MCS 260 – Introduction to Computer Science – Fall 2020 – Emily Dumas

Project 3 – Due Fri Nov 6 at 6:00pm Central time

RECIPIENT

This project description was prepared for Anne Example. It must not be shared with or shown to anyone else except for the course staff (instructor and TAs).

1. GOALS

This project will test the following concepts:

- Reading, understanding, and modifying existing code
- Writing and using programs that read and write files (without any keyboard input)
- Creating modules
- Writing documentation

2. OVERVIEW

This document describes a computer program you must write in Python and submit to Gradescope.

For this project, the program will be a utility to convert text files to HTML, supporting special syntax for variable substitution and bullet lists.

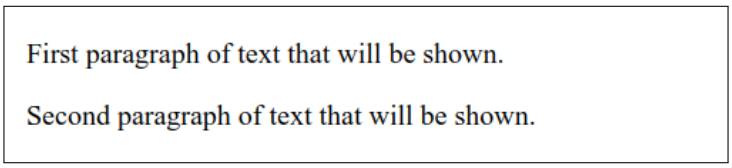
3. A BIT OF HTML

HTML or hypertext markup language is the language used to write documents that can be viewed in browsers such as Firefox, Chrome, and Safari. The basics of HTML (sufficient for this project) were discussed in Lecture 22 of MCS 260. This section reviews some of the concepts discussed in that lecture. On a first reading of the project description, you might skip this section if you found the lecture about HTML to be clear.

To discuss HTML it can be helpful to see a simple working example. Here is a very small HTML document:

```
<!DOCTYPE html>
<html>
<head><title>HTML document</title></head>
<body>
<p>First paragraph of text that will be shown.</p>
<p>Second paragraph

      of text that will be

shown.</p>
</body>
</html>
```

If this document is saved in a text file `example.html` and opened in a browser, it might look like this:



Notice that blank lines and indenting in the HTML file are ignored when it is rendered by the browser.

In this project, only a few aspects of HTML will be important, which we discuss these now. The variant of HTML we discuss here is called HTML5, and it is supported by all recent browser versions.

First, a HTML document is a text file that begins with an indicator that it contains HTML. The following line will accomplish this:

```
<!DOCTYPE html>
```

The rest of the HTML document consists of opening tags, closing tags, and text.

An opening tag is a name surrounded by < and >. For example:

- `<body>` is an opening body tag
- `<p>` is an opening p tag

A closing tag is a name preceded by </ and followed by >. For example:

- `</body>` is a closing body tag
- `</p>` is a closing p tag

Opening and closing tags of the same type form pairs like parentheses, breaking the HTML document into pieces with different logical functions.

Most of the document is contained inside a html tag. Everything after the html end tag is ignored.

Immedidately inside the html tag there are two tags: head and body. The section inside the head tag includes information about the document, such as its title. You won't need to worry about the head tag at all; a template will be provided that you will use for all HTML files you write.

The section inside of the body tag contains the body of the document, which is displayed in the browser window. This is the part of the HTML document that the program you create in this project will write.

There are three tags that can appear in the body of a HTML document that are important for this project:

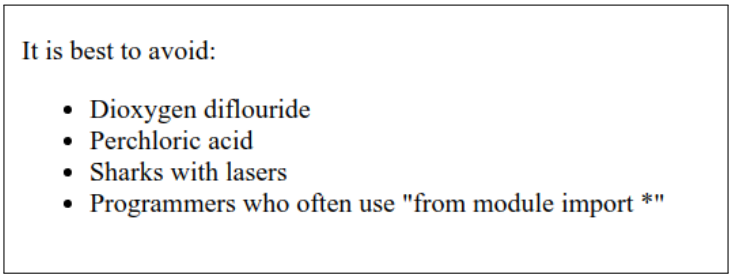
- `p` or paragraph tag: Contains a single paragraph of text. Line breaks inside this tag are ignored.
- `ul` or unordered list tag: Contains a bullet list of items.
- `li` of list item tag: The only tag allowed inside `ul`; contains an item in the list.

