FLASK for MACHINE LEARNING

Ayush Raj

Uplift ML Batch 1

Introduction to Web Dev (Flask)

- **Front End Development:** The part of a website that the user interacts with directly is termed the front end.

  HTML => provides structure to front end
  CSS => beautifies the front end
  Javascript => provides additional features

- **Backend Development:** Backend is the server-side of the website. It stores and arranges data, and also makes sure everything on the client-side of the website works fine.

  JavaScript , Python,Java etc

Introduction to Flask

- Flask is *micro-framework* . It only provides the necessary components for web development, such as **routing, request handling, sessions,** and so on. Other features like ORM can be either coded or implemented by using extensions.
- Flask uses Jinja2 , a template rendering engine, to render **HTML** which will display in the user's browser.
- Since flask uses Python and Python is not a native language to web development , so a (WSGI) server implements the web server side of the WSGI interface for running Python web applications.

Hello World App

- # app.py

```
from flask import Flask                    # import flask
Myapp = Flask(__name__)          # create an app instance

@Myapp.route("/")                          # at the end point /
def hello():                               # call method hello
    return "Hello World!"
        # which returns "hello world"
if __name__ == "__main__":       # on running python app.py
    Myapp.run(debug=True)        # run the flask app
```

- By default, flask runs a local server at port 5000.

Routing

- @Myapp.route("/") is a Python decorator that Flask provides to assign URLs in our app to functions easily.

  A **decorator** is a design pattern in **Python** that allows a user to add new functionality to an existing object without modifying its structure and are usually called before the definition of a function you want to decorate.

- When defining our route, values within carrot brackets <> indicate a variable; this enables routes to be dynamically generated.

```
@Myapp.route('/user/<username>')
def profile(username):
        #... Logic goes here


@Myapp.route('/<int:year>/<int:month>/<title>')
def article(year, month, title):
        #... Logic goes here
```

- Routes can accept string, int, float and path type of variables.

View Responses

- To render a Jinja2 page template, we first must import a built-in Flask function called render_template(). When a view function returns render_template(), it's telling Flask to serve an HTML page to the user which we generate via a Jinja template.

```
from flask import Flask, render_template
newapp = Flask(__name__, template_folder="Mytemplates")
@newapp.route("/")
def home():
    """Serve homepage template."""
    return render_template("index.html")
```

