Flask

Python Web Framework

Dr. Sarwan Singh
NIELIT Chandigarh

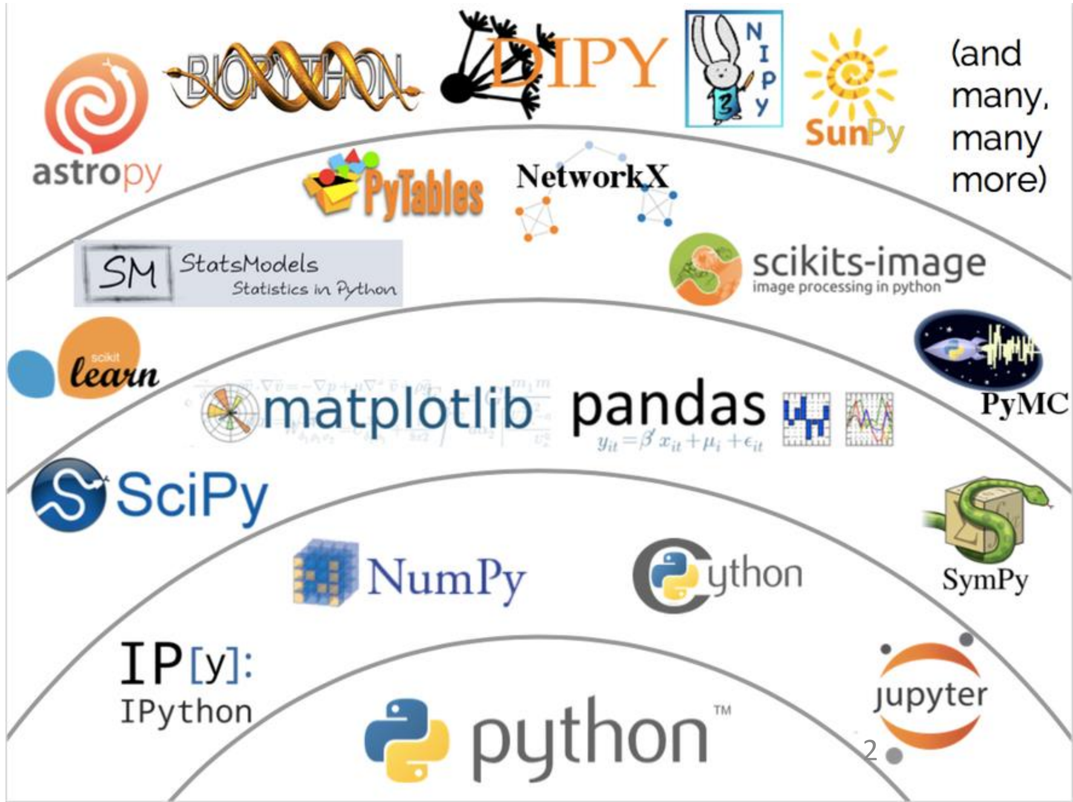
Agenda

- Introduction – History,
- First web application, Folder structure
- App routing
- Dynamic URL Building
- HTTP methods – POST, GET
- Flask templates, session, cookies
- URL redirect, user mgt

Artificial Intelligence

Machine Learning

Deep Learning



One guiding principle of Python code is that
“explicit is better than implicit”

References

- Fullstackpython, javatpoint

Important pip commands

- pip install pipreqs
- pip install Flask-SQLAlchemy
-

Introduction

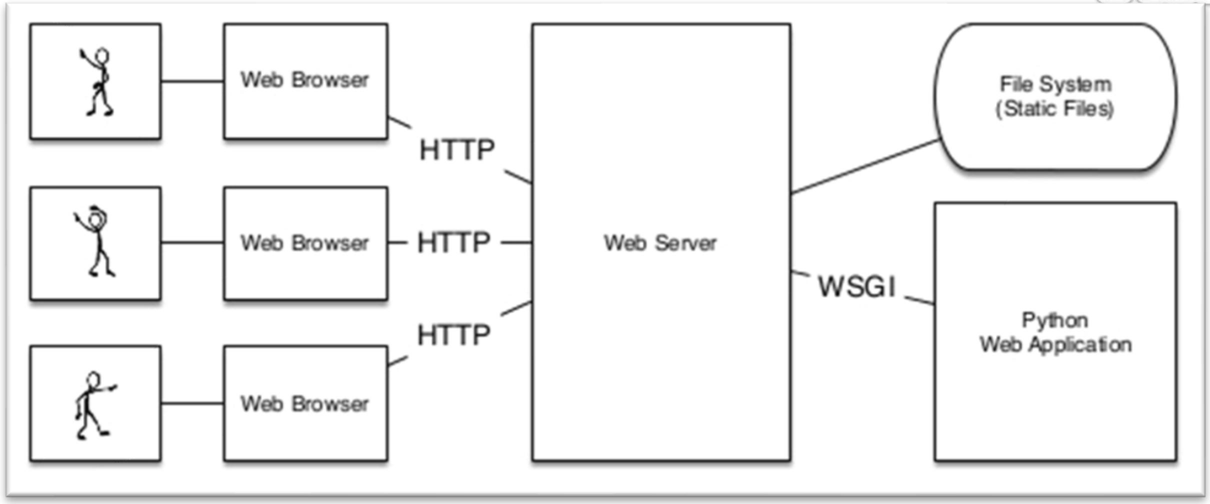
- Flask is a micro web framework written in Python.
- It is classified as a microframework because it does not require particular tools or libraries.
- It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions.
- Developed in 2010 by Armin Ronacher as an April Fool's Day joke
- Armin Ronacher leads an international group of python enthusiasts (POCCO).

