

HTML:

- HTML stands for HyperText Markup Language. It is a formatting system used by browsers to render the webpages we see on the internet. Using HTML, you can create a Web page with text, graphics, sound, and video
- It creates the structure or the skeleton of a webpage.
- HTML code is made up of elements called **tags** that denote the structure of a webpage. A tag is a keyword enclosed by angle brackets. There are opening and closing tags for many but not all tags. The affected text is between the two tags. The opening and closing tags use the same command except the closing tag contains an additional forward slash /.
- Whenever you have HTML tags within other HTML tags, you must close the nearest tag first.
- A comment in HTML is denoted by `<!-- -->`.
- A basic HTML web page looks as follows:

```

<!DOCTYPE html>
<html>
  <head>
    <title>My Website</title>
  </head>
  <body>
    <p>This is a paragraph</p>
  </body>
</html>

```

Here is a basic description of the above tags:

- **<!DOCTYPE html>**: This is here for legacy reasons. It mostly does nothing but when omitted some old browsers use a rendering mode that is incompatible with some specifications. Although it appears useless, it is necessary for any HTML file you produce.
- **<html>**: Tells the browser this is an HTML document. It's the container for all the other HTML elements, except the `<!DOCTYPE>` tag.
- **<head>**: A container for all the head elements (i.e. scripts, styles, meta information, etc).
- **<title>**: The name of the website. This will be shown in the "tab" on your browser
- **<body>**: The meat and potatoes of the website. This is where you will put all the "visual" elements.
- **<p>**: A paragraph tag that indicates that you are writing a paragraph.
- Below is a list of the tags and other useful stuff in alphabetical order.
 1. **Anchor:**
 - a. Denoted with the `<a>` tag.
 - b. The `<a>` tag defines a hyperlink, which is used to link from one page to another.
 - c. The most important attribute of the `<a>` element is the href attribute, which indicates the link's destination.
 - d. General Syntax: `...`
 - e. E.g. `this is a link`

2. **Bold:**
 - a. Denoted with the `` tag.
 - b. General syntax: ` ... `
 - c. E.g. `<p>This is normal text. This is bold text.</p>`
3. **Body:**
 - a. Denoted with the `<body>` tag.
 - b. The `<body>` tag defines the document's body.
 - c. The `<body>` element contains all the contents of an HTML document, such as text, hyperlinks, images, tables, lists, etc.
 - d. General syntax:


```
<body>
...
</body>
```
 - e. E.g.


```
<body>
The content of the document.
</body>
```
4. **Comments:**
 - a. Denoted by `<!-- -->`
 - b. E.g. `<!--This is a comment. Comments are not displayed in the browser-->`
5. **Doctype:**
 - a. Denoted by `<!DOCTYPE html>`
 - b. The `<!DOCTYPE>` declaration must be the very first thing in your HTML document, before the `<html>` tag.
 - c. This is here for legacy reasons. It mostly does nothing but when omitted some old browsers use a rendering mode that is incompatible with some specifications. Although it appears useless, it is necessary for any HTML file you produce.
6. **Head:**
 - a. Denoted with the `<head>` tag.
 - b. The `<head>` element is a container for all the head elements.
 - c. The `<head>` element can include a title for the document, scripts, styles, meta information, and more.
 - d. The following elements can go inside the `<head>` element `<title>`, `<style>`, `<base>`, `<link>`, `<meta>`, `<script>`, `<noscript>`.
 - e. General syntax:


```
<head>
...
</head>
```
 - f. E.g.


```
<head>
<title>Title of the document</title>
</head>
```
7. **Headings:**
 - a. Denoted with the `<h1>` to `<h6>` tags.
 - b. `<h1>` defines the most important heading. `<h6>` defines the least important heading.

