Homework #1
Due by Thursday 2/10, 11:59pm

Submission instructions:

1. For this assignment, you should turn in 6 files:
   - A '.pdf' file for the first question.
     Name your file: 'YourNetID_hw1_q1.pdf'
   - 5 '.py' files, one for each question 2-6. Name your files:
     'YourNetID_hw1_q2.py' and 'YourNetID_hw1_q3.py', etc.

   <u>Note</u>: your netID follows an abc123 pattern, not N12345678.

2. **You should submit your homework via Gradescope**.
   For Gradescope's autograding feature to work:
   a. Name all classes, functions and methods **exactly as they are in the assignment specifications**.
   b. Make sure there are **no print statements** in your code. If you have tester code, please put it in a "main" function and **do not call it**.

Question 1:

Draw the memory image for evaluating the following code:

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [lst1 for i in range(3)]
>>> lst2[0][0] = 10
>>> print(lst2)
```

Question 2:

a. Write a function **def** `shift(lst, k)` that is given a list of *N* numbers, and some positive integer `k` (where `k`<*N*). The function should shift the numbers circularly `k` steps to the left.

The shift has to be done **in-place**. That is, the numbers in the parameter list should reorder to form the correct output (you **shouldn't** create and return a new list with the shifted result).

For example, if `lst = [1, 2, 3, 4, 5, 6]` after calling `shift(lst, 2)`, `lst` will be `[3, 4, 5, 6, 1, 2]`

b. Modify your implementation, so we could optionally pass to the function a third argument that indicates the direction of the shift (either 'left' or 'right').

Note: if only two parameters are passed, the function should shift, by default, to the left.

Hint: Use the syntax for default parameter values.

Question 3:

a. Write a short Python function that takes a positive integer *n* and returns the sum of the squares of all the positive integers smaller than *n*.

b. Give a single command that computes the sum from section (a), relying on Python's list comprehension syntax and the built-in `sum` function.

c. Write a short Python function that takes a positive integer *n* and returns the sum of the squares of all the odd positive integers smaller than *n*.

d. Give a single command that computes the sum from section (c), relying on Python's list comprehension syntax and the built-in `sum` function.

Question 4:

a. Demonstrate how to use Python's list comprehension syntax to produce the list [1, 10, 100, 1000, 10000, 100000].

b. Demonstrate how to use Python's list comprehension syntax to produce the list [0, 2, 6, 12, 20, 30, 42, 56, 72, 90].

c. Demonstrate how to use Python's list comprehension syntax to produce the list ['a', 'b', 'c', … , 'z'], but without having to type all 26 such characters literally.

Question 5:

The *Fibonacci Numbers Sequence*, F_n, is defined as follows:
F_0 is 1, F_1 is 1, and $F_n = F_{n-1} + F_{n-2}$ for $n = 2, 3, 4, \ldots$
In other words, each number is the sum of the previous two numbers.
The first 10 numbers in Fibonacci sequence are: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

Note:
Background of Fibonacci sequence: https://en.wikipedia.org/wiki/Fibonacci_number

Implement a function **`def`** `fibs(n)`. This function is given a positive integer `n`, and returns a generator, that when iterated over, it will have the first `n` elements in the Fibonacci sequence.

For Example, if we execute the following code:

```
for curr in fibs(8):
   print(curr)
```

The expected output is:
1 1 2 3 5 8 13 21

Question 6:

You are given an implementation of a `Vector` class, representing the coordinates of a vector in a multidimensional space. For example, in a three-dimensional space, we might wish to represent a vector with coordinates <*5*,−*2*, *3*>.

For a detailed explanation of this implementation as well as of the syntax of operator overloading that is used here, please read sections 2.3.2 and 2.3.3 in the textbook (pages 74-78).

```
class Vector:
    def __init__(self, d):
        self.coords = [0]*d

    def __len__(self):
        return len(self.coords)

    def __getitem__(self, j):
        return self.coords[j]

    def __setitem__(self, j, val):
        self.coords[j] = val

    def __add__(self, other):
        if (len(self) != len(other)):
            raise ValueError("dimensions must agree")
        result = Vector(len(self))
        for j in range(len(self)):
            result[j] = self[j] + other[j]
        return result

    def __eq__(self, other):
        return self.coords == other.coords

    def __ne__(self, other):
        return not (self == other)

    def __str__(self):
      return '<'+ str(self.coords)[1:-1] + '>'

    def __repr__(self):
      return str(self)
```

a. The `Vector` class provides a constructor that takes an integer d, and produces a d-dimensional vector with all coordinates equal to 0. Another convenient form for creating a new vector would be to send the constructor a parameter that is some iterable object representing a sequence of numbers, and to create a vector with dimension equal to the length of that sequence and coordinates equal to the sequence values. For example, `Vector([4, 7, 5])` would produce a three-dimensional vector with coordinates $<4, 7, 5>$.
Modify the constructor so that either of these forms is acceptable; that is, if a single integer is sent, it produces a vector of that dimension with all zeros, but if a sequence of numbers is provided, it produces a vector with coordinates based on that sequence.
Hint: use run-time type checking (the `isinstance` function) to support both syntaxes.

b. Implement the `__sub__` method for the `Vector` class, so that the expression $u-v$ returns a new vector instance representing the difference between two vectors.

c. Implement the `__neg__` method for the `Vector` class, so that the expression $-v$ returns a new vector instance whose coordinates are all the negated values of the respective coordinates of v.

d. Implement the `__mul__` method for the `Vector` class, so that the expression $v*3$ returns a new vector with coordinates that are 3 times the respective coordinates of v.

e. Section (d) asks for an implementation of `__mul__`, for the `Vector` class, to provide support for the syntax $v*3$.
Implement the `__rmul__` method, to provide additional support for syntax $3*v$.

f. There two kinds of multiplication related to vectors:
   1. Scalar product – multiplying a vector by a number (a scalar), as described and implemented in section (d).
      For example, if $v = <1, 2, 3>$, then $v*5$ would be $<5, 10, 15>$.
   2. Dot product – multiplying a vector by another vector. In this kind of multiplication if $v = <v_1, v_2, \dots, v_n>$ and $u = <u_1, u_2, \dots, u_n>$ then $v*u$ would be $v_1*u_1 + v_2*u_2 + \dots + v_n*u_n$.
      For example, if $v = <1, 2, 3>$ and $u = <4, 5, 6>$, then $v*u$ would be 32 (1*4+2*5+3*6=32).

   Modify your implementation of the `__mul__` method so it will support both kinds of multiplication. That is, when the user will multiply a vector by a number it will calculate the scalar product and when the user multiplies a vector by another vector, their dot product will be calculated.

After implementing sections (a)-(f), you should expect the following behavior:

```
>>> v1 = Vector(5)
>>> v1[1] = 10
>>> v1[-1] = 10
>>> print(v1)
<0, 10, 0, 0, 10>

>>> v2 = Vector([2, 4, 6, 8, 10])
>>> print(v2)
<2, 4, 6, 8, 10>

>>> u1 = v1 + v2
>>> print(u1)
<2, 14, 6, 8, 20>

>>> u2 = -v2
>>> print(u2)
<-2, -4, -6, -8, -10>

>>> u3 = 3 * v2
>>> print(u3)
<6, 12, 18, 24, 30>

>>> u4 = v2 * 3
>>> print(u4)
<6, 12, 18, 24, 30>

>>> u5 = v1 * v2
>>> print(u5)
140
```