Here is an example of a document that uses all of these tags:

```
<!DOCTYPE html>
<html>
<head><title>HTML document</title></head>
<body>
<p>It is best to avoid:</p>
<ul>
  <li>Dioxygen diflouride</li>
  <li>Perchloric acid</li>
```

```
  <li>Sharks with lasers</li>
  <li>Programmers who often use &quot;from module import *&quot;</li>
</body>
</html>
```

This document will be displayed in a web browser as follows (though there may be slight differences between browsers and devices):



Note that indenting in the HTML file itself is optional.

4. Starter program

For this project you are provided with a starter program which performs a certain function. You must modify this program to add additional features. This section describes the starter program.

The starter program and related materials are available in the *starter pack*. You should have received a link to download the starter pack in the same email in which you received the link to this project description.

In the starter pack, the main program is `starter.py`, which expectes two command line arguments. The first is an input text filename, and the second is an output HTML filename.

This program reads the input text file and converts it to HTML, using blank lines in the input file to indicate breaks between paragraphs.

For example, if `in.txt` contains

```
The first line.
Still in the same paragraph.


New paragraph here.
And the new paragraph
is not yet
over.
But now it ends.
```

Then the command

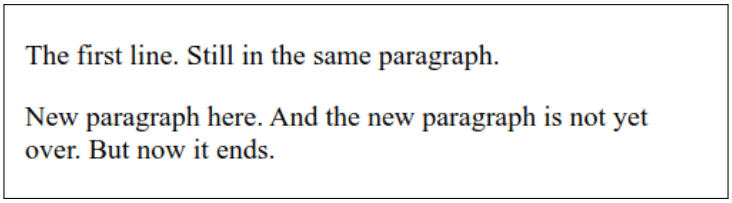
```
python starter.py in.txt out.html
```

will create a file `out.html` with the following content:

```
<!DOCTYPE html>
<html>
<head><title>HTML document</title></head>
<body>
```

```
<p>
The first line.
Still in the same paragraph.
</p>
<p>
New paragraph here.
And the new paragraph
is not yet
over.
But now it ends.
</p>
</body>
</html>
```

which may render in the browser as:



5. Required new features

Your main task in this project is to add three new features to the starter program: **variable expansion**, **comment lines**, and **bullet lists**. The resulting program must be called `template.py`.

This section describes these new features. If you prefer to see a complete description of what the final program needs to do, ignoring what is already in the starter program, see Section 6.

5.1. **Variable expansion.** Unlike the starter program, your submission will take three command line arguments: A variable filename, an input text filename, and an output HTML filename. For example, it might be called as

```
python template.py vars.dat in.template out.html
```

The first command line argument, the variable filename, is the name of a text file containing definitions of variables in the format `name=value`. Here name and value are arbitrary strings that do not contain the characters `=` or `$`. For example, `vars.dat` might contain:

```
recipient=Priscilla Overton
sender=Emily Dumas
```

In converting the input text file to HTML, your program will look for any instance of a variable name surrounded by dollar signs (e.g. `$recipient$`) and replace each with the corresponding variable value.

For example, if a line of `in.template` reads

```
Dear $recipient$,
```

then in the output HTML file it will be replaced with

```
Dear Priscilla Overton,
```

Note that it is permitted to have any number of variables on a line of text in the input text file, and they must all be replaced with their values as given in the variable file.

5.2. **Comment lines.** Any line of the input text file that begins with the character `#` should be considered as a comment line, and must be ignored. Comment lines do not indicate breaks between paragraphs, so for example

```
Welcome to the wonderful
# TODO: Determine if wonderful
22nd lecture of MCS 260!
```

contains a single paragraph.

If the character `#` appears anywhere other than the first character of a line, it is treated as part of a line of text and does not indicate a comment. (Note this is different from the way comments work in Python source code.)

