CS5001 HW3: Due on Canvas Sun Oct 29 at 11:59pm

You must work **either on your own or with one partner**. If you work with a partner, you and your partner must first register as a group with us then submit your work as a group on Canvas. To register, please send an email to the TAs at the following:

"Qing Chen" chen.qing1@northeastern.edu ; "Lingyu Hu" hu.lingyu@northeastern.edu ; "Mingtianfang Li" li.mingt@northeastern.edu ; "Kejian Tong" tong.ke@northeastern.edu

NOTE: If working in pairs, you can pair up with the same person for a maximum of two assignments/projects in this course. After that, you must either find another partner or complete the assignments individually. Any team in violation of this would receive a zero for their submissions after the first two submissions.

You may discuss background issues and general solution strategies with others, but the programs you submit must be the work of just you (and your partner). We assume that you are thoroughly familiar with the discussion of academic integrity that is on the course website. Any doubts that you have about "crossing the line" should be discussed with Tas or the instructor before the deadline.

Assignment Objectives. Using the string methods count and find. Boolean-valued functions. Helper functions. Iterating through a string with for. Loops.

1. Money from heavens!

Suppose that there is a machine that can dispense any amount of money you want! This machine always uses the fewest number of bills and coins that it can to dispense the specified amount of money. It only dispenses twenties ($20), tens ($10), fives ($5), ones ($1), quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cents).

Design and implement program that writes out what bills and coins would be dispensed if this machine existed for the amount of money that was entered. The money dispenser should be grammatically correct.

1.1 Design a solution

To design your program, start by thinking about how you would go about solving this problem using arithmetic operators and conditional statements, the elements of programming that we have learned thus far in this course. Take the time to write down your solution (step by step) in English then convert the English algorithm to Python using the stepwise refinement strategy we have discussed in class.

Hint: How would the % and // operators be useful to you for solving this problem?

NOTE: Recall, "%" is the modulus operator we discussed in class. "//" is the Floor division operator. It rounds the result down to the nearest whole number. E.g., 21//2 = 10; 15//2 = 7.

1.2 Testing

This is a great assignment to start thinking about testing. What inputs would you want to use to thoroughly test your solution? As you figure out these tests, you should document each one. To do this, create a text file called money_dispenser_tests.txt and in it specify what series of tests you will use to thoroughly test your solution. For each test, specify the amount to be entered and the expected output. One such test could be:

```
Enter amount: 123.45

6 twenties
3 ones
1 quarter
2 dimes
```

1.3 Implementation

Implement the algorithm that you created when you designed your solution in a file called money_dispenser.py .

Hint: You may have heard about the following built-in function:

Function: round(number, [ndigits])

Params: number--the value to be rounded

ndigits--the precision to use. If ndigits is omitted or is None, it returns the nearest integer to its input.

Returns: the rounded number

The behavior of round() for floats can be surprising: for example, round(2.675, 2) gives 2.67 instead of the expected 2.68. This is not a bug: it's a result of the fact that most decimal fractions can't be represented exactly as a float. To avoid this issue, you should use integers instead of floats whenever possible.

Rubrics:

5 points each for correctly printing the number of dispensed twenties ($20), tens ($10), fives ($5), ones ($1), quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cents). The correctness of your program will be tested against many test cases. The program should pass all the test cases to get full credit. A program that fails several test cases will get a very low grade.

2. Roman Numerals

Here are some Roman numeral strings and their associated values:

'I'	1	'II'	2	'MCMLXXXIV'	1984
'V'	5	'IV'	4	'MMMDCCCLXXXVIII'	3888
'X'	10	'IX'	9	'CDIX'	409
'L'	50	'XL'	40	'MMCXL'	2140
'C'	100	'XC'	90	'CI'	101
'D'	500	'CD'	400	'MLI'	1051
'M'	1000	'CM'	900	'CCCXL'	340

In this problem you implement various Boolean-valued functions that can be used to determine if a given string is a valid Roman numeral string. There are quite a few properties that need to be satisfied, e.g., there can be at most three X's, none of the characters in LXVI can come before an M, there can be at most one CM, etc. You will also implement a function that computes the value of a valid Roman numeral string. This is interesting because sometimes C, X, and I indicate negative values. For example, the value of CMXCIX is −100 + 1000 − 10 + 100 − 1 + 10 = 999.

