Homework 5

Intro to Programming (Term III/2017–18)

built on 2018/06/04 at 21:15:17 ***due:*** *thu, jun 15 @ 11:59pm*

This assignment will give you more practice on writing larger programs, focusing on nested loops and dictionaries. You will write some code and hand it in electronically. As with the last assignment:

- For every function you write that returns a value, write at least 5 test cases using `assert`. Put all these tests (`asserts`) in a separate file called `unittests.py`, which you will also hand in. You will need the **import** command to access code you keep in a different file.
- Before you hand in your solutions, delete or comment out all extraneous **print** statements. As a rule of thumb, if the question doesn't ask you to print, you shouldn't print.

New to this assignment:

- For tasks that require reading input from a file, don't write assert's for functions that perform file operations. Structure your code so that the main logic for that program can be tested by writing assert's for the other functions.
- This assignment will be graded out of 60 points although you may earn up 70 points. If you score over 60, that's extra credit.

Overview:

Problem	File Name
1.	`charhist.py`
2.	`bell.py`
3.	`sleuth.py`

Problem	File Name
4.	`karma.py`
5.	`sgroup.py`
*	`unittests.py`

Collaboration

We interpret collaboration very liberally. You may work with other students. However, each student ***must*** write up and hand in his or her assignment separately. Let us repeat: You need to write your own code. You must not look at or copy someone else's code. You need to write up answers to written problems individually. The fact that you can recreate the solution from memory will be taken as proof that you actually understood it, and you may actually be interviewed about your answers.

Logistics

We're using a script to grade your submission before any human being looks at it. Sadly, the script is not as forgiving as we are. *So, make sure you follow the instructions strictly.* It's a bad omen when the course staff has to manually recover your file because the script doesn't like it. Hence:

- Save your work in a file as described in the task description. This will be different for each task. **Do not save your file(s) with names other than specified.**
- You'll zip these files into a single file called `a5.zip` and you will upload this one zip file to Canvas before the due date.
- Before handing anything in, you should thoroughly test everything you write.
- At the beginning of each of your solution files, write down the number of hours (roughly) you spent on that particular task, and the names of the people you collaborated with as comments. As an example, each of your files should look like this:

```
# Assignment XX, Task YY
# Name: Eye Loveprograming
# Collaborators: John Nonexistent
# Time Spent: 4:00 hrs

... your real program continues here ...
```

- The course staff is here to help. We'll steer you toward solutions. Catch us in real-life or online on Canvas discussion.

Task 1: Character Histogram (10 points)

For this task, save your work in `charhist.py`

Write a function `charHistogram(filename)` that takes the file name of a text file and prints a text rendering of the histogram of the English letters that appear in that file. Specifically, your function will perform the following steps:

Step 1: Calculate the frequency of each letter in the English alphabet, treating everything as lower case. This means, for example, the string `"five I"` contains 2 `i`'s

Step 2: Display a histogram on the console using **print** to indicate the frequencies of all the letters. The histogram contains one line per letter, ordered from `a` to `z`. Only the letters with a nonzero frequency are displayed. The frequency of a letter is represented by the number of +'s. For example, if the letter `b` has frequency 15, we'll render `b ++++++++++++++++`, i.e., the letter `b` followed by a single space and 15 +'s.

Example: Consider the first few words of the famous filler text Lorem ipsum:

```
Lorem ipsum dolor sit amet, consectetur adipiscing
elit.  Praesent ac sem lorem.  Integer elementum
ultrices purus, sit amet malesuada tortor
pharetra ac.  Vestibulum sapien nibh, dapibus
nec bibendum sit amet, sodales id justo.
```

Your character histogram code should print the following to screen:

```
a ++++++++++++++++
b +++++
c +++++++
d +++++++
e ++++++++++++++++++++++++
g ++
h ++
i ++++++++++++++++
j +
l +++++++++
m ++++++++++++
n +++++++++
o +++++++++
p +++++++
r ++++++++++++
s ++++++++++++++++
t ++++++++++++++++++
u ++++++++++++
v +
```

Task 2: Love Triangle, Ring A Bell (15 points)

For this task, save your work in `bell.py`

You will be given a positive integer $n > 0$ and you will construct a pattern that is made up of n rows:

- Row 0 contains 1 number—the number 1
- Each subsequent row is one longer than the one before and follows the pattern that you'll discover.

