Homework 5: Website Generator

(Deadline as per Canvas)

This homework deals with the following topics:

- Reading and writing files
- Scraping and parsing information from a text file
- Very basic HTML
- Unit testing

General Problem Specification

The basic skeleton of a website is an HTML page. This HTML page is a text file with a certain format. Taking advantage of this fact, one can take an HTML template and create multiple pages with different values stored. This is exactly what we are going to do in this homework.

Many websites use these kinds of scripts to mass generate HTML pages from databases. We will use a sample text file as our 'database'.

Our goal will be to take a resume in a simplistic text file and convert it into an HTML file that can be displayed by a web browser.

What is HTML?

You do not need to know much HTML to do this assignment. You can do the assignment by just understanding that HTML is the language that your browser interprets to display the page. It is a tag-based language where each set of tags provides some basic information for how the browser renders the text that the tags enclose.

For example, `<h1>The Beatles</h1>` will be rendered by your browser with "The Beatles" as a heading with a large font. `<h1>` indicates the beginning of the heading and `</h1>` indicates the ending of the heading. An HTML webpage is typically divided into a head section and a body section. We have provided you with a basic website template. We want you to retain the head section and write only the body section via your Python program.

For more details about HTML, your best resource will be to use the w3schools website which can be found here: www.w3schools.com. This website has a ton of information and provides all of the common HTML you'll need to know for this assignment.

Input Test File

We will read a simple text file that is supposed to represent a student's resume. The resume has some key points but can be somewhat unstructured. In particular, the order of some of the information will definitely be different for different people.

The following are the things that you definitely DO know about the resume:

- Every resume will have a name, which will be written at the top. The top line in the text file will contain just the name.
- There will be a line in the file, which contains an email address. It will be a single line with just the email address and nothing else.
- Every resume will have a list of projects. Projects are listed line-by-line below a heading called "Projects". An example of what you might see in a resume file is something like this:

```
Projects
Worked on big data to turn it into bigger data
Applied deep learning to learn how to boil water
Styled web pages with blink tags.
Washed cars …
----------
```

- The list of projects ends with a single line that looks like '----------'. That is, it will have *at least* 10 minus signs. While this is an odd formatting requirement, this will actually make the assignment easier for you.
- Every resume will have a list of Courses. Courses are listed like:

```
Courses - CIT590, AB120
```

OR

```
Courses :- Pottery, Making money by lying
```

- The formatting for Courses will be the word "Courses" followed by some amount of punctuation, followed by the comma-separated list of courses.
- Your program should be able to look at the above example and then extract the

courses without including the '-' sign or the ':-'. Note that any type of punctuation could be between "Courses" and the list of courses.

Functions for Parsing the File

At the very least, you need to write one function for each piece of information that you want to extract from the text file. Contrary to previous homework assignments, in this assignment we will not provide you with a strict outline for the functions or what arguments to pass to them.

When we grade, we'll look at how modular your code is and how you decided to break up the functionality into separate functions. We'll also look at how you named the functions, what arguments they take, and how well you unit test the functions.

Here's a basic breakdown of the functionality required to read the file into memory, parse each section of the file to extract the relevant information, and write the final HTML-formatted information to a new file. You should be writing to a file, not appending.

Reading the File

- Since the resume file is pretty small, write a function that reads the file and stores it in memory as a list of lines.
- Then, you can use list and string manipulations to do all of the other necessary work.
- You should not prompt the user for a filename (or any other information). You can rely on the provided code which includes hardcoded resume filenames to read.

Detecting the Name

- Detect and return the name by extracting the first line.
- The one extra thing we want you to do, just for practice, is if the first character in the name string is not an uppercase letter (capital 'A' through 'Z'), consider the name invalid and ignore it. In this case, use 'Invalid Name' as the user's name.
- For example:
  `Brandon Krakowsky` is a valid name
  `brandon Krakowsky` is not a valid name, so your output html file will display 'Invalid Name' instead
- Another thing to note is that the name on the first line could have leading or trailing

whitespace, which you will need to remove.

