CS 100 Programming I
Project 2

Revision Date: November 18, 2015

Preamble

You may develop your code anywhere, but you must ensure it runs correctly under a Linux distribution before submission.

Justified!

> *People go down bad paths and they make bad decisions, but it's always justified in their heads.* – Maisie Williams

XML is the state of the art way to 'mark up' text for various purposes. On such purpose is for formatting text. When text is formatted, it is (usually) made to look pleasing to the eye. XML stands for eXtensible Markup Language; text is marked up by enclosing it between a start-tag, which in its most simple form looks like `<xxxx>`, and and an end-tag, which would look like `</xxxx>`. The word `xxxx` in the tags is just a placeholder with any word replacing it, but the word used in the start-tag word must match the word used in the end-tag. The words themselves have no meaning; meaning is given to them by an XML processor. Your task is to write such an XML processor.

As an example, one might specify that a section of text be formatted as a *left-justified* paragraph:

```
<p>
Now is the time. Time for what?
Those first two sentences were
too short and they   need   to be made more
interesting
while at the same time made longer
unless they are supposed to be short and then they should remain the way
  they

are
but they could be made more interesting,
I suppose. THAT sentence kind of rambled, didn't it?
</p>
```

One would then write an XML processor to correctly interpret those tags and convert the text above into a nicely formatted, left-justified paragraph of text.

1 Program actions

You are to write a program to read a file of text that is marked up with simple XML tags and produce appropriate output. The tags and should then be interpreted by an XML processor as instructions to produce a left-justified paragraph composed of the text between the tags. Given the previous example, your processor should produce the following output:

```
Now is the time. Time for what? Those first two sentences
were too short and they need to be made more interesting
while at the same time made longer unless they are supposed
to be short and then they should remain the way they are but
they could be made more interesting, I suppose. THAT
sentence kind of rambled, didn't it?
```

You should fit as many words as possible on a line for a given width (see the next section for an explanation of width). A word is considered a contiguous span of non-whitespace characters.

If the example text appears between the tags <r> and </r>, then the text is to be combined into a right-justified paragraph:

```
   Now is the time. Time for what? Those first two sentences
    were too short and they need to be made more interesting
 while at the same time made longer unless they are supposed
to be short and then they should remain the way they are but
        they could be made more interesting, I suppose. THAT
                         sentence kind of rambled, didn't it?
```

Note that in both left justification mode and right-justification mode <r>, extra whitespace between words is eliminated.

Finally, your program should be able to center lines of text. In centering, there is no combination of lines, each line is centered individually. Any extra whitespace between words within a line is preserved.

Here's what the output of the original example should look like when enclosed withing <c> and </c> centering tags:

```
  Now is the time. Time for what? Those first two sentences
  were too short and they need to be made more interesting
 while at the same time made longer unless they are supposed
to be short and then they should remain the way they are but
    they could be made more interesting, I suppose. THAT
            sentence kind of rambled, didn't it?
12345678901234567890123456789012345678901234567890123456789012345678901234567890
```

Observe how the extra space around the word 'need' is preserved. Lines that are longer than the width are simply left-justified.

Lines of text that are not enclosed within start- and end-tags are written to the console as is. For example, if the input text is:

```
    this
       as
        is
```

then the output of your program should be:

```
this
   as
    is
```

There may be any number of start- and end-tag pairs in an input file.

Input / output

The name of the file containing the marked up text is to be passed as a command-line argument, as in:

```
python format.py xxxx
```

where `xxxx` is the name of the file containing the marked up text.

Optionally, a width can be supplied:

```
python format.py xxxx NN
```

A width value of 50 (i.e. `NN` above is replaced with 50) means that a left or right justified line should be no longer than 50 characters. For right justification, the last character in each line should be located at the width value. For centering, the middle of lines less than the width should be located at half the width value. The default width value (to be used if no width is given on the command line) is 60 characters.

The input files are free format. That is to say, words may appear anywhere on a line as may the start- and end-tags. However, the start- and end-tags are guaranteed to be on a line by themselves. Lines containing a tag are to be ignored with respect to the output.

Error Detection

You should detect the following errors:

- A start tag followed by another start tag without an intervening (and matching) end tag
- An start tag without a matching end tag
- An end tag without a matching start tag

You should also detect too many or two few command-line arguments.

If you detect and error, you should end your program immediately with an informative error message (i.e., give enough information in the error message so that the user of your program will be able to easily fix the problem).

Reading and writing to files

To read from a file, you may use the *Scanner* class, which you can read about here:

The Art and Craft of Programming, Chapter 10

and:

The Art and Craft of Programming, Chapter 14

and download here:

```
wget troll.cs.ua.edu/cs100/python/projects/scanner.py
```

The *Scanner* class has a *readline* function that works the same as Python's *readline* function. The Scanner's readline function does not strip leading and trailing whitespace.

The chapters given above also discuss writing to a file.

Program Organization

Each small task should be implemented as a function. There is much commonality between the left and right justification and centering tasks. Points will be deducted for duplicate code that could be pushed of into a function that is called from the former locations of the duplicate code.

Name your main file *format.py*.

Stepwise refinement

You should use stepwise refinement in implementing this project.

Level 0 Write a program that prints the single command line argument. The command line argument is the name of the file containing the input data.

Level 1 Write a program that creates a Scanner object for the input data file and prints out each token in the file using the Scanner's *readtoken* function.

Level 2 Write a program that creates a Scanner object for the input data file and prints out each line in the file using the Scanner's *readline* function.

Level 3 Same as Level 2, but prints a useful message if a line containing `<p>` or `<r>` is encountered.

Level 4 Write a program that prints the input file a line at a time except between `<p>` and `</p>` tags. Between those tags, the program prints out a token at a time.

Level 5 Like Level 4, but processes `<r>` and `</r>` the same as `<p>` and `</p>`.

Level 6 Like Level 5, but does not print the individual tokens. Instead, the program joins up tokens into a single line. When the line gets longer than max number of characters, it prints the line and start a new line.

Level 7 Like Level 6, but prints the line just before adding a token that would make the line longer than the max number of characters.

Level 8 Like Level 7, but places spaces at the front of a line before printing if two conditions are both true: the line is shorter than the max number of characters *and* the program is processing tokens between `<r>` and `</r>` tags.

Level 9 Like Level 8, but adds error checking and the ability to change max number of characters per line.

Write programs with the names *level0.py*, *level1.py*, *etc.* When you are ready to move to the next level, copy the previous level's program. For example:

```
cp level0.py level1.py
```

copies the code in *level0.py* and creates a new file named *level1.py*.

Instructors and mentoring students have been informed that before they can help you, you must demonstrate what levels you have completed.

Compliance testing

To make sure that you have implemented your program correctly, create a file named *test0* that has a paragraph of text in it.

You should then be able to run the following command:

```
python3 format.py test2
```

and see the following output:

```
Hello.
This is a paragraph.
                                                This is another.
And another.
Goodbye.
```

Adjust the contents of test0 until you get the desired output.

If your code does not produce exactly the above output, you will receive a zero for this assignment.

Submission Instructions

Change to the directory containing your assignment. Run the command:

```
submit cs100 xxxx project2
```

Replace `xxxx` with your instructor's name.