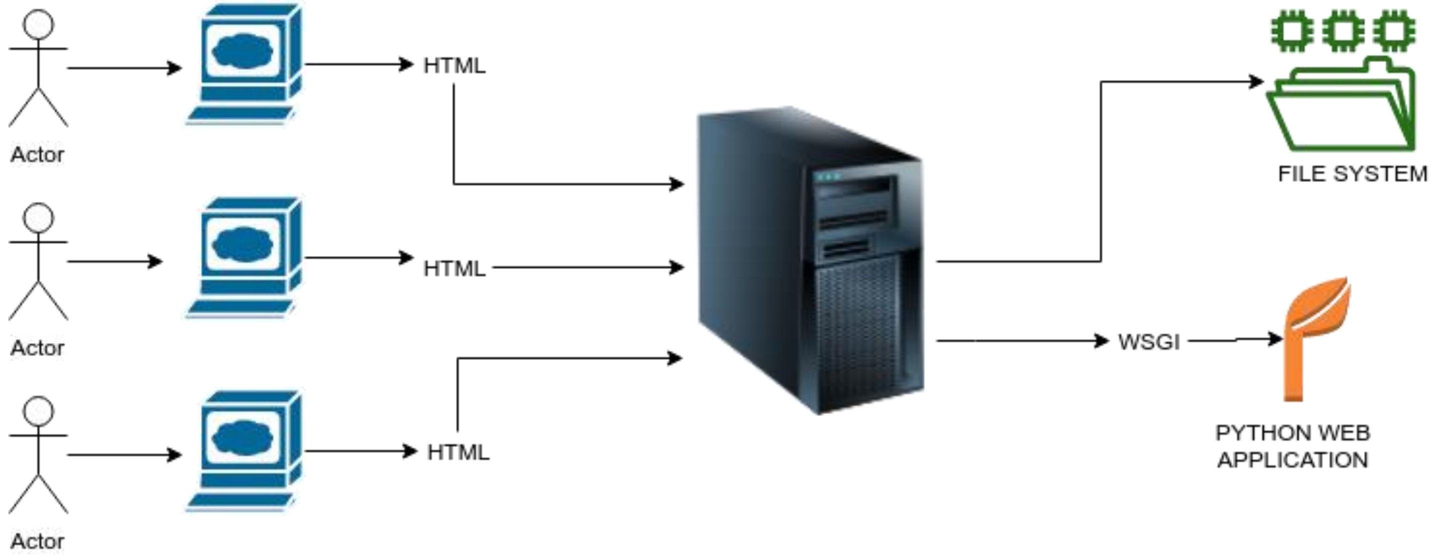
Introduction

- Flask is based on WSGI toolkit and jinja2 template engine.
  - **WSGI-** web server gateway interface which is a standard for python web application development.
  - **WSGI** is considered as the specification for the universal interface between the web server and web application.
  - **Jinja2** is a web template engine which combines a template with a certain data source to render the dynamic web pages.
- Flask is written in python, Licensed under BSD
- Has monolithic structure and dependencies and considered as a micro framework

Web Server Gateway Interface

- **WSGI** is the Web Server Gateway Interface. It is a specification that describes how a web server communicates with web applications, and how web applications can be chained together to process one request.
- Flask inherits its high WSGI usage





WEB SERVER GATEWAY INTERFACE

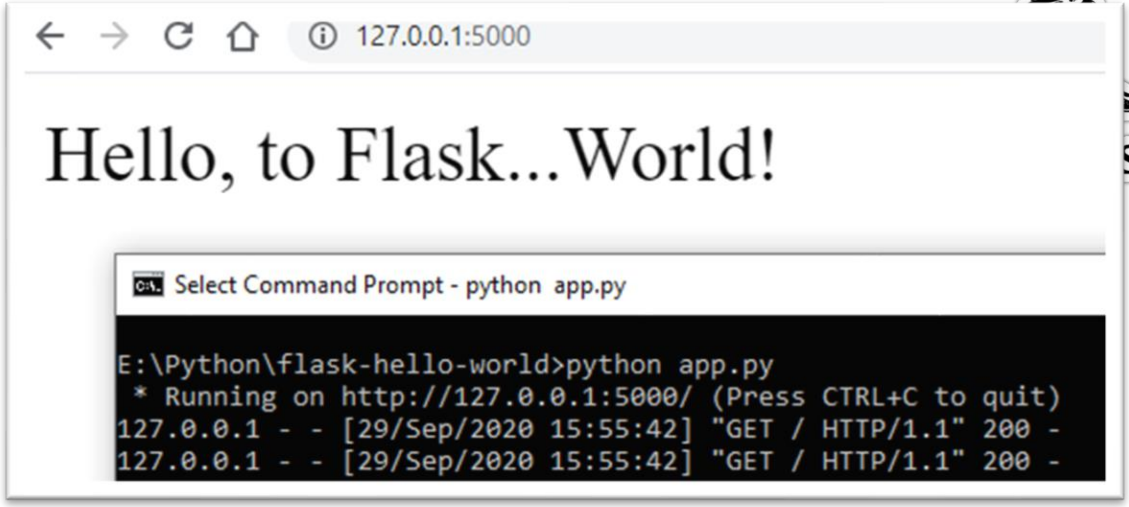
Hello world with Flask

- code shows

  "Hello, World!" on localhost port 5000 in a web browser

- when run with the

  python app.py command



127.0.0.1:5000

Hello, to Flask...World!

Select Command Prompt - python app.py

```
E:\Python\flask-hello-world>python app.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [29/Sep/2020 15:55:42] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [29/Sep/2020 15:55:42] "GET / HTTP/1.1" 200 -
```

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

@app.route('/')
def hello_world():
    return 'Hello, World!'

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

Hello world with Flask - details

- the name of the current module, i.e. __name__ is to be passed as the argument into the Flask constructor.
- The route() function of the Flask class defines the URL mapping of the associated function. The syntax is :

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

It accepts the following parameters.

- <u>rule</u>: It represents the URL binding with the function.
- <u>options</u>: It represents the list of parameters to be associated with the rule object

Hello world with Flask - details

- run method of the Flask class is used to run the flask application on the local development server. The syntax is :

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

It accepts the following parameters.

- host : The default hostname is 127.0.0.1, i.e. localhost.
- port : The port number to which the server is listening to. The default port number is 5000.
- debug : The default is false. It provides debug information if it is set to true.
- options : It contains the information to be forwarded to the server.

