Overview

With Homework 4, the goal is to complete the Python *static site generator* that you worked on in HW3. You will do this with the help of built-in modules in Python, and downloadable packages from PyPI.

In the end, you will have written a custom static site generator that gives you greater power in customizing the template and updating your site so that the content remains relevant. As before, this builds on your previous homework.

1 Requirements

1.1 Hard requirements

- **Must be deployed and functioning to GitHub Pages**
- Must detect files placed in the `content/` directory and build pages from there
- Must have a Pipfile generated by `pipenv`
- Must use Jinja to template the pages
- Must have new file structure, including a `manage.py` file, and a `utils.py`
  - The `manage.py` must support `new` and `build` operations
- Previous requirements apply: Should be built based on content in `content/` directory, the results should go into `docs/` and it should have at least 3 total pages.

1.2 Soft requirements

- Previous code cleanliness requirements apply: Extremely messy repos might be docked in grade. Keeping your code clean and clear will look the most professional. Remove unused files, add in commenting, and descriptive variable names are a must.
- Using branches and/or pull requests for development is not necessary: Feel free if you prefer to work entirely on master.

2 Homework Steps

To receive full credit, you must complete all phases. However, the earlier phases are worth much more, so you should attempt to fully complete each phase before moving on to the next.

As with the previous homework, your goal is to get your software working and meeting the requirements. The clues and code snippets offered may need to be modified or tweaked for your use. Focus on understanding and accomplishing each goal one at a time.

2.1 Phase 1 - Auto-discovery of content files

Background: Right now, adding new content requires editing your list. This is cumbersome. Your first goal is to make it so that your site generator will automatically "pick up" on any files placed in your `content/` directory, outputting their templated counterparts in `docs/`.

For Phase 1, we will no longer use a fixed list, but instead have the list of files be generated based on what is found in the `content/` directory. The most direct way to do this: Delete the contents of your `pages` list, leave all the rest of your existing code as-is, and create a new function (and accompanying invocation) to "auto-generate" the list. Then invoke the code from the previous homework to use the list.

- Other than deleting the initial contents of your `pages` list, accomplishing this Phase 1 ONLY requires the additions to your code to generate the pages list so that your existing code can use it to generate your site.
- Your task is to think about how the clue pieces of code fit together, and how you might generate the 3 dictionary values from just the input filename you derive from the items in the list generated by `glob`.

Reminder: Use print to help with debugging!

2.1.1 Clue 1

A clear first task then is to be able to loop through files in the `content/` directory. Python comes "batteries included" for this. One approach is with the module "glob".[1]

See if you can get the following code working, and examine it's output:

```
import glob
all_html_files = glob.glob("content/*.html")
print(all_html_files)
```

What type of data is `all_html_files`? Think about how you might use this code to "discover" files.

2.1.2 Clue 2

Yet another useful builtin package is the `os.path` package. It allows you to modify and extract useful parts from file paths.

See if you can get the following code working, and examine it's output:

```
import os

file_path = "content/blog.html"
file_name = os.path.basename(file_path)
print(file_name)
name_only, extension = os.path.splitext(file_name)
print(name_only)
```

[1] Glob is another word for using asterisk (`*`) as a placeholder, such as the asterisk in "wild-card expansion syntax" in Bash (such as `*.txt`.).

2.1.3 Clue 3

You will likely want to make your list start out empty, so that you can build it up from the contents of the `content/` directory. Examine the following code:

```
pages = []
pages.append({
    "filename": "content/index.html",
    "title": "Index",
    "output": "docs/index.html",
})
print(pages)
```

2.1.4 Phase Summary

To accomplish this, you must adapt your code so that it will apply templates to all files it finds in the content directory, generating parallel output files in the `docs/` directory.

2.2 Phase 2 - Jinja2 Templating

Background: Right now, you are using a lot of "replace" methods to do templating. This is much less professional, and limits what you can do with your templating.

Improve your `build.py` to use Jinja templates instead of the "home-made" replace-based templates you are using thus-far. For this to work, you must use `pipenv` to install `jinja2`.

