CMPSC 132: Programming and Computation II

Lab 8 (10 points) Due date: April 22nd, 2022, 11:59 PM

Goal: The goal of this lab is for you to practice higher-level functions and generators through the various problems present in this lab.

General instructions:

- The work in this assignment must be your own original work and be completed alone.
- The instructor and course assistants are available on Teams and with office hours to answer any questions you may have. You may also share testing code on Teams.
- A doctest is provided to ensure basic functionality and may not be representative of the full range of test cases we will be checking. Further testing is your responsibility.
- Debugging code is also your responsibility.
- You may submit more than once before the deadline; only the latest submission will be graded.

Assignment-specific instructions:

- Download the starter code file from Canvas. Do not change the function names or given starter code in your script.
- Additional examples of functionality are provided in each function's doctest
- If you are unable to complete a function, use the pass statement to avoid syntax errors
- See section 2 for extra practice problems on the topics discussed in Module 9 (optional, not for credit)

Submission format:

- Submit your code in a file named LAB8.py file to the Lab 8 Gradescope assignment before the due date.
- As a reminder, code submitted with syntax errors does not receive credit, please run your file before submitting.

Section 1: Required functions (10 points)

matrixCalculator(matrix1, matrix2, operation) (2 points)

Sums or subtract two square matrices of the same size. You must use list comprehension only for this function (you should only have three to five lines od code), otherwise, you will not receive credit. You can assume matrices and operations are always valid.

Matrix addition:

$$\mathbf{A}+\mathbf{B}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}+\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$$
$$=\begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2n}+b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \cdots & a_{mn}+b_{mn} \end{bmatrix}$$

Matrix subtraction:

$$\mathbf{A}-\mathbf{B}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}-\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$$
$$=\begin{bmatrix} a_{11}-b_{11} & a_{12}-b_{12} & \cdots & a_{1n}-b_{1n} \\ a_{21}-b_{21} & a_{22}-b_{22} & \cdots & a_{2n}-b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}-b_{m1} & a_{m2}-b_{m2} & \cdots & a_{mn}-b_{mn} \end{bmatrix}$$

Input		
list	matrix1	First matrix, represented as a nested list
list	matrix2	Second matrix, represented as a nested list
str	operation	‘add’ for addition, ‘sub’ for subtraction

Output	
list	The sum/subtraction of the two matrices

mulDigits(num, fn) (2 points)

Returns the multiplication of all digits in *num* for which *fn* returns True when receiving the digits as argument. You can assume that *fn* will always be a function that takes one number as argument and returns a boolean value. You are not allowed to use lists, tuples, strings or convert num to string using str(num).

Input		
int	num	A positive integer
function	fn	A function's code reference

Output	
int	Accumulated multiplication of all digits in num that evaluated to True when passed to *fn*

Examples:

```
>>> isTwo = lambda num: num == 2    # Simple anonymous function
>>> mulDigits(5724892472, isTwo)    # Only 2 evaluates to True
8
>>> def divByFour(num):             # Conventional function definition
...     return not num%4
>>> mulDigits(5724892472, divByFour) # Only 4 and 8 evaluate to True
128
```

getCount(x) (2 points)

Takes in a positive integer and returns a function that takes in an integer *num*, returning how many times *x* appears as a digit in *num*. You are not allowed to use lists, tuples, strings or convert num to string using str(num). Note that num//10 does not behave the same when *num* is negative, 562//10 returns 56 while -562//10 returns -57.

Input		
int	x	A positive integer that is less than 10

Output	
function	A function that checks how many times *x* appears in *num*. You can only assume *num* is an integer

Examples:

```
>>> digit = getCount(7)
>>> digit(945784578457077076)
6
>>> getCount(6)(-650621556)
3
```

itemize(seq) (1 point)

A generator function that takes in an iterable object and yields pairs (pos, item) where *item* is each element in *seq* and *pos* is the index of the item in the iteration, starting from 0. You are not allowed to use Python's built-in *enumerate* or *zip* methods, or any other built-in methods in *seq*.

Input		
many	seq	An iterable object

Output (yielded, not returned)	
tuple	Pair (position, item) for every element in *seq*

Examples:

```
>>> gen = itemize('We ARE!')
>>> next(gen)
(0, 'W')
>>> next(gen)
(1, 'e')
>>> next(gen)
(2, ' ')
>>> next(gen)
(3, 'A')
>>> next(gen)
(4, 'R')
>>> next(gen)
(5, 'E')
>>> next(gen)
(6, '!')
>>> next(gen)
Traceback (most recent call last):
...
StopIteration
```

frange(start, stop, step) (1 point)

A generator function that behaves just like Python's *range* function, but it allows to use of float values. Since the function must behave exactly like range, there are three different ways to invoke frange: frange(stop), frange(start, stop) and frange(start, stop, step).

