Web Applications

Name: _____ ( ) Class: _____ Date: _____

Lesson 4: Using Flask

Instructional Objectives:

By the end of this task, you should be able to:

- 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

---

Part 1: Dynamic Web Pages

Although we can use the web to simply request for and retrieve static HTML and CSS documents, the web becomes much more useful if the web server can customise its response to each HTTP request. For example, the following program uses what we have learnt so far about sockets, HTTP, HTML and CSS to generate a web page with random colours and text every time it is requested.

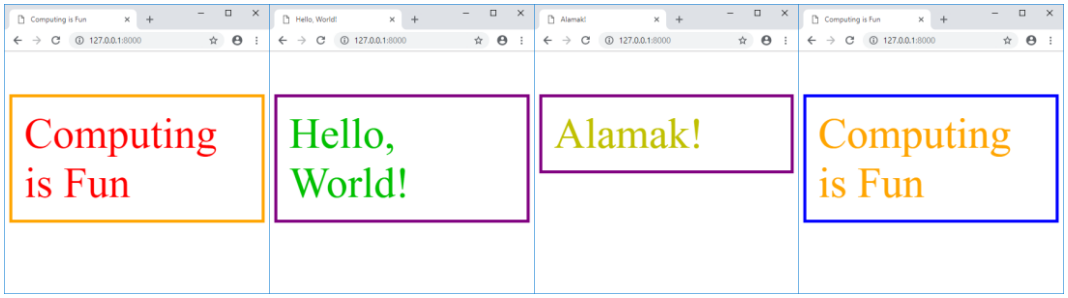
Program 1: `p01_server_without_flask.py`

```
1   import random
2   import socket
3
4   # HTTP uses \r\n to end lines instead of \n.
5   EOL = b'\r\n'
6
7   # List of possible colours.
8   COLOURS = [
9       'red', 'orange', 'blue', 'purple',
10      '#C0C000',  # dark yellow
11      '#00C000'   # dark green
12  ]
13
14  # List of possible messages.
15  MESSAGES = ['Hello, World!', 'Computing is Fun', 'Alamak!']
16
17  # Start listening for web browser requests on port 8000.
```

```
18  listen_socket = socket.socket()
19  listen_socket.bind(('127.0.0.1', 8000))
20  listen_socket.listen()
21
22
23  def handle_request(new_socket):
24      # Keep loading request until end of first line (\r\n).
25      request = b''
26      while EOL not in request:
27          received = new_socket.recv(1024)
28          if received == b'':
29              new_socket.close()
30              return
31          request += received
32
33      # Extract first line and split it into three.
34      # Requested path is always the second part.
35      index = request.index(EOL)
36      first_line = request[:index].decode()
37      path = first_line.split()[1]
38
39      # Start response with standard HTTP status line.
40      response = b'HTTP/1.1 200 OK' + EOL
41
42      # Generate CSS document for response if path is '/css'.
43      # Otherwise, generate HTML document for response.
44      if path == '/css':
45          # Generate CSS document.
46          border_colour = random.choice(COLOURS)
47          text_colour = random.choice(COLOURS)
48          body = 'p {{ border: 5px solid {0}; color: {1};'
49          body += 'font-size: 72px; padding: 20px; }}'
50          body = body.format(border_colour, text_colour)
51          body = body.encode()
52
53          # Append HTTP header fields to response.
54          response += b'Content-Type: text/css' + EOL
55          response += b'Content-Length: '
56          response += str(len(body)).encode() + EOL
57      else:
58          # Generate HTML document.
59          msg = random.choice(MESSAGES)
60          body = '<!DOCTYPE html>\n<html>'
61          body += '<head><title>{0}</title>'
62          body += '<link rel="stylesheet" href="/css"></head>'
63          body += '<body><p>{0}</p></body></html>'
64          body = body.format(msg).encode()
65
66          # Append HTTP header fields to response.
```

```
67          response += b'Content-Type: text/html' + EOL
68          response += b'Content-Length: '
69          response += str(len(body)).encode() + EOL
70
71      # Append empty line to response.
72      response += EOL
73
74      # Append document as message body.
75      response += body
76
77      # Send the response and close the socket.
78      new_socket.sendall(response)
79      new_socket.close()
80
81
82  while True:
83      new_socket, addr = listen_socket.accept()
84
85      handle_request(new_socket)
```

Run the program in the background, then view `http://127.0.0.1:8000/` in a web browser. Reload the browser a few times to see the web page's text and colours change each time:



How does the program work?

1. Lines 17 to 20 start the server by listening on port 8000.

2. When a web browser visits `http://127.0.0.1:8000/`, it sends a HTTP request via the new socket that is returned by `accept()` on line 83. The new socket is then passed to the `handle_request()` function on line 23.

3. Lines 24 to 37 read the HTTP request until a line ending is detected. However, if `recv()` ever returns an empty `bytes` object, the connection has been closed so the program stops processing the request immediately. Otherwise, it extracts the `path` component from the first line of the request.

4. Line 40 starts a new HTTP response with only a status line.

5. If `path` matches `'/css'` exactly, lines 45 to 51 generate a CSS document using random colours and store it in `body`. Lines 53 to 56 then generate the required HTTP header fields and append them to the response.

6. Otherwise, lines 58 to 64 generate a HTML document containing a random phrase and store the HTML document in `body`. Lines 66 to 69 then generate the required HTTP header fields and append them to the response.

7. Regardless of whether a CSS or HTML document was generated, lines 71 to 78 end the header fields with an empty line, append the generated document to the response and send the completed response back to the web browser.

8. Finally, line 79 closes the socket so the server can handle the next HTTP request and so on until Ctrl-C is pressed.

Writing dynamic web pages in this way can be complex and tedious. In particular, `p01_server_without_flask.py` requires the programmer to parse or construct HTTP request/status lines and header fields to follow the HTTP standard. It also requires the programmer to assemble long pieces of HTML and CSS code from Python strings.

Instead of having the programmer spend time on these tedious and error-prone tasks, it is usually better to use a **framework** instead. 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 the programmer only needs to configure or customise for his or her needs.

---

1 The following program runs a web server at `http://127.0.0.1:5000/` that simply repeats the first line of each HTTP request back to the browser.

```
import socket

EOL = b'\r\n'

listen_socket = socket.socket()
listen_socket.bind(('127.0.0.1', 5000))
listen_socket.listen()

def handle_request(new_socket):
    request = b''
    while EOL not in request:
        received = new_socket.recv(1024)
        if received == b'':
            return
        request += received
    index = request.index(EOL)
    first_line = request[:index]
```

```
    response = b'HTTP/1.1 200 OK' + EOL
    response += b'Content-Type: text/plain' + EOL
    response += b'Content-Length: '
    response += str(len(first_line)).encode() + EOL
    response += EOL
    response += first_line

    new_socket.sendall(response)
    new_socket.close()

while True:
    new_socket, addr = listen_socket.accept()

    handle_request(new_socket)
```

Run the program and visit each of the URLs below to tabulate the output that is displayed in the browser. An example has been completed for you.

No.	URL	Output
e.g.	`http://127.0.0.1:5000/hello`	`GET /hello HTTP/1.1`
1	`http://127.0.0.1:5000/`	
2	`http://127.0.0.1:5000/100/200`	
3	`http://127.0.0.1:5000/200/300/`	
4	`http://127.0.0.1:5000//300//400//`	
5	`http://127.0.0.1:5000/?key=value`	
6	`http://127.0.0.1:5000/#fragment`	
7	`http://127.0.0.1:5000/?key#hash`	

Web Applications

Part 2: The Flask Framework

To create dynamic web sites, we can use a web application framework named **Flask**. Unlike the other libraries and modules that we have used so far, Flask is not built into Python and typically needs to be installed separately. However, it should already be installed for you in the computers at your school.

To check if Flask is installed, run the following line of Python (bold text only):

```
>>> import flask
```

If you get an error on your home computer, you may need to install Flask by starting Command Prompt or PowerShell as an administrator and running (bold text only):

```
PS C:\Users\user> pip3 install flask
```

To run the above command successfully, you must have Internet access and Python must be included in your **PATH**, which is the list of directories that Command Prompt or PowerShell automatically searches through when looking for a command. Search the Internet or ask your teacher for help if you are unable to install Flask on your own.

