Python Backend Training (Flask)

Python Syntax and Basic Data Structures

1. Introduction to Python Syntax

- **Definition**: Python is a high-level, interpreted programming language known for its readability and ease of use.
- **Features**:
  - Simple syntax similar to English
  - Dynamically typed (no need to declare variable types)
  - Interpreted language (executed line by line)
- **Example**:

```
name = "Alice"
age = 25
print(f"Hello, my name is {name} and I am {age} years old.")
```

- **Real-world Usage**: Used in web development, data science, automation, and AI.

2. Variables

- **Definition**: Variables are used to store data in memory for later use.
- **Syntax**:

```
variable_name = value
```

- **Example**:

```
name = "Alice"
age = 25
```

- **Rules**:
  - Variable names must start with a letter or underscore (`_`), followed by letters, numbers, or underscores.
  - Python is case-sensitive (`age` and `Age` are different).

3. Data Types

- **Definition**: Python has various built-in data types that define the type of data a variable can store.
- **Common Data Types**:
  - **String (`str`)**: Text data.

    ```
    name = "Alice"
    ```

  - **Integer (`int`)**: Whole numbers.

    ```
    age = 25
    ```

  - **Float (`float`)**: Decimal numbers.

    ```
    height = 5.8
    ```

  - **Boolean (`bool`)**: Represents `True` or `False`.

    ```
    is_active = True
    ```

- **None**: Represents the absence of a value.

```
result = None
```

4. Lists

- **Definition**: A list is an ordered, mutable collection of elements.
- **Syntax**:

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])  # Output: apple
```

- **Operations**:
  - Append, remove, sort, iterate through elements
- **Real-world Usage**: Used in data storage, queue processing, and managing collections.

4. Tuples

- **Definition**: A tuple is an ordered, immutable collection of elements.
- **Syntax**:

```
coordinates = (10.5, 20.7)
print(coordinates[1])  # Output: 20.7
```

- **Operations**:
  - Access elements, unpacking, nesting tuples
- **Real-world Usage**: Used for fixed collections like GPS coordinates or configuration settings.

5. Dictionaries

- **Definition**: A dictionary is an unordered collection of key-value pairs.

- **Syntax**:

```
person = {"name": "Alice", "age": 25}
print(person["name"])  # Output: Alice
```

- **Operations**:
  - Adding/removing key-value pairs, iterating, updating values
- **Real-world Usage**: Used in JSON data handling, configuration files, and database queries.

Functions, Modules, and Libraries in Python

1. Functions

- **Definition**: A function is a reusable block of code that performs a specific task.
- **Syntax**:

```
def greet(name):
    return f"Hello, {name}!"
print(greet("Alice"))  # Output: Hello, Alice!
```

- **Types**:
  - Built-in functions (`print()`, `len()`, `range()`)
  - User-defined functions
  - Lambda functions (`lambda x: x * 2`)
- **Real-world Usage**: Used in automation, API calls, and mathematical calculations.

2. Modules

- **Definition**: A module is a file containing Python code (functions, classes, variables) that can be imported.

- **Example**:

```
import math
print(math.sqrt(16))  # Output: 4.0
```

- **Real-world Usage**: Used to organize large projects into reusable components.

3. Libraries

- **Definition**: A library is a collection of modules that provide additional functionality.
- **Popular Libraries**:
  - `requests` for HTTP requests
  - `numpy` for numerical computing
  - `pandas` for data analysis
- **Example**:

```
import requests
response = requests.get("https://api.github.com")
print(response.status_code)  # Output: 200
```

- **Real-world Usage**: Used in data science, web scraping, and machine learning.

Object-Oriented Programming (OOP) in Python

1. Introduction to OOP

- **Definition**: OOP is a programming paradigm based on objects and classes.
- **Key Concepts**:
  - **Class**: Blueprint for objects
  - **Object**: Instance of a class
  - **Methods**: Functions defined inside a class

- **Attributes**: Variables stored in an object

2. Creating a Class and Object

- **Example**:

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
    def display(self):
        return f"Car: {self.brand} {self.model}"
my_car = Car("Toyota", "Corolla")
print(my_car.display())
```

- **Real-world Usage**: Used in game development, GUI applications, and real-world simulations.

3. Inheritance and Polymorphism

- **Definition**: Inheritance allows one class to inherit methods and attributes from another class.
- **Example**:

```
class ElectricCar(Car):
    def __init__(self, brand, model, battery_size):
        super().__init__(brand, model)
        self.battery_size = battery_size
my_tesla = ElectricCar("Tesla", "Model S", "100 kWh")
print(my_tesla.display())  # Output: Car: Tesla Model S
```

- **Real-world Usage**: Used in frameworks like Django, where models inherit properties.

4. Encapsulation and Abstraction

- **Encapsulation**: Restricting direct access to object data.