Flask App routing

- App routing is used to map the specific URL with the associated function that is intended to perform some task
- visiting the particular URL mapped to some particular function, the output of that function is rendered on the browser's screen.

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

@app.route('/home')
def home():
    return "hello, welcome
to our website";

if __name__ =="__main__":
    app.run(debug = True)
```

App routing

- add the variable part to the URL
- reuse the variable by adding that as a parameter into the view function

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

@app.route('/home/<name>')
def home(name):
    return "hello,"+name;

if __name__ =="__main__":
    app.run(debug = True)
```

App routing

- converter can also be used in the URL to map the specified variable to the particular data type
  - string: default
  - int: used to convert the string to the integer
  - float: used to convert the string to the float.
  - path: It can accept the slashes given in the URL.

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

@app.route('/home/<int:age>')
def home(age):
    return "Age = %d"%age;

if __name__ =="__main__":
    app.run(debug = True)
```

App routing - another approach

- Routing can be performed for the flask web application using the add_url() function of the Flask class

add_url_rule(<url rule>, <endpoint>, <view function>)

- This function is mainly used in the case if the view function is not given and we need to connect a view function to an endpoint externally by using this function.

App routing - another approach

add_url_rule(<url rule>, <endpoint>, <view function>)

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

def about():
    return "This is about page";

app.add_url_rule("/about","about",about)

if __name__ =="__main__":
    app.run(debug = True)
```

Flask Dynamic URL Building

- url_for() function is used to build a URL to the specific function dynamically.
- The first argument is the name of the specified function, and then we can pass any number of keyword argument corresponding to the variable part of the URL.
- function is useful in the sense that we can avoid hard-coding the URLs into the templates by dynamically building them using this function.

Flask Dynamic URL Building

- script simulates the library management system which can be used by the three types of users, i.e., admin, librarian, and student.
- user() function which recognizes the user and then redirect the user to the exact function

```
from flask import *

app = Flask(__name__)

@app.route('/admin')
def admin():
    return 'admin'

@app.route('/librarian')
def librarion():
    return 'librarian'

@app.route('/student')
def student():
    return 'student'
```

```
@app.route('/user/<name>')
def user(name):
    if name == 'admin':
        return redirect(url_for('admin'))
    if name == 'librarian':
        return redirect(url_for('librarian'))
    if name == 'student':
        return redirect(url_for('student'))
if __name__ =='__main__':
    app.run(debug = True)
```

Flask Dynamic URL Building : Benefits

- It avoids hard coding of the URLs.
- We can change the URLs dynamically instead of remembering the manually changed hard-coded URLs.
- URL building handles the escaping of special characters and Unicode data transparently.
- The generated paths are always absolute, avoiding unexpected behavior of relative paths in browsers.
- If your application is placed outside the URL root, for example, in /myapplication instead of /, url_for() properly handles that for you.

Flask HTTP methods

- HTTP is the hypertext transfer protocol which is considered as the foundation of the data transfer in the world wide web.
- All web frameworks including flask need to provide several HTTP methods for data communication. The methods are :

1	GET (Default)	It is the most common method which can be used to send data in the unencrypted form to the server.
2	HEAD	It is similar to the GET but used without the response body.
3	POST	It is used to send the form data to the server. The server does not cache the data transmitted using the post method.
4	PUT	It is used to replace all the current representation of the target resource with the uploaded content.
5	DELETE	It is used to delete all the current representation of the target resource specified in the URL.

Flask HTTP methods - POST

- Code for POST requests at the server

Login.html

```
<html>
  <body>
    <form action = http://localhost:5000/login
             method = "post">
      <table>
     <tr><td>Name</td>
     <td><input type ="text" name ="uname"></td></tr>
     <tr><td>Password</td>
     <td><input type ="password" name ="pass"></td></tr>
     <tr><td><input type = "submit"></td></tr>
  </table>
    </form>
  </body>
</html>
```

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

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

@app.route('/login',methods = ['POST'])
def login():
      uname=request.form['uname']
      passwrd=request.form['pass']
      if uname== "ram" and passwrd== "sita":
          return "Welcome %s" %uname

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

Flask HTTP methods - get

- Data send through GET method is retrieved using :
  - uname=request.args.get('uname')
  - passwrd=request.args.get('pass')

Flask templates

- Flask facilitates us to return the response in the form of HTML templates

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

@app.route('/')
def message():
      return "<html><body><h1>Hi, welcome to NIELIT
</h1></body></html>"

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

Flask templates – rendering external HTML files

- Flask takes advantage of jinja2 template engine
- render_template() function which can be used to render the external HTML file to be returned as the response from the view function.
- Application directory
  - app.py
  - templates
    - show.html

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

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

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

Flask templates - Delimiters

- Jinja 2 template engine provides some delimiters which can be used in the HTML to make it capable of dynamic data representation.
- The template system provides some HTML syntax-placeholders for variables and expressions that are replaced by their actual values when the template is rendered.
- The jinja2 template engine provides the following delimiters to escape from the HTML.
    - {% ... %} for statements
    - {{ ... }} for expressions to print to the template output
    - {# ... #} for the comments that are not included in the template output
    - # ... ## for line statements

