CS 106 Winter 2020

Lab 02: Containers

Due: Wednesday, January 15th, 11:59pm

Revision note: The scope of Lab02 has been reduced since its initial release on Jan 13.

In this lab you will practice writing a few short functions that work with strings and arrays. You'll put your functions together into a single sketch, but the point is not about what the sketch actually does—it will mostly hold a few self-contained functions. But you have to test that your functions actually work. You do that by creating a `setup()` function that calls your functions with a few different arguments and displays the results.

In the `L02` folder, create a new, empty sketch called `Singles`. Inside that sketch, write the following functions:

a. Write a function `repeat()` that takes two values as input: a string and an integer that's at least 0. It returns a new string in which the provided string has been repeated the given number of times. So, for example, `repeat( "cat", 5 )` would return `"catcatcatcatcat"`. Repeating a string zero times should return the empty string `""`.

   (Hint: There are two approaches. If you want to loop it manually, you'll need a local variable to hold the value you're trying to create, and a `for` loop to build onto that variable with copies of the passed-in string. If you want to use a built-in function seen in lecture, there is one.)

b. Write a function `isIncreasing()` that takes an array of integers as input and returns a boolean value. The function checks if the array of integers is *strictly increasing*: that is, if each number in the array is strictly larger than the one before it. For example, the array `[ -3, 4, 9 ]` is strictly increasing, but `[ 1, 4, 2 ]` and `[ 3, 5, 5, 6 ]` are not.

   (Hint: one idea is to use a `for` loop to walk over the array, comparing each element with the one immediately after it. But because we're comparing *pairs* of elements, you'll need to stop looping one step early. Still, you'll only need about 5–7 lines of code for the function body.)

c. Write a function `countOccurrences()` that takes two values as input: an array of strings and a single search string in a separate parameter. The function returns the number of times that the search string occurs in the array. For example, if `arr` holds the array `[ "apple", "pear", "peach", "apple", "apple" ]`, then `countOccurrences( arr, "apple" )` would return `3`, and `countOccurrences( arr, "durian" )` would return `0`.

d. Write a function `doubleOrNothing()`. It takes an array of integers as input and returns a new array of integers, of the same length as the input. In the output, every positive number in the input array is doubled, and every negative number in the input array is replaced by zero. For example, if `arr` holds the array `[ 1, 2, -5, 8, -3, 12 ]`, then `doubleOrNothing( arr )` would return the array `[ 2, 4, 0, 16, 0, 24 ]` and arr would be unaffected.

e. Write a function `removeMultiples()`. It takes two values as input: an array of positive integers and a single search positive integer in a separate parameter. The function modifies the array it was given and does not return anything. It removes all the entries that are whole multiples of the search number. For example, if `arr` holds the array `[ 1, 2, 3, 4, 7, 12, 15, 81, 1 ]`, then calling

```
removeMultiples( arr, 3 );
print( arr );
```

would print `[ 1, 2, 4, 7, 1 ]` because 3, 12, 15, and 81 are all multiples of 3. Make sure you test with cases that have several yes-multiples in a row, like the 12-15-81 run shown here. You can use the modulo (%) operator to get the remainder of division by the second operand; for example, the following prints “yay.”

```
if( 81 % 3 === 0  &&  81 % 5 !== 0 ) { print("yay"); }
```

Note that these are "pure functions". Everything described as an "input" above should be passed to the function as a parameter (not via a global variable, or an external file, or user interaction in the sketch window). Everything described as "returned" or "output" should be produced via a `return` statement in the function body (not displayed in the sketch window or written to the console using `println()`). Your `setup()` function, which performs testing, can print outputs from test calls to the functions to the console, or draw them on your canvas, as you prefer.

Submission

When you are ready to submit, please follow these steps.

1. Please ensure that any sketches you submit run. It's better to submit a sketch that runs smoothly but implements fewer required features than one that has broken code for all features. If you get partway into a feature but can't make it work, comment it out so that the sketch works correctly without it.
2. If necessary, review the Code Style Guide and use Processing's built-in auto format tool. You do not need to use the precise coding style outlined in the guide, but whatever style you use, your code must be clear, concise, consistent, and commented.
3. If necessary, review the How To Submit document for a reminder on how to submit to LEARN.
4. Make sure to include a comment at the top of all source files containing your name and student ID number.
5. Create a zip file called `L02.zip` containing the entire `L02` folder and all its subfolders.
6. Upload `L02.zip` to LEARN. Remember that you can (and should!) submit as many times as you like. That way, if there's a catastrophe, you and the course staff will still have access to a recent version of your code.
7. If LEARN isn't working, and ***only*** if LEARN isn't working, please email your ZIP file to the course account (see the course home page for the address). In this case, you *must mail your ZIP file before the deadline.* Please use this only for emergencies, not "just in case". Submissions received after the deadline may receive feedback, but their marks will not count.