Temasek Junior College

2024 JC2 H2 Computing

Web Applications 8 – Jinja2 Operations

Section	4	Computer Networks
Unit	4.2	Web Applications
Objectives	4.2.3	Use HTML, CSS (for clients) and Python (for the server) to create a web application that is able to: - accept user input (text and image file uploads) - process the input on the local server - store and retrieve data using an SQL database - display the output (as formatted text/ images/table)
	4.2.4	Test a web application on a local server.

1 Jinja2 Filters

Jinja2 filtering is a powerful feature of the Jinja 2 template engine that enhances the flexibility and control over how data is presented in templates. **Jinja2 filters** are used to transform variables in the template, allowing developers to modify or format the output of a variable directly within the template, providing a way to clean up or adjust the data being displayed.

Jinja2 comes with a variety of built-in filters. Some common ones include:

- **`capitalize`**: capitalizes the first character of the string.
- **`lower`**: converts the string to lowercase.
- **`upper`**: converts the string to uppercase.
- **`default`**: sets a default value if the variable is undefined.
- **`length`**: returns the length of an object.
- **`replace`**: replaces parts of the string with another string.
- **`safe`**: marks the variable as safe for rendering (e.g., HTML code).

To apply a filter, we write a Jinja 2 expression with its Jinja2 variable, followed by the pipe character **|** , then the name of the filter i.e. of the form **`{{ expression|filter }}`**.

In particular, we shall study in detail the **`length`** filter and the **`safe`** filter.

1.1 Jinja2 `length` Filter

Since the **`len()`** function is not available in Jinja2 expressions, you might wonder how it is possible to output the length of a string or a list from a template.

One solution would be to perform the calculation in Python and pass the value over to the template as a separate Jinja2 variable. However, there is no real need to write additional Python code for simple length checks. Jinja2 provides a **`length`** filter that gives the same result as **`len()`**.

Recall that to apply a filter, we write a Jinja 2 expression with its Jinja2 variable, followed by the pipe character **|** , then the name of the filter.

Exercise 1
Create the HTML template **length.html**. Then create and run the server code **name_with_length_filter.py**.

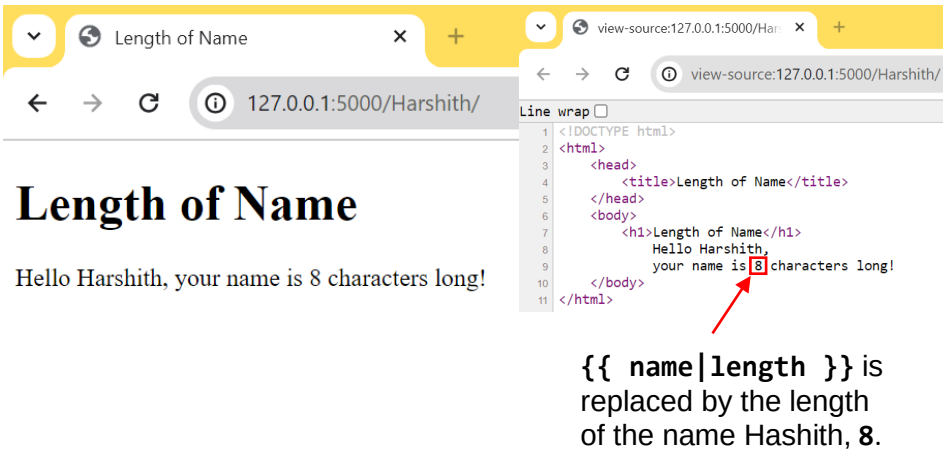
length.html

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Length of Name</title>
5    </head>
6    <body>
7      <h1>Length of Name</h1>
8      Hello {{ name }},
9      your name is {{ name|length }} characters long!
10   </body>
11 </html>
```

name_with_length_filter.py

