Version Control

Your lecturer will introduce version control, explaining why it an indispensable part of any developers' toolset. Apart from recording your history of changes -- and synchronising them to the cloud -- it makes collaboration in code projects far more manageable.

For version control, you will be using GitHub Desktop. This is a free, multi-platform application that integrates with github.com; download it here:

https://desktop.github.com/

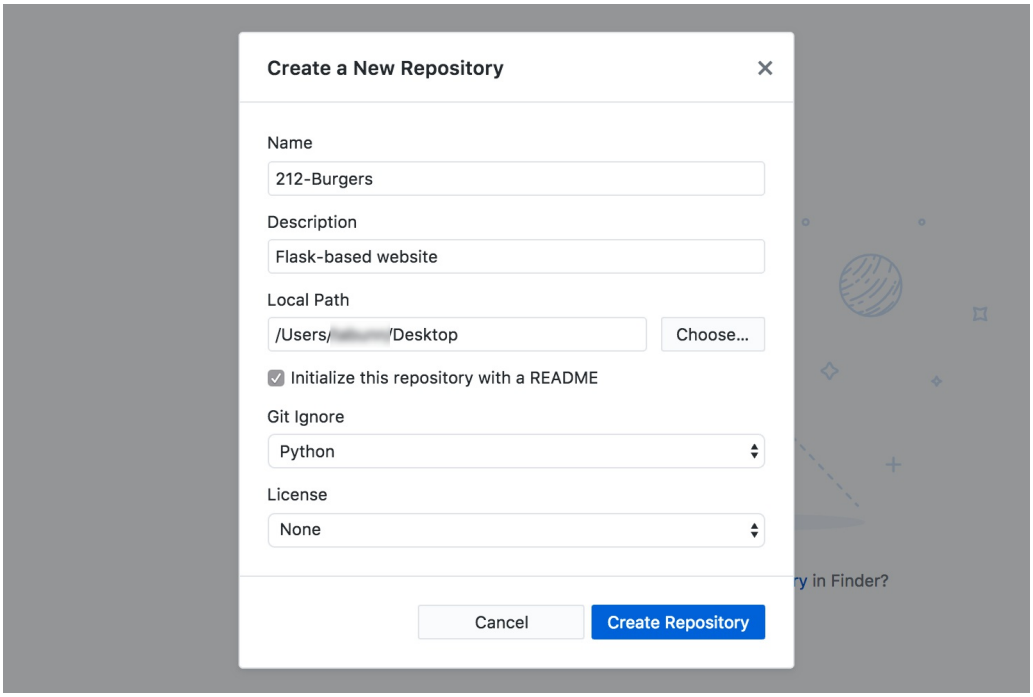
When running GitHub Desktop for the first time, you'll be prompted to enter your GitHub account details. If you don't have an account, sign-up for a free one at:

https://github.com/

Once you're done installing things, select **File > New Repository...**

In the modal that appears:

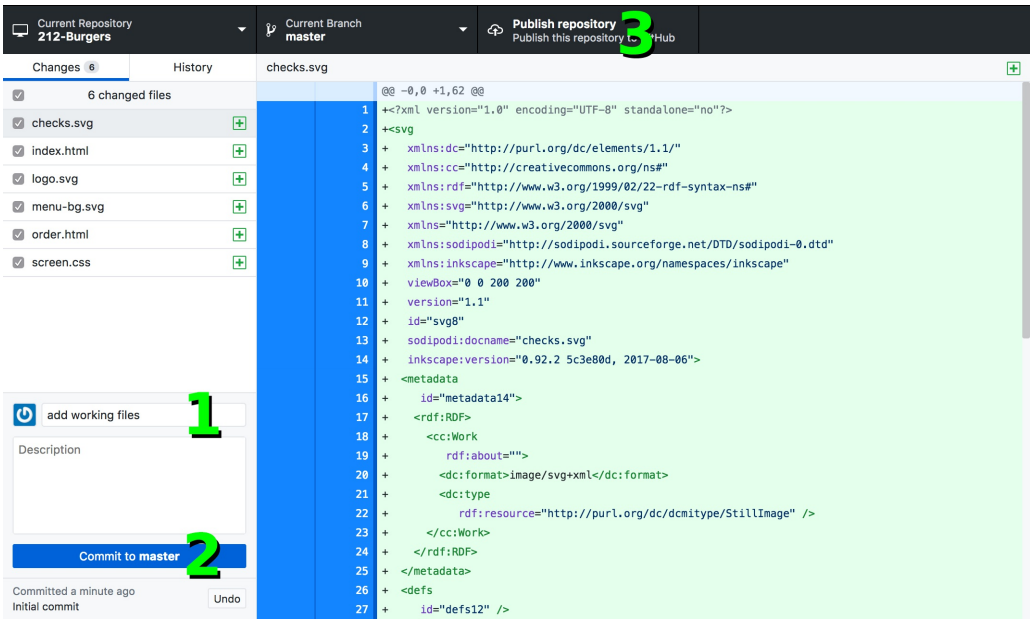
1. in the *Name* field, enter `212-Burgers`;
2. in the *Description* field, enter `Flask-based website`;
3. choose a *Local Path* where you prefer to work (i.e. your Desktop);
4. check/tick the README option;
5. for the *Git Ignore*, select Python;
6. do not worry about a license;
7. click **Create Repository**.



