Temasek Junior College
2023 JC2 H2 Computing
Web Applications 4 – Flask Basics

Objectives

- Use the **`flask.Flask()`** constructor to create a Flask application.
- Use the **`flask.Flask.route()`** decorator to associate functions with either a static path or a path containing one or more variable sections.
- Use **`flask.Flask.run()`** to start a Flask application.
- Use **`flask.render_template()`** to simplify sending of long HTML responses.
- Use **`flask.redirect()`** to send a **`HTTP 302 (Found)`** status code and have the browser request a different address
- Use the **`flask.request.form`** dictionary to retrieve decoded form data.
- Use the Jinja2 template language and **`flask.url_for()`** to dynamically generate links and URLs in HTML responses.

1 The Flask Framework

A **framework** is typically a module or library with ready-made generic solutions that a programmer can selectively override to customise certain behaviours. This lets the programmer get more done in less time as the framework already provides a working solution that only needs to be appropriately configured or customised to meet the task requirements.

To create dynamic web sites, we can use a web application framework named **Flask**.

Without any customisations, Flask already provides a basic web server that correctly implements HTTP and its many requirements. To create and run this basic web server, we need to create a **`flask.Flask`** object with the module's **`__name__`** as an argument and call the object's **`run()`** method.

Exercise 1
Try running the following code on both IDLE (as **`minimal.py`**) and Jupyter Notebook.

```
import flask

app = flask.Flask(__name__)

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

When this program is run, you should see some start-up messages that indicate the server can be accessed at **`http://127.0.0.1:5000/`**.

However, as the default web server is not configured to recognise any paths yet, you will receive a **`404 (Not Found)`** error when you visit that URL using a web browser.

Nevertheless, you should notice that Flask already provides a complete web server that correctly implements HTTP without additional work from the programmer.

Note that `5000` is just the default port number used by Flask. To use another port number, call the **`run()`** method with a different port argument.

For example, to use port 12345 instead, we would replace line `6` with:

```
app.run(port=12345)
```

To stop the Flask server, press `Ctrl-C` in the IDLE's shell window or the Jupyter Notebook cell.

2 Requests and Routes

The first and most important way to customise the web server that Flask provides is to configure which paths are recognised.

2.1 HTTP Requests

Each HTTP request starts with a request line specifying a **method** and a **path** as well as the version of HTTP being used.

Whether a HTTP request succeeds typically depends on whether the path refers to a web document that is recognised by the server and whether the method used is allowed for that web document.

Consider a HTTP server running on `127.0.0.1` and listening on port `5000`.

When we visit **`http://127.0.0.1:5000/readme.txt`**, the browser sends a HTTP request with a first line that contains the path **`/readme.txt`**, e.g.

```
GET /readme.txt HTTP/1.1
```

While HTTP paths such as **`/readme.txt`** look like file paths used to open and save files on a computer, they are just strings to a web server and **DO NOT** need to refer to real files or folders stored on the computer. Hence there does not need to be a real file named **`readme.txt`** for the web server to provide a valid response.

Another example is how some web servers dynamically generate a HTML error page when we visit a URL that the server does not recognise.

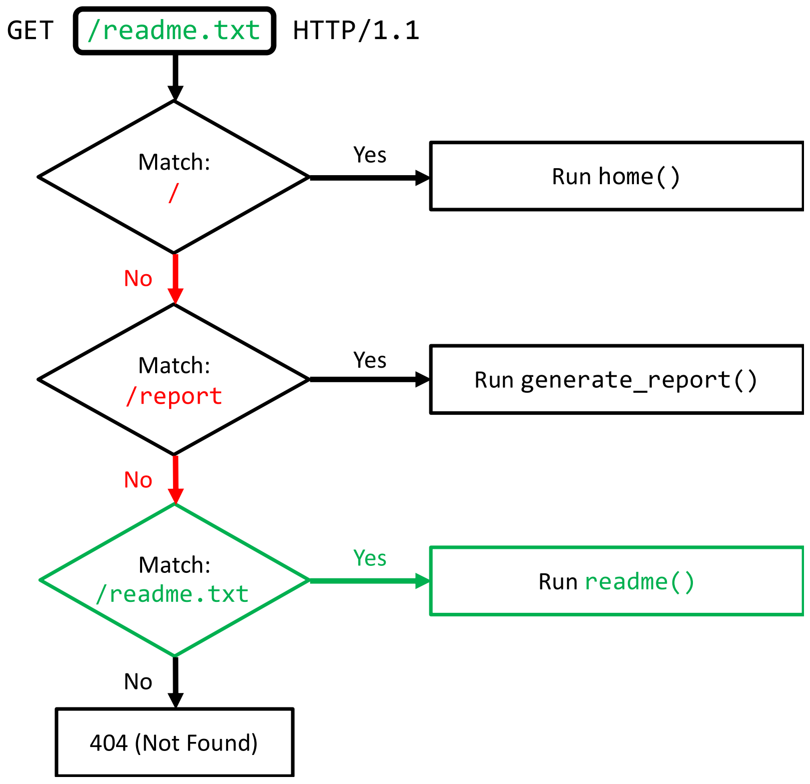
More details on HTTP will be discussed when you study networks.

2.2 Routing

2.2.1 Routing Basics

To handle HTTP requests, we can map specific paths to Python functions. For instance the path `/` can be mapped to a function named **`home()`** and **`/readme.txt`** can be mapped to a function named **`readme()`**.

When a HTTP request is received, Flask examines the received path and looks for a mapping. If a mapping is found, Flask runs the associated Python function to generate a response. When no mapping is found, a **`404 (Not Found)`** status code is produced instead.



This process is called **routing**, which is like how phone operators forward calls to different departments based on each caller's needs.

Each HTTP request is routed to a Python function for processing based on the requested path and method used. Each mapping of a path to a Python function is called a **route**.

To declare a route and associate a path to a Python function, we use a feature of Python called **decorators**, which allow us to alter the behaviour of a function without modifying its source code. This is done by adding the **decorations** immediately before the function's definition.

Each **decoration** starts with an **@** symbol followed by a decorator. For Flask, we typically use decorators that are generated using the **`route`** method of the main Flask object named **`app`**.

Exercise 2
Try running the following code on both IDLE (as **simple_routes.py**) and Jupyter Notebook.

```
1  import flask
2  
3  app = flask.Flask(__name__)
4  
5  @app.route('/')  # decoration
6  def home():
7    return 'Welcome'
8  
9  @app.route('/report')  # decoration
10 def generate_report():
11   return 'Everything is awesome'
12 
13 @app.route('/readme.txt') # decoration
14 def readme():
15   return 'READ ME'
16 
17 if __name__ == '__main__':
18   app.run()
```

Lines 5, 9 and 13 in **Exercise 2** give examples of decorations used in Flask.

Upon running the code in **Exercise 2**, do the following:

1) Go to `http://127.0.0.1:5000/`
   What do you see?

2) Go to `http://127.0.0.1:5000/report`
   What do you see?

3) Go to `http://127.0.0.1:5000/readme.txt`
   What do you see?

Notice that you will see the words `Welcome` when you visit `http://127.0.0.1:5000/`. This is because of the decoration on line 5, which maps the path '/' to the `home()` function, that returns the string `'Welcome'`.

Similarly, visiting `http://127.0.0.1:5000/report` gives the page with the words `Everything is awesome`. This is because the decoration on line 9 maps the path `'/report'` to the `generate_report()` function, that returns the string `'Everything is awesome'`.

Likewise, when you visit `http://127.0.0.1:5000/readme.txt`, you will see the message `READ ME` returned by the `readme()` function. This works because the path `'/readme.txt'` is mapped to the `readme()` by the decoration on line 13.

However, visiting any other URL that starts with `http://127.0.0.1:5000/` e.g. `http://127.0.0.1:5000/nothing` results in a `404 (Not Found)` error as no other paths have been mapped.

2.2.2 Fixed Routes and Trailing Slashes

Exercise 3
Try running the following code on both IDLE (as **`fixed_routes.py`**) and Jupyter Notebook.

```
1  import flask
2
3  app = flask.Flask(__name__)
4
5  @app.route('/')
6  def index():
7    return 'Routed to index()'
8
9  @app.route('/css')
10 def css():
11   return 'Routed to css()'
12
13 @app.route('/no_slash')
14 def no_slash():
15   return 'Routed to no_slash()'
16
17 @app.route('/optional_slash/')
18 def optional_slash():
19   return 'Routed to optional_slash()'
20
21 if __name__ == '__main__':
22   app.run()
```

Upon running the code in **Exercise 3**, do the following:

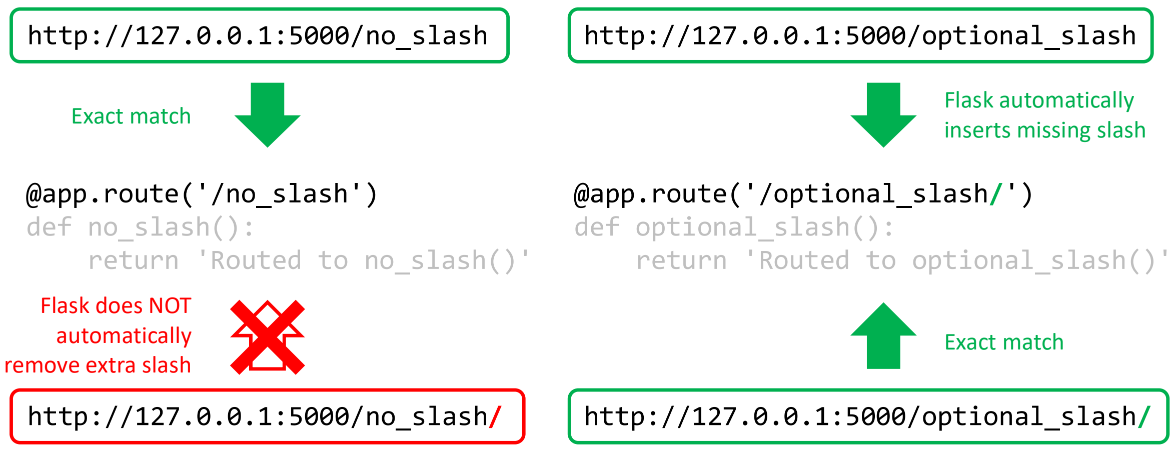
1) Go to **`http://127.0.0.1:5000/`**
You should see a simple page with the message **`Routed to index()`**. This is because the path component of **`http://127.0.0.1:5000/`** is **`'/'`**, which matches the route specified on line **5** for the **`index()`** function. This means that the request is being handled by the **`index()`** function.

2) Go to **`http://127.0.0.1:5000/css`**
You should see the message **`Routed to css()`**. In this case, the path component of **`http://127.0.0.1:5000/css`** is **`'/css'`**, which matches the route specified on line **`9`**. The request is handed over to the corresponding **`css()`** function defined on lines **`10`** and **`11`**.

3) Go to **`http://127.0.0.1:5000/CSS`**
A **`404 (Not Found)`** error is seen. This is because the routes are **case-sensitive**.

4) Go to **`http://127.0.0.1:5000/no_slash/`**
A **`404 (Not Found)`** error is seen. In general, a request's path and a route's path must match **exactly** for the route to be chosen. For line **`13`**, the route specified in the decoration is '**`/no_slash`**' (no trailing forward slash). This means the route is matched only exactly by **`http://127.0.0.1:5000/no_slash`** (no trailing forward slash).

5) Go to **`http://127.0.0.1:5000/optional_slash`**
Notice that the decoration used is **`'/optional_slash/'`**. Despite so, the page could be returned. This is because when Flask fails to find a route for a path that does NOT end with a slash, it will append a slash to the path and try again before giving up completely. This means that both **`http://127.0.0.1:5000/optional_slash`** (without a trailing slash) and **`http://127.0.0.1:5000/optional_slash/`** (with a trailing slash) both match the route for **`'/optional_slash/'`** and will run the associated **`optional_slash()`** function.



