

# AIML Assignment 3

## 1. What is Flask, and how does it differ from other web frameworks?

A: Flask is a lightweight micro-framework ideal for simple, extensible web apps, while Django is a full-featured framework best for scalable, feature-rich projects. Flask offers flexibility and a “build from scratch” approach, while Django comes with many built-in tools for rapid development.

## 2. Describe the basic structure of a Flask application.

A: Flask – (Creating first simple application)

Building a webpage using python.

There are many modules or frameworks which allow building your webpage using python like a bottle, Django, Flask, etc. But the real popular ones are Flask and Django. Django is easy to use as compared to Flask but Flask provides you with the versatility to program with.

To understand what Flask is you have to understand a few general terms.

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.

jinja2 Jinja2 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 a web application framework written in Python. Flask is based on the Werkzeug WSGI toolkit and Jinja2 template engine. Both are Pocco projects.

Installation:

We will require two packages to set up your environment. virtualenv for a user to create multiple Python environments side-by-side. Thereby, it can avoid compatibility issues between the different versions of the libraries and the next will be Flask itself

### virtualenv

pip install virtualenv

### Create Python virtual environment

virtualenv venv

### Activate virtual environment

```
windows > venv\Scripts\activate
```

```
linux > source ./venv/bin/activate
```

**For windows, if this is your first time running the script, you might get an error like below:**

**venv\Scripts\activate : File C:\flask\_project\venv\Scripts\Activate.ps1 cannot be loaded because running scripts is disabled on this system. For**

**more information, see about\_Execution\_Policies at <https://go.microsoft.com/fwlink/?LinkID=135170>.**

**At line:1 char:1**

**+ venv\Scripts\activate**

**+ FullyQualifiedErrorId : UnauthorizedAccess**

**This means that you don't have access to execute the scripts.**

**To solve this error, run the powershell as admin, when you right click on powershell icon, choose the option 'Run as administrator'. Now, the powershell will open in the admin mode.**

**Type the following command in Shell**

**Now, you will be prompted to change the execution policy. Please type A. This means Yes to all.**

**Flask**

**pip install Flask**

**After completing the installation of the package, let's get our hands on the code.**

**Python3**

```
# Importing flask module in the project is mandatory
# An object of Flask class is our WSGI application.
from flask import Flask

# Flask constructor takes the name of
# current module (__name__) as argument.
app = Flask(__name__)

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

```
@app.route('/')

# '/' URL is bound with hello_world() function.

def hello_world():

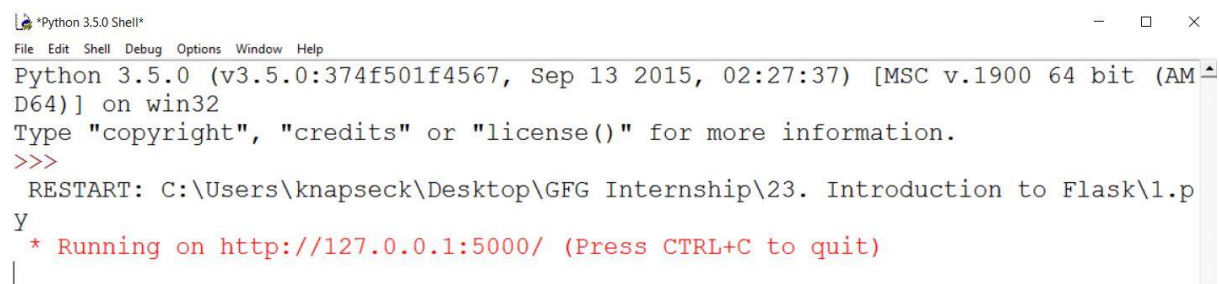
    return 'Hello World'


# main driver function

if __name__ == '__main__':

    app.run()
```

Save it in a file and then run the script we will be getting an output like this.

A screenshot of a Python 3.5.0 Shell window. The window has a title bar that says "\*Python 3.5.0 Shell\*" and a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area shows the following output: "Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:27:37) [MSC v.1900 64 bit (AMD64)] on win32", "Type \"copyright\", \"credits\" or \"license()\" for more information.", ">>>", "RESTART: C:\Users\knapseck\Desktop\GFG Internship\23. Introduction to Flask\1.py", and "\* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)". The cursor is at the end of the last line.

