

Quick Intro to Flask Part I

Version 1.03

Flask is a popular lightweight web application framework based on Python. It is designed to make getting started quick and easy, with the ability to scale up to complex applications.

This is the first installment of an introduction to web application programming using Flask. The coverage includes mostly read-only functionality. Future installments shall cover richer interactivity, transactions and basic security.

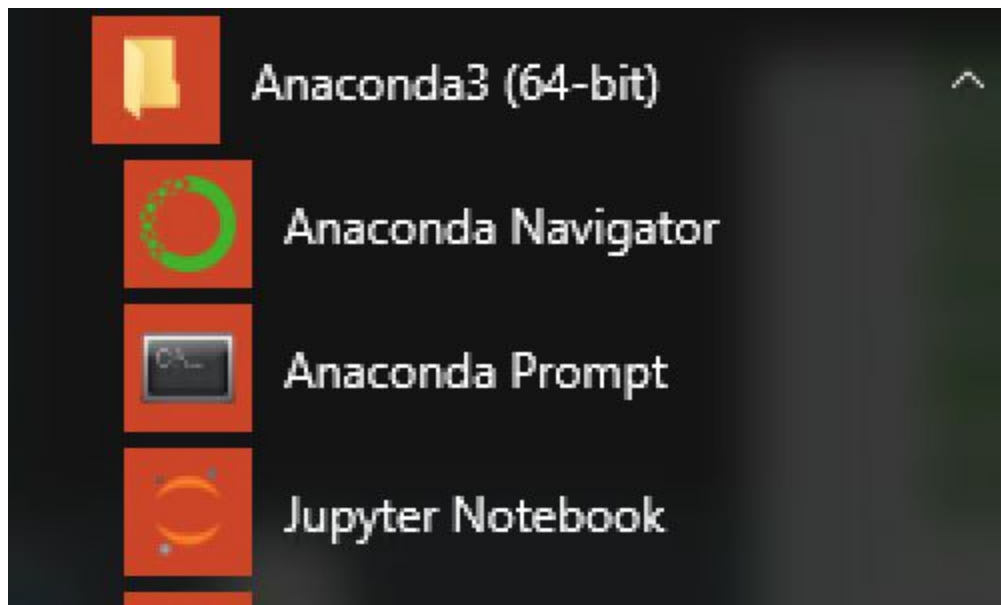
Setup Flask

It is assumed that your Anaconda installations are working properly.

We shall install Flask through Anaconda.

For Windows:

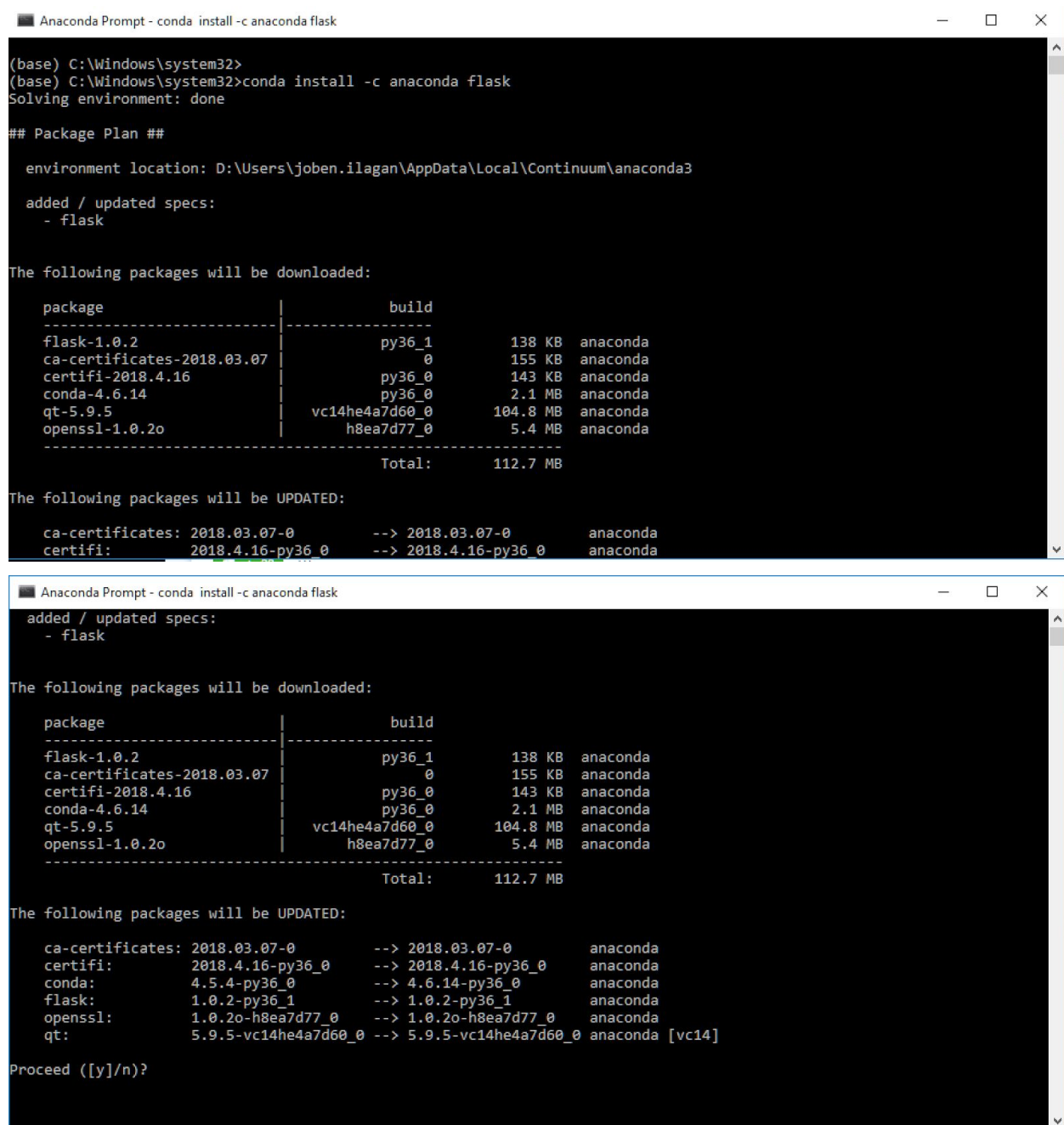
Open the Anaconda Prompt. You will use this for succeeding steps involving Windows Command Prompt.



At the prompt, run the following command:

```
conda install -c anaconda flask
```

You should see messages like the following:



```
Anaconda Prompt - conda install -c anaconda flask

(base) C:\Windows\system32>
(base) C:\Windows\system32>conda install -c anaconda flask
Solving environment: done

## Package Plan ##

  environment location: D:\Users\joben.ilagan\AppData\Local\Continuum\anaconda3

added / updated specs:
- flask

The following packages will be downloaded:

package | build | size | channel
-----|-----|-----|-----
flask-1.0.2 | py36_1 | 138 KB | anaconda
ca-certificates-2018.03.07 | 0 | 155 KB | anaconda
certifi-2018.4.16 | py36_0 | 143 KB | anaconda
conda-4.6.14 | py36_0 | 2.1 MB | anaconda
qt-5.9.5 | vc14he4a7d60_0 | 104.8 MB | anaconda
openssl-1.0.2o | h8ea7d77_0 | 5.4 MB | anaconda
-----|-----|-----|-----
Total: | | 112.7 MB |

The following packages will be UPDATED:

ca-certificates: 2018.03.07-0 --> 2018.03.07-0 anaconda
certifi: 2018.4.16-py36_0 --> 2018.4.16-py36_0 anaconda

The following packages will be downloaded:

package | build | size | channel
-----|-----|-----|-----
flask-1.0.2 | py36_1 | 138 KB | anaconda
ca-certificates-2018.03.07 | 0 | 155 KB | anaconda
certifi-2018.4.16 | py36_0 | 143 KB | anaconda
conda-4.6.14 | py36_0 | 2.1 MB | anaconda
qt-5.9.5 | vc14he4a7d60_0 | 104.8 MB | anaconda
openssl-1.0.2o | h8ea7d77_0 | 5.4 MB | anaconda
-----|-----|-----|-----
Total: | | 112.7 MB |

The following packages will be UPDATED:

ca-certificates: 2018.03.07-0 --> 2018.03.07-0 anaconda
certifi: 2018.4.16-py36_0 --> 2018.4.16-py36_0 anaconda
conda: 4.5.4-py36_0 --> 4.6.14-py36_0 anaconda
flask: 1.0.2-py36_1 --> 1.0.2-py36_1 anaconda
openssl: 1.0.2o-h8ea7d77_0 --> 1.0.2o-h8ea7d77_0 anaconda
qt: 5.9.5-vc14he4a7d60_0 --> 5.9.5-vc14he4a7d60_0 anaconda [vc14]

Proceed ([y]/n)?
```

When asked to proceed, just type **y** and press **Enter**.