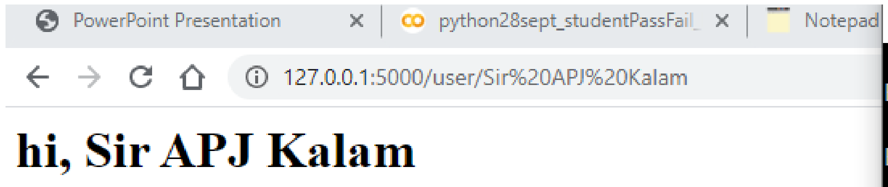
Flask templates

- Passing values to the html pages
-

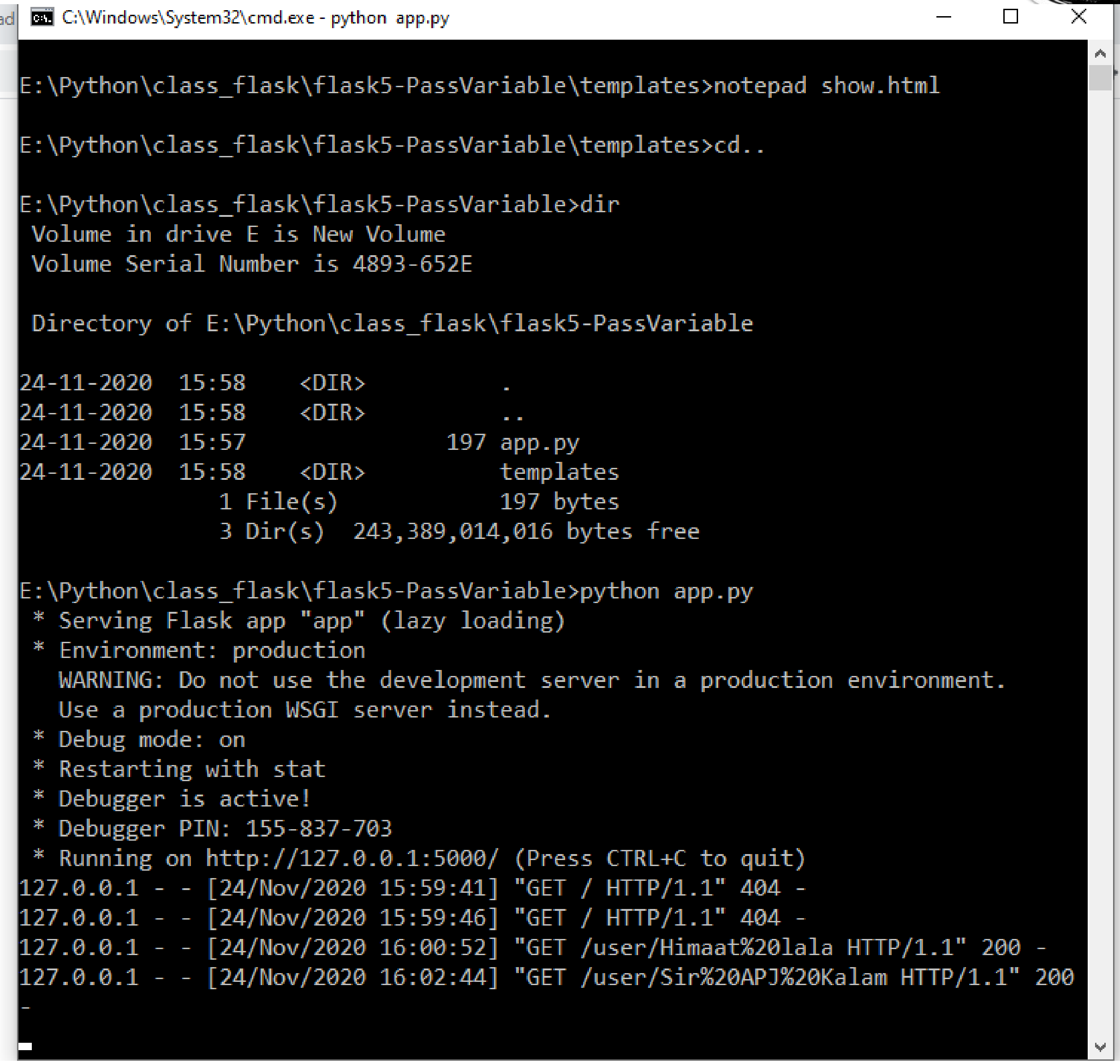
```
<html>
<head>
<title>Message</title>
</head>
<body>
<h1>hi, {{ name }} </h1>
</body>
</html>
```

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

@app.route('/user/<uname>')
def message(uname):
      return render_template
  ('show.html',name=uname)
if __name__ == '__main__':
   app.run(debug = True)
```


PowerPoint Presentation
python28sept_studentPassFail_
Notepad
127.0.0.1:5000/user/Sir%20APJ%20Kalam

hi, Sir APJ Kalam



```
E:\Python\class_flask\flask5-PassVariable\templates>notepad show.html

E:\Python\class_flask\flask5-PassVariable\templates>cd..

E:\Python\class_flask\flask5-PassVariable>dir
 Volume in drive E is New Volume
 Volume Serial Number is 4893-652E

 Directory of E:\Python\class_flask\flask5-PassVariable

24-11-2020  15:58    <DIR>          .
24-11-2020  15:58    <DIR>          ..
24-11-2020  15:57               197 app.py
24-11-2020  15:58    <DIR>          templates
               1 File(s)            197 bytes
               3 Dir(s)  243,389,014,016 bytes free

E:\Python\class_flask\flask5-PassVariable>python app.py
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 155-837-703
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [24/Nov/2020 15:59:41] "GET / HTTP/1.1" 404 -
127.0.0.1 - - [24/Nov/2020 15:59:46] "GET / HTTP/1.1" 404 -
127.0.0.1 - - [24/Nov/2020 16:00:52] "GET /user/Himaat%20lala HTTP/1.1" 200 -
127.0.0.1 - - [24/Nov/2020 16:02:44] "GET /user/Sir%20APJ%20Kalam HTTP/1.1" 200
-
```

Flask - Embedding Python statements in HTML

- Flask facilitates us the delimiter {%...%} which can be used to embed the simple python statements into the HTML.