2.1 Getting Set Up

Start by downloading the module Roman.py from the Canvas website. You will notice that it contains seven functions:

Not Implemented: AllCharsOK(R) AllFreqsOK(R) SingleOK(c,s,R) DoubleOK(s,R) Value(R)
Implemented: AllDoublesOK(R) AllSinglesOK(R)

As you proceed with the problem you can use the Application Script (suitably modified) to test your implementations.

2.2 Legal Characters

We define a string to be character-legal if it is made up of the characters M, D, C, L, X, V, and I. Implement a function AllCharsOK(R) that takes a non-zero-length string R as input and returns True if R is a character- legal string and returns False if it is not. (Your function should assume the length of its input is at least one; don't put in a test for the length being zero.) Thus, the value of AllCharsOK('IXIIIDDCM') is True and the value of AllCharsOK('3X') is False. Hint. Count the number of "good" characters in R and then compare that integer to len(R).

Make sure your implementation of AllCharsOK includes an appropriate doc string specification.

2.3 Frequency

A string is frequency-legal if it satisfies each of these rules:

F1. It has at most three M's.
F2. It has at most three C's.
F3. It has at most three X's.
F4. It has at most three I's.
F5. It has at most one D.
F6. It has at most one L.
F7. It has at most one V.

Thus, 'MM1984XVI' and 'IXIILDMVX' are frequency-legal strings while 'XXXX' and 'D3D' are not. (You may know that IIII is sometimes used for "4" on clock faces. Pay no attention to this. We are the Roman numeral authorities in this problem and that's final!)

Implement a function AllFreqsOK(R) that takes a non-zero-length string R and returns True if it is frequency-legal and returns False if it is not. (Your function should assume the length of its input is at least one; don't put in a test for the length being zero.)

Hint: you may find yourself using a conjunction of many Boolean expressions that is too long to fit in one line. To handle this, use parentheses to allow you to continue onto separate lines, like this:

```
return (number_buffalo > 1
        and deer == "playing"
        and antelope_playing
        and skies != "cloudy")
```

Make sure your implementation of AllFreqsOK includes an appropriate doc string specification.

2.4 Single Legal

It turns out that, due to certain ordering rules, MCMXIV is a valid Roman numeral but CIMVM is not. That is because an M cannot be preceded by an I or a V. The next set of rules explains "who can come before whom" in a Roman numeral. We say that a string R is single-legal if each of the following properties are satisfied:

S1. Every character in 'DLXVI' that occurs in R must come after the last occurrence of 'M' in R.
S2. Every character in 'LXVI' that occurs in R must come after the last occurrence of 'D' in R.
S3. Every character in 'LVI' that occurs in R must come after the last occurrence of 'C' in R .
S4. Every character in 'VI' that occurs in R must come after the last occurrence of 'L' in R.
S5. Every character in 'V' that occurs in R must come after the last occurrence of 'X' in R.

Noting the similarities in each of these conditions, it would be very handy to have available a function like this:

```
def SingleOK(c,s,R):

        """ Returns True if c is not in R,
        OR, if c is in R and not preceded by any character in s.
        Otherwise, False is returned.

        PreC: c is a character, s is a nonempty string, R is a nonempty string,
        and c is not in s.
         """
```

Indeed, if we had such a function then checking these rules for compliance would be very easy:

S1	is true if and only if	SingleOK('M','DLXVI',R)	is True
S2	is true if and only if	SingleOK('D','LXVI',R)	is True

S3	is true if and only if	SingleOK('C','LVI',R)	is True
S4	is true if and only if	SingleOK('L','VI',R)	is True
S5	is true if and only if	SingleOK('X','V',R)	is True

Note in the given module Roman.py that we have implemented a function AllSinglesOK(R) that determines whether or not the input string R is single-legal. For this pre-programmed function to work, you will have to implement SingleOK. We now offer some hints on how you might approach this Boolean challenge.

The string methods find and rfind can be used to look for first and last occurrences. We covered find

in lecture.

> If s1 and s2 are strings, then s1.find(s2) is an int whose value is the index of the first occurrence of s2 in s1. If there is no first occurrence, then the returned value is -1.

The string method rfind is similar:

> If s1 and s2 are strings, then s1.rfind(s2) is an int whose value is the index of the last occurrence of s2 in s1. If there is no last occurrence, then the returned value is -1.