You should see in your screen something similar if your installation is successful:

```
Anaconda Prompt
qt-5.9.5          | vc14he4a7d60_0    104.8 MB  anaconda
openssl-1.0.2o    | h8ea7d77_0        5.4 MB    anaconda
-----
Total:            112.7 MB

The following packages will be UPDATED:

ca-certificates: 2018.03.07-0 --> 2018.03.07-0    anaconda
certifi:         2018.4.16-py36_0 --> 2018.4.16-py36_0    anaconda
conda:           4.5.4-py36_0 --> 4.6.14-py36_0    anaconda
flask:           1.0.2-py36_1 --> 1.0.2-py36_1    anaconda
openssl:         1.0.2o-h8ea7d77_0 --> 1.0.2o-h8ea7d77_0    anaconda
qt:              5.9.5-vc14he4a7d60_0 --> 5.9.5-vc14he4a7d60_0    anaconda [vc14]

Proceed ([y]/n)? y

Downloading and Extracting Packages
flask-1.0.2          | 138 KB | ##### | 100%
ca-certificates-2018 | 155 KB | ##### | 100%
certifi-2018.4.16    | 143 KB | ##### | 100%
conda-4.6.14         | 2.1 MB | ##### | 100%
qt-5.9.5             | 104.8 MB | ##### | 100%
openssl-1.0.2o       | 5.4 MB | ##### | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done

(base) C:\Windows\system32>
```

For Mac:

Open Mac OS Terminal and type the following command:

```
conda install -c anaconda flask
```

```
JobenIlagan — conda install -c anaconda flask — 80x24
[(base) JobenMacbookPro:~ JobenIlagan$ conda install -c anaconda flask
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /anaconda3

  added / updated specs:
    - flask

The following packages will be downloaded:

package | build | size | channel
-----|-----|-----|-----
ca-certificates-2019.10.16 | 0 | 131 KB | anaconda
certifi-2019.9.11 | py37_0 | 154 KB | anaconda
conda-4.7.12 | py37_0 | 3.0 MB | anaconda
-----|-----|-----|-----
Total: | | 3.3 MB |

The following packages will be UPDATED:
```

When asked to proceed, just type **y** and press **Enter**.

```
JobenIlagan — conda install -c anaconda flask — 80x24

package | build | size | channel
-----|-----|-----|-----
ca-certificates-2019.10.16 | 0 | 131 KB | anaconda
certifi-2019.9.11 | py37_0 | 154 KB | anaconda
conda-4.7.12 | py37_0 | 3.0 MB | anaconda
-----|-----|-----|-----
Total: | | 3.3 MB |

The following packages will be UPDATED:

 flask                anaconda/osx-64::flask-1.0.2-py37_1 --> anaconda/noarch:
:flask-1.1.1-py_0
 openssl             pkgs/main::openssl-1.1.1d-h1de35cc_3 --> anaconda::openss
l-1.1.1-h1de35cc_0

The following packages will be SUPERSEDED by a higher-priority channel:

ca-certificates      pkgs/main --> anaconda
certifi              pkgs/main --> anaconda
conda                pkgs/main --> anaconda

Proceed ([y]/n)? █
```

You should see in your screen something similar if your installation is successful:

```
JobenIlagan — -bash — 80x24

flask                               anaconda/osx-64::flask-1.0.2-py37_1 --> anaconda/noarch:
:flask-1.1.1-py_0
openssl                             pkgs/main::openssl-1.1.1d-h1de35cc_3 --> anaconda::openss
l-1.1.1-h1de35cc_0

The following packages will be SUPERSEDED by a higher-priority channel:

ca-certificates                     pkgs/main --> anaconda
certifi                             pkgs/main --> anaconda
conda                              pkgs/main --> anaconda

Proceed ([y]/n)? y

Downloading and Extracting Packages
conda-4.7.12           | 3.0 MB      | ##### | 100%
ca-certificates-2019  | 131 KB     | ##### | 100%
certifi-2019.9.11     | 154 KB     | ##### | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
(base) JobenMacbookPro:~ JobenIlagan$
```

Hello World

Exercise: Setting up your first Flask Project

1. Open your MacOS Terminal or Windows Shell. Go to a folder or subdirectory you will use throughout this tutorial.

Create your project folder or subdirectory:

```
$ mkdir digitalcafe
$ cd digitalcafe
```

If you are on Windows Command Prompt:

```
C:\path\to\app>set FLASK_APP=app.py
C:\path\to\app>set FLASK_ENV=development
```

If you are on Windows PowerShell:

```
PS C:\path\to\app> $env:FLASK_APP = "app.py"
PS C:\path\to\app> $env:FLASK_ENV = "development"
```

If you are on Mac

```
$ export FLASK_APP=app.py
$ export FLASK_ENV=development
```

2. Create your first app:

Using your text editor, create a file **app.py** with the following contents:

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

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

Save your file.

To run your first application, go back to Windows Terminal, Windows Power Shell or Mac OS Terminal:

```
$ flask run
```

You should see messages similar to the one below:

On Windows:


```
Anaconda Prompt - flask run

2 Dir(s) 29,873,467,392 bytes free

(base) D:\Users\joben.ilagan\src>mkdir digitalcafe
(base) D:\Users\joben.ilagan\src>cd digitalcafe
(base) D:\Users\joben.ilagan\src\digitalcafe>dir
Volume in drive D is UserProfile
Volume Serial Number is FC80-D08A

Directory of D:\Users\joben.ilagan\src\digitalcafe

10/24/2019 08:48 AM <DIR>      .
10/24/2019 08:48 AM <DIR>      ..
               0 File(s)      0 bytes
               2 Dir(s) 29,868,429,312 bytes free

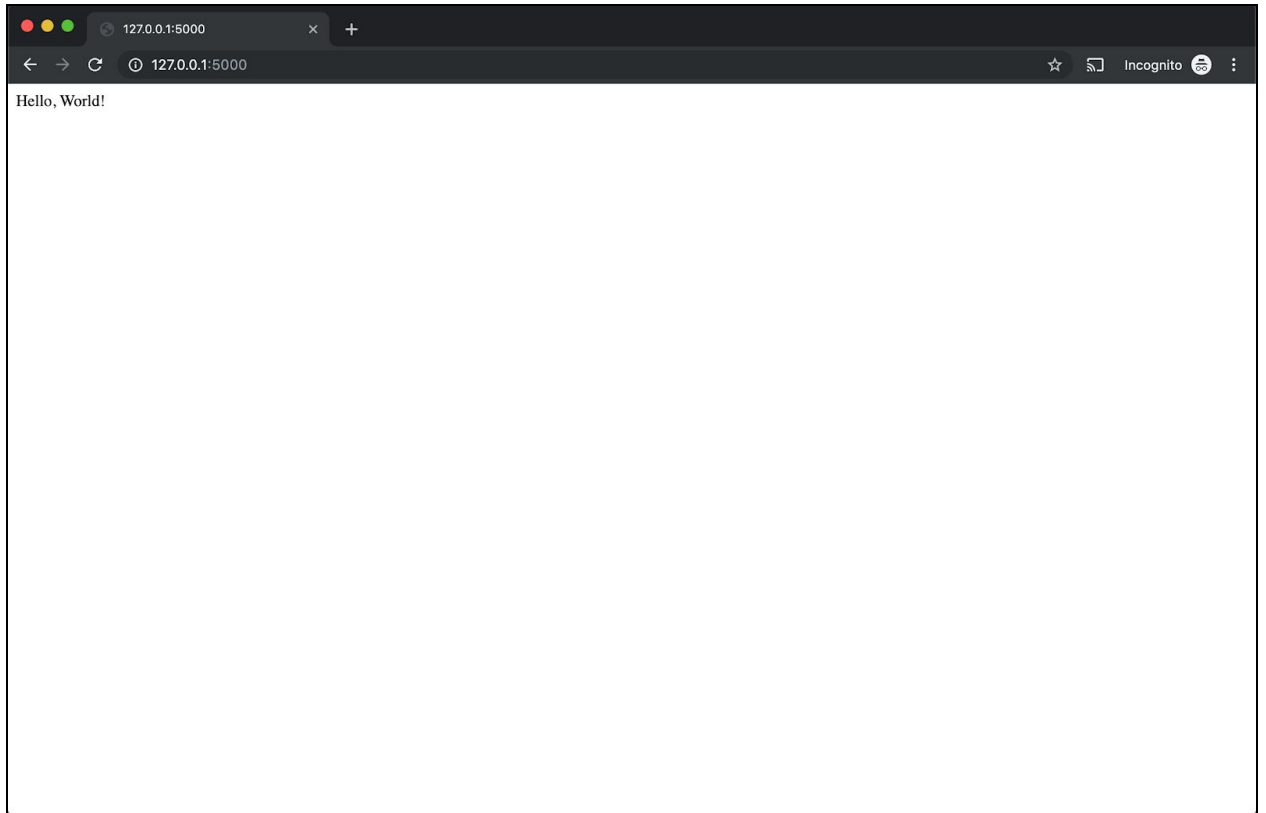
(base) D:\Users\joben.ilagan\src\digitalcafe>set FLASK_APP=app.py
(base) D:\Users\joben.ilagan\src\digitalcafe>set FLASK_ENV=development
(base) D:\Users\joben.ilagan\src\digitalcafe>flask run
* Serving Flask app "app.py" (lazy loading)
* Environment: development
* Debug mode: on
* Restarting with stat
* Debugger is active!
* Debugger PIN: 217-973-353
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

On Mac:

```
* Serving Flask app "app.py" (lazy loading)
* Environment: development
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 809-151-460
```

Open your web browser and type <http://127.0.0.1:5000/> in the URL. Note that the IP address 127.0.0.1 points to your local machine and 5000 is the TCP/IP Port Number. Most websites use Port 80, and this is the default port used by the web browsers.

You should see something like the following:



Something more useful

Exercise: Define routes

Modern web applications use meaningful Universal Resource Locators (URLs) to help users navigate through pages or screens.

In Flask, we use a **route** decorator to bind a function to a URL.

Edit **app.py** and replace the contents with the following:

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

@app.route('/')
def index():
    return 'Index Page. Place Home Page contents here.'

@app.route('/products')
```

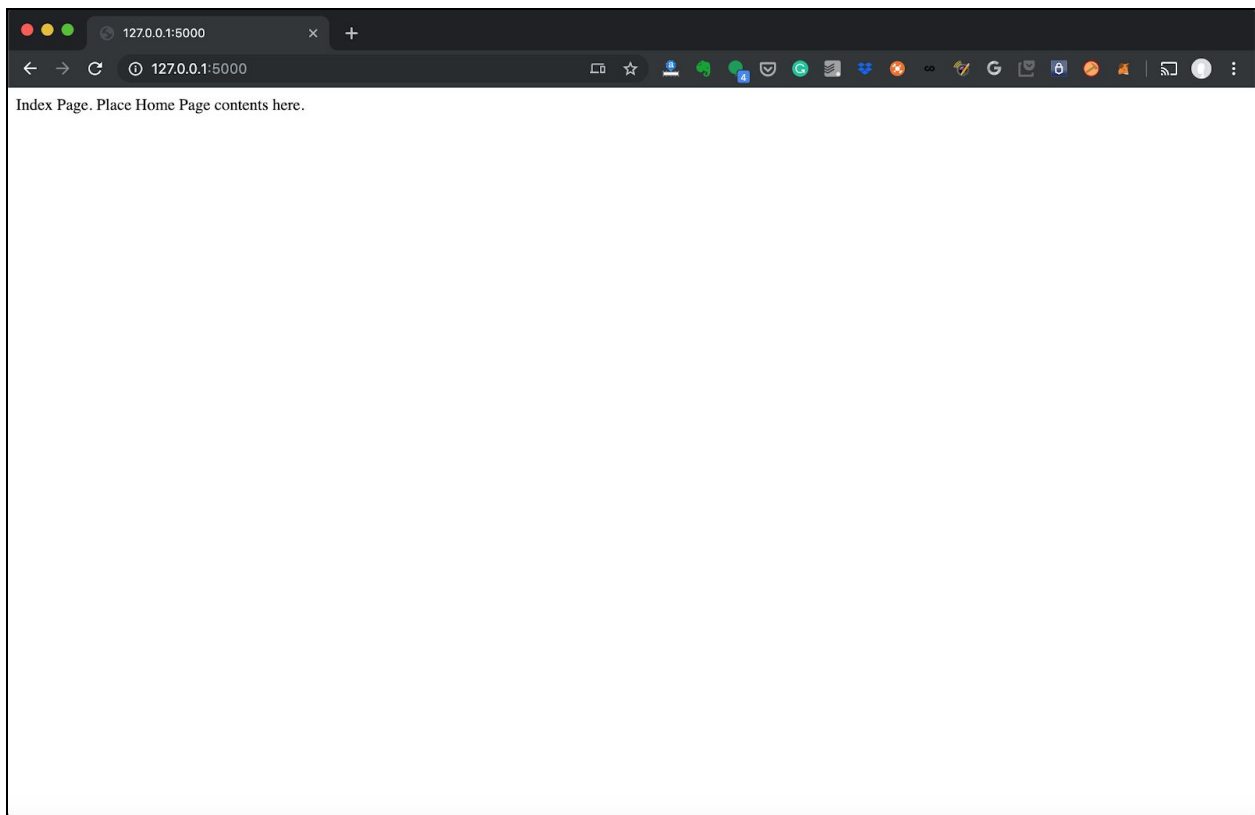


```
def products():  
    return 'Products Page. Place Products Page contents here.'  
  
@app.route('/branches')  
def branches():  
    return 'Branches Page. Place Branch Page contents here.'  
  
@app.route('/aboutus')  
def aboutus():  
    return 'About Us Page. Place About Us Page contents here.'
```

Save your changes. Since we are in development mode, there's no need to restart Flask.

