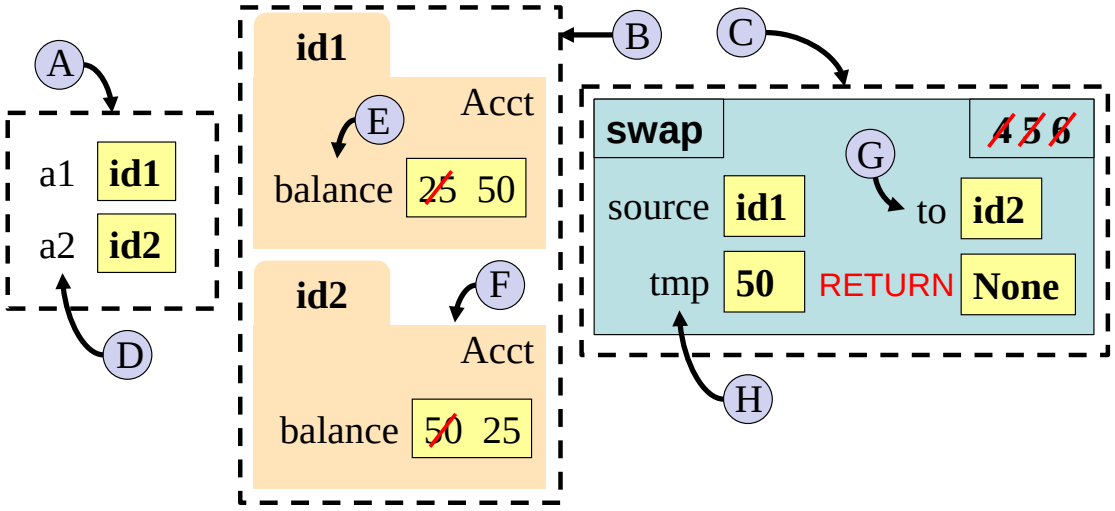
Last Name: _____ First Name: _____ Cornell NetID, all caps: _____

CS 1110 Regular Prelim 1 Solutions March 2017

1. **Object Diagramming and Terminology.**

   (a) [8 points] Suppose there is a class Acct defined in file `a2.py` where Acct objects have a `balance` attribute. Consider the following code and associated memory diagrams. Eight items in the diagram have been labeled (A) through (H). Match the vocabulary terms below the diagram with a labeled example by writing a letter next to each term.

```
1  from a2 import *
2  def swap(source, to):
3      """Swap balance of Acct <source> and balance of Acct <to>"""
4      tmp = source.balance
5      source.balance = to.balance
6      to.balance = tmp
7
8  a1 = Acct(50)
9  a2 = Acct(25)
10 swap(a1,a2)
```



Local Variable: _____ Global Variable: _____ Parameter: _____ Attribute: _____

Object: _____ Call Frame: _____ Heap Space: _____ Global Space: _____

Solution: H, D, G, E
F, C, B, A

(b) [9 points] Consider the following 6 lines entered in Python interactive mode. Diagram all variables and objects created at the end of execution. Do *not* diagram any call frames.

```
>>> x = 1
>>> y = x
>>> x = 2
>>> p = [1,2]
>>> q = p
>>> p[1] = 5
```

Solution: Here is a "typeable" description of the solution. An arrow ("->") means a change in value.

```
x: 1->2          y: 1
p: id1      q: id1
id1:  list object containing 1, 2->5
```

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

2. **String processing, testing.**

In the US, 10-digit telephone numbers are typically represented in one of the two following styles:

"Parenthetical": (555) 666-1110
"Dashed": 555-666-1110

There is no whitespace in a Dashed phone number: they are all exactly 12 characters long.

There is only one space in a Parenthetical phone number, and it is after the ")"; they are all exactly 14 characters long, counting the space.