```
1  import flask
2  from flask import render_template
3
4  app = flask.Flask(__name__)
5
6  @app.route('/<name>/')
7  def length_of_name(name):
8    return render_template('length.html', name=name)
9
10 if __name__ == '__main__':
11   app.run()
```

Try visiting several URLs using different names e.g. **http://127.0.0.1:5000/Harshith/** and examine how line **9** of the template uses the **length** filter to determine the number of characters in **name**.



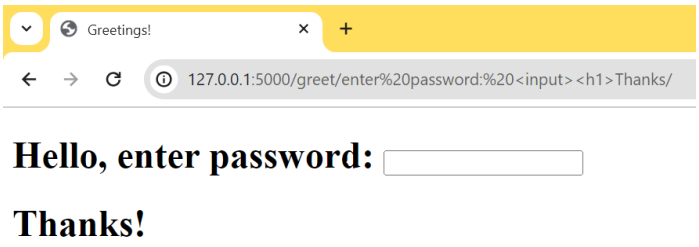
1.2 Jinja2 `safe` Filter

Recall that constructing HTML documents by concatenating Python strings incurs the risk of HTML injection.

`response_without_templates.py` from **Exercise 7** of **Web Applications 7**

```
1  import flask
2
3  app = flask.Flask(__name__)
4
5  @app.route('/')
6  def home():
7    html = '<!DOCTYPE html>\n<html>'
8    html += '<head><title>Home Page</title></head>'
9    html += '<body><h1>Welcome to my home page!</h1>'
10   html += '<p>My favourite web site is '
11   html += '<a href="http://example.com/">'
12   html += 'example.com</a>.</p>'
13   html += '<p>You can greet Alex '
14   html += '<a href="/greet/Alex/">here</a>.</p>'
15   html += '</body></html>'
16   return html
17
18 @app.route('/greet/<name>/')
19 def greet(name):
20   html = '<!DOCTYPE html>\n<html>'
21   html += '<head><title>Greetings!</title></head>'
22   html += f'<body><h1>Hello, {name}!</h1>'
23   html += '</body></html>'
24   return html
25
26 if __name__ == '__main__':
27   app.run()
```

If the URL **`http://127.0.0.1:5000/greet/enter%20password:%20<input><h1>Thanks/`** is entered when **`response_without_templates.py`** is running, users will be brought to what appears to be a legitimate form asking for the user's password.



However, in reality, the form does not exist at all as there was no HTML written in the Flask program to create it. The form was simply injected into the page via HTML code written in the URL's path.

The string `enter%20password:%20<input><h1>Thanks` was taken as the value of `name` in Line **18** of the Flask application. This was subsequently passed to the **`greet()`** function as a keyword argument in Line **19**. Eventually, the value was used in Line **22** and processed as a HTML script (**`%20`** is interpreted as a whitespace character, where **`%`** signifies that the next two characters represent a hexadecimal code, and **`20`** is that of a whitespace character in the standard ASCII character set).

Now, run the Flask application **`response_with_templates.py`**, which makes use of HTML templates instead of Python string concatenation.

`response_with_templates.py` from **Exercise 10** of **Web Applications 7**

```
1  import flask
2  from flask import render_template
3
4  app = flask.Flask(__name__)
5
6  @app.route('/')
7  def home():
8    return render_template('home.html')
9
10 @app.route('/greet/<name>/')
11 def greet(name):
12   return render_template('greet.html', visitor=name)
13
14 if __name__ == '__main__':
15   app.run()
```

The HTML templates can be obtained from **Exercises 8** and **9** of **Web Applications 7** as follows:

`templates/home.html` from **Exercise 8** of **Web Applications 7**

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Home Page</title>
5    </head>
6    <body>
7      <h1>Welcome to my home page!</h1>
8      <p>My favourite web site is
9        <a href="http://example.com/">example.com</a>.
10     </p>
11     <p>You can greet Alex
12       <a href="{{ url_for('greet', name='Alex') }}">here</a>.
13     </p>
14   </body>
15 </html>
```

templates/greet.html from **Exercise 9** of **Web Applications 7**

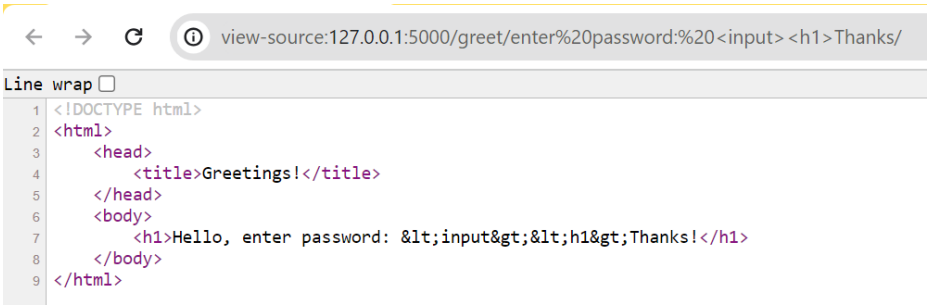
```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Greetings!</title>
5    </head>
6    <body>
7      <h1>Hello, {{ visitor }}!</h1>
8    </body>
9  </html>
```

Now visit **http://127.0.0.1:5000/greet/enter%20password:%20<input><h1>Thanks/** again.