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, like in the following program:

Program 2: `p02_minimal.py`

```
1  import flask
2
3  app = flask.Flask(__name__)
4
5  if __name__ == '__main__':
6      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)
```

Technical Note

The latest version of Flask (1.0) and IDLE have a known incompatibility that causes an `io.UnsupportedOperation: fileno` error to occur when calling `app.run()` on Windows. To work around this issue, use Flask 0.12 instead. To replace your current version of Flask, start Command Prompt or PowerShell as an administrator and run (bold text only):

```
PS C:\Users\user> pip3 install flask==0.12
```

An appropriate version of Flask should already be installed on your school's computers.

To stop the Flask server, press Ctrl-C in the IDLE's shell window or restart the shell by running Ctrl-F6.

Part 3: HTTP Requests and Routing

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

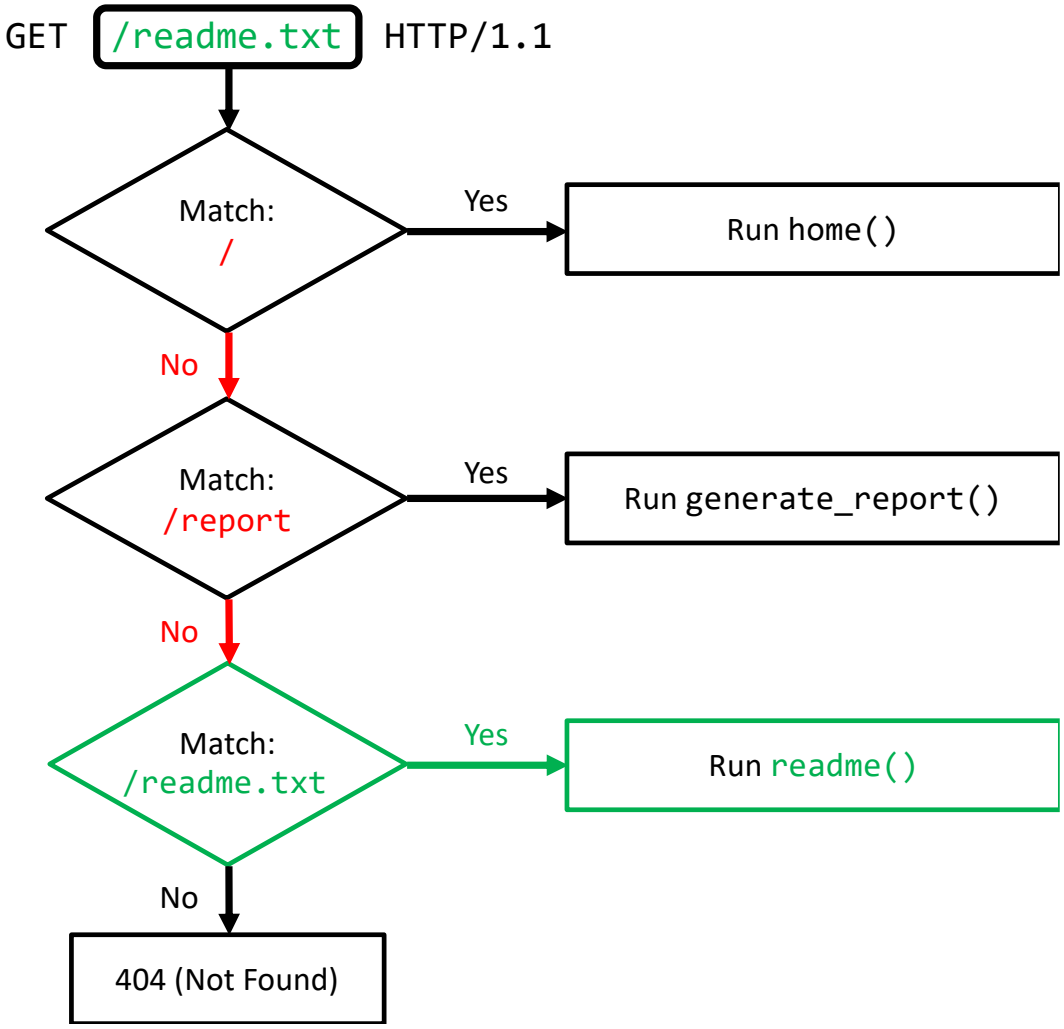
For example, suppose a HTTP server is running on 127.0.0.1 and listening on port 5000. When we visit `http://127.0.0.1:5000/readme.txt`, the browser sends a HTTP request with a first line that contains the path `/readme.txt`, like so:

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

Note that even though HTTP paths such as `/readme.txt` look similar to file paths that we may use to open and save files in Windows or macOS, to a web server they are just strings and do NOT need to refer to real files or folders that are stored on the computer. For instance, in this example 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 an HTML error page when we visit a URL that the server does not recognise.

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. Otherwise, no mapping is found and a 404 (Not Found) status code is produced instead.

Web Applications



This process is called **routing** and it is similar to how phone operators forward calls to different departments based on each caller's needs. Similarly, 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**. In Python, decorators let us alter the behaviour of a function without modifying its source code. We do this by adding **decorations** immediately before the function's definition. Each decoration starts with an at-sign or "pie" 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`, as demonstrated on lines 5, 9 and 13 of the following example:

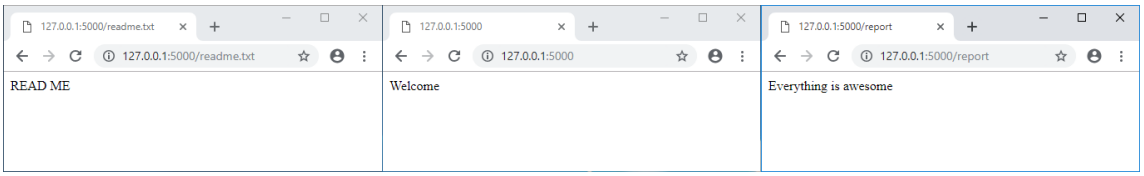
Program 3: `p03_simple_routes.py`

```
1
2
3
4
5
```

```
import flask

app = flask.Flask(__name__)

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

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

Run `03_simple_route.py` and visit `http://127.0.0.1:5000/readme.txt` to see the message "READ ME" returned by the `readme()` function. This works because the path `/readme.txt` is mapped to `readme()` by the decoration on line 13. Similarly, if we visit `http://127.0.0.1:5000/`, we see the message "Welcome" and if we visit `http://127.0.0.1:5000/report`, we see the message "Everything is awesome". However, visiting any other URL that starts with `http://127.0.0.1:5000/` (For example `http://127.0.0.1:5000/nothing`) results in a 404 (Not Found) error as no other paths have been mapped.



The next example uses multiple decorators to declare a variety of routes:

Program 4: `p04_complex_routes.py`