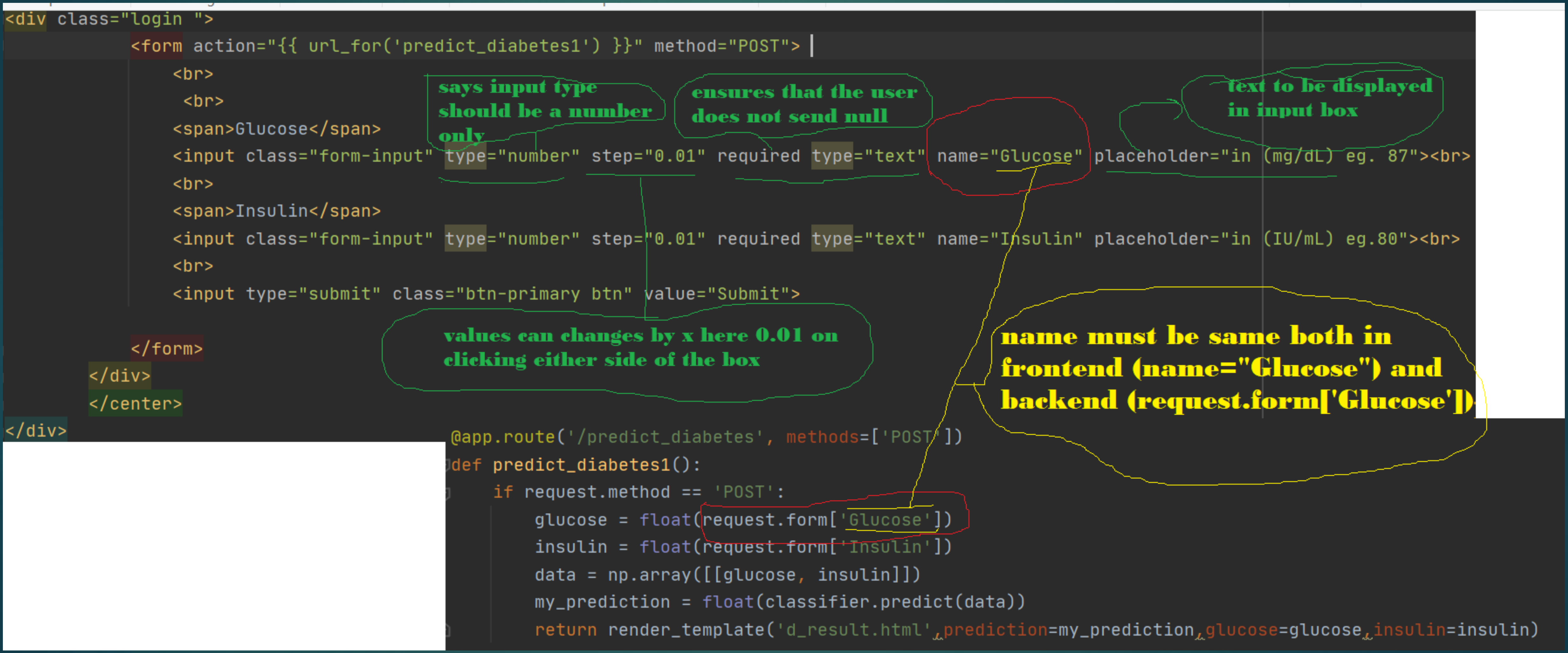
The Request Object

- request() is a of the "global“object. It's available to every route and contains all the context of a request made to the said route.
- So, to request a response from the server, there are mainly two methods:

GET : to request data from the server

POST : to submit data to be processed to the server.

Communicating from FrontEnd(HTML element) to BackEnd (Python Flask)



```
<div class="login ">
        <form action="{{ url_for('predict_diabetes1') }}" method="POST">
            <br>
             <br>
            <span>Glucose</span>
            <input class="form-input" type="number" step="0.01" required type="text" name="Glucose" placeholder="in (mg/dL) eg. 87"><br>
            <br>
            <span>Insulin</span>
            <input class="form-input" type="number" step="0.01" required type="text" name="Insulin" placeholder="in (IU/mL) eg.80"><br>
            <br>
            <input type="submit" class="btn-primary btn" value="Submit">

        </form>
    </div>
    </center>
</div>
```