The webpage that is generated now is as follows:

The HTML injection did not occur and **enter%20password:%20<input><h1>Thanks** is appropriately interpreted and displayed as plain text instead of being treated as HTML.

Viewing the source using **Ctrl** + **U**, we observe the following:



The **<** and **>** symbols are no longer interpreted as the opening and closing of HMTL tags. The Jinja2 template engine has automatically escaped this interpretation by using **&lt** for **<** and **&gt** for **>**, allowing them to be interpreted as the actual character representation i.e. lesser than (**<**) and greater than (**>**). At the same time **%20** continues to be correctly interpreted as a whitespace character This allows the placeholder Jinja2 expression **{{ visitor }}** to be used without concerns of HTML injection, as the appropriate value will be passed to the template to replace it during rendering.

What happens then if in the hypothetical situation, it is intentional for the string **enter%20password:%20<input><h1>Thanks** to be interpreted as HTML?

If we are sure that the string should be treated as HTML, the Jinja2 template engine can be "notified" with the use of the **safe** filter. This will allow HTML script to be passed to the template as the value to replace **{{ visitor }}**.

Recall that to apply a filter, we write a Jinja 2 expression with its Jinja2 variable, followed by the pipe character **|** , then the name of the filter.

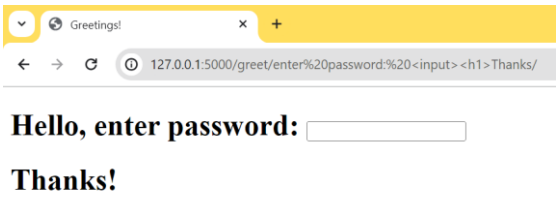
Exercise 2
Modify the **greet.html** template using Notepad++ and save it as **templates/greet_with_filter.html**

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Greetings!</title>
5    </head>
6    <body>
7      <h1>Hello, {{ visitor|safe }}!</h1>
8    </body>
9  </html>
```

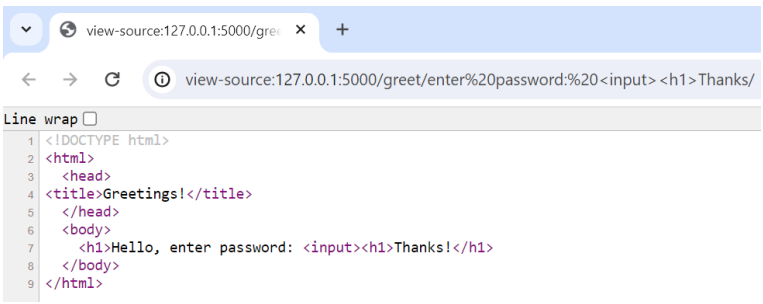
Exercise 3
Modify **response_with_templates.py** as follows and save it as **response_filter.py**. Try running the code on both IDLE and Jupyter Notebook.

```
1  import flask
2  from flask import render_template
3
4  app = flask.Flask(__name__)
5
6  @app.route('/')
7  def home():
8    return render_template('home.html')
9
10 @app.route('/greet/<name>/')
11 def greet(name):
12   return render_template('greet_with_filter.html', visitor=name)
13
14 if __name__ == '__main__':
15   app.run()
```

Now go to **http://127.0.0.1:5000/greet/enter%20password:%20<input><h1>Thanks/**. The webpage now displays as follows:



As seen, use of the **safe** filter allows the string **enter%20password:%20<input><h1>Thanks** to be interpreted as HTML appropriately in Line **7** of the final HTML script.



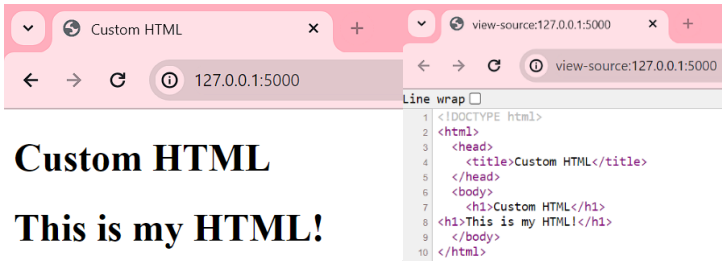
Exercise 4
Create the following template using Notepad++ and save it as **templates/custom.html**