- c. It is generally used for titles. Search engines use the headings to index the structure and content of your web pages.
- d. General syntax: `<h1> ... </h1>` where i is in 1, 2, 3, 4, 5, 6.
- e. E.g.

```
<h1>Heading 1</h1>
<h2>Heading 2</h2>
<h3>Heading 3</h3>
<h4>Heading 4</h4>
<h5>Heading 5</h5>
<h6>Heading 6</h6>
```

8. HTML:

- a. Denoted with the `<html>` tag.
- b. The `<html>` tag tells the browser that this is an HTML document.
- c. The `<html>` tag represents the root of an HTML document.
- d. The `<html>` tag is the container for all other HTML elements except for the `<!DOCTYPE>` tag.
- e. General syntax:

```
<html>
...
</html>
```

- f. E.g.

```
<!DOCTYPE HTML>
<html>
<head>
<title>Title of the document</title>
</head>

<body>
The content of the document.
</body>

</html>
```

9. Images:

- a. Denoted with the `` tag.
- b. The `` tag is empty, it contains attributes only, and does not have a closing tag.
- c. General syntax: ``
- d. The src attribute specifies the URL (web address) of the image.
- e. The alt attribute provides an alternate text for an image, if the user for some reason cannot view it (because of slow connection, an error in the src attribute, or if the user uses a screen reader). The value of the alt attribute should describe the image.
- f. E.g. ``

10. Italics:

- a. Denoted with the `<i>` tag.
- b. General syntax: `<i> ... </i>`
- c. E.g. `<p>This is normal text. <i>This is italic text.</i></p>`

11. Lists:

- a. Denoted with the `` tag.
- b. There are 2 types of lists, ordered and unordered:
 - i. Ordered Lists:
 - Denoted with the `` tag.
 - An ordered list can be numerical or alphabetical.
 - We use `` to define the list items.
 - General syntax:


```
<ol>
  <li>...</li>
  <li>...</li>
  ...
</ol>
```
 - E.g.


```
<ol>
  <li>Coffee</li>
  <li>Tea</li>
  <li>Milk</li>
</ol>
```
 - ii. Unordered Lists:
 - Denoted with the tag ``.
 - An unordered list will be displayed with bullets.
 - We use `` to define the list items.
 - General syntax:


```
<ul>
  <li>...</li>
  <li>...</li>
  ...
</ul>
```
 - E.g.


```
<ul>
  <li>Coffee</li>
  <li>Tea</li>
  <li>Milk</li>
</ul>
```

12. Paragraph:

- a. Denoted with the `<p>` tag.
- b. A paragraph tag that indicates that you are writing a paragraph.
- c. General syntax: `<p> ... </p>`
- d. E.g. `<p>This is some text in a paragraph.</p>`

13. Title:

- a. Denoted with the `<title>` tag.
- b. The `<title>` tag is required in all HTML documents and it defines the title of the document.
- c. The `<title>` element defines a title in the browser toolbar, provides a title for the page when it is added to favorites and displays a title for the page in search-engine results.
- d. General syntax: `<title> ... </title>`

- e. E.g. `<title>HTML Reference</title>`

14. Table:

- Denoted with the `<table>` tag.
- The `<tr>` element defines a table row. A `<tr>` element contains one or more `<th>` or `<td>` elements.
- The `<th>` element defines a table header. The text in `<th>` elements are bold and centered by default. The `<th>` element creates header cells which contain header information.
- The `<td>` element defines a table cell. The text in `<td>` elements are regular and left-aligned by default. The `<td>` element creates standard cells which contain data.
- E.g. A simple HTML table with two header cells and two data cells.

```
<table>
<tr>
  <th>Month</th>
  <th>Savings</th>
</tr>
<tr>
  <td>January</td>
  <td>$100</td>
</tr>
</table>
```

- Some tags contain information inside the **leading tag** (the first tag) called **attributes**. Examples of attributes are `` and `<a>`. Some tags don't have any attributes at all and some have a range of varying optional attributes.

JavaScript:

- JavaScript is the Programming Language for the Web.
- JavaScript can update and change both HTML and CSS.
- JavaScript can calculate, manipulate and validate data.

CSS:

- CSS stands for Cascading Style Sheets.
- CSS describes how HTML elements are to be displayed.
I.e. It deals with the layout/design of the webpage.

