Overview

This template has been roughly set up around a 'Model View Controller' or MVC design. This splits the functions of your site into three distinct categories:

- Models: Handles the program logic
- Views: Handles the returned HTML pages
- Controllers: Handles the requests for pages that the user sends

Typically the user will request a page (from the controller), the request will be interpreted and then passed to the program logic (the model) which will generate a new page to return to the user (the view).

This setup has been used to keep the logic of the code and the logic of the site separate.

If your site starts sprawling enough, you may wish to create distinct folders for each of these categories and then split the code into separate files within these folders.

Controllers

This is the handler for your server requests. Your get and post decorators will go here, before calling a model to handle any actual lifting.

All of your bottle relevant code should live here. The rest of your code should not be exposed to anything to do with bottle. Similarly, no SQL or other 'database' or logical code should live up here. You have been provided with static file loads for javascript, css and image files from the javascript, css and img directories respectively.

Models

The brains of the operation, here we perform whatever actual code we need, before calling our view object to return the templated HTML. This area should make calls to any 'databases' or other persistent storage that is handling user or other data.

HTTPS

The library bottle used in the template does not support ssl. To enable bottle server in the template support https/ssl in python, one python library of WSGI that Bottle supports is needed. There are many choices, such as Gunicorn, CherryPy, etc. If you are still unclear, just google sth like how to enable Bottle server support HTTPS python. It is definitely fine if you use some libraries other than Bottle.

Views

Simply loads our HTML files and renders any elements of the template.

It might be helpful to have an explanation of the View class. You do not actually need this to use bottle at all, but it's a primitive method of automating loading and rendering HTML templates.If you already have your own method of managing this, please feel free to disregard the explanation below. If you don't like how parts of this have been implemented, you are more than free to modify it for your own use.

The template has been modified to be more explicit and verbose in what it is doing rather than strictly the most efficient or Pythonesque method.

If you're completely lost: the point of this code is to "render HTML", all this really means is that we're going to take a string, modify it a bit and return it. HTML is effectively just some specially formatted text. I would suggest starting by looking at and building the polling site in tutorial 3.

All the code below is just to read text from a file, replace a few keywords and then return it.

Once you've completed the polling site you might wonder if hard coding all the HTML responses is the most efficient method. Depending on the size of the project it might or it might not be. It is entirely possible to hard code all the pages required for this assignment. However, one method of managing HTML is to store it in a separate file, then read and return it when required.

This is performed by the load_template method. It opens a file given by the filename argument that is found at the filename path (here it's "/template") and with the template extension (here it's ".html").

```
def load_template(self, filename):
        path = self.template_path + filename + self.template_extension
        file = open(path, 'r')
        text = ""
        for line in file:
            text += line
        file.close()
        return text
```

Of course some times you want dynamically generated HTML (for example, displaying a username after logging in). In order to do this we're going to need to do some string operations.

```
def string_format(string, format_dict):
    for keyword in format_dict:
        string = string.replace('{' + keyword + '}', format_dict[keyword])
    return string

string = "Thanks for logging in {user}."
format_dict = {"user":"Anon"}
formatted_string = string_format(string, format_dict)
print(formatted_string)
```

Or instead we can use the Python format function:

```
string = "My hovercraft is full of {things}"
formatted_string = string.format(things="eels")
print(formatted_string)
```

The simple_render method within View calls the Python safe_substitute function on a given template, this works in a very similar fashion to the format function we saw previously.

```
def simple_render(self, template, **kwargs):
        template = string.Template(template)
        template = template.safe_substitute(**kwargs)
        return  template
```

**kwargs is a Python default method of passing arbitrary keyword arguments (see tutorial 1!) as a dictionary, this lets us pass our dictionary around without actually having to worry about the contents.

Now let's say that there are some "global" dynamic template options we want to use, things that we can just pass into a template when it's called. For this we'll follow exactly the same method as above, but store these "global renders" as a member variable.

```
def __init__(self,
        template_path="templates/",
        template_extension=".html",
        **kwargs):
        self.template_path = template_path
        self.template_extension = template_extension
        self.global_renders = kwargs
```

Here we're using **kwargs again for the "global renders"...

```
def render(self, template, **kwargs):
         '''
            render
            A more complex render that joins global settings with local settings

            :: template :: The template to use
            :: kwargs :: The local key value pairs to pass to the template
        '''
        # Construct the head, body and tail separately
        rendered_body = self.simple_render(body_template, **kwargs)
        rendered_head = self.simple_render(header_template, **kwargs)
        rendered_tail = self.simple_render(tailer_template, **kwargs)

        # Join them
        rendered_template = rendered_head + rendered_body + rendered_tail

        # Apply any global templating that needs to occur
        rendered_template = self.global_render(rendered_template)

        # Return the template
        return rendered_template
```

...and here we build the header, tailer and body, join them together and then apply any global replacements we might need

Lastly let's consider some generic headers we can add to every page on the site (in the case of Drawing Straws, this is the menu bar and the image on every page), and the "tailer" to properly enclose the HTML. This can be more efficiently managed using proper rendering calls but this template was not built for flexibility so much as for ease of use on a few small sites.

Putting it all together we now get our load and render method:

```
def load_and_render(self, filename, header="header", tailer="tailer", **kwargs):
        '''
            Loads and renders templates

            :: filename :: Name of the template to load
            :: header :: Header template to use, swap this out for multiple
headers
            :: tailer :: Tailer template to use
            :: kwargs :: Keyword arguments to pass
        '''
        body_template = self.load_template(filename)
        header_template = self.load_template(header)
        tailer_template = self.load_template(tailer)

        rendered_template = self.render(
            body_template=body_template,
            header_template=header_template,
            tailer_template=tailer_template,
            **kwargs)

        return rendered_template
```

And that's the View class. It's quite a simple template framework compared to some of the more featured ones used in larger scale web development, but it provides the basic features required for this assignment.

If you are confused about all this I would suggest loading up the template site and looking at the "about" page. Modify the garble keyword on the line:

```
return view("about", garble=np.random.choice(garble))
```

To something like:

```
return view("about", garble="My String!")
```

Then reload the site and have a look at the "about" page. Have a look at the "{garble}" section of the "about.html" file in the templates directory.

I

SQL

Not strictly a requirement, if you want to use SQLite3 then some sample code has been provided in the SQL file. This code is not necessarily in a fully working state, and you will probably want to modify it extensively. If you are unsure why something is working in a particular way, be sure to read all the comments before making an Ed post.

Javascript

Also not strictly a requirement, if you wish to use javascript it **must** be loaded locally, CDNs and loading from external sites is prohibited.

End notes

I hope this helps explain what's going on. If you're still unsure about all of this please ask sooner rather than later.