15-112: Introduction to Programming and Computer Science, Spring 2020

Homework 3: Loops and Lists

Due: Tuesday, February 4, 2020 by 22:00

This assignment has 5 questions, for a total of 50 points.
To start this homework....

1. Create a folder named "week3"
2. Create a file called hw3.py and write all your code in that file.
3. When you have completed and fully tested hw3, submit hw3.py to Autolab. For this hw, you may submit up to 20 times (which is way more than you should require), but only your last submission counts.

Some important notes:

1. After you submit to Autolab, make sure you check your score. If you aren't sure how to do this, then ask a CA or Professor.
2. There is no partial credit on Autolab testcases. Your Autolab score is your Autolab score.
3. Read the last bullet point again. Seriously, we won't go back later and increase your Autolab score for any reason. Even if you worked really hard and it was only a minor error...
4. Do not use strings (in any form)or any string related operations this week. The autograder will reject your submission entirely if you do.
5. Do not hardcode the test cases in your solutions.
6. Remember the course's academic integrity policy. Solving the homework yourself is your best preparation for exams and quizzes; cheating or short-cutting your learning process in order to improve your homework score will actually hurt your course grade long-term.

1 Style

This is the first homework where you will be graded on style. In this course, I discourage (read forbid) you to read other student's code. However, as a software developer, you will need to read code much more often than writing. It pays to adopt good programming style so that your code is readable and, in turn, maintanable. Although there are several style guides out there, a majority of them have common conventions. You can read the official Python style guide at `https://www.python.org/dev/peps/pep-0008/`.

We will use the following rubric[1] for grading style points. You can lose a maximum of 5 points for style errors.

- Ownership
  - You must include your name and andrewId in a comment at the top of every file you submit.
  - This is good practice for later in life, when you will want to document all code that you contribute to projects.
  - 2-point error: not writing your name/andrewId in a submitted file
- Comments
  - You should write concise, clear, and informative comments that supplement your code and improve understanding.
  - Comments should be included with any piece of code that is not self-documenting.
  - Comments should also be included at the start of every function (including helper functions).
  - Comments should not be written where they are not needed.
  - 5-point error: not writing any comments at all.
  - 2-point error: writing too many or too few comments, or writing bad comments.
- Helper Functions (Top-Down Design)
  - You should use top-down design to break large programs down into helper functions where appropriate.
  - This also means that no function should become too long (and therefore unclear).
  - 5-point error: not using any helper functions (where helper functions are needed).
  - 2-point error: using too many or too few helper functions.
  - 2-point error: writing a function that is more than 20 lines long. Exceptions: blank lines and comments do not count towards this line limit, and this rule does not apply to graphics functions and init()/run() functions in animations.

[1]Adopted with minor modifications from `https://www.cs.cmu.edu/~112/notes/notes-style.html`

- Variable Names
  - Use meaningful variable and function names (whenever possible).
  - Variables and functions should be written in the camelCase format. In this format, the first letter is in lowercase, and all following words are uppercased (eg: tetrisPiece).
  - Variable names should not overwrite built-in function names; for example, str is a bad name for a string variable. Common built-in keywords to avoid include dict, dir, id, input, int, len, list, map, max, min, next, object, set, str, sum, and type.
  - 5-point error: not having any meaningful variable names (assuming variables are used).
  - 2-point error: using some non-meaningful variable names. Exceptions: i/j for index/iterator, c for character, s for string, and n/x/y for number.
  - 2-point error: not using camelCase formatting.
  - 2-point error: using a built-in function name as a variable.
- Unused Code
  - Your code should not include any dead code (code that will never be executed).
  - Additionally, all debugging code should be removed once your program is complete, even if it has been commented out.
  - 2-points error: having any dead or debugging code.
- Formatting
  - Your code formatting should make your code readable. This includes:
    - Not exceeding 80 characters in any one line (including comments!).
    - Indenting consistently. Use spaces, not tabs, with 4 spaces per indent level (most editors let you map tabs to spaces automatically).
    - Using consistent whitespace throughout your code.
    - Good whitespace: x=y+2, x = y+2, or x = y + 2
    - Bad whitespace: x= y+2, x = y +2, or x = y + 2
  - 2-point error: having bad formatting in any of the ways described above.

1. **Helper Functions**

   In this section, we ask you to write functions that can be useful in solving parts of later tasks. We hope that you will take advantage of these functions for the solution to those tasks. For these problems, you CANNOT use string functions or convert integers to strings.

   (a) [3 points] Write the function concatNumbers(x,y) which, given two non-negative integers x and y, returns a concatenation (sequential combination) of these two numbers as an integer. Following are some examples of calling this function and expected output:

   - concatNumbers(5,8) should return 58
   - concatNumbers(200,12) should return 20012
   - concatNumbers(25,8) should return 258
   - concatNumbers(8,25) should return 825
   - concatNumbers(12,200) should return 12200

   (b) [3 points] Write a function called sizeof(n) which, given an integer n, returns the number of digits in this integer. For example, calling this function with n = 5, should return 1. Calling the function with n = 3421, should return 4.