Flask Introduction:

- Flask is a web application framework written in Python. A web application framework or web framework represents a collection of libraries and modules that enables a web application developer to write applications without having to bother about low-level details such as protocols, thread management etc.
- A virtual environment is a tool that helps to keep dependencies required by different projects separate by creating isolated python virtual environments for them.

Flask Application:

- Here is a basic program with Flask:

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

```
@app.route("/")
def hello_world():
```

```
return 'Hello World'
```

```
if __name__ == '__main__':
    app.run()
```

The flask constructor takes the name of the current module (`__name__`) as argument. The `route()` function of the Flask class is a decorator, which tells the application which URL should call the associated function. The general syntax for the `route()` function is: `app.route(rule, options)`. The rule parameter represents URL binding with the function. The options is a list of parameters to be forwarded to the underlying Rule object. In the above example, `/` URL is bound with the `hello_world()` function. Hence, when the home page of the web server is opened in a browser, the output of this function will be rendered.

Finally the `run()` method of Flask class runs the application on the local development server. The general syntax for the `run()` function is `app.run(host, port, debug, options)` but all parameters are optional. The host is the hostname to listen on. The default for host is 127.0.0.1 (localhost). Set the host to '0.0.0.0' to have the server available externally. The default for port is 5000. The default for debug is false. If it is set to true, it provides debug information. The options are to be forwarded to the underlying Werkzeug server.

- A Flask application is started by calling the `run()` method. However, while the application is under development, it should be restarted manually for each change in the code. To avoid this inconvenience, enable debug support. The server will then reload itself if the code changes. It will also provide a useful debugger to track the errors if any, in the application.
- The debug mode is enabled by setting the debug property of the application object to true before running or passing the debug parameter to the `run()` method.

E.g.

```
app.debug = True
```

```
app.run()
```

```
app.run(debug = True)
```

- E.g. Suppose I am running this piece of code.

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

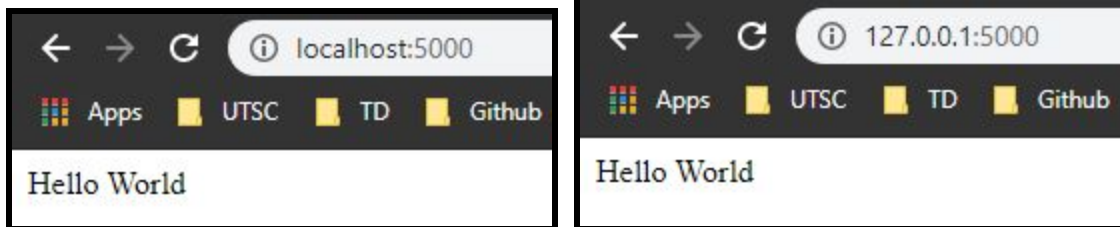
@app.route('/')
def hello_world():
    return 'Hello World'

if __name__ == "__main__":
    app.run()
```

If I run it on terminal, I'll see this

```
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ date
Tue Feb  4 12:59:52 STD 2020
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ python3 Flask_Test_1.py
* Serving Flask app "Flask_Test_1" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

If I type in `http://127.0.0.1:5000/` or `http://localhost:5000/`, I'll see this:



If I want to make any changes, I have to rerun my program.
Suppose I want to run this code:

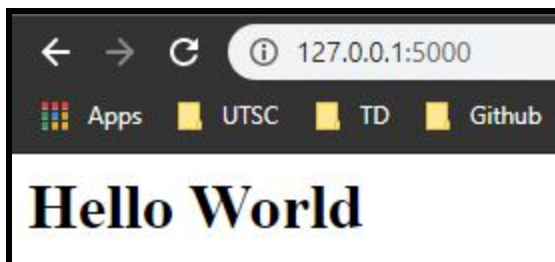
```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return '<h1> Hello World </h1>'

if (__name__ == "__main__"):
    app.run()
```

I need to rerun my program, as shown here:

```
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ date
Tue Feb  4 13:03:18 STD 2020
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ python3 Flask_Test_1.py
* Serving Flask app "Flask_Test_1" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [04/Feb/2020 13:03:28] "GET / HTTP/1.1" 200 -
```



However, if I put the debug statements in my code like such:


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

@app.route('/')
def hello_world():
    return '<h1> Hello World </h1>'

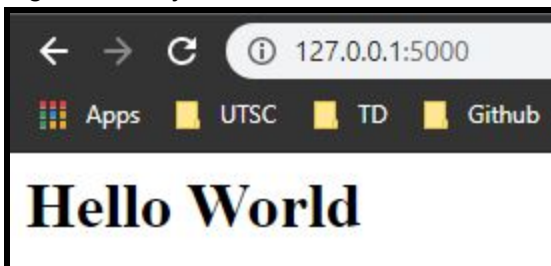
if (__name__ == "__main__"):
    app.debug = True
    app.run()
    app.run(debug = True)
```

and I rerun my program

```
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ date
Tue Feb  4 13:04:36 STD 2020
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ python3 Flask_Test_1.py
* Serving Flask app "Flask_Test_1" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  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: 333-205-022
```

If I make any changes to my code, I don't need to rerun my program.

Right now, my website looks like this



- Suppose I modify my code:

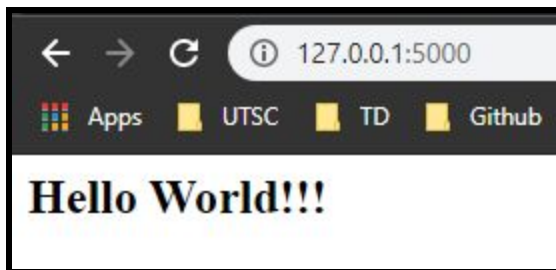
```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return '<h2> Hello World!!! </h2>'

if (__name__ == "__main__"):
    app.debug = True
    app.run()
    app.run(debug = True)
```

Without rerunning my program, my website changed.


```
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ date
Tue Feb  4 13:04:36 STD 2020
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop$ python3 Flask_Test_1.py
* Serving Flask app "Flask_Test_1" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  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: 333-205-022
* Detected change in '/mnt/c/Users/rick/Desktop/Flask_Test_1.py', reloading
* Restarting with stat
* Debugger is active!
* Debugger PIN: 333-205-022
127.0.0.1 - - [04/Feb/2020 13:06:36] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [04/Feb/2020 13:06:37] "GET / HTTP/1.1" 200 -
```



Flask Routing:

- Modern web frameworks use the routing technique to help a user remember application URLs. It is useful to access the desired page directly without having to navigate from the home page. The route() decorator in Flask is used to bind URL to a function.
- Suppose I want to create an about page. My code would look like this:

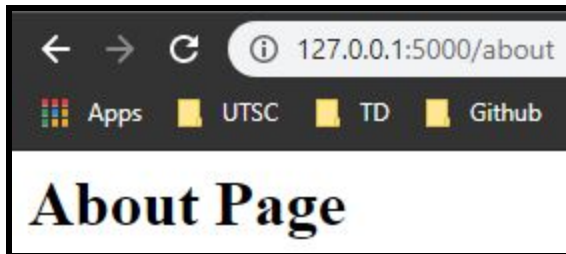
```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return '<h2> Hello World!!! </h2>'

@app.route('/about')
def about():
    return '<h1> About Page </h1>|'

if (__name__ == "__main__"):
    app.debug = True
    app.run()
    app.run(debug = True)
```

- If I type in `http://localhost:5000/about` in my browser, I see this:



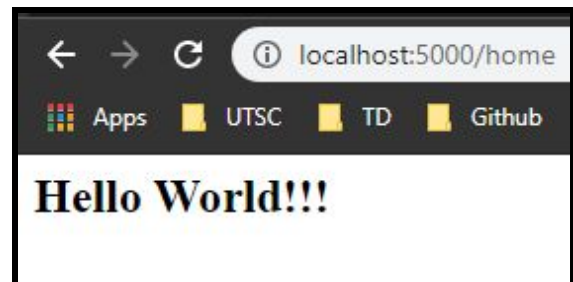
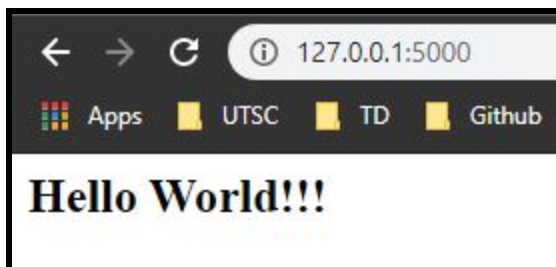
- If you want multiple routes handled by the same function, you add the multiple decorators before the function.
- E.g. Suppose I want to have `/` and `/home` route to the same page. My code looks like this:

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

@app.route('/')
@app.route('/home')
def hello_world():
    return <h2> Hello World!!! </h2>

if (__name__ == "__main__"):
    app.debug = True
    app.run()
    app.run(debug = True)
```