Grab a copy of the last week's class files here (to ensure that everybody is working on the same code):

http://stream.massey.ac.nz/mod/resource/view.php?id=2351394

Extract and add these to your repo directory (the "212-Burgers" on the Desktop). If you have done this correctly, the changes will reflect in the GitHub Desktop *Changes* list:



Now:

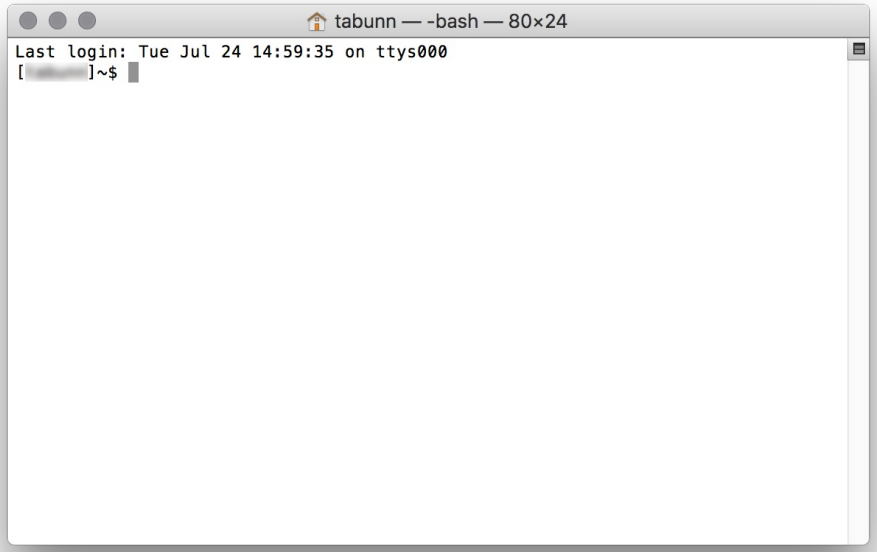
1. add a commit *Summary*;
2. click **Commit to master**;
3. then **Publish repository**.

You need not Publish/Push each time you commit, but ensure that you do this upon finishing your work session (i.e. leaving the classroom, office, or coffee shop) so that your changes are pushed to the cloud.

Command Line

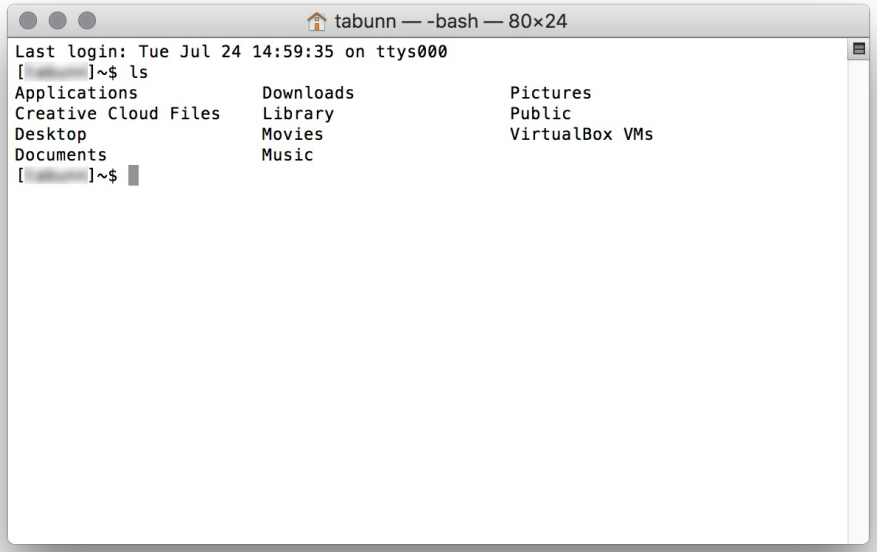
The command line looks intimidating, but knowing the basics will prove very useful for development work. Linux, Mac, and Windows all include some *terminal*/command-line environment. These notes will use the Mac terminal; if you are using Windows, the application you'll need is the *Command Prompt*.

