Homework 2

MPCS 51042 – Python Programming

Due: April 23rd 2019, 11:59 pm

Initial Setup

Make sure to perform a pull upstream inside your repository. This will grab the distribution code for hw2. The command is the following:

```
$ git pull upstream master
```

Style Guide

For this homework and all future homework assignments, we will follow the style guide used by the undergraduate Python course. It's located here: `https://classes.cs.uchicago.edu/archive/2018/fall/12100-1/style-guide/index.html`

Usage Statement

A *usage* statement is a printed summary of how to invoke a program that runs in the Terminal program. It includes a description of all the possible command-line arguments that the program might take in. The usage statement is normally invoked in two ways:

1. By a specific command-line argument (e.g. `-help` or `-h`)
2. When the user enters the wrong number of command-line arguments (not enough or too many) or invalid command-line arguments.

Here's the usage statement for the `git` command:

```
$ git --help
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

... Not showing the full usage statement it's very long...
```

Items within `[..]` are optional command-line arguments that are not required to be provided and `<..>` are required command-line arguments necessary to run the program.

You will write a usage statements in the first problem.

Problem 1

You will implement a program with the following usage statement:

```
usage: problem1 <command>

Command
    up     Print the entered integers in ascending order
    down   Print the entered integers in descending order
    same   Print the entered integers in the same order they were given.
```

Write a program that continually prompts a user to input integers until the user types `<quit>`. Based on the command given on the command line, the program will,

1. *up* - The program prints the entered integers in ascending order, along with the integer's original position (zero-based) among the numbers input by the user, in parentheses.
2. *down* - The program prints the entered integers in descending order, along with the integer's original position (zero-based) among the numbers input by the user, in parentheses.
3. *same* - The program prints the entered integers in same order they were given when entered along with the integer's original position (zero-based) among the numbers input by the user, in parentheses.

The program prints the usage statement if

1. the wrong number of command-line arguments (not enough or too many)
2. Incorrect command is provided (i.e., anything other than `up` or `down` or `same`

Test Cases input/output:

```
$ ipython problem1.py up
Please enter an integer: 3
Please enter an integer: -15
Please enter an integer: 1
Please enter an integer: <quit>
-15 (1)
1 (2)
3 (0)
```

The first value in our sorted collection of input values is -15. It was the second number (i.e., index 1) input. So we print "-15 (1)". The next value in our sorted collection is 1, the last (i.e., index 2) value input by the user. So we print "1 (2)". Finally, we print "3 (0)" because the highest value input was the first value input by the user (i.e., index 0).

```
$ ipython problem1.py down
Please enter an integer: 3
Please enter an integer: -15
Please enter an integer: 1
Please enter an integer: <quit>
3 (0)
1 (2)
-15 (1)
```

```
$ ipython problem1.py
usage: problem1 <command>

Command
    up     Print the entered integers in ascending order
    down   Print the entered integers in descending order
    same   Print the entered integers in the same order they were given.
```

```
$ ipython problem1.py blah foo blah
usage: problem1 <command>

Command
    up     Print the entered integers in ascending order
    down   Print the entered integers in descending order
    same   Print the entered integers in the same order they were given.
```

Assumptions/Requirements:

- You may assume that the user does not submit any non-integer values.
- Assume the values will be unique (i.e., no duplicates).
- **You cannot use any sorting functions that python provides**.

Place your solution inside the **hw2/problem1/problem1.py** file.

Problem 2

A 2014 article in the Chicago Tribune (`http://articles.chicagotribune.com/2014-02-20/news/ct-emanuel-city-empl`
`1_city-workers-chicago-public-city-employee-debt`) highlighted how Chicago mayor Rahm Emanuel indicated to city employees that they must pay any outstanding debt to the city (parking tickets, water bills, etc.) or face possible suspension/firing. In this problem, you will assess how successful (or not) mayor Emanuel has been in getting city employees to pay outstanding debt by analyzing open data `https://data.cityofchicago.org/` provided by the City of Chicago. You are provided a comma-separated value file **hw2/problem2/indebtedness.csv**.

Your task is to write a program that saves the total amount of debt owed by city employees for each date shown in the csv file to a file named **total_debt.txt**. The output to the file must be sorted by date, starting from the oldest entries (‘10/14/2011‘) and continuing to the present. The total amount owed should be displayed rounded to the nearest dollar with thousands separated by commas.

Small sample of the beginning information that will be saved to the **total_debt.txt** file:

```
10/14/2011: 2,553,610
10/21/2011: 2,326,302
10/28/2011: 2,132,597
11/04/2011: 1,950,591
11/11/2011: 1,810,242
...
```

The only package you may use for this problem is `datetime`. You may find the following code helpful:

```
import datetime
my_dates = ['06/24/2017', '03/24/2018', '11/11/2019', '02/11/2017']
my_dates.sort(key=lambda date: datetime.datetime.strptime(date, "%m/%d/%Y"))
print(my_dates)
```

Place your solution inside the **hw2/problem2/problem2.py** file.

