CSCA08 Assignment 0

Due: October 16, 2016. 11:55pm

Introduction

The goal of this assignment is to get you started reading and writing Python code. You'll understand how to use existing code, call existing functions, write your own functions, and call functions you've written. This work is not harder than what you've been doing in your exercises, but it will take substantially longer, so start early. Make sure you read this entire document all the way through at least once before you start.

What Not to Use

You can complete this assignment using only the material covered in the first three weeks of the course. In particular, you are not allowed to use any if-statements (there is no good reason to use them in this assignment!), and do not use string methods.

Word Search

A word search puzzle is a rectangular grid of letters that contains hidden words. In the word searches we use here, words can be hidden in one of four directions:

- left-to-right (horizontal forwards)
- right-to-left (horizontal backwards)
- top-to-bottom (vertical starting from the top)
- bottom-to-top (vertical starting from the bottom)

The first two of these directions are in the **horizontal** dimension, and the second two are in the **vertical** dimension.

Download and open a0.py. It is a Python file in which you will solve 12 tasks related to word searches. Not all tasks will be marked, but later tasks build on earlier ones. You are strongly encouraged to complete all tasks – you'll thank me later!

Starter Code

At the top of the file are some comments. Please read those comments now, and complete the header... go on... I'll wait here.

Next, you see a variable assignment to `PUZZLE1`. `PUZZLE1` is a word search with eight rows and eight columns in which the following words are hidden:

- thierry (2 instances)
- brian (2 instances)
- nick (2 instances)
- paco (1 instance[1])

[1] We weren't sure that the world could deal with more than one instance of Paco

Before continuing, find all of the words in the puzzle by hand. Remember that they can be in any of four directions. Here's a hint for finding the `brian`s: one instance is right-to-left , the other is bottom-to-top.

Below `PUZZLE1` is `PUZZLE2`. It's a bigger puzzle with the same words hidden in it, but we're not telling you the number of instances of each word that are in there. We'll come back to this puzzle later.

Below `PUZZLE2`, we have given you two starter functions that you will use in completing some of the tasks. These functions use Python concepts that we haven't covered yet, so don't concern yourselves with the implementations. (If you are interested, the code is well-commented, so that by reading the comments you should get an idea of what's happening.) Instead, to learn what the functions do, read the docstrings and experiment in the shell. The two functions are as follows:

`rotate_puzzle`

As indicated by the docstring, this function rotates a puzzle by 90 degrees to the left. ... What exactly does that mean? To get an idea of what the function does, run `a0.py` and call `rotate_puzzle` from the shell. For example, try `rotate_puzzle(PUZZLE1)`. Observe the effect. Try it on smaller "puzzles" too, like this: `rotate_puzzle('abc\ndef')`. Do you understand what the function is doing? (The `\n` there is just the code for a newline character.

`lr_occurrences`

This function takes a puzzle and a word, and returns the number of times the word occurs in the left-to-right direction in the puzzle. Play around with this function before continuing. You know the number of occurrences of each word in `PUZZLE1`; use this function to verify the number of left-to-right occurrences of these words. Try it on `PUZZLE2` as well![2]

On line 103 of `a0.py` is a comment indicating that your code should be below. You are not allowed to modify anything above this line, except for the header. Below this point, there are 12 tasks to complete. Be careful: they are not listed in the code in numerical order. Make sure you work on the tasks in order of number, not the order that they occur in the code.

Your Tasks

Here are your tasks for the word searches. Read the tasks below carefully, and follow along with the code so that you know where to add code and what to add. Part of the goal of this assignment is to give you practice following instructions exactly. What might be considered small mistakes in other disciplines can be surprisingly important in programming. For example, if you print something when you were supposed to return it, it is wrong. If you build a string in a different format than what is expected, it is wrong.

- Task 1: this task comprises four subtasks. All four are very similar, and are found in the `do_tasks` function.
  - Task 1a: you are adding a print function call that prints the number of times name occurs in the left-to-right direction in puzzle. Mind the hint in the code!
  - Task 1b, 1c, 1d: for each of these tasks, you want to print in the same format as task 1a. That is, each of these tasks will have two calls to print. Rotating puzzles is going to be extremely useful here. What happens to words and their directions when you rotate the puzzle?
  - This is a good time to run `a0.py` and observe the output. Notice that there is a call to `do_tasks` right below the indented body of the `do_tasks` function. The function call passes `PUZZLE1` and `'brian'` as arguments to `do_tasks`. Your code is working correctly if it runs with no errors and outputs the expected results for all four directions. If you get errors, or the output is wrong, go back over tasks 1a-d and debug your code.