```
http://127.0.0.1:5000/no_slash

@app.route('/no_slash')
def no_slash():
    return 'Routed to no_slash()'

http://127.0.0.1:5000/no_slash/
```

```
http://127.0.0.1:5000/optional_slash

@app.route('/optional_slash/')
def optional_slash():
    return 'Routed to optional_slash()'

http://127.0.0.1:5000/optional_slash/
```

Exact match
Flask does NOT automatically remove extra slash
Flask automatically inserts missing slash
Exact match

2.2.3 Multiple Decorators

Exercise 4
Try running the following code on both IDLE (as **`multiple_decorators.py`**) and Jupyter Notebook.

```
1  import flask
2
3  app = flask.Flask(__name__)
4
5  @app.route('/one/')
6  @app.route('/one/two/')
7  @app.route('/three/two/one')
8  def multiple():
9    return 'Routed to multiple()'
10
11 if __name__ == '__main__':
12   app.run()
```

Multiple paths can also be routed to the same function if required. For instance, the three decorations on lines **`5`** to **`7`** associate all of the following URLs with the **`multiple()`** function defined on lines **`8`** and **`9`**:

Try visiting

- **`http://127.0.0.1:5000/one/`**
- **`http://127.0.0.1:5000/one/two/`**
- **`http://127.0.0.1:5000/three/two/one`**

What do you observe?

2.2.4 Variable Routes

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

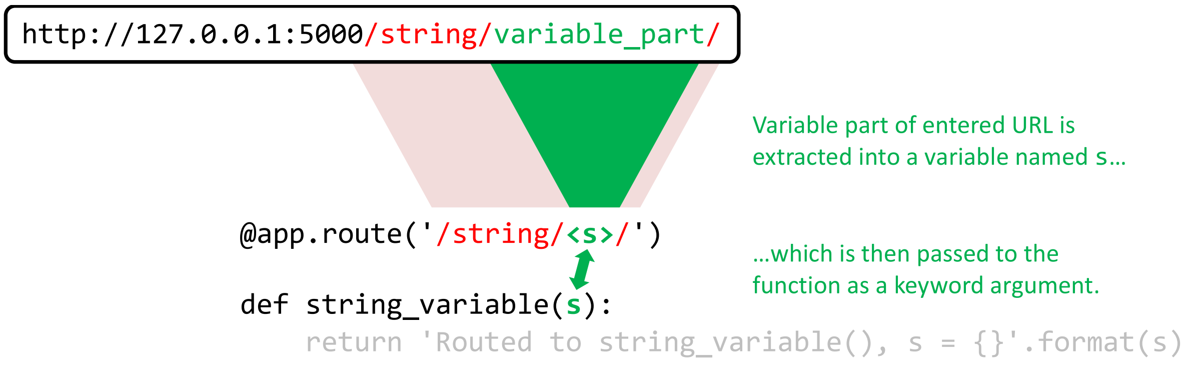
```
1   import flask
2
3   app = flask.Flask(__name__)
4
5   @app.route('/string/<s>/')
6   def string_variable(s):
7     return 'Routed to string_variable(), s = {}'.format(s)
8
9   if __name__ == '__main__':
10    app.run()
```

Flask routes can also have **variable** parts where each variable part in the route's path has a name surrounded by angled brackets **< >**.

By default, each variable part matches any non-empty sequence of characters that **does not** contain a slash /. The route is matched if all variable parts in the path can be matched in this way. When this happens, the variable parts are extracted into **`str`** values and passed to the associated Python function as keyword arguments.

For instance, line **5** defines a route specified by **`'/string/<s>/'`** that contains one variable part named **`s`**. This route matches any path starting with **`/string/`** followed by 1 or more non-slash characters and an optional trailing slash.

If a match is found, the variable part is extracted into a **`str`** value and passed to the function **`string_variable()`** as a keyword argument named **`s`**. Otherwise, Flask will try to find another route that matches, returning a **`404 (Not Found)`** error if none can be found.



```
http://127.0.0.1:5000/string/variable_part/

@app.route('/string/<s>/')
def string_variable(s):
    return 'Routed to string_variable(), s = {}'.format(s)
```

Variable part of entered URL is extracted into a variable named s...
...which is then passed to the function as a keyword argument.

The table below tabulates some URLs that match the **`'/string/<s>/'`** route and some URLs that do not. Note how the variable **`s`** **cannot** be matched to an empty string or any portion of the path that contains a slash **`/`**.

URL	Result	Content of s
`http://127.0.0.1:5000/string/hello/`	`200 OK`	`hello`
`http://127.0.0.1:5000/string/hello`	`200 OK`	`hello`
`http://127.0.0.1:5000/string/123`	`200 OK`	`123`
`http://127.0.0.1:5000/string/`	`404 Not Found`	
`http://127.0.0.1:5000/string/hi/there`	`404 Not Found`	
`http://127.0.0.1:5000/string//`	`404 Not Found`	

Exercise 6
Try running the following code on both IDLE (as `variable_integer_routes.py`) and Jupyter Notebook.

```
1   import flask
2
3   app = flask.Flask(__name__)
4
5   @app.route('/integer/<int:i>/')
6   def integer_variable(i):
7     return 'Routed to integer_variable(), i = {}'.format(i)
8
9   if __name__ == '__main__':
10    app.run()
```

Variable parts can also specify a **converter** using the syntax `<converter:name>` to modify the matching algorithm and convert the matched string to another type before being passed over to the function as a keyword argument.

For instance, line `5` defines a route specified by `'/integer/<int:i>'` with one variable part named `i` that uses the `int` converter. The `int` converter modifies the matching algorithm for `i` so only digits (i.e., 0 to 9) are accepted. This means that only paths that start with `/integer/` followed by one or more digits and an optional trailing slash will result in a match.

If a match is found, the digits portion of the path is extracted and converted to an int before being passed to the `integer_variable()` function as an `int` parameter. Otherwise, the match fails and Flask will try to find another route that matches, returning a `404 (Not Found)` if no matching route can be found.

The table below tabulates some URLs that match the `'/integer/<int:i>'` route and some URLs that do not. Note that as the `int` converter only accepts digits, negative integers that start with a minus sign **cannot** be matched by `<int:i>`.

URL	Result	Content of Variable `i`
`http://127.0.0.1:5000/integer/123/`	`200 OK`	`123`
`http://127.0.0.1:5000/integer/123`	`200 OK`	`123`
`http://127.0.0.1:5000/integer/0`	`200 OK`	`0`
`http://127.0.0.1:5000/integer/`	`404 Not Found`	
`http://127.0.0.1:5000/integer/-123`	`404 Not Found`	
`http://127.0.0.1:5000/integer/one`	`404 Not Found`	

Exercise 7
Complete the following program so it greets the user when the site is visited with the user's name in the path (as long as the name does not include a slash):

```
import flask

app = flask.Flask(__name__)

@app.route('/<name>')
def home(name):
  return 'Hello, {}!'.format(name)

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

2.2.5 Routing by HTTP Methods

Exercise 8
Try running the following code on both IDLE (as `http_routes.py`) and Jupyter Notebook.

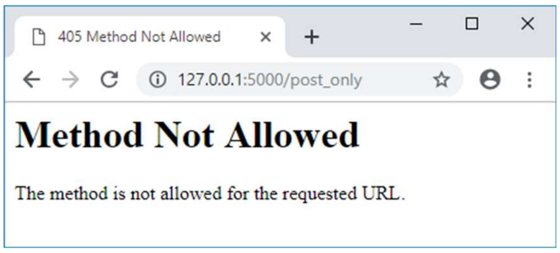
```
1   import flask
2
3   app = flask.Flask(__name__)
4
5   @app.route('/post_only/', methods=['POST'])
6   def post_only():
7     return 'Routed to post_only()'
8
9   if __name__ == '__main__':
10    app.run()
```

Besides matching based on paths, each route can also specify if it only applies to `GET` requests, `POST` requests or **both**. (Recall that `GET` is used to retrieve data without making changes while `POST` is used to submit or make changes to the server's data.)

To comply with this definition, a route for something that changes data on the server permanently (e.g., deletion) should not be accessible via `GET`. This prevents adding, deleting or updating of data on the server without submitting a form and sending a `POST` request.

To limit the HTTP methods accepted by a route, we pass in a list of permitted HTTP methods as a keyword argument named `methods` in the decorator.

For instance, in **Exercise 8**, line `5` specifies a route for `/post_only` that is only accessible using `POST`. If we try to access `http://127.0.0.1:5000/post_only` using `GET` by entering it directly into the address bar, we get a `405 (Method Not Allowed)` error instead.


405 Method Not Allowed

127.0.0.1:5000/post_only

Method Not Allowed

The method is not allowed for the requested URL.

