CS 1110 Final Exam Solutions May 15th, 2014

The Important First Question:

1. [2 points] When allowed to begin, write your last name, first name, and Cornell NetID at the top of each page. Solution: Always do this! It prevents disaster in cases where a staple fails.

2. [4 points] **Objects.** Consider the following code (docstrings omitted for exam brevity, line numbers added for reference).

```
 1  class Prof(object):
 2      def __init__(self, n):
 3          self.lname = n
 4
 5  ljl2 = Prof("Schlee")
 6  srm2 = Prof("Schmarschner")
 7
 8  lecturingprof = srm2
 9  lecturingprof.wise = True
10  lecturingprof = ljl2
11  print "Is Prof " + srm2.lname + " wise? " + str(srm2.wise)
12  print "Is Prof " + ljl2.lname + " wise? " + str(ljl2.wise)
```

List all output and/or errors that would be generated by running this code, in the order they are produced. For each thing you write, indicate what line number produces it.

In the case of errors, it suffices to explain what the problem is — you don't know have to know precisely what Python would call the error or print out.

Hint: line 9 does *not* cause an error. It would be wise (ha!) to understand why before proceeding; what does Python always do when asked to assign to a variable that doesn't exist?

Solution:

```
line 11: Is Prof Schmarschner wise? True
Traceback (most recent call last):
  File "/Users/llee/classes/cs1110/repo/Exams/final/alias.py", line 12, in <module>
    print "Is Prof " + ljl2.lname + " wise? " + str(ljl2.wise)
AttributeError: 'Prof' object has no attribute 'wise'
```

+2: right output for line 11. This requires understanding string concatenation and that srm2.lname should be "Schmarshcner" (1 point), and understanding that str(srm2.wise) would be "True"

+2: A description of the right error for line 12. Does not have to have the exact name of the error, but has to recognize that the object referred to by ljl2 does not have a "wise" attribute. No credit for answer saying that ljl2 does not have an lname attribute.

3. [10 points] **String processing, loops.** We say that a string is a *sentence* if it consists of "words" (non-empty sequences of non-space characters) separated by single spaces, with no spaces at the beginning or end of the string. A sentence is *chunked* by *delimiter* `dl` if an even number of its words are `dl`, and no two delimiters are consecutive. Here's an example of a sentence that is chunked by "!".

   "The ! Big Red Barn ! was ! blue !"

   The *interesting spans* of a chunked sentence are the sequences of words that appear between each *odd* occurrence of `dl` and the next occurrence of `dl`. So, "Big Red Barn" *is* an interesting span because it occurs between the 1st and 2nd "!". "was" is *not* an interesting span because it occurs after the 2nd "!" (and before the 3rd one).

   The *highlighted* version of a chunked sentence is one where the delimiters have been removed, every word in an interesting span has been capitalized, and every word not in an interesting span is in lowercase. For example, the highlighted version of the chunked sentence above is

   "the BIG RED BARN was BLUE"

   Implement the function below so it fulfills its specification.

   Hints (not requirements): Use `split` and/or `join` (see reference on page 2). Use a loop, but do *not* use a nested loop. Keep track of whether you're inside or outside an interesting span.

```
def highlight(input, dl):
    """Return: the highlighted version of the input.
    Pre: input: a sentence chunked by dl.  dl: non-empty string without spaces."""
```

Solution:

```
    inside = False
    output = []
    for word in input.split():
        if word == dl:
            inside = not inside
        else:
            output.append(word.upper() if inside else word.lower())
    return ' '.join(output)
## +1 iterating over the string, either as a list version or the original one
##   (if the original, hard to see how they'd get the next points)
## (-1 for no increment if doing a while loop)
## +1 not inside and not dl means lowercase
##     NOTE: printed exam had typo, so "leave alone" instead of lowercase OK
## +2 not inside and dl means start a span, don't print the dl
## +1 inside and not dl means uppercase
## +2 inside and dl means end a span, don't print the dl
## +1 right syntax for calling upper and lower
## +2 right output (could be keeping a string or a list all along)
##    (-1 if don't get the spacing right (e.g. double spaces or
##     extra spaces at the beginning or end); -1 if a list;
##     -1 if use join but incorrectly)
```

There is an alternate solution that splits on `dl`, but note that you have to be a little careful about handling spaces:

```
def highlight2(input, dl):
    spans = input.split(dl)
    for i in range(len(spans)):
        spans[i] = spans[i].strip()
        if i % 2 == 1:
            spans[i] = spans[i].upper()
        else:
            spans[i] = spans[i].lower()
    return ' '.join(spans).strip()
```

