CSC108H/A08H Strings Lab

At the end of the lab, please show your work to your TA, and return this handout. We will post the handout on the course website at the end of the week.

Objectives

- Explore type `str`
- Investigate using `str`s in `for` loops and `if` statements
- Use `methods` and `functions` and know the difference

1 Manipulating type `str`

The Python type `str` stands for string. A string is a sequence of characters. We can iterate over ("walk through") a `str` in a `for` loop and examine each character. In the Python shell, do the following (alternating who is driving and navigating):

1. Explore the exercises you did on paper earlier — if you don't think you had the right answer, or if you are not sure of some of the answers, try them in the Python shell.
2. Assign the string "superconductivity" to a variable `s`. (If your fingers are up to it, you may use the string "supercalifragilisticexpialidocious".) For the following print statements, recall that "+" combines strings (concatenates them), and "," can be used to separate items in a print statement.
   (a) Write a loop which prints each character in `s` on a separate line.
   (b) Write a loop which prints each character in `s` on the same line with a space after each character.
   (c) Write a loop which prints each character in `s` on the same line, with a space and a comma and another space after each character: `s , u , p , e , r , c ,` etc. Notice that there is a comma after the last character. We won't worry about this.
   (d) Write a loop similar to the previous but without a space between each character and the comma that follows it: `s, u, p, e, r, c,` etc. Again, don't worry about the final comma.

2 Functions involving strings

This section asks you to write the code for the functions specified in the table below. First, a word about correctness.

Before you start writing code for functions, you need to think about how you will make sure your code is correct. You have to know in your own mind, for every function, what the expected output (or return value) is for all possible values of the parameter. That way, after you've written your functions, you can check if they work as you expect.

Obviously, you can't list every possible argument value and outcome. You have to limit your prediction and checking to a relatively small set of values, so that you can do your testing in a reasonable amount of time. As you probably heard in lecture, the best way to do this is to pick some situations that are representative of others, and also to introduce both "usual" and "unusual" situations.

For example, predicting what your function will do with the empty string `''` as a parameter, is an *unusual* test case, since this situation is often unspecified. A *representative* test case might be the string `'f'` (or `'K'`) standing for *all* one-character strings. When we pick a representative test case, we do so believing something very strong: that if the function works for that one case, we can be confident it will work for *all* other cases in the category it represents. Obviously, we must choose carefully, or this may not be true!

1. Write test cases for at least the first function below: `longer`. Do this **before** you start writing the code, and show your TA.

   To write test cases, make a table like the example below, on a separate sheet. Each row consists of three columns: First, specific values for the first and second string arguments of the function; second, the value that you expect the function to return given those specific parameters; and last, a description of the purpose or significance of the test case (that is, what situation the case represents or why it is unusual).

Test cases for `longer`		
Values	**Result**	**Purpose**
`'hello', 'hi'`	`5`	`first string longer`
. . .	. . .	. . .

2. Now write the function(s) below. After you write `longer`, call the function with the values from your table, and see whether it passes your test cases. If it does, great! If it doesn't, also great! You've caught a bug! It means that you have found out about a defect in time, before you start actually trusting your function. It's a good idea to get in the habit of creating similar tables for every function you write.

   Switch the driver and navigator roles for each function. For this section, do not use any of Python's `str` methods. However, you may use `__builtins__` functions.

`longer:` `(str, str) -> int`	Given two strings, return the length of the longer string.
`earlier:` `(str, str) -> str`	Given two strings made up of lowercase letters, return the string that would appear earlier in the dictionary.
`count_letter:` `(str, str) -> int`	Given a string and a single-character string, count all occurrences of the second string in the first and return the count.
`display_character:` `(str, str) -> str`	Given a string and a single-character string, return a string consisting of the second, single character string repeated as many of times as it appears in the first string.
`where:` `(str, str) -> int`	Given a string and a single-character string, return the index of the *first* occurrence of the second string in the first. For example, `where("abc", "b")` should return `1`. If the second string is not in the first, return `-1`.

3 Using `str` Methods

Methods act much like functions. The only difference is that a method is a function that belong to specific data types and is designed to make use of the specific value it is 'called on'. For example, `'abc'.upper()`, a `str` method, uses the value `'abc'` to return its uppercase equivalent, `'ABC'`. What methods a value has depends on its type: `'abc'` has a method `upper()` because `'abc'` is a `str`. Explore the `str` methods using `dir` and `help`, and use them to complete the tasks below. Create strings in the Python shell and do the following (alternate driving and navigating). Each can be done with a single method call.

1. Check whether or not a string ends with the letter "h".
2. Check whether or not a string contains only numbers.
3. Replace every instance of the character "l" (*ell*) in a string with a "1" (*the number one*).
4. Given a string composed only of numbers, pad it with zeros so that it is exactly 6 digits long.
5. Check whether or not a string contains only lowercase letters.
6. Remove all leading zeroes from a string composed only of numbers.