On Mac, open the Terminal (command + spacebar, then type Terminal). It should look something like this:



The prompt (`[user]~$`) will vary depending on your settings and system, but this is not important. You may also have a white-on-black colour scheme (Windows), but most terminals allow one to customise the colours.

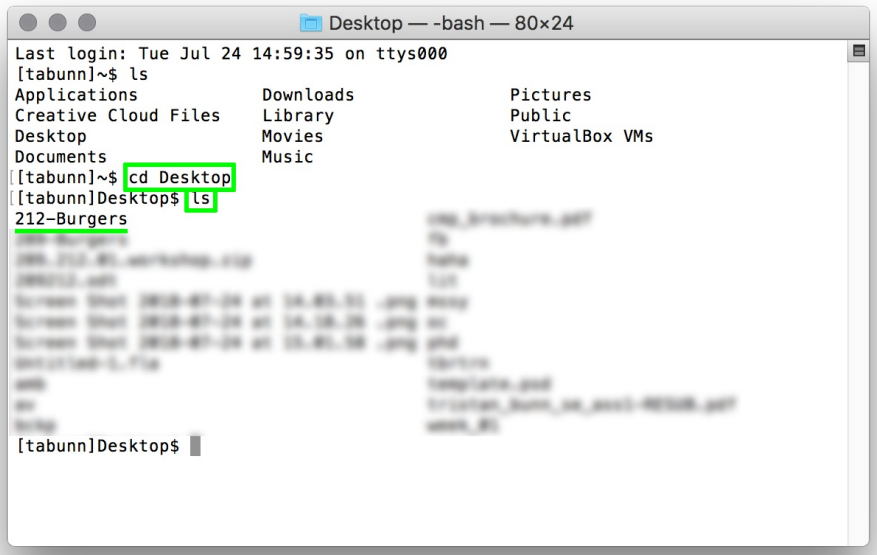
Now try your first command -- type `ls` (or on Windows `dir`). This lists the contents of your user's home directory:



`ls` lists the contents of the directory you are currently 'in'. By default, this is your home directory -- but you can change to the Desktop by typing:

```
cd Desktop
```

You can confirm that you are in the Desktop directory by listing your files:



As you have probably guessed, the `cd` command performs a **c**hange **d**irectory. Use `cd ..` if you ever need to enter a parent directory (go a folder up).

The next step is to `cd` to `212-Burgers`. In my case, the 212-Burgers directory is on the Desktop:

```
[tabunn]Desktop$ cd 212-Burgers/
[tabunn]212-Burgers$ ls
checks.svg      logo.svg        order.html
index.html      menu-bg.svg     screen.css
[tabunn]212-Burgers$
```

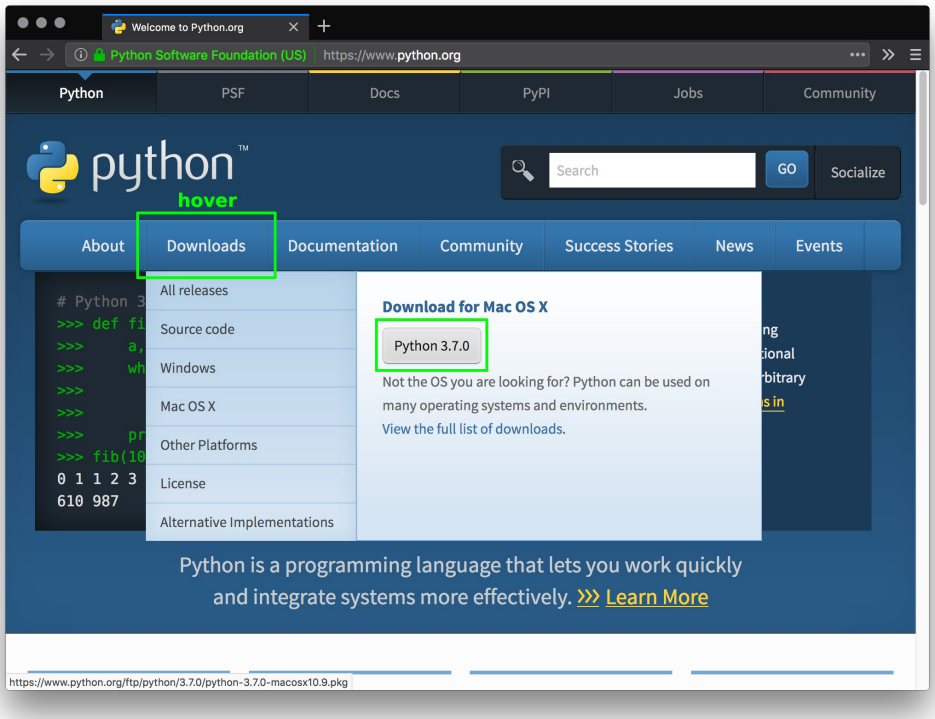
Change to the directory in wherever it is that your `212-Burgers` directory resides.