```
*Python 3.5.0 Shell*
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:27:37) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:\Users\knapseck\Desktop\GFG Internship\23. Introduction to Flask\1.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Then go to the URL given there you will see your first webpage displaying hello world there on your local server.

Digging further into the context, the `route()` decorator in Flask is used to bind a URL to a function. Now to extend this functionality our small web app is also equipped with another method `add_url_rule()` which is a function of an application object that is also available to bind a URL with a function as in the above example, `route()` is used.

### 3. How do you install Flask and set up a Flask project?

#### Step 1: Install Virtual Environment

Install Flask in a virtual environment to avoid problems with conflicting libraries. Check Python version before starting:

Python 3 comes with a virtual environment module called `venv` preinstalled. If you have Python 3 installed, skip to Step 2.

Python 2 users must install the `virtualenv` module. If you have Python 2, follow the instructions outlined in Step 1.

Install `virtualenv` on Linux

The package managers on Linux provides `virtualenv`.

For Debian/Ubuntu:

1. Start by opening the Linux terminal.

2. Use `apt` to install `virtualenv` on Debian, Ubuntu and other related distributions:

```
sudo apt install python-virtualenv
```

For CentOS/Fedora/Red Hat:

1. Open the Linux terminal.

2. Use `yum` to install `virtualenv` on CentOS, Red Hat, Fedora and related distributions:

```
sudo yum install python-virtualenv
```

## Install `virtualenv` on MacOS

1. Open the terminal.

2. Install `virtualenv` on Mac using `pip`:

```
sudo python2 -m pip install virtualenv
```

## Install virtualenv on Windows

1. Open the command line with administrator privileges.
2. Use `pip` to install *virtualenv* on Windows:

```
py -2 -m pip install virtualenv
```

Note: To install pip on Windows, follow our [How to install pip on Windows](#) guide.

## Step 2: Create an Environment

1. Make a separate directory for your project:

```
mkdir <project name>
```

2. Move into the directory:

```
cd <project name>
```

3. Within the directory, create the virtual environment for Flask. When you create the environment, a new folder appears in your project directory with the environment's name.

## Create an Environment in Linux and MacOS

- For Python 3:

To create a virtual environment for Python 3, use the *venv* module and give it a name:

```
python3 -m venv <name of environment>
```

- For Python 2:

For Python 2, use the *virtualenv* module to create a virtual environment and name it:

```
python -m virtualenv <name of environment>
```

Listing the directory structure with the [ls command](#) shows the newly created environment:

```
~/myproject $ ls  
venv
```

## Create an Environment in Windows

- For Python 3:

Create and name a virtual environment in Python 3 with:

```
py -3 -m venv <name of environment>
```

- For Python 2:

For Python 2, create the virtual environment with the *virtualenv* module:

```
py -2 -m virtualenv <name of environment>
```

List the folder structure using the `dir` command:

```
dir *<project name>*
```

The project directory shows the newly created environment:

```
C:\Users\crnag\Desktop\test>dir *test*  
Volume in drive C has no label.  
Volume Serial Number is B233-659C  
  
Directory of C:\Users\crnag\Desktop\test  
  
02/03/2021  04:18 PM    <DIR>          vtest  
               0 File(s)                0 bytes  
               1 Dir(s)  113,250,988,032 bytes free
```

## Step 3: Activate the Environment

Activate the virtual environment before installing Flask. The name of the activated environment shows up in the CLI after activation.

## Activate the Environment on Linux and MacOS

Activate the virtual environment in Linux and MacOS with:

```
. <name of environment>/bin/activate
```

```
~/myproject $ . venv/bin/activate  
(venv) ~/myproject $
```

## Activate the Environment on Windows

For Windows, activate the virtual environment with:

```
<name of environment>\Scripts\activate
```

## Step 4: Install Flask

Install Flask within the activated environment using `pip`:

```
pip install Flask
```

Flask is installed automatically with all the dependencies.

Note: `pip` is a Python package manager. To install `pip` follow one of our guides: [How to install pip on CentOS 7](#), [How to install pip on CentOS 8](#), [How to install pip on Debian](#), or [How to install pip on Ubuntu](#).