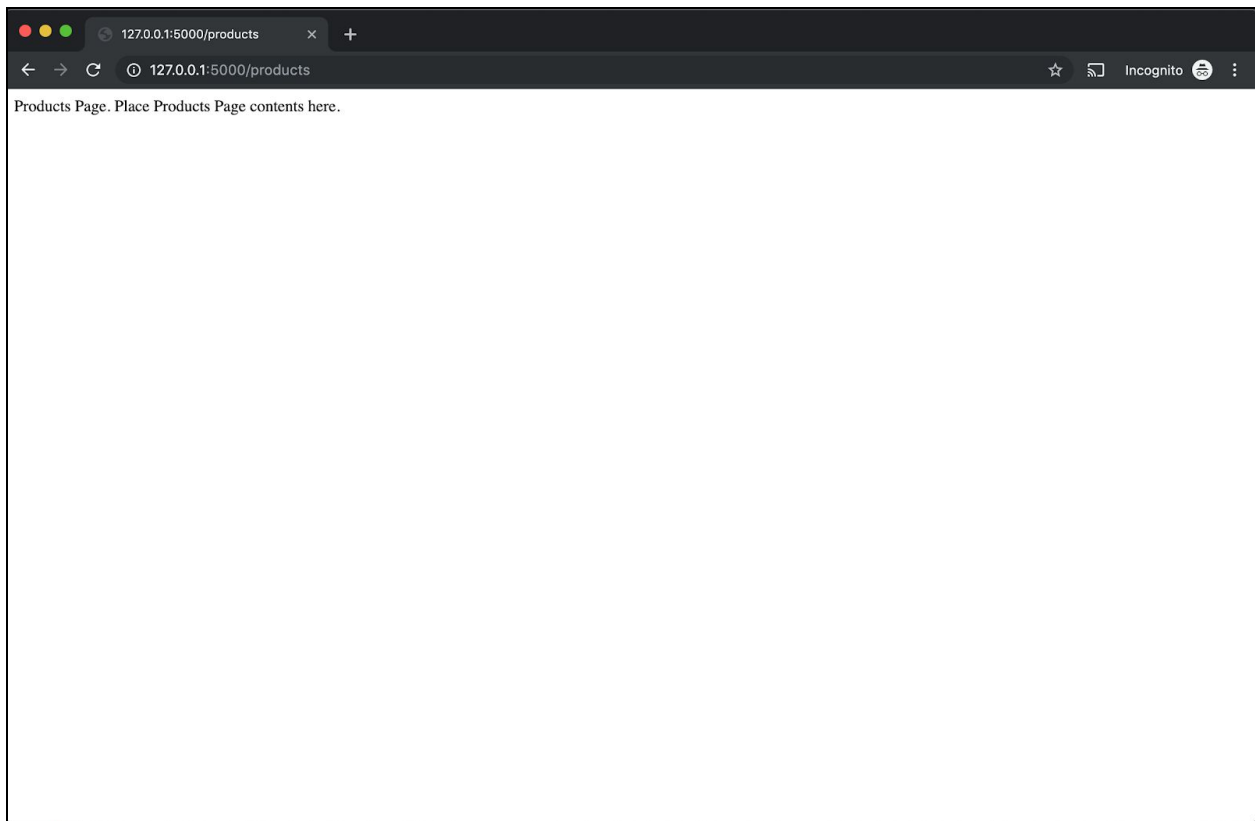
Reload the URL in your web browser.

Type <http://127.0.0.1:5000/> in the URL. You should see the following page:



Let's try another URL; this time, **/products**.

Type <http://127.0.0.1:5000/products> in the URL. You should see the following page:



Test the other two URLs:

<http://127.0.0.1:5000/aboutus>

<http://127.0.0.1:5000/branches>

Exercise: Review HTML tags from our previous lesson

1. Edit **app.py** to include test anchors/links pointing to URL routes defined in the previous section.

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

navbar = """
    <a href='/'>Home</a> | <a href='/products'>Products</a> |
    <a href='/branches'>Branches</a> | <a href='/aboutus'>About
    Us</a>
    <p/>
```

```

"""

@app.route('/')
def index():
    pagecontent = 'Index Page. Place Home Page contents here.'
    return navbar+pagecontent

@app.route('/products')
def products():
    pagecontent = 'Products Page. Place Products Page contents here.'
    return navbar+pagecontent

@app.route('/branches')
def branches():
    pagecontent = 'Branches Page. Place Branches Page contents here.'
    return navbar+pagecontent

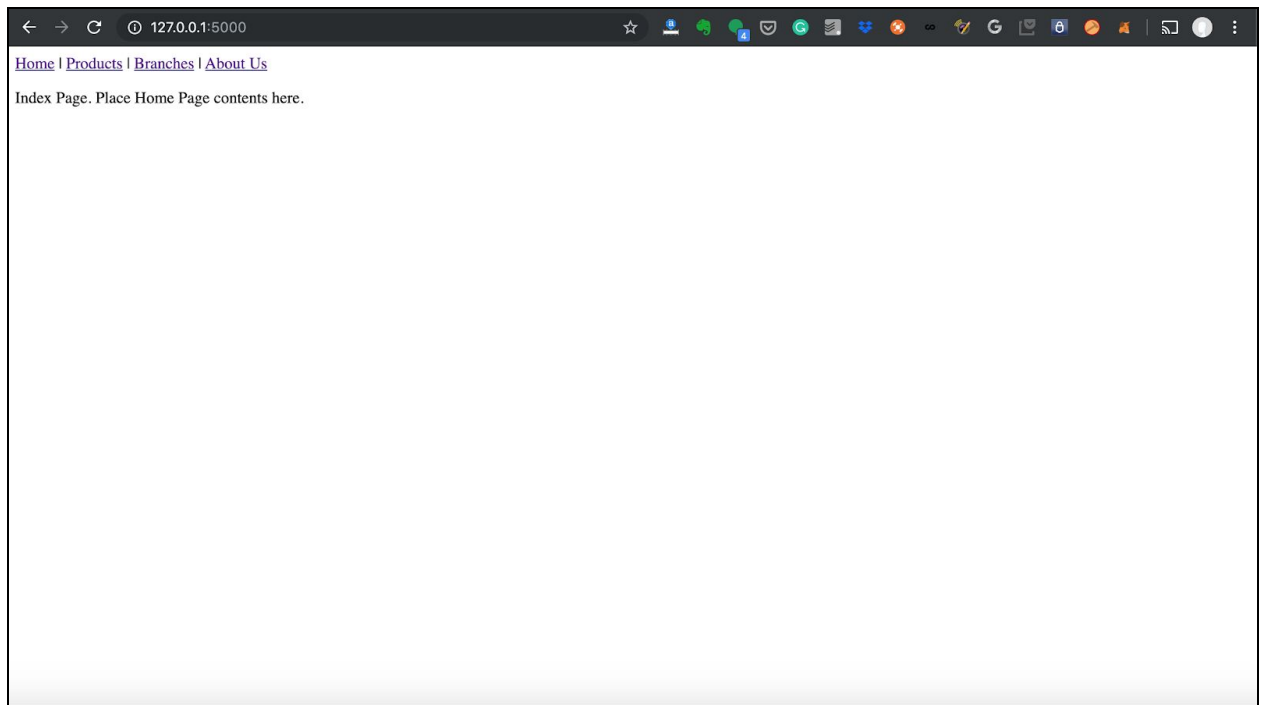
@app.route('/aboutus')
def aboutus():
    pagecontent = 'About Us Page. Place About Us Page contents here.'
    return navbar+pagecontent

```

Save your changes. There's no need to restart Flask since we are under **development** mode.

Reload the URL in your web browser.

Type <http://127.0.0.1:5000/> in the URL. You should see the following page:



Click through **Products**, **Branches** and **About Us**. Notice the resulting pages.

Templates

Flask configures the **Jinja2 template engine** for you automatically.

To render a template you can use the `render_template()` method. All you have to do is provide the name of the template and the variables you want to pass to the template engine as keyword arguments.

Exercise: render a template

1. Create a subdirectory named **templates** under the app directory **digitalcafe**.

```
$ mkdir templates
```

2. Create a template file **app.html** under the **templates** subdirectory with the following contents:

```
<html>
  <header>
```

```

</header>
<body>
    <a href='/'>Home</a> | <a href='/products'>Products</a>
|
    <a href='/branches'>Branches</a> | <a
href='/aboutus'>About Us</a>
    <p/>
    <p>{{ page }} Page. </p>
</body>
</html>

```

Note that `{{ page }}` is a template variable we can use to pass data from our app for rendering.

3. Update **app.py** with the following contents:

```

from flask import Flask
from flask import render_template

app = Flask(__name__)

navbar = """
    <a href='/'>Home</a> | <a href='/products'>Products</a> |
    <a href='/branches'>Branches</a> | <a href='/aboutus'>About
Us</a>
    <p/>
    """

@app.route('/')
def index():
    return render_template('app.html', page="Index")

@app.route('/products')
def products():
    return render_template('app.html', page="Products")

@app.route('/branches')
def branches():
    return render_template('app.html', page="Branches")

@app.route('/aboutus')

```

```
def aboutus():  
    return render_template('app.html', page="About Us")
```

Save the changes to the file. Reload <http://127.0.0.1:5000> in your web browser. Navigate through **Products**, **Branches** and **About Us**. Notice the resulting pages.

4. What if we have URL-specific page layout requirements?

Prepare the following template files under the **templates** directory or folder:

index.html

```
<html>  
  <header>  
  </header>  
  <body>  
    <a href='/'>Home</a> | <a href='/products'>Products</a>  
    |  
    <a href='/branches'>Branches</a> | <a  
href='/aboutus'>About Us</a>  
    <p/>  
    <h1>Home</h1>  
  </body>  
</html>
```

products.html

