Test 2: Compsci 101

Owen Astrachan

November 15, 2012

Name: _____

NetID/Login: _____

Honor code acknowledgment (signature) _____

	value	grade
Problem 1	24 pts.	
Problem 2	16 pts.	
Problem 3	20 pts.	
Problem 3	16 pts.	
TOTAL:	76 pts.	

This test has 8 pages (and there is an APT handout), be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.

In writing code you do not need to worry about specifying the proper `import statements`. Don't worry about getting function or method names exactly right. Assume that all libraries and packages we've discussed are imported in any code you write.

PROBLEM 1 : (*Stun Chaos Tipi (24 points)*)

You'll be asked to write code that references the list `nuts` below. The Python code you write should work with any values stored in the list `nuts`. *You can write one line of code or many for each of the tasks below.*

```
nuts = ["cashew", "filbert", "chestnut", "coconut", "macadamia", "peanut", "pecan" \
        "peanut", "peanut", "macadamia", "cashew"]
```

Part A (4 points)

Write code to store in variable `unic` the number of different values stored in `nuts` that begin with the letter 'c' (in the example this value is 3: `"cashew"`, `"chestnut"`, `"coconut"`).

Part B (4 points)

Write code to store in `solo` the number of strings in `nuts` that occur exactly once, this would be four in the example above: `"filbert"`, `"chestnut"`, `"coconut"`, and `"pecan"` each occur once.

Part C (4 points)

Write code to store in `truenut` a list of the (unique) strings in `nuts` that end with `'nut'`. In the example above this is `[chestnut,coconut,peanut]` (order of words doesn't matter)

Part D (6 points)

Write code to store in `nuttiest` the string that occurs most often in `nuts`. This is `"peanut"` in the example above. Assume the string that occurs most often is unique, don't worry about breaking ties.

Part E (6 points)

Write code to store in `freqs` the unique strings in `nuts` sorted by number of times each string occurs with the least frequently occuring string first. For strings that occur the same number of times, break ties alphabetically. For example, for the list `nuts` above the value stored in `freqs` is `["chestnut", "coconut", "filbert", "pecan", "cashew", "macadamia", "peanut"]`.

PROBLEM 2 : (*Songs in the Key of Life (16 points)*)

A data files stores purchase from an online music store like iTunes, part of such a data file, `purchases.txt`, is shown below.

```
ola@cs.duke.edu,Light My Fire,The Doors,0.99
rcd@yahoo.com,Ice Ice Baby,Vanilla Ice,0.99
ola@cs.duke.edu,Your Smiling Face,James Taylor,0.99
rbo@gmail.com,The Cave,Mumford and Sons,1.29
rcd@yahoo.com,Raise Your Glass,P!ink,1.29
rbo@gmail.com,Sleepyhead,Passion Pit,0.99
rbo@gmail.com,Landfill,Daughter,0.99
ola@cs.duke.edu,Raise Your Glass,P!ink,1.29
rbo@gmail.com,Raise Your Glass,P!ink,1.29
```

Each line has four strings separated by commas as shown: email address, track purchased, artist/group name, and price.

The code below, when run on the data file above, generates the output shown. You'll be asked to write code that processes a data file in this format.

```
def datadump(source):

    for line in source:
        line = line.strip().split(",")
        print line

datadump("purchases.txt")
```

If `"purchases.txt"` is the name of the file shown, the code above will print:

```
['ola@cs.duke.edu', 'Light My Fire', 'The Doors', '0.99']
['rcd@yahoo.com', 'Ice Ice Baby', 'Vanilla Ice', '0.99']
['ola@cs.duke.edu', 'Your Smiling Face', 'James Taylor', '0.99']
['rbo@gmail.com', 'The Cave', 'Mumford and Sons', '1.29']
['rcd@yahoo.com', 'Raise Your Glass', 'P!ink', '1.29']
['rbo@gmail.com', 'Sleepyhead', 'Passion Pit', '0.99']
['rbo@gmail.com', 'Landfill', 'Daughter', '0.99']
['ola@cs.duke.edu', 'Raise Your Glass', 'P!ink', '1.29']
['rbo@gmail.com', 'Raise Your Glass', 'P!ink', '1.29']
```

Part A (8 points)

Write code to print the email address of each unique customer and the amount of money spent by that customer. The email addresses should be printed in order of money spent, with the greatest amount spent first. Don't worry about ties.

For the data file above your code should print:

```
rbo@gmail.com $4.56
ola@cs.duke.edu $3.27
rcd@yahoo.com $2.28
```

```
def purchases(source):
    #you write code here to print email, money spent
```

```
if __name__  == "__main__":
    purchases("purchases.txt")
```

Part B (8 points)

Suppose a student writes code to create a dictionary in which the key is an email address and in which the corresponding value is a list of songs purchased by the person with the given email address. For the data file shown the dictionary would be:

```
{'rcd@yahoo.com': ['Ice Ice Baby', 'Raise Your Glass'],
 'ola@cs.duke.edu': ['Light My Fire', 'Landfill', 'Raise Your Glass'],
 'rbo@gmail.com': ['The Cave', 'Sleepyhead', 'Landfill', 'Raise Your Glass']
}
```

Write a function `topsong` whose *parameter is a dictionary in the format described*; the function returns a string: the song purchased by more people than any other song. Don't worry about ties.

```
def topsong(songd):
    '''
    songd is dictionary: key is email address
    corresponding value is list of song titles purchased
    by person with email address

    returns: song bought by most people
    '''
```

PROBLEM 3 : (*Adenosine Triphosphate (20 points)*)

Part A (10 points)

The write-up for the APT *ContestWinner* is at the end of this test. You're given code below that uses the following two-step idea to solve the problem, the code is only partially complete.