```
1   <!DOCTYPE html>
2   <html>
3     <head>
4       <title>Custom HTML</title>
5     </head>
6     <body>
7       <h1>Custom HTML</h1>
8       {{ my_html|safe }}
9     </body>
10  </html>
```

Exercise 5
Try running the following code on both IDLE (as **my_html.py**) and Jupyter Notebook.

```
1   import flask
2   from flask import render_template
3
4   app = flask.Flask(__name__)
5
6   @app.route('/')
7   def home():
8     return render_template('custom.html', \
9                            my_html='<h1>This is my HTML!</h1>')
10
11  if __name__ == '__main__':
12    app.run()
```

Visit **http://127.0.0.1:5000/**. The string passed to the template as **my_html** i.e. **'<h1>This is my HTML!</h1>'** is interpreted as HTML when the **safe** filter is used.



Exercise 6
Modify the **custom.html** template by removing the **safe** filter and save it as **templates/custom_no_filter.html**

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Custom HTML</title>
5    </head>
6    <body>
7      <h1>Custom HTML</h1>
8      {{ my_html }}
9    </body>
10 </html>
```

Exercise 7
Modify also **my_html.py** by changing the path specified in **render_template()** to **custom_no_filter.html** and saving it as **my_html_no_filter.py**. Try running the code on both IDLE and Jupyter Notebook

```
1  import flask
2  from flask import render_template
3
4  app = flask.Flask(__name__)
5
6  @app.route('/')
7  def home():
8    return render_template('custom_no_filter.html', \
9                           my_html='<h1>This is my HTML!</h1>')
10
11 if __name__ == '__main__':
12   app.run()
```

Visit **http://127.0.0.1:5000/**. Now that the **safe** filter is removed, the symbols **<** and **>** are escaped using **&lt** and **&gt**. The page will now display as follows:

Technical Note

- Restart your Flask program for changes (such as removing the **`safe`** filter) to take effect.
- If your Flask program stops reflecting changes made:

  Method 1
  - Restart your Flask program.
  - Hold down the **`Ctrl`** key while refreshing your browser to bypass the cache and perform a fresh HTTP request.

  Method 2
  - Run the Flask program using another port by replacing **`app.run()`** with say **`app.run(port=5001)`**.
  - There always lies a possibility of the selected port being used by other processes. If this happens, try another port.
  - **Avoid ports below `1024`** as they require elevated access rights (often administrator or root privileges). You will not have such rights when using examination laptops.
  - Some common approaches to selecting ports include but are not limited to:
    - One approach is to increment the port stepwise from **`5000`** to **`5001`**, **`5002`**, **`5003`**…
    - A second approach is to use common development ports in this order: **`5000`**, **`8000`**, **`8080`**, **`3000`**)
    - A third approach is to use higher unprivileged ports within the **`49152`** to **`65535`** range. There's a wider selection available, reducing the chance of encountering occupied ports consecutively, especially if you keep failing to select an available port using the first two approaches.

  Method 3
  - Run the Flask program in debug mode by replacing **`app.run()`** with **`app.run(debug=True)`**. This lets Flask reload itself when it detects any file changes so manual restarts are not needed.
  - Running Flask in debug mode however has several incompatibilities with IDLE and Jupyter Notebook. Hence this method is not suitable for use during examinations as the stipulated development environment is either IDLE or Jupyter Notebook.
  - There is no IDE that is absolutely "fully compatible" with Flask. However, several IDEs do offer excellent support and features that make them well-suited for Flask development. Examples of these IDEs include PyCharm, VSCode and Thonny.
  - If you wish to use debug mode, it is recommended that you:
    - run your Flask program on IDEs such as the aforementioned, or
    - run your Flask program in Command Prompt or PowerShell instead.

2 Jinja2 Statements

In a typical Flask application, majority of data processing and computation should be done using Python code. The results are then passed to the appropriate HTML templates as numbers, strings, lists or other appropriate data objects, where they replace the Jinja2 expressions that are used as placeholders, producing the final HTML script.

