CSE 337: Scripting Languages (Fall 2019)

Minor Homework #1

Assignment Due: Saturday, September 21, 2019, by 11:59 PM

Assignment Objectives

By the end of this assignment you should be able to design, code, run and test original Python functions that solve simple programming problems involving strings, lists, and recursion.

Getting Started

This assignment requires you to write Python code to solve several computational problems. To help you get started, we will give you a basic starter file for each problem. These files will contain *function stubs*[1] and a few tests you can try out to see if your code seems to be correct (**note that the test cases we give you in the starter files are just examples; we will use different test inputs to grade your work!**). You need to complete (fill in the bodies of) these functions for the assignments. **Do not, under any circumstances, change the names of the functions or their parameter lists.** The automated grading system will be looking for functions with those exact names and parameter lists; if you change anything related to the header of a function, the grading program will reject your solution and mark it as incorrect.

Directions

Solve each of the following problems to the best of your ability. The automated grading system will execute your solution to each problem several times, using different input values each time. Each test that produces the correct/expected output will earn 1 or more points. This assignment contains 3 problems, and is worth a total of 30 points. **Note that not every problem may be worth the same number of points!**

- ⚠ Each starter file has a comment block at the top with various important information. Be sure to add your information (name, ID number, and NetID) to the first three comment lines, so that we can easily identify your work. **If you leave this information out, you may not receive credit for your work!**
- ⚠ Submit your work as a set of individual Python files (one per problem). **DO NOT** zip or otherwise compress your files, and **DO NOT** place all of your functions in a single file. If you do so, the automated grading system will not be able to process your work and you will receive a failing grade for this assignment!
- ⚠ Each of your functions **MUST** use the names and parameter lists indicated in the starter code file. Submissions that have the wrong function names (or whose functions contain the wrong number of parameters) can't be graded by our automated grading system, and may receive a grading penalty (or may not be graded at all).
- ⚠ Each of your functions must explicitly ***return*** its final answer; the grading program will ignore anything that your code prints out. Along those lines, do **NOT** use `input()` anywhere within your functions (or anywhere before the `if __name__ == "__main__":` statement); your functions should get all of their input from their parameters. Programs that crash will likely receive a failing grade, so test your code thoroughly **with Python 3.7.4** before submitting it.
- ⚠ Blackboard will provide information on how to submit this assignment. You **MUST** submit your final work by the indicated due date and time. We will not accept (or grade) any work that is submitted after the due date and time.
- ⚠ **ALL** of the solution code that you submit (in each function) **MUST** be your own work! You may not receive code from or share code with anyone else, except for the instructor and the (current) course TAs.

[1]Stubs are functions that have no bodies, or have very minimal, bodies

Part I: Building Burgers (10 points)

(Place your answer to this problem in the "burger.py" file)

BetterBurger has hired you to develop a new food ordering system for its customers. Each customer will type in a code representing the structure of his or her order, and your program will calculate the total cost of the order.

An order code is defined as a (case-sensitive) string of letters; each letter represents some element of the burger (the protein type, a topping, or a condiment). The valid codes for protein types, toppings, and condiments are as follows:

Protein		
Letter	Patty Type	Price
`B`	Beef	$2.00
`T`	Turkey	$2.50
`V`	Veggie	$2.25

Toppings ($0.50 each)	
Letter	Topping
`l`	Lettuce
`t`	Tomato
`o`	Onion
`p`	Pickle
`j`	Jalapeño peppers
`c`	Cheese
`b`	Bacon
`s`	Sautéed onions
`m`	Sautéed mushrooms
`f`	Fried onions

Condiments (free)	
Letter	Condiment
`k`	Ketchup
`u`	Mustard
`y`	Mayonnaise
`a`	A.1. steak sauce
`q`	Barbecue sauce

For example, `"Tpmlmy"` represents a turkey burger with pickle, heavy/extra sautéed mushrooms, lettuce, and mayonnaise ($4.50), while `"Bcbqq"` represents a beef hamburger with cheese, bacon, and heavy/extra barbecue sauce ($3.00).

Complete the `burger()` function, which takes a single string parameter representing a customer order. This function returns either a number or a string, depending on the contents of its parameter. Your function should process its parameter one character at a time (a `for` loop is a good way to do this), keeping track of several values:

- The type of protein that has been ordered
- The total number of toppings that have been ordered
- The total number of condiments that have been ordered

A valid order **MUST** include exactly one protein, plus up to five toppings and up to two condiments. The elements of an order may be arranged in any order; for example, condiments and toppings may be mixed together, with the protein type at the end. Toppings and condiments may be repeated (indicating extra quantities of that topping or condiment); repeating a topping or condiment still counts toward the limit for that type of element, so 3 't' ("tomato") characters will only allow the customer to add up to 2 additional toppings (likewise, 2 copies of a single condiment means that no additional condiments may be ordered).

- If the parameter string is a valid order (exactly one protein, 0–5 toppings, and 0–2 condiments), your function should return a float representing the total cost of the order: $protein\_cost + (0.5 \times number\_of\_toppings)$ (don't worry if the cost doesn't have exactly two digits after the decimal point)
- If the order is invalid because it contains too many protein types (or no protein type at all), or because it contains too many toppings or condiments, your function should return the **exact** string "invalid order" (all lowercase).
- If the parameter string contains an invalid character (e.g., an 'x', which is not present in any of the three tables above), your function should immediately terminate and return the **exact** string "unrecognized order code" (all lowercase, with the exact spelling and spacing shown).

