Temasek Junior College

2024 JC2 H2 Computing

Web Applications 6 – The Flask Framework and Routing Basics

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

1 The Flask Framework

1.1 What is a Framework?

In computer science, a **framework** is a pre-built structure that helps developers create software applications. It provides a foundation with common functionalities and tools that developers can leverage instead of building everything from scratch. Some key characteristics of frameworks include but are not limited to:

- **Provision of structure**
  A framework establishes a foundation and organisation for the code, promoting well-structured and reliable software.

- **Reusability**
  Frameworks often come with pre-written code libraries that offer common functionalities, saving developers from writing the same code repeatedly.

- **Inversion of Control**
  Unlike libraries where your code calls upon the library functions, a framework might dictate the program flow by calling your code at specific points.

- **Standardisation**
  Frameworks often promote a specific way of developing applications, which can improve code consistency and make it easier for different developers to collaborate on a project.

There are many different frameworks available for different purposes and programming languages. Examples include Flask, Django and Ruby on Rails for web development, React and Angular for building user interfaces, and TensorFlow for machine learning applications.

1.2 Web Frameworks

In the context of web development, a **web framework** is a toolbox that provides a set of pre-written code, tools and libraries that simplify the process of building websites and web applications. The benefits of using web frameworks include but are not limited to:

- **Reduced development time and effort**
  Web frameworks contain pre-written code for common tasks, saving developers time and effort, allowing them to focus on unique features and functionality of their web application.

- **Improved code quality and maintainability**
  Web frameworks often enforce coding standards and best practices, which can help to improve the code quality and maintainability.

- **Enhanced security**
  Many web frameworks have built-in security features that can help to protect your web application from common attacks.

- **Easier collaboration**
  As web frameworks are based on common standards and practices, it can be easier for different developers to work on the same project.

1.3 The Flask Framework

In the 9569 H2 Computing curriculum, we shall learn how to use **Flask**, which is a **micro web framework** written in Python.

Being a micro web framework, Flask provides a lightweight and minimalist foundation for building web applications. Unlike larger web frameworks that offer many built-in features, Flask focuses on core functionalities like routing and templating. It allows web developers to choose additional features through extensions, providing more control and flexibility over the web application. The benefits of Flask include but are not limited to:

- **Simplicity**
  Flask is known for its ease of use and clear syntax, making it ideal for beginners learning web development with Python.

- **Flexibility**
  As Flask is a micro web framework, developers can customize it to fit specific needs by adding extensions for features like databases, user authentication, or form validation.

- **Scalability**
  Flask can be used to build simple web applications or complex ones as needed. It can scale well for larger projects by incorporating additional components.

1.4 Basic Flask Server

Without any customisations, Flask already provides a basic web server that correctly implements the **hypertext transfer protocol (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.

```
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`** error when you visit that URL using a web browser. The title on the browser tab will also read **`404 Not Found`**.

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

Note that value **`5000`** specified in **`http://127.0.0.1:5000/`** 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, replace line **`6`** with:

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

To stop the Flask server, press **`Ctrl`** + **`C`** concurrently 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 e.g.

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

- **Method `GET`**
  This indicates the type of HTTP request being made. In this case, `"GET"` signifies that the client is requesting to retrieve a resource from the server.

- **Path `/readme.txt`**
  This specifies the location of the resource on the server that the client wants to access. Here, `"/readme.txt"` indicates that the client is specifically requesting the file named `"readme.txt"`.

- **Protocol `HTTP/1.1`**
  This defines the version of the Hypertext Transfer Protocol (HTTP) that is being used for communication between the client and the server. `HTTP/1.1` is the most commonly used version of HTTP.

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`** i.e.

```
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 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.