```
<html>  
  <header>  
  </header>  
  <body>  
    <a href='/'>Home</a> | <a href='/products'>Products</a>  
    |  
    <a href='/branches'>Branches</a> | <a  
href='/aboutus'>About Us</a>  
    <p/>  
    <h1>Products</h1>  
  
    <table>  
      <tr><th>Code</th><th>Name</th><th>Price</th></tr>
```



```

        <tr><td><a
href="productdetails?code=100">100</a></td><td>Americano</td><t
d>125</td></tr>
        <tr><td><a
href="productdetails?code=200">200</a></td><td>Brewed
Coffee</td><td>100</td></tr>
        <tr><td><a
href="productdetails?code=300">300</a></td><td>Cappuccino</td><
td>120</td></tr>
        <tr><td><a
href="productdetails?code=400">400</a></td><td>Espresso</td><td
>120</td></tr>
    </table>
</body>
</html>

```

branches.html

```

<html>
  <header>
  </header>
  <body>
    <a href="/">Home</a> | <a href="/products">Products</a>
    |
    <a href="/branches">Branches</a> | <a
href="/aboutus">About Us</a>
    <p/>
    <h1>Branches</h1>
  </body>
</html>

```

aboutus.html

```

<html>
  <header>
  </header>
  <body>
    <a href="/">Home</a> | <a href="/products">Products</a>
    |

```

```

        <a href='/branches'>Branches</a> | <a
href='/aboutus'>About Us</a>
        <p/>
        <h1>About Us</h1>
    </body>
</html>

```

5. Update **app.py** to point to the new template files:

```

from flask import Flask
from flask import render_template

app = Flask(__name__)

navbar = """
    <a href='/'>Home</a> | <a href='/products'>Products</a> |
    <a href='/branches'>Branches</a> | <a href='/aboutus'>About
Us</a>
    <p/>
    """

@app.route('/')
def index():
    return render_template('index.html', page="Index")

@app.route('/products')
def products():
    return render_template('products.html', page="Products")

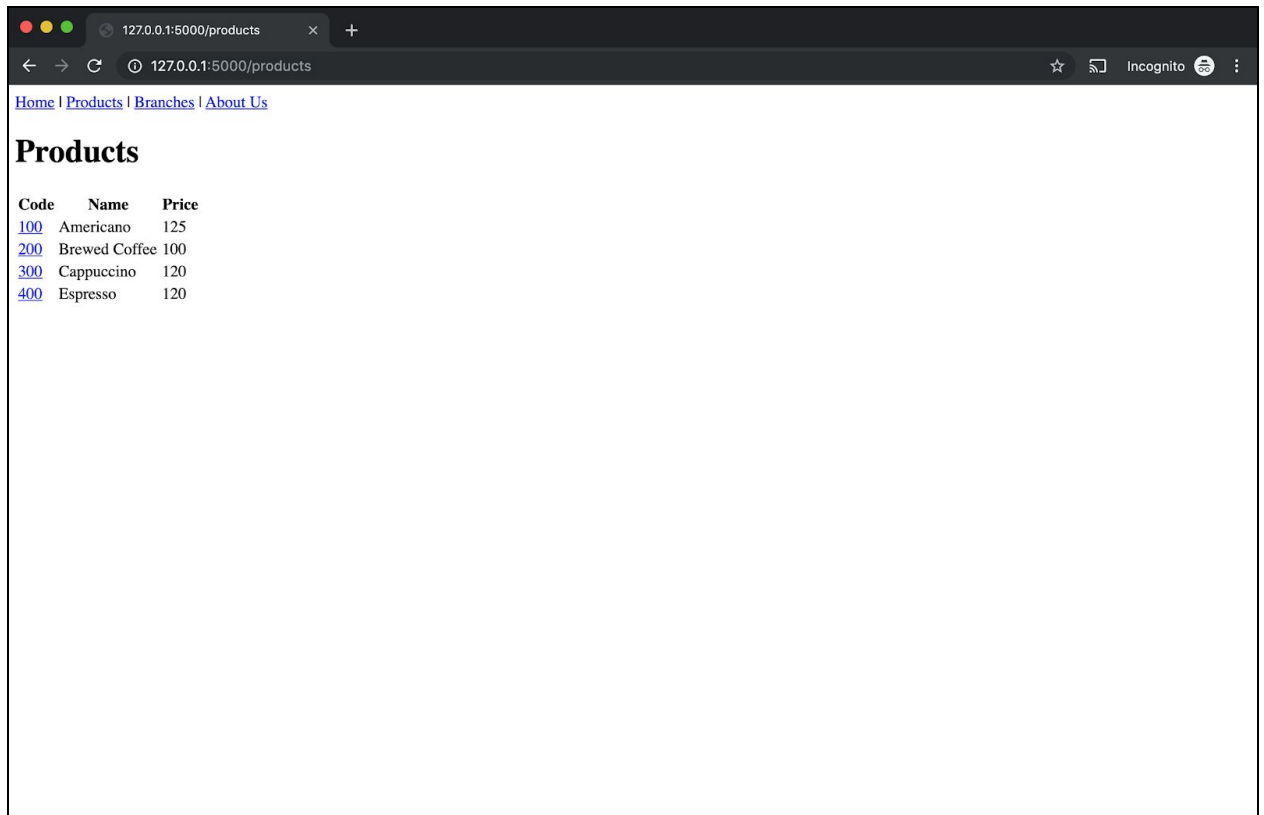
@app.route('/branches')
def branches():
    return render_template('branches.html', page="Branches")

@app.route('/aboutus')
def aboutus():
    return render_template('aboutus.html', page="About Us")

```

Save your changes and reload <http://127.0.0.1:5000> in your web browser. Navigate through the links.

For the Products page, you should see something like this:



Try clicking on Product Code **100** (Americano). We get an error. Let's fix this in the next section.

Accessing Request Data

Background Information:

What is HTTP?

The Hypertext Transfer Protocol (HTTP) is designed to enable communications between clients and servers.

HTTP works as a request-response protocol between a client and server.

A web browser may be the client, and an application on a computer that hosts a web site may be the server.

Example: A client (browser) submits an HTTP request to the server; then the server returns a response to the client. The response contains status information about the request and may

also contain the requested content.

Source: https://www.w3schools.com/tags/ref_httpmethods.asp

HTTP Request Methods

- **GET**
- **POST**
- PUT
- DELETE
- PATCH
- OPTIONS

For this session, we shall only work with the first two (GET and POST).

GET is used to request data from a specified resource.

Note that the query string (name/value pairs) is sent in the URL of a GET request:

<http://example.com/productdetails?code=100>

The more general form is as follows:

http://example.com/test/demo_form.php?name1=value1&name2=value2

POST is used to send data to a server to create/update a resource.

The data sent to the server with POST is stored in the request body of the HTTP request:

```
POST /buy HTTP/1.1
Host: example.com
code=100&qty=1
```

We will not deal with POST transactions in this session.

Flask can react to data a client sends to the server via the **global request object**.

We will need to import **request** as follows:

```
from flask import request
```

The current request method is available by using the `method` attribute. To access form data (data transmitted in a POST or PUT request) you can use the `form` attribute.

To access parameters submitted in the URL (?key=value) you can use the `args` attribute:

```
code = request.args.get('code', '')
```

Exercise: handle URL parameters

1. Create a new file **productdetails.html** under the **templates** folder/directory.