4. [12 points] **Recursion.** We say that an input `input` is *well-formatted* with respect to a list `labels` if (a) `input` is a list, and (b) `input` has length at least two, and (c) `input`'s first item is in the list `labels`, and (d) each of the remaining items in `input` is either a string or a well-formatted list. Here are some examples of well-formatted and non-well-formatted inputs:

`input`	`labels`	well-formatted?
`['VP', ['V', 'eat']]`	`['VP', 'V']`	True
`['NP', ['N', 'a', 'or', 'b'], 'c']`	`['NP', 'V', 'N']`	True
`[1, [2, 'oui', [1, 'no']], 'no']`	[1,2]	True
`['VP', ['V', 'eat']]`	`['VP']`	False: 'V' not in `labels`
`['VP', ['V']]`	`['VP', 'V']`	False: list `['V']` too short
`'VP'`	`['VP', 'V']`	False: `input` is not a list

Implement the following function recursively according to its specification.

```
def wellformatted(input, labels):
    """Returns: True if <input> is well-formatted with respect to <labels>,
    False otherwise.

    Pre: labels is a (possibly empty) list.
    """
```

Solution:

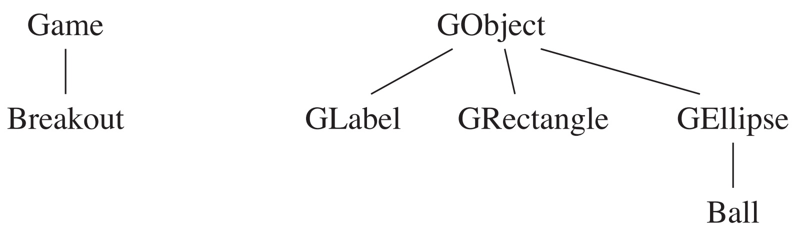
```
    if type(input) != list or len(input) < 2 or (input[0] not in labels):
        return False
    ## +1 nonlist base case
    ## +1 too short base case
    ## +1 check of input[0] not in labels
    ## +1 order correct (if you do len(input) before checking if list, could get error)
    ## +1 these should be ors, not ands. OK if implemented via if-elif cascase.
    else:
        for item in input[1:]:
            if type(item) != str and not wellformatted(item, labels):
                return False
        return True
    ## +1 loop through input starting at 1 *
    ## +1 check for type not being string *
    ## +2 (all or none): correct recursive call *
    ## +1 conjunction check (not disjunction) *
    ## +1 return False if one check fails *
    ## +1 return True if all checks pass

    ## A solution that uses tail recursion (i.e. checks the base case, then
    ##   makes a recursive call on input[1:]) instead of making recursive
    ##   calls on the items of the list, loses 6 points (all the points
    ##   marked '*' above.)
```

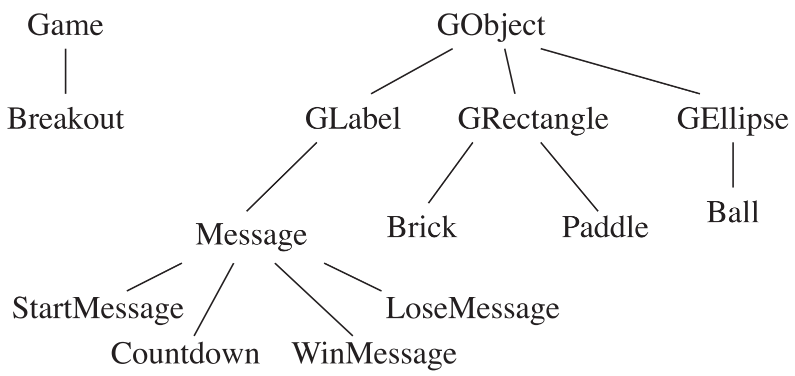
5. **Subclasses.**

Consider the accompanying code, which shows part of a solution to A5 that has a curious way of handling the flow of states in the game. Many details are omitted, but all the code related to adding and removing objects from the game is preserved.

(a) [3 points] Complete the class hierarchy below showing the relationships between all the classes in this code.



Solution:



(b) [3 points] Remember that when an object is created, Python calls the method `__init__` on that object automatically. The same name resolution process is followed as with any method call.

Now, to the question: line 45 executes only once. *During the execution of that line*, line 79 gets executed. At that point in the execution (i.e. when line 79 is executed),

(i) What is the class of the object referenced by `self`?
Solution: StartMessage

(ii) Where does the variable named `self.TEXT` reside? We want to know in which specific class, instance, frame, or module it is found—that is, where it would be drawn when diagramming the execution.
Solution: in the class StartMessage

(iii) At what line was the variable created?
Solution: 89

(c) [4 points] Remember that the "call stack" is the set of frames that exist at a particular time. For instance, during a game when the player loses, line 201 executes exactly once, and at the start of executing that line, the call stack is:

```
...
Breakout.update: 51
Ball.update: 201
```

Here we are including just the first line from each frame, indicating which method is executing and which line it is at. The frames appear in the order they were created. (Note that since there are multiple functions with the same name, it's important to include the class they are defined in.)