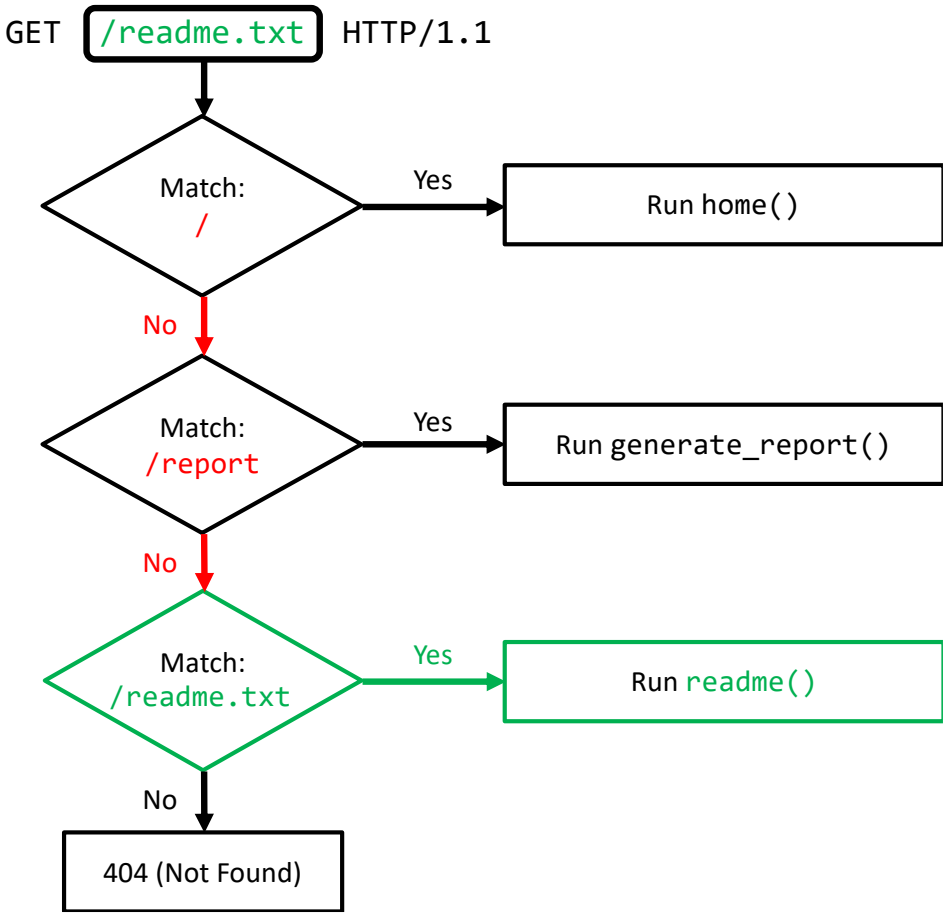
More details on HTTP will be discussed when you study networks.

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:

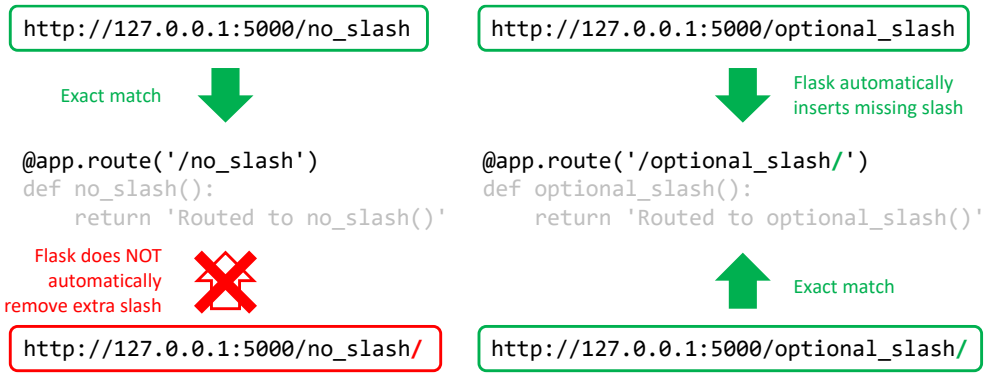
1) Go to **http://127.0.0.1:5000/**
You should see a simple page with the message **Routed to index()**. This is because the path component of **http://127.0.0.1:5000/** is **'/'**, which matches the route specified on line **5** for the **index()** function. This means that the request is being handled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



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.

