

Python & Flask

Databases and Interfaces

Matthew Pike & Yuan Yao

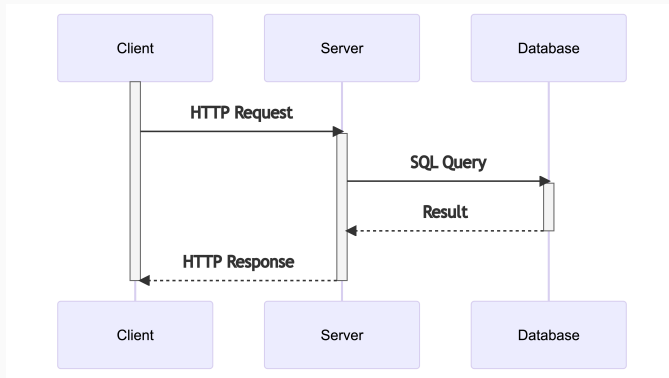
University of Nottingham Ningbo China (UNNC)

Overview

- A quick introduction to the Python programming language.
 - We will **use** Python as a tool, not as a focus of the module.
- A overview of the Flask web framework, specifically:
 - How to create Routes.
 - How to get user input from a web form.
- Using Jinja to create HTML templates:
 - Creating templates and binding data to them.
 - Template inheritance.

Web Applications

Recall: The Client - Server Model



- A web application is a software that operates on a web server, processing client (web browser) requests.
- Web applications frequently generate custom web pages for each user, drawing on database-stored data.
- They facilitate the creation of dynamic web pages that:
 - Offer a customised user experience.
 - Have user authentication capabilities.
 - Handle user-submitted data (such as forms).
 - Persist data in a database.
 - Integrate with other web services (like APIs, email systems, etc.).

Web Application Frameworks (WAF)

- A Web Application Framework (WAF) is a software framework designed to support and alleviate some of the workload in web application development, typically providing:
 - URL routing
 - Handling of requests
 - HTML templating
 - Database interaction
 - User authentication
 - Error management
 - Activity logging
- Some of the widely-used WAFs include:
 - Django (Python): <https://www.djangoproject.com/>
 - Ruby on Rails (Ruby): <https://rubyonrails.org/>
 - Laravel (PHP): <https://laravel.com/>


Flask is a lightweight WAF written in Python, designed for simplicity and rapid deployment.

Pros

- Minimalistic and straightforward to learn.
- Comprehensive documentation.
- Built-in server and debugger.
- Robust templating with Jinja.

Cons

- Hidden complexities can complicate debugging.
- Not as fully-featured as some other WAFs.

 Flask has recently been updated to version 3.0.0

Since the beginning of this 2023-2024 run of DBI and now, Flask has launched a significant update, advancing to version 3.0.0. This update introduces several improvements and also some breaking changes. However, for our use cases, the changes are not significant and we can continue to use the same code as before. Both versions should work similarly for the code we will be using in this module.

Python

- Python is a widely used, general-purpose programming language.
- It is a high-level language, noted for its readability, ease of writing, and a comprehensive standard library.
- Being an interpreted language, Python code does not require compilation prior to execution.
- It is free, open-source, and compatible with all principal operating systems.
- There are two primary branches of Python: Python 2 and Python 3.
 - Python 2 has reached end-of-life and is no longer supported, while Python 3 is the current and actively maintained version.
 - Only **Python 3** will be the version used in this module.

Python: Variables and Data Types

- Python has five standard data types:
 - Numbers
 - Strings
 - Lists
 - Tuples
 - Dictionaries
- Python is dynamically typed, meaning that the type of a variable is determined at runtime

```
x = 1 # Integer
y = 2.8 # Decimal number
z = True # Boolean
n = None # Absence of a value
name = "John" # String
fruits = ["apple", "banana", "cherry"] # list
numbers = (1, 2, 3) # Tuples
# Dictionary
population = {
    "UK": 66.65,
    "China": 1439.73
}
```

Python: Conditional Statements

- Python utilises `if`, `elif`, and `else` for conditional statements.
- Code blocks are delineated by indentation instead of curly brackets.
- Python does not have a `switch` statement.

```
x = 2
if x < 0:
    print("x is negative")
elif x == 0:
    print("x is zero")
else:
    print("x is positive")
```

Python: Loops

- Python uses **for** and **while** to implement loops
- Indentation is used to define code blocks, rather than curly brackets
- We can iterate over a list using a **for** loop

Example: for Loop

```
fruits = ["apple", "banana", "cherry"]  
  
for x in fruits:  
    print(x)
```