Virtual Environments

Python's virtual environment feature takes the headache out of working with different versions of Python and Python libraries.

Installing Python 3

Before going in further, you will need to install the latest version of Python 3. You can download this from the python.org website:



Once downloaded, run the installer.

Setting Up a Virtual Environment

With your command line ready and waiting in the `212-Burgers` directory, type in the following command:

```
python3 -m venv env
```

This will create a new directory named `env`. The `env` directory will contain your virtual environment files (all of the additional libraries you install, like Flask, for this project). You need to activate this using:

```
source env/bin/activate
```

You should now see an `(env)` in front of your prompt.

Flask is not included with Python; to install it, enter:

```
pip install flask
```

Flask Application Structure

Great! You are ready to run Flask ... almost.

Create a new file in your editor (Atom?), and save it in your `212-Burgers` directory as `run.py`. Add the following code:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return('<h1>hello world</h1>')
```

To explain the above: the `from flask import Flask` line imports the components you require; the `app = Flask(__name__)` line creates a new app instance; and the rest is responsible for serving up a landing page.

In the terminal, you will need to set two environmental variables using the following commands:

```
export FLASK_APP=run.py
export FLASK_DEBUG=1
```

On Windows, use `set` instead of `export`.

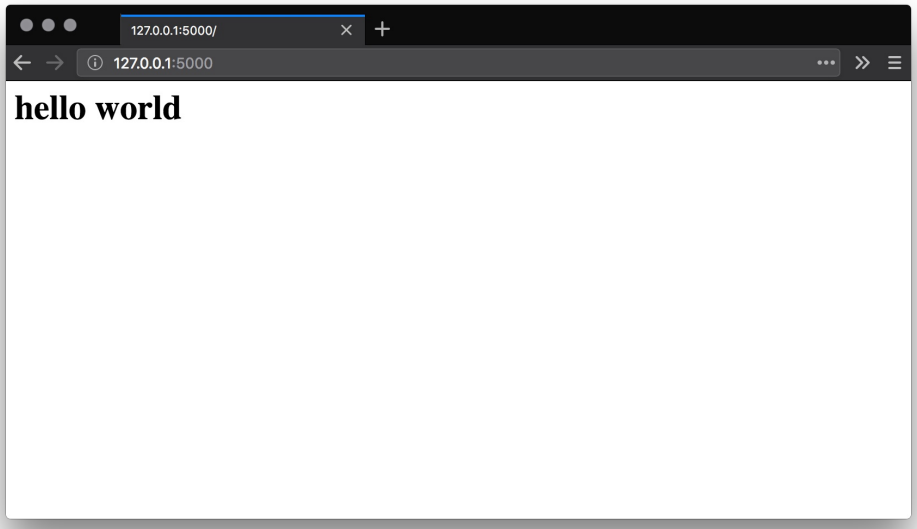
Now run your app server using:

```
flask run
```

The server will report that it is running:

```
(env) [tabunn]212-Burgers$ flask run
 * Serving Flask app "run.py" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 230-749-452
```

You can enter `http://127.0.0.1:5000` into your browser to see the result.



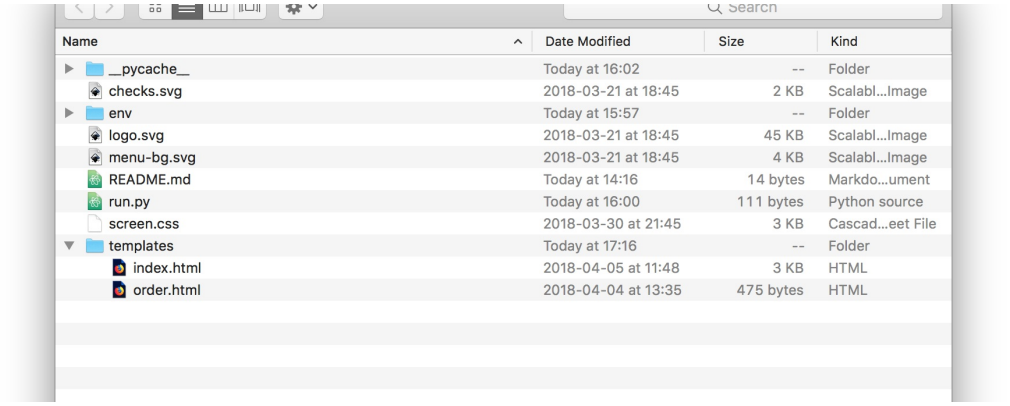
Now is a good point to commit your changes (using GitHub Desktop).

Flask Templates

Flask employs the Jinja2 library for website templating. Jinja2 allows you dice-up your html code into reusable components (i.e. a reusable header, footer, etc.).

Rendering

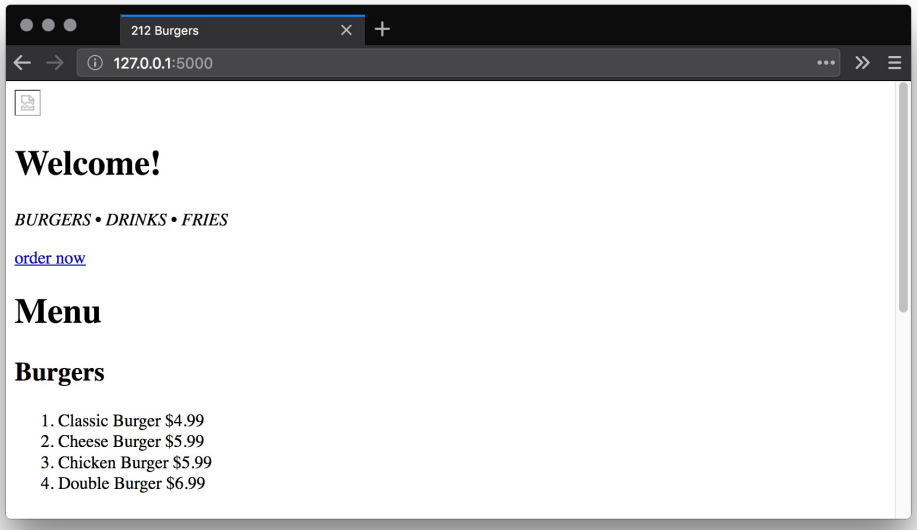
By default, Flask expects template files to be placed in a directory named "templates". Create a "templates" directory and move the .html files into it (leaving images, css, etc. behind).



Now edit the *run.py* code, adding `render_template` to the import (first) line and adjusting the `return` line:

```
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')
```

Check your browser. Things work, kind of.



Don't worry we'll fix the broken images and stylesheet shortly.

Variables

Next, you'll pass a variable into the template by adding it to the `render_template` function:

```
...
    return render_template('index.html', disclaimer='may contain traces of nuts')
```

To display the disclaimer, add a template variable to the **index.html** file:

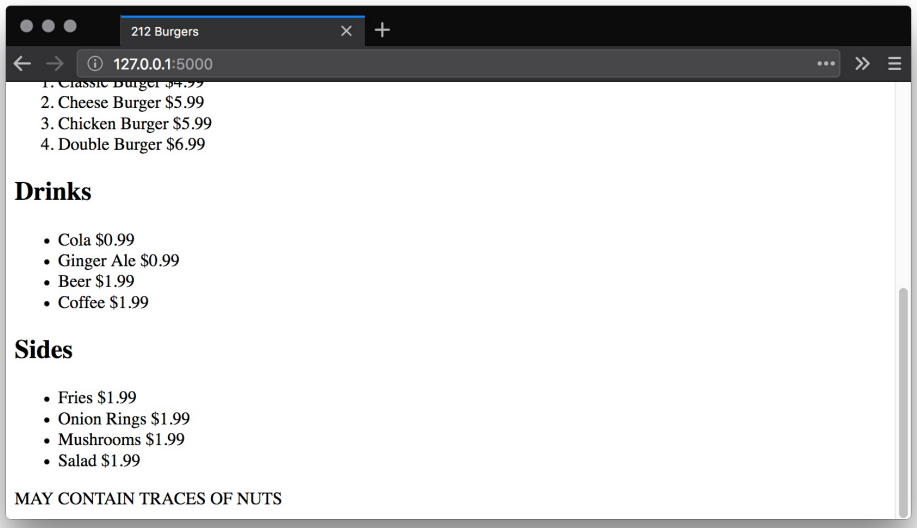
```
    ...

    <div id="footer">
      {{ disclaimer }}
    </div>

  </body>

</html>
```

Refresh your browser and verify that the disclaimer is displaying.

Filters

Add an `upper` filter to the disclaimer variable so that it displays in uppercase:

```
    ...

    <div id="footer">
      {{ disclaimer|upper }}
    </div>

  </body>

</html>
```



There are various filters available, including `lower`, `title`, `capitalize`, `safe`, and `trim`.

Control Structures

Python offers array-type structures. *Lists* are one such example.

Add a new burgers list to your *run.py* code:

```
from flask import Flask, render_template
app = Flask(__name__)

burgers = [
 'Classic Burger $4.99',
 'Cheese Burger $5.99',
 'Chicken Burger $5.99',
 'Double Burger $6.99'
]