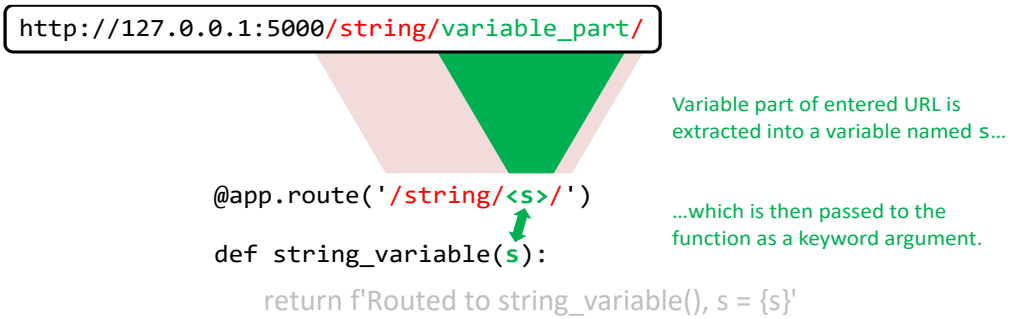
```
1   import flask
2
3   app = flask.Flask(__name__)
4
5   @app.route('/string/<s>/')
6   def string_variable(s):
7     return f'Routed to string_variable(), s = {s}'
8
9   if __name__ == '__main__':
10    app.run()
```

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.



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 f'Routed to integer_variable(), i = {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):

```
1   import flask
2
3   app = flask.Flask(__name__)
4
5   @app.route('/<name>')
6   def home(name):
7     return f'Hello, {name}!'
8
9   if __name__ == '__main__':
10    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.

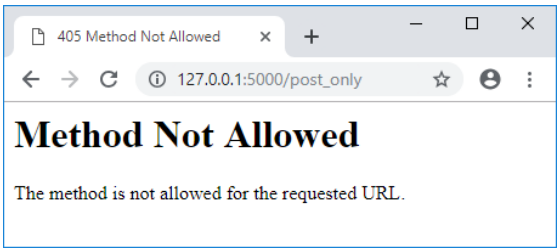
```
1   import flask
2
3   app = flask.Flask(__name__)
4
5   @app.route('/post_only/', methods=['POST'])
6   def post_only():
7     return 'Routed to post_only()'
8
9   if __name__ == '__main__':
10    app.run()
```

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.



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 f'Routed to string_variable(), s = {s}'
23
24 @app.route('/integer/<int:i>')
25 def integer_variable(i):
26   return f'Routed to integer_variable(), i = {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.

```
1  import flask
2
3  app = flask.Flask(__name__)
4
5  NAMES = ['January', 'February', 'March', 'April', 'May', 'June',
6  'July', 'August', 'September', 'October', 'November', 'December']
7
8  @app.route('/')
9  def home():
10   return 'Home'
11
12 @app.route('/<int:month>/')
13 def name_month(month):
14   if month in range(1, 13):
15     return f'Month {month}: {NAMES[month-1]}'
16   return 'Invalid month'
17
18 @app.route('/compare/<float:temp>/')
19 def compare_temp(temp):
20   if temp > 35.5:
21     return 'It\'s hot!'
22   if temp < 25.5:
23     return 'It\'s cold!'
24   return 'It\'s normal!'
25
26 @app.route('/greet/')
27 def greet():
28   return 'Hello!'
29
30 @app.route('/greet/<name>/')
31 def greet_name(name):
32   return 'Hello, {name}!'
33
34 @app.route('/data/', methods=['POST'])
35 def post_data():
36   return 'You are using POST'
37
38 @app.route('/data/', methods=['GET'])
39 def get_data():
40   return 'You are using GET'
41
42 if __name__ == '__main__':
43   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/`	