CS1110 Spring 2016

Assignment 1: Due on CMS Friday Feb 12 at 6pm

Updates in orange; last update 6:30pm Wed Feb 3

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 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.

Objectives. Mastering the assignment statement and conditional execution. Distinguishing between types and values. Practicing with strings and string slicing. Using `print`, `str`, `int`, `float`, `len`, `import`, `input`, and `raw_input`. You will get experience bridging the gap from math formula to Python code. You will get practice processing strings according to given rules. Finally, your ability to manipulate files and directories and the Komodo editor will be enhanced. The assignment is based on Lectures 1,2, and 3 and Labs 1 and 2.

1 The Golden Ratio

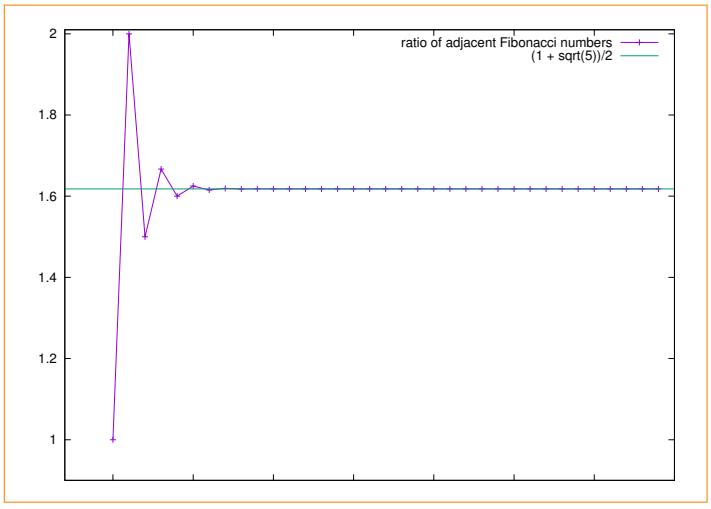
Given a positive integer n, the n-th Fibonacci number is given by

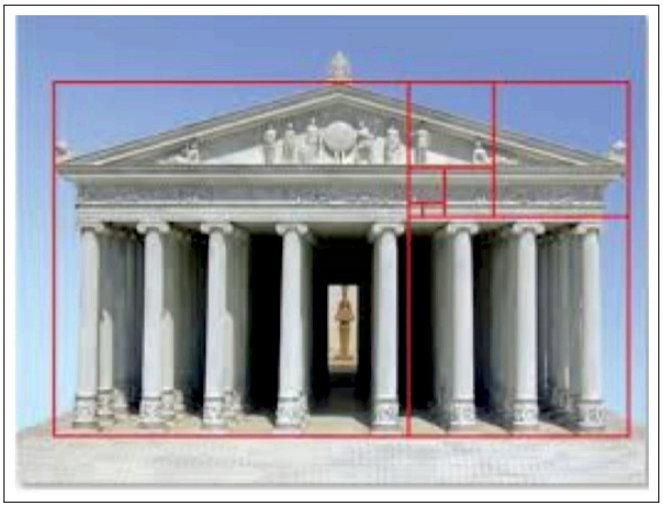
$$f_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right).$$

Despite the complexity of the right hand side, the Fibonacci numbers[1] are integers:

n	1	2	3	4	5	6	7	8	9	10	...	20	21	22	...
f_n	1	1	2	3	5	8	13	21	34	55		6765	10946	17711	

The ratios $r_n = f_{n+1}/f_n$ are interesting, and shown on the left below:





As n increases, the ratios look more and more like the *golden ratio* $\phi = (1+\sqrt{5})/2$. (It's practically impossible to see, but the y values are still changing in the fifth decimal place halfway through the sequence shown above.) The rectangle whose length-to-width ratio equals ϕ is (according to the Ancient Greeks) the most visually appealing of all rectangles, as hinted at above right.

[1] Aside: each Fibonacci number is the sum of its two predecessors, a fact that we will play with when we get to the topic of iteration.

Your first task. Write a Python script `Golden.py` that solicits a positive integer `n` using the `input` command and then produces a two-line table. The first line should display n, f_n and r_n. The second line should display $n+1$, f_{n+1}, and r_{n+1}. For example, if the input value for n is 35, then this should be displayed:

```
35     9227465    1.618033988749890
36    14930352    1.618033988749897
```

So that the data is nicely aligned and informative, use the `%2d` format for n, the `%10d` format for f_n, and the `%18.15f` format for r_n. The number of blanks in between the columns is not important as long as the six numbers are neatly displayed. Since square roots have to be computed, your code will have to import that function from the `math` module.