To produce a `POST` request, we need to reach the route by submitting a HTML form. This will be demonstrated later in this lesson.

2.3 Generating Paths from Function Names

Routes provide a mapping from paths to Python functions. However, we often need to go in the opposite direction and generate the path for a given Python function.

To do this, we call the **`url_for()`** function in the flask module and pass it a string with the function's name. If the path has any variables (e.g. **`s`** in **`"/string/<s>"`**), they should be provided as keyword arguments to **`url_for()`**.

Exercise 9
Try running the following code on both IDLE (as **`url_for.py`**) and Jupyter Notebook.

```
1  import flask
2  from flask import url_for
3
4  app = flask.Flask(__name__)
5
6  @app.route('/')
7  def home():
8    url1 = url_for('fixed_route')
9    url2 = url_for('string_variable', s='example')
10   url3 = url_for('integer_variable', i=2020)
11   print(url1)
12   print(url2)
13   print(url3)
14   return 'Check your shell or command prompt window'
15
16 @app.route('/fixed/')
17 def fixed_route():
18   return 'Routed to fixed()'
19
20 @app.route('/string/<s>')
21 def string_variable(s):
22   return 'Routed to string_variable(), s = {}'.format(s)
23
24 @app.route('/integer/<int:i>')
25 def integer_variable(i):
26   return 'Routed to integer_variable(), i = {}'.format(i)
27
28 if __name__ == '__main__':
29   app.run()
```

In **Exercise 9**, the code specifies some routes and prints three generated paths in the shell or command prompt window when the "root" site **`http://127.0.0.1:5000/`** is visited. Notice how **`url_for()`** is used on lines **`8`** to **`10`**.

The expected output is as follows (you may need to look for these lines among the other logging output from Flask):

```
/fixed/
/string/example
/integer/2020
```

These correspond to calling **`fixed_route()`**, **`string_variable(s='example')`** and **`integer_variable(i=2020)`** respectively.

Exercise 10

Enter the following code on both IDLE (as `practice.py`) and Jupyter Notebook.

```
import flask

app = flask.Flask(__name__)

NAMES = ['January', 'February', 'March', 'April', 'May', 'June',
'July', 'August', 'September', 'October', 'November', 'December]

@app.route('/')
def home():
  return 'Home'

@app.route('/<int:month>/')
def name_month(month):
  if month in range(1, 13):
    return 'Month {}: {}'.format(month, NAMES[month - 1])
  return 'Invalid month'

@app.route('/compare/<float:temp>/')
def compare_temp(temp):
  if temp > 35.5:
    return 'It\'s hot!'
  if temp < 25.5:
    return 'It\'s cold!'
  return 'It\'s normal!'

@app.route('/greet/')
def greet():
  return 'Hello!'

@app.route('/greet/<name>/')
def greet_name(name):
  return 'Hello, {}!'.format(name)

@app.route('/data/', methods=['POST'])
def post_data():
  return 'You are using POST'

@app.route('/data/', methods=['GET'])
def get_data():
  return 'You are using GET'

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

Using the code from **Exercise 10**, predict the output when each of the URLs below is visited using a browser. If you predict that a HTTP error would occur instead, write `"ERROR"` as the output.

After completing the table, visit the URLs to check your answers.

No.	URL	Output
1	`http://127.0.0.1:5000/`	
2	`http://127.0.0.1:5000/10`	
3	`http://127.0.0.1:5000/10/20/`	
4	`http://127.0.0.1:5000/20/`	
5	`http://127.0.0.1:5000/compare/35.4`	
6	`http://127.0.0.1:5000/compare/35.6/`	
7	`http://127.0.0.1:5000/compare/`	
8	`http://127.0.0.1:5000/greet/world/`	
9	`http://127.0.0.1:5000/greet/worLD/`	
10	`http://127.0.0.1:5000/Greet/world/`	
11	`http://127.0.0.1:5000/greet/Mei Yi/`	
12	`http://127.0.0.1:5000/greet/`	
13	`http://127.0.0.1:5000/data/`	

3 HTTP Responses and Status Codes

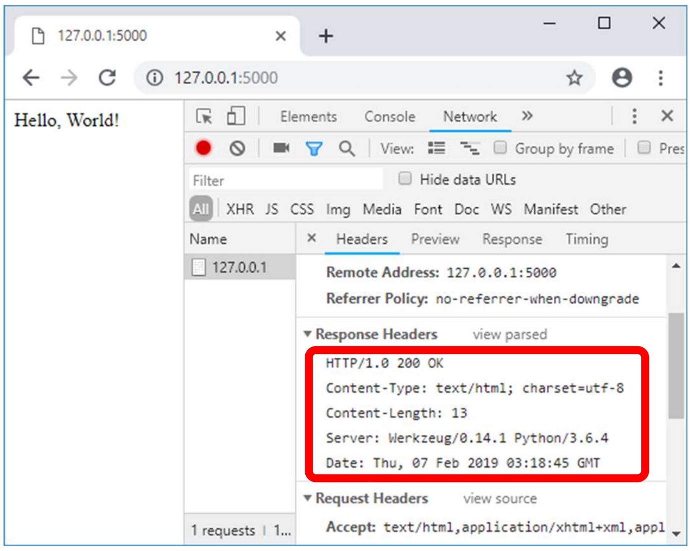
Thus far, we have being writing functions that return short strings that appear to display without problems in the web browser.

However, Flask has actually been prepending various headers behind the scenes to produce valid HTTP responses.

Exercise 11
Try running the following code on both IDLE (as helloworld.py) and Jupyter Notebook.

```
import flask

app = flask.Flask(__name__)

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

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

With Google Chrome's Developer Tools opened, visit **`http://127.0.0.1:5000/`** and examine the response headers. You should see that Flask has actually added various headers to form a complete HTTP response.



Hello, World!

Remote Address: 127.0.0.1:5000
Referrer Policy: no-referrer-when-downgrade

▾ Response Headers

```
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 13
Server: Werkzeug/0.14.1 Python/3.6.4
Date: Thu, 07 Feb 2019 03:18:45 GMT
```

▾ Request Headers

Accept: text/html,application/xhtml+xml,appl

Together with the content, the complete HTTP response sent to your browser is similar to the following:

```
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 13
Server: Werkzeug/0.14.1 Python/3.7.0
Date: Tue, 22 Jan 2019 08:01:52 GMT

Hello, World!
```

Notice that Flask assumes that our output has a Content-Type of `"text/html"`.

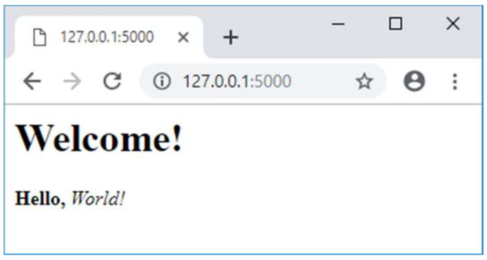
This means that Flask actually expects our function to return a full HTML document and not just a plain string.

However, most browsers are very forgiving and will treat our string as a snippet of HTML intended for the document's `<body>`.

Exercise 12
Try running the following code on both IDLE (as `return_html.py`) and Jupyter Notebook.

```
import flask

app = flask.Flask(__name__)

@app.route('/')
def index():
  return '<h1>Welcome!</h1> <b>Hello,</b> <i>World!</i>'

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

Exercise 12 shows that HTML tags can be used in our return value. Notice that the tags have been appropriately interpreted has HTML.



Welcome!

Hello, *World!*

However, be aware that this is not correct practice. In fact, the functions mapped to the routes should really return full HTML documents.

3.1 Changing the Status Code

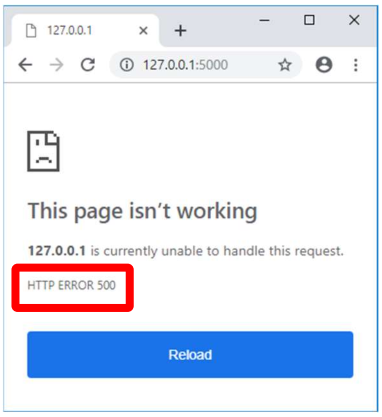
By default, Flask also assumes that our responses have a HTTP status code of `200 (OK)`. This is usually what we want, but if needed we can override this by returning a tuple instead of just a string. The replacement HTTP status code should be provided as the second item of the tuple.

Exercise 13
Try running the following code on both IDLE (as return_status.py) and Jupyter Notebook.

```
import flask

app = flask.Flask(__name__)

@app.route('/')
def index():
  return ('', 500)

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

Line **7** in **Exercise 13** demonstrates the use of a tuple to return the status code **500**, which represents an Internal Server Error.

If we run this program and try to visit **`http://127.0.0.1:5000/`**, we should be greeted with the status code **`HTTP 500 (Internal Server Error)`**:



```
This page isn't working

127.0.0.1 is currently unable to handle this request.

HTTP ERROR 500
```

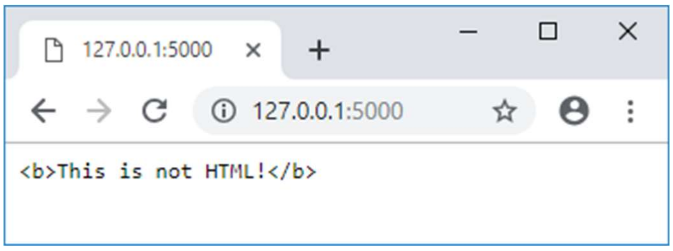
3.2 Changing the Response Headers

We can also add additional response headers by putting them into a Python dictionary and returning this dictionary as the third item of our return tuple. For instance, if we want the web browser to treat our response as plain text instead of HTML, we can replace the Content-Type header value with "`text/plain`" instead.

Exercise 14
Try running the following code on both IDLE (as **`plain_text.py`**) and Jupyter Notebook.

```
import flask

app = flask.Flask(__name__)

@app.route('/')
def index():
  headers = {'Content-Type': 'text/plain'}
  return ('<b>This is not HTML!</b>', 200, headers)

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

In **Exercise 14**, visiting **`http://127.0.0.1:5000/`** demonstrates that the string we return is no longer treated as HTML by the browser:



```
<b>This is not HTML!</b>
```

3.3 Redirecting to Another URL

Besides overriding the HTTP status code and response headers, Flask also lets us generate a response that tells the web browser to load a different URL instead. This is called a **redirect** and is useful when the location of a document has moved or when we want to let another Flask route take over the handling of a request.

To perform a redirect, import the `redirect()` function from the `flask` module and call it with the destination URL or path as the first argument. Then, use the response generated by `redirect()` as the return value of the function.

Exercise 15
Try running the following code on both IDLE (as `redirect_ext.py`) and Jupyter Notebook.