- **Abstraction**: Hiding complex details and exposing only necessary functionality.
- **Example**:

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute
    def deposit(self, amount):
        self.__balance += amount
    def get_balance(self):
        return self.__balance
account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # Output: 1500
```

- **Real-world Usage**: Used in banking systems, where user data needs protection.

Flask Backend Development - Step-by-Step Guide

1. Setting up Flask Environment

Step 1: Install Python

- Ensure Python (3.x) is installed.
- Check version:

```
python --version
```

- Download from Python Official Site.

Step 2: Create a Virtual Environment

- Navigate to your project folder:

```
mkdir flask_project && cd flask_project
```

- Create a virtual environment:

```
python -m venv venv
```

- Activate the virtual environment:
  - **Windows:**

```
venv\Scripts\activate
```

  - **Mac/Linux:**

```
source venv/bin/activate
```

Step 3: Install Flask

```
pip install flask
```

Step 4: Verify Installation

```
python -m flask --version
```

---

2. Flask Routing (GET, POST, PUT, DELETE)

Understanding Routes in Flask

- A route maps a URL to a function.

Step 1: Create `app.py` and Define Routes

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/')
def home():
    return "Welcome to Flask!"

@app.route('/hello/<name>')
def hello(name):
    return f"Hello, {name}!"

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

Step 2: Running the Flask App

```
python app.py
```

- Open `http://127.0.0.1:5000/` in a browser.

Step 3: Handling GET and POST Requests

```
@app.route('/data', methods=['GET', 'POST'])
def handle_data():
    if request.method == 'POST':
        data = request.json
        return {"message": "Data received", "data": data}
    return {"message": "Send a POST request with JSON data"}
```

Step 4: Implementing PUT and DELETE

```
@app.route('/update/<int:item_id>', methods=['PUT'])
def update_item(item_id):
```

```
    return {"message": f"Item {item_id} updated"}

@app.route('/delete/<int:item_id>', methods=['DELETE'])
def delete_item(item_id):
    return {"message": f"Item {item_id} deleted"}
```

---

3. Rendering Templates with Jinja2

Step 1: Create a `templates/` Folder

```
mkdir templates
```

Step 2: Create `index.html` Inside `templates/`

```
<!DOCTYPE html>
<html>
<head>
    <title>Flask Template</title>
</head>
<body>
    <h1>Hello, {{ name }}!</h1>
</body>
</html>
```

Step 3: Modify `app.py` to Render Templates

```
from flask import render_template

@app.route('/welcome/<name>')
def welcome(name):
    return render_template('index.html', name=name)
```

Step 4: Run and Test

- Start Flask and visit:

  ```
  http://127.0.0.1:5000/welcome/John
  ```

- The browser should display:

  ```
  Hello, John!
  ```

---

4. Handling Forms and User Input in Flask

Step 1: Install Flask-WTF

```
pip install flask-wtf
```

Step 2: Create a Form in `forms.py`

```
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField
from wtforms.validators import DataRequired

class NameForm(FlaskForm):
    name = StringField('Enter your name', validators=[DataRequired()])
    submit = SubmitField('Submit')
```

Step 3: Modify `app.py` to Handle Forms

```
from flask import Flask, render_template, request
from forms import NameForm

app = Flask(__name__)
app.config['SECRET_KEY'] = 'your_secret_key'

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

```
def form():
    form = NameForm()
    if form.validate_on_submit():
        return f"Hello, {form.name.data}!"
    return render_template('form.html', form=form)
```

---

5. Flask Blueprints for App Structure

Step 1: Create a `blueprints/` Directory

```
mkdir blueprints
```

Step 2: Define a Blueprint in `blueprints/sample.py`

```
from flask import Blueprint

sample_bp = Blueprint('sample', __name__)

@sample_bp.route('/sample')
def sample():
    return "This is a sample blueprint route."
```

Step 3: Register Blueprint in `app.py`

```
from blueprints.sample import sample_bp
app.register_blueprint(sample_bp, url_prefix='/bp')
```

---

6. Flask Debug Mode and Error Handling

Step 1: Enable Debug Mode

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

Step 2: Handle Errors Gracefully

```
@app.errorhandler(404)
def not_found(error):
    return "Page Not Found", 404

@app.errorhandler(500)
def internal_error(error):
    return "Internal Server Error", 500
```

Step 3: Run the Application

```
python app.py
```

- Test invalid URLs and observe error handling.

Flask RESTful API - Step-by-Step Guide

1. Introduction to RESTful APIs

What is a RESTful API?

- REST (Representational State Transfer) is an architectural style for designing networked applications.
- Uses HTTP methods (GET, POST, PUT, DELETE) for communication.
- Stateless, meaning each request contains all necessary information.
- Commonly returns data in JSON format.

Why Use RESTful APIs in Flask?

- Simplifies client-server communication.
- Allows easy integration with frontend applications or mobile apps.
- Supports structured and scalable backend development.

