Overview

Homework 3 involves vastly improving ("refactoring") your Python templating script you made in HW 2, to support many more pages, and even blog posts with different content inserted in different locations.

HW2 code was repetitive. It was not *DRY* (Don't Repeat Yourself). The goal of this refactor is to make the code less repetitive. With this addition, you will be able to add new pages simply by adding an item to a list, and your static site generator will do the rest of the work. You will also use a more sophisticated form of templating, which will permit you to more easily update your template files. If fully completed, you will also be able to more easily maintain a blog, with the blog pages and links being auto-generated.

1 Requirements

- **Must be deployed and functioning to GitHub Pages**
- Must have a Python script named "build.py" that generates the site
- All code in Python script must be in functions – no "loose" code (with the exception of the initial invocation of the "main()" function, see below)
- Python script must use lists to store information about all pages
- Python script must use at least one for loop to process all pages
- You must create and use at least 3 functions which, between them, use at least one return statement and at least one argument
- As with before, all code must be submitted as a Pull Request
- Don't worry about the build.sh file – you can delete that. From now on your site is Python only.

1.1 Directory structure

```
- templates/
    - base.html
- content/
    - index.html
    - contact.html
    - blog.html
    - projects.html
- docs/
    - css/           (CSS files, etc)
    - img/           (images, etc)
    - index.html
    - contact.html
    - blog.html
    - projects.html
- build.py
- README.md
```

NOTE: The css and img directories might have different names for you, that's okay.

1.2 Soft requirements

While these are not hard requirements, extremely messy repos might be docked in grade. Keeping your code clean and clear will look the most professional.

1.2.1 Clean repo and HTML

- If you used a pre-made Bootstrap template, they often come with a lot of useless files, often hundreds of them. Take time now to clean up unnecessary files like `package.json`, most JavaScript files, etc. Delete any file that is not necessary for your site.
- Reduce mentions of the company that made your template to fine-print in your copyright footer (what is typically minimally required to be professionally courteous)

1.2.2 Clean code

- *Comenting:* Your code should have a light sprinkling of useful comments explaining what you are doing.
- *Commented out code:* While its fine (and encouraged!) to comment out code while debugging, generally speaking "commented out" code should be deleted before submission.
- *Variable and function names:* Your variables, functions, and classes if applicable should have concise and descriptive names. Avoid single letter names.

2 Steps

2.1 Phase 1 - Using a "main" function

Your scripts initial code must be in a "main" function. The main function must be written and invoked as follows.

```
def main():
    top = open("templates/top.html").read()
    # etc..

# And at the bottom of the same file:
if __name__ == "__main__":
    main()
```

This is "best practices" for writing Python. This will cause `main()` to be invoked when your file is running as a script.[1]

2.1.1 Phase 1 Summary

To accomplish this, the first step is to take your existing static site generator, and refactor it so that all of its code is in a "main" function.

2.2 Phase 2 - List

You must use lists to store all content pages. This allows you to write better, smarter code that lets you grow your site on a day-to-day basis without much copy & paste.

Your `list` must contain dictionaries. Each dictionary has information[2] about a page on your site. It must also contain a file path that can be used to read the contents of the page.

[1] The reason this is best practice is that it also allows the code to be "imported" without main being run. Next week we'll go over in detail what "import" means.

[2] You can consider this *metadata*, which just means data about something else.

2.2.1 Clue

You should have a "pages" list in the following format:

```
pages = [
    {
        "filename": "content/about.html",
        "output": "docs/about.html",
        "title": "About Me",
    },
    {
        "filename": "content/projects.html",
        "output": "docs/projects.html",
        "title": "My Projects",
    },
    {
        "filename": "content/blog.html",
        "output": "docs/blog.html",
        "title": "My programming blog",
    },
    # and so on...
]
```

2.2.2 Phase 2 - Summary

To accomplish this create a list like this, and then refactor your `main()` code with for loop-that goes through this list templating each page. It's okay to put your list outside of your main function if you prefer.

2.3 Phase 3 - String replacement templating

The next big improvement is to combine your `top.html` and `bottom.html` pages into one more manageable `base.html` template file. The rationale here is that this makes it easier to edit your base style in one place, and avoid mistakes with not matching your tags correctly.[3]

With that end in mind, you will need to improve your template file to have a "placeholder" for where the body is supposed to be inserted. In other words, the final `templates/base.html` file will have the contents of `top.html`, followed by the placeholder {{`content`}}, followed by the contents of `bottom.html`. This "placeholder" of {{`content`}} is simply a spot where we are noting that we want to replace later on with the contents of each individual page. This allows our template files to be more manageable as it is only one file.[4]

2.3.1 Clue

The easiest way is by using the `replace` method of strings. Here is some sample code. Read it, understand it, and adapt it to your project:

```
# Read in the entire template
template = open("base.html").read()
```

[3] With two files, it's very easy to lose track of how many <`div`> elements you open in the `top.html`, so that you can close all of them in the `bottom.html`.

