PW SKILLS

Lesson Plan

Flask-class-2

Topics to be covered:

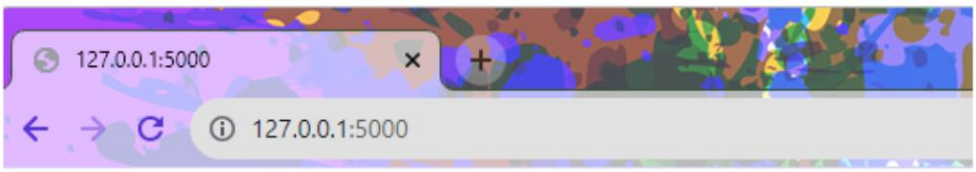
- Routing in Flask
- Views in Flask
- Request in Flask
- Response in Flask
- Templates in Flask
- Styling in Flask
- Database in Flask

Routing in Flask:

The process of mapping URLs (Uniform Resource Locators) to functions in your Flask application is referred to as routing. In other words, it specifies what should occur when a user accesses a given URL. Flask routes are created using the @app.route() decorator.

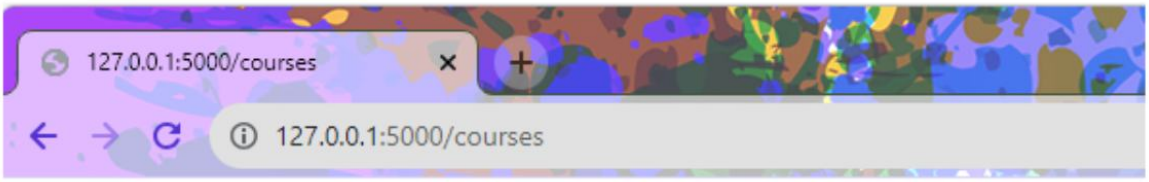
```python
from flask import Flask

# Create an instance of the Flask application
app = Flask(__name__)

# Define a route for the home page
@app.route('/')
def home():
    return 'Welcome to the home page!'

# Define a route for a specific page
@app.route('/courses')
def courses():
    return 'Data Science Courses!'

# Run the application
if __name__ == '__main__':
    app.run(debug=True)
```

URL : `http://127.0.0.1:5000/`



Welcome to the home page!

URL : `http://127.0.0.1:5000/courses/`



We've defined two routes in this example: '/' for the home page and '/courses' for the Course page. The matching messages will appear when you launch the Flask application and visit these URLs in your browser.

Views in Flask:

Views are Python functions that are linked to Flask routes. When a visitor reaches a specific URL, these functions manage the logic of what should be displayed. A view function's return value is normally the material that should be returned to the user's browser.

In the following example, **home(), courses()**, and **template()** are **view functions**.

```
# Import the required Flask modules
from flask import Flask, render_template
```

```
# Create an instance of the Flask application
app = Flask(__name__)

# Define a route for the home page
@app.route('/')
def home():
    return 'Welcome!'

# Define a route for the course page
@app.route('/courses')
def courses():
    return 'Data Science Courses!'

# Define a route for rendering an HTML template
@app.route('/template')
def template():
    return render_template('template.html')

# Run the application
if __name__ == '__main__':
    app.run(debug=True)
```

- Create a new folder named templates in the same directory as app.py.
- Inside the templates folder, create a file named template.html with the following content.

```
<!DOCTYPE html>
<html>
<head>
    <title>Flask Template Example</title>
</head>
<body>
    <h1>Welcome to the Data Science Courses</h1>
    <p>This is an example of using Flask to render an HTML template.</p>
</body>
</html>
```

Your Flask application is now accessible at http://127.0.0.1:5000/. You can view the related responses by visiting various URLs such as /, /courses, and /template. The /template URL renders the template.html template file and displays its content using the render_template() method.

@app.route('/contact', methods=['GET', 'POST']): This decorator is used to define a route for the Contact page and specifies that the view function should handle both GET and POST HTTP methods. By default, routes only handle GET requests.

That's a quick rundown of Flask routing and views. It enables you to effectively handle diverse URLs and design the logic for your online application. Keep in mind that Flask is a robust framework with plenty of additional features and capabilities.

Request in Flask:

The Request object represents an incoming HTTP request made to your Flask application by a client (for example, a web browser). It comprises request-specific information such as the request method (GET, POST, and so on), headers, form data, URL parameters, cookies, and so on. You can use this information to determine how to handle the request in your view functions.