```
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  @app.route('/one/')
22  @app.route('/one/two/')
23  @app.route('/three/two/one')
24  def multiple():
25      return 'Routed to multiple()'
26
27  @app.route('/string/<s>/')
28  def string_variable(s):
29      return 'Routed to string_variable(), s = {}'.format(s)
30
31  @app.route('/integer/<int:i>/')
32  def integer_variable(i):
33      return 'Routed to integer_variable(), i = {}'.format(i)
34
35  @app.route('/post_only/', methods=['POST'])
36  def post_only():
37      return 'Routed to post_only()'
38
39  if __name__ == '__main__':
40      app.run()
```

Fixed Routes

Run `04_complex_routes.py` and visit `http://127.0.0.1:5000/` with a web browser. You should see a simple page with the message "Routed to index()" indicating that your request was handled by the `index()` function defined on lines 6 to 7. This is because the path component of `http://127.0.0.1:5000/` is just a slash character `/`, which matches the route specified on line 5 for the `index()` function.

Now visit `http://127.0.0.1:5000/css` and you should see the message "Routed to css()" instead. In this case, the path component of `http://127.0.0.1:5000/css` is just `/css`, which matches the route specified on line 9. Hence, the request is handed over to the corresponding `css()` function defined on lines 10 and 11.

Note that routes are case-sensitive, so visiting `http://127.0.0.1:5000/CSS` instead of `http://127.0.0.1:5000/css` would lead to a 404 Not Found error.

Trailing Slashes

In general, a request's path and a route's path must match **exactly** for the route to be chosen. For instance, the route for `/no_slash` on line 13 is matched only by `http://127.0.0.1:5000/no_slash` (without a trailing slash). If we try visiting `http://127.0.0.1:5000/no_slash/` (with a trailing slash), we will get a 404 (Not Found) error instead.

However, if 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. For instance, 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.

Multiple Decorators

If needed, multiple paths can also be routed to the same function. For instance, the three decorations on lines 21 to 23 associate all of the following URLs with the `multiple()` function defined on lines 24 and 25:

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

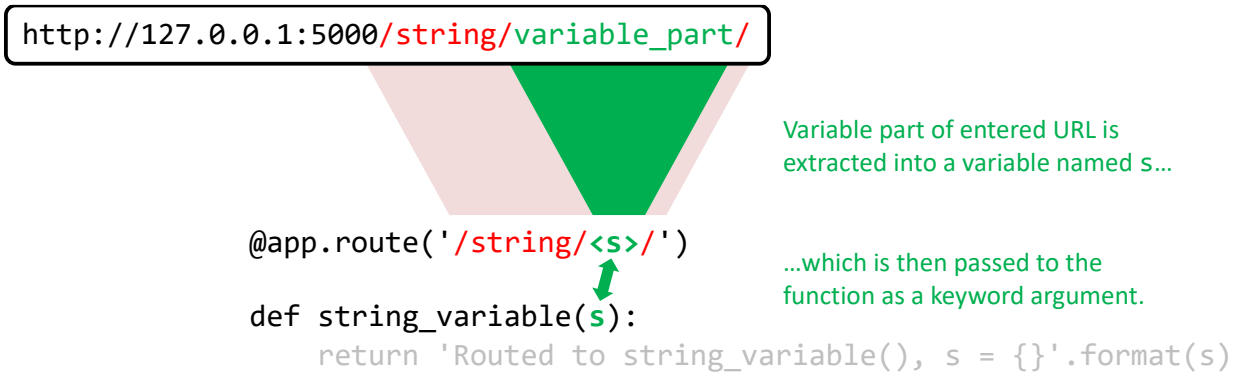
Note that trailing slashes are treated in exactly the same way as described previously, so `http://127.0.0.1:5000/three/two/one/` will result in a 404 (Not Found) error since there is no route that matches `/three/two/one/` (with a trailing slash) exactly.

Variable Routes

Flask routes can also have **variable** parts. Each variable part in the route's path has a name surrounded by angled brackets < and >. 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 27 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.

Web Applications



The following table 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 Variable 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	

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 31 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 1 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.

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	

Note that as the `int` converter only accepts digits, negative integers that start with a minus sign CANNOT be matched by `<int:i>`.

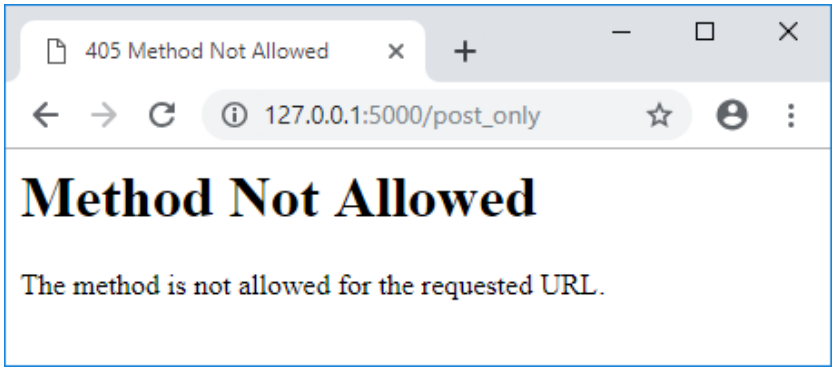
2 Complete the following program so it greets the user when the site is visited with the user's name in the path (as long as the name does not include a slash):

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


_____________________________
def home(_____________________):
    return 'Hello, {}!'.format(_______________)

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

Routing by HTTP Methods

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 the server's data permanently (e.g., deletion) should not be accessible using `GET`. In other words, a user should not be able to add, delete or update data on the server without submitting a form and sending a `POST` request instead.

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, line 35 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 of a web browser, we get a 405 (Method Not Allowed) error instead.



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

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()`.

The following example 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.

Program 5: `p05_url_lookup.py`

```
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()
```

The expected output for `05_url_lookup.py` 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.

---

3 The following program runs a web server at `http://127.0.0.1:5000/`.

```
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()
```

With this program running in the background, 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.

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/	

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

Web Applications

Part 4: HTTP Responses and Status Codes

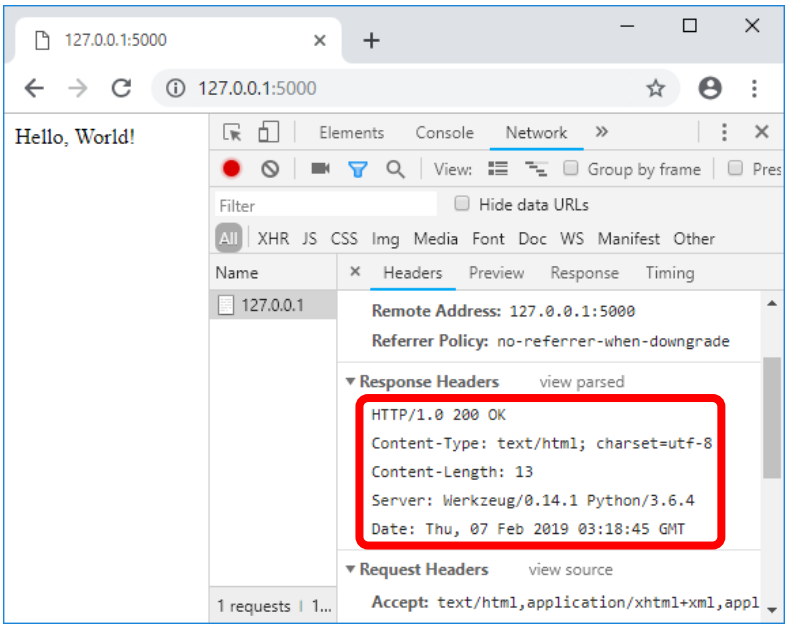
So far, our functions have been returning short strings that appear to display without problem in the web browser. However, Flask has actually been prepending various headers behind the scenes to produce valid HTTP responses.

Run the following program:

Program 6: `p06_hello_world.py`

```
1   import flask
2
3   app = flask.Flask(__name__)
4
5   @app.route('/')
6   def index():
7       return 'Hello, World!'
8
9   if __name__ == '__main__':
10      app.run()
```

With Google Chrome's Developer Tools open, 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.



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

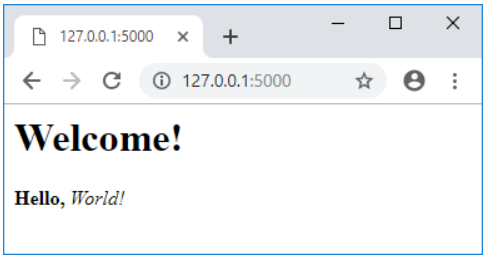
```
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 13
```

```
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>`. For instance, the following program demonstrates that HTML tags can be used in our return value:

Program 7: `p07_returning_html.py`

```
1   import flask
2
3   app = flask.Flask(__name__)
4
5   @app.route('/')
6   def index():
7       return '<h1>Welcome!</h1> <b>Hello,</b> <i>World!</i>'
8
9   if __name__ == '__main__':
10      app.run()
```



However, be aware that is non-standard behaviour and our functions that are mapped to routes should really return full HTML documents.