[4] It also prepares us for the next step, which is using a templating system called Jinja2 which uses a similar syntax.

```
# Read in the content of the index HTML page
index_content = open("content/index.html").read()

# Use the string replace
finished_index_page = template.replace("{{content}}", index_content)
open("docs/index.html", "w+").write(finished_index_page)
```

If you prefer using different code other than the `replace` method, to achieve the same results, feel free.[5]

2.3.2 Phase 3 - Summary

To accomplish this, rewrite your templates to be a single file including the placeholder where the "body" or contents of each page should go.

2.4 Phase 4 - Function refactor

Background: Currently, all your code is in a single function. Clean code should be distributed between multiple functions to make it easier to read and understand.

Your `main` function should only be for "kicking off". Break up your code into at least 2 other functions. It is up to you how to structure these functions, but between them they should use arguments and return values.

2.4.1 Clue

Here is an example of how code might be broken

```
def apply_template(content):
    # TODO: Read in template, do string replacing, and return result
    return results

def main():
    content = open('docs/index.html')
    resulting_html_for_index = apply_template(content)
```

2.4.2 Phase 4 - Summary

To accomplish this, move your code into separate functions, and use arguments and return values to pass necessary information between them.

2.5 Phase 5 - Advanced templating

Background: Currently, all your pages will end up with the same title in the `<title></tItle>` tags, causing them to look the same when in your browser's tabs or bookmarks.

Once you get your content page list working, and the string replacement working, the next big step is making your string replacement templating system more advanced.

In addition to {{`content`}}, create several other placeholders for other text. You will need to *at least* include one called {{`title`}}, which you will use for the page title in the `<title></title>` tags. You should also

[5] One option is `string.Template`. Keep in mind that next Homework we will be using a real templating system, called Jinja, so no need to get too involved at this point.

devise a way to make the navigation link for each page appear active when it is clicked. This can be done by putting some text next to the link to the page (such as >> or -), or by changing CSS. This will also require templating, since the navigation links should be in your base template, not in your content pages.

If you have time, add in additional templating for things like copyright year, or generating the links themselves so that you only need to add the page to the `content` list, and it will automatically add the page to the list of content links.

2.5.1 Phase 5 - Summary

To accomplish this, replace your title tag with a placeholder, and add code to populate the title based on which page is being viewed. Also, use more templating to spruce up other aspects of your site.

2.6 Phase 6 (Bonus) - Blog templating

1. Each blog page must get its own dedicated page.
2. The `blog` page should now contain only the titles (and, optionally, lead sentences or snippets) of each blog post.

The Python code must contain another list in the following format:

```
blog_posts = [
    {
        "filename": "blog/1.html",
        "date": "September 3rd, 2018",
        "title": "My experience so far as a noob at Kickstart Coding",
    },
    {
        "filename": "blog/2.html",
        "date": "September 10th, 2018",
        "title": "So far I really like Ubuntu Linux",
    },
    {
        "filename": "blog/3.html",
        "date": "September 15th, 2018",
        "title": "My thoughts on Python so far",
    },
    # and so on...
]
```

It must contain at least 3 blog posts. These can have placeholder text if necessary, although you are encouraged to write actual content here if you have time!

Using a for loop, you must loop through this list generating each of these pages into docs. You must also use placeholders, and an additional for loop when generating your your blog.html page, which will auto-generate links to each of your blog posts. Thus, when complete, to add a new blog page, all you need to do is update the list, write the contents in `blog/`, and then regenerate the site, and the "newsfeed" blog.html page will also receive the update.

For this phase, if you have time to complete it, the final directory structure should look like this:

```
- templates/
    - base.html
    - blog_base.html
```

```
- blog/
    - 1.html
    - 2.html
    - 3.html
- content/
    - index.html
    - contact.html
    - blog.html
    - projects.html
- docs/
    - css/    (CSS files, etc)
    - img/    (images, etc)
    - index.html
    - contact.html
    - blog.html
    - blog_post_1.html
    - blog_post_2.html
    - blog_post_3.html
    - projects.html
- build.py
- README.md
```

2.6.1 Phase 6 (Bonus) - Summary

To accomplish this, add a new template called `blog_base.html`, that should in turn use the `base.html` template. This new template should now be used for the blog posts.

3 Getting help

This is a hard assignment which leaves a lot of implementation details up to you. Get help RIGHT AWAY if/when you get stuck.