Although it is possible to perform some data processing and computation using Jinja2, performing complex logic in a template is not recommended. Nevertheless, sometimes it is useful to perform some simple logic in a template, such as to selectively render parts of a template or to repeat a portion of the template for every item in a list.

To perform these tasks, Jinja2 supports control flow in a template using **Jinja2 statements** made up of commands surrounded by an opening **{%** and a closing **%}**.

Unlike placeholders surrounded by double braces **{{ expression }}** that are usually replaced with actual values passed over from the Flask program, the contents enclosed within **{% statement %}** do not produce any output.

2.1 Jinja2 `if` Statements

Similar to the **`if`** statement in Python, the Jinja2 **`if`** statement is used to selectively include or exclude portions of the template. Excluded portions of a template are simply not rendered.

However, unlike Python statements, Jinja2 statements are not written based on indentation. As such, there needs to be an **`endif`** command to indicate the end of the conditional construct. In addition, the **`if`**, **`elif`** and **`else`** clauses are demarcated by separate `{%` and `%}` blocks. As such colons are no longer needed.

Exercise 8
Create the HTML template **`results.html`**. Then create and run the server code **`results_using_if.py`**.

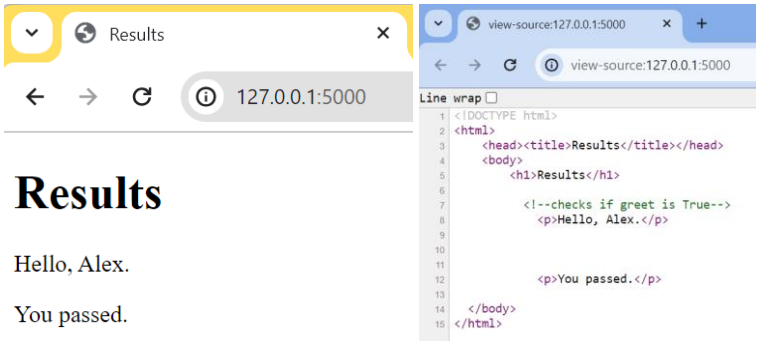
`results.html`

```
1  <!DOCTYPE html>
2  <html>
3    <head><title>Results</title></head>
4    <body>
5      <h1>Results</h1>
6
7      {% if greet %}  <!--checks if greet is True-->
8        <p>Hello, {{ name }}.</p>
9      {% endif %}
10
11     {% if show_score %}  <!--checks if show_score is True-->
12       <p>Your score is {{ score }}%.</p>
13     {% elif score >= 50 %}
14       <p>You passed.</p>
15     {% else %}
16       <p>You failed.</p>
17     {% endif %}
18   </body>
19 </html>
```

results_using_if.py

```
1   import flask
2   from flask import render_template
3
4   app = flask.Flask(__name__)
5
6   @app.route('/')
7   def home():
8     return render_template('results.html', greet=True,
9       name='Alex', show_score=False, score=72)
10
11  if __name__ == '__main__':
12    app.run()
```

Visit **http://127.0.0.1:5000/** to see the template rendered for a score of 72.



Viewing the source using **Ctrl** + **U**, we observe that Lines **9** to **11** of the HTML script are not rendered. These lines correspond to the **elif** and **else** clauses, which do not run since the **if** clause has been satisfied.

Adjust the values assigned to **greet**, **show_score** and **score** on lines **8** and **9** to include the following combinations:

Combination	greet	show_score	score
1	True	False	27
2	True	True	72
3	True	True	27
4	False	True	72
5	False	True	27
6	False	False	72
7	False	False	27

Predict the output of each combination.

Then, restart the server for each combination and visit **http://127.0.0.1:5000/** to see how the template is rendered differently. Remember to use **Ctrl** + **U** to view the source each time to see how the final HTML script changes dynamically.

In each case, verify using the output whether you prediction was correct.

2.2 Jinja2 `for...in...` Statements

Similar to the **`for`** loop in Python, the Jinja2 **`for...in...`** statement is used to repeat the rendering of a portion of the template for every item in a list, tuple, string or dictionary.

However, recall that Jinja2 statements are not written based on indentation. As such, there needs to be an **`endfor`** command to indicate the end of the loop construct.

A typical use of **`for...in...`** is to output the contents of a collection in a table.