```
<html>
<body>
<h2> printing table of {{n}}</h2>
{% for i  in range(1,11): %}
    <h3>{{n}} X {{i}} = {{n * i}} </h3>
{% endfor %}
</body>
</html>
```

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

@app.route('/table/<int:num>')
def table(num):
      return render_template('print-table.html',n=num)

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

Flask – referring static files in HTML

- CSS or JavaScript file can be referred in HTML
- Files are kept under static folder in application folder

```
<html>
<head>
    <title>Message</title>
    <link rel="stylesheet"
href="{{ url_for('static', filename='css/style.css') }}">
</head>

<body>
...
</body>
</html>
```

Flask Request Object

- In the client-server architecture, the request object contains all the data that is sent from the client to the server.

Form	It is the dictionary object which contains the key-value pair of form parameters and their values.
args	It is parsed from the URL. It is the part of the URL which is specified in the URL after question mark (?).
Cookies	It is the dictionary object containing cookie names and the values. It is saved at the client-side to track the user session.
files	It contains the data related to the uploaded file.
method	It is the current request method (get or post).

Form data retrieval on the template

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

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

@app.route('/success',methods = ['POST', 'GET'])
def print_data():
   if request.method == 'POST':
      result = request.form
      return render_template("result_data.html",result = result)

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

- application contains three files, i.e., app.py, student.html, and result_data.html.

Form data retrieval on the template

- application contains three files, i.e., app.py, student.html, and result_data.html

```
<html>
<body>
   <p><strong>Thanks for the registration.
    Confirm your details</strong></p>
    <table border = 1>
      {% for key, value in result.items() %}
        <tr>
          <th> {{ key }} </th>
          <td> {{ value }} </td>
        </tr>
      {% endfor %}
    </table>
</body>
</html>
```

Flask Cookies

- The cookies are stored in the form of text files on the client's machine
- Cookies are used to track the user's activities on the web and reflect some suggestions
- In flask, the cookies are associated with the Request object as the dictionary object of all the cookie variables and their values transmitted by the client.
- Flask facilitates us to specify the expiry time, path, and the domain name of the website.

response.setCookie(<title>, <content>, <expiry time>)

read the cookies stored on the client's machine using

request.cookies.get(<title>)

Flask session

- session is similar to cookies but data is stored on the server, instead of client
- session can be defined as the duration for which a user logs into the server and logs out. set the session variable to a specific value on the server.

  session[<variable-name>] = <value>

- To remove a session variable, use the pop() method on the session object and mention the variable to be removed.

  session.pop(<variable-name>, none)

Flask Redirect and Errors

- redirect() function - redirects the user to some specified URL with the specified status code.
- An HTTP status code is a response from the server to the request of the browser.
  - When we visit a website, a request is sent to the server, and the server then responds to the browser's request with a three-digit code: the HTTP status code. This status code also represents the error.

```
Flask.redirect(<location>,<status-code>, <response> )
```

location	It is the URL where the response will be redirected.
status code	It is the status code that is sent to the browser's header along with the response from the server.
response	It is the instance of the response that is used in the project for future requirements.

Flask Redirect and Errors

- application contains files,

i.e., app.py, login.html,

```
<html>
<body>
  <form method = "post" action = "http://localhost:5000/validate">
     <table>
        <tr><td>Email</td><td>
                  <input type = 'email' name = 'email'></td></tr>
        <tr><td>Password</td><td>
                  <input type = 'password' name = 'pass'></td></tr>
        <tr><td><input type = "submit" value = "Submit"></td></tr>
     </table>
  </form>
</body>
</html>
```

Flask Redirect and Errors

- application contains files,

i.e., app.py, login.html

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

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

@app.route('/validate', methods = ["POST"])
def validate():
    if request.method == 'POST' and
                    request.form['pass'] == 'abc':
        return redirect(url_for("success"))
    return redirect(url_for("login"))

@app.route('/success')
def success():
    return "logged in successfully"
if __name__ == '__main__':
   app.run(debug = True)
```

abort() function

- abort() function is used to handle the cases where the errors are involved in the requests from the client side, such as bad requests, unauthorized access and many more.

  Flask.abort(code)

  - 400: for bad requests
  - 401: for unauthorized access
  - 403: for forbidden
  - 404: for not found
  - 406: for not acceptable
  - 415: for unsupported media types
  - 429: for too many requests

Flask Redirect and Errors

- application contains files,

i.e., app.py, login.html

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

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

@app.route('/validate', methods = ["POST"])
def validate():
    if request.method == 'POST' and
                    request.form['pass'] == 'abc':
        return redirect(url_for("success"))
  else:  abort(401)

@app.route('/success')
def success():
    return "logged in successfully"
if __name__ == '__main__':
   app.run(debug = True)
```

Flashing message in Flask

- Flask provides the flash() method, in the same way, the client-side scripting language like JavaScript uses the alerts or the python GUI framework Tkinter uses the dialogue box or the message box.
- flash() method is used to generate informative messages in the flask. It creates a message in one view and renders it to a template view function called next.
  - flash(message, category)
  - **message:** it is the message to be flashed to the user.
  - **Category:** It is an optional parameter. Which may represent any error, information, or warning.

Flashing message in Flask

- get_flashed_messages() method is called in the HTML template to display the passed message by flash(message, category)

  get_flashed_messages(with_categories, category_filter)

  - **with_categories:** This parameter is optional and used if the messages have the category.
  - **category_filter:** This parameter is also optional. It is useful to display only the specified messages.

Flashing message in Flask

- application contains files,

i.e., app.py, login.html

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

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

@app.route('/login',methods = ["GET","POST"])
def login():
   error = None;
   if request.method == "POST":
      if request.form['pass'] != 'jtp':
         error = "invalid password"
      else:
         flash("you are successfuly logged in")
         return redirect(url_for('home'))
   return render_template('login.html',error=error)

if   name   == '   main   ':
```

Flashing message in Flask

- application contains files,

i.e., index.html, login.html, app.py

```
<html>
<body>
   {% with messages = get_flashed_messages() %}
        {% if messages %}
             {% for message in messages %}
                  <p>{{ message }}</p>
             {% endfor %}
        {% endif %}
     {% endwith %}
<h3>Welcome to the website</h3>
<a href = "{{ url_for('login') }}">login</a>
</body>
</html>
```

Flashing message in Flask

- application contains files,

i.e., login.html, index.html, app.py

```
<html>
<body>
  {% if error %}
    <p><strong>Error</strong>: {{error}}</p>
  {% endif %}

  <form method = "post" action = "/login">
    <table>
      <tr><td>Email</td><td><input type = 'email' name = 'email'></td></tr>
      <tr><td>Password</td><td><input type = 'password' name = 'pass'></td></tr>
      <tr><td><input type = "submit" value = "Submit"></td></tr>
    </table>
  </form>
</body>
</html>
```