5.3. **Bullet lists.** A line of the input text file that begins with the character `@` designates an item in a bullet list. The rest of the line (without `@`) should be placed inside an `li` tag and written to the output HTML file. If the previous line of the file did not begin with `@`, then any currently-open `p` tag should be closed, and an opening `ul` tag should be written before the `li` tag.

While a `ul` tag is open, reading a line that does not begin with `@` or `#` ends the current bullet list, and results in a closing `ul` tag being written.

Thus, for example, the following input text file

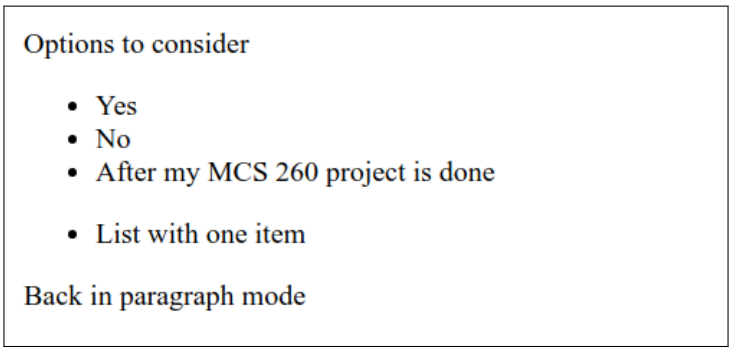
```
Options to consider
@ Yes
@ No
# TODO: rephrase the next one
@ After my MCS 260 project is done

@ List with one item
Back in paragraph mode
```

generates the following HTML:

```
<!DOCTYPE html>
<html>
<head><title>HTML document</title></head>
<body>
<p>
Options to consider
</p>
<ul>
<li>Yes</li>
<li>No</li>
<li>After my MCS 260 project is done</li>
</ul>
<ul>
<li>List with one item</li>
</ul>
<p>
```

```
Back in paragraph mode
</p>
</body>
</html>
```

which may display in the browser as follows:



6. Operational specification

This section describes how the final program must operate. In contrast, Section 5 described only the *differences* between the starter program and the final program.

It is probably best to first read Section 5 carefully, and then read this section when you want full details because you've started to consider how to implement the features.

The program is called `template.py`. Three command line arguments are required:

- `sys.argv[1]` — the variable filename
- `sys.argv[2]` — the input filename
- `sys.argv[3]` — the output filename

The variable filename and input filename must both refer to files that exist and can be read by the program. The output filename must refer to a file that can be opened for writing by the program.

The following operations are performed.

First, the variable file is read and each line is interpreted as the definition of a variable. The text before `=` on a line is the name of the variable, and the text after `=` is its value. Collectively, the data read in this step are called the *templating variables*. The variable file is closed.

Next, the output file is opened for writing and the following HTML header is written to it:

```
<!DOCTYPE html>
<html>
<head><title>HTML document</title></head>
<body>
```

Now the input file is read and processed line-by-line as follows:

- A line that is not blank and which does not start with `#` or `@`: This is a regular line of text. If a `ul` tag is open in the output file, it is closed. If a `p` tag is not open in the output file, one is opened. The contents of this line are transformed by variable substitution (see Section 6.1) and then written to the output file.
- A line that is blank: If a `p` tag is open in the output file, it is closed. If a `ul` tag is open in the output file, it is closed.

- A line starting with `#`: The entire line is ignored; nothing is written to the output file.
- A line starting with `@`: If a `p` tag is open in the output file, it is closed. If a `ul` tag is *not* open in the output file, one is opened. A `li` tag is opened in the output file. The `@` character is removed from the input line, and the rest of the input line is transformed by variable substitution (see Section 6.1) and written to the output file. The `li` tag is closed in the output file.

Once the last line of the input file is processed, the following cleanup steps are performed:

- If a `p` tag is open in the output file, it is closed.
- If a `ul` tag is open in the output file, it is closed.

The following HTML footer is written to the output file:

```
</body>
</html>
```

The output file is closed and the program exits.