To use the Request object in Flask, you must first import it from the **flask** module. The **request** global variable provides access to the Request object.

```python
from flask import Flask, request

app = Flask(__name__)

@app.route('/courses, methods=['GET', 'POST'])
def courses():
    if request.method == 'POST':
        name = request.form.get('name')
        return f'Course name is , {name}! (via POST)'
    else:
        name = request.args.get('name')
        return f'Course name is, {name}! (via GET)'

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

In this example, we established a route /courses that supports both GET and POST methods. The request.args dictionary is used to get the **name** parameter from the URL when a client makes a GET request to /courses?name=Data Science. If the client submits a POST request with form data containing the **name** field, we get the data using **request.form**.

Response in Flask:

The Response object represents the HTTP response returned to the client by your Flask application after processing a request. It contains data such as the response content, headers, status code, cookies, and other information. You can modify the response to include the information you want to return to the client.

In Flask, you typically return response data directly from your view functions, and Flask generates a Response object based on the returned data.

Let's consider an example of a Response object:

```python
from flask import Flask, make_response

app = Flask(__name__)

@app.route('/technicalCourses)
def technicalCourses():
    response_data = 'There are 100+ technical courses!'
    response = make_response(response_data)
    response.headers['Content-Type'] = 'text/plain'
    response.status_code = 200
    return response

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

When a client reaches the /technicalCourses route in this example, the view function technicalCourses() delivers the response data "There are 100+ technical courses!" To construct a Response object, we utilize the make_response() function. The Content-Type header is then set to 'text/plain' to indicate that the response contains plain text, and the status code is set to 200 to signal a successful response.

View functions in Flask can return a variety of data formats, including texts, JSON objects, HTML, and others. Flask automatically translates the data based on the content type into a suitable Response object.

Overall, Flask's Request and Response objects provide a strong method for handling incoming HTTP requests and generating responses, allowing you to easily develop dynamic web applications.

Templates in Flask:

When constructing web applications using Flask, templates are used to separate the presentation logic (HTML) from the application logic (Python code). Templates enable the creation of dynamic HTML pages by containing placeholders (variables) that are changed with actual data during runtime. The use of CSS (Cascading Style Sheets) in your HTML templates to manage the visual appearance of your web pages is referred to as styling in Flask. Let's go through both topics in depth with a detailed example.

To work with templates in Flask, you normally use a templating engine such as Jinja2, Flask's default templating engine. Jinja2 allows you to include dynamic content in your HTML templates, such as variables and control structures.

- Create a new folder named templates in the same directory as app.py.
- Inside the templates folder, create two files named home.html and courses.html with the following content:

```python
from flask import Flask, render_template

app = Flask(__name__)

# Route for the home page
@app.route('/')
def home():
    return 'Welcome!

# Route for the about page
@app.route('/course')
def courses():
    return render_template('course.html')

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

'home.html'

```
<!DOCTYPE html>
<html>
<head>
    <title>Home</title>
</head>
```

```
<body>
    <h1>Welcome </h1>
    <p>Hello, This is Course home page.</p>
</body>
</html>
```

'courses.html'

```
<!DOCTYPE html>
<html>
<head>
    <title>Courses</title>
</head>
<body>
    <h1>Courses</h1>
    <p>There are multiple Data Science courses!</p>
</body>
</html>
```

Your Flask application is now accessible at http://127.0.0.1:5000/. You can view the related responses by visiting various URLs such as /, /courses, and /template. The /template URL renders the template.html template file and displays its content using the render_template() method.

@app.route('/contact', methods=['GET', 'POST']): This decorator is used to define a route for the Contact page and specifies that the view function should handle both GET and POST HTTP methods. By default, routes only handle GET requests.

That's a quick rundown of Flask routing and views. It enables you to effectively handle diverse URLs and design the logic for your online application. Keep in mind that Flask is a robust framework with plenty of additional features and capabilities.

Request in Flask:

The Request object represents an incoming HTTP request made to your Flask application by a client (for example, a web browser). It comprises request-specific information such as the request method (GET, POST, and so on), headers, form data, URL parameters, cookies, and so on. You can use this information to determine how to handle the request in your view functions.

To use the Request object in Flask, you must first import it from the **flask** module. The **request** global variable provides access to the Request object.

```python
from flask import Flask, request

app = Flask(__name__)

@app.route('/courses, methods=['GET', 'POST'])
def courses():
    if request.method == 'POST':
        name = request.form.get('name')
        return f'Course name is , {name}! (via POST)'
    else:
        name = request.args.get('name')
        return f'Course name is, {name}! (via GET)'

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

In this example, we established a route /courses that supports both GET and POST methods. The request.args dictionary is used to get the name parameter from the URL when a client makes a GET request to /courses?name=Data Science. If the client submits a POST request with form data containing the name field, we get the data using request.form.