Example: while Loop

```
x = 0  
while x < 6:  
    print(x)  
    x += 1
```

Python: Functions

- Python employs **def** to declare functions, followed by the function name and parentheses.
- Arguments are specified within the parentheses.
- A colon (:) initiates the function's code block, which must be indented.
- Functions conclude with a **return** statement to output a value; without this, they return **None**.
 - **None** is a special Python value that represents the absence of a value.

```
def square(x):  
    return x * x  
  
def say_hello(name):  
    print("Hello " + name)
```

Python: Importing Modules

- A module is a Python file with definitions and statements.
- Modules are brought into a program using the **import** statement.
- Specific functions within a module can be imported using **from**.
- Python's extensive standard library provides a wide range of functionalities to enhance our programs.

```
import math
from math import sqrt
# Here we use the use of the square root
# function from the math module

# We use the module name to access
# the function (math.sqrt)
print(math.sqrt(4))

# Whereas here we can use the
# function name directly, since
# we have explicitly imported it
print(sqrt(4))
```


How we will use Python

- Python will serve as a tool rather than the central subject of the module.
- We will use Python for:
 - Developing web applications.
 - Interacting with databases.
- Learning Python in depth is not the module's aim:
 - Essential Python knowledge will be acquired as we go along.
 - The emphasis will be on web application and database concepts.
 - Your Python skills won't be directly tested; however, Python will be used for labs and coursework.

Flask

Flask: Hello World

```
from flask import Flask      # Import the Flask class
app = Flask(__name__)        # Create an instance of the Flask class

@app.route("/")              # Bind the hello() function to the / URL
def hello():                 # Define the hello() function
    return "Hello World!"    # Return the string "Hello World!"

if __name__ == "__main__":   # The main entry point for the application
    app.run(port=5000,        # Run the app on port 5000
            debug=True)       # Enable debugging
```

- In Flask, a route associates a URL with a function, executing that function whenever the URL is visited.
- The `route()` decorator is applied to link a function to a specific URL.
 - Example: The `hello()` function is linked to the `/hello` URL:

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

- **Note:**

- `@app.route("/hello")` specifies a direct route that will respond to the exact URL `/hello`.
- `@app.route("/hello/")` is an implicit route and will match the `/hello` and `/hello/` URLs

Flask: Variable Sections in URLs

- We can define variable sections in URLs using angle brackets
- The variable section will be passed as a parameter to the function. The names must match the parameters in the function definition
- Additionally, we can specify the types of the variables using a converter:
 - **string** (default)
 - **int**: accepts positive integers
 - **float**: accepts positive floating point values

```
from flask import Flask
app = Flask(__name__)

@app.route("/hello/<name>")
def hello(name):
    if name == "John":
        return "Hello John!"
    else:
        return "Hello Stranger!"

@app.route("/square/<int:x>")
def square(x):
    return str(x * x)
```

Flask: Specifying Route Methods

- By default, routes will respond to **GET** requests
- We can specify which HTTP methods the route will respond to using the **methods** parameter
 - `@app.route("/hello", methods=["GET", "POST"])`
 - The route will respond to both **GET** and **POST** requests
 - `@app.route("/hello", methods=["POST"])`
 - The route will only respond to **POST** requests
 - If a **GET** request is made, a **405 Method Not Allowed** error will be returned
- In situations where we want to respond to both **GET** and **POST** requests, we can use the **request** object to determine which method was used
 - `from flask import request`
 - `if request.method == "GET":`
 - `if request.method == "POST":`

Flask: Route Methods Example

```
# We need to import the request object before we can use it
from flask import Flask, request

# __name__ is a special variable that represents the name of the module
app = Flask(__name__)

# The route will respond to both GET and POST requests
@app.route("/hello", methods=["GET", "POST"])
def hello():
    # Determine the request method
    if request.method == "GET":
        return "Hello GET!"
    elif request.method == "POST":
        # Process the POST request - this could be a form submission
        return "Hello POST!"
```

Flask: Handling Form Data

- **Data Submission:** When a user submits a form, the data is transmitted to the server, according to the specified **action** and **method** attributes.
- **Flask's request Object:** provides access to the data sent by the client:
 - `request.form` for accessing data sent via **POST**.
 - `request.args` for accessing query string data sent via **GET** method.
- **Method Specifics:**
 - **GET** method transmits data in the URL as a query string.
 - **POST** method sends data within the request body of the HTTP request.
- **Accessing Data:**
 - Data can be retrieved using the `get()` method of `request.form` or `request.args`, using the HTML form field's **name** attribute as the key.
 - For instance, to access data from `<input type="text" name="username">`, use `request.form.get("username")` for POST or `request.args.get("username")` for GET.

