Lab 7: Iteration Patterns and Multi-Dimensional lists

Purpose

1. To learn to apply the map, filter, and reduce iteration patterns for processing elements of a list.
2. To learn to use for-loop, while-loop, and list comprehension for processing elements of a list.
3. To learn to process multi-dimensional lists with nested loops.

Description

This assignment requires solving several relatively simple problems using specific iteration patterns. You will likely have very similar code in the functions for each pattern.

Map Pattern

Each value in the result list is determined by computation on the value in the corresponding position (i.e. at the same index) of the input list.

Filter Pattern

Values in the result list are determined by a conditional test on each value in the input list. Only those values that satisfy the condition will be copied to the result list. Typically, the values in the result list should be in the same relative order as in the input list.

Reduce Pattern

Values in a list are combined in some manner (e.g. arithmetic or relational operation) to compute a single result value. For example, the functions `min` and `max` take multiple values and return the lowest and highest of them, respectively.

Implementation

Create a file **lab7.py**, and write your implementations of the following functions in the file. You are required to write at least 3 tests for each of the functions below. You may not use Python's **`pop, remove, del, index, sum, mean, max, min, map`**, **`filter`**, or **`reduce`** built-in functions. You may use `append()`.

All top level code (code that does not belong to any function) must be placed in the if __name__ == '__main__': statement block at the bottom of your file. You might not be able to get any credit for your work if you have any top level code outside the if statement block.

At the bottom of your file, in the if statement block, add the following two lines for using doctest as usual:

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Map Pattern

Implement one of these functions using a list comprehension, one using a `for` loop, and one using a `while` loop.

`square_all(int_list:list)->list`

Returns a new list whose elements are the square of each value in the input list.

1. Write the function docstring.
2. Write three examples in doctest's format in the docstring.
3. Implement the function.
4. Test the function using doctest.

`add_n_all(int_list:list, num:int)->list`

Returns a new list with `num` added to each element in the input list.

1. Write the function docstring.
2. Write three examples in doctest's format in the docstring.
3. Implement the function.
4. Test the function using doctest.

`is_even_all(int_list:list)->list`

Returns a new list containing Boolean values representing whether each corresponding integer in the input list is even (`True`) or odd (`False`).

1. Write the function docstring.
2. Write three examples in doctest's format in the docstring.
3. Implement the function.
4. Test the function using doctest.

Filter Pattern

Implement one of these functions using a list comprehension, one using a `for` loop, and one using a `while` loop.

`are_positive(int_list:list)->list`

Returns a new list of all positive values in the input list.

1. Write the function docstring.
2. Write three examples in doctest's format in the docstring.
3. Implement the function.
4. Test the function using doctest.

`are_greater_than_n(int_list:list, num:int)->list`

Returns a new list of all integers in the input list that are greater than the `num` parameter.

1. Write the function docstring.
2. Write three examples in doctest's format in the docstring.
3. Implement the function.
4. Test the function using doctest.

`are_divisible_by_n(int_list:list, num:int)->list`

Returns all integers in the input list that are divisible by the `num` parameter.

Reduce Pattern

Implement one of these functions using a `for` loop and one using a `while` loop.

`my_avg(my_list:list)->float`

Returns the average of all values in the input list. Do not use the built-in `sum` **and** `mean` function, you may use `len()`.

1. Write the function docstring.
2. Write three examples in doctest's format in the docstring.
3. Implement the function.
4. Test the function using doctest.

`index_of_smallest(my_list:list)->int`

Returns the index of the smallest value in the input list. If the list is empty, then the function returns `-1`. If there is more than one smallest value, return the index of the first occurrence of it (moving from left to right).

1. Write the function docstring.
2. Write three examples in doctest's format in the docstring.
3. Implement the function.
4. Test the function using doctest.

Multi-dimensional list

The elements of the symmetric Pascal matrix are obtained from:

$$P_{ij} = \frac{(i-1)!}{(j-1)!\,(i-j)!}$$

, where P is a n by n matrix, i denotes row, and j denotes column in n by n matrix.
Hence, P_{ij} refers to a cell at i-th row and j-th column. Note that i and j start from 1. So, the number in cell P_{32} shall be (3-1)!/(2-1)!(3-2)! when j <= i. In the upper triangle region, j > i.
Create a function **`pascal_triangle(dim:int)->list`** that generates a Pascal Triangle (elements in the upper right half triangle will be 0), not the symmetric matrix. You must use either while loops or for loops. Use math.factorial() for !. Add the following line in your file to import the math module.

```
import math
```

This is a recommended way to implement pascal_triangle function:

1. Create a dim by dim matrix (a 2 dimensional list) of 0s using two nested for loops. For example, you can create a dim by dim matrix of 0s as follows:

```
matrix = [] #empty matrix that will be populated with 0s
for i in range(dim): #rows
    row = [] #create a new row
    for j in range(dim): #columns
        row.append(0)
    matrix.append(row) #add the new row to the matrix
```

2. Then, use another set of two nested loops to fill in the cells in the lower triangle region of the matrix.
3. You can access the cell at i-th row and j-th column in Python with P[i][j], where i and j start from 0. For example, P[i][j] = 0 to set the cell at i-th row and j-th column to 0.Do not forget that you need to do conversions between 0-based i, j and 1-based i, j.

Write the function docstring and write at least one small example. For example, `pascal_triangle(4)` should return the following list:

```
[[1.0, 0, 0, 0], [1.0, 1.0, 0, 0], [1.0, 2.0, 1.0, 0], [1.0, 3.0, 3.0, 1.0]]
```

, which is the following matrix:

```
1.0    0   0   0
1.0  1.0   0   0
1.0  2.0 1.0   0
1.0  3.0 3.0 1.0
```

This is an example of two nested for loops. Paste the following lines in the console and run the code. Try to understand how the loops work and figure out how you can use them to access elements in a two dimensional list (matrix) one by one.

```
n = 4
for i in range(dim):
    for j in range(dim):
        print(i, j)
```

1. Write the function docstring.
2. Write three examples in doctest's format in the docstring.
3. Implement the function.
4. Test the function using doctest.

This is how you can write an example whose expected output does not fit in one line. … (Three dots followed by a space) indicates that it is a continuation of the previous line. You need to 1)define a variable for an expected output first, 2)write an expression to check equality between the output from the function and the value of the variable, and 3)write the expected result of the expression, e.g. True.

```
>>> expected = [[1.0, 0, 0, 0], [1.0, 1.0, 0, 0],
... [1.0, 2.0, 1.0, 0], [1.0, 3.0, 3.0, 1.0]]
>>> pascal_triangle(4) == expected
True
"""
```

Submission

Submit lab7.py to Gradzilla and get your work graded before submitting it to Canvas. DO NOT CHANGE THE NAMES OF THE FUNCTIONS, OR YOUR FILE WILL NOT RECEIVE ANY POINTS.

Submit `lab7.py` to Canvas before the due date.