- Note: Do not use the *istitle()* function in Python. This returns True if ALL words in a text start with an upper case letter, AND the rest of the characters in each word are lower case letters, otherwise False. This function will incorrectly identify a name like `Edward jones` as being an invalid name, when it's actually valid.

Detecting the Email

- Detect and return the email address by looking for a line that has the '@' character.
- For an email to be valid:
    - The last four characters of the email need to be either '.com' or '.edu'.
    - The email contains a lowercase English character after the '@'.
    - There should be no digits or numbers in the email address.
- These rules will accommodate `lbrandon@wharton.upenn.edu` but will not accommodate `lbrandon@python.org` or `lbrandon2@wharton.upenn.edu`
- For example:

  `lbrandon@wharton.upenn.edu` is a valid email

  `lbrandon@wharton2.upenn.com` is not a valid email

  `lbrandon2@wharton.upenn.com` is also not a valid email
- The email string could have leading or trailing whitespace, which will need to be stripped.
- We are fully aware that these rules are inadequate. However, we want you to use these rules and only these rules.
- If an email string is invalid based on the given rules, consider the email address to be missing. This means your function should return an empty string and your output resume file will not display an email address.
- DO NOT GOOGLE FOR A FUNCTION FOR THIS. Googling for solutions to your homework is an act of academic dishonesty and in this particular case, you will get solutions involving crazy regular expressions, which is a topic we haven't yet discussed in class. (In general, your code should never involve a topic that we have not discussed in class.). Plus, you can easily achieve the required functionality without the use of a regular expression.

Detecting the Courses

- Detect and return the courses as a list by looking for the word “Courses” in the list and then extract the line that contains that word.
- Then make sure you extract the correct courses. In particular, any random punctuation after the word “Courses” and before the first actual course needs to be ignored.
- You are allowed to assume that every course begins with a letter of the English alphabet.
- Note that the word “Courses”, the random punctuation, or individual courses in the list could have leading or trailing whitespace that needs removed.

Detecting the Projects

- Detect and return the projects as a list by looking for the word “Projects” in the list.
- Each subsequent line is a project, until you hit a line that contains ‘----------’. This is NOT an underscore. It is (at least) ten minus signs put together. You have reached the end of the projects section if and only if you see a line that has at least 10 minus signs, one after the other.
- If you detect a blank line between project descriptions, ignore that line.
- Each project could have leading or trailing whitespace that needs removed.

Writing the HTML

Once you have gathered all the pieces of information from the text file, we want you to programmatically write HTML. Here are the steps for that:

- Start by saving the file *resume_template.html* that is provided in the same directory as your code.
- Preview the file in a text editor that does HTML syntax highlighting (e.g. Sublime Text). Note that opening this page in your browser will give you a blank web page since the file only contains a header and an empty body.
- You are going to programmatically copy the HTML in *resume_template.html*, fill in the empty `<body>` with the resume content, then write the final HTML to a **new file** *resume.html*. You should be writing to a file, not appending to a file. (Note: you should not modify or overwrite the *resume_template.html* file. You should copy the information from this file, modify it as needed, and then write the modified information to a new file, *resume.html*. Think about how you can save the information from *resume_template.html* in program memory to help you accomplish this.)
- More specifically, your Python code will do the following:

o Open and read *resume_template.html*

o Read every line of HTML into program memory

o Remove the last 2 lines of HTML (the </body> and </html> lines). (You can delete these lines, and you will programmatically add them back later)

o Add all HTML-formatted resume content (as described below).

o Add the last 2 lines of HTML back in (the </body> and </html> lines).

o Write the final HTML to a new file *resume.html*

Why are we doing this? Because the HTML in *resume_template.html* looks something like the below, and we need to start by removing the last two lines of HTML (closing </body> and </html> tags) in order to insert the resume content in the correct location.

```
random header stuff
<html>
<head> lots of style rules we won’t worry about
</head>
<body>
</body>
</html>
```

We want to put our resume content in between the body tags to make it look like this:

```
random header stuff
<html>
<head> lots of style rules we won’t worry about </head>
<body>
HTML-formatted resume content goes here
</body>
</html>
```

In order to write proper HTML you will need to write the following helper function:

def surround_block(tag, text):

- This function surrounds the given text with the given HTML tag and returns the string
- For example, `surround_block(‘h1’, ‘The Beatles’)` would return

```
‘<h1>The Beatles</h1>’
```

You’re going to display the email address in your webpage as an active email link. The proper way to create a link in HTML is to use the `<a>` and `</a>` tags. The `<a>` tells where the link should start and the `</a>` indicates where the link should end. Everything between these two tags will be displayed as a link and the target of the link is added to the `<a>` tag using the *href* attribute.

