Python & Flask

Databases and Interfaces

Matthew Pike & Yuan Yao

University of Nottingham Ningbo China (UNNC)

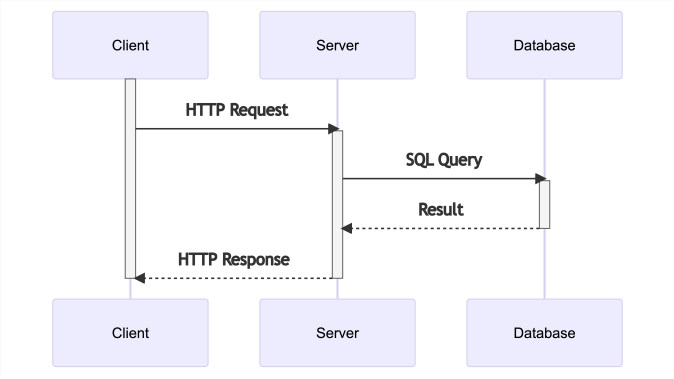
Overview

This Lecture

- In this lecture we will bring together the concepts of databases and web interfaces to create a web application that uses a database to store and retrieve data
  - Finally, we have: Databases **and** Interfaces together!
- A quick introduction to the Python programming language
  - We will **use** Python as a tool, not as a focus of the course
- A overview of the Flask web framework, specifically:
  - How to create Routes
  - How to get user input from a web form
  - How to use tempaltes
  - How to interact with SQLite databases from Python

Web Applications

Recall: The Client - Server Model



Web Applications

- A web application is a program that runs on a web server and responds to requests from clients (web browsers)
- Web Applications often generate tailored web pages for each client, using data stored in a database
- Web applications are used to make dynamic web pages, which:
  - Provide a more personalised experience for the user
  - Ability to authenticate users
  - Process user input (form data)
  - Store data in a database
  - Interact with other web services (e.g. APIs, email, etc.)

Web Application Frameworks (WAF)

- A web application framework (WAF) is a software framework that systematizes some of the common tasks involved in developing a web application. Typically this includes:
  - URL routing
  - Request handling
  - Templating
  - Database access
  - Authentication
  - Error handling
  - Logging
- Several popular WAFs include:
  - Django (Python): https://www.djangoproject.com/
  - Ruby on Rails (Ruby): https://rubyonrails.org/
  - Laravel (PHP): https://laravel.com/

Flask

> **Note**
>
> In DBI, we will be using Flask as our WAF. We installed Flask in Lab 1.

Flask is a lightweight WAF written in Python. It is designed to make getting started quick and easy.

Pros

- Lightweight and Easy to learn
- Good documentation
- Integrated server and debugger
- Powerful templating engine (Jinja2)

Cons

- Lots of “magic” happening behind the scenes which can make debugging difficult
- Not as feature rich as other WAFs

Python

Python Overview

- Python is a general-purpose programming language that is widely used
- A high-level language that is easy to read and write (and learn!) and has an extensive standard library
- Python is an interpreted language, meaning that it does not need to be compiled before it is run
- It's free, open source and runs on all major operating systems
- There are two main versions of Python: Python 2 and Python 3
  - Python 2 is no longer supported and Python 3 is the current version
  - We will be using **Python 3** 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"] # A list
numbers = (1, 2, 3) # Tuples
# Dictionaries
population = {
    "UK": 66.65,
    "China": 1439.73
}
```

Python: Conditional Statements

- Python uses `if`, `elif` and `else` to implement conditional statements
- Indentation is used to define code blocks, rather than curly brackets
- There is no `switch` statement in Python

```
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

`for` Loop

```
fruits = ["apple", "banana",
          "cherry"]

for x in fruits:
    print(x)
```

`while` Loop

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

Python: Functions

- Python uses `def` to define functions followed by the function name and parentheses
- Parameters are defined inside the parentheses
- The code block within every function starts with a colon (:) and is indented
- A function can return a value using the `return` statement else it will return `None`

```
def square(x):
    return x * x

def say_hello(name):
    print("Hello " + name)
```

Python: Importing Modules

- A module is a file containing Python definitions and statements
- The `import` statement is used to import modules
- We can import specific functions from a module using `from`
- Python has a large standard library which we can use to extend the functionality of our programs

```
import math
from math import sqrt
# Here we 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))
```

