tutorialspoint
SIMPLYEASYLEARNING

About the Tutorial

Flask is a web application framework written in Python. Armin Ronacher, who leads an international group of Python enthusiasts named Pocco, develops it. Flask is based on Werkzeug WSGI toolkit and Jinja2 template engine. Both are Pocco projects.

Audience

This tutorial has been prepared for anyone who has a basic knowledge of Python and has an urge to develop websites. After completing this tutorial, you will find yourself at a moderate level of expertise in developing websites using Flask.

Prerequisites

Before you start proceeding with this tutorial, we are assuming that you have hands-on experience on HTML and Python. If you are not well aware of these concepts, then we will suggest you to go through our short tutorials on HTML and Python.

Copyright & Disclaimer

Table of Contents

1. FLASK – OVERVIEW

What is Web Framework?

Web Application Framework or simply Web Framework represents a collection of libraries and modules that enables a web application developer to write applications without having to bother about low-level details such as protocols, thread management etc.

What is Flask?

Flask is a web application framework written in Python. It is developed by **Armin Ronacher**, who leads an international group of Python enthusiasts named Pocco. Flask is based on the Werkzeug WSGI toolkit and Jinja2 template engine. Both are Pocco projects.

WSGI

Web Server Gateway Interface (WSGI) has been adopted as a standard for Python web application development. WSGI is a specification for a universal interface between the web server and the web applications.

Werkzeug

It is a WSGI toolkit, which implements requests, response objects, and other utility functions. This enables building a web framework on top of it. The Flask framework uses Werkzeug as one of its bases.

Jinga2

Jinga2 is a popular templating engine for Python. A web templating system combines a template with a certain data source to render dynamic web pages.

Flask is often referred to as a micro framework. It aims to keep the core of an application simple yet extensible. Flask does not have built-in abstraction layer for database handling, nor does it have form a validation support. Instead, Flask supports the extensions to add such functionality to the application. Some of the popular Flask extensions are discussed later in the tutorial.

2. FLASK – ENVIRONMENT

Prerequisite

Python 2.6 or higher is usually required for installation of Flask. Although Flask and its dependencies work well with Python 3 (Python 3.3 onwards), many Flask extensions do not support it properly. Hence, it is recommended that Flask should be installed on Python 2.7.

Install virtualenv for development environment

virtualenv is a virtual Python environment builder. It helps a user to create multiple Python environments side-by-side. Thereby, it can avoid compatibility issues between the different versions of the libraries.

The following command installs **virtualenv**.

```
pip install virtualenv
```

This command needs administrator privileges. Add **sudo** before **pip** on Linux/Mac OS. If you are on Windows, log in as Administrator. On Ubuntu **virtualenv** may be installed using its package manager.

```
Sudo apt-get install virtualenv
```

Once installed, new virtual environment is created in a folder.

```
mkdir newproj

cd newproj

virtualenv venv
```

To activate corresponding environment, on **Linux/OS X**, use the following:

```
venv/bin/activate
```

On **Windows**, following can be used:

```
venv\scripts\activate
```

We are now ready to install Flask in this environment.

```
pip install Flask
```

The above command can be run directly, without virtual environment for system-wide installation.

3. FLASK – APPLICATION

In order to test **Flask** installation, type the following code in the editor as **Hello.py**

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

@app.route('/')
def hello_world():
    return 'Hello World’
if __name__ == '__main__':
    app.run()
```

Importing flask module in the project is mandatory. An object of Flask class is our **WSGI** application.

Flask constructor takes the name of **current module (__name__)** as argument.

The **route()** function of the Flask class is a decorator, which tells the application which URL should call the associated function.

```
app.route(rule, options)
```

- The **rule** parameter represents URL binding with the function.
-
- The **options** is a list of parameters to be forwarded to the underlying Rule object.

In the above example, **'/'** URL is bound with **hello_world()** function. Hence, when the home page of web server is opened in browser, the output of this function will be rendered.

Finally the **run()** method of Flask class runs the application on the local development server.

```
app.run(host, port, debug, options)
```

All parameters are optional

host	Hostname to listen on. Defaults to 127.0.0.1 (localhost). Set to ‘0.0.0.0’ to have server available externally
port	Defaults to 5000
debug	Defaults to false. If set to true, provides a debug information
options	To be forwarded to underlying Werkzeug server.

The above given **Python** script is executed from Python shell.

```
Python Hello.py
```

A message in Python shell informs you that

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Open the above URL **(localhost:5000)** in the browser. **'Hello World'** message will be displayed on it.

Debug mode

A **Flask** application is started by calling the **run()** method. However, while the application is under development, it should be restarted manually for each change in the code. To avoid this inconvenience, enable **debug support**. The server will then reload itself if the code changes. It will also provide a useful debugger to track the errors if any, in the application.

The **Debug** mode is enabled by setting the **debug** property of the **application** object to **True** before running or passing the debug parameter to the **run()** method.