Just to be clear, here is an example:

```
>>> s = 'axyzbxyzcxyzd'

>>> s.find('xyz')

1

>>> s.rfind('xyz')

9
```

Returning to the implementation of SingleOK, if c is not in R then there is very little to do. Just return the value True. If c is in R, then you will need a loop to oversee the checking of each character in s. You have to make sure that no character in s that also appears in R ever comes before the last occurrence of the character c in R.

Here are some clarifying examples:

> SingleOK('C','LVI','MDV') is True because there is no 'C' in MDV'.
> SingleOK('C','LVI','MCCV') is True because 'V' does not come before the last 'C'.
> SingleOK('C','LVI','MCVCV') is False because there is a 'V' before the last 'C'.

Make sure your implementation of SingleOK includes a doc string specification.

2.5 Double legal

We say that a string R is double legal if each of the following properties are satisfied:

D1 The string ’CM’ can occur at most once in R.
If ’CM’ does occur in R, then the ’C’ in ’CM’ is the first ’C’ in R.

D2 The string ’CD’ can occur at most once in R.
If ’CD’ does occur in R, then the ’C’ in ’CD’ is the first ’C’ in R.

D3 The string ’XC’ can occur at most once in R.
If ’XC’ does occur in R, then the ’X’ in ’XC’ is the first ’X’ in R.

D4 The string ’XL’ can occur at most once in R.
If ’XL’ does occur in R, then the ’X’ in ’XL’is the first ’X’ in R.

D5 The string ’IX’ can occur at most once in R.
If ’IX’ does occur in R, then the ’I’ in ’IX’ is the first ’I’ in R.

D6 The string ’IV’ can occur at most once in R.
If ’IV’ does occur in R, then the ’I’ in ’IV’ is the first ’I’ in R.

To make sure you understand these rules, consider D1.

D1 holds if R = ’MCMXC’ because there is only one occurrence of ’CM’ and the ’C’ in ’CM’ is the first ’C’ in the string.

D1 holds if R = ’MCX’ because there is no occurrence of ’CM’.

D1 does not hold if R = ’MCXCM’ because the ’C’ in ’CM’ is not the first occurrence of a ’C’

Given the similarity of D1-D6, it would be very handy to have available the following function:

```
def DoubleOK(s,R):

    """ Returns True if s is not in R or if s occurs once in R
    and the first occurrence of s[0] is the first occurrence of s.
    Otherwise the value False is returned.


    PreC: s is a length-2 string and R is a nonempty string.
    """
```

Indeed, we could easily check that D1-D6 hold by using DoubleOK:

D1 is true if and only if DoubleOK(’CD’,R) is True

D2 is true if and only if DoubleOK('CM',R) is True

D3 is true if and only if DoubleOK('XL',R) is True

D4 is true if and only if DoubleOK('XC',R) is True

D5 is true if and only if DoubleOK('IV',R) is True

D6 is true if and only if DoubleOK('IX',R) is True

The function AllDoublesOK given in Roman.py checks to see if a given string is double-legal by using DoubleOK in this way. For it to work, you will have to implement DoubleOK.

Make sure your implementation includes a doc string specification.

2.6 Computing the Value

A string R is a Roman numeral string if it is character-legal, frequency-legal, single-legal, and double-legal. Each character in a Roman numeral string has a numerical value: 'M' is 1000, 'D' is 500, 'C' is 100, 'L' is 50, 'X' is 10, 'V' is 5, an 'I' is 1.

The value of R is obtained by first adding up the values associated with each character to obtain the preliminary value, e.g.,

'MCMLXXXIV' ----> 1000 + 100 + 1000 + 50 + 10 + 10 + 10 + 1 + 5 = 2186

This results in an "overcount" because the value of 'C' in 'CM' is -100 and the value of 'I' in IV is -1. To correct for this we need to make an adjustment to the preliminary value:

'MCMLXXXIV' ----> 2186 - 200 - 2 = 1984

In general, adjustments to the preliminary value have to be made because:

If 'CM' occurs then this 'C' is really worth -100.
If 'CD' occurs then this 'C' is really worth -100.
If 'XC' occurs then this 'X' is really worth -10.
If 'XL' occurs then this 'X' is really worth -10.
If 'IX' occurs then this 'I' is really worth -1.
If 'IV' occurs then this 'I' is really worth -1.