## Step 5: Test the Development Environment

1. Create a simple Flask application to test the newly created [development environment](#).
2. Make a file in the Flask project folder called *hello.py*.

3. Edit the file using a [text editor](#) and add the following code to make an application that prints "*Hello world!*":

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
    return 'Hello world!'
```

Note: Pick any name for the project except *flask.py*. The Flask library is in a *flask.py* file.

4. Save the file and close.

5. Using the console, navigate to the project folder using the `cd` command.

6. Set the *FLASK\_APP* environment variable.

- **For Linux and Mac:**

```
export FLASK_APP=hello.py
```

- **For Windows:**

```
setx FLASK_APP "hello.py"
```

Note: Windows users must restart the console to set the environment variable. Learn more about setting environment variables by reading one of our guides: [How to set environment variables in Linux](#), [How to set environment variables in MacOS](#), [How to set environment variables in Windows](#).

7. Run the Flask application with:

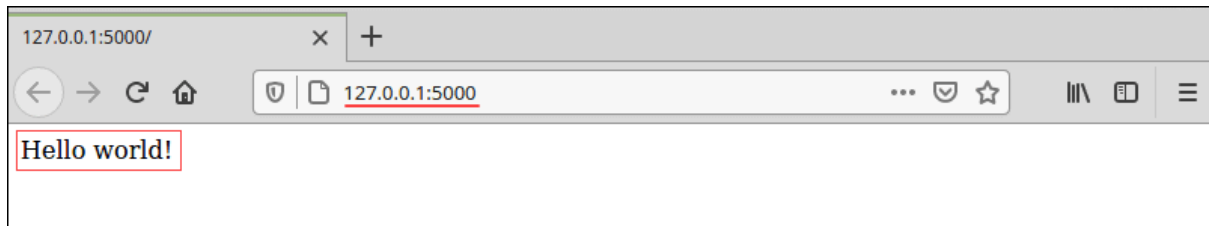
```
flask run
```



```
(venv) ~/myproject $ flask run
* Serving Flask app "hello.py"
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

The output prints out a confirmation message and the address.

8. Copy and paste the address into the browser to see the project running:



4. Explain the concept of routing in Flask and how it maps URLs to Python functions

## Basic Routing in Flask

Routing in Flask determines how incoming requests are handled based on the URL a user has requested. Flask uses the `route()` decorator method of the Flask application instance to define routes and then bind them to appropriate view functions. To demonstrate basic routing in Flask, we start by importing the `Flask` class from the `flask` module:

```
from flask import Flask
```

Once we have the `Flask` class, we can create the application instance and store it in a variable called `app`. The `Flask` class takes in a `__name__` argument, which is a special Python variable denoting the name of the current module containing the Flask application instance:

```
app = Flask(__name__)
```

Using the application instance, we have access to its various methods and decorators, which we can use to define routes, handle requests, and perform other tasks in our web application. However, for this example, we'll be interested in the `route()` decorator, a special method which, when applied to a function in Flask, turns it into a view function that will handle web requests. It takes in a mandatory URL pattern and optional HTTP methods as its arguments. The `route()` decorator enables us to associate a URL pattern with the decorated function, essentially saying that if a user visits the URL defined in the decorator, the function will be triggered to handle this request:

```
@app.route('/')
```

```
def index():  
    return "This is a basic flask application"
```

In the code snippet above, we have the `route()` decorator applied to the `index()` function, meaning that the function will handle requests to the root URL `'/'`. So when a user accesses the URL, Flask will trigger the `index()` function that will return the string "This is a basic Flask application", and it will be displayed in the browser. To ensure the application runs when this module is invoked with Python in the command line, add the `if __name__` check:

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

Putting all the above code snippets into one file gives us the following Flask application:

```
# app.py  
from flask import Flask  
  
app = Flask(__name__)  
  
@app.route('/')  
def index():  
    return "This is a basic flask application"  
  
if __name__ == '__main__':  
    app.run()
```

By using the `route()` decorator, we can define routes for different URLs and map them to the appropriate view functions that will generate the desired response. This allows us to create a structured and organized web application with distinct functionalities for different routes.