```
app.debug=True

app.run()

app.run(debug=True)
```

4. FLASK – ROUTING

Modern web frameworks use the routing technique to help a user remember application URLs. It is useful to access the desired page directly without having to navigate from the home page.

The **route()** decorator in Flask is used to bind URL to a function. For example:

```
@app.route(‘/hello’)
def hello_world():
    return ‘hello world’
```

Here, URL **‘/hello’** rule is bound to the **hello_world()** function. As a result, if a user visits http://localhost:5000/hello URL, the output of the **hello_world()** function will be rendered in the browser.

The **add_url_rule()** function of an application object is also available to bind a URL with a function as in the above example, **route()** is used.

A decorator’s purpose is also served by the following representation:

```
def hello_world():
    return ‘hello world’
app.add_url_rule(‘/’, ‘hello’, hello)
```

5. FLASK – VARIABLE RULES

It is possible to build a URL dynamically, by adding variable parts to the rule parameter. This variable part is marked as **<variable-name>.** It is passed as a keyword argument to the function with which the rule is associated.

In the following example, the rule parameter of **route()** decorator contains **<name>** variable part attached to URL **'/hello'**. Hence, if the http://localhost:5000/hello/TutorialsPoint is entered as a **URL** in the browser, **'TutorialPoint'** will be supplied to **hello()** function as argument.

```
from flask import Flask
app = Flask(__name__)
@app.route('/hello/<name>')
def hello_name(name):
  return 'Hello %s!' % name
if __name__ == '__main__':
    app.run(debug=True)
```

Save the above script as **hello.py** and run it from Python shell. Next, open the browser and enter URL http://localhost:5000/hello/TutorialsPoint.

The following output will be displayed in the browser.

```
Hello TutorialsPoint!
```

In addition to the default string variable part, rules can be constructed using the following converters:

int	accepts integer
float	For floating point value
path	accepts slashes used as directory separator character

In the following code, all these constructors are used.

```
from flask import Flask
```

```
app = Flask(__name__)
@app.route('/blog/<int:postID>')
def show_blog(postID):
    return 'Blog Number %d' % postID
@app.route('/rev/<float:revNo>')
def revision(revNo):
    return 'Revision Number %f' % revNo
if __name__ == '__main__':
    app.run()
```

Run the above code from Python Shell. Visit the URL http://localhost:5000/blog/11 in the browser.

The given number is used as argument to the **show_blog()** function. The browser displays the following output:

Blog Number 11

Enter this URL in the browser: http://localhost:5000/rev/1.1

The **revision()** function takes up the floating point number as argument. The following result appears in the browser window:

Revision Number 1.100000

The URL rules of Flask are based on **Werkzeug's** routing module. This ensures that the URLs formed are unique and based on precedents laid down by Apache.

Consider the rules defined in the following script:

```
from flask import Flask
app = Flask(__name__)
@app.route('/flask')
def hello_flask():
    return 'Hello Flask'
@app.route('/python/')
def hello_python():
    return 'Hello Python'
if __name__ == '__main__':
    app.run()
```

Both the rules appear similar but in the second rule, trailing slash **(/)** is used. As a result, it becomes a canonical URL. Hence, using **/python** or **/python/** returns the same output. However, in case of the first rule, **/flask/** URL results in **404 Not Found** page.

6. FLASK – URL BUILDING

The **url_for()** function is very useful for dynamically building a URL for a specific function. The function accepts the name of a function as first argument, and one or more keyword arguments, each corresponding to the variable part of URL.

The following script demonstrates use of **url_for()** function.

```
from flask import Flask, redirect, url_for
app = Flask(__name__)
@app.route('/admin')
def hello_admin():
    return 'Hello Admin'
@app.route('/guest/<guest>')
def hello_guest(guest):
    return 'Hello %s as Guest' % guest
@app.route('/user/<name>')
def hello_user(name):
    if name=='admin':
        return redirect(url_for('hello_admin'))
    else:
        return redirect(url_for('hello_guest',guest=name))
if __name__ == '__main__':
    app.run(debug=True)
```

The above script has a function **user(name)** which accepts a value to its argument from the URL.

The **User()** function checks if an argument received matches **'admin'** or not. If it matches, the application is redirected to the **hello_admin()** function using **url_for(),** otherwise to the **hello_guest()** function passing the received argument as guest parameter to it.

Save the above code and run from Python shell.

Open the browser and enter URL as:

http://localhost:5000/hello/admin