CS 112 – Project #3 **Due Date: Sunday October 7th, 11:59pm.** **Loops, Lists**

The purpose of this assignment is to practice using loops and basic lists effectively.

Background:
Loop statements allow us to run the same block of code repeatedly, with the chance to use different values for variables each time as the accumulated effects pile up. Lists and strings are sequences that can be inspected value-by-value (and modified at will, for lists). We will use loops to define functions that perform calculations over sequences and numbers.

- Project Basics document (part of assignment): http://cs.gmu.edu/~marks/112/projects/project_basics.pdf
- Project Three tester file: http://cs.gmu.edu/~marks/112/projects/tester3p.py
- no template provided – create your file from scratch and include the functions defined below.

Grading

```
Code passes shared tests:     90
Well-commented/submitted:     10
TOTAL:                       100
```

+5 extra credit for count_isolated()

What can and can't I use or do?

Many built-in functions in python would make our tasks so trivial that you wouldn't really be learning, just "phoning it in". The following restrictions are in place for the entire project, and individual functions may list further restrictions (or pointed reminders). The whole point of this assignment is to practice writing loops, and seeing lists in the most basic way possible. There are indeed many different approaches that can "hide the loop" inside a function call, conversion to a different type, and other ways, but we want to make sure you're getting the practice intended from this assignment. ***Learning to code has different goals than finishing programs.***

Restrictions:

- you can't `import` anything
- no usage of sets, dictionaries, file I/O, or self-made classes. Just focus on lists and loops.
- no `zip()` function, and no list comprehensions
- no built-in functions other than those in the "allowed" list.
- you must not modify the original list arguments – but you may make a copy and then mangle your copy.

Allowed:

- variables, assignments, selection statements, loops (of course!), indexing/slicing.
- basic operators: `+`, `-`, `*`, `/`, `//`, `%`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `in`, `not in`, `and`, `or`, `not`.
- if you need to build up a list, you can start with the empty list `[]` and `.append()` values into it.
- only these built-in functions/methods can be used: `len()`, `range()`, `str()`, `int()`, `abs()`, `.append()`, `.extend()`, `.pop()`
- you can write your own version of other functions you want, and call them anywhere in your project.

Hints

In my solution, I only used the following things:

- basic operators, indexing; assignment, selection, and loop statements; just two built-ins, `len()`, `range()`.
- when the answer is a list, I build it up from an initial `[]` by calling `append()` repeatedly.
- you'll *need* to use at least one loop per function, if not more!

Functions

- **`def sum_divisors(n):`** Given a positive integer **`n`**, calculate and return the total of all of its positive divisors. Note: **`n`** is also a divisor of itself.
  - **Assume:** **`n`** is a positive integer.
  - **Restrictions**: remember, you may not call **`sum()`**.

```
sum_divisors(6) →   12   # divisors: 1,2,3,6
sum_divisors(1) →   1    # only divisor: 1
sum_divisors(7) →   8    # divisors: 1,7
```

- **`def pi(precision):`** One can approximate the value of pi by using the Leibniz formula (given below) that calculates a theoretically infinite sum. In practice, though, we run a finite summation where the larger the number of terms the better the approximation of pi. Given **`precision`** as a **`float`**, this function approximates the value of pi using the Leibniz formula. The function should stop the infinite summation when the improvement of the approximation becomes smaller than the provided **`precision`**.

$$\pi = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

Leibniz formula (source: http://forum.codecall.net/uploads/monthly_03_2012/post-80377-13333829507556.png)

  - **Assume:** **`precision`** is a positive **`float`** value.
  - **Restrictions**: you must use a loop in your implementation to receive credit.

```
pi(1.0)    →   3.466666666666667 # total of the first three terms since 4/5<1.0
pi(0.5)    →   3.3396825396825403# total of the first five terms since 4/9<0.5
pi(0.1)    →   3.189184782277596 # total of the terms up to and including 4/41
```