2.2.1 Clue 1

First, you must get set up with a new Python "virtualenv" in your `git` repo. Three commands to do this:

```
pipenv --python 3.6
pipenv shell
pipenv install jinja2
```

In subsequent times working on the homework, run `pipenv shell` from the same directory to regain access to the "virtualenv" with `jinja2`.

2.2.2 Clue 2

See if you can get the following code working, and examine it's output:

```
from jinja2 import Template
index_html = open("index.html").read()

template_html = open("base.html").read()
template = Template(template_html)
template.render(
    title="Homepage",
    content=index_html,
)
```

Can you understand what it is intended to do? Use this as a clue for incorporating Jinja templating.

2.2.3 Phase Summary

To accomplish this, you must use jinja2 instead to do the string replacements ("templating") on your site.

2.3 Phase 3 - Improved templating

Background: Right now, if you add new content pages, you will have to update your `base.html` file to match. This is not DRY. Jinja offers functionality with for-loops to make this much more DRY.

After you have added Jinja templating and beefed up your markdown support, you should make it so that your site's page navigation is auto-generated. Use the Jinja loop feature for this.

2.3.1 Clue

Some example Jinja2 code:

```
{% for page in pages %}
    <a href="{{ page.output_filename }}">{{ page.title }}</a>
{% endfor %}
```

You'll also need to include in the Jinja2 context a list of all pages while rendering, e.g. something like: `template.render(pages=pages, content=content_html)`

2.3.2 Styling the active link

To receive full credit for this feature, style the active link differently than the others. This might involve conditionally adding new classes or bullet point to distinguish it from the other links.[2]

2.3.3 Phase Summary

Utilize the power of Jinja to autogenerate a list of links in your navigation section of your site.

2.4 Phase 4 - Splitting into utils.py and manage.py

Background: In Python web development, it's traditional to use a file called `manage.py` as your main *entry point*[3] , but leave most of the other code in other files.

1. Split your code into two files: `utils.py` and `manage.py`.
2. All your functions should be in `utils.py`. Running `utils.py` directly should do nothing.
3. Going forward, `manage.py` will be the only file you run directly (with `python3 manage.py`). It in turn should use `import` to bring in code from `utils.py`, and then invoke the necessary functions found in `utils.py` to perform the same functionality as before.

[2] You might consider using Jinja "if-statements", or filters, to check which page is active right now, and which one is in the loop, and thus conditionally style the active page differently than the other pages. This might involve either adding a new HTML/CSS class, or something like a bullet point or arrow.

[3] The term "entry point" refers to the file you run from the CLI to kick off your application. You'll see `manage.py` as the conventional entry point for Django, as well as other web frameworks such as Flask.

2.4.1 Phase Summary

Split your code into two files, and use `import` to allow one to invoke functions from the other.

2.5 Phase 5 - Command-line arguments for manage.py

Background: Full-featured static site generators provide functionality to make it easier to quickly create new pages. Using command-line arguments[4] we will add extra functionality to `manage.py` to support this.

Modify `manage.py` to add the following functionality:

- `python3 manage.py build` — should do the same functionality as your `build.py` script did before. Just run the old code.
- `python3 manage.py new` — it should create a new placeholder file in `content/` directory. This is brand-new functionality that will require new code in `manage.py` and `utils.py`.[5] For example, it should create a file called `content/new_content_page.html`, with contents that look like:[6]

  ```
  <h1>New Content!</h1>

  <p>New content...</p>
  ```

- `python3 manage.py` — Without arguments, it should use `print`[7] to give hints on how to use the program. For example, print out:

  ```
  Usage:
      Rebuild site:    python manage.py build
      Create new page: python manage.py new
  ```

2.5.1 Clue

```
import sys
print("This is argv:", sys.argv)
command = sys.argv[1]
print(command)
if command == "build":
    print("Build was specified")
elif command == "new":
    print("New page was specified")
else:
    print("Please specify 'build' or 'new'")
```

[4] *Command-line arguments* are the space separated words we add to the end of running a command from within Bash. They are called command-line arguments because it resembles using arguments when invoking a function. For example, when we run `git add -A`, then `add` and `-A` are the two CLI arguments passed to the `git` command. In our case, if we run `python3 manage.py build`, then `build` is the singular command-line argument passed to our `manage.py` script.