Exercise 9
Create the HTML template **`table.html`**. Then create and run the server code **`table_using_for.py`**.

`table.html`

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Table of Results</title>
5    </head>
6    <body>
7      <h1>Table of Results</h1>
8      <table>
9        <tr>
10         <th>Subject Name</th>
11         <th>Score</th>
12       </tr>
13       {% for subject in results %}
14       <tr>                                <!--start a new row-->
15         <td>{{ subject }}</td>            <!--1st cell in row-->
16         <td>{{ results[subject] }}</td>   <!--2nd cell in row-->
17       </tr>                               <!--end the row-->
18       {% endfor %}
19     </table>
20   </body>
21 </html>
```

`table_using_for.py`

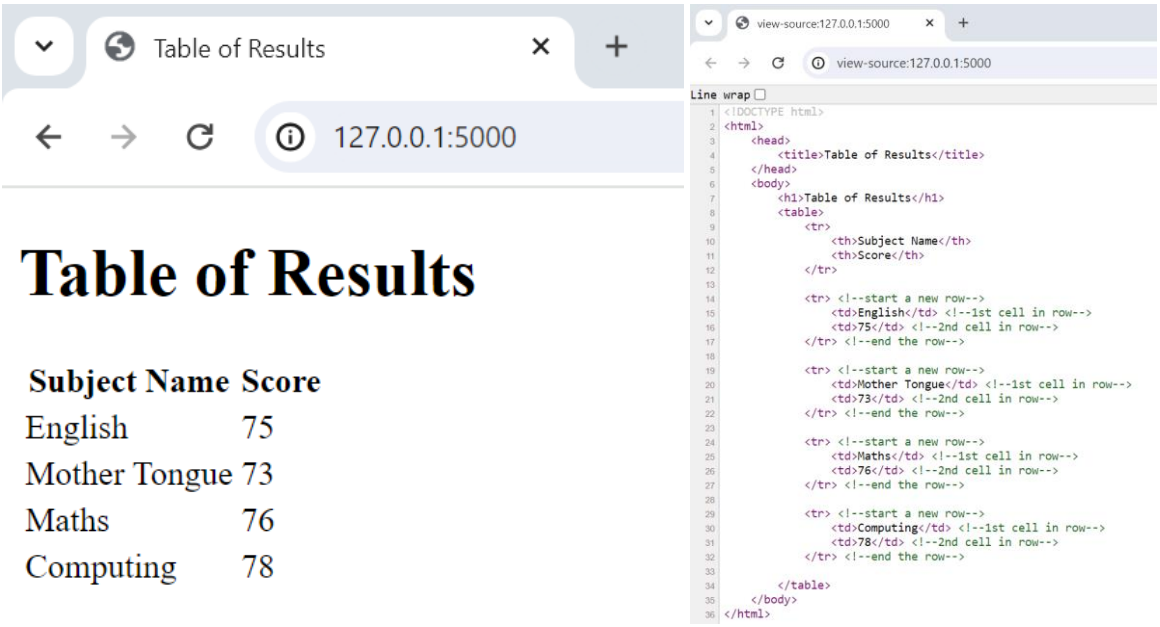
```
1  import flask
2  from flask import render_template
3
4  app = flask.Flask(__name__)
5
6  @app.route('/')
7  def home():
8    results = {'English': 75, \
9               'Mother Tongue': 73, \
10              'Maths': 76, \
11              'Computing': 78}
12
13   return render_template('table.html', results=results)
14
15 if __name__ == '__main__':
16   app.run()
```

In **Exercise 9**, **`table.html`** expects a Jinja2 variable named **`results`** containing a dictionary mapping subject names to scores.

The program **`table_using_for.py`** creates a hardcoded dictionary **`results`** matching this format and passes it to the template for rendering (hence the keyword argument **`results=results`** in Line **`14`** of the Flask program).

Visit **`http://127.0.0.1:5000/`** to see how the hardcoded data in the dictionary is rendered as a HTML table.

Notice that in the final HTML script, Lines **`14 – 17`**, **`19 – 22`**, **`24 – 27`** and **`29 – 32`** have been iteratively created by the **`for...in...`** statement written in the source HTML script in **`table.html`**



In web applications that will be designed in the 9569 H2 Computing curriculum, data would most likely be retrieved from an SQL database instead of hardcoding them. Hence the Flask application would likely contain also sqlite3 code.