```
<html>
  <header>
  </header>
  <body>
    <a href="/">Home</a> | <a href="/products">Products</a>
  |
    <a href="/branches">Branches</a> | <a
href="/aboutus">About Us</a>
    <p/>
    <h1>Product Details</h1>
    <table>
      <tr><td>Code</td><td>{{ code }}</td></tr>
      <tr><td>Description</td><td><i>Place description
here</i></td></tr>
      <tr><td>Price</td><td><i>999</i></td></tr>
    </table>
    <a href="/products">Back</a>
  </body>
</html>
```

2. Modify **app.py** to process the request for product details:

```
from flask import Flask
from flask import render_template
from flask import request

app = Flask(__name__)

navbar = ""
```

```

        <a href="/">Home</a> | <a href="/products">Products</a> |
        <a href="/branches">Branches</a> | <a href="/aboutus">About
Us</a>
    </p>
    """

@app.route('/')
def index():
    return render_template('index.html', page="Index")

@app.route('/products')
def products():
    return render_template('products.html', page="Products")

@app.route('/productdetails')
def productdetails():
    code = request.args.get('code', '')
    return render_template('productdetails.html', code=code)

@app.route('/branches')
def branches():
    return render_template('branches.html', page="Branches")

@app.route('/aboutus')
def aboutus():
    return render_template('aboutus.html', page="About Us")

```

Save your changes and reload the URL <http://127.0.0.1:5000> in your browser. Navigate through the links.

Apply HTML Headers and Footers

Exercise: apply standard headers and footers across all pages

1. Create a file **header.html** in the **templates** folder.

```

<html>
  <header>
  </header>
  <body>
    <a href="/">Home</a> | <a href="/products">Products</a> |

```



```
    <a href='/branches'>Branches</a> | <a href='/aboutus'>About  
    Us</a>  
  </p>
```

2. Create a file **footer.html** in the **templates** folder.

```
  <div>  
    Copyright (c) 2019 Digital Coffee Inc. All rights reserved.  
  </p>  
    View our privacy policy <a href="privacy.html">here</a>.  
  </div>  
</body>  
</html>
```

3. Update the main html templates.

aboutus.html

```
{% include "header.html" %}  
    <h1>About Us</h1>  
{% include "footer.html" %}
```

branches.html

```
{% include "header.html" %}  
    <h1>Branches</h1>  
{% include "footer.html" %}
```

index.html

```
{% include "header.html" %}  
    <h1>Home</h1>  
{% include "footer.html" %}
```

productdetails.html

```
{% include "header.html" %}  
    <h1>Product Details</h1>  
    <table>  
      <tr><td>Code</td><td>{{ code }}</td></tr>
```

```

        <tr><td>Description</td><td><i>Place description
here</i></td></tr>
        <tr><td>Price</td><td><i>999</i></td></tr>
    </table>
    <a href="/products">Back</a>
{% include "footer.html" %}

```

products.html

```

{% include "header.html" %}
    <h1>Products</h1>

    <table>
        <tr><th>Code</th><th>Name</th><th>Price</th></tr>
        <tr><td><a
href="productdetails?code=100">100</a></td><td>Americano</td><t
d>125</td></tr>
        <tr><td><a
href="productdetails?code=200">200</a></td><td>Brewed
Coffee</td><td>100</td></tr>
        <tr><td><a
href="productdetails?code=300">300</a></td><td>Cappuccino</td><
td>120</td></tr>
        <tr><td><a
href="productdetails?code=400">400</a></td><td>Espresso</td><td
>120</td></tr>
    </table>
{% include "footer.html" %}

```

Save your changes and reload the URL <http://127.0.0.1:5000> in your browser. Navigate through the links. There shouldn't be any visual changes at this point.

Checkpoint: Apply Bootstrap

Now that we have a central header location, it will be easy to apply css across all pages.

Exercise: Include bootstrap css and style directives

1. Update **header.html** with the following:

```

<html>
<header>

```

```

    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/boot
strap.min.css"
integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhc
Wr7x9JvoRxT2MZw1T" crossorigin="anonymous"/>
    <style>
        .footer {
            font-size:10px;
        }
    </style>
</header>
<body>
    <a href="/">Home</a> | <a href="/products">Products</a> |
    <a href="/branches">Branches</a> | <a href="/aboutus">About
Us</a>
    <p/>

```

2. Update footer.html

```

<div class="footer">
    Copyright (c) 2019 Digital Coffee Inc. All rights reserved.
</div>
    View our privacy policy <a href="privacy.html">here</a>.
</div>
</body>
</html>

```

3. Save header.html and footer.html. Reload and test pages.

Helper Functions

A **helper function** is a **function** that performs part of the computation of another **function**.

Helper functions are used to make your programs easier to read by giving descriptive names to computations. They also let you reuse computations, just as with **functions** in general.

Source: *Creating Helper Functions*

(<https://web.cs.wpi.edu/~cs1101/a05/Docs/creating-helpers.html>)

We will be using helper functions to mimic access to a database. The of mimicking actual functionality without actually implementing the function itself is through a mechanism known as a **mock function**.

For prototyping purposes, we will hard-code return values of database helper or mock functions. This is one of the techniques used by startups to come up with demonstrable minimum viable products. As fleshing out the real implementation requires time and effort and usually involves database design and management, the decision on whether or not to proceed with implementation is deferred up until after the establishment or rejection of product-market fit.

Exercise: Create a starter helper module

1. We shall mimic a database by creating a small dictionary of our products in server memory. Create a file **database.py** under the **digitalcafe** folder/directory with the following lines of code:

```
products = {
    100: {"name": "Americano", "price": 125},
    200: {"name": "Brewed Coffee", "price": 100},
    300: {"name": "Cappuccino", "price": 120},
    400: {"name": "Espresso", "price": 120}
}
```

```
def get_product(code):
    return products[code]
```

Saved your changes.

2. Update **app.py** and replace the contents with the following lines of code:

```
from flask import Flask
from flask import render_template
from flask import request
import database as db

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html', page="Index")

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

```

        return render_template('products.html', page="Products")

@app.route('/productdetails')
def productdetails():
    code = request.args.get('code', '')
    product = db.get_product(int(code))

    return render_template('productdetails.html', code=code,
product=product)

@app.route('/branches')
def branches():
    return render_template('branches.html', page="Branches")

@app.route('/aboutus')
def aboutus():
    return render_template('aboutus.html', page="About Us")

```

Save your changes.

3. Replace the contents of **productdetails.html** under the **templates** folder or directory with the following lines of code:

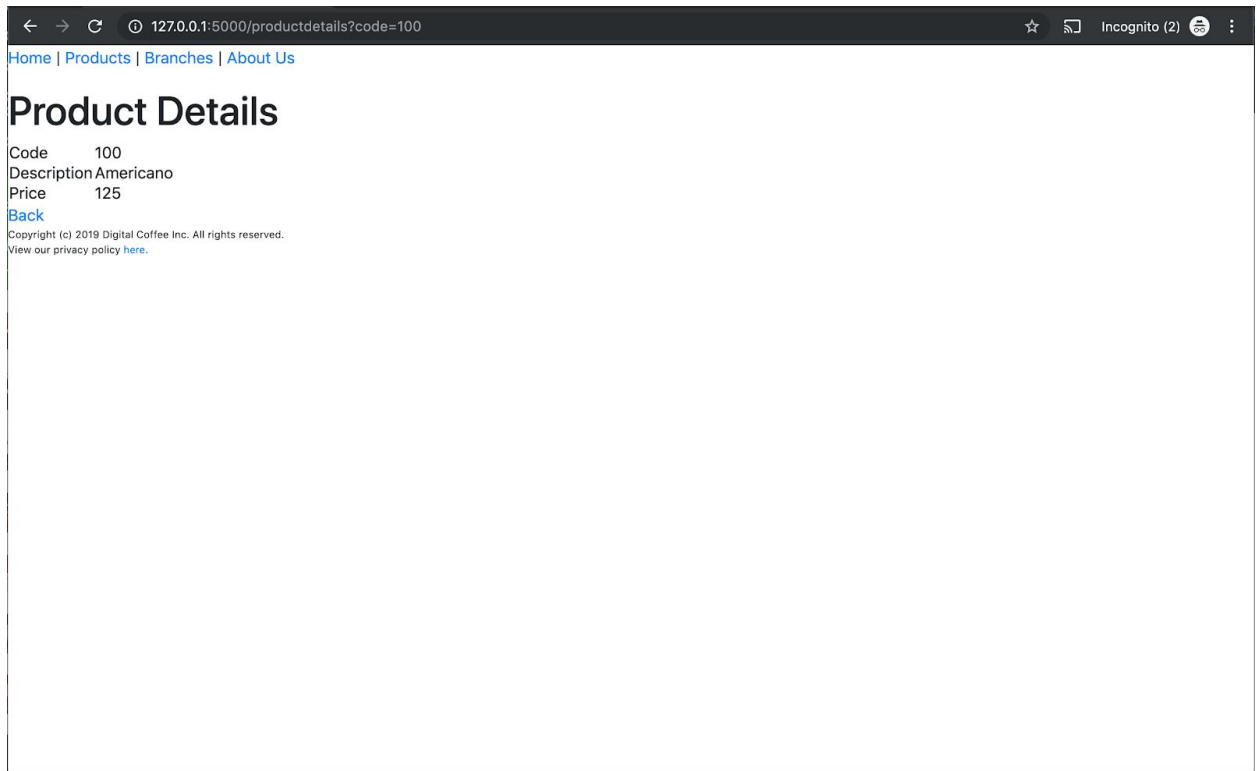
```

{% include "header.html" %}
<h1>Product Details</h1>
<table>
    <tr><td>Code</td><td>{{ code }}</td></tr>
    <tr><td>Description</td><td>{{ product["name"]
}}</td></tr>
    <tr><td>Price</td><td>{{ product["price"]
}}</td></tr>
</table>
<a href="/products">Back</a>
{% include "footer.html" %}