1. Find the number in `events` that occurs maximally/most-often; store the maximal number of occurrences in variable `most`.
2. Process the elements of `events` in order, updating a dictionary of how many times each element has occurred, when this count is the same as `most`, return the element.

In the third example of the APT writeup, both 123 and 456 occur three times, so the maximal value is three. As the list `[123,123,456,456,456,123]` is processed in step-two, the count of 456 becomes three when the count of 123 is still two. So 456 is returned as soon as its stored count becomes three.

Complete the code below by adding fewer than eleven lines of code so that the function is correct (all green) and the idea above is used to get all-green.

```
def getWinner(events):
    most = max([events.count(x) for x in events])
    d = {}
    for elt in events:
```

Part B (10 points)

The APT *KingSort* is attached at the end of the test. To solve this APT a helper function that returns an `int` corresponding to a roman numeral as a string is used, e.g., `roman_to_int("XLIV")` returns 44, `roman_to_int("III")` returns 3, `roman_to_int("XXXIX")` returns 39 — in general assume that the function `roman_to_int` called in the code below is correct for use in the APT. You are to add two lines of code in the function `getSorted` below so that it is all green. Note that the return statement refers to a list named `tups` so be sure to assign a value to a variable of that name.

Each of the two lines you write should call `sorted` and should specify both a list and a key to sort on. This means that each line should refer to a list (being sorted) and use `key=operator.itemgetter` with an appropriate value.

Note that `"Charles VII"` should come before `"Jean II"` in the final list returned.

Briefly explain why you wrote each line as well as the order in which the lines occur

```
def roman_to_int(rom):
   if rom == "I": return 1
   if rom == "II": return 2
   # rest not shown

def getSortedList(kings):
    pairs = [x.split() for x in kings]
    data = [(x[0],x[1],roman_to_int(x[1])) for x in pairs]









    return [x[0]+" "+x[1] for x in tups]
```

Briefly explain why you wrote each line as well as the order in which the lines occur

PROBLEM 4 : (*Anthropo, Homeo, Meta, Poly (16 points)*)

Two words are *isomorphic* if they have the same pattern: each letter in one word maps to a corresponding letter in the other word. Mapping a letter means replacing it by another without re-using a letter in the replacement. For example:

- `"abca"` and `"zbxz"` are isomorphic since we map 'a' to 'z', 'b' to 'b', and 'c' to 'x'.
- `"abca"` and `"zxxz"` are *not* isomorphic since we can map 'a' to 'z' and 'b' to 'x', but we cannot map 'c' to 'x' since 'x' has already been used in being mapped to by 'b'. This is a **re-use** error in re-using 'x'.
- `"abca"` and `"zxyp"` are *not* isomorphic since we map 'a' to 'z', map 'b' to 'x', map 'c' to 'y', but we cannot map 'a' to 'p' since 'a' is already mapped to 'z'. This is a **conflict** error in mapping 'a'.

Each letter must map to a letter, no two letters can map to the same letter, but a letter can map to itself.

Part A: (6 points)

Fill in the table by indicating which pairs of words are isomorphic. If words are *not* isomorphic, indicate why.

words	**isomorphic?**	**reason**
`"aab"`, `"ccz"`	yes	
`"xyza"`, `"abcb"`	no	'y' and 'a' both map to 'b', **re-use** on 'b'
`"aabbc"`, `"xxyyc"`		
`"abab"`, `"xyxy"`		
`"aab"`, `"bba"`		
`"abcdc"`, `"xyzyz"`		
`"aacdc"`, `"bbzfy"`		
`"abcda"`, `"abcfa"`		

Part B (6 points)

Write the function `iso` below that returns True if its two string parameters are isomorphic and returns False otherwise. The strings have the same length.

```
def iso(a,b):
    '''
    a and b are strings
    return True if a is isomorphic to b, False otherwise
    '''
    if len(a) != len(b):
        return False
```

Part C: (4 points)

Complete the function `pair_count` that returns the number of unordered pairs that are isomorphic in a list of strings. Call the function `iso` and assume it works as specified. *Do not write more than five lines of code.*

For example:

call	**return**	**isomorphic pairs**
`pair_count(["abca", "zbxz", "opqr"])`	1	`("abca","zbxz")` is the only isomorphic pair
`pair_count(["aa","ab","bb", "cc","cd"]`	4	`("aa","bb")`, `("aa","cc")`, `("ab","cd")` `("bb","cc")`

Complete the function below by calling `iso`:

```
def pair_count(words):
    '''
    words is a list of strings, return number
    of pairs of isomorphic strings in words
    '''

    c = 0



    for i in range(len(words)):




        for j in range(i+1,len(words)):

















    return c
```

APT: ContestWinner

Problem Statement

Exactly one million contestants, numbered 1 through 1,000,000, took part in a programming contest. The rules of the contest are simple: the winner is the contestant who solves the largest number of tasks. If there are more contestants tied for most tasks solved, the winner is the one who was the first to have all of their tasks solved.