Flask: Forms Example

HTML Form

```
<form
  action="/hello"
  method="POST"

>
<input
  type="text"
  name="uName"
/>
<input
  type="submit"
  value="Submit"
/>
</form>
```

Flask (Python)

```
from flask import Flask, request
app = Flask(__name__)

@app.route("/hello", methods=["POST"])
def hello():
    # Get the uName from the form
    uname = request.form.get("uName")
    return "Hello " + uname + "!"
```

Flask: The `url_for()` Function

- We can use the `url_for()` function to generate a URL to a specific function
 - `url_for("hello")` - will return the URL route that is mapped to the `hello()` function
- `url_for()` is especially useful when we are using templates - it allows us to change the URL of a route without having to change the URL in the HTML template.
- For Example:
 - `<form action="{{ url_for('hello') }}" method="POST">`
 - `Hello John!`
- We can also use `url_for()` to access static files (e.g. CSS files, JavaScript files, images) in the `static` folder
 - `url_for("static", filename="style.css")`

- **Navigating Between Pages:**
 - Often, we need to navigate the user to different pages, especially after actions like form submissions.
 - Flask's `redirect()` function facilitates this:
 - Import it using `from flask import redirect`.
 - Use `return redirect(url_for("hello"))` to redirect to a different endpoint, where "hello" is the endpoint name.
- **Handling Errors:**
 - Handling errors is crucial, particularly for scenarios like accessing non-existent pages.
 - Flask provides the `abort()` function for this purpose:
 - Import with `from flask import abort`.
 - Invoke `abort(404)` to send a 404 Not Found error, or use other appropriate HTTP status codes for different errors.

Flask: Redirect and Abort Example

```
# Note: We need to import the redirect and abort functions
from flask import Flask, redirect, url_for, abort
app = Flask(__name__)

@app.route("/hello/")
def hello(): return "Hello World!"
# Redirect to the hello() function
@app.route("/redirect/")
def redirect_to_hello(): return redirect(url_for("hello"))
# Abort with a 404 error
@app.route("/abort/")
def abort_404(): abort(404)
```

Using Templates for Dynamic Content

- **Jinja Templating Engine:**
 - Flask uses the Jinja templating engine for rendering dynamic HTML templates.
 - <https://jinja.palletsprojects.com/>
- **Benefits of Templates:**
 - Templates facilitate the reuse of HTML code, enhancing maintainability.
 - They allow for the dynamic passing of data to HTML, making web pages more interactive and user-responsive.
- **Storing and Rendering Templates:**
 - Store templates within the `templates` folder in the Flask application structure.
 - Render these templates using the `render_template()` function.
 - Remember to import it with `from flask import render_template`.
 - Data can be passed to templates as parameters in the `render_template()` function, allowing dynamic content rendering.

Flask: Templates Example

Flask (Python)

```
from flask import Flask
from flask import render_template
app = Flask(__name__)

@app.route("/hello/<name>")
def hello(name):
    # Capitalize the first letter
    # of the name
    cname = name.capitalize()
    # Pass the name to the template
    return render_template(
        "hello.html", name=cname)
```

HTML Template (hello.html)

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

Jinja: Key Features

- **Sandboxed Execution:**
 - Ensures code runs in a secure, sandboxed environment, mitigating security risks.
- **Template Inheritance:**
 - Facilitates the reuse and extension of HTML templates.
 - Use `{% extends "base.html" %}` to inherit from a base template like `base.html`.
- **Debugging:**
 - Offers clear error reporting with specific line numbers and the problematic line of code.
- **User-Friendly Syntax:**
 - Jinja's syntax is intuitive, bearing resemblance to Python:
 - `{{ variable }}` to display the value of a variable.
 - `{% if condition %} ... {% endif %}` for conditional statements.
 - `{% for item in list %} ... {% endfor %}` to iterate over a list.
 - `{% block content %} ... {% endblock %}` to define blocks of content that can be overridden in the child template.
 - `{% with var = value %} ... {% endwith %}` to define a variable within a template.