   (c) [3 points] In this task, you will write a function that takes some number of digits from the left part of an integer and returns that value. This function should be named getLeftkDigits and should take two arguments n and k, where n is the integer value and k is the number of digits to extract from the left. You can assume that n will always be a non-negative integer. For example:

   - getLeftkDigits(1234,2) should return 12
   - getLeftkDigits(1234,3) should return 123
   - getLeftkDigits(1234,5) should return 1234
   - getLeftkDigits(0,1) should return 0

   (d) [3 points] In this task, you will write a function that removes some digits from the left part of an integer and returns the remaining digits as an integer. This function should be named removeLeftkDigits and should take two arguments n and k, where n is the integer value and k is the number of digits to remove from the left. You can assume that n will always be a non-negative integer. For example:

   - removeLeftkDigits(1234,2) should return 34
   - removeLeftkDigits(1234,3) should return 4
   - removeLeftkDigits(1234,5) should return 0
   - removeLeftkDigits(0,1) should return 0

2. **Kaprekar Numbers**

   A Kaprekar number is a non-negative integer, the representation of whose square can be split into two possibly-different-length parts (where the right part is not zero) that add up to the original number again. For instance, 45 is a Kaprekar number, because

45**2 = 2025 and 20+25 = 45. You can read more about Kaprekar numbers at https://en.wikipedia.org/wiki/Kaprekar_number. The first several Kaprekar numbers are: 1, 9, 45, 55, 99, 297, 703, 999 , 2223, 2728

(a) [4 points] Write the function isKaprekarNumber(n) that takes a non-negative integer n and returns true if n is a Kaprekar number and false otherwise.

(b) [4 points] write the function nthKaprekarNumber(n) that takes a non-negative int n and returns the nth Kaprekar number, where as usual we start counting at n==0

3. **List as Integers**

In this problem we will use normal Python integers to represent lists of integers. Since we plan to include multiple integers in a single integer, we will start by encoding integers with a prefix of the number of digits in that integer. For example, we might encode 25 as 225, where the first 2 says there are 2 digits. But then we'd have to encode 1234567890 with a count of 10. But then we'd have to know how many digits our count itself has! And we are now recursing. Oh no!

Our solution, which is not fully general but which is good enough for our purposes, is to first have a count-count of the number of digits in the count. That first count-count will always be exactly one digit, so the count itself can be between 1 and 9 digits. So the largest digit count is 999,999,999. So this approach does not allow numbers with one billion or more digits. We can live with that restriction.

We also have to deal with the sign (+/-) of the number. Normally this might be 0 for positive and 1 for negative, but leading 0's can cause confusion, so we'll use 1 for positive and 2 for negative.

Thus, we will encode numbers as such: [sign-digit] [count-count] [count] [number]. So, for example, to encode 789, the sign-digit is 1 (positive). The count is 3, so the count-count is 1. Thus, the entire encoding of 789 is 113789.

For these problems, you CANNOT use string functions or convert integers to strings.

For another example, to encode -1234512345, the sign-digit is 2 (negative). The count is 10, so the count-count is 2. Thus, the entire encoding of -1234512345 is 22101234512345.

(a) [5 points] Write the function encode(n) that takes a possibly-negative Python integer and returns the encoded integer as described above. For example:

- encode(789) should return 113789
- encode(-789) should return 213789
- encode(1234512345) should return 12101234512345
- encode(-1234512345) should return 22101234512345
- encode(0) should return 1110

(b) [5 points] Now write the decode function that preforms the inverse operation of the encode function. Given an encoded integer, this function will return the original integer back. The function returns two values as a list, the first element in the list is the decoded value starting from the left side. The second element is any remaining portion that were not decoded.

- decode(113789) should return [789,0]
- decode(213789) should return [-789,0]
- decode(12101234512345) should return [1234512345,0]
- decode(22101234512345) should return [-1234512345,0]
- decode(1110) should return [0,0]
- decode(113789113789) should return [789,113789] since after decoding 789 from the left, we are have 113789 remaining.

(c) [5 points] We will use the encoding functionality to encode a whole list of integers as one list. Write a function called encodeList(L), where L is a List of integers. The values should be encoded such that value at index 0 is to the left most side of the final integer. This function should return all elements in the list encoded within a single integer. for example:

- encodeList([789,-789,1234512345]) should return 11378921378912101234512345

(d) [5 points] The last part of this task does the inverse of the previous task. In this task, you write a function called decodeList that takes an encoded list as a parameter and passes back a list of all the integers. for example:

- decodeList(11378921378912101234512345) should return [789,-789,1234512345]

4. **So many triangles**

So there are different types of triangles. I knew for the longest time that there are right triangles, isosceles, scalene, and equilateral triangles but, until I started teaching my son Geometry over one summer, I did not know that there were obtuse angled triangles, acute angled triangles, and even acute angled isosceles triangles. The following is a list of triangles that I found and I am sure there are more

- Equilateral triangle.
- Right triangle.
- Obtuse angled scalene triangle.
- Obtuse angled isosceles triangle.
- Acute angled scalene triangle.
- Acute angled isosceles triangle.

The cool thing is that you can use the measure of the three sides to figure out if, infact, the three sides can represent a triangle or not , and if they can, what kind of triangle they represent.

(a) [6 points] Your task is to write a function that will take three values as input parameters. These values represent the measure of three sides of a triangle and you can assume that these numbers would be integers. You function will determine whether these three sides represent a triangle and if so what kind of triangle. Based on this determination, the function should return the appropriate value based on the following list (Notice that the function never returns 2 - don't ask me why):

- Sides cannot be a triangle - return 0
- Right triangle - return 1.
- Equilateral triangle - return 3.
- Obtuse angled scalene triangle - return 4.
- Obtuse angled isosceles triangle - return 5.
- Acute angled scalene triangle - return 6.
- Acute angled isosceles triangle - return 7.

You should call this function whichTriangle.

5. **Hopscotch**

   (a) [4 points] Write a function called findExit, that takes list of integers as input parameters. Each number in this list represents the number of hops you need to make. You start from index 0 and make as many hops as the number at index 0. Each time you land on an index, make the number of hops at that index. Determine whether you can reach the last index or not. Return True if you reach the last index, and False otherwise. In an empty list,there is no last index, so you can never reach it. For example:

   - findExit([2,0,1,0]) returns True
   - findExit([1,1,0,1]) returns False
   - findExit([1,2,0,3,1]) returns False