Last Name: _____ First Name: _____ Cornell NetID, all caps: _____

CS 1110 Prelim 2 Solutions April 2017

1. [10 points] **For-loops**. Implement the following specification. Your implementation must make effective use of a for-loop.

```
def followers(wordlist, starter):
    """Returns a list of all words that immediately follow starter in wordlist,
        in the order they appear in wordlist.
        (Returns the empty list if there aren't any).
        Does not alter wordlist.

    Preconditions:
       wordlist is a list of nonempty strings, none of which contain spaces
          wordlist might be itself empty
       starter is a nonempty string with no spaces

    Example: if wordlist is ["a", "man", "a", "plan", "a"],
        if starter is "a", returns ["man", "plan"]
        if starter is "flower", returns [] """
    ### Your implementation must make effective use of a for-loop


    result = []
    for s in wordlist:
        result.append(wordlist[wordlist.index(starter)+1])
        wordlist = wordlist[wordlist.index(starter):]
    result
```

Solution:
Must be careful about "falling off the end" of the list.

```
    output = []
    for i in range(len(wordlist)-1):
        if wordlist[i] == starter:
            output.append(wordlist[i+1])
    return output
```

Alternate solution:

```
    output = []
    for i in range(len(wordlist)):
        if i+1 < len(wordlist) and wordlist[i] == starter:
        # or, if i < len(wordlist) - 1 ...
            output.append(wordlist[i+1])
    return output
```

The following IS INCORRECT:

```
output = []
for w in wordlist[:len(wordlist)-1]:
    if w == starter:
        i = wordlist.index(w)  #### THIS ALWAYS USES THE LEFTMOST FOLLOWER
        output.append(wordlist[i+1])
return output
```

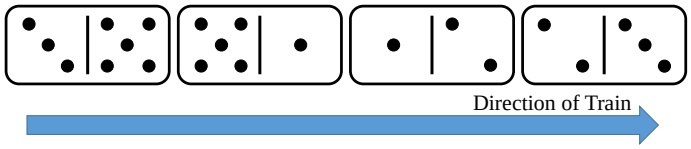
2. [16 points] **Recursion.** Make effective use of recursion to implement the following new method, **games**, for class Outcome. Some relevant specifications are given on the next page.

```
class Outcome(object):
    """Class invariant given on next page, for space reasons."""

    def games(self, team):
        """Returns an int representing the number of games that team played in
        this Outcome and all its sub-Outcomes.

        Precondition: team is a non-empty string.

        For examples of output for games(), see the next page."""


        ### You may NOT use the teams() method from A4.
        ### You MUST use _extract_name() --- specification on the next page.

        ### Look at the examples for games() on the next page before you start.
```

Solution:

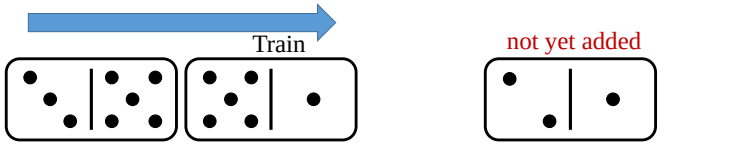
```
g0 = 0 # number of times team has played in the top-level Outcome
if team in [_extract_name(self.input1), _extract_name(self.input2)]:
    g0 = 1

g1 = 0 # number of times team has played in first subOutcome
if isinstance(self.input1, Outcome):
    g1 = self.input1.games(team)

g2 = 0 # number of times team has played in second subOutcome
if isinstance(self.input2, Outcome):
    g2 = self.input2.games(team)

return g0 + g1 + g2
```

3. [24 points] **Classes**. Dominoes are rectangular tiles with one number on each side. There are multiple games in which players place dominoes on a table in a row called a "train". A domino can be placed in a train next to another domino if their adjacent numbers match:



There is a skeleton of a Domino class below and on the following pages. Complete this class by adding code under each function specification.

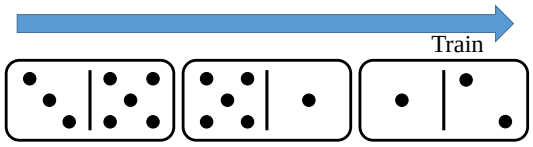
When you have finished, `addDomino` should have the following behavior. Before adding a domino [2 | 1], the train might look like this:



side1 = 5
side2 = 1
prior = 
next = None
outwardFacingSide = 1

side1 = 2
side2 = 1
prior = None
next = None
outwardFacingSide = -1

After [2 | 1] has been added (and flipped), the train will look like this:



side1 = 5
side2 = 1
prior = 
next = 
outwardFacingSide = 1

side1 = 2
side2 = 1
prior = 
next = None
outwardFacingSide = 2

Note that `side1` and `side2` for [2 | 1] have not changed.

```
class Domino(object):
    """ An instance represents a Domino game piece

    Attributes:
        side1: the value on side 1 of the domino piece [int 1..6]
        side2: the value on side 2 of the domino piece [int 1..6]
        prior: the domino immediately preceeding this domino [Domino or None]
        next: the domino that follows this domino [Domino or None]
        outwardFacingSide: the value on the side that is facing out,
            or -1 if unattached [int -1 or 1..6] """
```

```
def __init__(self, n1, n2):
    """ Makes new domino have value n1 on side 1 and n2 on side2
    and not attached to any domino train (prior and next are None).
    outwardFacingSide should default to -1.

    Precondition: n1, n2 are integers in [1..6]"""
```

