Previous Next

Unit 3 / Lesson 3 / Project 2

Hello World in Flask

Estimated Time: 1-2 hours

In the previous assignment you learned about web apps. In the rest of this unit you'll learn how to build web apps in Python using **Flask**, a web micro-framework. Flask provides a number of tools designed to simplify the most common tasks you'll encounter when writing web apps. It features a small core that can be easily customized and extended to easily add functionality to your apps.

Your first Flask app

Get started by creating a really simple "Hello World" Flask application. You'll first need to create a new directory and install Flask in a virtualenv. Virtualenv is a tool which allows you to create an isolated Python environment. This means that you can install libraries on a per-project basis. So, for example, you could have one project using version 0.9 of Flask, and another using version 0.10.

- Create a directory for the project - `mkdir flask_hello_world`
- Move into the project directory - `cd flask_hello_world`
- Create a new virtualenv - `python3 -m venv env`
- Activate the virtualenv - `source env/bin/activate`

- Install Flask - `pip install flask`

There are couple of things to note here. You will see a number of warnings and messages while Flask and its dependencies are being built. This is perfectly normal – just make sure that one of the final messages says that Flask was successfully installed.

Note that when you are inside an activated virtualenv you can deactivate it at any time by running `deactivate`. It can be re-activated by sourcing the activate script again.

Now in your project directory create a new file called *hello_world.py*. In this file you'll initialize Flask and then create a *view* function that will greet you when you visit a certain URL. Add the following content to your file:

```
from flask import Flask
from os import environ

app = Flask(__name__)

@app.route("/")
@app.route("/hello")
def say_hi():
    return "Hello World!"

if __name__ == "__main__":
    app.run(host=environ['IP'],
            port=int(environ['PORT']))
```

Here's what is happening:

- You import the `Flask` class. This will be the basis of the application.
- You then import the `environ` dictionary from `os` to get access to environment variables from Cloud9.

- Then you create an instance of the class called `app`, passing in the `__name__` variable to tell the app where it is being run from.
- Next you create a function called `say_hi`, which returns the string `"Hello World!"`. This function is decorated twice using the `app.route` method. This method is used to decorate every Flask view. It says that when you visit either the root URL '/' or the `/hello` URL of the Flask application, the `say_hi` function should run. You will take a closer look at decorators in a moment, so don't worry if this isn't entirely clear yet.
- In the main block you run the application using the `app.run` method. You use the `host` and `port` arguments to tell the application to listen on the values from the Cloud9 workspace environment.

Now try running the app and visiting the route. Fire up the app by running `python3 hello_world.py` from the command line. You should see something like this:

```
$ python3 hello_world.py
 * Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
```

This tells you that the Flask development server is listening for requests on port 8080. Try making a request to the `say_hi` view and seeing what happens. There are two ways to do this from Cloud9:

1. *The easy way*. Click the Preview button in the top bar once you've started your program, and choose "Preview running application". This will open a preview window in your workspace.
2. *The other way*. Cloud9 does some systems magic so your Flask application has a URL you can reach from outside. That URL is in the address bar of the preview window of the "easy way" above, but it's also in the Cloud9 terminal environment. You can find it by typing `echo`

`$C9_HOSTNAME` in the terminal. The result will be something like `thinkful-pip-1-thinkful.c9.io`. with `thinkful-pip-1` replaced by the name of your Cloud9 workspace, and with `thinkful` replaced by your user name on Cloud9. Open a new browser tab and put that into the address bar, and you should see your "Hello, world" there.

Try accessing the `say_hi` view by adding `/hello` to the URL of the page which preview opened. Visiting this URL should cause the browser to display `Hello World!` message, which is the string you told our view to return.

A note on decorators

In the previous code you made use of a decorator to tell Flask what action to perform when a certain URL is visited. You can think of decorators as a way to quickly add features to your functions. For example the `app.route` decorator adds the feature "when the user visits the specified URL, the decorated function should be run". Flask provides several useful decorators to our application.

To decorate a function you use the `@` symbol, followed by the name of the decorator. For example:

```
@decorator
def function():
    pass
```

In actual fact this is just some syntactic sugar for the following code:

```
def function():
    pass
function = decorator(function)
```

Think about what is happening here. Initially `function` is just a regular function which you know and love. Then it is redefined to be the result of passing it into another object. So what is `function` replaced with when it is redefined, and how does this allows you to add a feature?

Take a look at this example decorator:

```
def twist(function):
    def wrapper():
        print("Shep Schwab shopped at Scott's Schnapps shop")
    return wrapper
```

The decorator is a function called `twist`, which takes a single argument. This argument is the function which you want to decorate. Then there is a nested function, called `wrapper` which for now prints a tongue-twister. Finally the `twist` function returns the inner `wrapper` function.

What happens when you use this? Look at how it is used it to decorate a function:

```
@twist
def spoon():
    print("A well-boiled icicle")
```

And remember that this code is the equivalent to:

```
def spoon():
    print("A well-boiled icicle")
spoon = twist(spoon)
```

The function `spoon` is replaced by the return value of the `twist` function. So `spoon` is replaced by the inner `wrapper` function of the `twist` decorator. When you try calling `spoon` you should see the tongue-twister rather than the spoonerism being printed.

At this point you have replaced the functionality of `spoon` with the functionality of `wrapper` rather than extending it. But remember that you are passing the `spoon` function into the decorator. Try making a slight change:

```
import functools

def twist(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        print("Shep Schwab shopped at Scott's Schnapps shop")
        function(*args, **kwargs)
    return wrapper
```

The function which is passed in is now being called inside the `wrapper` function. So when you call the decorated spoon function (which is actually the `wrapper` function), the tongue-twister will be printed, and then the original undecorated `spoon` function will be called, printing the spoonerism. (The use of `*args` and `**kwargs` allows you to pass any number of arguments on to this function. It means you don't need to care what parameters it expects - they'll all be passed along. Thanks to functools.wraps, the documentation will still be correct, too.)

If you wanted to pass argument into the decorator (like in `app.route`) you need to add one more level of nesting. So for example, to to allow the tongue-twister to be changed you could use the following code:

```
import functools


def twist(twister):
    def decorator(function):
        @functools.wraps(function)
        def wrapper(*args, **kwargs):
            print(twister)
            function(*args, **kwargs)
            print(twister)
```