Response in Flask:

The Response object represents the HTTP response returned to the client by your Flask application after processing a request. It contains data such as the response content, headers, status code, cookies, and other information. You can modify the response to include the information you want to return to the client.

In Flask, you typically return response data directly from your view functions, and Flask generates a Response object based on the returned data.

Let's consider an example of a Response object:

```python
from flask import Flask, make_response

app = Flask(__name__)

@app.route('/technicalCourses)
def technicalCourses():
    response_data = 'There are 100+ technical courses!'
    response = make_response(response_data)
    response.headers['Content-Type'] = 'text/plain'
    response.status_code = 200
    return response

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

When a client reaches the /technicalCourses route in this example, the view function technicalCourses() delivers the response data "There are 100+ technical courses!" To construct a Response object, we utilize the make_response() function. The Content-Type header is then set to 'text/plain' to indicate that the response contains plain text, and the status code is set to 200 to signal a successful response.

View functions in Flask can return a variety of data formats, including texts, JSON objects, HTML, and others. Flask automatically translates the data based on the content type into a suitable Response object.

Overall, Flask's Request and Response objects provide a strong method for handling incoming HTTP requests and generating responses, allowing you to easily develop dynamic web applications.

Templates in Flask:

When constructing web applications using Flask, templates are used to separate the presentation logic (HTML) from the application logic (Python code). Templates enable the creation of dynamic HTML pages by containing placeholders (variables) that are changed with actual data during runtime. The use of CSS (Cascading Style Sheets) in your HTML templates to manage the visual appearance of your web pages is referred to as styling in Flask. Let's go through both topics in depth with a detailed example.

To work with templates in Flask, you normally use a templating engine such as Jinja2, Flask's default templating engine. Jinja2 allows you to include dynamic content in your HTML templates, such as variables and control structures.

- Create a new folder named templates in the same directory as app.py.
- Inside the templates folder, create two files named home.html and courses.html with the following content:

```
from flask import Flask, render_template

app = Flask(__name__)

# Route for the home page
@app.route('/')
def home():
    return 'Welcome!

# Route for the about page
@app.route('/course')
def courses():
    return render_template('course.html')

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

'home.html'

```
<!DOCTYPE html>
<html>
<head>
    <title>Home</title>
</head>
```

```
<body>
    <h1>Welcome </h1>
    <p>Hello, This is Course home page.</p>
</body>
</html>
```

'courses.html'

```
<!DOCTYPE html>
<html>
<head>
    <title>Courses</title>
</head>
<body>
    <h1>Courses</h1>
    <p>There are multiple Data Science courses!</p>
</body>
</html>
```

Your Flask application is now accessible at http://127.0.0.1:5000/. When you visit the root URL, the home.html template with the content "Hello, This is Course home page." The render_template() function in the home() view function provides the name variable in the template. Similarly, if you go to http://127.0.0.1:5000/course, the courses.html template will be displayed with the content "There are multiple Data Science courses!." The render_template() function in the courses() view function provides the information in the template.

Styling in Flask:

CSS (Cascading Style Sheets) can be used to style your Flask templates. CSS gives you the ability to customize the appearance of HTML elements such as fonts, colors, margins, padding, and layout. CSS styles can be applied inline within HTML tags, used within the <style> tag in your template's <head> section, or linked to an external CSS file.

Here's an easy way to apply CSS styles to the home.html template:

```
            background-color: #f0f0f0;
            margin: 0;
            padding: 0;
        }

        h1 {
            color: #007bff;
        }

        p {
            color: #333;
        }
    </style>
</head>
<body>
    <h1>Welcome</h1>
    <p>Hello, This is Course home page.</p>
</body>
</html>
```

In this example, we used inline CSS within the <style> tag in the <head> section of the home.html template to apply some basic styles to the h1 and p components. For more complex styling, utilize an external CSS file and link it to your templates with the <link> tag.

That's a quick rundown on how to use templates and styling with Flask. Templates allow you to create dynamic HTML pages, while CSS allows you to modify their visual look. Flask, in conjunction with templating engines such as Jinja2, makes it simple to create dynamic and visually appealing online applications.

Database in Flask:

A database can be integrated with Flask to store and retrieve data from your web application. Flask supports a variety of database systems, but for this example, we'll use SQLite, a lightweight, serverless, file-based database.