```
import flask
from flask import redirect

app = flask.Flask(__name__)

@app.route('/')
def index():
  return redirect('http://example.com')

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

Instead of redirecting to an external site, we usually want to redirect the user to another of our routes instead. In such cases, we should use `url_for()` to look up the correct path based on the function that we want to reach:

Exercise 16
Try running the following code on both IDLE (as `redirect_int.py`) and Jupyter Notebook.

```
1  import flask
2  from flask import redirect, url_for
3
4  app = flask.Flask(__name__)
5
6  @app.route('/new_url/')
7  def moved_index():
8    return 'You have reached the new URL!'
9
10 @app.route('/')
11 def index():
12   return redirect(url_for('moved_index'))
13
14 if __name__ == '__main__':
15   app.run()
```

Notice how `url_for()` is used to look up the path for `redirect()` on line `12` instead of hardcoding the redirected path as a string.

4 Templates and Rendering

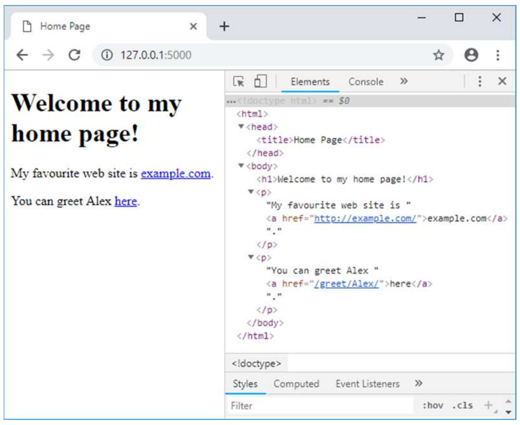
Instead of returning short HTML snippets or redirecting users, we can return full HTML documents complete with headings and hyperlinks in our Flask application.

Exercise 17
Try running the following code on both IDLE (as **`response_without_templates.py`**) and Jupyter Notebook.

```
import flask

app = flask.Flask(__name__)

@app.route('/')
def home():
  html = '<!DOCTYPE html>\n<html>'
  html += '<head><title>Home Page</title></head>'
  html += '<body><h1>Welcome to my home page!</h1>'
  html += '<p>My favourite web site is '
  html += '<a href="http://example.com/">'
  html += 'example.com</a>.</p>'
  html += '<p>You can greet Alex '
  html += '<a href="/greet/Alex/">here</a>.</p>'
  html += '</body></html>'
  return html

@app.route('/greet/<name>/')
def greet(name):
  html = '<!DOCTYPE html>\n<html>'
  html += '<head><title>Greetings!</title></head>'
  html += '<body><h1>Hello, {}!</h1>'.format(name)
  html += '</body></html>'
  return html

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

Upon visiting **`http://127.0.0.1:5000/`**, a full HTML page appears, complete with page title, headings and hyperlinks.

The View Source feature **`(Ctrl-U)`** can be used to verify that the HTML for the page comes directly from the **`home()`** function of the above Python program.



Welcome to my home page!

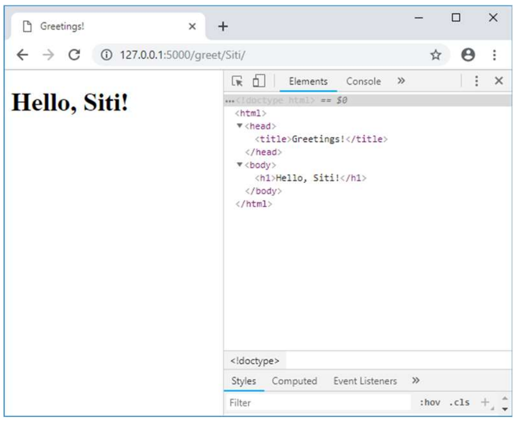
My favourite web site is example.com.

You can greet Alex here.

```
<!doctype html> == $0
<html>
 ▼<head>
    <title>Home Page</title>
  </head>
 ▼<body>
    <h1>Welcome to my home page!</h1>
   ▼<p>
      "My favourite web site is "
      <a href="http://example.com/">example.com</a>
      "."
    </p>
   ▼<p>
      "You can greet Alex "
      <a href="/greet/Alex/">here</a>
      "."
    </p>
  </body>
</html>
```

<!doctype>

Visit **`http://127.0.0.1:5000/greet/Siti/`**. The result has a page title and the greeting for Siti is formatted to look like a heading using the **`<h1>`** tag.



Hello, Siti!

```
<!doctype html> == $0
<html>
▼<head>
    <title>Greetings!</title>
  </head>
▼<body>
    <h1>Hello, Siti!</h1>
  </body>
</html>
```

<!doctype>

4.1 The Need for Templates

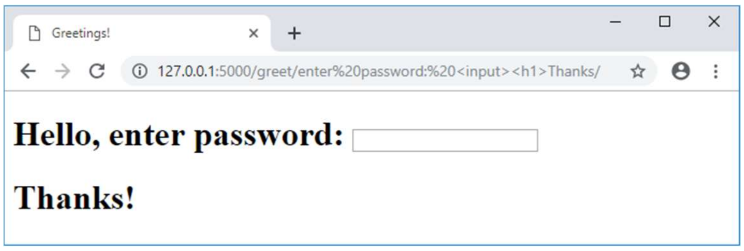
Generating dynamic HTML content the way shown in **Exercise 17** is powerful but dangerous.

Notice that in the **`greet_name()`** function we inserted the `name` variable into our output without checking its contents. This lets a malicious user inject his or her own code into the output and potentially cause all kinds of mischief.

For instance, the following link we bring users to what appears to be a legitimate form asking for the user's password.

`127.0.0.1:5000/greet/enter%20password:%20<input><h1>Thanks/`

However, in reality, the form does not come from our application at all and was injected into the page from HTML code in the URL's path.



Hello, enter password:

Thanks!

Besides possible security issues, constructing HTML documents by joining Python strings can also become quite messy. For instance, it is easy to be confused between the use of HTML and Python in the **`home()`** and **`greet_name()`** functions.

This is why, in practice, we usually do not generate HTML responses by manually manipulating strings in Python code. Instead, we put the HTML content in a separate file called a **template** with placeholders for where the dynamic content should be inserted.

When we need to output HTML, we use a **template engine** to load the template and fill in the placeholders. The template engine also helps to escape special characters such as **<** and **>** when filling in the placeholders so that HTML injection such as the case above is avoided. This process of filling in the placeholders to produce the final HTML that is used for the response is called **rendering**.

4.2 The Jinja2 Template Engine

Flask provides a built-in template engine named **Jinja2**.

By default, Flask expects all templates to be located in a subfolder named `templates`.

To start using Jinja2, create a subfolder named `templates` in the folder where your Flask programs are stored, then save the following file in this folder as `home.html`:

Exercise 18
Create the following template using Notepad++ and save it as `templates/home.html`

```
<!DOCTYPE html>
<html>
  <head>
    <title>Home Page</title>
  </head>
  <body>
    <h1>Welcome to my home page!</h1>
    <p>My favourite web site is
      <a href="http://example.com/">example.com</a>.
    </p>
    <p>You can greet Alex
      <a href="{{ url_for('greet', name='Alex') }}">here</a>.
    </p>
  </body>
</html>
```

Exercise 19
Create the following template using Notepad++ and save it as `templates/greet.html`

```
<!DOCTYPE html>
<html>
  <head>
    <title>Greetings!</title>
  </head>
  <body>
    <h1>Hello, {{ visitor }}!</h1>
  </body>
</html>
```

In a Jinja2 template, placeholders are surrounded by double braces and have the form `{{ expression }}` where `expression` is a **Jinja2 expression**.

When using templates, be careful not to confuse Jinja2 expressions with Python expressions. Although they are very similar, Jinja2 expressions and Python expressions have different syntaxes and operate in different environments.

For instance, `len()` and many other standard Python functions are not available in Jinja2 expressions. You will learn more about the differences between the two as you progress.

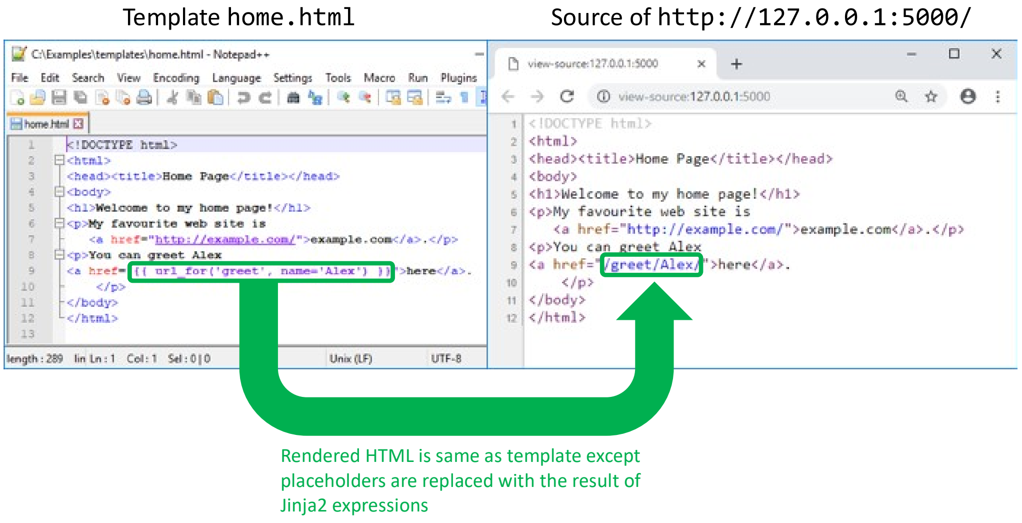
Recall that the process of filling in the placeholders of a template is called **rendering**. The task of rendering a template is typically performed by a function named `render_template()` that can be imported from the `flask` module.

Exercise 20
Try running the following code on both IDLE (as `response_with_templates.py`) and Jupyter Notebook.