The first *five* rows are

```
[[1],
 [1,  2],
 [2,  3,  5],
 [5,  7, 10, 15],
 [15, 20, 27, 37, 52]
]
```

Your task: Implement a function `loveTri(n)` that returns the first n rows of this (as a list of lists). (*Hint: how does a number relate to the number to the left of it and the number above that number?*)

Task 3: Word Sleuth (15 points)

For this task, save your work in `sleuth.py`

Word sleuth (also known under various other names) is a popular word game found in newspapers and puzzle books. In such a puzzle, letters of words are placed in a grid, and the objective is to find and mark all the words hidden inside the grid.

You can read words:

- horizontally from left to right;
- vertically from top to bottom;
- diagonally from top-left to bottom-right; or
- diagonally from bottom-left to top-right.

A word w appears in a grid if w can be found along one of these directions of reading, with the letters of w appearing consecutively along the path of reading.

In this task, you'll implement a function `wordSleuth(grid, words)` that takes as parameters:

- `grid` — a 2-dimensional list representing the input grid, and
- `words` — a list of words whose presence you look for in `grid`.

Your function will return a list of *all the words* in the `words` list that appear in the grid. The words may appear in any order in the output list but may *not contain duplicates*.

How to Get Started? You will want to break this problem into a few functions, each representing a logical unit of task. We suggest writing at least the following helper functions:

- `containsWord(grid, w)` takes the grid from above and one *single* word and checks if the grid contains the word `w`. To implement this function, you may wish to further break it down into (i) a function that walks the grid in a direction you specify and the string read-out that it encounters; and (ii) a function that checks if the word appears in the read-out string.
- `makeUnique(lst)` takes a list of words `lst` (with possible duplicates) and returns a list of words from `lst`, each with exactly one copy.

Example: Suppose you call `wordSleuth` with `words = ["bog", "moon", "rabbit", "the", "bit", "raw"]` and the following grid:

```
[["r","a","w","b","i","t"],
 ["x","a","y","z","c","h"],
 ["p","q","b","e","i","e"],
 ["t","r","s","b","o","g"],
 ["u","w","x","v","i","t"],
 ["n","m","r","w","o","t"]]
```

This should return a list with the following members `["raw", "bit","rabbit", "bog", "the"]`. Your program must output the same set of words but may (and probably will) output them in a different order. Note: even though `bit` appears twice in the grid, we only list it once in the output.

Ground Rules: For this task, you must **not** import any package. You will build everything from scratch.

Performance Expectations: While in this class, we aren't crazy about speedy code, your code must *not* be excessively slow. For this problem, it means finishing under 1 second on a 75×75 grid with about 50 words in the input.

Task 4: Karma Points (15 points)

For this task, save your work in `karma.py`

Inspired by the concept of brownie points, karma points are a hypothetical social currency that can be earned by doing good deeds and "lost" when doing things that are frowned upon. Contrary to popular belief, karma points can, in fact, be transferred at your fingertips.

You will bring this concept closer to reality. In this task, you'll implement a function `keepTabs(actions)` that take as input a list of actions (an action log book) and returns a dictionary storing the karma points of all the characters mentioned in `actions`. People with *zero* net karma points will be removed from the dictionary before the function returns.

Each entry of `actions` comes in one of the following two forms:

- a name (a sequence of English letters without any punctuation marks or spaces) followed by either `++` or `--`, indicating positive karma and negative karma, respectively. For example, `"Jack++"` or `"Kanat--"`. The names are case-sensitive and may appear in mixed upper- and lower- cases.
- a name followed by `->` then another name, indicating a transfer of all karma points from the first person to the second. For example, `"Abby->Jeff"` means Abby gives all her karma points to Jeff. That is, Jeff inherits all karma points of Abby, be it negative or positive. After that, Abby's karma point resets to 0.

We guarantee that there will *not* be extra spaces or any irrelevant punctuation marks anywhere.

Example: Say we call `keepTabs` on the following actions:

```
actions = ["Jim++", "John--", "Jeff++", "Jim++", "John--", "John->Jeff", "Jeff--",
    "June++", "Home->House"]
```

Initially everyone has 0 karma points. For each positive deed, the point goes up by 1, and for each negative deed, it goes down by 1. As a result:

- `"Jim++"` gives +1 to Jim. `"John--"` gives −1 to John. `"Jeff++"` gives +1 to Jeff. `"Jim++"` gives +1 to Jim. `"John--"` gives −1 to John. *Therefore, now Jim has* 2 *points, John,* −2 *points, and Jeff,* 1 *point.*
- Then `"John->Jeff"` transfers all karma points from John to Jeff. Hence, John's −2 points is given to Jeff. John now has 0 points and Jeff has $1 + (-2) = -1$ points.
- Then `"Jeff--"` gives −1 to Jeff. And `"June++"` gives +1 to June.
- Then `"Home->House"` has no effect because `Home` didn't have any karma point previously (treated as 0) and neither did `House`.

