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.

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

1. Host : Hostname to listen on. Defaults to 127.0.0.1 (localhost). Set to ‘0.0.0.0’ to have server available externally
2. **Port : D**efaults to 5000
3. **Debug :** Defaults to false. If set to true, provides a debug information
4. **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)

Chapter 2

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\_world)

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/**](http://localhost:5000/hello/)**jay** is entered as a **URL** in the browser, “jay” 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 {}!'.format(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/jay.**

The following output will be displayed in the browser.

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 {}'.format(postID)

@app.route('/rev/<float:revNo>')

def revision(revNo):

return 'Revision Number {}’.format(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.

Chapter 3

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 {} as Guest'.format(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/user/admin**

The application response in browser is −

Hello Admin

Enter the following URL in the browser − **http://localhost:5000/user/mvl**

The application response now changes to −

Hello mvl as Guest

Http protocol is the foundation of data communication in world wide web. Different methods of data retrieval from specified URL are defined in this protocol.

The following table summarizes different http methods −

**GET :** Sends data in unencrypted form to the server. Most common method

**HEAD :** Same as GET, but without response body

**POST :** Used to send HTML form data to server. Data received by POST method is not cached by server

**PUT :** Replaces all current representations of the target resource with the uploaded content

**DELETE :** Removes all current representations of the target resource given by a URL

By default, the Flask route responds to the **GET** requests. However, this preference can be altered by providing methods argument to **route()** decorator.

In order to demonstrate the use of **POST** method in URL routing, first let us create an HTML form and use the **POST** method to send form data to a URL.

Save the following script as login.html

<html>

<body>

<form action = "http://localhost:5000/login" method = "post">

<p>Enter Name:</p>

<p><input type = "text" name = "nm" /></p>

<p><input type = "submit" value = "submit" /></p>

</form>

</body>

</html>

Now enter the following script in Python shell.

from flask import Flask, redirect, url\_for, request

app = Flask(\_\_name\_\_)

@app.route('/success/<name>')

def success(name):

return 'welcome {}'.format(name)

@app.route('/login',methods = ['POST', 'GET'])

def login():

if request.method == 'POST':

user = request.form['nm']

return redirect(url\_for('success',name = user))

else:

user = request.args.get('nm')

return redirect(url\_for('success',name = user))

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug = True)

After the development server starts running, open **login.html** in the browser, enter name in the text field and click **Submit**.



Form data is POSTed to the URL in action clause of form tag.

**http://localhost/login** is mapped to the **login()** function. Since the server has received data by **POST** method, value of ‘nm’ parameter obtained from the form data is obtained by −

user = request.form['nm']

It is passed to **‘/success’** URL as variable part. The browser displays a **welcome** message in the window.



Change the method parameter to **‘GET’** in **login.html** and open it again in the browser. The data received on server is by the **GET** method. The value of ‘nm’ parameter is now obtained by −

User = request.args.get(‘nm’)

Here, **args** is dictionary object containing a list of pairs of form parameter and its corresponding value. The value corresponding to ‘nm’ parameter is passed on to ‘/success’ URL as before

Chapter 5

It is possible to return the output of a function bound to a certain URL in the form of HTML. For instance, in the following script, **hello()** function will render **‘Hello World’** with **<h1>** tag attached to it.

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def index():

return '<html><body><h1>Hello World</h1></body></html>'

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug = True)

However, generating HTML content from Python code is cumbersome, especially when variable data and Python language elements like conditionals or loops need to be put. This would require frequent escaping from HTML.

This is where one can take advantage of **Jinja2** template engine, on which Flask is based. Instead of returning hardcode HTML from the function, a HTML file can be rendered by the **render\_template()** function.

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def index():

return render\_template(‘hello.html’)

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug = True)

Flask will try to find the HTML file in the templates folder, in the same folder in which this script is present.

* Application folder
  + Hello.py
  + templates
    - hello.html

The term **‘web templating system’** refers to designing an HTML script in which the variable data can be inserted dynamically. A web template system comprises of a template engine, some kind of data source and a template processor.

Flask uses **jinja2** template engine. A web template contains HTML syntax interspersed placeholders for variables and expressions (in these case Python expressions) which are replaced values when the template is rendered.

The following code is saved as **hello.html** in the templates folder.

<!doctype html>

<html>

<body>

<h1>Hello {{ name }}!</h1>

</body>

</html>

Next, run the following script from Python shell.

from flask import Flask, render\_template

app = Flask(\_\_name\_\_)

@app.route('/hello/<user>')

def hello\_name(user):

return render\_template('hello.html', name = user)

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug = True)

As the development server starts running, open the browser and enter URL as − **http://localhost:5000/hello/mvl**

The **variable** part of URL is inserted at **{{ name }}** place holder.



The **jinja2** template engine uses the following delimiters for escaping from HTML.