## Basic Routing in Flask

**Routing in Flask determines how incoming requests are handled based on the URL a user has requested. Flask uses the `route()` decorator method of the Flask application instance to define routes and then bind them to appropriate view functions. To demonstrate basic routing in Flask, we start by importing the Flask class from the flask module:**

```
from flask import Flask
```

Once we have the Flask class, we can create the application instance and store it in a variable called `app`. The Flask class takes in a `__name__` argument, which is a special Python variable denoting the name of the current module containing the Flask application instance:

```
app = Flask(__name__)
```

Using the application instance, we have access to its various methods and decorators, which we can use to define routes, handle requests, and perform other tasks in our web application. However, for this example, we'll be interested in the `route()` decorator, a special method which, when applied to a function in Flask, turns it into a view function that will handle web requests. It takes in a mandatory URL pattern and optional HTTP methods as its arguments. The `route()` decorator enables us to associate a URL pattern with the decorated function, essentially saying that if a user visits the URL defined in the decorator, the function will be triggered to handle this request:

```
@app.route('/')  
  
def index():  
  
    return "This is a basic flask application"
```

In the code snippet above, we have the `route(/)` decorator applied to the `index()` function, meaning that the function will handle requests to the root URL `'/'`. So when a user accesses the URL, Flask will trigger the `index()` function that will return the string "This is a basic Flask application", and it will be displayed in the browser. To ensure the application runs when this module is invoked with Python in the command line, add the `if __name__` check:

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

Putting all the above code snippets into one file gives us the following Flask application:

```
# app.py  
  
from flask import Flask  
  
  
app = Flask(__name__)  
  
  
@app.route('/')  
  
def index():  
  
    return "This is a basic flask application"
```

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

By using the `route()` decorator, we can define routes for different URLs and map them to the appropriate view functions that will generate the desired response. This allows us to create a structured and organized web application with distinct functionalities for different routes.

**5. What is a template in Flask, and how is it used to generate dynamic HTML content?**

# Templates

You've written the authentication views for your application, but if you're running the server and try to go to any of the URLs, you'll see a `TemplateNotFound` error. That's because the views are calling `render_template()`, but you haven't written the templates yet. The template files will be stored in the `templates` directory inside the `flaskr` package.

Templates are files that contain static data as well as placeholders for dynamic data. A template is rendered with specific data to produce a final document. Flask uses the [Jinja](#) template library to render templates.

In your application, you will use templates to render [HTML](#) which will display in the user's browser. In Flask, Jinja is configured to *autoescape* any data that is rendered in HTML templates. This means that it's safe to render user input; any characters they've entered that could mess with the HTML, such as `<` and `>` will be *escaped* with *safe* values that look the same in the browser but don't cause unwanted effects.

Jinja looks and behaves mostly like Python. Special delimiters are used to distinguish Jinja syntax from the static data in the template. Anything between `{{` and `}}` is an expression that will be output to the final document. `{%` and `%}` denotes a control flow statement like `if` and `for`. Unlike Python, blocks are denoted by start and end tags rather than indentation since static text within a block could change indentation.

## The Base Layout

Each page in the application will have the same basic layout around a different body. Instead of writing the entire HTML structure in each template, each template will *extend* a base template and override specific sections.

`flaskr/templates/base.html`

```
<!doctype html>
<title>{% block title %}{% endblock %} - Flask</title>
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
<nav>
  <h1>Flask</h1>
  <ul>
    {% if g.user %}
      <li><span>{{ g.user['username'] }}</span>
      <li><a href="{{ url_for('auth.logout') }}">Log Out</a>
    {% else %}
      <li><a href="{{ url_for('auth.register') }}">Register</a>
      <li><a href="{{ url_for('auth.login') }}">Log In</a>
```

```

    {% endif %}
</ul>
</nav>
<section class="content">
  <header>
    {% block header %}{% endblock %}
  </header>
  {% for message in get_flashed_messages() %}
    <div class="flash">{{ message }}</div>
  {% endfor %}
  {% block content %}{% endblock %}
</section>

```

`g` is automatically available in templates. Based on if `g.user` is set (from `load_logged_in_user`), either the username and a log out link are displayed, or links to register and log in are displayed. `url_for()` is also automatically available, and is used to generate URLs to views instead of writing them out manually.

After the page title, and before the content, the template loops over each message returned by `get_flashed_messages()`. You used `flash()` in the views to show error messages, and this is the code that will display them.