Jinja: Control Structures - If Statements

- **Conditional Content Display:**
 - The `if` statement in Jinja templates allows for conditional rendering based on certain conditions.
 - This can be particularly useful for displaying different content based on variable values.
- **Example: Different Messages for Different Names:**
 - Here's how you can use the `if` statement to display a tailored message according to the value of a variable `name`.
- **Checking Variable Definition:**
 - `if` statements can also be used to check if a variable is defined - this is useful for displaying different content based on whether a variable is defined or not.

```
<!-- If name is defined -->
{% if name %}
    {% if name == "John" %}
        <p>Hello John!</p>
    {% elif name == "Jane" %}
        <p>Hello Jane!</p>
    {% else %}
        <p>Hello {{ name }}!</p>
    {% endif %}
{% else %}
    <!-- Name is not defined -->
    <p>Hello Stranger!</p>
{% endif %}
```

Jinja: Control Structures - For Loops

- Iterating Over Lists:

- The `for` loop in Jinja can iterate over lists in templates.
- Useful for displaying repetitive elements like a list of names.

```
<ul>
{% for name in names %}
    <li>{{ name }}</li>
{% endfor %}
</ul>
```

- Iterating Over Dictionaries:

- The `for` loop can also iterate over dictionaries.
- An example is displaying names along with ages from a dictionary.

```
<ul>
{% for name, age in people.items() %}
    <li>
        {{ name }} is {{ age }} years old
    </li>
{% endfor %}
</ul>
```

- We can use the **extends** tag to inherit from another template
- We can use the **block** tag to define blocks of content that can be overridden in the child template
- Template inheritance promotes code reuse and makes it easier to maintain templates
- Typically:
 - We will define a base template that contains the common elements of the website (e.g. header, footer, navigation bar)
 - We will then define child templates that inherit from the base template and override the blocks of content that are specific to that page
 - We can then use the **render_template()** function to render the child template

Jinja: Template Inheritance Example

Base Template (base.html)

```
<!DOCTYPE html>
<html>
<head>
  <title>
    {% block title %}
    {% endblock %}
  </title>
</head>
  <body>
    {% block content %}
    {% endblock %}
  </body>
</html>
```

Child Template (hello.html)

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

{% block title %}
  Hello!
{% endblock %}

{% block content %}
  <p>
    Welcome to my website!
  </p>
{% endblock %}
```

Flask + Jinja: Messaging with `flash()`

- Using `flash()` to Send Messages:
 - The `flash()` function in Flask is used to send temporary messages to the user, often indicating the outcome of an action (like success, failure, or error).
 - These messages are typically set in route functions following an operation or user interaction.
- Retrieving Messages with `get_flashed_messages()`:
 - To access these flashed messages, use the `get_flashed_messages()` function.
 - This is commonly done in the base template to display messages across various pages.
- Setting up `flash()`:
 - A secret key must be configured for the Flask application to use `flash()` securely.
 - The value of the secret key doesn't matter, but it should be a secure, random string.

```
from flask import Flask, flash, redirect, url_for, render_template, request
app = Flask(__name__)
app.secret_key = 'your_secret_key' # Replace with your secret key
```

Flask + Jinja: Sending and Retrieving Messages Example (1/2)

Using the `flash()` function to send a message:

```
from flask import Flask, flash, redirect, url_for, render_template
app = Flask(__name__)
app.secret_key = 'i-love-dbi'
@app.route('/')
def index():
    flash("Welcome to the site!") # Send a message
    return redirect(url_for('display')) # Redirect to the display page
@app.route('/display')
def display(): return render_template('display.html')
if __name__ == '__main__': app.run(debug=True)
```

Flask + Jinja: Sending and Retrieving Messages Example (2/2)

Displaying the message using the `get_flashed_messages()` function:

```
<!DOCTYPE html>
<html>
<head> <title>Flash Message Example</title> </head>
<body>
    {% with messages = get_flashed_messages() %}
        {% if messages %}
            <ul>{% for message in messages %}
                <li>{{ message }}</li>
            {% endfor %}</ul>
        {% endif %}
    {% endwith %}
</body>
</html>
```

Typical Flask Application Structure

- We've covered a lot of Flask concepts in this lecture, so let's take a look at how they all fit together in a typical Flask application.
- The following is a typical Flask application structure:

```
|— app.py           # The main application file
|— static           # Static files (CSS, JavaScript, images)
|   |— style.css
|— templates        # Jinja/HTML templates
|   |— base.html    # Base template
|   |— hello.html   # Child template inheriting from base.html
|   |— display.html # Another child template
```


- Flask Documentation - Flask's official documentation
 - <https://flask.palletsprojects.com>
- Jinja Documentation - Jinja's official documentation
 - <https://jinja.palletsprojects.com>
- Python Documentation - Python's official documentation
 - <https://docs.python.org/3/>
- Flask Mega-Tutorial - A comprehensive tutorial on Flask
 - <https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world>
 - This is a very comprehensive tutorial on Flask, covering many topics that we will not cover in this module.