(a) [10 points] Implement the following function according to its specification.

```
def phone_to_paren(s):
    """ Returns: a string representing the phone number s in Parenthetical form.

    Precondition: s is a non-empty string that *would* be a valid Dashed phone number
        EXCEPT that it possibly has spaces around the dashes.

    Examples:  '555-666-1110'
               '555 -  666 -  1110'
               '555  -   666-1110'
              ... all yield the same result, '(555) 666-1110' """
```

Solution:

One solution:

```
idash1 = s.index('-')
idash2 = s.rindex('-')
areacode = s[:idash1].strip()
second = s[idash1+1:idash2].strip()
third = s[idash2+1:].strip()
return '(' + areacode + ') ' + second + '-' + third
```

Another solution:

```
parts = s.split('-')
return '(' + parts[0].strip() + ') ' + parts[1].strip() + '-' + parts[2].strip()
```

A one-line solution:

```
return '(' + '-'.join(map(str.strip, s.split('-'))).replace('-', ') ', 1)
```

The above uses `replace` with three arguments: `somestring.replace('-', ')', 1)` replaces just the first '-' in a string `somestring` with a close parenthesis ')'.

(b) [6 points] Consider the following function specification.

```
def area_code(s):
    """ Returns: an int representing the area code (first three digits)
    of a telephone number in string s.

    Precondition: s is a non-empty string that *would* be a valid Dashed or
    Parenthetical phone number, EXCEPT it possibly has spaces around the dashes."""
```

Write **three conceptually distinct test cases** for this function in the table below.

Solution: `"(555) 666-1110"` is a valid Parenthetical. `"(555) 666 -  1110"` is a would-be Parenthetical except with spaces around the dashes. `"555  - 666  - 1110"` is a would-be Dashed except with spaces around the dashes. `"555-666-1110"` is a valid Dashed.

Inputs that violated the preconditions were not valid solutions.

Test case #1

Input and expected output:

Rationale:

Test case #2

Input and expected output:

Rationale:

Test case #3

Input and expected output:

Rationale:

3. [10 points] **Time for Objects!** The Time class is defined so that Time objects have two attributes, `hours` and `minutes`.

   Write the body for the function below so that it implements its specification.

   You may assume it is being defined in the same file as the definition for Time, so that you can create a new Time object with, say, 1 hour 30 minutes via the call `Time(1,30)`.

```
def mirror_time(t):
    """Modifies Time object t to be its "mirror image" on the clock.
    Does NOT create a new object; does not return a value.

    To get a "mirror image" time:
        1. If the hours are 12, then the hours should stay the same.
        Otherwise, take 12 and subtract the hours to get the new hours.
        2. If the minutes are 0, then the minutes should stay the same.
        Otherwise, take 60 and subtract the minutes to get the new minutes.
    Examples: 10:10 -> 2:50
             12:00 -> 12:00
              4:30 -> 8:30
    Precondition: t is a Time object with hours <= 12, minutes <= 59 """.
```

Solution:

```
if t.hours != 12:
    t.hours = 12 - t.hours
if t.minutes != 0:
    t.minutes = 60 - t.minutes
```

4. [16 points] **Banking on objects.** Suppose you are working on a file that defines two classes:

   - Acct: Acct objects have a float attribute, `balance`
   - Person: Person objects have two attributes:
     - `partner`, which is either a Person object or `None`
     - `bank_acct`, which is an Acct object.

   Inside the same file that defines the classes Acct and Person is also the following function header and specification. Write the body of the function so that it implements its specification.

```
def groupem(p1, p2):
    """ 1. Creates a new Acct whose balance is the sum of the balances of
        p1 and p2's bank_acct objects;

        2. Changes the balances of both p1's bank_acct and p2's bank_acct to 0

        3. Changes both p1's bank_acct and p2's bank_acct to the new Acct

        4. Makes the partner of p1 be p2 and the partner of p2 be p1.

        Preconditions: p1 and p2 are Persons whose partners are both None."""

    # hint: if p1 is a Person, p1.bank_acct is an Acct. So you can write
    #       (p1.bank_acct).balance, or even p1.bank_acct.balance
```