Problem 3

Write a function that joins together single components of a path to produce a full path with directories separate by slashes. For example, it should operate in the following manner:

```
$: ipython
In [1]: import problem3
In [2]: problem3.full_paths(['usr', ['lib', 'bin'], 'config', ['x', 'y', 'z']])
Out[2]:['/usr/lib/config/x',
 '/usr/lib/config/y',
 '/usr/lib/config/z',
 '/usr/bin/config/x',
 '/usr/bin/config/y',
 '/usr/bin/config/z']
In [3]: problem3.full_paths(['codes', ['python', 'c', 'c++'], ['Makefile']], base_path='/home/user/')
Out[3]:['/home/user/codes/python/Makefile',
 '/home/user/codes/c/Makefile',
 '/home/user/codes/c++/Makefile']
```

The function definition should look as follows:

```
def full_paths(path_components, base_path='/'):
    ...
```

The `path_components` argument accepts an iterable in which each item is either a list of strings or a single string. Each item in `path_components` represents a level in the directory hierarchy. The function should return every combination of items from each level. With the `path_components` list, a string and a list containing a single string should produce equivalent results as the second example above demonstrates. The `base_path` argument is a prefix that is added to every string that is returned. The function should return a list of all the path combinations (a list of strings). Order does matter. The function must maintain the order of how they are position in the original list.

If you need to check whether a variable is iterable, the "Pythonic" way to do this is

```
from collections.abc import Iterable

if isinstance(x, Iterable):
    ...
```

However, note that strings are iterable too! You are allowed to use functionality from the standard library for this problem.

Assumptions/Requirements:

- You may assume the argument `path_components` is either a string or a list of strings.
- You don't have to worry about nested lists, (e.g., `['codes', ['python', ['c','c++'],'java'], ['Makefile'] ]`)

Place your solution inside the **hw2/problem3/problem3.py** file.

Problem 4

Nowadays we take word completion for granted. Our phones, text editors, and word processing programs all give us suggestions for how to complete words as we type based on the letters typed so far. These hints help speed up user input and eliminate common typographical mistakes (but can also be frustrating when the tool insists on completing a word that you don't want completed).

You will implement two functions that such tools might use to provide command completion. The first function, `fill_completions`, will construct a dictionary designed to permit easy calculation of possible word completions. A problem for any such function is what vocabulary, or set of words, to allow completion on. Because the vocabulary you want may depend on the domain a tool is used in, you will provide `fill_completions` with a representative sample of documents from which it will build the completions dictionary. The second function, `find_completions`, will return the set of possible completions for a start of any word in the vocabulary (or the empty set if there are none). In addition to these two functions, you will implement a simple main program to use for testing your functions.

Specifications

- `fill_completions(fd)` returns a dictionary. This function takes as input an opened file. It loops through each line in the file, splitting the lines into individual words (separated by whitespace) and builds a dictionary:
  - The keys of the dictionary are tuples of the form `(n, l)` for a non-negative integer `n` and a lowercase letter `l`.
  - The value associated with key `(n, l)` is the set of words in the file that contain the letter `l` at position `n`. For simplicity, all vocabulary words are converted to lower case. For example, if the file contains the word `"Python"` and `c_dict` is the returned dictionary, then the sets `c_dict[0, "p"]`, `c_dict[1, "y"]`, `c_dict[2, "t"]`, `c_dict[3, "h"]`, `c_dict[4, "o"]`, and `c_dict[5, "n"]` all contain the word `"python"`.
  - Words are stripped of leading and trailing punctuation.
  - Words containing non-alphabetic characters are ignored, as are words of length 1 (since there is no reason to complete the latter).
- `find_completions(prefix, c_dict)` returns a set of strings. This function takes a prefix of a vocabulary word and a completions dictionary of the form described above. It returns the set of vocabulary words in the completions dictionary, if any, that complete the prefix. It the prefix cannot be completed to any vocabulary words, the function returns an empty set.
- `main()`, the test driver:
  1. Opens the file named `"articles.txt"`. This file contains the text of recent articles pulled from BBC.
  2. Calls `fill_completions` to fill out a completions dictionary using this file.
  3. Repeatedly prompts the user for a prefix to complete.
  4. Prints each word from the set of words that can complete the given prefix (one per line). If no completions are possible, it should just print `"No completions"`.
  5. Quit if the user enters the word `<quit>`.

- To call the `main()` function, put a block at the end of your script with the follow lines (we will discuss this technique later on in the class). This allows your script to run when executed from a command line.

```
    if __name__ == '__main__':
        main()
```

Test Cases input/output

```
$ ipython problem4.py
Enter prefix: za
  zara
  zakharova
  zapad
Enter prefix: lum
  lumley
  lump
  lumet
Enter prefix: multis
  No completions
Enter prefix: <quit>
```

Assumptions/Requirements:

- The output order in this problem does not matter.

Place your solution inside the **hw2/problem4/problem4.py** file.