NYU, Tandon School of Engineering

CS-UY 1134 Data Structures and Algorithms — Summer 2024

Homework #1
Due by Friday 5/31, 11:59pm

Submission instructions:

1. For this assignment, you should turn in 7 files:
   - 2 '.pdf' file for question 1 and 4.
   - 5 '.py' files, one for each coding question
   - name your files using the pattern
     'YourNetID_hwX_qX.py' or
     'YourNetID_hwX_qX.pdf'.

   Note: 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. 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 3:

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 $\langle 4, 7, 5 \rangle$.

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 $v = \langle 1, 2, 3 \rangle$, then $v*5$ would be $\langle 5, 10, 15 \rangle$.
2. Dot product – multiplying a vector by another vector. In this kind of multiplication if $v = \langle v_1, v_2, \ldots, v_n \rangle$ and $u = \langle u_1, u_2, \ldots, u_n \rangle$ then $v*u$ would be $v_1*u_1 + v_2*u_2 + \ldots + v_n*u_n$.
   For example, if $v = \langle 1, 2, 3 \rangle$ and $u = \langle 4, 5, 6 \rangle$, 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
```

Question 4:

Give a θ characterization, in terms of *n*, of the running time of the following four functions:

```
def example1(lst):
 """Return the sum of the prefix sums of sequence
      S.""" n = len(lst)
      total = 0
      for j in range(n):
            for k in range(1+j):
                  total += lst[k]
      return total


def example2(lst):
"""Return the sum of the prefix sums of sequence
      S.""" n = len(lst)
      prefix = 0
      total = 0
      for j in range(n):
           prefix += lst[j]
           total += prefix
      return total


def example3(n):
       i = 1
       sum = 0
       while (i < n*n):
             i *= 2
             sum += i
       return sum


def example4(n):
       i = n
       sum = 0
       while (i > 1):
             for j in range(i):
                   sum += i*j
       i //= 2
       return sum
```

Question 5:

Implement the function **def** `two_sum(srt_lst, target)`. This function is given:

- `srt_lst` - a list of integers arranged in a **sorted** order
- `target` - an integer

When called, it returns two indices (collected in a tuple), such that the elements in their positions add up to `target`. If there are no such indices, the function should return `None`.

For example, if `srt_lst=[-2, 7, 11, 15, 20, 21]`, and `target=22`, your function would return (1, 3) because `srt_lst[1]` + `srt_lst[3]`=7+15=22

Note: Pay attention to the running time of your function. Aim for a linear time algorithm.

Question 6:

Implement the function **def** `split_parity(lst)`. That takes `lst`, a list of integers. When called, the function changes the order of numbers in `lst` so that all the odd numbers will appear first, and all the even numbers will appear last. Note that the inner order of the odd numbers and the inner order of the even numbers don't matter.

For example, if `lst` is a list containing [1, 2, 3, 4], after calling `split_parity`, `lst` could look like: [3, 1, 2, 4].

Implementation requirements:

1. You are **not allowed** to use an auxiliary list (a temporary local list).
2. Pay attention to the running time of your implementation. An efficient implementation would run in a linear time. That is, if n is the length of `lst`, the running time should be $\Theta(n)$.

Question 7:

The number ***e*** is an important mathematical constant that is the base of the natural logarithm. ***e*** also arises in the study of compound interest, and in many other applications.
Background of ***e***: https://en.wikipedia.org/wiki/E_(mathematical_constant)

e can be calculated as the sum of the infinite series:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

The value of ***e*** is approximately equal to 2.71828. We can get an approximate value of ***e***, by calculating only a partial sum of the infinite sum above (the more addends we add, the better approximation we get).

Implement the function **`def`** `e_approx(n)`. This function is given a positive integer n, and returns an approximation of ***e***, calculated by the sum of the first (n+1) addends of the infinite sum above.

To test your function, use the following main:

```
def main():
    for n in range(15):
        curr_approx = e_approx(n)
        approx_str = "{:.15f}".format(curr_approx)
        print("n =", n ,"Approximation:",approx_str)
```

Note: Pay attention to the running time of `e_approx`. By calculating the factorials over for each addend, your running time could get inefficient. An efficient implementation would run in $\Theta(n)$.