- **`def span(nums):`** Given a list of numbers, return the difference between the largest and smallest number. If the list is empty, return zero.
  - **Assume:** **`nums`** is a list of any length, and all values are numbers.
  - **Restrictions**: remember, you may not call **`min(),max()`** or other related built-in functions.

```
span([1,0,-10,8.5]) →   18.5 # largest is 8.5; smallest is -10
span([3,3,3]        →   0    # 3 is the only value
span([])            →   0    # no value at all.
```

- **`def single_steps(nums):`** Given a list of numbers, count and return how many neighboring values in the list differ by one.
  - **Assume:** **`nums`** is a list of any length, and all values are integers.
  - **Restrictions**: no extra restrictions.

```
single_steps([0,1,8,7,6])     →   3    # three pairs differ by 1: (0,1), (8,7), (7,6)
single_steps([0,-3,8,6])      →   0    # no neighboring values differ by 1
single_steps([])              →   0    # empty list
```

- **`def remove_echo(xs):`** An echo is a contiguous sequence of identical values. Given **`xs`** as a list of values, this function returns a copy of **`xs`** with all but one value for each echo removed.
  - **Assume:** **`xs`** is a sequence of any length.
  - **Restrictions**: remember, you may not call **`.index()`**; do not modify xs.

```
remove_echo([1,1,3,3,3,2,1,2,2])          → [1,3,2,1,2]
remove_echo([1,3,2,1,2])                  → [1,3,2,1,2]           # no echo
remove_echo(["B","a","l","l","o","o","n"]) → ["B","a","l","o","n"] # a list of strings
remove_echo([])                           → []
```

- **`def even_product_2d(grid):`** Given a list of lists of numbers, named **`grid`**, multiply all the even numbers in grid and return the product. If there are no even numbers, return **`1`**.
  - **Assume:** **`grid`** is a list of lists of numbers, e.g. **`[[1,2,3],[4,5,6]]`**.
  - **Restrictions**: no extra restrictions.

```
even_product_2d([[1,2,3],[4,5,6]])      →   48  # evens are: 2,4,6
even_product_2d([[2,1,7],[],[-3,5]])    →   2   # 2 is the only even value
even_product_2d([[]])                   →   1   # no even value
```

Extra Credit Function

Solve this problem for extra credit (up to +5%) and good practice.

- **`count_isolated(grid):`** Given **`grid`** as a 2D list of characters representing a two-dimensional board. Assume that we use string containing a single period (**`'.'`**) to represent an **empty** cell. One cell can have up to eight neighbors, which are cells adjacent to it either horizontally, vertically, or diagonally. A cell is **isolated** if all its neighbor cells are empty. This function counts and returns how many ***non-empty*** cells in the board are isolated. Make sure to check the horizonal, vertical, and diagonal neighbors.
  - **Assume:** **`grid`** is a list of lists of characters
  - **Assume:** all rows in the grid have the same number of columns.
  - **Restrictions**: no extra restrictions.
  - **`Examples:`**

```
# one non-empty cell; all its
# neighbors are empty
grid1 = [['.','.','.'],
         ['.','X','.'],
         ['.','.','.']]

count_isolated(grid1)     →  1
```

`grid1:` 3x3 board, one non-empty cell.

```
...
.X.
...
```

```
# all cells empty
grid2 = [['.','.','.'],
         ['.','.','.'],
         ['.','.','.']]

count_isolated(grid2)     → 0
```

`grid2:` 3x3 board, all cells empty

```
...
...
...
```

```
# cell A has cell B as its diagonal neighbor;
# cell C has cell B as its vertical neighbor;
# cell B has two non-empty neighbors (A and C)
grid3 = [['.','.','.','A'],
         ['X','.','B','.'],
         ['.','.','C','.']]

count_isolated(grid3)     → 1
```

`grid3:` 3x4 board, (three rows, four columns). Four non-empty cells, but only one (cell **`X`**) is isolated.

```
...A
X.B.
..C.
```