6.1. **Variable substitution.** Transformation by variable substitution means that if the name of a templating variable is surrounded by `$` characters, then the variable name and the `$` characters are replaced by the value of the templating variable.

For example, if there is a templating variable named `foo` with value `bar`, then variable substitution would change the string

```
The word $foo$ is often used in CS for an arbitrary name; I say $foo$ to that!
```

to

```
The word bar is often used in CS for an arbitrary name; I say bar to that!
```

6.2. **Error handling.** If `template.py` is called with the wrong number of command line arguments, it must print a helpful error message describing the correct number of arguments and their functions.

The program `template.py` does not need to handle any other type of error. For example, it is acceptable for the program to exit with an uncaught exception if any of the input files is inaccessible, or if opening the output file fails.

If the output HTML filename specified on the command line already exists, this is not an error. The file should be opened for writing (overwriting any existing contents).

6.3. **Efficiency.** Your program must be able to process a variable and input file of up to 50 lines and write output in less than 5 seconds. This is an extraordinarily generous time budget, as a typical solution will take just a few hundredths of a second. It is almost certain that you will not need to pay attention to the speed of any operation in your program.

However, if your program is slow enough, then the autograder will run out of time in testing your program. In this case you will not receive credit. It should be clear in your local testing of the program whether it is slow enough for this to be an issue.

7. Source code specification

This section describes how your program must be written and submitted.

7.1. **Files.** You must submit three files:

- `template.py`
- `varsub.py`
- `README.txt`

In Gradescope, note that you can either submit a `.zip` archive containing all three files, or you can submit them one by one using the upload dialog box. The upload interface allows you to add more files before submitting, and each time you submit you need to include all three.

The file `template.py` must contain the main program described above.

The file `varsub.py` must contain a module imported by `template.py` that defines the following functions:

- `substitute(vars,s)` — Accepts a dictionary `vars` of templating variables and a string `s` and replaces references to templating variables in `s` with the values of those variables in `vars`, returning the result.

The main program `template.py` must use both of these functions and must not duplicate their functionality elsewhere.

The file `README.txt` must contain a description of how to use the program, written in your own words, which would allow someone who had not read this specification to understand what it does and how to use it. The contents of this file will be evaluated during code review. To receive full credit, `README.txt` must be written in full sentences, must be assume an audience that has no familiarity with your program or the project description, and must describe both the function of the program and an example of how to use it.

7.2. **General requirements.** The program must be able to run with Python 3.6 (which is consistent with all programming practices taught in this course).

The only modules you are allowed to import are `sys` and `os`.

Every source file must have a file-level docstring, and every function must have a descriptive docstring.

The file-level docstring must be the first line of the file. In `template.py` it must be followed by these comments:

- The following comment line, verbatim: `# MCS 260 Fall 2020 Project 3`
- A comment line consisting of your full name
- A comment line beginning with `# Declaration:`, and then containing a full sentence that says, in your own words, that you are the sole author of the program you are submitting and that you followed the rules from the course syllabus in preparing it. (This part can span multiple lines.)

Important: If the the declaration comment is missing or if the declaration is not true, no credit will be given for Project 3.

Here is an example of an acceptable start of `template.py`:

```
"""Text to HTML converter supporting bullet lists and variable replacement"""
# MCS 260 Fall 2020 Project 3
# Srinivasa Ramanujan
# Declaration: I, Srinivasa Ramanujan, am the sole author of this code, which
# was developed in accordance with the rules in the course syllabus.
```

Your source code will be evaluated on the basis of readability. You are expected to use descriptive variable names for the most important variables (e.g. `credit` as opposed to `c`). Single-letter names are permissible for the variable of a for loop.

Comments should be included whenever the intent of a line is not immediately apparent. Comments on every line would be excessive, but it is expected that this project will involve at least 5 lines that contain descriptive comments. Judging the correct comment density involves an element of subjectivity. If you are unsure about whether your code has enough comments, just ask. (You can ask about this by submitting your code to Gradescope and then emailing the instructor, or by including source code directly in an email to the instructor.)