For example, a link to google.com would look like this:

```
<a href = ”https://www.google.com/”>Click here to go to Google</a>
```

The <a> tag also provides the option to specify an email address as the target of the link. To do this, you use the “mailto: email address” format for the *href* attribute.

For example, an email link to abc@example.com would look like this:

```
<a href = "mailto: abc@example.com">Send Email</a>
```

In order to write proper HTML for the email link, you will need to write the following helper function:

def create_email_link(email_address):

- This function creates an email link with the given email_address
- To cut down on spammers harvesting the email address from your webpage, this function should display the email address with an [aT] instead of an @
- For example, `create_email_link(‘tom@seas.upenn.edu’)` would return

```
<a href="mailto:tom@seas.upenn.edu">tom[aT]seas.upenn.edu</a>
```

- Note: If (for some reason) the given email address does not contain @, use the email address as is and don't replace the @*
- For example, `create_email_link(‘tom.at.seas.upenn.edu’)` would return

```
‘<a href="mailto:tom.at.seas.upenn.edu">tom.at.seas.upenn.edu</a>’
```

*Note: the *create_email_link* function does not determine if the email address is valid as described in the “Detecting the Email” section above. Detecting a valid email address should be separate from creating the email link. Even though every resume should contain an email address with an @ symbol, we are asking that you create the function this way to practice what to do for unexpected inputs.

<u>Now break the writing of the resume into the following steps:</u>

1. In order to format the resume content, we have to make sure that all the text is enclosed within the following tags:

```
<div id="page-wrap">
</div>
```

Since we typically write things line-by-line, this initial step will write the 1st line `<div id="page-wrap">`. We'll then fill in the actual resume content in multiple steps, then step 5 below will close out the all open tags. So, this initial step just writes the 1st line above.

2. The **basic information section** contains the name and email address enclosed in the proper tags. An example is below:

```
<div>
<h1>Brandon Krakowsky</h1>
<p>Email: <a
href=”mailto:lbrandon@wharton.upenn.edu”>
lbrandon[aT]wharton.upenn.edu</a></p>
</div>
```

   Think about how you can do that. In particular, think about how the *create_email_link* and *surround_block* functions can be used to achieve this. Write another function to make this entire intro section of the resume and then write it out to a file. Note that the email address should be preceded by "Email:" in your final output.

3. For the **projects section**. you will have to produce output like the following. Assume that you have the data about the projects by reading through the original file and stored that in some data structure (decide whether you want to use list, set, tuple or dictionary)

```
<div>
<h2>Projects</h2>
<ul>
<li>built a robot</li>
<li>fixed an ios app</li>
</ul>
</div>
```

   Note that you MUST have `<li>` tags surrounded by `<ul>` tags in your output. If you do not exactly match this format, you will have issues with the project bullet points not displaying correctly. `<li>` is short for **l**ist, and `<ul>` is short for **u**nordered **l**ist.

4. For the **courses section**, we actually just want to list the courses. We do not want to create bullet points. We also want the heading to be a bit smaller in size and enclosed in `<h3></h3>` tags. So here is an example of generated HTML:

```
<div>
<h3>Courses</h3>
```

```
<span>Algorithms, Race car training</span>
</div>
```

Write a function that takes in courses (you can decide whether you want a list of courses or a string of courses) and then write out the HTML, in the form above, to the file.

5. After writing all of this content we just need to remember to close the HTML tags. To do this, we need to write the following 3 lines to the file:

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

   Note that you MUST add the `</div>` tag as well as the `</body>` and `</html>` tags or your HTML will not display correctly.