Flask

Flask: Hello World

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

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

if __name__ == "__main__":
    app.run(port=5000,
            debug=True)
```

Flask: Routing

- A route is a URL that Flask maps to a function, calling that function when the route is accessed
- We use the `route()` decorator to bind a function to a URL
  - Example: Here we bind the `hello()` function to the `/hello` URL:

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

- Note:
  - `@app.route("/hello")` is an explicit route and will only match the `/hello` URL
  - `@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: Forms

- When the user submits a form, the data is sent to the server
- Flask provides `request` object containing the data sent by the user
  - `request.form` - form data sent using the `POST` method
  - `request.args` - query string data sent using the `GET` method
- Remember, in simple terms:
  - Forms using `GET` send data in the URL
  - Forms using `POST` send data in the request body
- We can access the data using the `get()` method, with the `name` of the form field (in HTML) as the parameter
  - `request.form.get("name")`
  - `request.args.get("name")`
- For Example:
  - `<input type="text" name="username">`
  - `request.form.get("username")`

POST is safer

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, \
    render_template, request
app = Flask(__name__)

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

Flask: Static Files

- Web pages are often made up of static files such as images, CSS files, and JavaScript files
- In Flask, we should use the `static` folder to store these files
- We can access these files using the `url_for()` function
  - `url_for("static", filename="style.css")`
- For Example:
  - `<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">`

Flask: Redirects and Abort

- It's typical that we want to send the user from one page to another, for example, after they submit a form
- To redirect the user to another endpoint, we can use the `redirect()` function:

  redirect是重定向函数，输入一个URL后，自动跳转到另一个URL所在的地址

  - `from flask import redirect`
  - `return redirect(url_for("hello"))`
- We will also want to handle errors, for example, if the user tries to access a page that doesn't exist:
  - `from flask import abort`
  - `abort(404)`

Flask: Redirect and Abort Example

```
from flask import Flask, redirect, url_for, abort
app = Flask(__name__)

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

@app.route("/redirect")
def redirect_to_hello(): return redirect(url_for("hello"))

@app.route("/abort")
def abort_404(): abort(404)
```

Templating with Jinja2

Flask: Templates

- Flask uses the Jinja2 templating engine to render HTML templates
- Templates allow us to reuse HTML code and pass data to the template
- Templates are stored in the `templates` folder and are rendered using the `render_template()` function
    - We can pass data to the template using parameters in the `render_template()` function

Flask: Templates Example

Flask (Python)

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

@app.route("/hello/<name>")
def hello(name):
    # Capitalize first letter
    cname = name.capitalize()
    # Pass date to 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>
```

Jinja2

- Jinja2 is a templating language for Python that is used by Flask
- It allows us to use variables and control structures in HTML templates
- Jinja2 supports:
  - Sandboxed execution - code is run in a sandboxed environment for security purposes
  - Template Inheritance - templates can inherit from other templates
    - `{% extends "base.html" %}` - extends the `base.html` template
  - Easy to debug - errors are reported with line numbers and the offending line of code
  - Easy to use syntax - similar to Python
    - `{{ variable }}` - prints the value of the variable
    - `{% if condition %} ... {% endif %}` - `if` statement
    - `{% for item in list %} ... {% endfor %}` - `for` loop

Jinja2: Control Structures

`if` statement

```
{% if condition %}
    ...
{% elif condition %}
    ...
{% else %}
    ...
{% endif %}
```

`for` loop

```
{% for item in list %}
    ...
{% endfor %}
```

Jinja2: Template Inheritance

- 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

Jinja2: 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 {{ name }}
{% endblock %}

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

Databases and Interfaces

Python: SQLite

- Python has a built-in SQLite3 module that allows us to interact with SQLite databases
- We can use the `sqlite3` module by importing it
  - `import sqlite3`
- Using this module, we can now draw upon our existing SQL knowledge to interact with SQLite databases from Python/Flask
  1. Connect to the database and create a cursor object
  2. Execute SQL commands using the cursor object
  3. Present the results to the user via a web interface

Python + SQLite: Connect to Database and SELECT

```
import sqlite3
# Connect to the database
conn = sqlite3.connect("Students.db")
# Make the results easier to work with
conn.row_factory = sqlite3.Row
# Create a cursor object
cur = conn.cursor()
# Execute SQL commands using the cursor object
cur.execute("SELECT * FROM Student")
# Get the results
rows = cur.fetchall()
# Close the connection
conn.close()
```

Python + SQLite: INSERT

Using **with** will automatically close the database connection when the code block is finished

```
import sqlite3
# Connect to the database
with sqlite3.connect("Students.db") as conn:
    # Create a cursor object
    cur = conn.cursor()
    # Execute SQL commands using the cursor object
    cur.execute(""" INSERT INTO Student
                    VALUES (NULL, 'John', 'Smith')
    """)
    conn.commit()
```

Python + SQLite: Parameterized Queries

- We can use parameterized queries to use python variables in our SQL queries
- We do this using the `?` placeholder in our SQL query
- We then pass a tuple containing the values to be inserted into the query as the second parameter to the `execute()` method
- Parameterized queries are more secure than string concatenation and help prevent SQL injection attacks

```
cur.execute("""
    INSERT INTO Student
    VALUES (NULL, ?, ?)
    """,
    (fname, lname)
)
```

A common Gotcha: Single Value Tuples

```
cur.execute("""
    INSERT INTO Student
    VALUES (NULL, ?, "Smith")
    """, (fname,)
)
```

Python + SQLite: Handling Errors

- Web applications should be reliable, robust and handle errors gracefully
- We can use the `try` and `except` statements to handle errors
  - We should think of this as a `try` to execute the code and `except` if there is an error, but dont crash if there is an error

```
try:
    cur.execute("""
        INSERT INTO Student
        VALUES (1, ?, ?)""",
        ("Bob", "Clark")
    )
    conn.commit()
except sqlite3.Error as e:
    print("""
    An error occurred when
    adding a student to the
    database: """, e)
```