Changing the Status Code

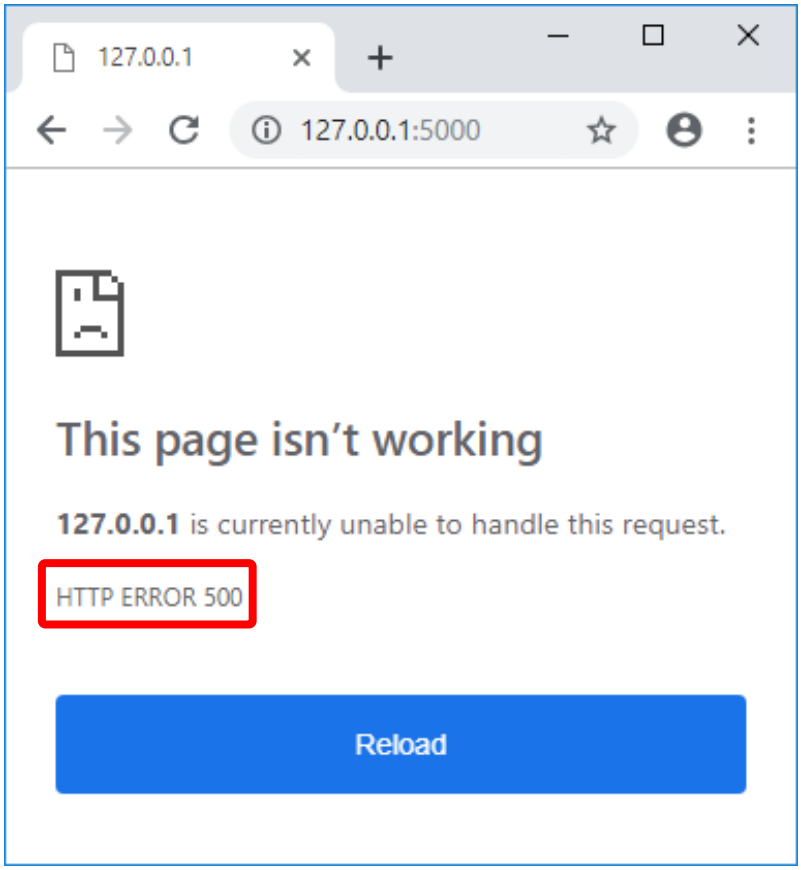
By default, Flask also assumes that our responses have a HTTP status code of 200 (OK). This is usually what we want, but if needed we can override this by returning a tuple instead of just a string. The replacement HTTP status code should be provided as the second item of the tuple, as demonstrated on line 7 of the following example:

Program 8: `p08_returning_status_500.py`

```
1   import flask
2
3   app = flask.Flask(__name__)
```

```
4
5   @app.route('/')
6   def index():
7       return ('', 500)
8
9   if __name__ == '__main__':
10      app.run()
```

Recall that a HTTP status code of 500 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 an HTTP 500 (Internal Server Error) status code, as expected:



Changing the Response Headers

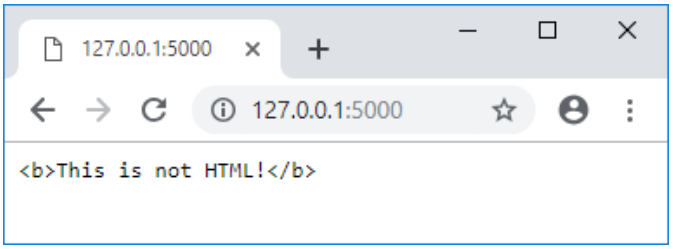
We can also add additional response headers by putting them into a Python dictionary and returning this dictionary as the third item of our return tuple. For instance, if we want the web browser to treat our response as plain text instead of HTML, we can replace the Content-Type header value with "`text/plain`"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

Program 9: `p09_returning_plain_text.py`

```
1   import flask
2
3   app = flask.Flask(__name__)
```

```
4
5   @app.route('/')
6   def index():
7       headers = {'Content-Type': 'text/plain'}
8       return ('<b>This is not HTML!</b>', 200, headers)
9
10  if __name__ == '__main__':
11      app.run()
```

Running this program and visiting `http://127.0.0.1:5000/` demonstrates that the string we return is no longer treated as HTML by the browser:



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 that as the first argument. Then, use the response generated by `redirect()` as the return value of the function, as in the following example:

Program 10: `p10_redirect_example.py`

```
1   import flask
2   from flask import redirect
3
4   app = flask.Flask(__name__)
5
6   @app.route('/')
7   def index():
8       return redirect('http://example.com')
9
10  if __name__ == '__main__':
11      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:

Program 11: p11_redirect_using_url_for.py

```
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.

Part 5: Templates and Rendering

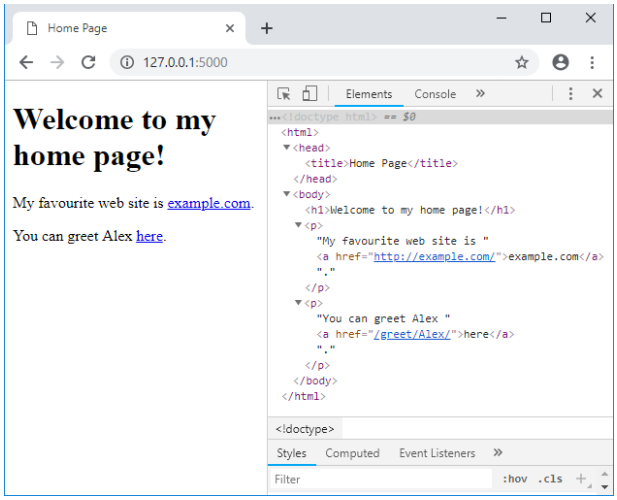
Instead of returning short HTML snippets or redirecting users, the following program illustrates how we can return full HTML documents complete with headings and hyperlinks in our Flask application:

Program 12: p12_html_response_without_templates.py

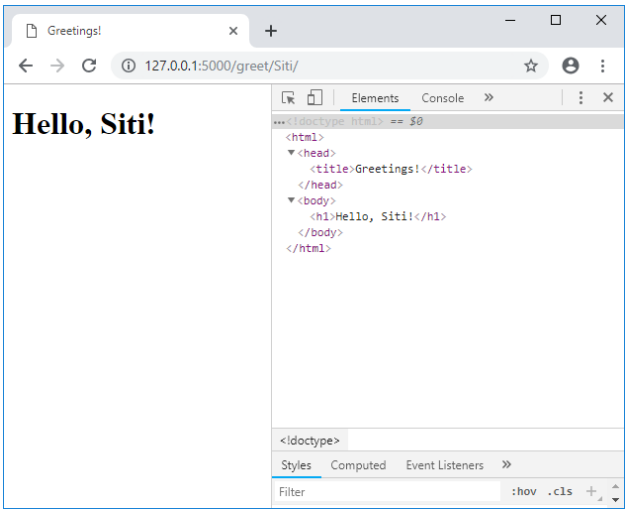
```
1   import flask
2
3   app = flask.Flask(__name__)
4
5   @app.route('/')
6   def home():
7       html = '<!DOCTYPE html>\n<html>'
8       html += '<head><title>Home Page</title></head>'
9       html += '<body><h1>Welcome to my home page!</h1>'
10      html += '<p>My favourite web site is '
11      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 += '<body><h1>Hello, {}!</h1>'.format(name)
23      html += '</body></html>'
24      return html
25
26  if __name__ == '__main__':
27      app.run()
```

Run the above program and visit http://127.0.0.1:5000/. Notice that a full HTML page appears, complete with a page title, headings and hyperlinks. Use the View Source feature by pressing Ctrl-U to verify that the HTML for the page comes directly from the `home()` function of the above Python program.



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



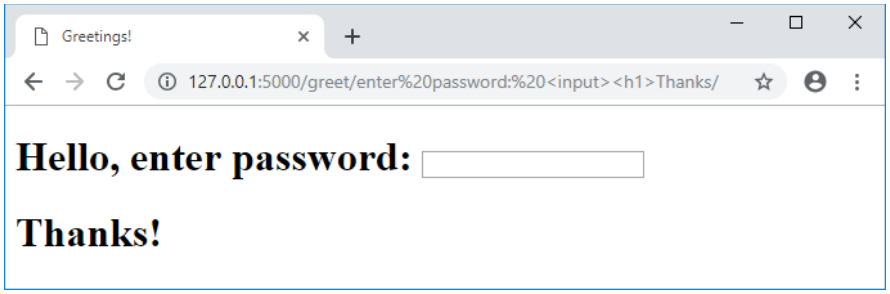
Why Use Templates?

Generating dynamic HTML content in this way is powerful but also 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, if we follow the following link:

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

We will get what appears to be a legitimate form asking for the user's password. 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.



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. Whenever 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 similar to 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**.

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`:

Template 1: `templates/home.html`

```
1  <!DOCTYPE html>
2  <html>
3  <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>
```

Also save the following file in the `templates` folder as `greet.html`:

Template 2: `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 Jinja2 and Python in the following sections.