6. And most importantly, you need to close the file -- otherwise it will NOT save!

Starter Code

We're providing you with a starter file *make_website.py*, which contains some functions to get you started:

def generate_html(txt_input_file, html_output_file):

- You need to implement this function in order to do the following:
  - Call other function(s) in your program (that you define) to load the given *txt_input_file*
  - Call other function(s) in your program (that you define) to get the name, email address, list of projects, and list of courses
  - Call other function(s) in your program (that you define) to write all of the info to the given *html_output_file*

def main():

- This function is the starting point for your program. It is implemented for you and does the following:
  - Calls the *generate_html* function to generate a *resume.html* file from the provided sample *resume.txt* file. **DO NOT REMOVE OR UPDATE THIS CODE!**
  - Calls the *generate_html* function multiple times to generate additional *resume.html* files for other test *resume.txt* files. **AGAIN, DO NOT REMOVE OR UPDATE THIS CODE.**

You **MUST** uncomment each call to the *generate_html* function when you're ready to test how your program handles each additional test r*esume.txt* file.

```
      #generate_html('TestResumes/resume_bad_name_lowercase
/resume.txt',
'TestResumes/resume_bad_name_lowercase/resume.html')

#generate_html('TestResumes/resume_courses_w_whitespace/re
sume.txt',
'TestResumes/resume_courses_w_whitespace/resume.html')

#generate_html('TestResumes/resume_courses_weird_punc/resu
me.txt',
'TestResumes/resume_courses_weird_punc/resume.html')

#generate_html('TestResumes/resume_projects_w_whitespace/r
esume.txt',
 'TestResumes/resume_projects_w_whitespace/resume.html'
)
#generate_html('TestResumes/resume_projects_with_blanks/re
sume.txt',
 'TestResumes/resume_projects_with_blanks/resume.html'
)
#generate_html('TestResumes/resume_template_email_w_whites
pace/resume.txt',
 'TestResumes/resume_template_email_w_whitespace/resume.htm
l ')

#generate_html('TestResumes/resume_wrong_email/resume.txt',
 'TestResumes/resume_wrong_email/resume.html')
```

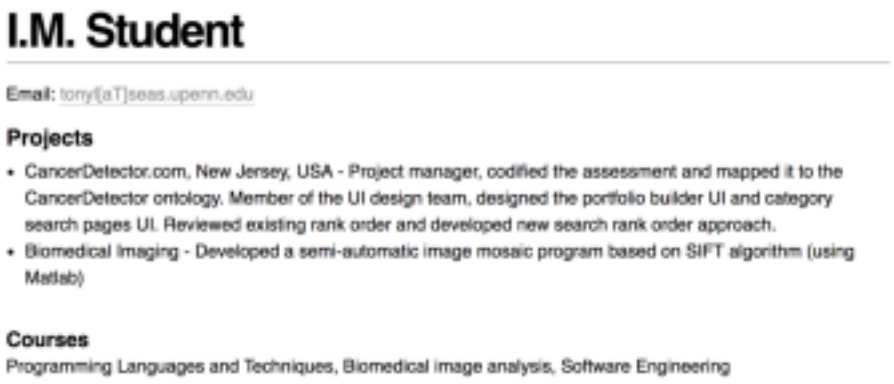
Unit Testing

We're providing you with a starter unit testing file *make_website_test.py*. Currently, it contains only two test functions, *test_surround_block* and *test_create_email_link*. Make sure you pass all of the tests in the unit testing file, plus, be sure to write other test functions to test the functions in your program. You do not need to have tests for functions that write to a file, but you should otherwise have unit tests for every function that you create. You should be testing both typical examples and edge cases.

Although we are providing example resumes, you may find it helpful to create your own sample resumes to test on as well. If you do, include them in your final submission.