The starter program includes a few comments that will not be accurate or relevant in your final program. These must be deleted. There will be a deduction for any inaccurate or irrelevant comments in the source code you submit.

In the review of your source code, the ability of the reviewer to understand the way your code works is important. It is theoretically possible to write an extremely complicated program that meets the operational specifications but which is impossible for a human to understand. In the code review scoring, code that is very difficult to understand may be subject to a penalty. If you are unsure about the level of understandability of your code, ask.

8. Acceptable use policy and plagiarism

The program you submit will be run by an autograder as part of its evaluation. Submitting a program that attempts to sabotage, circumvent, or examine the autograder, or which is designed to bypass course policies in any other way, is a violation of the ACCC acceptable use policy.

Giving or receiving aid to other students on projects in MCS 260 is prohibited. Submitting code of which you are not the sole author (for example, adapting work of another student or from a textbook or online resource) constitutes plagiarism.

Violations of policies described in this section will be reported to the Dean of Students for disciplinary proceedings.

9. Suggestions

9.1. **Read this document and ask questions early.** Avoid putting off working on the project to the last few days. The earlier you work on it enough to have questions, the more likely you are to get answers to those questions while there is still time to use the answers. Office hours tend to be very full in the last week. Email is always an option.

9.2. **Study the starter program.** You should understand how the starter program works before you try to modify it. Ask questions if anything is unclear.

9.3. **Suggested development steps.** After you've read the starter program, a good way to begin development of your program is:

- Copy `starter.py` to a new file called `template.py`. Add the required comments to the first few lines.
- Begin adding features to `template.py`, testing it frequently. (See below.)
- When you believe you are finished, review the source to make sure every comment is accurate and that no more comments are needed.
- Write `README.txt`.

9.4. **How to test.** We strongly recommend that you develop your project as follows: Break the work into the smallest pieces that you can add to a working program to produce another working program. (You can't test your work until the program is functional again!) For each of these pieces:

(1) Make a backup copy of the source code in another directory.
(2) Edit `template.py` or other source file to add the feature you are working on.
(3) Test the program locally, i.e. using the Python interpreter on your computer. Go back to step (2) and make changes if it does not work.
(4) Submit the program to the autograder and review the report. Some failures may be expected (e.g. if they involve features you haven't added yet) but look carefully for unexpected failures. Return to step (2) if any such failures are found.

9.5. **Start working early and submit often.** It is important to spread your work on this complicated project over a long period of time. If you haven't submitted any code to the autograder by the time there are **8 days left** before the deadline, then you are seriously behind. If you begin working on the project during the week it is due, you will be disastrously behind. Please plan ahead to avoid these possibilities, and understand that starting late will mean that course staff are less able to help you if you get stuck.

9.6. **Browsers are forgiving, but the autograder is not.** If you open a HTML output file from your program in the browser and it displays as you expect, that does not necessarily mean it followed the specifications in this document. Browsers are very tolerant of bad HTML, and HTML itself makes things like end tags optional in many cases.

To receive full credit, your program must precisely adhere to the specifications from Section 6. In particular it must create matching pairs of start and end tags.

10. Examples in the starter pack

The starter pack contains several examples of input and output of the starter program in the subdirectory `starter-examples`. See `README.txt` in this directory for a guide to using them. These input/output examples should work with the start program right away, and should not work after you've started to make changes to it.

The starter pack also contains several examples of input and output for the final program in the subdirectory `template-examples`. See `README.txt` in this directory for a guide to using them. These input/output examples should work when your program is complete. Some of them may work at an earlier stage of your project development. Comparing your project's behavior in these examples to the expected output is a good way to test your code locally. Furthermore, reading these examples of input and expected output is a good way to test your understanding of the project specifications.

11. Revision history

- 2020-10-15 Initial publication.