Problems arise if the input value for n is too big. This will be explained later on in the course. So to keep things simple in this assignment we require $n \leq 35$. The prompt in the `input` statement that solicits n should make this restriction (and that the input be a positive integer) clear to the user.

Remove any print statements that were part of your debugging strategy, and submit your implementation of `Golden.py` to CMS.

2 LOL? FWM!

This problem lets you probe how quickly you super-text-ers can enter a specified length-3 string. Code from the `random` module is used to generate a test string. The test string is displayed and code from the `datetime` module is used to time how long it takes for you to "re-enter" the test string.

To get started, download the skeleton script `LOL.py` that is available on the assignments page of the course website. We reproduce it here for your convenience:

The only thing you need to know about the workings of this script is what it assigns to the variables `S1`, `S2`, `t1` and `t2`. This is what happens:

(a) the value of `S1` is the random length-3 test string.

(b) the value of `S2` is your response (a string). It can be anything.

(c) the value of `t1` is the initial timestamp (a string). It encodes the exact time when you are first shown the test string.

(d) the value of `t2` is the final timestamp (a string). It encodes the exact time when you finish typing in your response.

Play with this script so that you get a sense of what it does. You will first be prompted as follows:

```
Press the return key when you are ready.
```

As soon as you do that the test string is generated and displayed and then you are asked to respond:

```
                          The test string: tpq
Enter the test string as fast as you can:
```

At this point the "clock is ticking." You enter your response and after that, the two timestamp strings are displayed:

```
      The test string: tpq
Enter the test string: tpq

2016-02-01 13:48:44.453000
2016-02-01 13:48:46.965000
```

The timestamp strings encode a date and time. All we care about is the "seconds slice" which begins in position 17 and extends to the end of the string. In this example, the subtraction

$$46.965000 - 44.453000 = 2.512000 \tag{1}$$

reveals that your response took 2.512 seconds. In this problem you are to add code to the end of `LOL.py` so that it exhibits this behavior:

1. It prints the message `Correct response.` if your response is exactly the same as the test string.
2. It prints the message `There is a character mismatch in your response.` if your response is length-3 but does not agree with the test string.
3. It prints the message `Your response has the wrong number of characters.` if your response is not a length-3 string.
4. It prints the elapsed time (in seconds) to three decimal places using information in the two time stamps.

As you organize the processing of (1), (2), and (3) keep in mind that the order of operations in computing is very important.

Regarding the elapsed time, it is not always as simple as the subtraction in equation (1) suggests. Suppose we have these two timestamps:

```
2016-02-01 13:48:44.453000
2016-02-01 13:49:06.965000
```

Elapsed time is NOT given by 6.965 - 44.452. Figure out how to deal with this situation so that your finished implementation of `LOL.py` reports the correct elapsed time. Assume that the elapsed time is < 60 seconds. This means that you don't have to worry about what your program does if someone takes more than a minute to answer.

Here is a sample dialog that indicates how the "correctness" output line and the elapsed time output line should be formatted:

```
                          The test string: xwo
Enter the test string as fast as you can: xwoo

2016-02-01 15:05:10.664000
2016-02-01 15:05:14.939000

Entered the wrong number of characters!
Elapsed Time =  4.275  seconds
```

Be sure to comment your code. Submit the finished version of `LOL.py` to CMS. It is important that you remove all print statements that were part of your debugging strategy.

3 Slash-Date to Dash-Date

There are several ways that you can encode a date as a string:

```
'7/4/2016'    '07-04-2016'     '7JUL16'      'July 4, 2016'
```

In this problem you are to write a script `Slash2Dash.py` that uses keyboard input to solicit a date string in slash format and then displays the equivalent date in dash format. Here are some defining examples:

Example	Slash Format	Dash Format
1	'7/4/2016'	'07-04-2016'
2	'12/23/15'	'12-23-2015'

3	'12/23/16'	'12-23-2016'
4	'12/23/17'	'12-23-1917'
5	'12/23/18'	'12-23-1918'
6	'12/23/2017'	'12-23-2017'
7	'13/42/0001'	'13-42-0001'
8	'2/29/2015'	'02-29-2015'

We need rules that define what it takes to be a slash-date string and rules that tell us how to produce the equivalent dash-date string. Here we go. A string is a *slash-date string* if

S1. Each of its characters is either a digit or a slash `'/'`.

S2. It contains exactly two slashes.

