6.00 Introduction to Computer Science and Programming
Fall 2008

6.00: Introduction to Computer Science and Programming

Problem Set 10: OOP I did it again

Handed out: Thursday, November 6, 2008
Due: Thursday, November 13, 2008

Introduction

We hope you enjoyed the word game from problem sets 5 & 6 because it's back! In this problem set, you will fill in code for a graphical version of the 6.00 word game, with support for single player, two player, and vs. computer game modes.

Along the way, you will implement classes to encapsulate and manage the data and functions for the word game. You will also practice manipulating instances of these classes through methods. Finally, you will see your classes in action as they interact with a graphical user interface (GUI) module for the 6.00 word game.

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

Download and save these files into the same folder.

- ps10.py: The code template for the word game
- ps10_test.py: A skeleton test harness for the classes in ps10.py. You should add tests to this file.
- ps10_gui.py: The code for the word game graphical user interface. You should not need to modify this file.
- words.txt: The list of words from Problem sets 5 and 6

Problem 1: Install wxPython

Before you begin the problem set, you will need a toolkit for making a graphical user interface (GUI) in python. We will use the wxPython toolkit for this purpose.

First make sure you have Python 2.5.2. You can check what version of Python you have by going to the "Help" menu and clicking "About IDLE" from the Python Shell window in IDLE.

With Python 2.5.2 installed, you can download and install wxPython. Download information for wxPython can be found here.

Follow the instructions given in the installer. After the installation, start up IDLE and enter "import wx" at the prompt. If no error message appears, you have successfully installed the wxPython toolkit.

Install the wxPython toolkit. You do not have to submit anything for this problem.

Problem 2: Representing a Hand

In problem set 5, you managed a number of data structures and functions pertaining to a player's hand. The data for the hand was stored as a dictionary that mapped characters to their corresponding frequencies in the hand.

A number of functions were used to manage the hand—`deal_hand()` was used to initialize a new hand, `display_hand()`, was used to print the hand to the screen in a readable format, `update_hand()` was used to remove the letters in a word from the hand, and `is_valid_word()` used the hand to check if the letters necessary for making up a word were present.

We are now going to encapsulate the functions and data associated with a hand in a class called `Hand`. We have already provided the constructor method, `__init__()`, which takes as an argument the initial size of the hand (the `initialSize` parameter).

The constructor also takes an optional second argument (the `initialHandDict` parameter), which is a dictionary representation (mapping characters to frequencies) that is used to initialize the hand to a particular set of characters. This argument is used for testing purposes. If no second argument is given, the constructor initializes the hand with random characters.

The class stores the initial size of the hand as the attribute `self.initialSize` and the dictionary representation for the hand as the attribute `self.handDict`.

You will need to fill in the remaining functions according to the specifications to have a class representation of a hand.

Fill in the `update()`, `containsLetters()`, `isEmpty()`, and `__eq__()` methods in the `Hand` class according to the specifications. Your implementation for each of these methods should be relatively short (probably around 10–20 lines or fewer per method).

Feel free to make use of the solutions to problem set 5 and problem set 6 in your code. We have provided a skeleton test harness in `ps10_test.py`. Use `testHand()` to test your `Hand` class. You should add additional tests to `testHand()` to adequately test your `Hand` class.

```
class Hand(object):
    """
    A class representation of a hand.
    """
    .
    .
    .
    def update(self, word):
        """
        Remove letters in word from this hand.

        word: The word (a string) to remove from the hand
        postcondition: Letters in word are removed from this hand
        """
        # TODO

    def containsLetters(self, letters):
        """
        Test if this hand contains the characters required to make the input
        string (letters)

        returns: True if the hand contains the characters to make up letters,
        False otherwise
        """
        # TODO

    def isEmpty(self):
        """
        Test if there are any more letters left in this hand.
```

```
            returns: True if there are no letters remaining, False otherwise.
            """
            # TODO

        def __eq__(self, other):
            """
            Equality test, for testing purposes

            returns: True if this Hand contains the same number of each letter as
            the other Hand, False otherwise
            """
            # TODO
        .
        .
        .
```

Problem 3: Representing a Player

In problem sets 5 and 6, you initialized and managed the state of a player. You kept track of the player's score, their remaining time, and their current hand.

In this problem set, you will encapsulate this information in a `Player` class, except you are no longer keeping track of time. You will write a set of methods to access and modify this data. We have provided the constructor method, `__init__()`, for the `Player` class. The constructor takes as arguments an ID number (`idNum`) (either 1 for player 1 or 2 for player 2) and a `Hand` object (`hand`) for the player's hand and stores them as class attributes `self.idNum` and `self.hand`, respectively.