Hint: You can simplify your classification tests by using the `in` operator with a single string that holds all the valid letters for a particular category. For example,

```
letter in "BTV"
```

will return `True` if (and only if) the variable `letter` matches any protein type.

Hint: Use separate variables to track the total number of ingredients in each of the three categories. As soon as you encounter a character that would cause you to exceed the specified limit for a particular category (or an invalid character), your code should *immediately* terminate the function by returning an appropriate error message. If you successfully reach the end of the string, then you can calculate and return the final price.

Examples:

Function Call	**Return Value**	**Notes**
`burger("Bck")`	`2.5`	Valid: Beef burger with cheese and ketchup
`burger("Tpmlmy")`	`4.5`	Valid: Turkey burger with assorted add-ons
`burger("altop")`	`invalid order`	No protein option included
`burger("VtojsT")`	`invalid order`	Two protein options selected
`burger("lsucjV")`	`4.25`	Valid, even though the protein is at the end
`burger("Bqcbksmfy")`	`invalid order`	Too many condiments ordered
`burger("Toxpk")`	`unrecognized order code`	‘x’ is not a valid option
`burger("Vqltopjm")`	`invalid order`	Too many toppings ordered

Part II: Identifying Duplicated Programs (10 points)

(Place your answer to this problem in the "duplicates.py" file)

Occasionally, a computer may end up with multiple copies of the same program installed. This can cause serious problems if the wrong version of a program is mistakenly executed (for example, a buggy or outdated version). Complete the `duplicates()` function, which takes a single argument: a list of strings. Each string begins with a forward slash character (`"/"`) and represents the full path to a program installed on a particular Unix system (e.g., `"/usr/local/bin/grep"`). This function returns a new list containing one copy of the name of every program that appears more than once in the argument list (accounting for the fact that each copy of the program will have a different path prefix). For example, if the argument contained `"/bin/mvdir"` and `"/opt/local/bin/mvdir"`, the list of duplicate programs would contain (a single copy of) `"mvdir"`. If there are no duplicate programs, your function should return an empty list.

NOTE 1: Remember that your result should only contain the *program names,* not the associated paths. You may find the `rfind()` function and string slicing to be extremely helpful in solving this problem.

NOTE 2: You **MAY NOT** use Python's `set` or `frozenset` classes in your solution!

Examples:

Function Call	Return Value
`duplicates(["/bin/less", "/opt/local/bin/nice", "/usr/local/bin/host", "/usr/local/bin/apropos", "/usr/local/bin/touch"])`	`[]`
`duplicates(["/opt/local/bin/cp", "/usr/local/bin/ls", "/usr/bin/touch", "/usr/bin/head", "/bin/ls"])`	`["ls"]`
`duplicates(["/opt/local/bin/su", "/usr/bin/yacc", "/bin/strings", "/usr/bin/nice", "/usr/bin/su", "/usr/bin/rm", "/usr/bin/awk", "/opt/local/bin/uptime", "/bin/ed", "/usr/bin/talk", "/usr/bin/less", "/opt/local/bin/cd", "/usr/local/bin/cd", "/usr/local/bin/sed", "/opt/local/bin/sudo", "/usr/bin/uptime", "/usr/bin/gzip", "/usr/bin/uniq", "/usr/local/bin/man", "/usr/local/bin/top"])`	`["su", "cd", "uptime"]`
`duplicates(["/usr/bin/echo", "/bin/screen", "/usr/bin/rm", "/opt/local/bin/grep", "/usr/local/bin/talk", "/opt/local/bin/echo", "/bin/awk", "/bin/vi", "/usr/bin/vi", "/usr/bin/touch", "/bin/make", "/bin/su", "/usr/bin/less", "/opt/local/bin/quota", "/usr/local/bin/grep", "/opt/local/bin/w", "/usr/local/bin/vi", "/bin/sed", "/usr/local/bin/touch", "/bin/emacs"])`	`["echo", "vi", "grep", "touch"]`

Part III: Recursive Order-Checking (10 points)

(Place your answer to this problem in the "is_decreasing.py" file)

Complete the `is_decreasing()` function, which takes a non-empty list of integer values as its only argument. This function calls itself recursively to determine whether the elements of the list are in decreasing (technically, non-increasing) order. If they are, the function returns the built-in Python value `True`; otherwise, the function returns the built-in value `False` (note that these are **NOT** strings!).

HINT: An empty (or single-element) list is always in decreasing order. A list with two or more elements is in decreasing order if the rest of the list (meaning the second element onward) is in decreasing order **and** its first element is greater than or equal to the first element of that sublist; in other words, the list [A, B, C, . . .] is in decreasing order if [B, C, . . .] is in decreasing order and A is greater than B.

NOTE 1: Your code **MUST** use recursion in its solution in order to receive **ANY** credit for this problem. Slicing will also be extremely helpful (if not mandatory).

NOTE 2: Your code **MAY NOT** use any of Python's built-in sorting operations (like the `sorted()` function) in its solution.

Examples:

Function Call	Return Value
`is_decreasing([])`	`True`
`is_decreasing([2])`	`True`
`is_decreasing([28, 28, 24, 14])`	`True`
`is_decreasing([98, 39, 37, 36, 68])`	`False`
`is_decreasing([4, 14, 43, 26, 49, 39, 4, 20])`	`False`
`is_decreasing([97, 95, 75, 74, 67, 66, 48, 42, 42, 18, -5, -5])`	`True`