Implement a function Value(R) that takes a Roman numeral string and returns an int that is its value. (You don't have to check whether the input is a valid Roman numeral string, so don't put in a test for this.)

Note that if you have long sequences of additions that make some lines very long, you can use the same trick as mentioned above: starting an expression with a left parenthesis allows Python to understand that you are continuing on to another line until the corresponding right parenthesis is encountered.

Make sure your implementation includes a doc string specification.

Submit your finished version of the module Roman.py to Canvas. It should include implementations of these functions:

- AllCharsOK **(12 points)**
- AllFreqsOK **(12 points)**
- SingleOK
- AllSinglesOK **(12 points)**
- DoubleOK
- AllDoublesOK **(12 points)**
- Value **(12 points)**

BTW. Even after all this work our Roman numeral system isn't perfect. By our definitions, 'IVII' is a legal Roman numeral string and its value is 1 + 5 + 1 + 1 − 2 = 6.

2.7 Sample outputs

Below are some sample outputs expected from a correctly working program. **Your programs should output the messages and results, values as shown below**.

```
Enter a Roman Numeral String  (do not surround with quotes): IX
AllCharsOK(R)   is  True
AllFreqsOK(R)   is  True
AllSinglesOK(R) is  True
AllDoublesOK(R) is  True
Value = 9

Enter a Roman Numeral String  (do not surround with quotes): CCCXL
AllCharsOK(R)   is  True
AllFreqsOK(R)   is  True
AllSinglesOK(R) is  True
AllDoublesOK(R) is  True
Value = 340

Enter a Roman Numeral String  (do not surround with quotes): MMMDCCCLXXXVIII
AllCharsOK(R)   is  True
AllFreqsOK(R)   is  True
AllSinglesOK(R) is  True
AllDoublesOK(R) is  True
Value = 3888

Enter a Roman Numeral String  (do not surround with quotes): 3X
AllCharsOK(R)   is  False
```

```
AllFreqsOK(R)   is  True
AllSinglesOK(R) is  True
AllDoublesOK(R) is  True
Not a valid Roman numeral string.

Enter a Roman Numeral String  (do not surround with quotes): XXXX
AllCharsOK(R)   is  True
AllFreqsOK(R)   is  False
AllSinglesOK(R) is  True
AllDoublesOK(R) is  True
Not a valid Roman numeral string.

AllCharsOK(R)   is  True
AllFreqsOK(R)   is  False
AllSinglesOK(R) is  False
AllDoublesOK(R) is  True
Not a valid Roman numeral string.

Enter a Roman Numeral String  (do not surround with quotes): MCXCM
AllCharsOK(R)   is  True
AllFreqsOK(R)   is  True
AllSinglesOK(R) is  False
AllDoublesOK(R) is  False
Not a valid Roman numeral string.

Enter a Roman Numeral String  (do not surround with quotes): MCMLXXXIV
AllCharsOK(R)   is  True
AllFreqsOK(R)   is  True
AllSinglesOK(R) is  True
AllDoublesOK(R) is  True
Value = 1984
```

What to submit?

1. Prepare a README.txt file containing the following:

   a) Include a quick summary of how you run your program money_dispenser.py.
   b) Include a quick summary of how you run your program Roman.py.
   c) If any of the programs is not working, include what is the issue in your opinion and how would you fix it if you had more time?

2. Submit your files money_dispenser.py, Roman.py, README.txt inside a single zip file on Canvas.

Additional directions

Following directions apply to both the problems above. Negative numbers indicate the penalty to be applied on your final score for a problem/direction if the directions are not followed.

1. Start your programs with a docstring (comment) summarizing what it is doing, what are the inputs, and the expected output. (-5 points)
2. At the beginning, as three comments, include your name (both teammates if applicable), date, and your email ids. (-5 points)
3. For every variable defined in your program, include a brief comment alongside explaining what it stores. (-5 points)
4. README.txt file needs to be submitted as described above. (-10 points)
5. Submit all the files (.py and .txt files) as a single zip folder/file on Canvas. (-5 points)
6. No new submissions, code files can be accepted after the deadline. Please double check and ensure that you are submitting everything needed to run your programs, and the code files are the best versions you want evaluated. Only the material submitted in the zip file uploaded on Canvas before the deadline shall be graded.