```python
import flask
from flask import render_template

app = flask.Flask(__name__)

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

@app.route('/greet/<name>/')
def greet(name):
  return render_template('greet.html', visitor=name)

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

Upon visiting `http://127.0.0.1:5000/`, view the source by pressing `Ctrl-U` to verify that the template is indeed rendered and no placeholders are present.

Click the link to visit `http://127.0.0.1:5000/greet/Alex/` and verify that there are no placeholders in its HTML source as well.


Template home.html
Source of http://127.0.0.1:5000/
C:\Examples\templates\home.html - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins
home.html
length : 289 lin Ln : 1 Col : 1 Sel : 0 | 0 Unix (LF) UTF-8
view-source:127.0.0.1:5000
Rendered HTML is same as template except placeholders are replaced with the result of Jinja2 expressions

```html
<!DOCTYPE html>
<html>
<head><title>Home Page</title></head>
<body>
<h1>Welcome to my home page!</h1>
<p>My favourite web site is
   <a href="http://example.com/">example.com</a>.</p>
<p>You can greet Alex
<a href="{{ url_for('greet', name='Alex') }}">here</a>.
   </p>
</body>
</html>
```

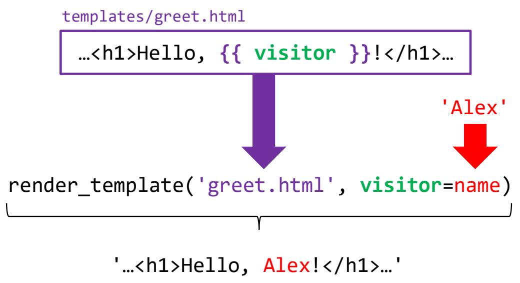
```html
<!DOCTYPE html>
<html>
<head><title>Home Page</title></head>
<body>
<h1>Welcome to my home page!</h1>
<p>My favourite web site is
   <a href="http://example.com/">example.com</a>.</p>
<p>You can greet Alex
<a href="/greet/Alex/">here</a>.
   </p>
</body>
</html>
```

4.3 Passing Values to Templates

The **`render_template()`** function accepts the name of a template in the `templates` subfolder as its first argument, followed by 0 or more keyword arguments assigning values to Jinja2 variables that may be used by the template.

For instance, in **Exercise 20**, line **`12`** of the Python code renders a template named **`greet.html`** and specifies that within the template, the value of **`name`** should be assigned to a Jinja2 variable named **`visitor`**.

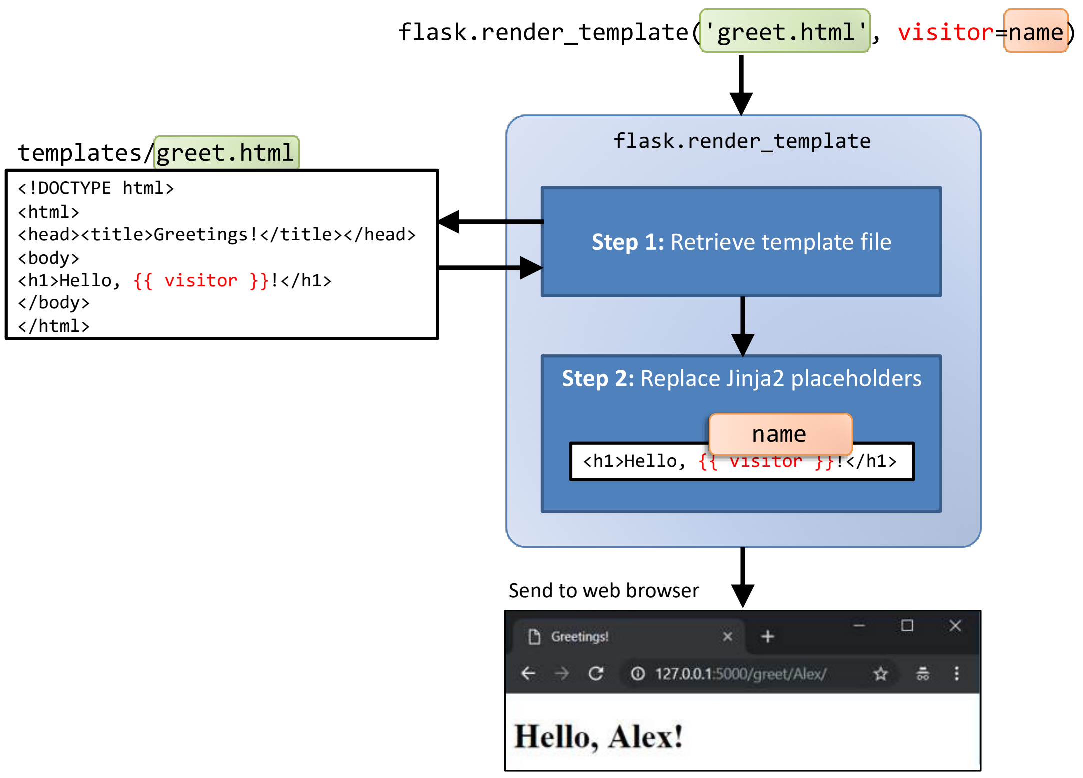
This corresponds to line **`5`** of the HTML template **`greet.html`** where the Jinja2 placeholder **`{{ visitor }}`** is replaced by this value.


templates/greet.html
…<h1>Hello, {{ visitor }}!</h1>…
'Alex'
render_template('greet.html', visitor=name)
'…<h1>Hello, Alex!</h1>…'

Do not confuse Jinja2 variables with Python variables.

If **`render_template()`** is called without any keyword arguments, values from the Python environment are NOT passed to the template. To use Python values in the generated HTML, we must pass them over as keyword arguments when **`render_template()`** is called.

The following diagram illustrates how **`render_template()`** retrieves a template file and replaces its placeholders to produce the final HTML that is sent to the browser:


flask.render_template('greet.html', visitor=name)
flask.render_template
Step 1: Retrieve template file
Step 2: Replace Jinja2 placeholders
name
<h1>Hello, {{ visitor }}!</h1>
templates/greet.html
Send to web browser
Greetings!
127.0.0.1:5000/greet/Alex/
Hello, Alex!

```
<!DOCTYPE html>
<html>
<head><title>Greetings!</title></head>
<body>
<h1>Hello, {{ visitor }}!</h1>
</body>
</html>
```

Exercise 21A

Create the following four templates and save them in a **`templates/`** subfolder.

`templates/A.html`

```
<!doctype html>
<html>
  <head>
    <title>A</title>
  </head>
  <body>
    A
  </body>
</html>
```

`templates/B.html`

```
<!doctype html>
<html>
  <head>
    <title>B</title>
  </head>
  <body>
    B
  </body>
</html>
```

`templates/C.html`

```
<!doctype html>
<html>
  <head>
    <title>C</title>
  </head>
  <body>
    C
  </body>
</html>
```

`templates/D.html`

```
<!doctype html>
<html>
  <head>
    <title>D</title>
  </head>
  <body>
    D
  </body>
</html>
```

Each template in Exercise 21A is meant to be shown for a different URL:

URL	Template to use
`http://127.0.0.1:5000/A`	`A.html`
`http://127.0.0.1:5000/B`	`B.html`
`http://127.0.0.1:5000/C`	`C.html`
`http://127.0.0.1:5000/D`	`D.html`

Exercise 21B
Complete the code for `app.py` shown below. Lines `4`, `8`, `12`, and `16` are missing

```
1  import flask
2  app = flask.Flask(__name__)
3
4
5  def view_A():
6    return flask.render_template('A.html')
7
8
9  def view_B():
10   return flask.render_template('B.html')
11
12
13 def view_C():
14   return flask.render_template('C.html')
15
16
17 def view_D():
18   return flask.render_template('D.html')
19
20 if __name__ == '__main__':
21   app.run()
```

Exercise 21C
Generalise the Python program code and HTML templates so that only one template is needed.

`template.html` (Complete lines `5` and `9`)

```
1  <!doctype html>
2  <html>
3    <head>
4      <title>
5
6      </title>
7    </head>
8    <body>
9
10   </body>
11 </html>
```

`app.py` (Complete lines `4` to `6`)

```
1  import flask
2  app = flask.Flask(__name__)
3
4
5  def view(      ):
6
7
8  if __name__ == '__main__':
9      app.run()
```

Notice that this application allows other combinations of URL, for example:
`http://127.0.0.1:5000/BBB` and `http://127.0.0.1:5000/q`.

[Challenge] Edit the program so only the four cases given are allowed and error code 500 is returned for other cases.

4.4 Avoiding Hardcoded Links

Although most Python functions are not available from Jinja2, **`render_template()`** automatically includes Flask's `url_for()` function so paths can be generated based on the Python function that needs to be called instead being hardcoded. This is most useful for creating HTML links, as demonstrated by the following lines in **Exercise 18**:

```
<p>You can greet Alex
    <a href="{{ url_for('greet', name='Alex') }}">here</a>.
```

Using `url_for()` is more future-proof than hardcoding the path like this:

```
<p>You can greet Alex <a href="/greet/Alex/">here</a>.
```

While hardcoded links are shorter, they need to be updated every time we change the path that is routed with a function.

Using `url_for()` lets us update our paths without changing every template that links to it.

Exercise 22
Create the following program `app.py`. Thereafter, use the `url_for()` function to complete the `links.html` template so that each link leads the respective message.

`app.py`

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

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

@app.route('/greeting')
def hello():
  return 'Hello, World!'