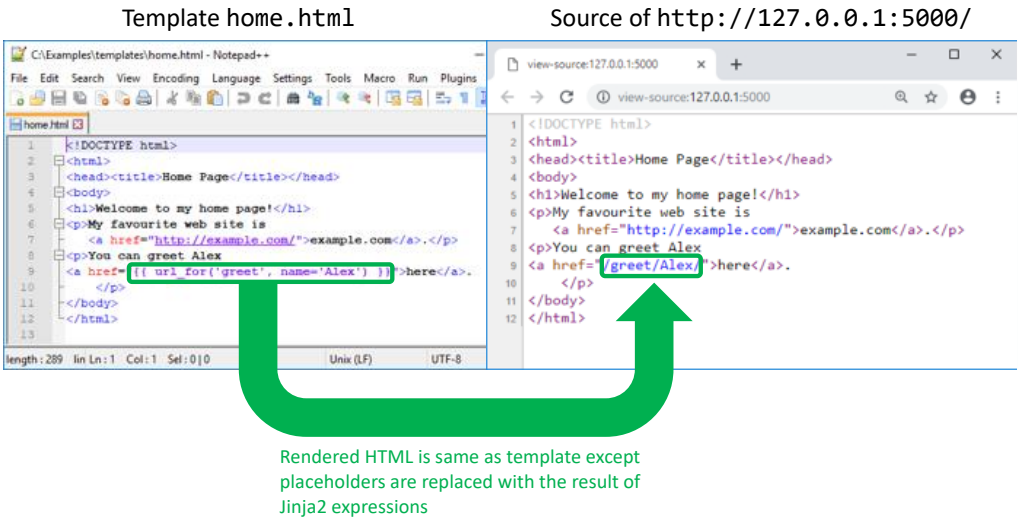
Recall that the process of filling in the placeholders of a template is called **rendering**. The task of rendering a template is typically performed by a function named `render_template()` that can be imported from the `flask` module, like so:

Program 13: `p13_html_response_with_templates.py`

```
import flask
from flask import render_template

app = flask.Flask(__name__)

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

@app.route('/greet/<name>/')
```

```
11  def greet(name):
12      return render_template('greet.html', visitor=name)
13
14  if __name__ == '__main__':
15      app.run()
```

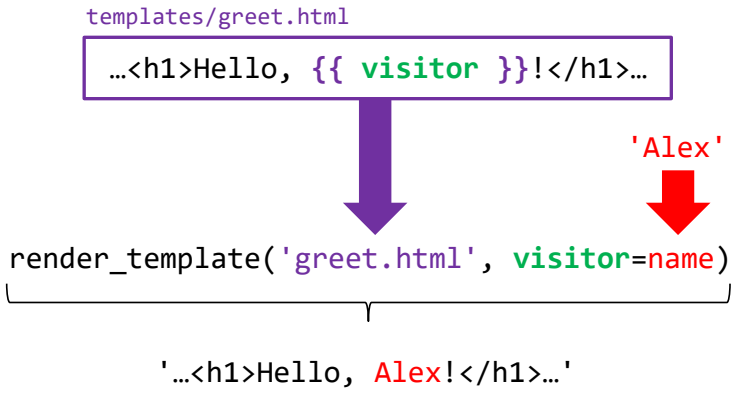
Run this program and visit `http://127.0.0.1:5000/`. View Source on the page by pressing Ctrl-U to verify that the template is indeed rendered and no placeholders are present. Click the link to visit `http://127.0.0.1:5000/greet/Alex/` and verify that there are no placeholders in its HTML source as well.



Passing Values to Templates

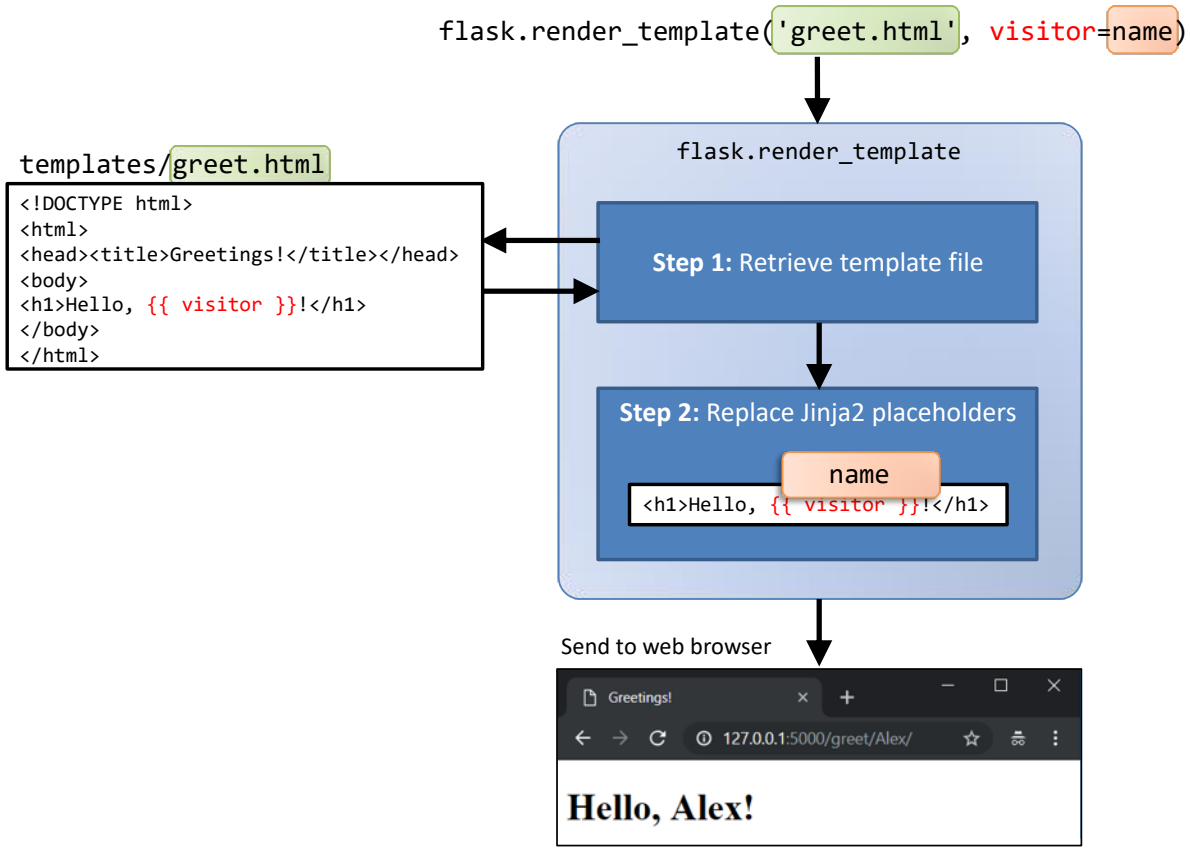
The `render_template()` function accepts the name of a template in the `templates` subfolder as its first argument, followed by 0 or more keyword arguments assigning values to Jinja2 variables that may be used by the template.

For instance, line 12 of `p13_html_example_with_templates.py` 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 `greet.html` where the Jinja2 placeholder `{{ visitor }}` is replaced by this value.



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:



4 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 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

Complete the following program accordingly:

app.py

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

_____________________________
def view_A():
    return flask.render_template('A.html')


_____________________________
def view_B():
    return flask.render_template('B.html')


_____________________________
def view_C():
    return flask.render_template('C.html')


_____________________________
def view_D():
    return flask.render_template('D.html')

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

5 Generalise the previous program and templates so that the application only needs one template instead of four templates:

templates/template.html

```
<!doctype html>
<html>
<head><title>_____________</title></head>
<body>_________________</body>
</html>
```

app.py

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


_____________________________
def view(_____________________):
    return flask.render_template('template.html',
        _______________)

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

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

Go further! Edit the program above so only the four cases given in Question 4 are allowed and return error code 500 for other cases.

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 lines 8 and 9 of `home.html`:

```
<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>.
```

Hardcoded links are shorter but 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 having to change every template that links to it.