...
```

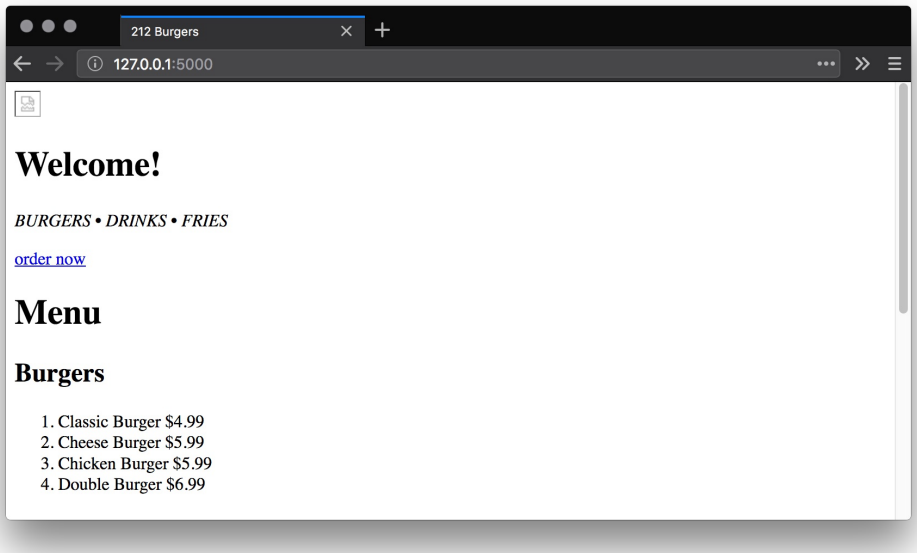
Then pass the list to your template:

```
...

@app.route('/')
def index():
```

```
    return render_template('index.html', disclaimer='may contain traces of nuts', burgers=burgers)
```

In this case, you'll need the template to loop through the list of items the `burgers` variable holds, repeating some HTML for each value. Edit your *index.html* code:

```
...

<div id="menu">

  <h1>Menu</h1>

  <div>
    <h2>Burgers</h2>
    <ol class="menu-items">
      {% for burger in burgers %}
      <li>{{ burger }}</li>
      {% endfor %}
    </ol>
 </div>

 <div>
   <h2>Drinks</h2>
   ...
```



Although it may not be visually apparent without the stylesheet working, the problem here is that you have removed all of the . This requires a list of lists ... I know, this sounds complicated, but replace the following code in each file and read through it -- comparing it to what you had before -- to make sense of what is going on:

run.py

```
from flask import Flask, render_template
app = Flask(__name__)

burgers = [
 ['Classic Burger', '$4.99'],
 ['Cheese Burger', '$5.99'],
 ['Chicken Burger', '$5.99'],
 ['Double Burger', '$6.99']
]

...
```

index.html

```
    ...

    <div id="menu">

      <h1>Menu</h1>

      <div>
        <h2>Burgers</h2>
        <ol class="menu-items">
          {% for burger in burgers %}
          <li>{{ burger[0] }} <span class="price">{{burger[1]}}</span></li>
          {% endfor %}
        </ol>
      </div>

      <div>
        <h2>Drinks</h2>
        ...
```

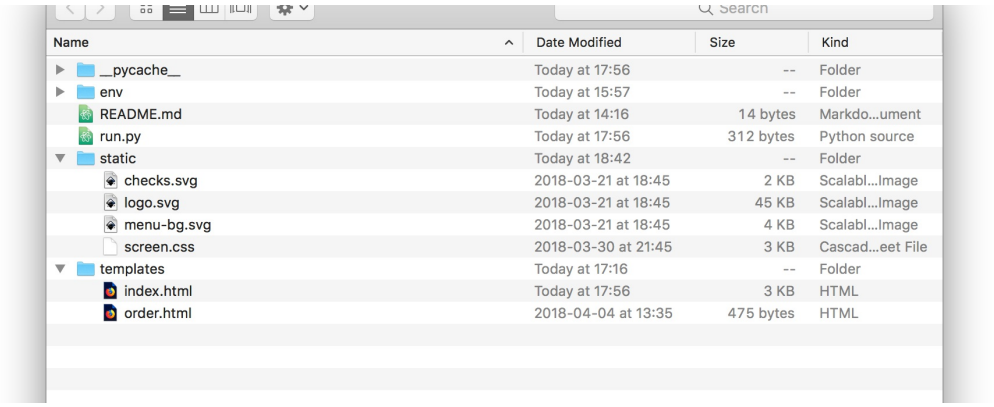
The result will appear the same -- but when you re-connect the stylesheet, the price is wrapped with its own class and can, therefore, be targeted separately by CSS (to right-align it).

Now give the *Drinks* and *Sides* categories the same treatment -- creating a Python list for each and replacing the HTML with `{% for ...` structures.

Links and Static files

Static files are those which require no processing. For example, images are simply served-up without Flask having to replace any variables in them.

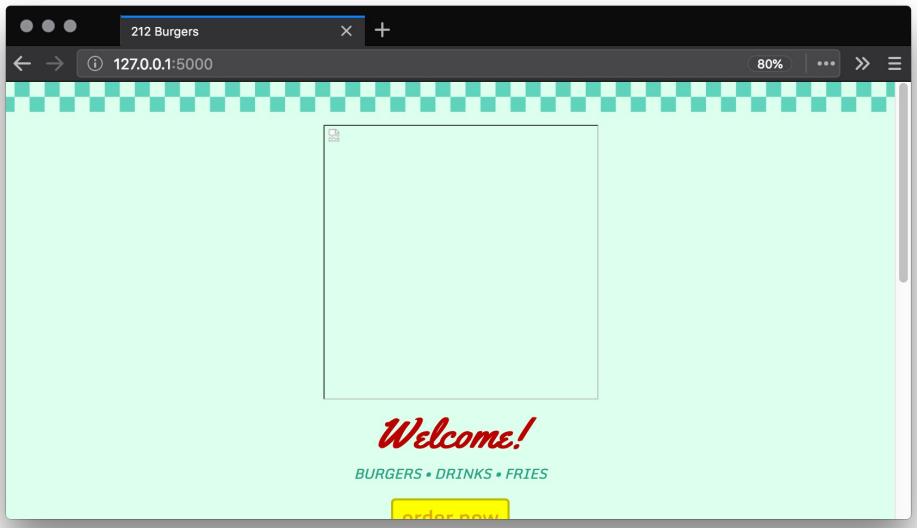
Create a new directory named `static`. Move your CSS and graphics files into this (any JS files would, hypothetically, also go in here).



Now amend the CSS path in your *index.html* document:

```
<!DOCTYPE html>

<html>

  <head>

    ...
    <link rel="stylesheet" href="{{ url_for('static', filename='screen.css') }}" />

  </head>

  ...
```

This reconnects the stylesheet, but the image logo path is still broken:



See if you can fix the logo.

Inheritance

Templates can inherit from other templates. In this case, you will create an order page that inherits all of the properties of the landing page.

Firstly, add a new static link:

```
...

  <div id="header">

    <img src="{{ url_for('static', filename='logo.svg') }}" id="logo" />
    <h1>Welcome!</h1>
    <em>BURGERS &bull; DRINKS &bull; FRIES</em>
    <p>
      <a class="btn-yellow" href="{{ url_for('order') }}>order now</a>
    </p>

  ...
```

The `url_for('order')` will automatically insert the route that matches `order` -- but as this route has yet to be added, Flask will throw an error. This will be fixed shortly.

You'll now need to dice-up your HTML code into sensible template components. Create an empty new file and save it in the templates directory as "base.html". Copy and paste in the following code:

```
<!DOCTYPE html>

<html>

  <head>
    {% block head %}
    <meta charset="utf-8">
    <title>{% block title %}{% endblock %}</title>
    <meta name="description" content="24 hour food delivery. Order burgers, drinks, fries, and more!" />
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <link rel="stylesheet" href="{{ url_for('static', filename='screen.css') }}" />
    {% endblock %}
  </head>

  <body>

    {% block content %} blah blah {% endblock %}

    {% block footer %}
    <div id="footer">
      {{ disclaimer|upper }}
    </div>
    {% endblock %}

  </body>

</html>
```

What you should note about the above code is that is actually the *index.html* code **with all of the body content stripped out**. A few other template elements have been added, namely those beginning with `{% block ...`

Now create an empty new "order.html" file. Add the following code to it:

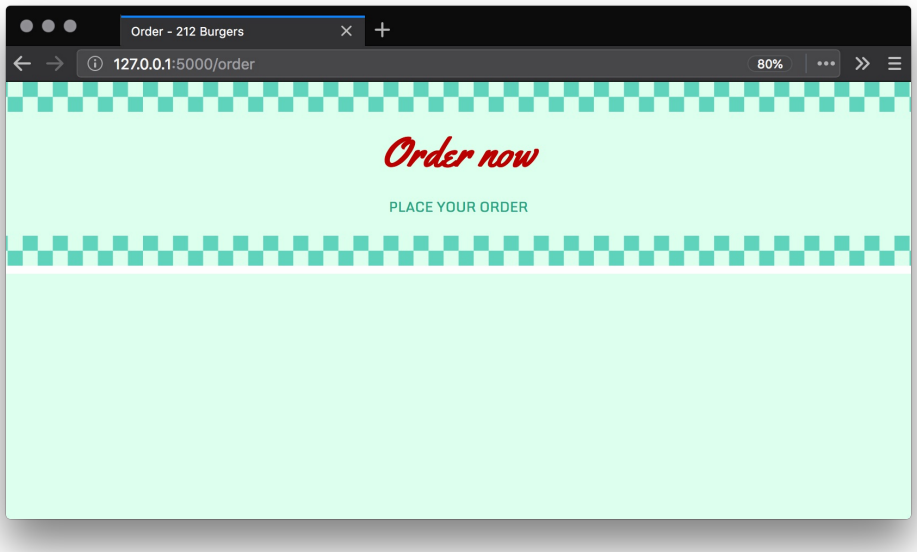
```
{% extends "base.html" %}