@app.route('/<int:year>')
def report(year):
  return 'year is ' + str(year)

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

`templates/links.html`

```
<!doctype html>
<html>
  <head>
    <title>Links</title>
  </head>
  <body>
    <p><a href=                        >Hello, World!</a></p>
    <p><a href=                                   >2023</a></p>
  </body>
</html>
```

5 Jinja 2 Filters

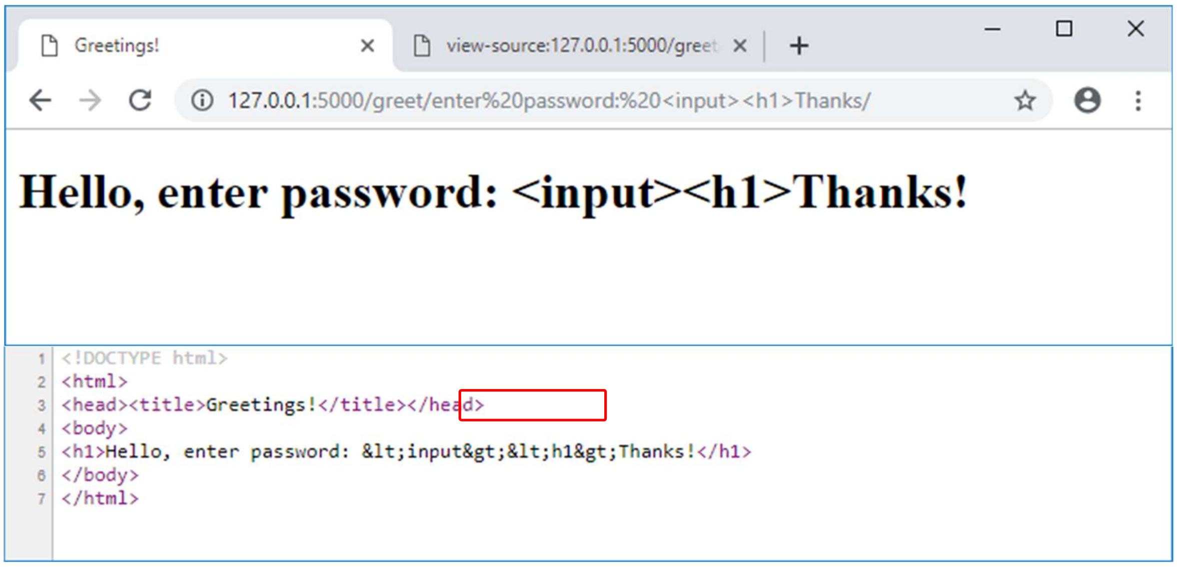
5.1 The `safe` Filter

Run the code in **Exercise 20**. Visit the URL that previously allowed HTML injection into the greeting page when templates was not used:

```
http://127.0.0.1:5000/greet/enter%20password:%20<input><h1>Thanks/
```

This time, however, the injected code is displayed as plain text instead of being treated as HTML.

If you press **`Ctrl-U`** to view source, you would notice that the lesser-than **<** and greater-than **>** signs have been escaped with the appropriate HTML entities:


Greetings!
view-source:127.0.0.1:5000/greet
127.0.0.1:5000/greet/enter%20password:%20<input><h1>Thanks/

Hello, enter password: <input><h1>Thanks!

```
<!DOCTYPE html>
<html>
<head><title>Greetings!</title></head>
<body>
<h1>Hello, enter password: &lt;input&gt;&lt;h1&gt;Thanks!</h1>
</body>
</html>
```

Jinja2 automatically performs this escaping for us so user input can be used in Jinja2 expressions freely without any safety concerns.

However, if we are sure that the result of a Jinja2 expression is safe and should be treated as raw HTML, we can notify Jinja2 of this fact by using the **`safe` filter**.

To apply a filter to an expression, we follow the expression with a bar character | and the name of the filter.

Exercise 23
Create the HTML template **`custom.html`**. Then create and run the server code **`raw_html_with_safe_filter.py`**.

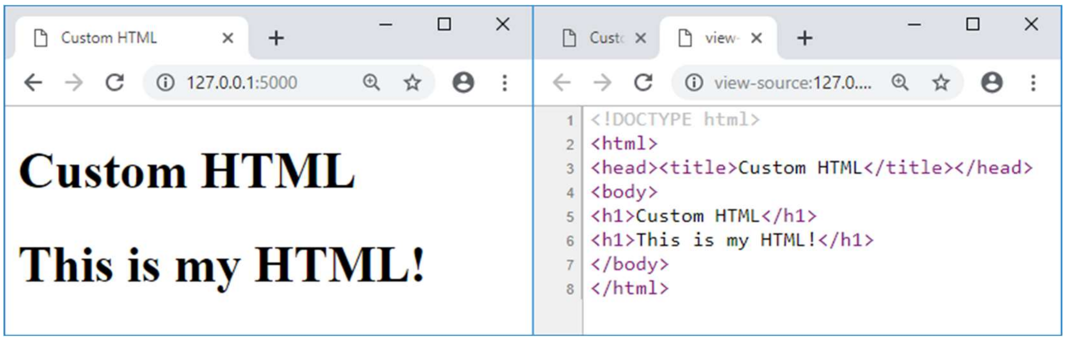
`custom.html`

```
<!DOCTYPE html>
<html>
  <head>
    <title>Custom HTML</title>
  </head>
  <body>
    <h1>Custom HTML</h1>
    {{ my_html|safe }}
  </body>
</html>
```

`raw_html_with_safe_filter.py`

```
import flask
from flask import render_template

app = flask.Flask(__name__)

@app.route('/')
def home():
  return render_template('custom.html',
    my_html='<h1>This is my HTML!</h1>')

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

Visit **`http://127.0.0.1:5000/`** The string passed to the template as **`my_html`** is rendered as raw HTML as shown

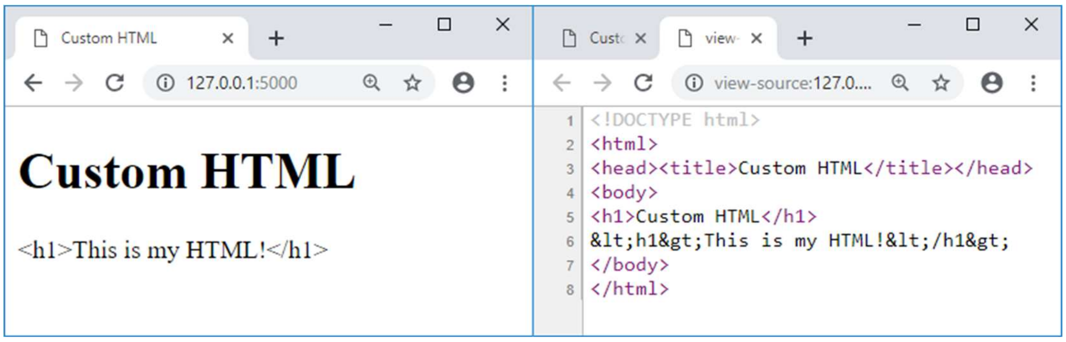


Custom HTML

This is my HTML!

```
<!DOCTYPE html>
<html>
<head><title>Custom HTML</title></head>
<body>
<h1>Custom HTML</h1>
<h1>This is my HTML!</h1>
</body>
</html>
```

However, if we remove the `safe` filter from **`custom.html`**, **`my_html`** is rendered such that the special characters < and > are escaped, as expected:



Custom HTML

<h1>This is my HTML!</h1>

```
<!DOCTYPE html>
<html>
<head><title>Custom HTML</title></head>
<body>
<h1>Custom HTML</h1>
&lt;h1&gt;This is my HTML!&lt;/h1&gt;
</body>
</html>
```

Technical Note

Restart your program for any changes (such as removing the safe filter) to take effect.

If your program stops reflecting changes made, restart the program, then hold down the `Ctrl` key while refreshing your browser to bypass its cache and perform a fresh HTTP request.

Alternatively, run Flask 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. However, running Flask in debug mode has several incompatibilities with IDLE and Jupyter Notebook.

1. When using debug mode, the usual start-up messages that appear when `app.run()` is called will not appear in IDLE's shell window.

2. Similarly, output from `print()` and error messages produced by your program will not appear in IDLE's shell window.

3. Pressing `Ctrl-C` or restarting IDLE's shell will not stop the server properly, so the next time you try to restart your program, you may receive an error message informing you that port number 5000 is already in use. As a workaround, save your work, then press `Ctrl-Alt-Del` to start Task Manager and close all tasks with the name `python.exe` or `pythonw.exe`. This will close IDLE and any other stray servers that may still be running. After all copies of Python are closed, restart IDLE and try running your program again. This method however does not work for examination laptops.

4. For examination laptops, use a fresh port to circumvent the issue.

If you wish to use debug mode, it is recommended that you run your program from the Command Prompt or PowerShell instead or use an alternative IDE that is fully compatible.

5.2 The `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 list from a template.

One solution would be perform the calculation in Python and pass the value over to the template as a separate Jinja2 variable. However, for simple length checks Jinja2 provides a **`length`** filter that gives the same result as **`len()`** but uses a different syntax:

Exercise 24
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()
```

Visit different URLs such as **`http://127.0.0.1:5000/Elizabeth/`** or **`http://127.0.0.1:5000/Siti/`** and examine how line `7` of the template uses the **`length`** filter to determine the number of characters in **`name`**.

6 Jinja2 Statements

In a typical Flask application, the majority of data processing and computation should be done using Python code. The outputs from this computation are then passed to Jinja2 as numbers, strings, lists and other plain data objects. This data will then used to fill in a template to produce the final HTML.

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 `{%` and `%}`. Unlike placeholders surrounded by double braces `{{` and `}}` that are usually replaced with output, the contents of `{%` and `%}` do not produce any output.

6.1 *`if`* *Statement*

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, Jinja2 is not grouped by indentation, so there needs to be an `endif` command to indicate where the if statement ends. In addition, since the `if`, `elif` and `else` clauses are now demarcated by separate `{%` and `%}` blocks, colons are no longer needed:

Exercise 25
Create the HTML template **`results.html`**. Then create and run the server code **`results_using_if.py`**.