S3. There are either one or two digits before the first slash. These digits define the *month slice.*

S4. There are either one or two digits in between the two slashes. These digits define the *day slice.*

S5. There are either two or four digits after the second slash. These digits define the *year slice.*

Notice that a string can be a dash-date string without encoding an actual date. See Examples 7 and 8. The definition of a *dash-date string* is similar:

D1. Each of its characters is either a digit or a dash `'-'`.

D2. It contains exactly two dashes.

D3. There are two digits before the first dash. These digits define the *month slice.*

D4. There are two digits in between the two dashes. These digits define the *day slice.*

D5. There are four digits after the second dash. These digits define the *year slice.*

Next, we need rules that tell us how to convert a slash-date string to an equivalent dash-date string:

Month Conversion. The month slice in the dash-date string is the same as the month slice in the slash-date string *unless* the latter consists of a single digit. In that case the former is obtained by concatenating `'0'` to the front of the latter. Thus, in Example 1 `'7'` becomes `'07'`.

Day Conversion. The day slice in the dash-date string is the same as the day slice in the slash-date string *unless* the latter consists of a single digit. In that case the former is obtained by concatenating `'0'` to the front of the latter. See Example 1 where the `'4'` becomes `'04'`.

Year Conversion. The year slice in the dash-date string is the same as the year slice in the slash-date *unless* the latter consists of two digits. In that case the former is obtained by concatenating either `'19'` or `'20'` to the front of the latter. If the numerical value of the year slice in the ~~slice-date~~ slice-dash string is between and including 0 and 16, then concatenate `'20'` on to the front. (See Examples 1,2, and 3.) Otherwise, concatenate `'19'` as in Examples 4 and 5.

Use `raw_input` to acquire the input slash-date string. (You may want to play with the `ShowRawInput.py` demo associated with Lecture 2.) Your program can assume that the input string satisfies **S1**-**S5**. Later on in the course we will discuss graceful ways to handle "bad input."

Your implementation of `Slash2Dash.py` will require string slicing, string concatenation, and some boolean expressions. Your code will have to handle a couple of different situations. Do not try to figure out all these situations at once. Instead, you should develop your solution in stages; something like this:

Stage 1. Your initial version of `Slash2Dash` should use `raw_input` to store the incoming slash-date string in a variable (say `s`) and then compute the location of the two slashes. That is, write code that assigns the index of the first slash to a variable and the index of the second slash to another variable. Because of **S3** you know that either `s[1]` or `s[2]` houses the first slash. Figure out which. Reason similarly for the second slash. You might find it handy to involve the `len` function.

Use `print` statements to check your location variables for correctness. In particular, print out their values to affirm that the location of the first slash and the second slash are properly computed. Note that there are $8 = 2 \times 2 \times 2$ cases to check because the month, day, and year slices each have two possible lengths, e.g., `'7/4/67'`, `'7/14/67'`, `'12/1/67'`, `'12/14/67'`, `'7/4/1967'`, `'7/14/1967'`, `'12/1/1967'`, `'12/14/1967'`.

Stage 2. (Don't do this until you have completed **Stage 1.**) Extract the month slice from `s` and compute the month slice for the dash-date string. Look at the Month Conversion rule above. Use `print` statements to check your computed slice for correctness.

Use `print` statements to check your location variables for correctness. That is, don't write the rest of the program yet. Instead, just get it to where it is getting the month part write. To test your program at this stage, try the input '7' and see if the output of your program is '07'. (What other kind of input should you try to make sure the month part of your program is working?)

Stage 3. (Don't do this until you have completed **Stage 2.**) Extract the day slice from `s` and compute the day slice for the dash-date string. Look at the Day Conversion rule above. Use `print` statements to check your computed slice for correctness.

Stage 4. (Don't do this until you have completed **Stage 3.**) Extract the year slice from `s` and compute the year slice for the dash-date string. Look at the Year Conversion rule above. Use `print` statements to check your computed slice for correctness.

Stage 5. (Don't do this until you have completed **Stage 4.**) Using concatenation, assemble the dash-date string from what has been computed in **Stages 2**, **3**, and **4**. Print the final result.

Some sample dialogs:

```
Enter a valid slash-date:  7/4/76
07-04-1976

Enter a valid slash-date:  7/4/2020
07-04-2020

Enter a valid slash-date:  20/99/1976
20-99-1976
```

Thus, once you compute the dash-date string you should just print it "as is".

Be careful about blanks, since they're hard to see! For instance in this example,

```
Enter a valid slash-date:    7/4/76
```

the user has actually entered `'__7/4/76'`, and your program should not work on that input.

Submit `Slash2Dash.py` to CMS. It is important that you remove all print statements that were part of your debugging strategy.