Solution:

```
    self.side1 = n1
    self.side2 = n2
    self.prior = None
    self.next = None
    self.outwardFacingSide = -1
```

```
def __str__(self):
    """Returns: The string representation of this Domino
    in the form: "Domino: side1|side2". For example:
    if side1 is 5, side2 is 6, this returns "Domino: 5|6"
    Note: | is a key on your keyboard; just draw a vertical bar"""
```

Solution:

```
    text = "Domino " + str(self.side1) + "|" + str(self.side2)
```

```
def canExtend(self):
    """Returns: True if no domino follows this one in a train,
    False otherwise."""
```

Solution:

```
    return self.next is None
```

```
def addDomino(self, d):
    """Returns: True if domino d can be added to the train ending at
    the current domino, and False otherwise. If d can be added, this function
    sets this domino's next to be d, sets the prior of d to be this
    domino, and updates the outwardFacingSide attribute of d.

    A domino d can be added to the current domino if: 1) d has side1 or side2
    equal to this domino's outwardFacingSide value, 2) the domino on which this
    is being called is the end of a train (canExtend returns True for this domino),
    and 3) the prior of d is None.

    Precondition: d is a Domino """
```

Solution:

```
    if not self.canExtend() or not d.input is None:
        return False
    if d.side1 == self.outwardFacingSide:
        d.prior = self
        self.next = d
        d.outwardFacingSide = d.side2
        return True
    elif d.side2 == self.outwardFacingSide:
        d.prior = self
        self.next = d
        d.outwardFacingSide = d.side1
        return True
    else:
        return False
```

Last Name: _____ First Name: _____ Cornell NetID: _____

4. [18 points] **Name resolution and inheritance**.

```
class A(object):
    c = 3

    def f(self):
        self.c = 5
        return 10

    def g(self):
        return self.f()

class B(A):

    def f(self):
        c = 4
        return 14


a = A()
b = B()

print a.g()
print b.g()
print a.c
print b.c
```

Put your object and class folder diagrams here (do *not* draw any function frames):

We guarantee that no errors result from running the code above.

1. **In the space above**, draw the object and class folders that result by running this code. You should include method names and class variables in the class folders, and your object folders must have the type label in the upper right corner.

   You only need to draw two class folders, in addition to any folders for objects.

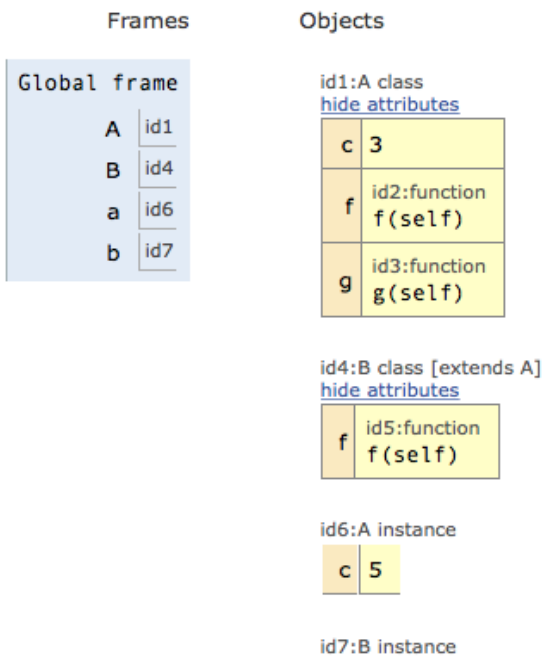
   Do *not* draw any frames for function calls.

   Remember that class folders have their tab on the right-hand side, whereas object folders have their tab on the left-hand side.

2. **In the space below**, write down what will get printed by each of the four print statements when they are executed in the order shown. (No credit for the print-statement output if you do not provide the diagrams requested above.)

Solution:

Last Name: _____ First Name: _____ Cornell NetID: _____




```
10   # the f in A; also changes the c in object a
14   # b.g is the g in A, which calls f which refers to B's f
5    # the c in object a was changed two lines above
3    # the c for object b is not in b, or in B, but in A
```

5. [5 points] **Loop correctness/invariants.** Here is a specification:

```
def first_below(thelist, limit):
  """Returns: index of leftmost item in thelist that has value less than
     limit.  (Returns -1 if no such item exists in thelist.)

     Precondition: thelist is a possibly empty list of ints. limit is an int.

   Example input/output pairs:

   first_below([4, 2, 5, -2], 3) -->  1     first_below([4, 10, 5,-2], 3) -->  3
   first_below([4,  2, 5,-2], 5) -->  0     first_below([4, -2, 5, 2],-3) --> -1
   first_below([4, -2, 5, 2],-2) --> -1     first_below([],5) --> -1 """
```

Below are two attempted implementations. **Exactly one** is correct; the other one fails its test cases, and one potential issue is that it fails to initialize or maintain its invariant.

Your job: correct the wrong implementation so that it agrees with the written invariant and thereby works correctly. Do so by changing **exactly one line of (non-commented) code**: **circle the offending line and then write the correct version next to it.**

```
i = 0
# INVARIANT: thelist[0..i-1] are all >= limit. So i is next place to check
while i < len(thelist) and thelist[i] >= limit:
    i = i + 1
if i < len(thelist):
    return i
else:
    return -1
```

```
j = 0
# INVARIANT: thelist[0..j] are all >= limit. So j+1 is next place to check
while j+1 < len(thelist) and thelist[j+1] >= limit:
    j = j + 1
if j+1 == len(thelist):
    return -1
else:
    return j+1
```

Solution:
Second version starts j off wrong; it should be j = -1, not j = 0

6. [1 point] **Fill in 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.