>>> largest
(<__main__.Triangle object at xxxxx>, <__main__.Triangle object at xxxxx>)
>>> for e in largest: print e
Triangle with base 3.0 and height 8.0
Triangle with base 4.0 and height 6.0
```

You can check that the shape in the tuple is actually the shape you originally added to the `ShapeSet` by using the `is` keyword, which checks for object equality:

```
>>> t = Triangle(6,6)
>>> c = Circle(1)
>>> ss = ShapeSet()
>>> ss.addShape(t)
>>> ss.addShape(c)
>>> largest = findLargest(ss)
>>> largest
(<__main__.Triangle object at xxxxx>,)
>>> largest[0] is t
True
>>> largest[0] is c
False
```

Problem 4

Currently, to create a new `ShapeSet`, we are required to first create several individual shapes, then add them one by one to the `ShapeSet`. If we want to use a large number of shapes or if we want to reuse the same shapes from one program to the next, writing the code to create each can be tedious and redundant.

To solve this problem, we can store shape information in a file, then write a function to read the file line-by-line, creating a `ShapeSet` with the information provided.

Write a function `readShapesFromFile` that will create and return a `ShapeSet` based on the information provided in the given file.

```
def readShapesFromFile(filename):
    """
    Retrieves shape information from the given file.
    Creates and returns a ShapeSet with the shapes found.
    filename: string
    """
    # TO DO
```

The file information will be stored as follows:

```
circle,4
square,3
circle,2
triangle,4,4
square,4
circle,4
```

For a sample shape file, please see shapes.txt

Note that circles and squares have one argument (side length and radius, respectively), while triangles have two (base and height). You will need to find a way to deal with either case while you read the file.

For more information on how to read from a file, you may want to look at the code from ps8. The information on strings at the Python Library Reference may also be helpful.

You can test your code by printing out the `ShapeSet` returned; this will also help you to test your `ShapeSet` and other functions:

```
>>> ss = readShapesFromFile("shapes.txt")
>>> print ss
Circle with radius 4.3
Circle with radius 2.0
Circle with radius 4.5
Circle with radius 3.3
Circle with radius 6.0
Circle with radius 19.4
Square with side 3.0
Square with side 4.0
Square with side 16.1
Square with side 2.7
Square with side 10.9
Triangle with base 4.2 and height 4.0
Triangle with base 3.0 and height 6.6
Triangle with base 1.3 and height 4.0
Triangle with base 4.0 and height 1.0
Triangle with base 2.6 and height 1.2
```

Hand-In Procedure

1. **Save.** All your code should be in a single file called `ps9.py`.
2. **Time and collaboration info.** At the start of the file, in a comment, write down the number of hours (roughly) you spent on this problem set, and the names of whomever you collaborated with. For example:

```
# Problem Set 9
# Name: Jane Lee
# Collaborators: John Doe
# Time: 1:30
```

```
    ... your code goes here ...
```

3. **Sanity checks.** After you are done with the problem set, do these sanity checks:

    - Run the `ps9.py` file, and make sure it can be run without errors.
    - Inspect the `ShapeSet` created from parsing the file in Problem 4; does it match the specifications?