[5] Not strictly required, but to make this command more useful, it should ask the title of the new page via use of Python's "input" function, and use that as the title in the new file. If you can, also make it use the title as the filename, only "sanitized" so that non-alphanumeric characters are replaced with underscores.

[6] If your content pages tend to have a certain structure of <`div`>s and such, it should use that structure instead to get your new content page going.

[7] Or, alternatively, `sys.stderr.write`.

2.5.2 Feature Summary

Modify the `manage.py` file to also offer the ability to create new placeholder content via a `python3 manage.py new` command.

2.6 (Bonus) Phase 6 - Markdown support

Background: Right now, you are creating all of your content in HTML. Building an online presence requires writing frequently. If you just want to have the ability to throw together a simple blog post in just a few minutes without struggling with HTML syntax and update on a weekly basis, you'll want to use a blog-writing language such as Markdown.

"Markdown" is a simplified version of HTML[8] that is intended to make it easier for humans to read and write. Since browsers cannot read it directly, it must first get transformed into real HTML using a Python package. Many popular blogging systems use it to allow easy formatting of blog posts – you may use it right now.[9]

Your goal is to integrate Markdown into your project. You should start by renaming your content pages to `.md` instead of `.html`. Since Markdown supports HTML, it's okay to leave HTML mixed in. The output should still be `.html`, since browsers cannot read Markdown.

In order to do the Markdown to HTML format conversion, you will need to add code to your project. The most popular Markdown package to do this is called, predictably, `markdown`, and can be installed the same way you installed `jinja2`. Read more about Markdown here: wikipedia.org/wiki/Markdown

2.6.1 Example Markdown code

```
title: About me

# Header

Paragraphs with *styling* and [a link to Google](http://google.com/)

![And an image](image.jpg)

<div>And HTML is okay, too</div>
```

2.6.2 Clue 1

Look at the following code. Can you understand what it is doing? It will not work in your project precisely as-is, but should present the right approach.

```
import markdown
md = markdown.Markdown(extensions=["markdown.extensions.meta"])
data = """title:  My New Blog
author: Jane Q Hacker

Welcome to my ~~site~~ *blog*
"""
html = md.convert(data)
```

[8] You can also mix in real HTML, so in a way it is just making HTML more user-friendly.

[9] Facebook Chat and Google Chat use something similar, as does Mattermost, Slack, Ghost, and many others. It's simply the syntax where we can create italic and bold text by using patterns like `_italic_` or `*bold*`, and can separate out paragraphs by simply hitting <Enter> twice instead of typing out <p> tags.

```
title = md.Meta["title"][0]
author = md.Meta["author"][0]
print(title, "by", author)
print(html)
```

2.6.3 Clue 2 - Metadata

Notice the colon separated data at the top of the example markdown? You can add "metadata" like this directly in your markdown files. This won't be in the final HTML, but will allow you to add extra information like the page title. [10] The sample code provided also shows how you can access it from within Python.

2.6.4 Feature Summary

To accomplish this, one (or all) of your pages should be written in Markdown instead of HTML, and your build process should turn those Markdown pages into true, fully-templated HTML pages.

2.7 (Bonus) Phase 7 - Auto-generated blog pages

Background: Tech blogs are great ways to improve your online presence and career. With the groundwork we've laid, it's time to add blogging functionality to your static site generator.

Your site should have a "blog" page, that shows summaries of all the blog articles you have written. This requires you to put multiple files in the `blog/` directory, each file designating another blog post. As with content pages, read blog pages in as markdown files, convert to HTML, and then template using Jinja. You might also consider adding `new-blog-post` command to your `manage.py`.[11]

You will need a new template for the blog in the `templates/` directory, such as `templates/blog_pages.html` which loops through all the blog pages displaying their contents.