6 Enter the following program:

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()
```

Use the `url_for()` function to complete the following template so that each link will display the respective message when clicked:

templates/links.html

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

    <p><a href="_____________________">Hello, World!</a></p>

    <p><a href="_____________________">year is 2017</a></p>

</body>
</html>
```

The safe Filter

With `p13_html_response_with_templates.py` running, visit the URL that previously allowed us to inject HTML into the greeting page:

```
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 less-than < and greater-than > signs have been escaped with the appropriate HTML entities:

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, as in the example below:

Template 3: `templates/custom.html`

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

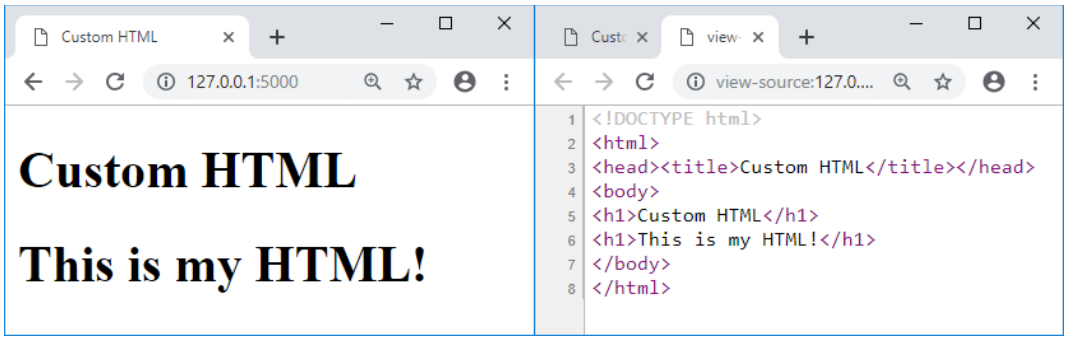
Save this `custom.html` in the templates subfolder and run the following program from the parent folder:

Program 14: `p14_raw_html_with_safe_filter.py`

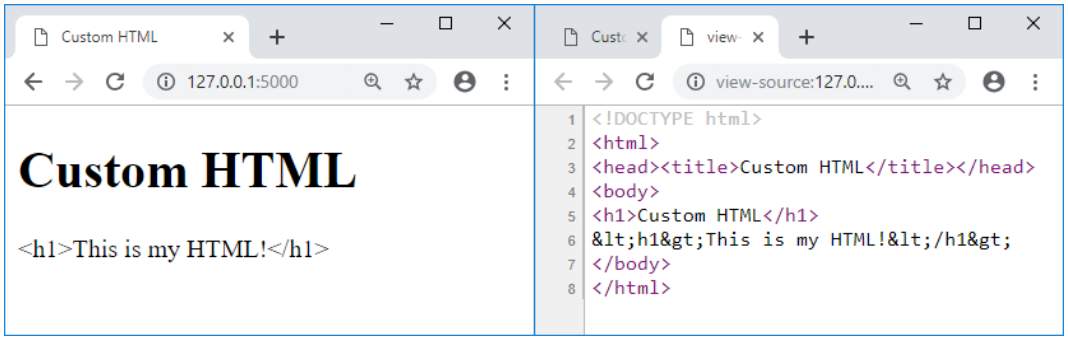
```
1  import flask
2  from flask import render_template
3
4  app = flask.Flask(__name__)
5
6  @app.route('/')
7  def home():
8      return render_template('custom.html',
```

```
9           my_html='<h1>This is my HTML!</h1>')
10
11  if __name__ == '__main__':
12      app.run()
```

Visit `http://127.0.0.1:5000/` and you should see that the string passed to the template as `my_html` is rendered as raw 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:



Technical Note

You must restart your program for any changes (such as removing the `safe` filter) to take effect. If your program appears to stop reflecting changes that you have made, restart the program, then hold down the Ctrl key while refreshing so your browser bypasses its cache and performs a fresh HTTP request.

Alternatively, you may wish to run Flask in debug mode by replacing `app.run()` with `app.run(debug=True)`. This lets Flask reload itself whenever it detects any file changes so manual restarts are no longer needed. However, running Flask in debug mode has several incompatibilities with IDLE:

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.

These issues only occur when using IDLE. If you still wish to use debug mode, it is highly recommended that you run your program from the Command Prompt or PowerShell instead or use an alternative IDE.

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:

Template 4: `templates/length.html`

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

Save the above `length.html` in the `templates` subfolder and run the following program from the parent folder:

Program 15: `p15_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`.

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 is 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. For our needs, we will cover two types of Jinja2 statements: `if` statements and `for-in` statements.

`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:

Template 5: `templates/results.html`

```
1   <!DOCTYPE html>
2   <html>
3   <head><title>Results</title></head>
4   <body>
5   <h1>Results</h1>
6
7   {% if greet %}
8       <p>Hello, {{ name }}.</p>
9   {% endif  %}
```

```
10
11  {% if show_score  %}
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
19  </body>
20  </html>
```

Run the following program and visit `http://127.0.0.1:5000/` to see the template rendered for a score of 72:

Program 16: `p16_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()
```

To explore further, 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/` again to see how the template is rendered differently. In each case, verify that the template's behaviour matches up with what you would expect.

for-in Statement

Similar to the `for-in` 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, like the `if` statement 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:

Template 6: templates/table.html

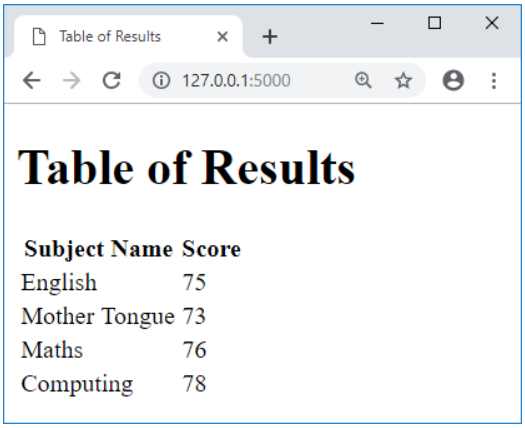
```
<!DOCTYPE html>
<html>
<head><title>Table of Results</title></head>
<body>
<h1>Table of Results</h1>

<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>
```

For this example, `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:

Program 17: p17_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()
```

Run the following program and visit `http://127.0.0.1:5000/` to see how the hardcoded data in the dictionary is rendered as a HTML table:



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

Part 6: Static Files

Recall that requested paths are treated as strings for route matching by Flask and do not usually refer to the location of real files or folders that are stored on the server. This lets us return 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 in order to display properly. However, unlike the main content, these additional resources do not change often, so instead of running a Python function to generate the response, Flask lets us create a subfolder named `static` (in the same location as the `templates` subfolder) 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.

For example, create a subfolder named `static` In the folder where your Flask programs are stored, then save the following file in this folder as `styles.css`:

Style Sheet 1: `static/styles.css`

```
1  body {
2      background: yellow;
3  }
4
5  h1 {
6      border-bottom: 1px solid red;
7      color: red;
8  }
```

Now, let's create a template that makes use of this style sheet. For simplicity, we can hardcode the URL of the style sheet `/static/styles.css`. However, it is preferred to use `url_for()` in case the path for static files changes in the future:

Template 7: `templates/stylish.html`

```
1   <!DOCTYPE html>
2   <html>
3   <head>
4       <title>Stylish Page</title>
5       <link
6           rel="stylesheet"
7           href="{{ url_for('static', filename='styles.css') }}">
8       <!--
9           The above can also be written as:
10          <link rel="stylesheet" href="/static/styles.css">
11      -->
12  </head>
13  <body>
14  <h1>Stylish Page</h1>
15  <p>Look at how stylish this page is!</p>
16  </body>
17  </html>
```

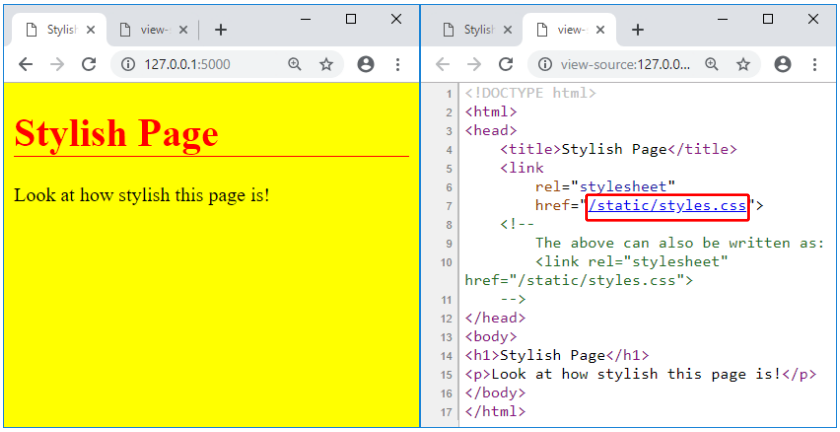
Note how the path of the static file within the `static` subfolder is passed to `url_for()` as a keyword argument named `filename` on line 7.

Finally, let us write a simple Python program to render this template:

Program 18: `p18_static_style_sheet.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('stylish.html')
9
10  if __name__ == '__main__':
11      app.run()
```

Visit `http://127.0.0.1:5000/` with this program running and you should see the styled page. If you use the View Source feature by pressing Ctrl-U, you can also verify that the style sheet URL placeholder is replaced with a path starting with `/static/`:



Part 7: Processing Form Data

So far, we have been able to get input from the user by extracting variables from request URLs. However, it is more common for users to provide input to a web application by filling in a HTML form and submitting it. When a HTML form is submitted, the browser collects all the input as key-value pairs and encodes it into a single string. This string is then sent to the server using a HTTP request. Depending on how the HTML form is configured, either a GET request or POST request is made.