* {% ... %} for Statements
* {{ ... }} for Expressions to print to the template output
* {# ... #} for Comments not included in the template output
* # ... ## for Line Statements

In the following example, use of conditional statement in the template is demonstrated. The URL rule to the **hello()** function accepts the integer parameter. It is passed to the **hello.html** template. Inside it, the value of number received (marks) is compared (greater or less than 50) and accordingly HTML is conditionally rendered.

The Python Script is as follows −

from flask import Flask, render\_template

app = Flask(\_\_name\_\_)

@app.route('/hello/<int:score>')

def hello\_name(score):

return render\_template('hello.html', marks = score)

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug = True)

Note that the conditional statements **if-else** and **endif** are enclosed in delimiter **{%..%}**.

Run the Python script and visit URL **http://localhost/hello/60** and then **http://localhost/hello/30** to see the output of HTML changing conditionally.

The Python loop constructs can also be employed inside the template. In the following script, the **result()** function sends a dictionary object to template **results.html** when URL **http://localhost:5000/result** is opened in the browser.

The Template part of **result.html** employs a **for loop** to render key and value pairs of dictionary object **result{}** as cells of an HTML table.

Run the following code from Python shell.

from flask import Flask, render\_template

app = Flask(\_\_name\_\_)

@app.route('/result')

def result():

dict = {'phy':50,'che':60,'maths':70}

return render\_template('result.html', result = dict)

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug = True)

Save the following HTML script as **result.html** in the templates folder.

<!doctype html>

<html>

<body>

<table border = 1>

{% for key, value in result.items() %}

<tr>

<th> {{ key }} </th>

<td> {{ value }} </td>

</tr>

{% endfor %}

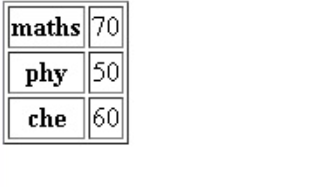
</table>

</body>

</html>

Here, again the Python statements corresponding to the **For** loop are enclosed in {%..%} whereas, the expressions **key and value** are put inside **{{ }}**.

After the development starts running, open **http://localhost:5000/result** in the browser to get the following output.



Project Code

**Database Codes**

create database admindatabase;

create table users(

uid int PRIMARY KEY AUTO\_INCREMENT,

uname varchar(20),

uemail varchar(30),

upassword varchar(15) );

Save the below code in notepad file as loginapp.html

<div align="center">

<h1>Login Page</h1>

<form action="/validuser" method="POST">

Username : <input type="text" name="uname" required><br><br>

Password : <input type="password" name="upassword" required><br><br>

<input type="submit" value="LOGIN">

<br><br>

Don't have an Account...!

<a href="/register">Register</a>

</form>

<h3><a href="/">Login</a>

<a href="/register">Register</a>

</h3>

</div>

Save the below code as register.html in nodepad

<div align="center">

<h1>Register Page</h1>

<form action="/adduser" method="POST">

Username : <input type="text" name="uname" required><br><br>

Email id : <input type="email" name="uemail" required><br><br>

Password : <input type="password" name="upassword" required><br><br>

<input type="reset" value="Reset">

<input type="submit" value="Submit">

</form>

<h3><a href="/">Login</a>

<a href="/register">Register</a>

</h3>

</div>

Save the below code as home.html

<div align="center">

<h1>Welcome to Home Page</h1>

</div>

Save the below code as DBM.py file

import pymysql as p

def getconnect():

return p.connect(host="localhost", user="root", password="", database="mydatabase")

print("Connection Done...!")

def insertdata(t):

db =getconnect()

cr = db.cursor()

sql = "insert into users(uname, uemail, upassword) values(%s, %s, %s)"

cr.execute(sql, t)

print("Data Inserted Successfully...!")

db.commit()

db.close()

def getdata():

db =getconnect()

cr = db.cursor()

sql = "select uname, upassword from users"

cr.execute(sql)

data = cr.fetchall()

print("Data fetched Successfully...!")

db.commit()

db.close()

print(data)

return data

from flask import \*

app = Flask(\_\_name\_\_)

@app.route("/")

def page1():

return render\_template("login.html")

@app.route("/register")

def page2():

return render\_template("register.html")

@app.route("/adduser", methods=["POST"])

def page3():

uname = request.form["uname"]

uemail = request.form["uemail"]

upassword = request.form["upassword"]

t = (uname, uemail, upassword)

insertdata(t)

return redirect("/")

@app.route("/home")

def home():

return render\_template("home.html")

@app.route("/validuser", methods=["POST"])

def page4():

database = getdata()

u = request.form["uname"]

print(u)

p = request.form["upassword"]

if(u, p) in database:

return redirect("/home")

else:

return redirect("/register")

if(\_\_name\_\_=="\_\_main\_\_"):

app.run(debug=True)[