Extra Credit Lab (10 points) **Due date:** December 2, 2024, 11:59 PM

Before you start…

Keep in mind these tips from your instructor:

- Watch the lectures in 9 before you start this assignment.
- For this assignment, you have to complete two parts, in Part 1 **you may not use any non-anonymous functions**. All functions should be lambda functions. In Parts 2 and 3 you will complete traditional functions and classes, but you are allowed to use map or lambda functions when implementing them (not required)
- Remember that with lambda expressions, you must use list comprehensions if a loop is needed, you cannot use traditional for loops, so you may find it helpful to first write the program out as a traditional function with the for loops you are used to in order to understand each problem. While this is a good approach, **please remember to remove all these attempts before submitting to Gradescope,** otherwise, your code will not receive credit.
- Note that when we say write a lambda function with a given name, what we mean is that you should assign that name to a lambda function that does the task. For example, if it said to write a lambda function `plus_one` that took in a number and returned that number plus one, you would write:

```
plus_one = lambda x: x+1
```

- For initial testing, instead of a doctest, the starter code contains assertions to test each function. Do not modify the given testing code. If your code works as intended, you should see the OK message when running each testing function. An `AssertionError` will be raised by the interpreter for the first failed test.
- Remember there is no partial credit given for failed cases or code that does not follow the given directions.
- Ask questions using our Lab 8 channel in Microsoft Teams

REMINDER: As you work on your coding assignments, it is important to remember that passing the examples provided does not guarantee full credit. While these examples can serve as a helpful starting point, it is ultimately your responsibility to thoroughly test your code and ensure that it is functioning correctly and meets all the requirements and specifications outlined in the assignment instructions. Failure to thoroughly test your code can result in incomplete or incorrect outputs, which will lead to deduction in points for each failed case.

Create additional test cases and share them with you classmates on Teams, we are in this together!

90% > Passing all test cases 10% > Clarity and design of your code

Part 1: For the following problems you will be working with either vectors (lists), or matrices (lists of lists). For each question, you should write a lambda function that does what the question asks. Unless otherwise specified, you may not call other functions within your lambda expression. There is no need to validate the input, you can assume the adequate data type will be provided.

1. **(1 pt)** Write the lambda function `vector_plus_one` that takes in a list as a parameter and returns a new list with every element increased by one.

```
>>> vector_plus_one([1, 2, 3])
[2, 3, 4]
```

2. **(1 pt)** Write the lambda function `collatz_steps` that takes in a list as a parameter, and returns a new list that has removed any non-integers and integers less than 1, and replaces every positive integer left number with a step in the hailstone formulas. As a reminder:
    - If a number is odd, it becomes 3 * n + 1
    - If a number is even, it becomes n // 2

   You cannot make any assumptions about the data types in the list

```
>>> collatz_steps([1, 2, 0, "test", 3, 4, 5])
[4, 1, 10, 2, 16]
```

3. **(1 pt)** Given a number representing the expected size of the matrix, write the lambda function `exchange_matrix` that returns the exchange matrix of the corresponding size. An exchange matrix is a matrix where every value is 0 except for the diagonal from the top right to the bottom left, which is 1.

```
>>> exchange_matrix(3)
[[0,0,1], [0,1,0], [1,0,0]]
```

4. **(1 pt)** Given a matrix, write a lambda function `get_nonzero` that returns a one-dimensional list of the indices of non-zero elements, where each index is a tuple (row, column). The matriz could be rectangular (*m x n*) or squared (*n x n*)

```
>>> get_non_zero([[0,0,1],[0,7,0],[15,1,0]])
[(0, 2), (1, 1), (2, 0), (2, 1)]
```

Part 2: For the following problems, complete your implementation using the definitions provided in your starter code.

mulDigits(num, fn) (1 pt)

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).

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) (1 pt)

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.

Examples:

```
>>> digit = getCount(7)
>>> digit(945784578457077076)
6
>>> getCount(6)(-65062156)
3
```

Dual_Iterator (3 pts)

An iterator that can iterate through all the elements of some sequence (list, string), and when reaching the end of the sequence, it loops back to the beginning. The iterator also includes the method reverse() that allows us to traverse in the reverse order, so when we reach the first element of the sequence, it should loop to the last element of the iterable. You will need to initialize the iterator settings in the constructor of the Dual_Iterator class. You are not allowed to use any built-in reverse methods such as reverse(), [::-1], etc.

The implementation of the __iter__ method has been provided in the starter code. Do not modify it. You should only implement the following methods:

- __init__ method: iterator settings. Takes a sequence as argument. In lecture 9.3 and 9.31, we presented examples of iterator implementations. Revise those examples so you can determine any additional settings besides the sequence (our solution uses only 2 more attributes)
- __next__: returns the next value. It does not raise a StopIteration exception since this is an infinite iterator (loops back)
- reverse(): Takes no arguments and enables the change of direction in the iterator: from forwards to backwards and vice versa

Examples:

```
>>> it = Dual_Iterator([2, 4, 6, 8, 10])
>>> next(it)
2
>>> next(it)
4
>>> next(it)
6
>>> it.reverse()      # it stopped at 6, change direction
>>> next(it)
4
>>> next(it)
2
>>> next(it)
10
>>> it.reverse()     # it stopped at 10, change direction
>>> next(it)
2
>>> next(it)
4
>>> next(it)
6
>>> it.reverse()      # it stopped at 6, change direction
>>> next(it)
```

```
4
>>> next(it)
2
>>> next(it)
10
>>> next(it)
8
>>> next(it)
6
>>> it2 = Dual_Iterator([2, 4, 6, 8, 10])
>>> [next(it2) for _ in range(12)]
[2, 4, 6, 8, 10, 2, 4, 6, 8, 10, 2, 4]
>>> it2.reverse()
>>> [next(it2) for _ in range(12)]
[2, 10, 8, 6, 4, 2, 10, 8, 6, 4, 2, 10]
```

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

A generator function that behaves just like Python's *range* function, but it allows the 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]
```

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
```