`results.html`

```
<!DOCTYPE html>
<html>
  <head>
    <title>Results</title>
  </head>
  <body>
    <h1>Results</h1>

    {% if greet %}
      <p>Hello, {{ name }}.</p>
    {% endif  %}

    {% if show_score  %}
      <p>Your score is {{ score }}%.</p>
    {% elif score >= 50  %}
      <p>You passed.</p>
    {% else  %}
      <p>You failed.</p>
    {% endif %}

  </body>
</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.

Adjust the **`greet`**, **`name`**, **`show_score`** and **`score`** keyword arguments on lines **`8`** and **`9`**. Restart the server and visit `http://127.0.0.1:5000/` to see how the template is rendered differently. In each case, verify that the output matches up with what you would expect.

6.2 `for-in` Statement

Similar to the **`for`** loop statement 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, since Jinja2 is not grouped by indentation, there needs to be an **`endfor`** command to indicate where the **`for`** statement ends.

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

Exercise 26
Create the HTML template **`table.html`**. Then create and run the server code **`table_using_for.py`**.

table.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Table of Results</title>
  </head>
  <body>
    <h1>Table of Results</h1>
    <table>
      <tr>
        <th>Subject Name</th>
        <th>Score</th>
      </tr>
      {% for subject in results  %}
      <tr>
        <td>{{ subject }}</td>
        <td>{{ results[subject] }}</td>
      </tr>
      {% endfor %}
    </table>
  </body>
</html>
```

```
table_using_for.py
import flask
from flask import render_template

app = flask.Flask(__name__)

@app.route('/')
def home():
  results = {
    'English': 75,
    'Mother Tongue': 73,
    'Maths': 76,
    'Computing': 78
  }
  return render_template('table.html', results=results)

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

In **Exercise 26**, **`table.html`** expects a Jinja2 variable named **`results`** containing a dictionary mapping subject names to scores. The following program creates a hardcoded dictionary matching this format and passes it to the template for rendering.

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

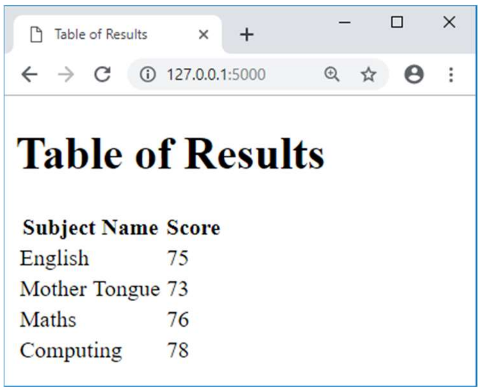


Table of Results

Subject Name	Score
English	75
Mother Tongue	73
Maths	76
Computing	78

In a real web application, we would most likely retrieve the subject names and scores from an SQL database instead of hardcoding them.

7 Static Files

In Flask, a requested path is treated as a string for route matching. The path usually does not refer to the location of real files or folders stored on the server. This allows return of a dynamically-generated response using `render_template()` instead of returning a static file.

However, most HTML documents do not exist in isolation. They typically request for additional resources such as style sheets and images from additional URLs for proper display. Unlike the main content, these additional resources do not change often.

Instead of writing Python functions to generate the required responses, Flask allows the creation of a subfolder `static` (in the same level as `templates`) to store these resources. Flask then sets up a view named `'static'` that (by default) routes any path starting with `/static/` to the contents of this subfolder.

Exercise 27
Create a subfolder `static` in the folder where your Flask programs are stored. Thereafter, create `styles.css` in this folder.

Create a HTML template `stylish.html` that makes use of `styles.css`

Create server code `static_style_sheet.py` that will render `stylish.html`.

`styles.css`

```css
body {background: yellow;}
h1 {border-bottom: 1px solid red; color: red;}
```

`stylish.html`

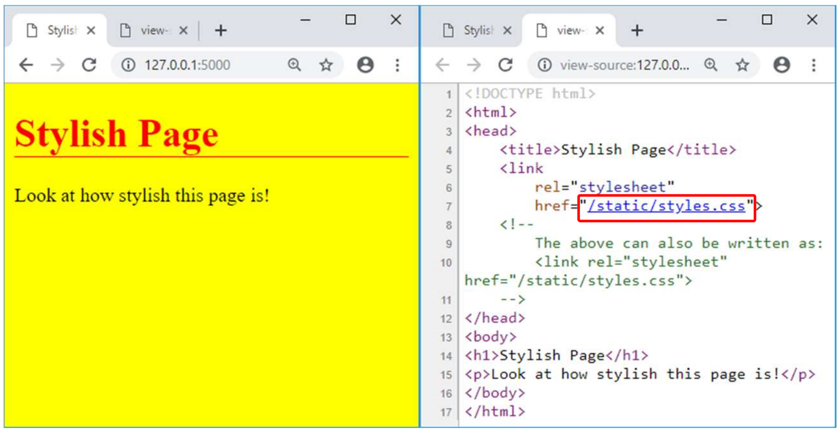
```html
<!DOCTYPE html>
<html>
  <head>
    <title>Stylish Page</title>
    <link rel="stylesheet"
    href="{{ url_for('static', filename='styles.css') }}">
    <!-- For simplicity, the above can also be written as:
      <link rel="stylesheet" href="/static/styles.css">.
      Not recommended though as the link will break if the
      path of the static files changes. -->
  </head>
  <body>
    <h1>Stylish Page</h1>
    <p>Look at how stylish this page is!</p>
  </body>
</html>
```

`static_style_sheet.py`

```python
import flask
from flask import render_template

app = flask.Flask(__name__)

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

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

Note how the path of the static file **`styles.css`** stored within the static subfolder is passed to **`url_for()`** as a keyword argument **`filename`** on line **`7`** of the **`stylish.html`** template.

Visit **`http://127.0.0.1:5000/`** and view source using **`Ctrl-U`**. Verify that the style sheet URL placeholder is replaced with a path starting with /static/:



Stylish Page

Look at how stylish this page is!

```
<!DOCTYPE html>
<html>
<head>
    <title>Stylish Page</title>
    <link
        rel="stylesheet"
        href="/static/styles.css">
    <!--
        The above can also be written as:
        <link rel="stylesheet"
href="/static/styles.css">
    -->
</head>
<body>
<h1>Stylish Page</h1>
<p>Look at how stylish this page is!</p>
</body>
</html>
```

8 Processing Form Data

Thus far, we have been getting input from the user by extracting variables from request URLs. However, it is more common to provide input to a web application via a HTML form.

When a HTML form is submitted, the browser collects all the input as key-value pairs and encodes it into a single string, which is then sent to the server using a HTTP request. Depending on how the HTML form is configured, either a GET or POST request is made.

8.1 `GET` Requests

The default behaviour for HTML forms is to submit a GET request, where the encoded form data is visible in the query portion of the request URL.

For instance, the URL **`http://example.com/hello?name=bala&age=18`** contains a query with two key-value pairs: one for the key **`'name'`** and another for **`'age'`**.

Parsing key-value pairs encoded in query strings can be tedious and error-prone. Flask simplifies that task by handling this parsing and allows the data to be accessed as a dictionary from a **`request`** object that can be imported from the flask module.

Exercise 28A
Create a HTML form template **`form_with_get.html`**.

```
<!DOCTYPE html>
<html>
  <head><title>Form using GET method</title></head>
  <body>
    <form action="{{ url_for('process_with_get') }}">
      <p>Input s: <input name="s"></p>
      <p><input type="submit"></p>
    </form>
  </body>
</html>
```

Notice that in line 5 of **Exercise 28**, `url_for()` was used with a function name of `'process_with_get'` to generate the path that the form data will be submitted to.

Exercise 28B
Create a HTML template `analysis_results.html`.

```
<!DOCTYPE html>
<html>
  <head><title>String Analysis</title></head>
  <body>
    <p>You entered s: {{ s }}</p>
    <p>This string has
       {{ num_vowels }} vowels(s) and
       {{ num_words }} word(s).
  </body>
</html>
```

The template in **Exercise 28B** is used to display the results of processing, which in the case of the code, is to display the number of vowels and number of words in the submitted name.

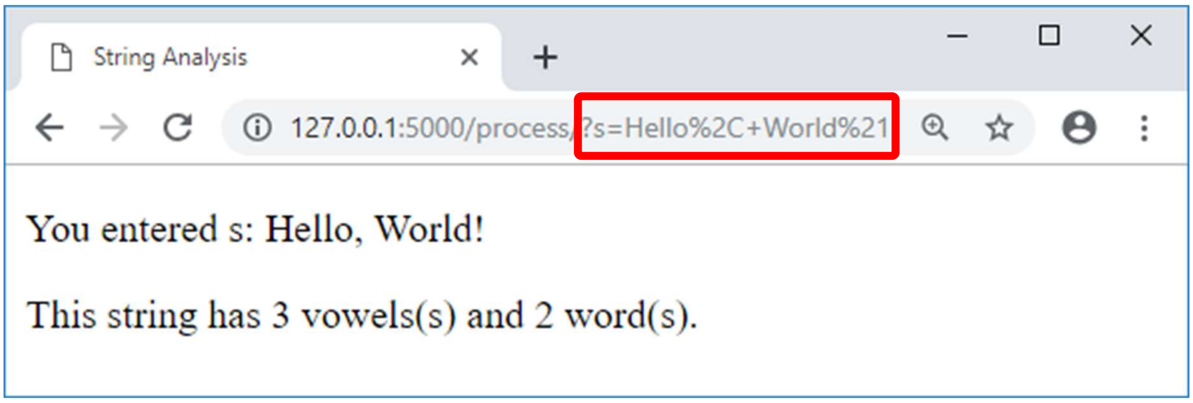
Exercise 28C
Write server code `server.py` that will render the HTML templates in **Exercises 28A** and **28B** as appropriate.

```
import flask
from flask import render_template, request

VOWELS = ['a', 'e', 'i', 'o', 'u']

app = flask.Flask(__name__)

@app.route('/')
def form():
  return render_template('form_with_get.html')

@app.route('/process/')
def process_with_get():
  if 's' in request.args:
    s = request.args['s']
    lower_s = s.lower()
    num_vowels = 0
    for vowel in VOWELS:
      num_vowels += lower_s.count(vowel)
    num_words = len(s.split())
    return render_template('analysis_results.html',
                           s=s,
                           num_vowels=num_vowels,
                           num_words=num_words)
  return 'No form data found!'

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

Notice that following about the Flask application in **Exercise 28C:**

- the `request` object is important from the flask module in line `2`.
- the submitted value of `s` is accessed via the request.args dictionary in line `15.`
- there are two routes:
  - `'/'` renders the HTML form in **Exercise 28 A**.
  - /process/ is associated with the `process_with_get()` function that analyses `s` and renders the result.

Visit `http://127.0.0.1:5000/` and fill in the form. Upon submission, observe that the value of `s` entered is encoded in the resulting URL. This is an implication of using the `GET` method.



You entered s: Hello, World!

This string has 3 vowels(s) and 2 word(s).

Visit `http://127.0.0.1:5000/process/`. Notice that it is possible to manually access the results page without having to fill in and submit the form.

In doing so, the `request.args` dictionary will not include a value for `s` and the error message `'No form data found!'` as coded in line `25`, will be returned instead.

Consider whether this is desirable when the purpose of the form is to collect input.

8.2 `POST` Requests

Submitting form data using a **`GET`** request has several disadvantages:

1. Submitted data is recorded in the resulting URL, which lets anyone viewing the browser history obtain the data that was sent.

2. **`GET`** requests are not supposed to make changes to the server's data. Using data submitted with a **`GET`** request to add, delete or update a database violates HTTP standards.

3. Some browsers and server software limit the length of URLs. Excessively long form data submitted using **`GET`** may become truncated.

To overcome these disadvantages, configure HTML forms to submit the data using **`POST`** requests instead. This can be achieved by setting the **`method`** attribute of the **`<form>`** tag to **`"post"`** or **`"POST"`**.

Using a different HTTP method allows distinction between requests from users clicking a link or entering the URL versus requests from users submitting a form.

An advantage of this is that the same URL can be used for both *displaying* the form as well as *processing* the form.

Exercise 29A
Create a HTML form template **`form_with_post.html`**.