To work with databases in Flask, you must first install the Flask-SQLAlchemy package. It includes an ORM (Object-Relational Mapping) architecture that lets you communicate with the database through Python classes and objects rather than raw SQL queries.

```
from flask import Flask, render_template, request, redirect, url_for
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
```

```
# Configure the SQLite database
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///data.db'
db = SQLAlchemy(app)

# Create a simple data model for a BlogPost
class BlogPost(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100), nullable=False)
    content = db.Column(db.Text, nullable=False)

    def __repr__(self):
        return f"BlogPost(id={self.id}, title='{self.title}')"

# Create the database tables (only for demonstration purposes, in a
production setting, use database migrations)
db.create_all()

# Route for displaying all blog posts
@app.route('/')
def index():
    posts = BlogPost.query.all()
    return render_template('index.html', posts=posts)
```

```
# Create the database tables (only for demonstration purposes, in a
production setting, use database migrations)
db.create_all()

# Route for displaying all blog posts
@app.route('/')
def index():
    posts = BlogPost.query.all()
    return render_template('index.html', posts=posts)

# Route for adding a new blog post
@app.route('/add', methods=['GET', 'POST'])
def add():
    if request.method == 'POST':
        title = request.form['title']
        content = request.form['content']
        new_post = BlogPost(title=title, content=content)
        db.session.add(new_post)
        db.session.commit()
        return redirect(url_for('index'))
    return render_template('add.html')

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

1. Create a new folder named templates in the same directory as app.py.

2. Inside the templates folder, create two HTML files named index.html and add.html with the following content:

index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Blog Posts</title>
</head>
<body>
    <h1>Blog Posts</h1>
    <ul>
        {% for post in posts %}
            <li>
                <h2>{{ post.title }}</h2>
                <p>{{ post.content }}</p>
            </li>
        {% endfor %}
    </ul>
    <a href="{{ url_for('add') }}">Add a new post</a>
</body>
</html>
```

add.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Add New Post</title>
</head>
<body>
    <h1>Add a New Post</h1>
    <form method="post">
        <label for="title">Title:</label><br>
        <input type="text" id="title" name="title" required><br><br>
        <label for="content">Content:</label><br>
        <textarea id="content" name="content"
required></textarea><br><br>
        <input type="submit" value="Submit">
    </form>
    <a href="{{ url_for('index') }}">Back to Home</a>
</body>
</html>
```

In this example, we've built a simple blog application that allows users to browse existing blog entries and make new ones. SQLAlchemy is used to create a data model BlogPost that represents a blog post with id, title, and content attributes.

The index() view function pulls all blog entries from the database and displays them in the index.html template.

The form submission for adding a new blog post is handled by the add() view function. When the user accepts the form, SQLAlchemy's session is used to build a new BlogPost object, which is then added to the database. The user is returned to the main page after adding a new post.

Make sure to run the db.create_all() method before launching the program to create the database. To handle database schema changes in a production environment, you would normally employ database migrations.

RESTful APIs:

RESTful APIs are a style of creating web services that adhere to the Representational State Transfer (REST) principles. Resources are represented as URLs in a RESTful API, and various HTTP methods are used to conduct CRUD (Create, Read, Update, Delete) activities on these resources. Because of its simplicity and versatility, Flask is a good platform for developing RESTful APIs. Let's look at a real-world example of using Flask to create a simple To-Do List API.

Let's consider a To-Do List API, Let's create a To-Do List API that allows users to manage their tasks. Users should be able to add, retrieve, update, and delete tasks.

Code Example:

1. Install Flask and create a new directory for your project.

2. Create a file named `app.py` and add the following code:

```
from flask import Flask, jsonify, request

app = Flask(__name__)

# Sample data for storing tasks
tasks = [
    {
        'id': 1,
        'title': 'Buy groceries',
        'description': 'Buy milk, eggs, and bread.',
        'done': False
    },
```

```
    data = request.get_json()
    new_task = {
        'id': len(tasks) + 1,
        'title': data['title'],
        'description': data['description'],
        'done': False
    }
    tasks.append(new_task)
    return jsonify(new_task), 201

# Route to update an existing task
@app.route('/tasks/<int:task_id>', methods=['PUT'])
def update_task(task_id):
    task = next((t for t in tasks if t['id'] == task_id), None)
    if task:
        data = request.get_json()
        task['title'] = data['title']
        task['description'] = data['description']
        task['done'] = data['done']
        return jsonify(task)
    else:
```