GET Requests

The default behaviour for HTML forms is to submit a GET request. When this method is used, 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. Thankfully, if a query string is present, Flask does this parsing for us and lets us access it as a dictionary from a `request` object that can be imported from the `flask` module.

To illustrate this, create the following two templates in your `templates` subfolder. The first template is used to display the HTML form:

Template 8: `templates/form_with_get.html`

```
1   <!DOCTYPE html>
2   <html>
3   <head><title>GET Form</title></head>
4   <body>
5   <form action="{{ url_for('process_with_get') }}">
6       <p>Input s: <input name="s"></p>
7       <p><input type="submit"></p>
8   </form>
9   </body>
10  </html>
```

Notice that on line 5 we use `url_for()` with a function name of `'process_with_get'` to generate the path that the form data will be submitted to. We will write the code for `process_with_get()` later.

The second template is used to display the results of processing. In this case, let us display the number of vowels and number of words in the submitted name. Save the following file in the same folder as `analysis_results.html`:

Template 9: `templates/analysis_results.html`

```
1   <!DOCTYPE html>
2   <html>
3   <head><title>String Analysis</title></head>
4   <body>
5   <p>You entered s: {{ s }}</p>
6   <p>This string has
7       {{ num_vowels }} vowels(s) and
8       {{ num_words }} word(s).
9   </body>
10  </html>
```

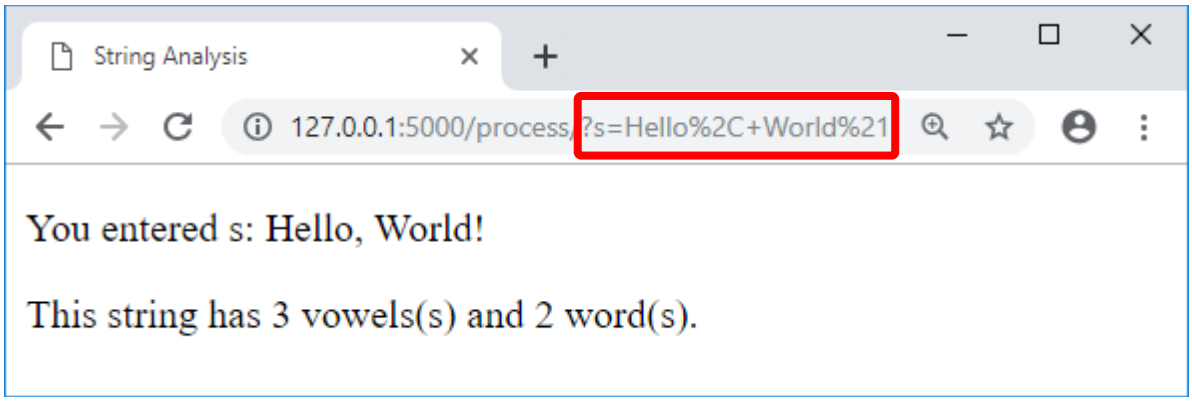
Next, let us write a Flask application with two routes: the root `/` route simply renders the form while the `/process/` route is associated with a Python function named `process_with_get()` that analyses `s` and renders the result:

Program 19: `p19_analysis_with_get.py`

```
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('/')
9   def form():
10      return render_template('form_with_get.html')
11
12  @app.route('/process/')
13  def process_with_get():
14      if 's' in request.args:
15          s = request.args['s']
16          lower_s = s.lower()
17          num_vowels = 0
18          for vowel in VOWELS:
19              num_vowels += lower_s.count(vowel)
20          num_words = len(s.split())
```

```
21          return render_template('analysis_results.html',
22              s=s, num_vowels=num_vowels, num_words=num_words)
23      return 'No form data found!'
24
25  if __name__ == '__main__':
26      app.run()
```

Notice that the `request` object is imported from the `flask` module on line 2 and that the submitted value of `s` is accessed via the `request.args` dictionary on line 15. Visit `http://127.0.0.1:5000/` and fill in the form. When you submit the form, notice that the value of `s` you entered is encoded in the resulting URL.



Also note that it is possible to manually visit `http://127.0.0.1:5000/process/` without filling in the form. If you do so, the `request.args` dictionary will not include a value for 's' and the error message on line 23 will be returned instead.

POST Requests

Submitting form data using a GET request has several disadvantages:

1. Firstly, the submitted form data is recorded in the resulting URL, which lets anyone view your browser history to obtain the form data that was sent.

2. Secondly, GET requests are not supposed to make changes to the server's data. If we use data submitted with a GET request to add, delete or update data from a database, we are not following the HTTP standard.

3. Finally, some browsers and server software limit the length of URLs, so there is a risk that overly long form data submitted using GET may get truncated.

To overcome these disadvantages, we can configure HTML forms to submit the data using POST requests instead. We do this by setting the `method` attribute of the `<form>` tag to `"post"` or `"POST"`. Using a different HTTP method also lets us distinguish between requests from users clicking a link or entering the URL in an address bar versus requests from users submitting a form. We can take advantage of this to use the same URL for both *displaying* the form as well as *processing* the form.

To illustrate this, create the following template in your `templates` subfolder:

Template 10: `templates/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 besides adding the `method` attribute on line 5, we have also removed the `action` attribute so the form is submitted to the same URL used to generate the form.

Next, let us adapt `19_analysis_with_get.py` to combine both routes and distinguish between the two situations by looking at the HTTP method used. The HTTP method can be easily determined via the `request.method` string:

Program 20: `p20_analysis_with_post.py`

```
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, num_vowels=num_vowels, num_words=num_words)
21      return 'No form data found!'
22
23  if __name__ == '__main__':
24      app.run()
```

Here, `index()` checks if the request method is `'GET'`, and if so, simply renders the form and exits. Otherwise, the request method is assumed to be `'POST'` and the submitted form data is processed instead.

Notice that unlike GET requests where submitted form data is available from the `request.args` dictionary, for POST the submitted form data is placed in a separate `request.form` dictionary instead. For instance, the submitted value of `s` is accessed via the `request.form` dictionary on line 13.

Also note that the route decorator on line 8 must explicitly include `'POST'` in its list of allowed methods as Flask routes are only accessible via `'GET'` by default.

Now, with the program running, visit `http://127.0.0.1:5000/` and fill in the form. When you submit the form, notice that the value of `s` you entered in NOT visible in the resulting URL and that the URL displayed in the address bar remains the same.

Notice on line 12 that we still check if the `request.form` dictionary has a value for `'s'` before proceeding. Even though it is relatively hard for an average user to send phony POST requests, it is not technically difficult and our program must be prepared to handle POST requests with incomplete or invalid form data. In this case, if there is no value for `'s'` in `request.form`, we return an error message on line 21 instead.

7 Enter the following and save it as a template named `form.html`:

`templates/form.html`

```
<!doctype html>
<html>
<head><title>Greeting Form</title></head>
<body>
    <form method="POST">
        <input type="text" name="name">
        <input type="text" name="address">
        <input type="submit">
    </form>
</body>
</html>
```

Complete the following program and template so 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:

`templates/results.html`

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

```
    Hello _____________, your address is: _______________
</body>
</html>
```

app.py

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

# Use the space below









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

Part 8: Handling File and Image Uploads

Recall that an `<input>` tag can have its type attribute set to "`file`". This allows the user to upload files for submission with the form. However, 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.