The variable `Brick.num_bricks` is mentioned in the code only at lines 150, 154, 165, and 166. During a game that is won, it is assigned the value 3 exactly twice. At the first time it gets that value, what is the call stack? Use a format like the example above, and only mention functions that are defined in `breakout.py`.

Solution:

```
...
Breakout.update: 51         # +1
StartMessage.update: 100    # +1
Brick.__init__: 154         # +1
# and +1 for not having anything else in there
# -1 for being in the wrong order
```

(d) [12 points] Complete the subclass Countdown of Message that shows the message "Get ready!" and then, 90 frames after it was created and added to the game, removes itself from the game and "serves" by adding a new Ball instance to the game. The constants `TEXT` and `DELAY` should determine the message and the time delay before serving the ball. Be sure your code adheres to the provided class invariant. (To save time on the exam, there's no need to write specifications.)

```
class Countdown(Message):
    """See spec in code handout, line 106"""

    TEXT = 'Get ready!'
    DELAY = 90
```

Solution:

```
    # +1 for defining an initializer at all *
    def __init__(self):
        Message.__init__(self)
        self.frames_left = self.DELAY
        # +2 for call to superclass initializer (1 for try, 1 for right) *
        # +1 for initializing frames_left
        # +1 for correct use of self and/or Countdown to access DELAY
        # +1 for giving frames_left the right name and using DELAY
        # (-2 for directly using DELAY as the counter)

    # +1 for defining update at all (OK to forget dt) *
    def update(self, dt):
        self.frames_left -= 1
        if self.frames_left == 0:
            Breakout.the_game.add_game_object(Ball())
            Breakout.the_game.remove_game_object(self)
        # +1 for decrement
        # +1 for testing against zero (off by one OK)
        # +1 for accessing Breakout.the_game correctly
        # +1 for creating a new ball and adding it
        # +1 for removing self

    # -1 for every two minor syntax issues (missing colons, etc.)
    # Solutions that did not define any methods at all but contained
    # much of the correct code were usually around 6/12, losing the points
    # indicated by '*' and an additional 2 for using the undefined
    # name 'self'.
```

6. [10 points] **Invariants.** Suppose we are given the task of rearranging a string so that certain characters are moved to the beginning and the other characters are moved to the end. Implement this method to the specification given below, following the comments in the code.

```
def partition_string1(s1, s2):
    """Return: a string that has the same characters as s1, only reordered
    so that all the characters that appear in s2 are at the beginning, and
    all the characters that do not appear in s2 are at the end.  The ordering
    of the characters within each of the two segments is not important.

    Examples:
        s1            ; s2    ; some correct results
        ---------------------------------------------------------
        'abracadabra' ; 'abc' ; 'abacaabardr', 'aaaabbcrdr', ...
        'foobar'      ; 'ob'  ; 'boofar', 'oboarf', ...
        'foobar'      ; ''    ; 'foobar', 'oofarb', ...
        'a'           ; 'b'   ; 'a'
    """

    # This function works by converting the input to a list and rearranging
    # the list in place using swaps, following the invariant below.  Your
    # code must agree with the invariant and postcondition for full credit.

    # convert to a list b


    # initialize counters



    # inv: b[0..i-1] in s2; b[j+1..len(b)-1] not in s2

    while





    # post: b[0..j] in s2; b[j+1..len(b)-1] not in s2
    # convert b back to a string and return
```

Solution:

```
# convert to a list b
b = list(s1)          # +1

# initialize counters
i = 0                 # +1
j = len(b) - 1        # +1

# inv: b[0..i-1] in s2; b[j+1..n-1] not in s2
while j != i - 1:     # +2 for correct condition (*)
    if b[i] in s2:    # +1 checking correctly
        i += 1        # +1 for valid update in no-swap case
    else:
        b[i], b[j] = b[j], b[i]   # +1 for correct swap
        j -= 1                    # +1 for correct increment
# post: b[0..j] in s2; b[i+1..n-1] not in s2

# convert b back to a string and return
return ''.join(b)     # +1

# (*) various equivalent conditions are possible including
# i != j + 1, i < j + 1, i - 1 < j, etc.

# Note that "while j != i" or "while i < j" also produces
# correct return values but *does not* truthify the postcondition
# so -1.
```