CSC108H Lab 4 Part 2: Loops and Strings

Objectives

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

1 Using `str` Methods

Methods act much like functions. The only difference is that a method is a function that belongs to a specific data type and is designed to make use of the specific value on which it is called. For example, `'abc'.upper()`, a `str` method, uses the value `'abc'` to return its uppercase equivalent, `'ABC'`. The methods available for a specific value depend on the value's type: `'abc'` has a method `upper()` because `'abc'` is a `str`. Explore the `str` methods using `dir(str)` and `help(str.method_name)`, 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. Write your answers down somewhere (paper or notepad file).

1. Determine whether a string ends with the letter "h".
2. Determine whether a string contains only numbers.
3. Replace every instance of the character "x" with "y".
4. Given a string composed only of numbers, pad it with zeros so that it is exactly 6 digits long.
5. Determine whether a string contains only lowercase letters.
6. Remove all leading zeroes from a string composed only of numbers.

2 Functions involving strings

This section asks you to write the code for a few functions. First, however, a word about correctness.

As part of our recipe for writing functions, you write an example call and the expected return value. Then, after writing the function, you call the function with the example value and confirm that your code works in that case. But just doing one or two examples isn't enough to be sure your function works for all possible values of the parameters.

Obviously, you can't test 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. The best way to do this is to introduce both "usual" (representative) and "unusual" situations. The first example in the docstring should be a "usual" case.

For example, a *usual* test case might be the string `'f'` (or `'H'`) 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!

Predicting what your function will do with the empty string `''` as a parameter is an *unusual* test case, since this situation is often unspecified.

You have been given a handout that contains tables of test cases for each function. Each row consists of three columns: First, specific values for each of the function arguments; 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).

1. Begin by completing the test cases for the function `longer` on a piece of paper. Do this **before** you start writing the code and, if your TA is available, show your TA your table. Next, write `longer` following the design recipe. Only include one or two of the most *usual* test cases in the docstring. In step 6 of the design recipe, call the function with all 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 a defect before you or others actually start using and trusting the function.

2. Continue this process for each function: Complete the appropriate section of the table on a piece of paper, develop the function following the design recipe, test the function on all the examples from your table, and fix any bugs you discover.

   Switch driver and navigator roles for each function. For this section, **do not use any of Python's `str` methods. However, you may use builtin functions such as `len`, `max`, etc.**

`def longer(s1, s2):`	Given two strings `s1` and `s2`, return the length of the longer string.
`def earlier(s1, s2):`	Given two strings `s1` and `s2` made up of lowercase letters, return the string that would appear earlier in the dictionary.
`def count_letter(s, char):`	Given string `s` and a single-character string `char`, count all occurrences of `char` in `s` and return the count. Remember that you may *not* use `str.count()`
`def repeat_character(s, char):`	Given string `s` and a single-character string `char`, return a string consisting of `char` repeated as many times as it appears in `s`.
`def where(s, ch):`	Given string `s` and single-character string `ch`, return the index of the *first* occurrence of `ch` in `s`. For example, `where('abc', 'b')` should return `1`. If `ch` is not in `s`, return `-1`. Remember that you may *not* use `str.find()` or `str.index()` or any other string method.

When you have successfully completed all these challenges, show your work to your TA. If you want, you can go back and change your functions `count_letter`, `repeat_character` and `where` to be case-insensitive.