says input type should be a number only

ensures that the user does not send null

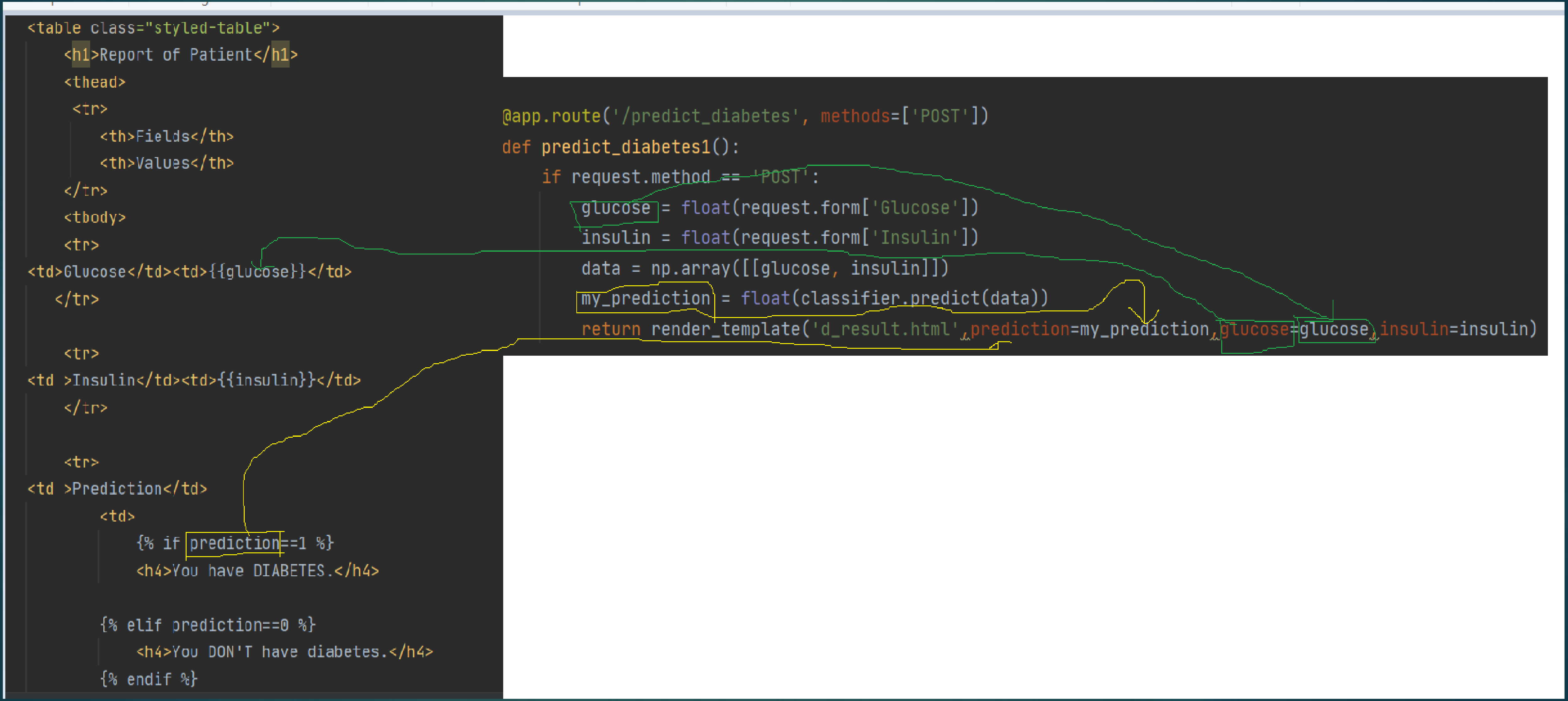
text to be displayed in input box

values can changes by x here 0.01 on clicking either side of the box

name must be same both in frontend (name="Glucose") and backend (request.form['Glucose'])

```
@app.route('/predict_diabetes', methods=['POST'])
def predict_diabetes1():
    if request.method == 'POST':
        glucose = float(request.form['Glucose'])
        insulin = float(request.form['Insulin'])
        data = np.array([[glucose, insulin]])
        my_prediction = float(classifier.predict(data))
        return render_template('d_result.html',prediction=my_prediction,glucose=glucose,insulin=insulin)
```

Communicating from BackEnd (Flask python variable)to FrontEnd (Jinja 2 variable in HTML File)



```
<table class="styled-table">
    <h1>Report of Patient</h1>
    <thead>
     <tr>
        <th>Fields</th>
        <th>Values</th>
    </tr>
    <tbody>
    <tr>
<td>Glucose</td><td>{{glucose}}</td>
   </tr>

    <tr>
<td >Insulin</td><td>{{insulin}}</td>
    </tr>

    <tr>
<td >Prediction</td>
        <td>
            {% if prediction==1 %}
            <h4>You have DIABETES.</h4>

        {% elif prediction==0 %}
            <h4>You DON'T have diabetes.</h4>
        {% endif %}
```

```
@app.route('/predict_diabetes', methods=['POST'])
def predict_diabetes1():
    if request.method == 'POST':
        glucose = float(request.form['Glucose'])
        insulin = float(request.form['Insulin'])
        data = np.array([[glucose, insulin]])
        my_prediction = float(classifier.predict(data))
        return render_template('d_result.html',prediction=my_prediction,glucose=glucose,insulin=insulin)
```

Front End Back End Communication

- All variables in python and jinja2 are case sensitive.
- href/action="{{ url_for('predict_diabetes1') }}“ looks for the url of corresponding to this function i.e. 'predict_diabetes1, url found is ‘/predict_diabetes’
- Routing url name can differ from function name that follows it.

Jinja 2

- {% ... %} for Statements

```
 {% for n in my_list %}
 <li>{{n}}</li>
{% endfor %}



{% if prediction==1 %}
		<h4>You have DIABETES.</h4>
{% elif prediction==0 %}
		<h4>You DON'T have diabetes.</h4>
{% endif %}
```