```

Save your changes and reload the URL in your web browser. Navigate to Products and click on the code of Americano (100).

You should see something like this:



Exercise: Generate a dynamic list of products

What if the list of products and certain product details change? In the real world, these changes are done in a database and that the web application simply retrieves whatever the database contains. For this exercise, we shall make changes to the products dictionary contents and have our web application pick up these changes.

While we're at it, we shall replace the hard-coded values in our **products.html** template with data retrieved from our mock database. This way, we don't need to change our web application code every time our database entries are updated.

Note that since we are still using mock functions, the product definitions are still hard-coded. In future lessons, we shall discard these mock functions altogether and access an actual database.

1. Create a helper function that returns a list of products. Update **database.py** with this new function. The contents of **database.py** should now have the following:

```
products = {
    100: {"name": "Americano", "price": 125},
    200: {"name": "Brewed Coffee", "price": 100},
    300: {"name": "Cappuccino", "price": 120},
```



```

        400: {"name": "Espresso", "price": 120}
    }

def get_product(code):
    return products[code]

def get_products():
    product_list = []

    for i, v in products.items():
        product = v
        product.setdefault("code", i)
        product_list.append(product)

    return product_list

```

Save your changes.

2. Update the function **products()** in **app.py**. The new code for the whole file should be the following:

```

from flask import Flask
from flask import render_template
from flask import request
import database as db

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html', page="Index")

@app.route('/products')
def products():
    product_list = db.get_products()
    return render_template('products.html', page="Products",
        product_list=product_list)

@app.route('/productdetails')
def productdetails():
    code = request.args.get('code', '')
    product = db.get_product(int(code))

```

```

        return render_template('productdetails.html', code=code,
                                product=product)

@app.route('/branches')
def branches():
    return render_template('branches.html', page="Branches")

@app.route('/aboutus')
def aboutus():
    return render_template('aboutus.html', page="About Us")

```

3. Change the **products.html** template. Replace the hard-coded area with a Jinja loop.

```

{% include "header.html" %}
<h1>Products</h1>

<table>
    <tr><th>Code</th><th>Name</th><th>Price</th></tr>
    {% for product in product_list %}
        <tr><td><a
href="productdetails?code={{product["code"]}}">{{product["code"]}}</a></td><td>{{product["name"]}}</td><td>{{product["price"]}}</td></tr>
        {% endfor %}
    </table>
{% include "footer.html" %}

```

Save your changes and reload the URL in your web browser. Navigate to the **Products** page.

You may be wondering what actually changed and what was the big deal with going through all that?

4. Update the mock database with new product entries and price changes. The new code for **database.py** should have the following:

```

products = {
    100: {"name": "Americano", "price": 125},

```

```
200: {"name": "Brewed Coffee", "price": 110},
300: {"name": "Cappuccino", "price": 120},
400: {"name": "Espresso", "price": 120},
500: {"name": "Latte", "price": 140},
600: {"name": "Cold Brew", "price": 200}
}
```

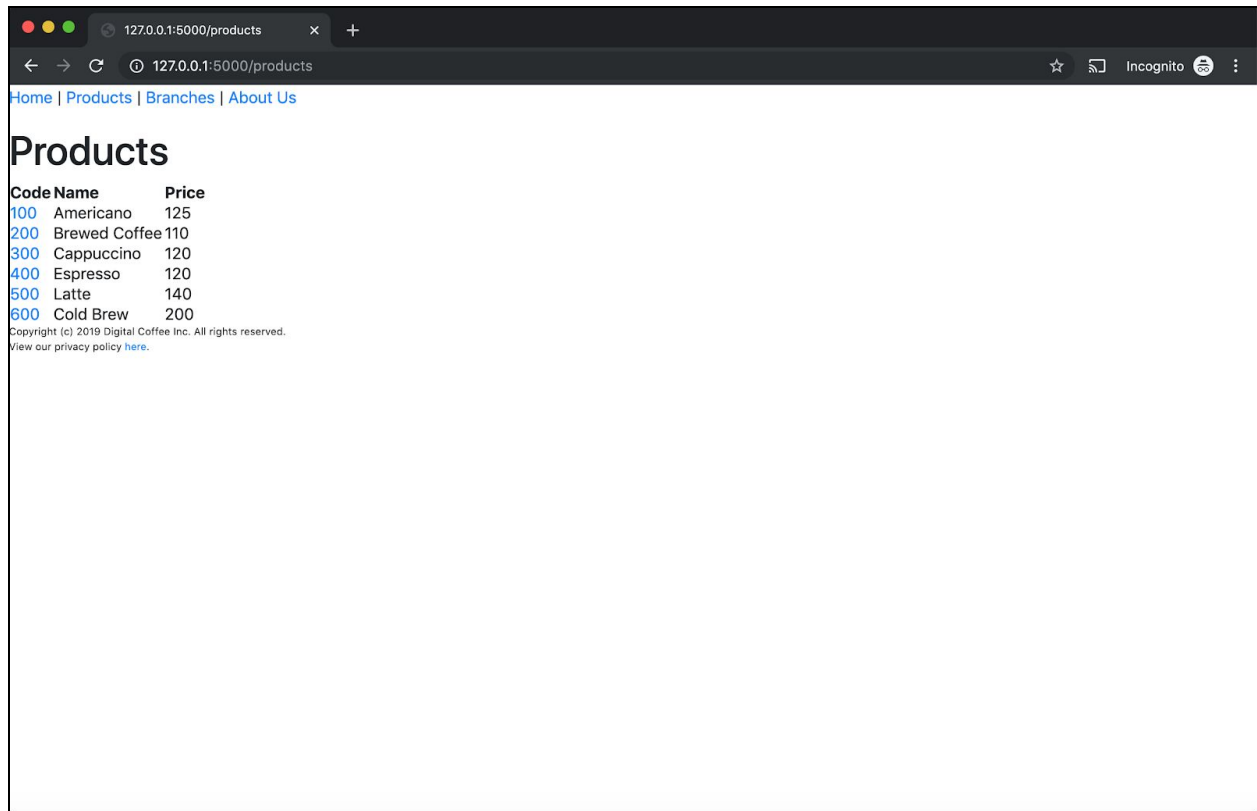
```
def get_product(code):
    return products[code]

def get_products():
    product_list = []

    for i, v in products.items():
        product = v
        product.setdefault("code", i)
        product_list.append(product)

    return product_list
```

Save your changes. Reload the **Products** page. You should see something similar to the following:



Take-Home Quiz #2

Total Points: 20

Due Date: Tuesday, October 29, 6:00PM

This will be a **Group** quiz.

Commit your Flask application folder into your Group GitHub repository. Once done, notify me via email at jbilagan@ateneo.edu.