There are three blocks defined here that will be overridden in the other templates:

1. `{% block title %}` will change the title displayed in the browser's tab and window title.
2. `{% block header %}` is similar to `title` but will change the title displayed on the page.
3. `{% block content %}` is where the content of each page goes, such as the login form or a blog post.

The base template is directly in the `templates` directory. To keep the others organized, the templates for a blueprint will be placed in a directory with the same name as the blueprint.

## Register

`flaskr/templates/auth/register.html`

```

{% extends 'base.html' %}

{% block header %}
  <h1>{% block title %}Register{% endblock %}</h1>
{% endblock %}

{% block content %}
  <form method="post">
    <label for="username">Username</label>
    <input name="username" id="username" required>
    <label for="password">Password</label>
    <input type="password" name="password" id="password" required>
    <input type="submit" value="Register">
  </form>

```

```
{% endblock %}
```

`{% extends 'base.html' %}` tells Jinja that this template should replace the blocks from the base template. All the rendered content must appear inside `{% block %}` tags that override blocks from the base template.

A useful pattern used here is to place `{% block title %}` inside `{% block header %}`. This will set the title block and then output the value of it into the header block, so that both the window and page share the same title without writing it twice.

The `input` tags are using the `required` attribute here. This tells the browser not to submit the form until those fields are filled in. If the user is using an older browser that doesn't support that attribute, or if they are using something besides a browser to make requests, you still want to validate the data in the Flask view. It's important to always fully validate the data on the server, even if the client does some validation as well.

## Log In

This is identical to the register template except for the title and submit button.

flaskr/templates/auth/login.html

```
{% extends 'base.html' %}

{% block header %}
    <h1>{% block title %}Log In{% endblock %}</h1>
{% endblock %}

{% block content %}
    <form method="post">
        <label for="username">Username</label>
        <input name="username" id="username" required>
        <label for="password">Password</label>
        <input type="password" name="password" id="password" required>
        <input type="submit" value="Log In">
    </form>
{% endblock %}
```

## **6. Describe how to pass variables from Flask routes to templates for rendering.**

The web server works (python flask) but when I go to the website, where the value of animal should be (dog) it shows the variable name animal. (There is more to the code but this is the most simplistic version which is the same concept.

Let's say I have these lines of code in my python script running python flask.

```
animal = dog  
return render_template('index.html', value=animal)
```

## **7. How do you retrieve form data submitted by users in a Flask application?**

```
<h3>I like the animal: {{ value }}</h3>
```

```
animal = 'dog'
```

```
return render_template('index.html', value=animal)
```

Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications. It began as a simple wrapper around Werkzeug and Jinja and has become one of the most popular Python web application frameworks.

[Read this article to know more about Flask](#) **Create form as HTML**

We will create a simple HTML Form, very simple Login form

- html

```
<form action="{{ url_for('gfg')}}" method="post">
<label for="firstname">First Name:</label>
<input type="text" id="firstname" name="fname" placeholder="firstname">
<label for="lastname">Last Name:</label>
<input type="text" id="lastname" name="lname" placeholder="lastname">
<button type="submit">Login</button>
```

It's a simple HTML form using the post method. The only thing that is unique is the action URL. `url_for` is a Flask way of creating dynamic URLs where the first argument refers to the function of that specific route in flask. In our form it will create a dynamic route which has `gfg` function in the flask app.

Top VS Code Extensions for Web Developer in 2024!! [GeeksforGeeks](#)

## Create Flask application

Start your virtual environment

```
pip install virtualenv
```

```
python3 -m venv env
```

```
pip install flask
```

Now we will create the flask backend which will get user input from HTML form

- Python3

```
# importing Flask and other modules
from flask import Flask, request, render_template

# Flask constructor
app = Flask(__name__)

# A decorator used to tell the application
# which URL is associated function
@app.route('/', methods = ["GET", "POST"])
def gfg():
    if request.method == "POST":
        # getting input with name = fname in HTML form
        first_name = request.form.get("fname")
        # getting input with name = lname in HTML form
        last_name = request.form.get("lname")
        return "Your name is " + first_name + last_name
    return render_template("form.html")

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

## Working -

Almost everything is simple. We have created a simple Flask app, if we look into code

- importing flask and creating a home route which has both *get and post methods*
- defining a function with name **gfg**
- if requesting method is post, which is the method we specified in the form we get the input data from HTML form



- you can get HTML input from Form using name attribute and request.form.get() function by passing the name of that input as argument
  - request.form.get("fname") will get input from Input value which has name attribute as fname and stores in first\_name variable
  - request.form.get("lname") will get input from Input value which has name attribute as lname and stores in last\_name variable
- The return value of POST method is by replacing the variables with their values Your name is "+first\_name+last\_name
- the default return value for the function gfg id returning home.html template
- you can review [what-does-the-if- name - main -do](#) from the article

## Output – Code in action