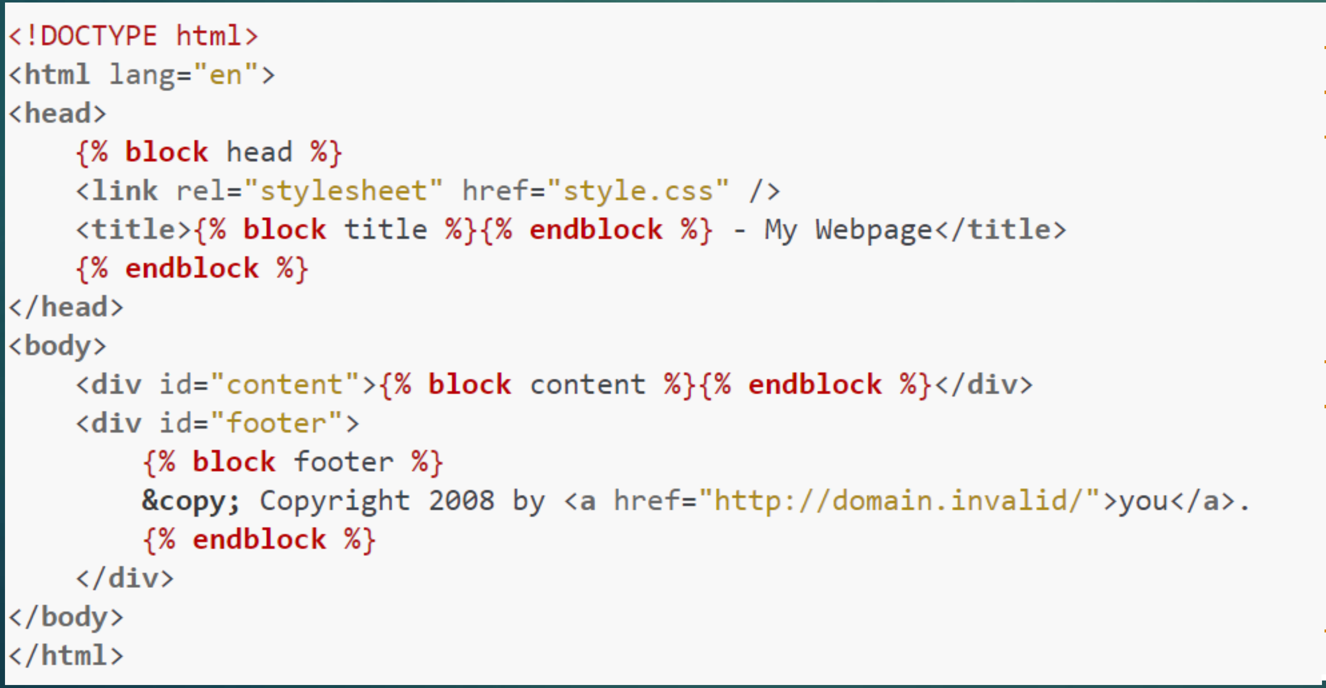
- {{ ... }} for Expressions to print to the template output

```
{{variable_name}}
```

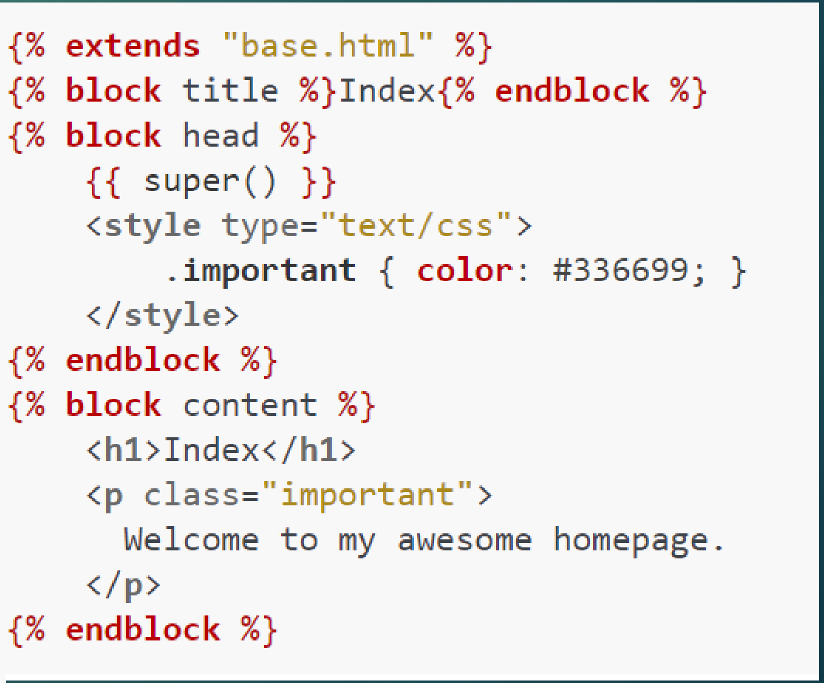
- {# ... #} for Comments not included in the template output

Jinja 2 Template Inheritance

- Template inheritance allows you to build a base “skeleton” template that contains all the common elements of your site and defines **blocks** that child templates can override.
- Let base.html (left one), defines a simple HTML skeleton document to be used for many pages.
- The {% extends %} tag is the key here. It tells the template engine that this template “extends” another template. When the template system evaluates this template, it first locates the parent. The extends tag should be the first tag in the template



```
<!DOCTYPE html>
<html lang="en">
<head>
    {% block head %}
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}{% endblock %} - My Webpage</title>
    {% endblock %}
</head>
<body>
    <div id="content">{% block content %}{% endblock %}</div>
    <div id="footer">
        {% block footer %}
        &copy; Copyright 2008 by <a href="http://domain.invalid/">you</a>.
        {% endblock %}
    </div>
</body>
</html>
```



```
{% extends "base.html" %}
{% block title %}Index{% endblock %}
{% block head %}
    {{ super() }}
    <style type="text/css">
        .important { color: #336699; }
    </style>
{% endblock %}
{% block content %}
    <h1>Index</h1>
    <p class="important">
      Welcome to my awesome homepage.
    </p>
{% endblock %}
```