CS1110 Spring 2016

Assignment 3: Due Friday Mar 4 at 6pm on CMS

Updates in orange/red; last update 12:10pm Sun Feb 28

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 in CMS (this requires an invitation issued by one of you on CMS and the other of you accepting it on CMS) and then submit your work as a group.

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 a member of the teaching staff before the deadline.

Topics. Using the string methods `count` and `find`. Boolean-valued functions. Helper functions. Iterating through a string with `for`. Iteration in graphics using `for`. The assignment is based on lectures through February 25 and Lab 4.

1 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 satified, 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 $-100 + 1000 - 10 + 100 - 1 + 10 = 999$.

1.1 Getting Set Up

Start by downloading the module `Roman.py` from the course 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.

1.2 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[1] `R` as input and returns `True` if `R` is a character-legal string and returns `False` if it is not. 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.

[1] Your function should assume the length of its input is at least one; don't put in a test for the length being zero.

1.3 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[2].

Implement a function `AllFreqsOK(R)` that takes a non-zero-length string[3] `R` and returns `True` if it is frequency-legal and returns `False` if it is not.

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.

1.4 Single Legal

It turns out that, due to certain ordering rules, `MCMXIV` is 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:

[2]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!

[3]Your function should assume the length of its input is at least one; don't put in a test for the length being zero.

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.

1.5 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.

1.6 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[4] and returns an `int` that is its value.

[4]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 the corresponding right parenthesis is encountered.

Make sure your implementation includes a doc string specification.

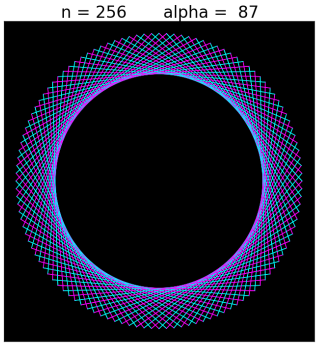
Submit your finished version of the module `Roman.py` to CMS. It should include implementations of these functions:

```
AllCharsOK
AllFreqsOK
SingleOK
AllSinglesOK
DoubleOK
AllDoublesOK
Value
```

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 $1 + 5 + 1 + 1 - 2 = 6$.

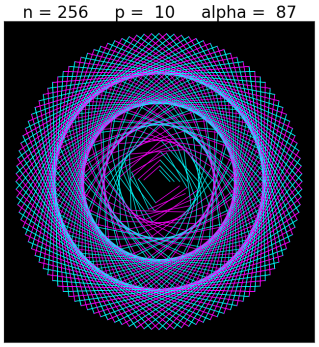
2 Rings and Spirals

In this problem you write a procedure that can draw a *ring* like this



This will involve a `for`-loop that oversees the drawing of lots of line segments. (Note that `SimpleGraphics` has a procedure for drawing line segments.) Each line segment has its endpoints on a given circle. The "end" of one line segment marks the beginning of the next line segment. We will call these line segments *spokes*.

You will also write a procedure that draws a sequence of nested rings thereby obtaining what we will call a *spiral*, e.g.,



This will also involve a loop. Each time through the loop body a new ring will be added to the window.

2.1 Getting Set Up

Download the module `Spiral.py` from the course website. It is set up for you to develop the two procedures that are required for this part of the assignment.

Put (a copy of) the by-this-time-familiar file `SimpleGraphics.py` in the same directory.

2.2 Drawing a Ring

The geometry of a ring is defined by r (the radius), n (the number of spokes), and α (the spoke angle in degrees). The endpoints of the spokes are systematically located on the circle $x^2 + y^2 = r^2$. In particular, for $k = 0, 1, 2, \ldots, n-1$, the endpoints of spoke k are

$$\left(r\cos\left(\frac{\pi k\alpha}{180}\right), r\sin\left(\frac{\pi k\alpha}{180}\right)\right) \qquad \text{and} \qquad \left(r\cos\left(\frac{\pi (k+1)\alpha}{180}\right), r\sin\left(\frac{\pi (k+1)\alpha}{180}\right)\right).$$

Implement a procedure

```
DrawRing(n,r,alpha,c1,c2)
```

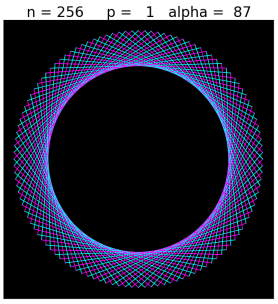
that draws a radius-r ring made up of `n` spokes. The parameters `c1` and `c2` should be rgb lists. The ~~odd~~even-indexed spokes should have color `c1` and the ~~even~~odd-indexed spokes should have color `c2`. (To be clear, the first spoke has index 0, and should have color c1.) The parameter `n` is a positive `int`, the parameter `alpha` is a positive `int`, and the parameter `r` is a positive `float`.