Notice that in the starter code, the function definition has **args* as a parameter. This will allow you to pass multiple arguments to the function instead of limiting the call to only three arguments. The initialization for *start*, *stop* and *step* has been implemented for you in the starter code:

```
if len(args) == 1:      #  frange(stop)
    stop = args[0]
elif len(args) == 2:    #  frange(start, stop)
    start = args[0]
    stop = args[1]
elif len(args) == 3:    #  frange(start, stop, step)
    start = args[0]
    stop = args[1]
    step = args[2]
```

Input		
int/float	start	Optional. A number specifying at which position to start. Default is 0
int/float	stop	Required. A number specifying at which position to stop (not inclusive)
int/float	step	Optional. A number specifying the increment between each number in the sequence. Default is 1

Output (yielded, not returned)	
int/float	Current element in the range rounded to 3 decimals. To round you can use the round method as `round(value, #ofDigits)`

Examples:

```
>>> seq=frange(5.5, 1.5, -0.5)
>>> next(seq)
5.5
>>> next(seq)
5.0
>>> next(seq)
4.5
>>> next(seq)
4.0
>>> next(seq)
3.5
```

```
>>> next(seq)
3.0
>>> next(seq)
2.5
>>> next(seq)
2.0
>>> next(seq)
Traceback (most recent call last):
...
StopIteration
```

genFib(fn) (2 points)

A generator function that yields all Fibonacci numbers *x* for which *fn*(*x*) returns a truthy value. You can assume that *fn* will be defined to work correctly with all Fibonacci numbers. As a reminder, the Fibonacci sequence is a series of numbers where a number is the addition of the last two numbers, starting with 0, and 1:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55…

Input		
function	fn	The function to use in the Fibonacci sequence

Output (yielded, not returned)	
int	Current element in the Fibonacci sequence that evaluates to a truthy value when passed to *fn*

Examples:

```
>>> evens = genFib(lambda x: x % 2 == 0)
>>> [next(evens) for _ in range(15)]
[0, 2, 8, 34, 144, 610, 2584, 10946, 46368, 196418, 832040, 3524578, 14930352,
63245986, 267914296]
```

Section 2: Optional questions

These questions are optional. It is recommended that you complete these problems on your own time to practice and reinforce these concepts in preparation for Quiz 3. Answers are available in a separate file in the LAB8 Canvas assignment.

Without using the Python interpreter, based on the code below, complete the mapper call so it prints the output displayed in red:

```
def mapper(fn, num):
    i = 0
    while i < num:
        print(fn(i))
        i = i + 1

>>> mapper(lambda x: ______, 4)
1
3
5
7
```

Using list comprehension syntax, write the code to create a list that multiplies every part of any list by three.

```
>>> list1 = [3, 4, 5, 6]
>>> list2 = _________________
>>> print(list2)
[9, 12, 15, 18]
```

Using list comprehension syntax, write the code to create a list that contains the first letter of each word

```
>>> words = ['this', 'is', 'a', 'list', 'of', 'words']
>>> letters = _________________
>>> print(letters)
['t', 'i', 'a', 'l', 'o', 'w']
```

Using list comprehension syntax, write the code to create a list that contains the numbers in a string

```
>>> text = 'Hello 12345 World'
>>> numbers = ___________________
>>> print(numbers)
['1', '2', '3', '4', '5']
```

Section 2: Optional questions

Consider the following example:

```
>>> feet = [5, 46, 57.8, 4.6, 2564.846]
>>> feet = list(map(int, feet))
>>> uneven = filter(lambda x: x%2, feet)
>>> type(uneven)
?
>>> print(list(uneven))
?
```

What is printed when the code is executed?

Rewrite the lines of code in the above example, using list comprehensions

```
>>> feet = [5, 46, 57.8, 4.6, 2564.846]
>>> feet = ?
>>> uneven = ?
>>> type(uneven)
>>> print(list(uneven))
```

Using list comprehension, implement the function *coupled(list1, list 2)*, which takes in two lists and returns a list that contains lists with the i-th elements of two sequences coupled together. You can assume the lengths of two sequences are the same.

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> coupled(x, y)
[[1, 4], [2, 5], [3, 6]]
>>> a = ['c', 6]
>>> b = ['s', '1']
>>> coupled(a, b)
[['c', 's'], [6, '1']]
```

Section 2: Optional questions

Just like functions, generators can also be higher-order. Write the generator function *genReminder*(*n*) that takes a positive integer *n* and yields *n* different generators. The first generator is a generator of multiples of *n* (numbers where the remainder is 0), the second is a generator of natural numbers with remainder 1 when divided by *n*, and so on. The last generator yields natural numbers with remainder *n* - 1 when divided by *n*. ***Hint***: Consider defining an inner generator function. Each yielded generator varies only in that the elements of each generator have a particular remainder when divided by n.

Tip: You can use the naturals() generator function discussed in the video lectures to create a generator of infinite natural numbers.

```
>>> numbers = genReminder(5)

# Verify it returns generators
>>> import types
>>> [isinstance(item, types.GeneratorType) for item in numbers]
[True, True, True, True, True]

>>> numbers = genReminder(5)

# Traverse each iterator
>>> for i in range(5):
...     print(f'First 6 natural numbers with remainder {i} when divided by 5:')
...     gen = next(numbers)
...     for _ in range(6):     # get items in current generator
...         print(next(gen))
First 6 natural numbers with remainder 0 when divided by 5:
5
10
15
20
25
30
First 6 natural numbers with remainder 1 when divided by 5:
1
6
11
16
21
26
First 6 natural numbers with remainder 2 when divided by 5:
2
```

```
7
12
17
22
27
First 6 natural numbers with remainder 3 when divided by 5:
3
8
13
18
23
28
First 6 natural numbers with remainder 4 when divided by 5:
4
9
14
19
24
29
```