Furthermore, `http://localhost:5000/` and `http://localhost:5000/home` routes to the same page.



Flask Templates:

- While it is possible to return the output of a function bound to a certain URL in the form of HTML, generating HTML content from Python code is cumbersome, especially when variable data and Python language elements like conditionals or loops need to be put. This would require frequent escaping from HTML. Instead of returning hardcoded HTML from the function, a HTML file can be rendered by the `render_template()` function.

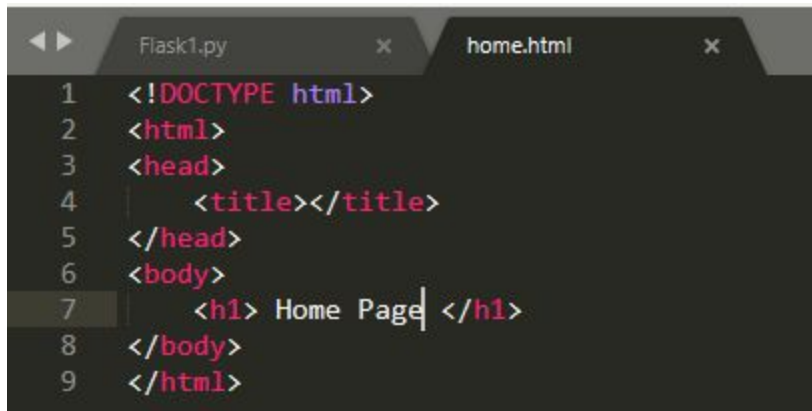
- To use templates, you'll need to create a template directory in the same directory where your python code is.

Flask will try to find the HTML file in the templates folder, in the same folder in which this script is present.

```
Application folder
├── Hello.py
└── templates
    └── hello.html
```

- You'll also need to import `render_template` from Flask.
- E.g.

I have this html file called `home.html`.



```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title></title>
5 </head>
6 <body>
7   <h1> Home Page </h1>
8 </body>
9 </html>
```

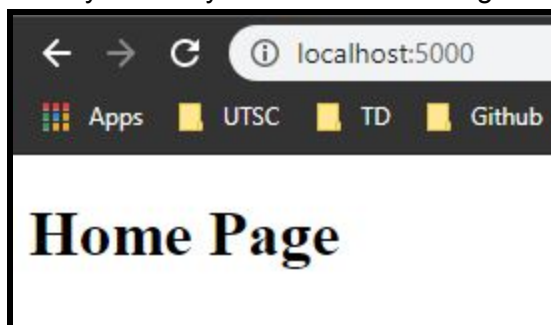
In my python program, I imported `render_template` from Flask and used it like such:

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

@app.route('/')
@app.route('/home')
def hello_world():
    return render_template('home.html')

if (__name__ == "__main__"):
    app.debug = True
    app.run()
    app.run(debug = True)
```

The layout of my website didn't change.



Flask Variables:

- It is possible to build a URL dynamically, by adding variable parts to the rule parameter. This variable part is marked as <variable-name>. It is passed as a keyword argument to the function with which the rule is associated.
- In addition to the default string variable part, rules can be constructed using the following int, float or path.
- The term 'web templating system' refers to designing an HTML script in which the variable data can be inserted dynamically. A web template system comprises of a template engine, some kind of data source and a template processor.
- Flask uses jinja2 template engine. A template language is simply HTML with variables and other programming constructs like conditional statement and for loops etc. This allows you to do some logic in the template itself. The template language is then rendered into plain HTML before being sent to the client. A web template contains HTML syntax interspersed placeholders for variables and expressions which are replaced values when the template is rendered.
- The jinja2 template engine uses the following delimiters for escaping from HTML.
 - {% ... %} for If statements and for loops
 - {{ ... }} for Expressions to print to the template output
 - {# ... #} for Comments not included in the template output
 - # ... ## for Line Statements
- E.g.