```
1   <!DOCTYPE html>
2   <html>
3     <head><title>POST Form</title></head>
4     <body>
5       <form method="post">
6         <p>Input s: <input name="s"></p>
7         <p><input type="submit"></p>
8       </form>
9     </body>
10  </html>
```

Notice that in **Exercise 29A**, besides adding the **`method`** attribute on line `5`, the **`action`** attribute of the **`<form>`** tag has also been omitted (compare with **Exercise 28A**). This will allow the form to be submitted to the same URL used to generate the form.

Exercise 29B
Adapt `server.py` written in **Exercise 28C** to combine both routes and distinguish between the situations of rendering the form and the results page by looking at the HTTP method used.

The HTTP method can be obtained via the `request.method` string

```
1  import flask
2  from flask import render_template, request
3
4  VOWELS = ['a', 'e', 'i', 'o', 'u']
5
6  app = flask.Flask(__name__)
7
8  @app.route('/', methods=['GET', 'POST'])
9  def index():
10   if request.method == 'GET':
11     return render_template('form_with_post.html')
12   if 's' in request.form:
13     s = request.form['s']
14     lower_s = s.lower()
15     num_vowels = 0
16     for vowel in VOWELS:
17       num_vowels += lower_s.count(vowel)
18     num_words = len(s.split())
19     return render_template('analysis_results.html',
20                            s=s,
21                            num_vowels=num_vowels,
22                            num_words=num_words)
23   return 'No form data found!'
24
25 if __name__ == '__main__':
26   app.run()
```

In **Exercise 29B**, the `index()` function contains code to check if the request method is `GET`, and if so, renders the form. Otherwise, the request method is assumed to be `POST` and the submitted form data is processed instead.

Unlike `GET` requests where submitted form data is available via the `request.args` dictionary, for `POST` requests, submitted form data is placed in the `request.form` dictionary instead. This explains why the value of `s` is accessed via the `request.form` dictionary on line `13`.

The decorator on line `8` must include `'POST'` in its list of allowed methods as Flask routes are only accessible via `'GET'` by default.

Visit `http://127.0.0.1:5000/` and fill in the form. Upon submission, observe that the value of s entered in no longer visible in the resulting URL, which now remains the same.

Notice that in line `12`, there is still a check if the `request.form` dictionary has a value for `s`.

While it is relatively hard for an average user to send phony `POST` requests, it is not technically impossible and the program must be prepared to handle `POST` requests with incomplete or invalid form data. If there is no value for `s` in `request.form`, the error message on line `23` should be returned instead.

Exercise 30A

Create the following HTML template, `form.html`

```
1  <!doctype html>
2  <html>
3    <head>
4      <title>Greeting Form</title>
5    </head>
6    <body>
7      <form method="POST">
8        <input type="text" name="name">
9        <input type="text" name="address">
10       <input type="submit">
11     </form>
12   </body>
13 </html>
```

Exercise 30B

Complete the HTML template `results.html` and server code `app.py` so that visiting `http://127.0.0.1:5000/` will render `form.html` and submitting the form will display `Hello` ***`name`***`, your address is:` ***`address`***, where ***`name`*** is the content of the first text field and ***`address`*** is the content of the second text field:

`results.html` **(Complete line 7)**

```
1  <!doctype html>
2  <html>
3    <head>
4     <title>Greeting</title>
5    </head>
6    <body>
7      Hello               ,  your  address  is:
8    </body>
9  </html>
```

`app.py` **(Complete lines 2 and the code from line 6 onwards)**

```
1  import flask
2  from flask import
3  app = flask.Flask(__name__)
4
5  # Use the space below
6
7
8
9
10
11
12
13
14 if __name__ == '__main__':
15   app.run()
```

9 Handling File and Image Uploads

Recall that an **`<input>`** tag can have its **`type`** attribute set to **`"file"`**. This allows files to be uploaded for submission via HTML forms.

For file uploads to work properly, the enclosing **`<form>`** must also be configured to use **`POST`** and include an additional **`enctype`** attribute that is set to **`"multipart/form-data"`**. This attribute means that one or more sets of data are combined in a single body.

Exercise 31A
Create the HTML template, **`form_with_file_upload.html`** to create a form for uploading photos as well as a link for seeing all the photos that have been uploaded

```
<!DOCTYPE html>
<html>
  <head><title>Photo Upload</title></head>
  <body>
    <form method="post" enctype="multipart/form-data">
      <p>Photo: <input name="photo" type="file"></p>
      <p><input type="submit"></p>
    </form>
    <p><a href="{{ url_for('view') }}">View photos</a></p>
  </body>
</html>
```

Exercise 31B
Create the HTML template, **`view_file_uploads.html`** for a page to view the photos uploaded when the link to view these photos in **Exercise 31A** is clicked.

```
<!DOCTYPE html>
<html>
  <head><title>View Photos</title></head>
  <body>
    {% for photo in photos %}
      <img src="{{ url_for('get_file', filename=photo) }}"
      alt="{{ photo }}">
    {% endfor %}
    <p><a href="{{ url_for('home') }}">Home</a></p>
  </body>
</html>
```

For the main Flask server application, we create three routes:
- one for uploading photos,
- one for seeing all the uploaded photos
- one for retrieving the image data of an uploaded photo given its filename.

We also initialize and use a simple SQLite database to keep track of the photos uploaded:

Exercise 31C
Create a Flask code **server.py** for the photo uploading and viewing application.

```
import flask, os, sqlite3
from flask import render_template, request
from flask import send_from_directory
from werkzeug.utils import secure_filename

if not os.path.isfile('db.sqlite3'):
  db = sqlite3.connect('db.sqlite3')
  db.execute('CREATE TABLE photos(photo TEXT)')
  db.commit()
  db.close()

app = flask.Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def home():
  if request.method == 'POST' and \
    request.files and 'photo' in request.files:
    # Save file
    photo = request.files['photo']
    filename = secure_filename(photo.filename)
    path = os.path.join('uploads', filename)
    photo.save(path)
    # Add filename to database
    db = sqlite3.connect('db.sqlite3')
    db.execute('INSERT INTO photos(photo) VALUES(?)',
               (filename,))
    db.commit()
    db.close()
  return render_template('form_with_file_upload.html')

@app.route('/view')
def view():
  db = sqlite3.connect('db.sqlite3')
  cur = db.execute('SELECT photo FROM photos')
  photos = []
  for row in cur:
    photos.append(row[0])
  db.close()
  return render_template('view_file_uploads.html',
                         photos=photos)

@app.route('/photos/<filename>')
def get_file(filename):
  return send_from_directory('uploads', filename)

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

Before running the program, create an empty **uploads** subfolder (same level as **templates** and **static**) to store the uploaded photos.

Next, run the program and visit **http://127.0.0.1:5000/** to upload some photos (i.e., GIF, JPG or PNG files). When done, click on the "View" link to see the uploaded photos.

Important Notes on File Uploading

- If you try to upload non-photo files such as PDF files, the file will still be uploaded but will not be displayed properly.

- Unlike normal form data, submitted files are not accessed from `request.form` but from a separate `request.files` dictionary (e.g. line `19`).
    - Each value in this dictionary is a file object with a filename attribute as well as a `save()` method that accepts a file path and writes the submitted file onto the server's file system using the provided file path.

- Be careful when let users specify filenames for reading or writing files on the server's file system as file paths can use special folder names such as .. to access parent folders that contain source code or your server's configuration files.
    - To prevent this from happening, pass the filename through the `secure_filename()` function provided in the `werkzeug.utils` module first (e.g. line `20`). This function returns a modified filename with all special characters replaced so it can be safely treated like a normal filename. This modified filename is then used to form a file path that is guaranteed to be in the `uploads` subfolder (e.g. line `21`).

- To actually view uploaded photos, two routes are needed:
    - one for generating the HTML document that will display all the photos on a single page.
    - one for accessing each photo's file data.

- For the second route, use `send_from_directory()` (e.g. line `44`) to avoid the same security issues that come from using paths or filenames provided by users.
    - To use `send_from_directory()`, call it with the name of a subfolder that the requested file must be stored in as well as the file's filename. Then return the result of this call directly.

- As an alternative to configuring a new route to access each uploaded photo, the uploads can be saved instead in the `static` subfolder.
    - The existing `'static'` route that Flask provides by default can then be used.
    - This however requires care to not let users overwrite files errorneously.

[Challenge] Can you edit Program 21 to only allow files with extensions GIF, JPG or PNG to be uploaded?

10 `Request.args vs Request.form vs Request.Files`

Below is a summary of how **`request.args`**, **`request.form`** and **`request.files`** are used to access the different kinds of data submitted by the user.

Attribute	**`request.args`**	**`request.form`**	**`request.files`**
Contents	Dictionary of field names and their associated values from query portion of URL	Dictionary of field names and their associated values	Dictionary of file upload names and their associated **`FileStorage`** objects
HTTP Method	Usually **`GET`**, but also works with **`POST`** if URL has query portion	**`POST`** only	**`POST`** only
Typical Use	Reading form data submitted using **`GET`**	Reading form data submitted using **`POST`**	Saving files submitted using **`POST`**
Other			Form must specify **`enctype= "multipart/form-data"`**

11 Summary

You have learned how to use Flask to route HTTP requests, send HTTP responses, use templates and process form data. Combined with the ability to read and write from SQLite databases, you now have the building blocks to create all manners of creative web applications.