---

2. Flask-RESTful for API Endpoints

Step 1: Install Flask-RESTful

```
pip install flask-restful
```

Step 2: Create `app.py` and Define a Basic API

```
from flask import Flask
from flask_restful import Api, Resource

app = Flask(__name__)
api = Api(app)

class HelloWorld(Resource):
    def get(self):
        return {"message": "Hello, World!"}

api.add_resource(HelloWorld, '/')

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

Step 3: Run the API

```
python app.py
```

- Open `http://127.0.0.1:5000/` in a browser or use Postman.

3. Request Handling (GET, POST, PUT, DELETE)

Step 1: Define an API with Multiple HTTP Methods

```
from flask import request

class User(Resource):
    users = {}

    def get(self, user_id):
        return {"user_id": user_id, "data": self.users.get(user_id, "Not found")}

    def post(self, user_id):
        self.users[user_id] = request.json
        return {"message": "User created", "user_id": user_id, "data": self.users
[user_id]}

    def put(self, user_id):
        if user_id in self.users:
            self.users[user_id] = request.json
            return {"message": "User updated", "user_id": user_id, "data": self.use
rs[user_id]}
        return {"message": "User not found"}, 404

    def delete(self, user_id):
        if user_id in self.users:
            del self.users[user_id]
            return {"message": "User deleted"}
        return {"message": "User not found"}, 404

api.add_resource(User, '/user/<int:user_id>')
```

Step 2: Testing the API

- **GET** request: `http://127.0.0.1:5000/user/1`

- **POST** request with JSON data:

```
{
  "name": "John Doe",
  "email": "john@example.com"
}
```

- **PUT** request to update user data.
- **DELETE** request to remove a user.

---

4. Running and Debugging the API

- Use Postman or cURL to test the API endpoints.
- Enable `debug=True` in `app.run()` to see real-time errors.
- Ensure `Content-Type: application/json` is set for POST and PUT requests.

This completes the guide to building RESTful APIs using Flask-RESTful!

Flask Security and Authentication - Step-by-Step Guide

1. JSON Response and Data Serialization

What is JSON Serialization?

- JSON (JavaScript Object Notation) is a lightweight data format used for communication between the server and client.
- Serialization converts Python objects (like dictionaries and lists) into JSON format.

Step 1: Returning JSON Responses in Flask

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/json')
def json_response():
    data = {"message": "Hello, World!", "status": "success"}
    return jsonify(data)

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

Step 2: Using Marshmallow for Advanced Serialization

```
pip install flask-marshmallow marshmallow
```

```
from flask_marshmallow import Marshmallow

ma = Marshmallow(app)

class UserSchema(ma.Schema):
    class Meta:
        fields = ("id", "name", "email")

user_schema = UserSchema()
users_schema = UserSchema(many=True)

@app.route('/user')
def get_user():
    user = {"id": 1, "name": "John Doe", "email": "john@example.com"}
    return user_schema.jsonify(user)
```

2. Authentication & Authorization with JWT Tokens

What is JWT?

- JSON Web Tokens (JWT) are used for authentication and authorization.
- A token is issued upon login and verified on each request.

Step 1: Install Flask-JWT-Extended

```
pip install flask-jwt-extended
```

Step 2: Implement JWT Authentication

```
from flask import request
from flask_jwt_extended import JWTManager, create_access_token, jwt_requi
red, get_jwt_identity

app.config['JWT_SECRET_KEY'] = 'your_secret_key'
jwt = JWTManager(app)

users = {"admin": "password123"}

@app.route('/login', methods=['POST'])
def login():
    data = request.json
    username = data.get("username")
    password = data.get("password")

    if users.get(username) == password:
        access_token = create_access_token(identity=username)
        return jsonify(access_token=access_token)
    return jsonify({"error": "Invalid credentials"}), 401

@app.route('/protected', methods=['GET'])
@jwt_required()
```

```
def protected():
    current_user = get_jwt_identity()
    return jsonify({"message": f"Welcome {current_user}!"})
```

Step 3: Testing JWT Authentication

1. Send a **POST** request to `/login` with JSON body:

```
{
  "username": "admin",
  "password": "password123"
}
```

2. Use the returned `access_token` in the **Authorization Header** to access `/protected`:

```
Authorization: Bearer <access_token>
```

---

3. Flask-Security for User Management

Step 1: Install Flask-Security

```
pip install flask-security-too
```

Step 2: Configure User Authentication

```
from flask_security import Security, SQLAlchemyUserDatastore, UserMixin, R
oleMixin
from flask_sqlalchemy import SQLAlchemy

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///users.db'
app.config['SECURITY_PASSWORD_SALT'] = 'random_salt'
db = SQLAlchemy(app)

class Role(db.Model, RoleMixin):
```

```
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), unique=True)

class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(120), unique=True)
    password = db.Column(db.String(255))
    active = db.Column(db.Boolean)

user_datastore = SQLAlchemyUserDatastore(db, User, Role)
security = Security(app, user_datastore)

db.create_all()
```

Step 3: Registering and Logging in Users

- Flask-Security automatically provides login, logout, and registration routes.
- Access `http://127.0.0.1:5000/login` to log in users.

This completes the guide to Flask security, authentication, and user management! 🚀