Specification

```
filename: ContestWinner.py

def getWinner(events):
    """
    return int based on events,
    a list of int values
    """

    # you write code here
```

During the contest the judges were keeping a log of all accepted solutions. You are given this log as `events`, a list of int values. The i-th element of events is the number of the contestant who submitted the i-th accepted solution (both indices are 0-based).

For example, if `events = [4, 7, 4, 1]`, this is what happened during the contest:

- Contestant 4 solved her first task.
- Contestant 7 solved his first task.
- Contestant 4 solved her second task.
- Contestant 1 solved his first task.
- Compute and return the number of the contestant who won the contest.

Constraints

- `events` will contain between 1 and 50 elements, inclusive.
- Each element of `events` will be between 1 and 1,000,000, inclusive.

Examples

1. `events = 4,7,4,1]`

   `Returns: 4`

   Example from the problem statement.

2. `events = [10,20,30,40,50]`

   `Returns: 10`

3. `events = [123,123,456,456,456,123]`

   `Returns: 456`

4. `events = [1,2,2,3,3,3,4,4,4,4]`

   `Returns: 4`

APT: KingSort

Problem Statement

Every good encyclopedia has an index. The entries in the index are usually sorted in alphabetic order. However, there are some notable exceptions. In this task we will consider one such exception: the names of kings.

Specification

filename: KingSort.py

```
def getSortedList(kings):
    """
    return string list based on kings,
    kings is a string list
    """

    # you write code here
```

In many countries it was common that kings of the same name received ordinal numbers. This ordinal number was written as a Roman numeral and appended to the actual name of the king. For example, "Louis XIII" (read: Louis the thirteenth) was the thirteenth king of France having the actual name Louis.

In the index of an encyclopedia, kings who share the same name have to be sorted according to their ordinal numbers. For example, Louis the 9th should be listed after Louis the 8th.

You are given `kings`, a list of strings. Each element of `kings` is the name of one king. The name of each king consists of his actual name, a single space, and a Roman numeral. Return a list of strings containing the names rearranged into their proper order: that is, the kings have to be in ascending lexicographic order according to their actual name, and kings with the same name have to be in the correct numerical order.

Notes and Constraints

- The Roman numerals for 1 through 10 are I, II, III, IV, V, VI, VII, VIII, IX, and X.
- The Roman numerals for 20, 30, 40, and 50 are XX, XXX, XL, and L.
- The Roman numeral for any other two-digit number less than 50 can be constructed by concatenating the numeral for its tens and the numeral for its ones. For example, 47 is 40 + 7 = "XL" + "VII" = "XLVII".
- Standard string comparison routines give the correct ordering for the actual names of kings.
- Formally, given two different strings A and B we say that A is lexicographically smaller than B if either (A is a prefix of B) or (there is at least one index where A and B differ, and for the smallest such index the character in A has a lower ASCII value than the character in B).

- Each actual name of a king will be a string containing between 1 and 20 characters, inclusive.
- Each actual name will start by an uppercase letter ('A'-'Z').
- Each other character in each actual name will be a lowercase letter ('a'-'z').
- `kings` will contain between 1 and 50 elements, inclusive.
- Each element of `kings` will have the form "ACTUALNAME ORDINAL", where ACTUALNAME is an actual name as defined above, and ORDINAL is a valid Roman numeral representing a number between 1 and 50, inclusive.
- The elements of `kings` will be pairwise distinct.

Examples

1. ```
   kings = ["Louis IX", "Louis VIII"]

   Returns: ["Louis VIII", "Louis IX" ]
   ```

   Louis the 9th should be listed after Louis the 8th.

2. ```
   kings = ["Richard III", "Richard I", "Richard II"]

   Returns: ["Richard I", "Richard II", "Richard III" ]
   ```

3. ```
   kings = ["John X", "John I", "John L", "John V"]

   Returns: ["John I", "John V", "John X", "John L" ]
   ```

4. ```
   kings = ["Philippe VI", "Jean II", "Charles V", "Charles VI", "Charles VII",
   "Louis XI"]
   Returns:

   ["Charles V", "Charles VI", "Charles VII",
   "Jean II", "Louis XI", "Philippe VI"]
   ```

   These are the French monarchs who ruled between 1328 and 1483.

5. ```
   kings = ["Philippe II", "Philip II"]
   Returns: ["Philip II", "Philippe II" ]
   ```

   "Philippe" and "Philip" are distinct names, and "Philip" is lexicographically smaller than "Philippe".
   ```