In this problem, you will fill in the remaining methods of the `Player` class according to the specifications to have a class representation of a `Player`.

Fill in the `getHand()`, `addPoints()`, `getPoints()`, `getIdNum()` and `__cmp__()` methods in the `Player` class. Again, your implementation for these methods should be very short.

We have provided a skeleton test harness in `ps10_test.py`. Use `testPlayer()` to test your `Player` class. You should add additional tests to `testPlayer()` to adequately test your `Player` class.

```
    class Player(object):
        """
        General class describing a player.
        Stores the player's ID number, hand, and score.
        """
        .
        .
        .
        def getHand(self):
            """
            Return this player's hand.

            returns: the Hand object associated with this player.
            """
            # TODO
        def addPoints(self, points):
            """
            Add points to this player's total score.

            points: the number of points to add to this player's score

            postcondition: this player's total score is increased by points
```

```
        """
        # TODO
    def getPoints(self):
        """
        Return this player's total score.

        returns: A float specifying this player's score
        """
        # TODO
    def getIdNum(self):
        """
        Return this player's ID number (either 1 for player 1 or
        2 for player 2).

        returns: An integer specifying this player's ID number.
        """
        # TODO
    def __cmp__(self, other):
        """
        Compare players by their scores.

        returns: 1 if this player's score is greater than other player's score,
        -1 if this player's score is less than other player's score, and 0 if
        they're equal.
        """
        # TODO
    .
    .
    .
```

Problem 4: Representing a Computer Player

A computer player can be thought of as a specialization of the `Player` class. Rather than implementing a class for the computer player from scratch, we will simply have the `ComputerPlayer` class inherit from the `Player` class and add the `pickBestWord()` method, which implements the computer player's algorithm for picking the best word, given its hand and the word list (a parameter of the method).

You should use the slow greedy algorithm that we implemented in problem set 6 (`pick_best_word`). The parameter `wordlist` is an instance of class Wordlist (which is provided). It will be useful to look at the implementation of the Wordlist class that is provided and make sure that you understand the methods available for a Wordlist instance.

Implement the `pickBestWord()` method in the `ComputerPlayer` class. As you reimplement the algorithm, if you are adapting code from previous problem set solutions, you must make the code compatible with the classes in this problem set. You should make use of the other classes to access information regarding the computer's hand and the available words in the word list.

We have provided a skeleton test harness in `ps10_test.py`. Use `testComputerPlayer()` to test your `ComputerPlayer` class. You should add additional tests to `testComputerPlayer()` to adequately test your `ComputerPlayer` class.

```
class ComputerPlayer(Player):
    """
    A computer player class.
    Does everything a Player does, but can also pick a word using the
    pickBestWord() method.
    """
    def pickBestWord(self, wordlist):
```

```
        """
        Pick the best word available to the computer player.

        returns: The best word (a string), given the computer player's hand and
        the wordlist
        """
        # TODO
    .
    .
    .
```

Problem 5: The Graphical User Interface

Having implemented and tested the `Hand`, `Player`, and `ComputerPlayer` classes, you should now be able to run the game using the graphical user interface.

When you start the GUI, you are presented with three options, corresponding to different game modes. Choosing one will show the main game play window and prompt you so you can prepare to start playing. In Solo mode, you are a single player playing through a hand. In Vs. Computer mode, you play first, then the computer player will play. In Vs. Human mode, you play first, then your friend plays. In the Vs. modes, both players are dealt the same initial hand, so make sure your friend isn't watching while you play!

In the play window, you can enter your words in the text box and either hit Enter or press the Submit button. The status bar along the bottom will display messages concerning whether your word was accepted, and accepted words will be added to the word history list box. On the left and right you will see the stats displays for players 1 and 2, respectively. To end a hand, use up all your characters or enter a '.'. You may also close the window at any time to stop the game.

When playing against the computer, you won't be able to interact with the main window, but you can see the words that the computer player chooses as it's playing.

Load `ps10_gui.py` and run it. Test the different game modes and make sure your code works for all of them. You do not have to submit anything for this problem, but be sure to test your code thoroughly.

Hand-In Procedure

1. **Save.** Your modified code should be in a file called `ps10.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 10
# 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 `ps10_gui.py` and `ps10_test.py` files, and make sure they can be run without errors.