So this is where everyone ends up, recorded in a dictionary that `keepTabs` should return: `{'Jeff': -2, 'June': 1, 'Jim': 2}`. Notice that at the end, `John` has 0 karma points and so doesn't show up in the output.

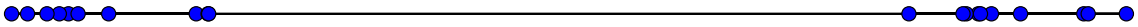
Task 5: Self Grouping (15 points)

For this task, save your work in `sgroup.py`

How can we automatically split seemingly-random data points into meaningful groups? For example, consider the heights of 20 people presented in the ascending order:

```
71.4, 72.73, 74.36, 75.38, 76.15, 76.96, 79.51, 86.82, 87.81, 87.87, 146.38, 150.89,
151.16, 152.18, 152.36, 153.27, 155.7, 160.99, 161.36, 164.55
```

Say we happen to know that there are *two* age groups in this population—kids and adults. *How can we tell them apart?* It may not be clear at first glance, but plotting this reveals an interesting trend:



With the plot, it is visually trivial to split this dataset into two groups: there is a big gap in the middle, separating the heights into two logical sides. The more involved question—and the basis for this problem—is, **how can we discover this gap automatically?**

Answering this and similar questions has been a topic of extensive research, fueled by real-world problems ranging from deciding letter grades to giving suggestions on Facebook (e.g., who are your close friends?).

What we did in our brain was essentially locating the widest gap visually. This makes intuitive and mathematical sense because the widest gap is where our data points are the most dissimilar.

Generalizing this idea gives us a simple heuristic that often works okay on real data. Say we want to identify k groups. We can proceed as follows:

Step 1: Find the largest gap in the data.

Step 2: Split the data at that point.

This gives us 2 groups. To further split the data, we find the largest gap that hasn't been split and split it there. This gives us one more group. We can repeat this process until we get k groups.

Let us illustrate this idea with a simple example. Suppose our input is the following numbers: $1, 1.2, 4.5, 4.7, 9.1, 9.8, 9.9$. They look as follows in plot:

As before, we assume the input is sorted from small to large. And we wish to split it into $k = 3$ groups. To begin, we'll treat the input as one big group:

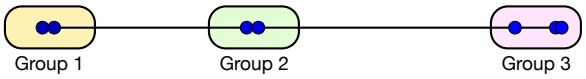
$$[1, 1.2, 4.5, 4.7, 9.1, 9.8, 9.9]$$

Then, we look for the largest gap between two consecutive elements. In this example, the largest gap is in between 4.7 and 9.1 (you can check that). That is where we want to split. After that, the list becomes:

$$[1, 1.2, 4.5, 4.7] \quad [9.1, 9.8, 9.9]$$

Next, we find the largest gap between two successive elements in all groups. This time, the largest gap is in the first group (to know this, we'll have to check the other group, too). The gap is between 1.2 and 4.5. Hence, we further split it there, leading to the following groups (illustration on right):

$$[1, 1.2] \quad [4.5, 4.7] \quad [9.1, 9.8, 9.9]$$



We now have 3 groups, so we can stop and output them.

REMARKS: Mathematically, this process partitions a list into k sublists so that the *total gap*—the sum of all gaps—between sublists is maximized. It makes sense intuitively and aligns with out intuition that different groups should be dissimilar, so the process attempts to maximize how different they are.

YOUR TASK: You will write a function `separate(data, k)` that partitions `data` into k sublists that maximize dissimilarity using the process described above. The input data is represented by a list of numbers (i.e. int or float) where each element corresponds to each data point. For this task, your implementation *must* find the largest gap and split the dataset at the gap, repeating this process until you have k groups. If there are ties, pick the left-most gap. To keep things simple, we will assume that the input list is always ordered from small to large.

Hence, for example, `separate([1, 1.2, 4.5, 4.7, 9.1, 9.8, 9.9], 3)` should return `[[1, 1.2], [4.5, 4.7], [9.1, 9.8, 9.9]]`.