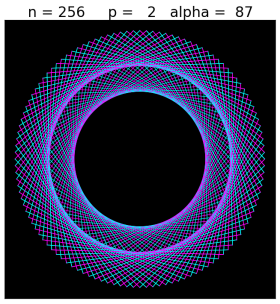
A suggestion: you can see a lot of terms in common in the endpoint recipes above. Why not store common values in a variable once, instead of computing the same thing more than once?

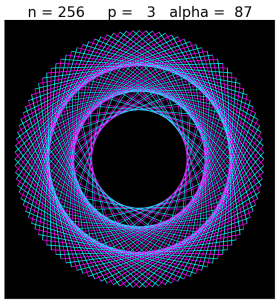
Make sure you fully specify `DrawRing`. You are not allowed to use lists in this problem. Remember that `DrawRing` is a procedure–it does not return any values. Your implementation of `DrawRing` needs to work before you proceed to the next part of the problem.

2.3 Drawing a Spiral

A spiral is a sequence of rings. Here are examples of a 1-ring, 2-ring, and 3-ring spiral:







Spiral geometry is defined by a positive integer p (the number of rings), an integer n that is a power of two (the number of spokes in the outermost ring), a positive number r (the radius of the outermost ring), and a positive integer α (the spoke angle for every ring). The radii of the inner rings and the number of spokes that they have are defined as follows:

> If ρ is the radius of a given ring in the spiral, then $\rho\cos(\pi\alpha/360)$ is the radius of its inner ring neighbor.
>
> If m is the number of spokes in a given ring in the spiral, then its inner ring neighbor has $m/2$ spokes.

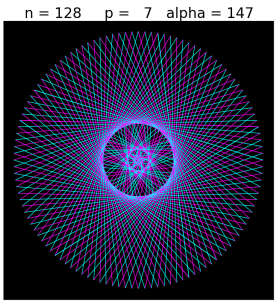
Implement a procedure

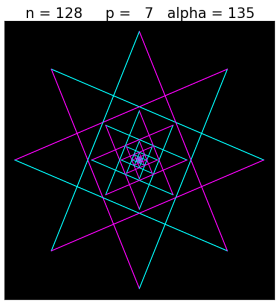
```
DrawSpiral(n,r,alpha,c1,c2,p)
```

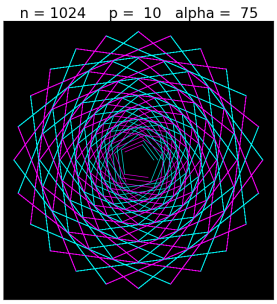
that draws a spiral. The parameter `n` is a positive `int` and a power of two. It specifies the number of spokes in the outermost ring. The parameter `r` is a positive `float` that specifies the radius of the outermost ring. The parameter `alpha` is a positive `int` that specifies the spoke angle of every ring. The parameters `c1` and `c2` should be rgb lists. Every ~~odd~~even-indexed spoke in the spiral should have color `c1` and every ~~even~~odd-indexed spoke in the spiral should have color `c2`. (The first spoke has index 0 and should have color c1.) The parameter `p` is a positive `int` that specifies the number of rings in the spiral.

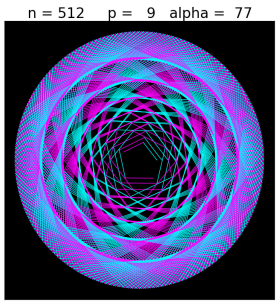
Submit your finished version of `Spiral.py` to CMS. It should include your implementations of `DrawRing` and `DrawSpiral`. Make sure that both are fully specified.

Here are some interesting examples upon which you can test your implementation:









3 Submission checklist and CMS notes

1. Make sure your submitted `.py` files begin with the header comments listing:
   (a) The name of the file
   (b) The name(s) and netid(s) of the person or group submitting the file
   (c) The date the file was finished

   If there were other people who contributed to your thought processes in developing your code, it is a courtesy to mention them as well.
2. We strongly encourage you to use print statements to test and debug your code, but remove such print statements from your code before uploading to CMS.
3. Do not hit "reload" on the CMS assignment submission page after the submission deadline passes: this will trigger another upload of your files, which can result in your submission being counted as late.
4. If you submit earlier than two hours before the submission deadline, CMS allows you to download your files to check that they are what you think you submitted. We thus strongly suggest that you submit a version early and do this check. (If you want to make changes, no problem: CMS allows you to overwrite an older version by uploading a new version.)