1. Add more products in the database:

Code	Name	Price
1000	Tiramisu	150
1100	Red Velvet	130
1200	Mango Cream Pie	200

2. Add branches in our mock database as a dictionary. Place this under the **products** dictionary definition.

```
branches = {
    1: {"name": "Katipunan"},
    2: {"name": "Tomas Morato"},
    3: {"name": "Eastwood"},
    4: {"name": "Tiendesitas"},
    5: {"name": "Arcovia"},
}
```

Create a two new helper functions:

- **get_branch(code)**
- **Get_branches**

The code is provided here to make things easier for you:

```
def get_branch(code):
    return branches[code]

def get_branches():
    branch_list = []

    for i,v in branches.items():
        branch = v
        branch.setdefault("code", i)
        branch_list.append(branch)

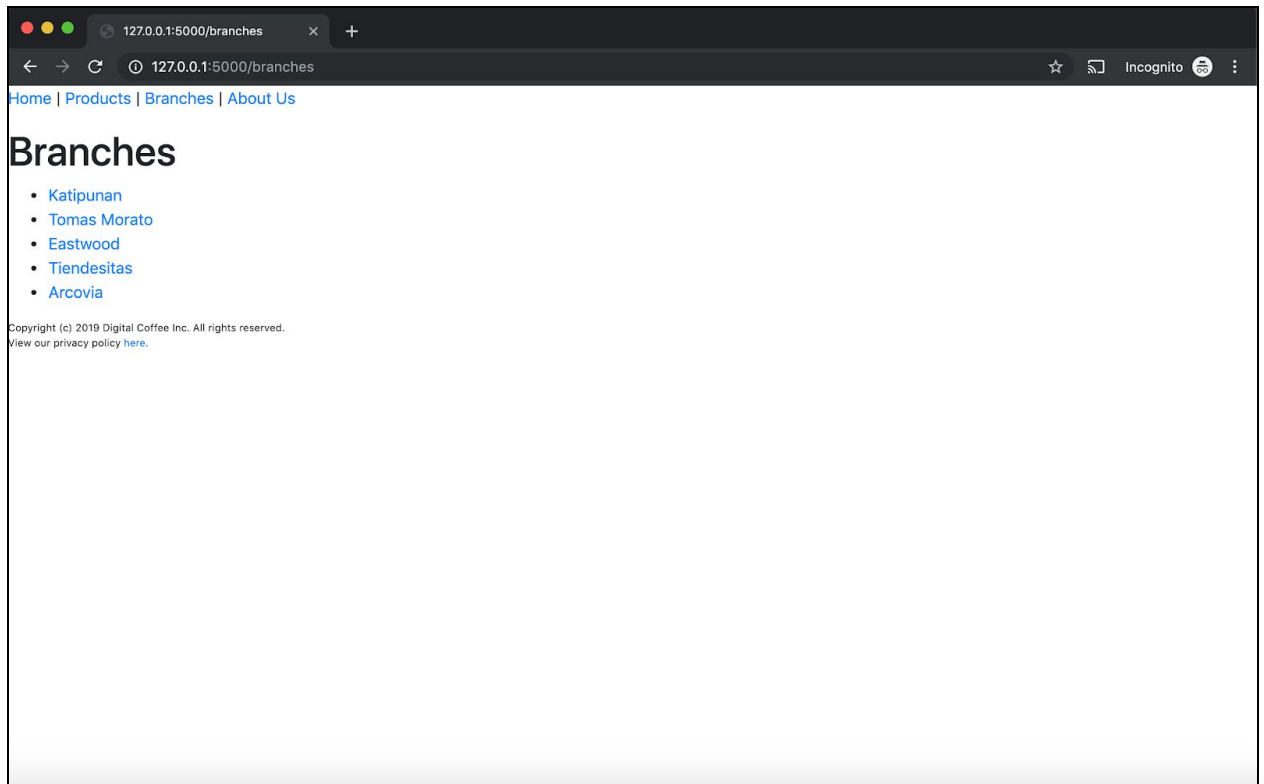
    return branch_list
```

Display these branches in a list using the **branches.html** template file. Unlike Products, use an unordered list ().

Of course, you will also have to modify your **app.py** code for this to work.

Do not display the branch code. Instead, make the Branch Name clickable.

Your resulting screen should look like the following:



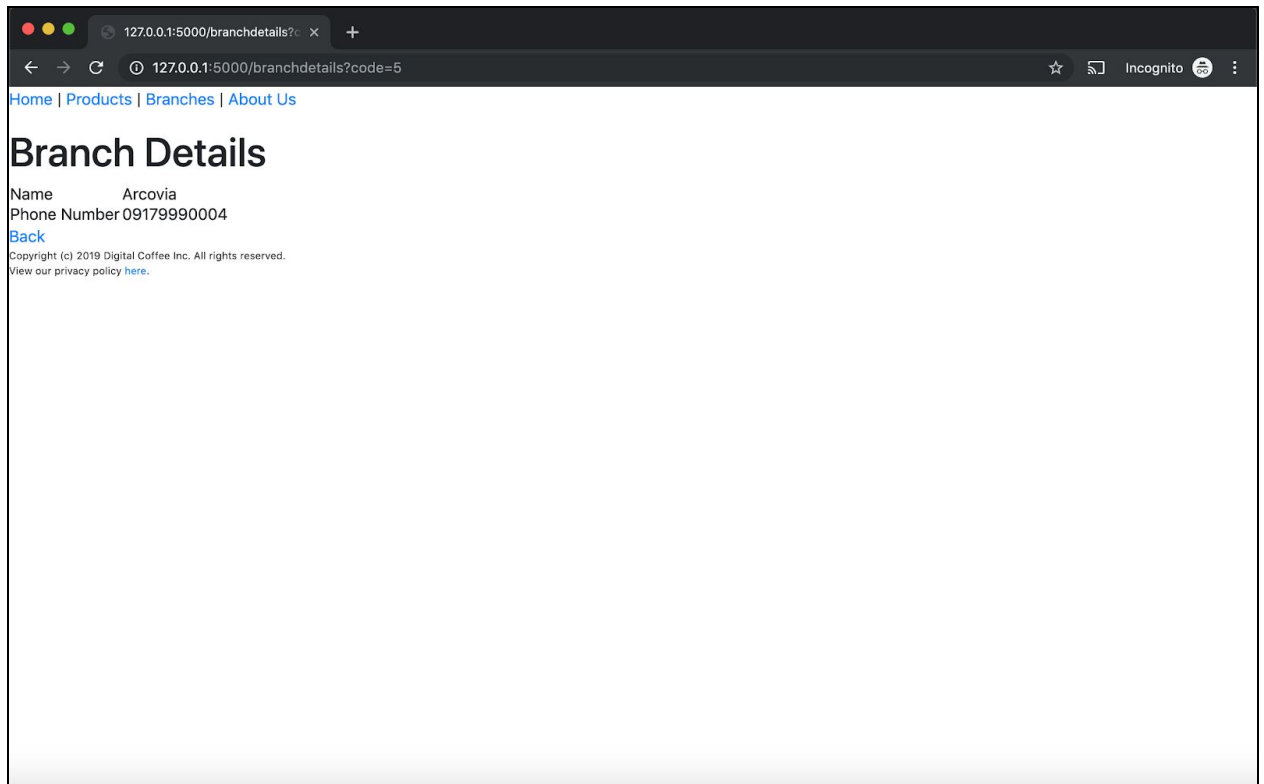
3. Create the template file **branchdetails.html**. Display the Branch Name and Phone Number. It is all up to you how you will display these.

For this to work, you will need to add **phonenumber** as an attribute key in the **branches** mock database.

```
branches = {
  1: {"name": "Katipunan", "phonenumber": "09179990000"},
  2: {"name": "Tomas Morato", "phonenumber": "09179990001"},
  3: {"name": "Eastwood", "phonenumber": "09179990002"},
  4: {"name": "Tiendesitas", "phonenumber": "09179990003"},
  5: {"name": "Arcovia", "phonenumber": "09179990004"},
}
```

If you do copy the template from **productdetails.html**, MAKE SURE you change the target of the **Back** link accordingly.

The resulting Branch Details page should look something like this:



Note to UX Practitioners in class:

Feel free to alter the layout of the web app and to use more modern ways of using html tags as you see fit. The material was designed in such a way that we cover the major technical portions of the Flask web framework as quickly as possible, so UX-related concerns were not given the same importance.