CS 116 Winter 2013

Assignment 09
Due at 10:00 am on Wednesday, April 3

- All solutions are to be in Python.
- Each question requires the use of dictionaries and/or classes in a nontrivial way.
- Do NOT use recursion in your solutions. You may use abstract list functions.
- Do not import any modules except the `check` module.
- Do not use any global variables.
- Download the testing module from the course Web page. Include `import check` in each solution file.
- Be sure to use the strings that exactly match those specified on the assignment and interface. Changing them in any way may result in the loss of correctness marks.
- You are encouraged to use helper functions in your solutions as needed. Include them in the same file as your solution, but make helper functions separate functions from the main function, i.e. do NOT make them local functions. You do not need to provide examples and tests for these helper functions.
- Do not copy the purpose directly from the assignment description. The purpose should be written in your own words and include reference to the parameter names of your functions.
- Do not cut-and-past examples (or other text) from the assignment description, as this can result in your code failing all tests
- The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.
- Assignments will not be accepted through email. Course staff will not debug code emailed to them.
- Test data for all questions will always meet the stated assumptions for consumed values.
- Read the course Web page for more information on assignment policies and how to organize and submit your work. Follow the instructions in the style guide.
- Specifically, your solutions should be placed in files a09qY.py, where Y is a value from 1 to 4.
- Download the interface file from the course Web page.
- For full marks, it is not sufficient to have a correct program. Be sure to follow all the steps of the design recipe given in the Python Style Guide for CS116, including the definition of constants and helper functions where appropriate.

Coverage: Module 9
Language: Python

We will represent a card by using a Python class: the class has two fields, `suit`, which is a string from the set that contains "clubs", "spades", "diamonds", "hearts", and `value`, which is an integer from 1 to 10 inclusive. The Python definition for this class is:

```
class card:
 ''' A card is an object card(suit, value) where
     * suit is one of 'clubs', 'spades', 'diamonds', 'hearts', and
     * value is an integer in the range 1..10'''
```

You can assume there is an `__init__` method which constructs the `card` objects, along with `__repr__`, `__eq__`, and `__ne__` methods. For the examples below, assume that:

```
card1=card('spades',8),
card2=card('hearts', 5),
card3=card('diamonds',6) and
card4=card('clubs',5).
```

1. (a) Write the function `red_odd`, which consumes a list of `card`s, `hand`, and produces a list of those `card`s which are red (i.e., "diamonds" or "hearts") and have an odd value, in the order they appear in the consumed list. You should not mutate the consumed list. For example:
   `red_odd([card1, card2, card3, card4]) => [card2]` and
   `red_odd([card1, card4]) => []`.

   (b) Write the function `higher_in_rank` that consumes a `card`, `base_card`, and produces a list of all `card`s which have a rank higher than `base_card`, sorted by increasing order of rank. The ranking of cards is as follows:
   - In terms of suits, clubs are lower than diamonds, which are lower than hearts, which are lower than spades, regardless of the value.
   - If two cards have the same suit, the card with the larger value has higher rank.

   For example:
   ```
   higher_in_rank(card1) => [card('spades', 9), card('spades', 10)]
   higher_in_rank(card3) =>
   [card('diamonds',7), card('diamonds', 8), …, card('hearts',1),
   …, card('hearts',10), card('spades', 1), …, card('spades', 10)]
   ```

2. Write the function `go_fish`, which consumes a list of `card`s, `hand`, and a positive integer `v`, and produces None. If `hand` has some `card`s with value `v`, then mutate the `hand` by removing all `card`s with that value from `hand`. If there is no such `card` in `hand`, print "Go Fish!" (and do not mutate `hand`). For example:
   ```
   L = [card1, card2, card3, card4]
   go_fish(L, 9) should print 'Go Fish!'
   go_fish(L, 5) should mutate L so that L => [card1, card3]
   ```

3. Write the function `in_or_out` which consumes three parameters: a dictionary `D` (with strings as the key, and non-negative integers as the value), a string `k` (representing the key) and an integer `t` (representing the threshold). The function `in_or_out` produces None. You can assume all the keys are unique. The function should do the following:
   - If the given key does not occur in the dictionary, add an entry to the dictionary containing the key and the integer 0.
   - If there is an entry with the given key, and value associated with the key is less than the threshold, increment the value associated with the key by one.
   - If there is an entry with the given key, and the value associated with the key is at least as large as the threshold, remove that key from the dictionary.

   For example, consider the following sequence of calls to `in_or_out`, where the

value of the dictionary is shown after each call:

```
L = {}
in_or_out(L, "bob", 3)
L => {'bob': 0}
in_or_out(L, "bob", 3)
L => {'bob': 1}
in_or_out(L, "mary", 4)
L => {'bob': 1, 'mary':0}
in_or_out(L, "mary", 0)
L => {'bob': 1}
in_or_out(L, "natalie", 6)
L => {'bob': 1, 'natalie': 0}
in_or_out(L, "bob", 3)
L => {'natalie': 0, 'bob': 2}
```

4. Write the function `memory` which consumes a non-empty dictionary, `D`, containing string keys and integer values, and produces None. Your function will repeatedly prompt the user to enter a command (as a single uppercase character) and perform the actions described below: notice that you will continually prompt the user until they enter in the letter '`Q`'. You should view the dictionary as a set of variables (the strings) which have particular values (the integer associated with a particular string). Note that you should mutate the consumed dictionary on user inputs `I` and `D`.
   - When the user types `Q`, print out the key and value (separated by one space) of every entry in the dictionary, one entry per line and return (i.e., your function should terminate and return). Your output should be alphabetically sorted by key.
   - When the user types `G`, prompt the user for the two keys they wish to compare. Print `True` if the first entered key value is greater than the second entered key value and `False` otherwise. You may assume both keys exist in `D`.
   - When the user types `L`, prompt the user for the two keys they wish to compare. Print `True` if the first entered key value is less than the second entered key value and `False` otherwise. You may assume both keys exist in `D`.
   - When the user types `E`, prompt the user for the two keys they wish to compare. Print `True` if the values associated with the keys are equal and `False` otherwise. You may assume both keys exist in `D`.
   - When the user types `I`, prompt the user for the key whose value they wish to increase by one. Then, increase the value associated with the key specified by the user. If there is no such variable in the dictionary, do nothing.
   - When the user types `D`, prompt the user for the key whose value they wish to decrease by one. Then, decrease the variable specified by the user. If there is no such variable in the dictionary, do nothing.
   - When the user types `A`, print out the average of all the variable values (as a floating point number with 2 decimal points).
   - You can assume that no other command (other than `Q, G, L, E, I, D, A`) will be entered.

The following interaction should be illustrative (user input is shown in *`italics`*). Note that all prompts are included in the provided interface file.

```
init_D = {'bob':3,'ed':5}
```

```
memory(init_D)
Enter a command: G
Enter the first variable: bob
Enter the second variable: ed
False
Enter a command: A
4.00
Enter a command: I
Which variable to increment: bob
Enter a command: D
Which variable to decrement: ed
Enter a command: E
Enter the first variable: bob
Enter the second variable: ed
True
Enter a command: Q
bob 4
ed 4
init_D => {'bob':4, 'ed':4}
```

Note about testing: you should test your programs using `set_input`, and each individual test should finish with a `'Q'` command.