Phase 1: Basic Python Programming

1. Setting Up the Environment

- **Instructions**:
  - **Install Python**:
    - Download the Python version 3.10 from python.org.
    - Follow the installation instructions and ensure to check "Add Python to PATH" during installation.
  - **Choose an IDE**:
    - Install an IDE like Visual Studio Code or PyCharm.
    - Set up a new project folder for the exercises.
  - **Create a Virtual Environment**:
    - Open a terminal and navigate to your project folder.
    - Run the following command to create a virtual environment:

```
python -m venv myenv
```

    - Activate the virtual environment:
      - On Windows:

```
myenv\Scripts\activate
```

      - On macOS/Linux:

```
source myenv/bin/activate
```

2. Basic Python Syntax and Data Types

- **Instructions**:
  - **Hello World**:
    - Create a file named `hello.py` and write the following code:

```
print("Hello, World!")
```

    - Run the script in the terminal:

```
python hello.py
```

- **Data Types**:
  - Write scripts to explore data types:

```
# Numbers
a = 5         # Integer
b = 3.14      # Float
print(type(a), type(b))

# Strings
name = "Alice"
print(name)

# Lists
fruits = ["apple", "banana", "cherry"]
print(fruits[0])

# Tuples
point = (2, 3)
print(point)

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

3. Control Structures

- **Instructions**:
  - **Conditionals**:
    - Write a script that checks if a number is even or odd:

```
num = int(input("Enter a number: "))
if num % 2 == 0:
    print("Even")
else:
    print("Odd")
```

  - **Loops**:
    - Create a loop to print numbers from 1 to 10:

```
for i in range(1, 11):
    print(i)
```

  - **Number Guessing Game**:

- Create a simple number guessing game:

```
import random

secret_number = random.randint(1, 100)
guess = None

while guess != secret_number:
    guess = int(input("Guess the number (1-100): "))
    if guess < secret_number:
        print("Too low!")
    elif guess > secret_number:
        print("Too high!")
    else:
        print("Correct!")
```

4. Functions and Modules

- **Instructions**:
  - **Defining Functions**:
    - Write a function to calculate the factorial of a number:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))  # Output: 120
```

  - **Creating a Module**:
    - Create a file named `utils.py` and add utility functions:

```
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y
```

    - In another script, import and use these functions:

```
from utils import add, subtract
```

```
print(add(10, 5))  # Output: 15
```

5. Basic File Handling

- **Instructions**:
  - **Reading and Writing Files**:
    - Write a script to read and write to a text file:

```
# Writing to a file
with open("example.txt", "w") as file:
    file.write("Hello, World!")

# Reading from a file
with open("example.txt", "r") as file:
    content = file.read()
    print(content)  # Output: Hello, World!
```

Phase 2: Introduction to Web Development

6. Understanding Web Basics

- **Instructions**:
  - **Learn HTTP**:
    - Read about the basics of HTTP requests and responses.
    - Explore common HTTP methods: GET, POST, PUT, DELETE.
  - **RESTful APIs**:
    - Understand REST principles, focusing on resources, endpoints, and HTTP methods.

7. Setting Up a Simple Web Server

- **Instructions**:
  - **Install Flask**:
    - In the terminal, run:

```
pip install Flask
```

  - **Create a Basic Flask App**:
    - Create a file named `app.py`:

```
from flask import Flask
```

```
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello, World!"

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

- Run the Flask app:

```
python app.py
```

- Open a browser and go to `http://127.0.0.1:5000/` to see the message.

8. Routing and Views

- **Instructions**:
  - **Multiple Routes**:
    - Extend `app.py` with additional routes:

```
@app.route("/about")
def about():
    return "This is the about page."
```

  - **Using Templates**:
    - Install Jinja2 (comes with Flask) and create an `index.html` file:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Home</title>
  </head>
  <body>
    <h1>Welcome to My Flask App</h1>
  </body>
</html>
```

    - Modify `app.py` to render the template:

```
from flask import render_template
```

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

Phase 3: Building a Simple Backend Application

9. Building RESTful APIs with Flask

- **Instructions**:
  - **Create a Simple CRUD API**:
    - Create a `books` resource:

```
books = []

@app.route("/books", methods=["GET"])
def get_books():
    return {"books": books}

@app.route("/books", methods=["POST"])
def add_book():
    new_book = request.json
    books.append(new_book)
    return new_book, 201
```

10. Data Storage with SQLite

- **Instructions**:
  - **Install SQLAlchemy**:
    - Run:

```
pip install Flask-SQLAlchemy
```

  - **Create a Database Model**:
    - Modify `app.py` to set up a database:

```
from flask_sqlalchemy import SQLAlchemy

app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///books.db"
db = SQLAlchemy(app)

class Book(db.Model):
```

```
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100), nullable=False)
    author = db.Column(db.String(100), nullable=False)
```

- **Perform CRUD Operations**:
  - Update your API to use the `Book` model for storing books in the database.

11. Handling Errors and Validating Input

- **Instructions**:
  - **Implement Error Handling**:
    - Use Flask's error handling decorators to manage errors.

```
@app.errorhandler(404)
def not_found(e):
    return {"error": "Not found"}, 404
```

  - **Validate Incoming Data**:
    - Implement input validation using the `jsonschema` library or Flask-WTF.

Phase 4: Enhancing the Application

12. User Authentication

- **Instructions**:
  - **Install Flask-Login**:
    - Run:

```
pip install Flask-Login
```

  - **Create User Authentication**:
    - Set up user registration and login routes, using hashed passwords (consider using `werkzeug.security`).

13. Testing the Application

- **Instructions**:
  - **Write Unit Tests**:
    - Create a test file `test_app.py` using `unittest` or `pytest`:

```
import unittest
from app import app
```

```
class BasicTests(unittest.TestCase):
    def setUp(self):
        self.app = app.test_client()

    def test_home(self):
        response = self.app.get('/')
        self.assertEqual(response.data, b'Hello, World!')
```

14. Deploying the Application

- **Instructions**:
  - **Deploying to Heroku**:
    - Create a `requirements.txt` using:

```
pip freeze > requirements.txt
```

    - Create a `Procfile` with:

```
web: python app.py
```

    - Follow [Heroku's deployment guide](https://devcenter

.heroku.com/articles/getting-started-with-python) to deploy your application.

Phase 5: Real-World Project

15. Capstone Project

- **Instructions**:
  - **Choose a Project**:
    - Decide on a project like a blog platform, task manager, or simple e-commerce API.
  - **Define Features**:
    - List features (e.g., user authentication, CRUD operations, etc.).
  - **Follow Best Practices**:
    - Use Git for version control, write clean code, and maintain documentation.
  - **Optional Frontend Integration**:
    - If desired, create a frontend using a framework like React, Vue, or Angular to interact with the backend API.

Additional Resources

- **Books**:
  - *Fluent Python* by Luciano Ramalho

    - *Automate the Boring Stuff with Python* by Al Sweigart
- **Online Courses**:
    - Coursera
    - Udemy
    - freeCodeCamp
- **Documentation**:
    - Python Official Documentation
    - Flask Documentation
    - SQLAlchemy Documentation