Resume Files

Using the sample *resume.txt* file that we provided, the generated *resume.html* file should display like this:



I.M. Student

Email: tony[[aT]]seas.upenn.edu

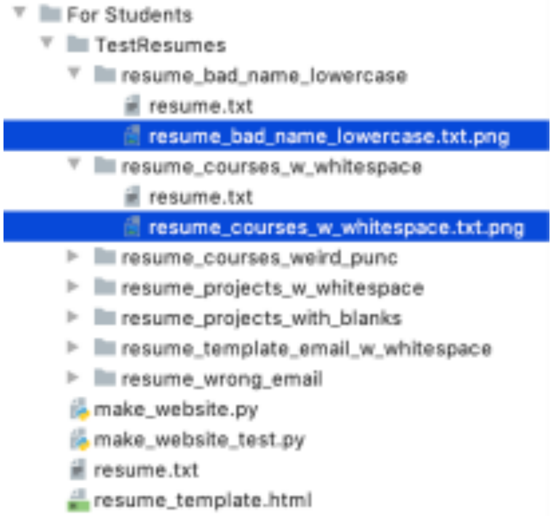
Projects

- CancerDetector.com, New Jersey, USA - Project manager, codified the assessment and mapped it to the CancerDetector ontology. Member of the UI design team, designed the portfolio builder UI and category search pages UI. Reviewed existing rank order and developed new search rank order approach.
- Biomedical Imaging - Developed a semi-automatic image mosaic program based on SIFT algorithm (using Matlab)

Courses

Programming Languages and Techniques, Biomedical image analysis, Software Engineering

We're also providing 7 other different test *resume.txt* files that your program should be able to handle. Each one is in its own subdirectory inside the "TestResumes" directory. Your program will load the test *resume.txt* file in each subdirectory and generate a *resume.html* file. (Some of the code for doing this is provided in the starter file *make_website.py*.) To help with your testing, each subdirectory also contains an image (.PNG) of what the generated *resume.html* file is supposed to look like.



```
▼ For Students
  ▼ TestResumes
    ▼ resume_bad_name_lowercase
        resume.txt
        resume_bad_name_lowercase.txt.png
    ▼ resume_courses_w_whitespace
        resume.txt
        resume_courses_w_whitespace.txt.png
    ▶ resume_courses_weird_punc
    ▶ resume_projects_w_whitespace
    ▶ resume_projects_with_blanks
    ▶ resume_template_email_w_whitespace
    ▶ resume_wrong_email
    make_website.py
    make_website_test.py
    resume.txt
    resume_template.html
```

What to Submit

You will submit the following files:

1. *make_website.py*: the program you wrote
2. *make_website_test.py* [and *any other .txt files you created for testing*]: the unit tests you wrote for all your functions and if you created other .txt files (e.g. different versions of your own resume) to be read by your main program or your unit tests, please include those with your submission. Part of grading your assignment will be to run your unit tests. We want to make sure they pass if they reference other .txt files.

3. *resume.txt*: the sample text resume file provided to be read by your program
4. *resume.html*: the html resume file that was generated when you ran your program
5. The entire “TestResumes” folder: with subdirectories containing additional text *resume.txt* files and the generated *resume.html* file for each. You **MUST** generate the *resume.html* in each subdirectory before you submit your entire program.

Evaluation

It is important that you understand that in this assignment there is more to it than just getting your code to work. Please post on the class forums or come to office hours if you have questions.

1. Functionality – 15 points
    - a. Does your program successfully convert a resume text file into a resume HTML file?
    - b. Can that HTML file be rendered in a web browser? Test it out in at least 2 different browsers to confirm this.
    - c. Is your program able handle the additional test resume text files and successfully convert them to resume HTML files? Do the resume HTML files look like the images provided in each testing subdirectory?
2. Unit Testing – 10 points
    - a. Did you write a test for each function you wrote? An exception to this rule would be a function that writes to a file aa this is difficult to test with unit testing.
    - b. Are you doing a good job of testing all the different cases for each function? You should be testing both typical examples and edge cases.
3. Design and Style – 15 points
    - a. Since we do not give you specific functions, we will be evaluating your design.
    - b. Did you follow style conventions as defined in this course? Do all functions have docstrings and do all non-trivial lines of code have comments?
    - c. Did you use the best data structures and variable types for each situation?
    - d. Did you simplify the logic of your functions? In other words, if it is possible to do the same thing with fewer lines of code, did you do so?