Flask-SQLite

Flask SQLite

- Flask can make use of the SQLite3 module of the python to create the database web applications
- CRUD (create - read - update - delete) application

Flashing message in Flask

- create a database **Student** and the table students in SQLite using the python script - students.py

```
import sqlite3

con = sqlite3.connect("employee.db")
print("Database opened successfully")

con.execute("create table Students (id INTEGER PRIMARY KEY AUTOINC
REMENT, name TEXT NOT NULL, email TEXT UNIQUE NOT NULL, address TE
XT NOT NULL)")

print("Table created successfully")

con.close()
```

Basic CRUD application in the flask

```python
from flask import *
import sqlite3

app = Flask(__name__)
@app.route("/")
def index():
    return render_template("index.html");
```

```html
<html>
<head>     <title>home</title>  </head>
<body>
   <h2>welcome to the Student Management</h2>
   <a href="/add">Add Student data</a><br><br>
   <a href ="/view">List Records</a><br><br>
   <a href="/delete">Delete Record</a><br><br>
</body>
</html>
```

Basic CRUD application in the flask

Insert new record

Add.html

```
<html>
<head>      <title>Add student </title>  </head>
<body>
<h2>Student Information</h2>
   <form action = "/savedetails" method="post">
   <table>
      <tr><td>Name</td><td><input type="text" name="name"></td></tr>
      <tr><td>Email</td><td><input type="email" name="email"></td></tr>
      <tr><td>Address</td><td><input type="text" name="address"></td></tr>
      <tr><td><input type="submit" value="Submit"></td></tr>
   </table>
   </form>

</body>
</html>
```

```
from flask import *
import sqlite3

app = Flask(__name__)
@app.route("/")
def index():
    return render_template("index.html");

@app.route("/add")
def add():
    return render_template("add.html")

if __name__ == "__main__":
    app.run(debug = True)
```

• Update app.py

```
@app.route("/savedetails",methods = ["POST","GET"])
def saveDetails():
    msg = "msg"
    if request.method == "POST":
        try:
            name = request.form["name"]
            email = request.form["email"]
            address = request.form["address"]
            with sqlite3.connect("student.db") as con:
                cur = con.cursor()
                cur.execute("INSERT into students (name, email, addres
s) values (?,?,?)",(name,email,address))
                con.commit()
                msg = " Student successfully Added "
        except:
            con.rollback()
            msg = "  can not add the student to the list"
        finally:
            return render_template("success.html",msg = msg)
            con.close()
```

- Update app.py

```
@app.route("/savedetails",methods = ["POST","GET"])
def saveDetails():
    msg = "msg"
    if request.method == "POST":
        try:
            name = request.form["name"]
            email = request.form["email"]
            address = request.form["address"]
            with sqlite3.connect("student.db") as con:
                cur = con.cursor()
                cur.execute("INSERT into students (name, email, addres
s) values (?,?,?)",(name,email,address))
                con.commit()
                msg = " Student successfully Added "
        except:
            con.rollback()
            msg = "  can not add the student to the list"
        finally:
            return render_template("success.html",msg = msg)
            con.close()
```

CRUD - View the records in table

```
@app.route("/view")
def view():
    con = sqlite3.connect("student.db")
    con.row_factory = sqlite3.Row
    cur = con.cursor()
    cur.execute("select * from students")
    rows = cur.fetchall()
    return render_template("view.html",rows = rows)
```

CRUD - View the records in table view.html

```
<html>
<head>     <title>List</title>  </head>
<body>
 <h3>student Information</h3>
<table border=5>
   <thead>  <td>ID</td>  <td>Name</td>  <td>Email</td>  <td>Address</td>
   </thead>
     {% for row in rows %}
         <tr>    <td>{{row["id"]}}</td>  <td>{{row["name"]}}</td>
                    <td>{{row["email"]}}</td>   <td>{{row["address"]}}</td>     </tr>
   {% endfor %}
</table>
<br><br>   <a href="/">Go back to home page</a>   </body>
</html>
```

CRUD - View the records in table delete records

```
@app.route("/delete")
def delete():
    return render_template("delete.html")

@app.route("/deleterecord",methods = ["POST"])
def deleterecord():
    id = request.form["id"]
    with sqlite3.connect("student.db") as con:
        try:
            cur = con.cursor()
            cur.execute("delete from students where id = ?",id)
            msg = "record successfully deleted"
        except:
            msg = "can't be deleted"
        finally:
            return render_template("delete_record.html",msg = msg)
```

CRUD - View the records in table delete records

delete.html

```
<html>
<head>
   <title>delete record</title>
</head>
<body>
    <h3>Remove Employee from the list</h3>

     <form action="/deleterecord" method="post">
                    Student Id <input type="text" name="id">
     <input type="submit" value="Submit">
</form>
</body>
</html>
```

CRUD - View the records in table delete records

delete_record.html

```
<html>
<head>
   <title>delete record</title>
</head>
<body>
     <h3>{{msg}}</h3>
     <a href="/view">View List</a>
</body>
</html>
```

Some add-ons

Flask folder structure

- Flask itself is very flexible.
- It has no certain pattern for a project folder structure, which is very good for experienced developers to organize things in their own favors.

```
project/
      __init__.py
      models/
            __init__.py
            base.py
            users.py
            posts.py
            ...
      routes/
            __init__.py
            home.py
            account.py
            dashboard.py
            ...
      templates/
            base.html
            post.html
            ...
      services/
            __init__.py
            google.py
            mail.py
```

Folder Structure

- There is no fixed folder structure.
- Django has seperate init in each folder, in flask its not compulsory
- But here we did same and unify the init process in one

```
# project/__init__.py
from flask import Flask


def create_app()
    from . import models, routes, services
    app = Flask(__name__)
    models.init_app(app)
    routes.init_app(app)
    services.init_app(app)
    return app
```