[2]Hint: For testing purposes, you can input any string, so just try picking a string that you can see in the puzzle

- Task 2: For this task, you are calling `do_tasks` again. Heed the advice given in the comments! (Make sure you run your code frequently to make sure you haven't accidentally broken anything.)
- Task 3: now you get to write your own function. Carefully read the docstring for `total_occurrences` to figure out what to do. What are you supposed to calculate? Are you returning it or printing it? What do the parameters mean? What does the example and type contract tell you? Make sure you are clear on these before writing the code! Also, think back to earlier tasks you have done and how they might help you write this function. Once you have written the code, test the function by calling it from the shell. It's best to know now if there are problems before you start to rely on this function. Test it on small puzzles that you create by-hand. They don't have to be real puzzles; small examples are often most useful for isolating bugs. Test on words that do not exist in the puzzle, words that exist once, and words that exist more than once. Then, test on `PUZZLE1` and `PUZZLE2` – your function should work on any puzzle.
- Task 4: here you call the `total_occurrences` function that you just wrote. Run the program after this step to make sure your function call is working.
- Task 5: another function to write! Except this time, you have to write everything: example, description, type contract, and body (code). The `in_puzzle_horizontal` function should return `True` iff the given word can be found in puzzle in one or both horizontal directions. So if you find the word going left-to-right, or find it going right-to-left (or both), return `True`, otherwise return `False`. As in task 3, test the function from the shell on small puzzles before moving on.
- Task 6: add a call of the `in_puzzle_horizontal` function you just wrote in task 5. As usual, run the program and compare the output to what is expected!
- Task 7: Now run your tasks on `PUZZLE2` (the big puzzle). Observe the output. Make sure it is correct before continuing!
- Task 8: the `in_puzzle_vertical` function should return `True` iff the given word can be found in puzzle in one or both vertical directions. So if you find the word going top-to-bottom, or find it going bottom-to-top (or both), return `True`, otherwise return `False`. Keep using the function design recipe! You're not being asked to add a call of this function to your code, but you should definitely call the function a few times with different parameters to make sure it is working correctly!
- Task 9: the `in_puzzle` function should return `True` iff the given word can be found anywhere in puzzle. So if you find the word in at least one of the four directions, you return `True`, otherwise you return `False`. Think carefully of how you can use existing functions to make this function a lot simpler!
- Task 10: the `in_exactly_one_dimension` function should return `True` iff the given word can be found in puzzle in exactly one of the two dimensions (horizontal or vertical), but not both. So if you find the word horizontally (left-to-right or right-to-left) or you find it vertically (top-to-bottom or bottom-to-top), and assuming you don't find it both horizontally and vertically, then return `True`, otherwise return `False`. Again, think about what earlier functions can make this task easier? This is a question you should always be asking when you write a new function. (Be careful with your booleans on this one!)
- Task 11: the `all_horizontal` function should return `True` iff all occurrences of the supplied word are horizontal in the puzzle. (If word is not in the puzzle at all, then `True` is to be returned.)
- Task 12: the `at_most_one_vertical` function should return `True` iff word occurs at most once in the puzzle and that occurrence (if present) is vertical.

Type Checker

We have included a file called `typechecker.py` that runs each of your functions and checks that the **type** of the return value is correct. Again, you don't have to understand this code (I wouldn't expect you to at this stage), you just have to put it in the same directory as your `a0.py` file and run it.

This file serves 2 purposes, first it calls your functions, so you can be sure that they are named correctly (if they aren't it will crash. Secondly it checks the type (not the value, just the type) of each function's return, so you know if you've at least got that part right. If there is anything wrong, you will see an error message. If the types are correct, nothing will happen, except a single print statement.

This doesn't mean that your function works (for example, replacing any of the boolean functions with just the line `return True` will still pass the type checker, but it's a good first test. You will still want to test your code further.

Marking

Your assignment will be marked for correctness. This means that your code must do exactly what is described in this handout and starter code. In addition, we strongly encourage you to consider the following criteria as you work on this assignment – paying attention to these will help you as we progress through the course!

- Formatting style: Make sure that you follow PEP-8 style guidelines that we have introduced.
- Programming style: Your variable names should be meaningful and your code as simple and clear as possible.
- Commenting: be sure to include accurate and complete docstrings in your functions. Follow the design recipe we have been using! In addition, include internal comments for pieces of code that aren't trivial.
- Code re-use: you should have as little duplicated code as possible. If you find yourself repeating code, there's a good chance you could find a simpler (lazier) method.
- One more time... this will be marked by an auto-marker. So if your file or one of your functions is named incorrectly, you won't get any marks for that component of your work.

What to Submit

Submit `a0.py` on MarkUs. Your file must be named exactly as given here (check that MarkUs says you have submitted all required files after you're done submitting). You do not need to submit `typechecker.py`.

Before you submit:

- Ensure that you have read & added your name and login to the header at the top of the file
- Test your code for PEP-8 compliance
- Run the type checker and make sure it gives no errors
- Re-test all examples

Happy Searching!