Solution:

```
    # Making some aliases to reduce typing
    old_a1 = p1.bank_acct
    old_a2 = p2.bank_acct
    new_acct = Acct(old_a1.balance + old_a2.balance)

    old_a1.balance = 0.0
    old_a2.balance = 0.0

    p1.bank_acct = new_acct
    p2.bank_acct = new_acct

    p1.partner = p2
    p2.partner = p1
```

5. [7 points] **The import of import.** Suppose file andersen.py defines a function `is_the_one` that takes a string as input, performs some computation, and returns a Boolean.

   And, suppose file lee.py also defines a function `is_the_one` that takes a string as input, but performs some *possibly different* computation, and returns a Boolean.

   Finally, suppose you are writing code in a third file matrix.py, which is currently empty.

   Write code to be placed in the third file that:

   - stores in variable `neo` the result of calling the andersen.py version of `is_the_one` on the string `"Thomas"`; and
   - stores in variable `oracle` the result of calling the lee.py version of `is_the_one` on the string `"Thomas"`

   Solution: Multiple solutions are possible, but many close variants are incorrect.

   Most straightforward version:

```
import andersen
import lee
neo = andersen.is_the_one("Thomas")
oracle = lee.is_the_one("Thomas")
```

   Fancy version using keyword `as` (which we didn't talk about in lecture)

```
from andersen import is_the_one as f1
from lee import is_the_one as f2
neo = f1("Thomas")
oracle = f2("Thomas")
```

   Interleaved version — NOT recommended, but legal:

```
from andersen import *
neo = is_the_one("Thomas")
from lee import *
oracle = is_the_one("Thomas")
```

6. **For-loop analysis**. Consider the following function header and specification:

```
def vowel_in_common(s1,s2):
    """Let the vowels be defined as a, e, i, o, u.

       If there is some vowel v contained in both s1 and s2, return a list
       of the index of the first v in s1 and the index of the first v in s2.

       If there is more than one such vowel v, v = the alphabetically first one.
       If s1 and s2 have no vowel in common, return the list [-1, -1].

    Preconditions: s1 and s2 are non-empty strings.

    Examples:
       vowel_in_common("brad", "angelina") -> [2, 0]
       vowel_in_common("romeo", "romeo") -> [3, 3]
       vowel_in_common("dan", "phil") -> [-1, -1]
    """
```

Here is one proposed implementation of `vowel_in_common`. It may or may not be correct.

```
    for v in ['a', 'e', 'i', 'o', 'u']:
        i1 = s1.find(v)
        i2 = s2.find(v)
        if i1 != -1 and i2 != -1:
            return [i1, i2]
        else:
            return [-1, -1]
```

(a) [6 points] Suppose someone makes the call

```
indices = vowel_in_common("brad", "angelina"),
```

What is the value of local variable `v` just before the function returns? Solution: "a"

What is the return value of the call? Solution: [2, 0]

(b) [4 points] Suppose someone makes the call

```
indices = vowel_in_common("romeo", "romeo"),
```

What is the value of local variable `v` just before the function returns? Solution: "a"

What is the return value of the call? Solution: [-1, -1]

So, this loop is actually incorrect.

(c) [7 points] Here is an alternative proposed implementation of `vowel_in_common`. It may or may not be correct.

```
found = "" # first vowel found in both
for v in ['a', 'e', 'i', 'o', 'u']:
    if v in s1 and v in s2 and found == "":
        found = v

if v == "":
    return [-1, -1]
else:
    return [s1.index(v), s2.index(v)]
```

Suppose someone makes the call

`indices = vowel_in_common("brad", "angelina")`,

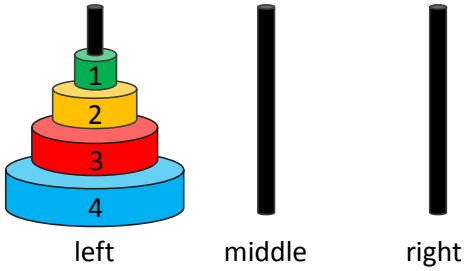
For the alternative implementation, what is the value of local variable `v` just before the function returns? Solution: "u" (loop goes to last list item)

What is the value of local variable `found` just before the function returns? Solution: "a"

What is the return value of the call? Solution: an error (an index error, although students don't need to specify this: no "u" in `s1` (or `s2`).