```

form.html
1 <form action="{{ url_for('gfg') }}" method="post">
2 <label for="firstname">First Name:</label>
3 <input type="text" id="firstname" name="fname" placeholder="firstname">
4 <label for="lastname">Last Name:</label>
5 <input type="text" id="lastname" name="lname" placeholder="lastname">
6 <button type="submit">Login</button>
7

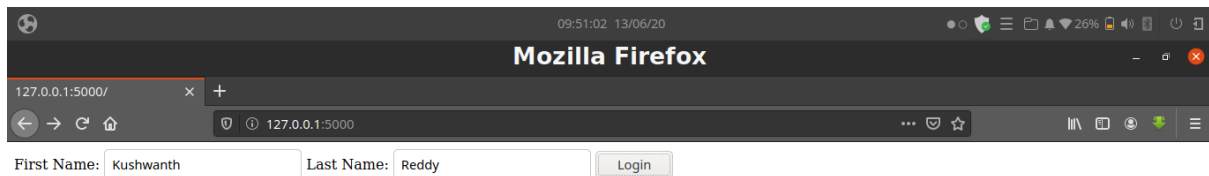
form.py
1 from flask import Flask, request, render_template #importing Flask and other modules
2 app = Flask(__name__) # Flask constructor
3
4 # A decorator used to tell the application
5 # which URL is associated function
6 @app.route('/', methods=["GET", "POST"])
7 def gfg():
8     if request.method == "POST":
9         first_name = request.form.get("fname") #getting input with name=fname in HTML form
10        last_name = request.form.get("lname") #getting input with name=lname in HTML form
11        return "Your name is "+first_name+last_name
12        return render_template("form.html")
13
14 if __name__ == '__main__':
15     app.run()
16
  
```

## flask server running

```

kushwanth@codekushi:~/Downloads/gfg$ python3 form.py
* Serving Flask app "form" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
  
```

## html form



## returning data from html template

### 8. What are Jinja templates, and what advantages do they offer over traditional HTML?

A Jinja template is simply a text file. Jinja can generate any text-based format (HTML, XML, CSV, LaTeX, etc.).

Jinja2:Advantages:

- Jinja2 has a larger community and is more widely used, which means there are more resources available for troubleshooting.

- Jinja2 has a more flexible syntax, which can be useful for more complex use cases.

- Jinja2 has a larger number of plugins available, which can be helpful for adding additional functionality.

Disadvantages:

- Jinja2 can be more difficult to learn and use, especially for beginners.

- Jinja2 has slower performance when dealing with complex templates.

- Jinja2's more flexible syntax can also make it more prone to errors.

Overall, both Mako and Jinja2 have their own advantages and disadvantages, and the choice between them will depend on the specific needs of your project.

### 9.Explain the process of fetching values from templates in Flask and performing arithmetic calculations.

Making a Request with fetch

1. `fetch()` takes two arguments, a URL and an object with other options, and returns a Promise . ...

By default, the GET method is used. ...

To send data, use a data method such as POST, and pass the body option. ...

To send form data, pass a populated `FormData` object.

**10. Discuss some best practices for organizing and structuring a Flask project to maintain scalability and readability.**

Build a Scalable Flask Web Project From Scratch

- Prerequisites.
- Project Overview.
- Get Started. Create a Virtual Environment. ...
- Initiate Your Flask Project. Run the Flask Development Server. ...
- Leverage Blueprints.
- Introduce Templates. Build a Base Template. ...
- Improve the User Experience. Include a Navigation Menu. ...
- Conclusion.

2. **Promise . ...**
3. **By default, the GET method is used. ...**
4. **To send data, use a data method such as POST, and pass the body option. ...**
5. **To send form data, pass a populated FormData object.**