```

1  | from flask import Flask, render_template
2  | app = Flask(__name__)
3  |
4  | '''
5  | This program shows how to pass information to an html file
6  | '''
7  |
8  | @app.route('/')
9  | def hello_world():
10 |     return render_template('home.html')
11 |
12 | @app.route('/<name>')
13 | def name(name):
14 |     return render_template('hello.html', name=name)
15 |
16 | if (__name__ == "__main__"):
17 |     app.debug = True
18 |     app.run()
19 |     app.run(debug = True)

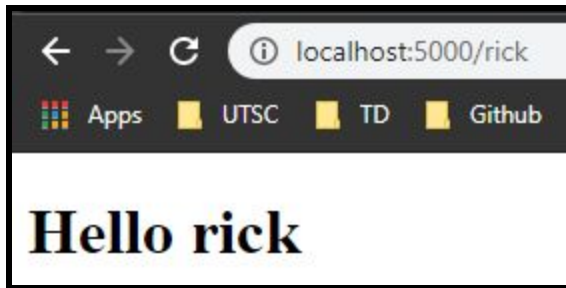
```

```

|<!DOCTYPE html>
|<html>
|<head>
|    <title></title>
|</head>
|<body>
|    <h1> Hello {{name}} </h1>
|</body>
|</html>

```

The webpage would look like this:



Flask If Statements:

- In Flask, if-else and endif are enclosed in delimiter {%..%}.
- You need to end an if-else statement with endif.
- E.g. Suppose we have these 2 pieces of code.

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

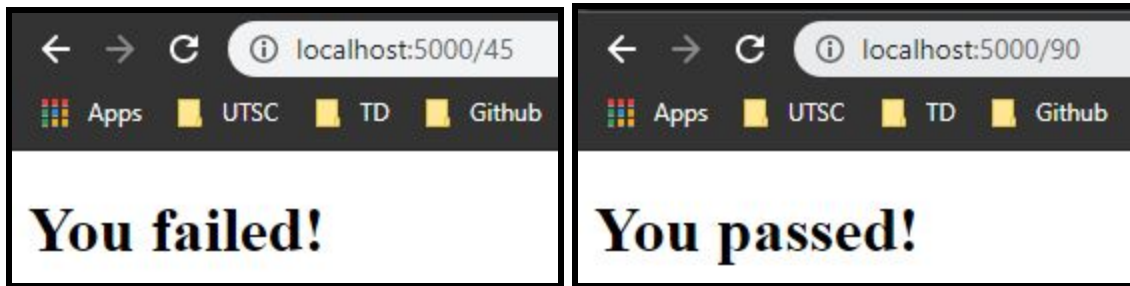
@app.route('/')
@app.route('/home')
def hello_world():
    return render_template('home.html')

@app.route('/<int:mark>')
def mark(mark):
    return render_template('mark.html', mark=mark)

if (__name__ == "__main__"):
    app.debug = True
    app.run()
    app.run(debug = True)
```