For example, the following template presents a form for uploading photos as well as a link for seeing all the photos that have been uploaded:

Template 11: `templates/form_with_file_upload.html`

```
1   <!DOCTYPE html>
2   <html>
3   <head><title>Photo Upload</title></head>
4   <body>
5   <form method="post" enctype="multipart/form-data">
6       <p>Photo: <input name="photo" type="file"></p>
7       <p><input type="submit"></p>
8   </form>
9   <p><a href="{{ url_for('view') }}">View photos</a></p>
10  </body>
11  </html>
```

The template for viewing the uploaded photos is provided below:

Template 12: templates/view_file_uploads.html

```
1   <!DOCTYPE html>
2   <html>
3   <head><title>View Photos</title></head>
4   <body>
5   {% for photo in photos %}
6       <img src="{{ url_for('get_file', filename=photo) }}"
7            alt="{{ photo }}">
8   {% endfor %}
9   <p><a href="{{ url_for('home') }}">Home</a></p>
10  </body>
11  </html>
```

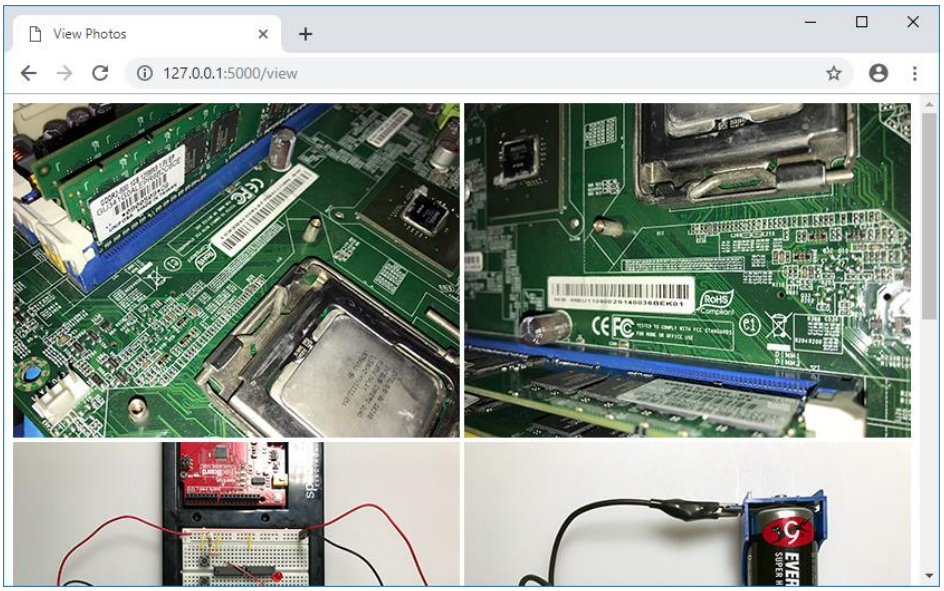
For the main Flask application, we create three routes: one for uploading photos, one for seeing all the uploaded photos and one for retrieving the image data of an uploaded photo given its filename. We also initialize and use a simple SQLite database to keep track of the photos uploaded:

Program 21: p21_file_uploads.py

```
1   import flask, os, sqlite3
2   from flask import render_template, request
3   from flask import send_from_directory
4   from werkzeug.utils import secure_filename
5
6   if not os.path.isfile('db.sqlite3'):
7       db = sqlite3.connect('db.sqlite3')
8       db.execute('CREATE TABLE photos(photo TEXT)')
9       db.commit()
10      db.close()
11
12  app = flask.Flask(__name__)
13
14  @app.route('/', methods=['GET', 'POST'])
15  def home():
16      if request.method == 'POST' and \
17          request.files and 'photo' in request.files:
18          # Save file
19          photo = request.files['photo']
20          filename = secure_filename(photo.filename)
21          path = os.path.join('uploads', filename)
22          photo.save(path)
23          # Add filename to database
24          db = sqlite3.connect('db.sqlite3')
25          db.execute('INSERT INTO photos(photo) VALUES(?)',
```

```
26              (filename,))
27          db.commit()
28          db.close()
29      return render_template('form_with_file_upload.html')
30
31  @app.route('/view')
32  def view():
33      db = sqlite3.connect('db.sqlite3')
34      cur = db.execute('SELECT photo FROM photos')
35      photos = []
36      for row in cur:
37          photos.append(row[0])
38      db.close()
39      return render_template('view_file_uploads.html',
40          photos=photos)
41
42  @app.route('/photos/<filename>')
43  def get_file(filename):
44      return send_from_directory('uploads', filename)
45
46  if __name__ == '__main__':
47      app.run()
```

Before running the program, create an empty `uploads` subfolder 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 you are done, click on the "View" link to see the uploaded photos.



Go further! Try to upload non-photo files such as PDF files. The file will still be uploaded but will not be displayed properly. Can you edit Program 21 to only allow files with extensions GIF, JPG or PNG to be uploaded?

Note that 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.

In general, we should be careful whenever we 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, on line 20 we pass the filename through the `secure_filename()` function provided in the `werkzeug.utils` module first. This function returns a modified filename with all special characters replaced so it can be safely treated like a normal filename. We then use this filename on line 21 to form a file path that is guaranteed to be in our `uploads` subfolder.

To actually view uploaded photos, we need two routes: one for generating the HTML document that will display all the photos on a single page and another for accessing each photo's file data. For the second route, we use `send_from_directory()` on line 44 to avoid the same security issues that come from using paths or filenames provided by users. To use `send_from_directory()`, we call it with the name of a subfolder that the requested file must be stored in as well as the file's filename. We then return the result of this call directly.

As an alternative to configuring a new route to access each uploaded photo, we can instead save our uploads in the `static` subfolder and make use of the existing `'static'` route that Flask provides by default. However, this requires us to be careful not to let users overwrite files that they are not supposed to.

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"`

Part 9: Conclusion

In this practical task, 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.

As a conclusion, look back at `p01_server_without_flask.py` from the beginning of this practical task. The following program demonstrates how the same application that displays random colours and text can be written using the Flask framework that you are now familiar with:

Program 22: `p22_server_with_flask.py`

```
1   import random
2   import flask
3
4   app = flask.Flask(__name__)
5
6   # List of possible colours.
7   COLOURS = [
8       'red', 'orange', 'blue', 'purple',
9       '#C0C000',  # dark yellow
10      '#00C000'   # dark green
11  ]
12
13  # List of possible messages.
14  MESSAGES = ['Hello, World!', 'Computing is Fun', 'Interesting']
15
16  @app.route('/css')
17  def css():
18      border_colour = random.choice(COLOURS)
19      text_colour = random.choice(COLOURS)
20      return (
21          flask.render_template('example.css',
22              border_colour=border_colour,
23              text_colour=text_colour),
24          { 'Content-Type': 'text/css' }
25      )
26
27  @app.route('/')
28  def html():
29      msg = random.choice(MESSAGES)
30      return flask.render_template('example.html', msg=msg)
31
32  if __name__ == '__main__':
33      app.run()
```

This program also requires the following templates:

Template 13: templates/example.html

```
1   <!DOCTYPE html>
2   <html>
3
4   <head>
5       <title>{{ msg }}</title>
6       <link rel="stylesheet" href="{{ url_for('css') }}">
7   </head>
8
9   <body>
10      <p>{{ msg }}</p>
11  </body>
12
13  </html>
```

Template 14: templates/example.css

```
1   p {
2       border: 5px solid {{ border_colour }};
3       color: {{ text_colour }};
4       font-size: 72px;
5       padding: 20px;
6   }
```

Compare p01_server_without_flask.py to p22_server_with_flask.py. Which do you think is easier to write and maintain? Why do you think so?

While there are many other alternative approaches to developing web applications, we hope that learning Flask provides you with a firm foundation for the underlying principles and makes further exploration of the topic easier in the future.

Web Applications

`flask` Module Summary

Name	Description
`Flask(`*`name`*`)`	Creates a Flask application object
`redirect(`*`path`*`)`	Redirects user to the given path when used as a return value from a decorated function
`render_template(`*`filename, var=value`*`)`	Renders Jinja2 template with the given filename using the given variable values and returns the rendered response as a `str`
`request`	Accesses current `Request` object
`send_from_directory(`*`folder, filename`*`)`	Sends file from given directory (first argument) with given filename (second argument) when used as a return value from a decorated function
`url_for(`*`name, var=value`*`)`	Returns the path that is mapped to the given function name and given variable values

`werkzeug` Module Summary

Name	Description
`secure_filename(`*`filename`*`)`	Replaces all characters that have special meanings (e.g., path separators) in the given `str` with underscores

`Flask` Class Summary

Name	Description
`route(`*`path, methods=["GET"]`*`)`	Maps the given path to the decorated function but limits access to given list of HTTP methods; the HTTP methods are `"GET"` and `"POST"`
`run()`	Runs the Flask application with debugging disabled
`run(debug=True)`	Runs the Flask application with debugging enabled

`Request` Class Summary

Name	Description
`args`	Returns a dictionary of field names and their associated values from query portion of URL
`files`	Returns a dictionary of file upload names and their associated `FileStorage` objects
`form`	Returns a dictionary of field names and their associated values
`method`	Returns either `"GET"` or `"POST"`

`FileStorage` Class Summary

Name	Description
`filename`	Returns the name of the uploaded file
`save(`*`path`*`)`	Saves the uploaded file to the given path