Creating web application

- The __name__ variable passed to the Flask class is a Python predefined variable, which is set to the name of the module in which it is used.
- The application then imports the routes module,

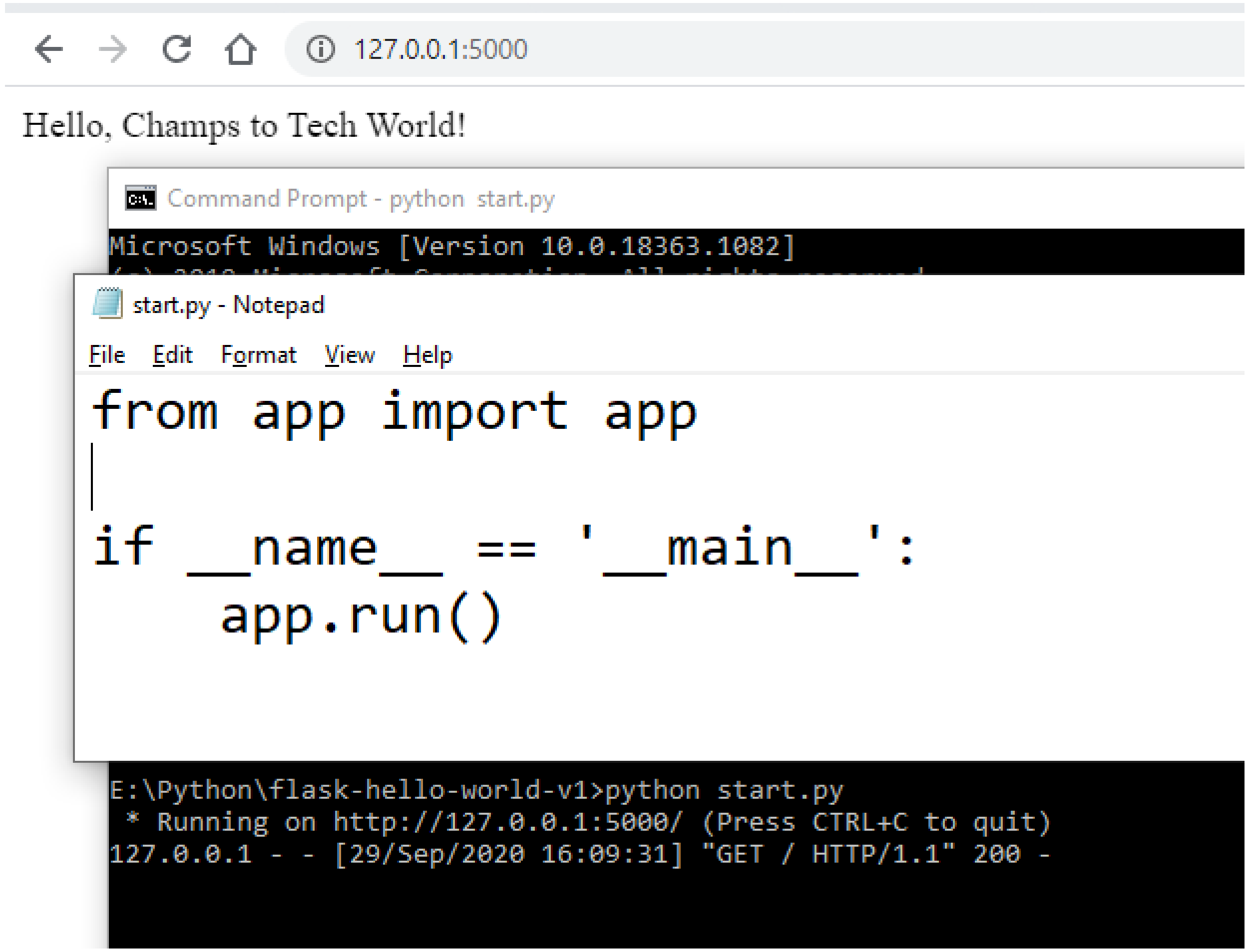
```
app/
    __init__.py
    templates/
    models/
    controllers/
    (or other names)
```

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

- The app package is defined by the app directory and the __init__.py script, and is referenced in the from app import routes statement.
- The app variable is defined as an instance of class Flask in the __init__.py script, which makes it a member of the app package.

```
from app import app

@app.route('/')
@app.route('/index')
def index():
    return "Hello, World!"
```



Hello, Champs to Tech World!

```
Microsoft Windows [Version 10.0.18363.1082]
```

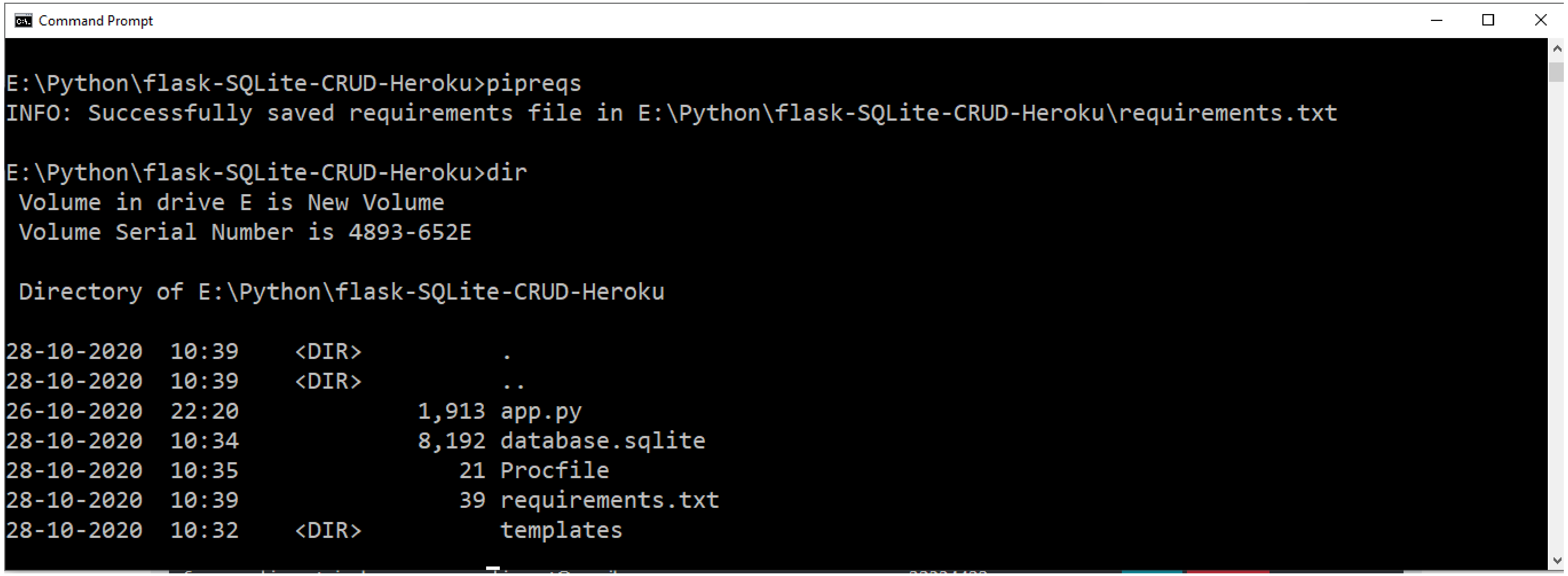
```python
from app import app

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