2.7.1 Extra blog features

- **Syntax highlighted code snippets** - Look into how to tweak your Python, and add the CSS necessary so that Markdown will correctly syntax-highlight your code snippets in your blog page.
- **Disqus** - Commenting via Disqus: If you already have Disqus[12] on your page, make sure Disqus is configured to serve different comment sections for each of your different blog pages. This might require more templating tweaks.

2.7.2 Phase Summary

To accomplish this, create a coding blog populated from markdown files in the `blog/` directory.

[10] "Metadata" means data about the file. This would include anything that is not in the body of the content itself, such as title, date of publication, author, and so on. It might be displayed in different parts of your final project, such as in the link, or in the page title.

[11] This should use the current date and time as the posting time for the metadata of the new file it creates.

[12] Disqus is a third-party service that you can embed with relative ease by copying and pasting some HTML. It is the most popular way of adding comment sections. It was mentioned in earlier homework.

3 Bonus Assignment: Package and publish

A final bonus assignment is to actually publish and release your static site generator as a free software tool. Just like we can install `pipenv`, and use that as a command-line tool, you can make it so that other people can download and install your tool and use it to make their own sites.

NOTE: This is a time-consuming process. In the end, however, you (and everyone in the world) will be able to install your static site generator using pipenv or pip, and run it directly in the terminal just like any other application in the terminal such as `git`, etc. Chat with an instructor on the best steps for this.

A tiny tool you might use as reference on how small tools might look when they are published: github.com/michaelpb/stowage/

If you do decide to go down this difficult route, then here are some suggested steps:

1. Come up with a novel name / branding for your static site generator. For reference, a few popular static-site generators are `jeckyll`, `hugo`, and `sphinx`. This will be what other people would call your application, and it's name on the command-line, on PyPI, and on GitHub.
2. Separate out all your Python code into a new repo, separate from your site. From now on, your Python code will be a SSG that you use like a tool to *generate* your homepage. The repo name should be the same as the tool name you invented. It should have no trace of your personal website, either in the source code, or in the documentation or HTML.
3. Alongside the `build` and `new` commands, create an `init` command for your `manage.py` file. This should generate a `templates/` directory, a `docs/` directory, and a `content/` directory, with a simple placeholder template and one or two placeholder content pages. This is to help future users get started making a new site using your tool.
4. Write a high quality README file that clearly describes how to use your static site generator, and put it in README.rst (a very similar format to Markdown that is the default for PyPI).[13]
5. Actually publish it on PyPI. This requires creation of a `setup.py` file, among many other things. Ensure you have your main function in your `manage.py` file being used as the main "entry point" for your application. [14] As was warned, this is not an easy process. A tutorial is below: `peterdowns.com/posts/first-time-with-pypi.html`
6. Finally, make sure everything is in working order, and represents work others will find useful! "Pretend to not be you", and install your package and try using it, only based on the documentation you provide. Can you get a website up and running? Make sure your application gives feedback to users if they are doing something wrong, having a "help" menu of some sort and friendly error messages.

3.1 More tips

- If you want to go really deep into industry-standard, correctly built Python packages, there are "cookiecutters" for this. It might be easier than following tutorials. We will get to cookiecutters later, but if you are curious now: `github.com/audreyr/cookiecutter-pypackage`
- If you want to improve your CLI flag capabilities, including `--help` options, check out the `argparse` module: `stackoverflow.com/questions/774824/explain-python-entry-points`
- When you are done, you should list it as a project on your homepage.

[13] You can use a tool called `pandoc` to generate RST so you don't have to learn something new. After installing `pandoc` via `apt` (Ubuntu Linux) or `brew` (macOS), you can run `pandoc README.md -o README.rst` to convert.

[14] Look up the "entry point" console scripts python-packaging.readthedocs.io/en/latest/command-line-scripts.html Also see the setup.py of the stowage example mentioned above.