{% block title %}Order - 212 Burgers{% endblock %}

{% block head %}
    {{ super() }}
    <style type="text/css">
      body {
        background-color: #DFE;
```

```
    }
    </style>
{% endblock %}

{% block content %}
    <div id="header">
      <h1>Order now</h1>
      <p>PLACE YOUR ORDER</p>
    </div>
{% endblock %}
```

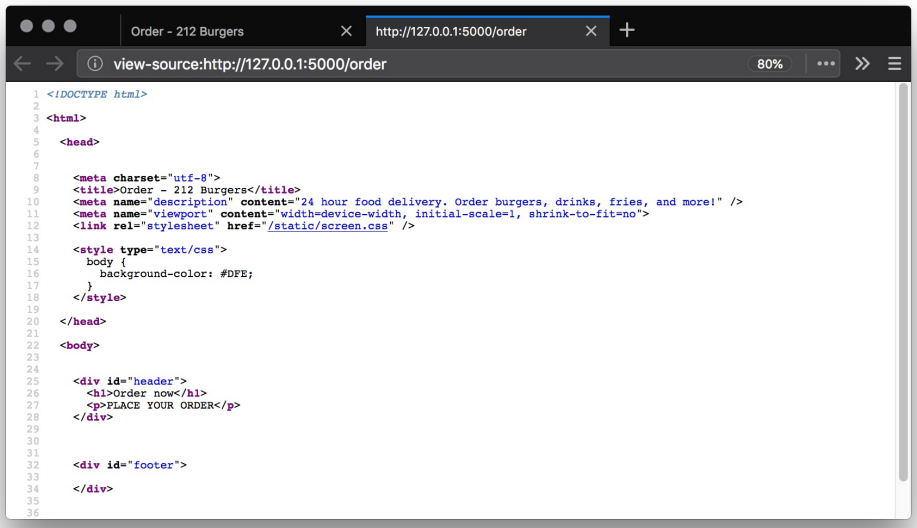
Note how the first line indicates that this extends upon the "base.html" template. Wherever you see `{% block ...`, the code within it replaced by the corresponding template code -- unless there is a `{{ super() }}`, in which case it is appended/added. To make more sense of this concept, add the following route to your *run.py*:

```
...

@app.route('/order')
def order():
    return render_template('order.html')
```

Type `127.0.0.1:5000/order` into the browser address bar and observe the result:



What is more interesting is the source code. Right-click in your browser window and select **View Page Source**:



Study this to understand how Flask has replaced the `{% block ...` code.

Now strip out what you do not need in the *index.html* file, inheriting what you can from the "base.html" template instead. This leaves you with the following code:

index.html

```
{% extends "base.html" %}

{% block title %}212 Burgers{% endblock %}

{% block content %}
    <div id="header">

      <img src="{{ url_for('static', filename='logo.svg') }}" id="logo" />
      <h1>Welcome!</h1>
      <em>BURGERS &bull; DRINKS &bull; FRIES</em>
      <p>
        <a class="btn-yellow" href="{{ url_for('order') }}">order now</a>
      </p>

    </div>

    <div id="menu">

      <h1>Menu</h1>

      <div>
```

```
        <h2>Burgers</h2>
        <ol class="menu-items">
          {% for burger in burgers %}
          <li>{{ burger[0] }} <span class="price">{{ burger[1] }}</span></li>
          {% endfor %}
        </ol>
      </div>

      <div>
        <h2>Drinks</h2>
        <ul class="menu-items">
          {% for drink in drinks %}
          <li>{{ drink[0] }} <span class="price">{{ drink[1] }}</span></li>
          {% endfor %}
        </ul>
      </div>

      <div>
        <h2>Sides</h2>
        <ul class="menu-items">
          {% for side in sides %}
          <li>{{ side[0] }} <span class="price">{{ side[1] }}</span></li>
          {% endfor %}
        </ul>
      </div>

    </div>
{% endblock %}
```

In Closing

In future, to run you Flask server, use the following sequence of terminal commands:

```
cd ~/Desktop/212-Burgers
source env/bin/activate
export FLASK_APP=run.py
export FLASK_DEBUG=1
flask run
```

Of course, the `cd` line will vary depending on the location of your project files.

Do not forget to commit and push your changes.

end