```
E:\Python\flask-hello-world-v1>python start.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [29/Sep/2020 16:09:31] "GET / HTTP/1.1" 200 -
```

To make requirements file

- pip install pipreqs



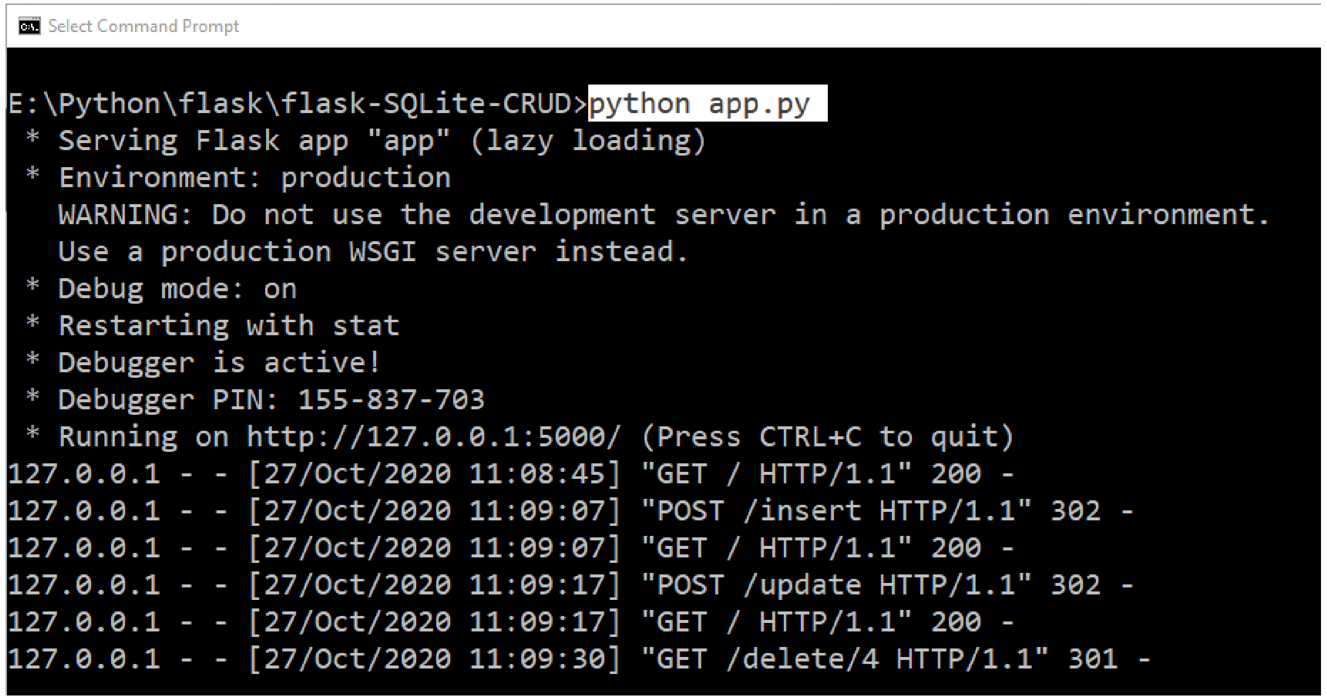
Command Prompt

```
E:\Python\flask-SQLite-CRUD-Heroku>pipreqs
INFO: Successfully saved requirements file in E:\Python\flask-SQLite-CRUD-Heroku\requirements.txt

E:\Python\flask-SQLite-CRUD-Heroku>dir
 Volume in drive E is New Volume
 Volume Serial Number is 4893-652E

 Directory of E:\Python\flask-SQLite-CRUD-Heroku

28-10-2020  10:39    <DIR>          .
28-10-2020  10:39    <DIR>          ..
26-10-2020  22:20             1,913 app.py
28-10-2020  10:34             8,192 database.sqlite
28-10-2020  10:35                21 Procfile
28-10-2020  10:39                39 requirements.txt
28-10-2020  10:32    <DIR>          templates
```


Select Command Prompt

```
E:\Python\flask\flask-SQLite-CRUD>python app.py
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 155-837-703
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [27/Oct/2020 11:08:45] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [27/Oct/2020 11:09:07] "POST /insert HTTP/1.1" 302 -
127.0.0.1 - - [27/Oct/2020 11:09:07] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [27/Oct/2020 11:09:17] "POST /update HTTP/1.1" 302 -
127.0.0.1 - - [27/Oct/2020 11:09:17] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [27/Oct/2020 11:09:30] "GET /delete/4 HTTP/1.1" 301 -
```