So, this loop is also incorrect.

7. [10 points] **Lists**. The Towers of Hanoi is a famous math puzzle that involves moving circular disks (with a hole in the middle) from one tower to another. There are three towers: "left", "middle", and "right", and each disk has a unique size. The goal is to move all of the disks from the left tower to the right tower without putting a larger disk on top of a smaller disk.



The towers can be represented in Python as lists of integers. Each disk has its own unique associated integer. The first element in the list represents the bottom of the tower, and the last element represents the top. For example, the above setup could be represented as:

```
left = [4, 3, 2, 1]
middle = []
right = []
```

Below is the specification of a function `move` to move disks (integers) from one list to another. For example, after executing the following code:

```
left = [4, 3, 2, 1]
middle = []
move(left, middle)
```

`left` should contain `[4, 3, 2]` and `middle` should contain `[1]`.

```
def move(from_tower, to_tower):
    '''Tries to move a disk from from_tower to to_tower. More specifically,

    * if to_tower is empty, or if to_tower is not empty and its last element is
    larger than the last element of from_tower, then removes the integer at the
    end of from_tower and appends it to the end of to_tower.

    * otherwise, does nothing.

    Precondition: from_tower and to_tower are lists of integers representing
    disks. from_tower has at least one disk. to_tower may be an empty list.
    Procedure, not a fruitful function, so no return.'''
```

Implement this function on the next page.

```
# Put your code for function move below.
```

Solution:

```
i_from = len(from_tower)-1 # index of top of from_tower
i_to = len(to_tower)-1 # index of top of to_tower UNLESS to_tower is empty

# Tricky issue: don't want to check the last item in to_tower if
#   to_tower is empty!
if to_tower == [] or \
  to_tower[i_to] > from_tower[i_from]: # Short circuit eval means do not
                                       # get here unless to_tower != []
    to_tower.append(from_tower.pop(i_from))
```

8. [10 points] **String Processing.** Write this function's body so that it implements its specification.

```
def glue(name1, stop1, name2, start2):
    """Returns:
        -1 if stop1 is not a valid index for name1 or if
                  start2 is not a valid index for name2
        Otherwise, returns a new string formed by concatenating:
           the substring of name1 starting at index 0 and ending at index stop1
           with
           the substring of name2 starting at index start2 and going to the end

    Preconditions:
        name1 and name2 are non-empty strings of lowercase letters
        stop1 and start2 are nonnegative ints

    Examples:
           glue("jules", 1, "vincent", 3) -> "jucent"
           glue("jules", 4, "vincent", 4) -> "julesent"
           glue("jules", 2, "vincent", 0) -> "julvincent",
           glue("jules", 5, "vincent", 4) -> -1
           glue("jules", 1, "vincent", 100) -> -1
    """
```

Solution:

```
if stop1 >= len(name1) or start2 >= len(name2):
    return -1
else:
    return name1[0:stop1+1] + name2[start2:]
```

Alternative solution using try/except:

```
try:
    # Trick: just throw IndexError if stop1 or start2 aren't good indices;
    # storing in a garbage variable for student readability
    # We need this line because string slicing accepts invalid indices.
    just_index_check = [name1[stop1], name2[start2]]

    return name1[0:stop1+1] + name2[start2:]
except IndexError:
    # this could also happen if name1/name2 empty
    return -1
```

9. [1 point] 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.