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title></title>
5 </head>
6 <body>
7     {% if mark >= 50%}
8         <h1> You passed! </h1>
9     {%else%}
10        <h1> You failed! </h1>
11    {%endif%}
12 </body>
13 </html>
```


The webpage looks like this:



Flask For Loops:

- For loops are enclosed in `{%...%}`.
- Expressions are enclosed in `{{...}}`.
- E.g. Suppose I have these 2 pieces of code:

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

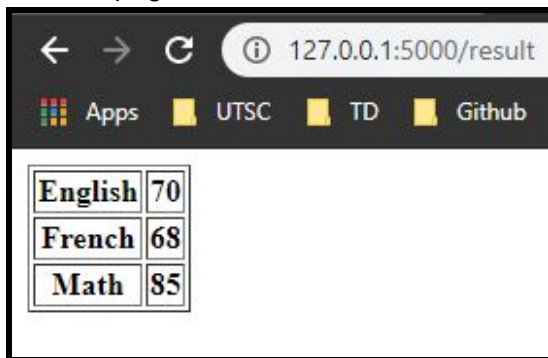
@app.route('/')
@app.route('/home')
def hello_world():
    return render_template('home.html')

@app.route('/result')
def result():
    result_dict = {"Math": 85, "English": 70, "French": 68}
    return render_template('result.html', result = result_dict)

if (__name__ == "__main__"):
    app.debug = True
    app.run()
    app.run(debug = True)
```

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title></title>
5 </head>
6 <body>
7     <table border = 1>
8         {% for key, value in result.items() %}
9         <tr>
10             <th> {{key}} </th>
11             <th> {{value}} </th>
12         </tr>
13         {% endfor %}
14     </table>
15 </body>
16 </html>
```

The webpage looks like this:



Template Inheritance:

- Template inheritance is when a child template can inherit or extend a base template. This will allow you to define your template and a clear and concise way and avoid complexity.
- Imagine, you have an application with multiple pages and each page has the same header. If you want to change something in the header you would need to go through all the templates and change the header for each one which can be a tedious task. Thanks to template inheritance, you only need to create a base template that contains the common header code once and then make all the other templates extend the base template.
- You can use the `{% extends %}` and `{% block %}` tags to work with template inheritance. The `{% extends %}` tag is used to specify the parent template that you want to extend from your current template and the `{% block %}` tag is used to define and override blocks in the base and child templates. You need to end a `{% block %}` tag with the `{% endblock %}` tag. You don't need to put the name of the block in the endblock tag, but it's good practice to, especially if you have multiple blocks.
- E.g. Suppose I have these code snippets:

```

Template_Inheritance.py x BaseTemplate.html x home.html x about.html x
1  from flask import Flask, render_template
2  app = Flask(__name__)
3
4  '''
5  This program shows how to do template inheritance.
6  Template inheritance is when a child template can inherit or extend a base template.
7  This will allow you to define your template and a clear and concise way and avoid complexity.
8  '''
9
10 @app.route('/')
11 def hello_world():
12     return render_template('home.html')
13
14 @app.route('/about')
15 def about():
16     return render_template('about.html')
17
18 if (__name__ == "__main__"):
19     app.debug = True
20     app.run()
21     app.run(debug = True)

```



```

Template_Inheritance.py x home.html x about.html x BaseTemplate.html x
1 <!-- The extends tag is used to specify the parent
2     | template that you want to extend from your current template.-->
3 {% extends "BaseTemplate.html" %}
4
5 <!-- The block tag is used to override blocks child templates.-->
6 {% block content %}
7     <body>
8         <h1> Home Page </h1>
9     </body>
10 <!-- You don't need to put the name of the block in the endblock tag,
11     | but it's good practice to, especially if you have multiple blocks.-->
12 {% endblock content%}

```

```

Template_Inheritance.py x home.html x about.html x BaseTemplate.html x
1 <!-- The extends tag is used to specify the parent
2     | template that you want to extend from your current template.-->
3 {% extends "BaseTemplate.html" %}
4
5 <!-- The block tag is used to override blocks child templates.-->
6 {% block content %}
7     <body>
8         <h1> About Page </h1>
9     </body>
10 <!-- You don't need to put the name of the block in the endblock tag,
11     | but it's good practice to, especially if you have multiple blocks.-->
12 {% endblock content%}

```

```

Template_Inheritance.py x home.html x about.html x BaseTemplate.html x
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title> Title </title>
5     <h1> This will be in both home.html and about.html. </h1>
6 </head>
7 <body>
8     <!-- The block tag is used to define blocks in the base template.
9         | The block and endblock tags must be positioned this way.-->
10     {% block content %}{% endblock %}
11 </body>
12 </html>

```

The webpages look like this:

