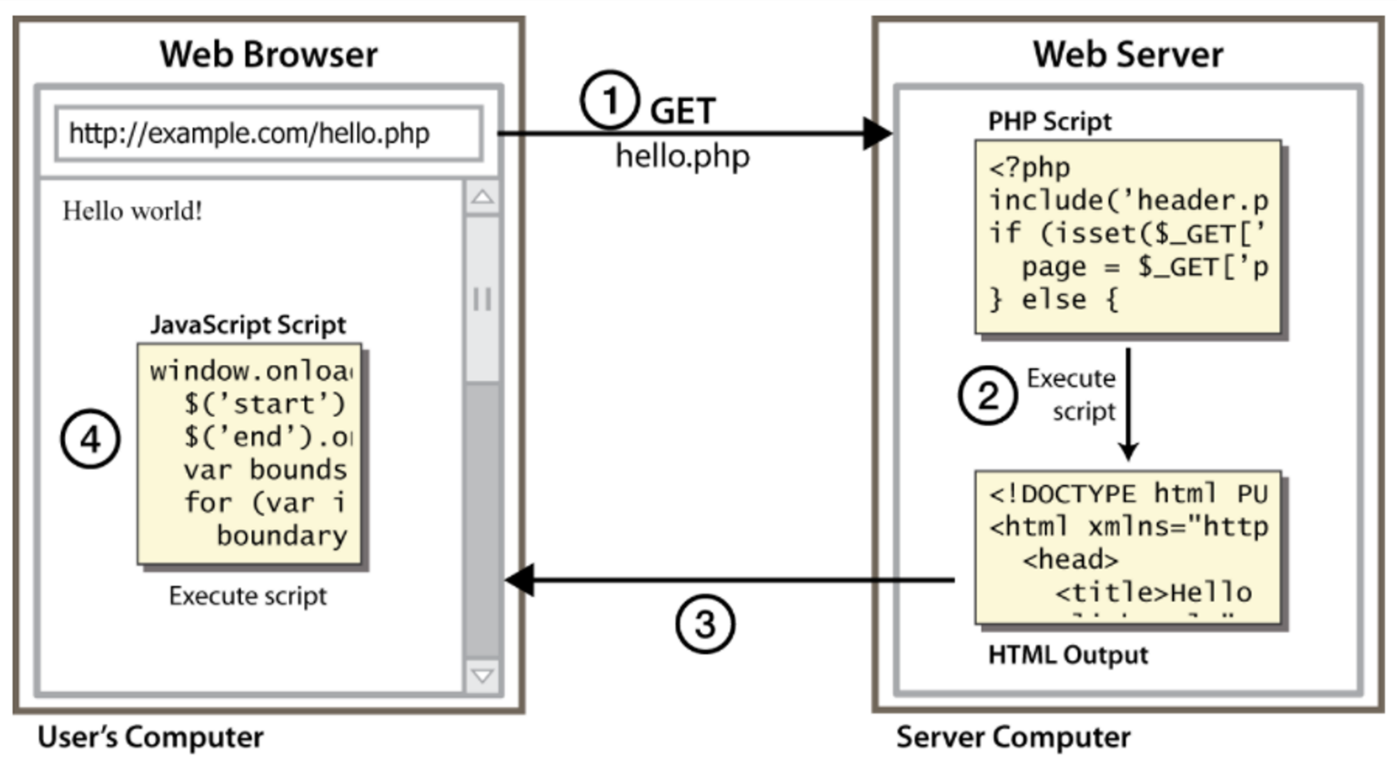
Python & Flask

Databases and Interfaces
Dr Matthew Pike & Prof Linlin Shen

Web Applications and Server-side Scripting



```
window.onloa
  $('start')
  $('end').o
  var bounds
  for (var i
    boundary
```

```
<?php
include('header.p
if (isset($_GET['
  page = $_GET['p
} else {
```

```
<!DOCTYPE html PU
<html xmlns="http
  <head>
    <title>Hello
```

http://courses.cs.washington.edu/courses/cse190m/11sp/lectures/slides/images/figure_2_client_side_scripting.png

Web Applications

- Are used to make web pages dynamic:
    - Provide different content depending on context
    - Authenticate users
    - Process form information
    - Interface with other services:
        - Databases
        - E-mail
        - Other Web Applications

Web Application Frameworks

- A web application framework (WAF) systematizes some routine aspects of web applications, typically including:
  - Centralizing the applications “use cases” and associated logic of the web app
  - The WAF provides mechanisms for detailing how incoming requests should be handled and routed.
  - A Templating system is often provided to separate data and the layout of the web app
- Several different systems exists in many different programming languages:
  - Django (Python) - https://www.djangoproject.com/
  - Ruby on Rails (Ruby) - https://rubyonrails.org/
  - CakePHP (PHP) - https://cakephp.org/

Flask

- Flask is a (micro) WAF written in Python
- Pros
    - It uses Jinja2 as its templating engine, to combine dynamic data and static templates into dynamically built web pages.
    - Integrated development server and debugger – Great for learning!
    - Flask is well documented
    - Works with Python 2.7 and Python 3.
        - We will only be using Python 3.
- Cons
    - There is lot of “magic” happening in the background which is hidden from the developer

Python

Python Overview

- Python is an **interpreted**, high-level programming language which is dynamically typed
- It has high-level built-in data structures and the extensive standard library
- Python's simple, easy to learn syntax emphasizes readability and therefore is great for beginner programmers (and beyond!)
- Python supports modules and packages, which encourages program modularity and code reuse.
- Available, without charge for all major platforms, and can be freely distributed.
- There are two major versions of Python (2 and 3).
  - We will only be using Python 3
- Please consult the example - 000_python.py
  - Run it from your terminal using – python3 000_python.py

Variables and Types

- Python has five standard data types:
  - Numbers
  - String
  - List
  - Tuple
  - Dictionary
- Python is dynamically typed, meaning we do not need to explicitly state a variable’s type

```python
a = 13  # Integer
b = 1.3 # Decimal Number
c = "Hello!" # String
d = True # Boolean
e = None # indicating the absence of a value


# Here we define a list of names
names = ["Alex", "Bob", "Charlie", "Dan"]


# Tuple
point = (12.34, 56.78)


# Dictionary
# {key:value, key1:value1, .... }
population = {"UK": 66796807,
             "China": 1427647786}
```

Conditionals

- Like other languages
- Not the absents of brackets or braces, a colon (:) is used instead
- There is no “switch” statement in Python

```
a = 13

# If conditional,
if a > 0:
    print("Number is positive")
elif a < 0:
    print("Number is negative")
else:
    print("Number is 0")
```

Loops

- Python has ”while” and “for” loops
- These are like those in familiar programming languages
- We can also “iterate” through lists using for loops

```
for i in [1,2,3,4,5]:
    print(i)
```

Functions

- Function blocks begin with the keyword **def** followed by the function name and parentheses
- Parameters are defined inside the parentheses
- The code starts with a colon (:) and is indented.
- A function may (optionally) return a value back to its caller. Otherwise **None** is returned.

```
def square(x):
# Simply return the number (parameter)
# timed by itself.
   return x * x


def sayHi(a, b="Bob"):
   print("Hello", a, "and hello", b)



sayHi("Alex")
sayHi("Jack", "Jill")
```

Imports and Modules

- A module is a file consisting of Python code
- A module can define functions, classes and variables.
- We use the **import** statement to specify which modules we want to include in our program
- **from** allows us to specify attributes from the module, that we want to include
- There is a large, pre-existing standard library that we can utilise in our Python code

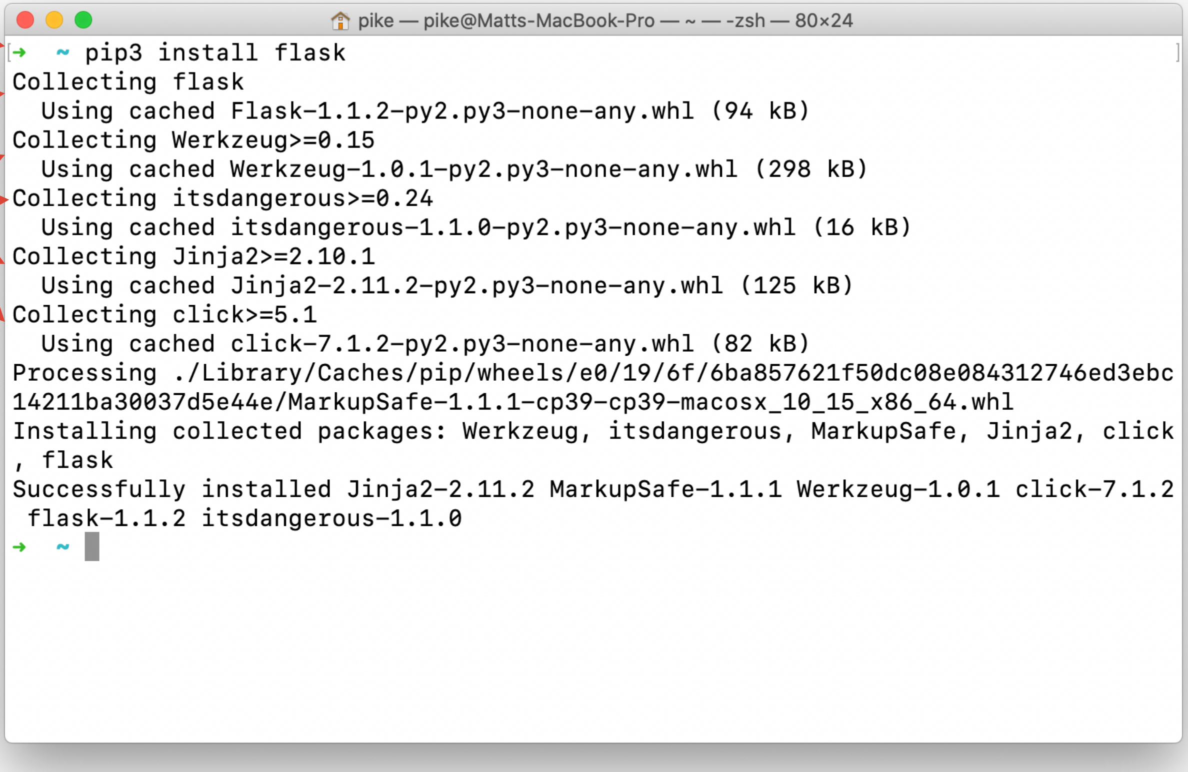
```python
import math

from os import getcwd
```

```python
# We can use the a method provided by the math library
print(math.sqrt(12345))

# We explicitly imported getcwd from the
# os module, therefore
# we do not need to prepend "os." before it.
print(getcwd())
```

Getting Started with Flask

Install Flask

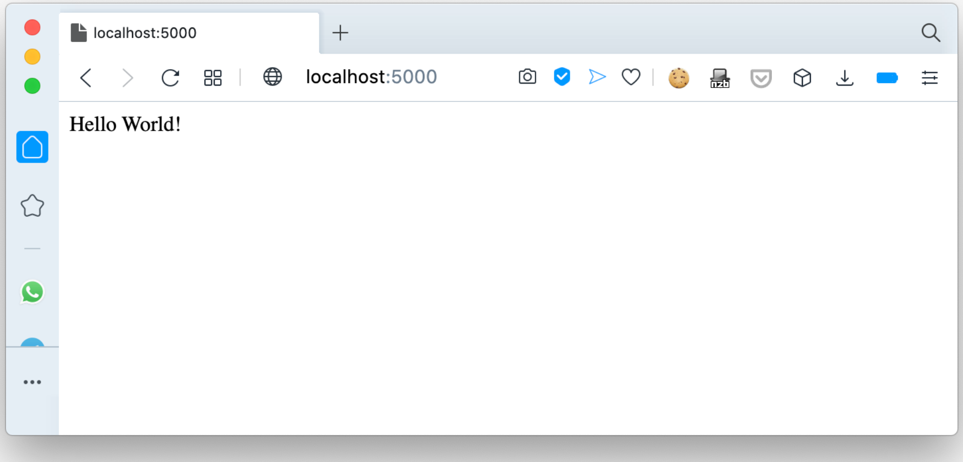
- pip3 install flask
    - Will install flask
    - And all associated dependencies
- This will install the flask "globally"
- You should probably use `virtualenv`, but again, for simplicity, we'll install Flask globally
- https://virtualenv.pypa.io



```
➜  ~ pip3 install flask
Collecting flask
  Using cached Flask-1.1.2-py2.py3-none-any.whl (94 kB)
Collecting Werkzeug>=0.15
  Using cached Werkzeug-1.0.1-py2.py3-none-any.whl (298 kB)
Collecting itsdangerous>=0.24
  Using cached itsdangerous-1.1.0-py2.py3-none-any.whl (16 kB)
Collecting Jinja2>=2.10.1
  Using cached Jinja2-2.11.2-py2.py3-none-any.whl (125 kB)
Collecting click>=5.1
  Using cached click-7.1.2-py2.py3-none-any.whl (82 kB)
Processing ./Library/Caches/pip/wheels/e0/19/6f/6ba857621f50dc08e084312746ed3ebc
14211ba30037d5e44e/MarkupSafe-1.1.1-cp39-cp39-macosx_10_15_x86_64.whl
Installing collected packages: Werkzeug, itsdangerous, MarkupSafe, Jinja2, click
, flask
Successfully installed Jinja2-2.11.2 MarkupSafe-1.1.1 Werkzeug-1.0.1 click-7.1.2
 flask-1.1.2 itsdangerous-1.1.0
➜  ~ 
```

Moodle Resource: 001_hello_world.py

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

@app.route('/')
def hello_world():
    return 'Hello World!'


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



Hello World!

Run this example using – python3 001_hello_world.py

Note – There are other (arguably more proper) ways of running Flask processes. For simplicity, we'll use this approach in this module.

Routes

Routing

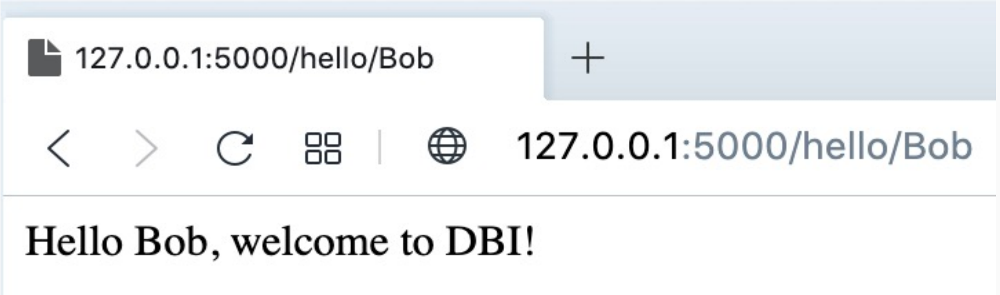
- Routing is the mapping of URLs to Python functions
- The **001_hello_world.py** example routes the shortest possible URL, namely / to your **hello_world** function
- Note difference between /hello and /world/
    - /hello is an explicit path and will not reroute /hello/
    - /world/ will route both /world and /world/
- We can define routes additional routes (002_routes.py)

```python
@app.route('/’)
def hello_world():
    return 'Hello World!'


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


@app.route('/world/')
def world():
    return "World"
```

Variables in URLs

- What if you want a *set* of URLS, based on some *pattern* to go to a function?
- Flask will parse the URL and pass the arguments to your function
- Example - (002_routes.py)

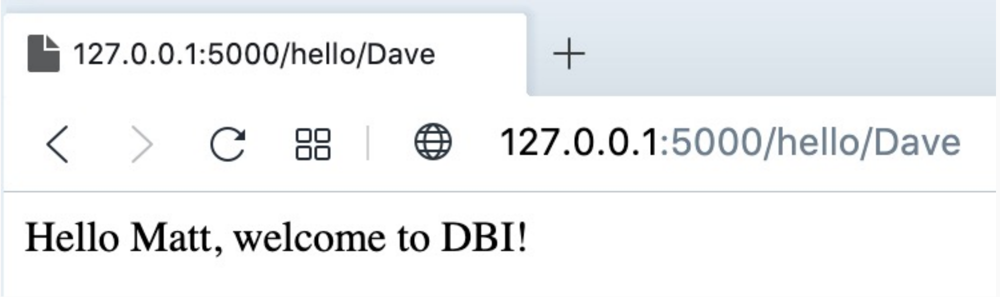
```
@app.route('/hello/<name>')
def hello_person(name):
    if name == "Dave":
        name = "Matt"
    return "Hello {}, welcome to
                    DBI!".format(name)
```



127.0.0.1:5000/hello/Bob

127.0.0.1:5000/hello/Bob

Hello Bob, welcome to DBI!



127.0.0.1:5000/hello/Dave

127.0.0.1:5000/hello/Dave

Hello Matt, welcome to DBI!

Static Resources

Static Resources

- Web sites have a lot of static pages:
  - CSS
  - JS
  - Images
- These files are not dynamic (their content does not change between reloads), they are therefore called ”static” resources (or assets).
- The static files are put in a folder called ”static”
- URLs are generated with **url_for** but with 'static' as the first argument and a keyword argument of filename for the rest.

Example

```
<head>
    <title> My Amazing Website </title>

    <link rel="stylesheet” href="{{url_for('static',filename=‘style.css')}}">

</head>
```

This will (dynamically) retrieve the URL for the specified resource.

We can use this for retrieving the URL of our web app functions also.

e.g. url_for(”show_world”) => ‘/world/’

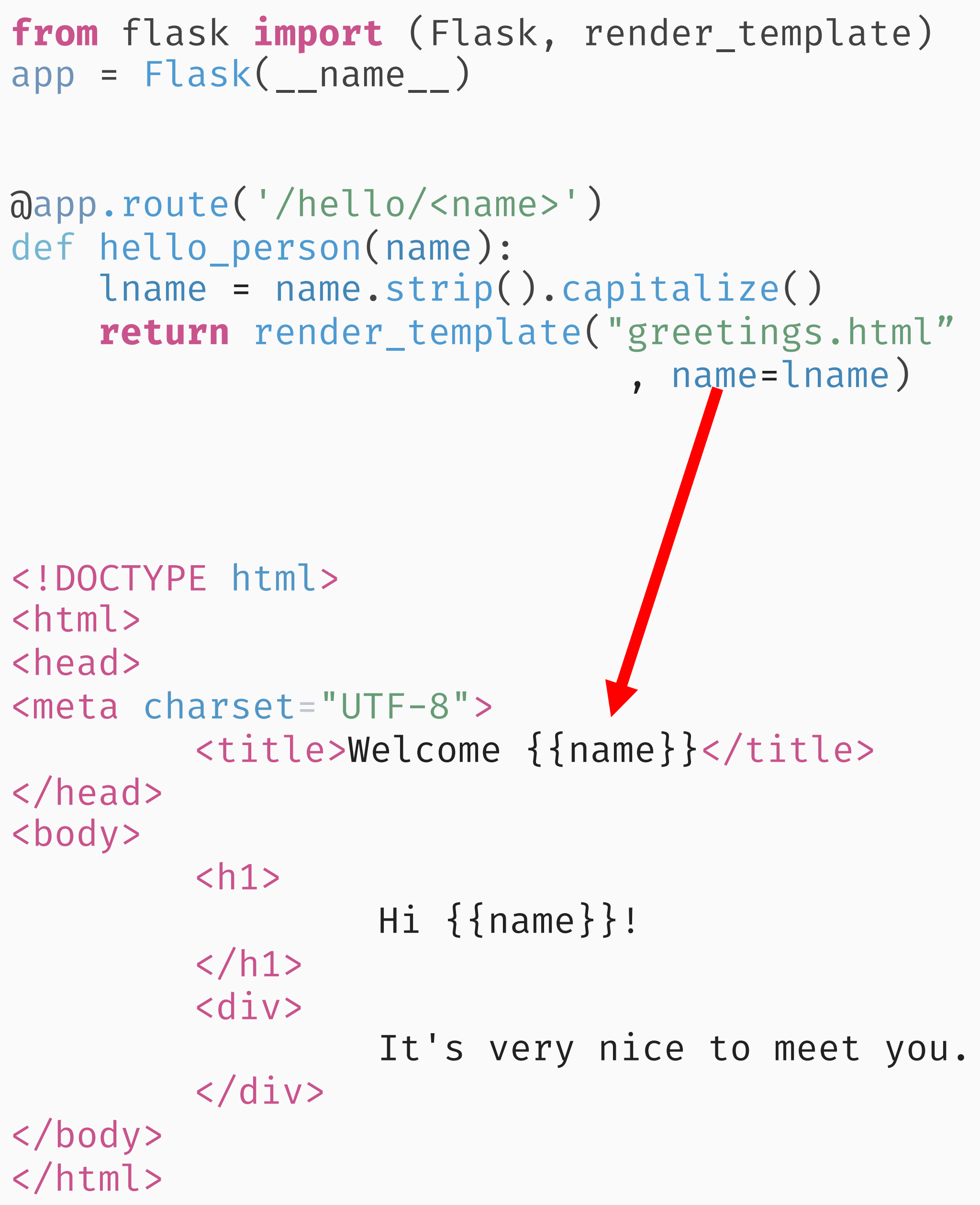
Specify that we want the URL of a static resource.

Specify the filename of the static resource we want the URL for.

Templates

Templates

- Allow us to separate data from layout
- Templates should be in a subfolder name ”templates”
- We need to import an additional dependency from the flask library –
    - **render_template**
- We return **render_template** from our mapped function
- See Example 003_templates.py and templates/greetings.html



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


@app.route('/hello/<name>')
def hello_person(name):
    lname = name.strip().capitalize()
    return render_template("greetings.html”
                            , name=lname)
```

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
        <title>Welcome {{name}}</title>
</head>
<body>
        <h1>
                Hi {{name}}!
        </h1>
        <div>
                It's very nice to meet you.
        </div>
</body>
</html>
```

Jinja2

- Flask uses Jinja2 as it’s templating engine
  - https://jinja.palletsprojects.com
- Jinja supports
  - Sandboxed execution – care needs to be taken, as data often comes from user provided input, which may be malicious.
  - Template inheritance – Templates may appear within other templates.
  - Easy to debug - Line numbers of exceptions directly point to the correct line in the template.
  - Easy to use syntax –
    - We use {{ and }} to print to the template output
    - We use {% and %} for statements (if, for, etc)
    - We use {# and #} for comments

Statements

- We can utilize python statements (for, if, etc) within our templates.
- Example
  - **For** loop are enclosed in {%..%}
  - **key and value** (from the Python dictionary) are put inside **{{ }}**
- See Examples
  - 003_templates.py
  - templates/module_credits.html

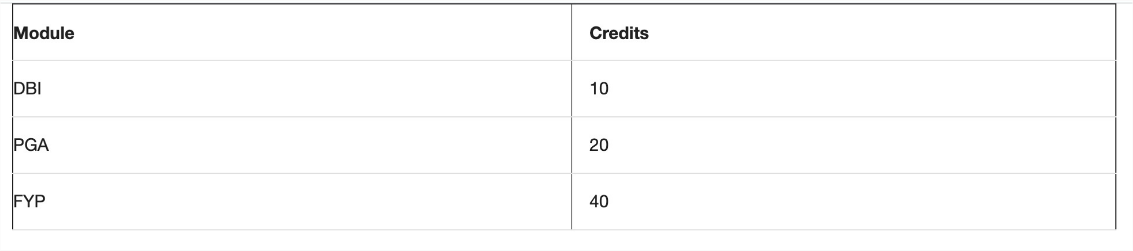
Flask (Python)

```
@app.route('/modules')
def get_modules_credits():
   mod_creds = {'DBI':10,'PGA':20,'FYP':40}
   return render_template('module_credits.html’,
                           result = mod_creds)
```

Template

```
    {% for key, value in result.items() %}
    <tr>
            <td> {{ key }} </td>
            <td> {{ value }} </td>
    </tr>
    {% endfor %}
```

Result



Module	Credits
DBI	10
PGA	20
FYP	40

Template Inheritance

- Very powerful, but simple concept

- We’re able to include templates within other templates using –
  - {% extends “....html” %}

- Typically, we will define a base template which is used on all pages of our web app

- Children templates will inherit the base template and introduce their own content

```
/* Base.html */

<!DOCTYPE html>
<html lang="en">
<head>
{% block head %}

<meta charset="utf-8">
<title>{% block title %}{% endblock %} - DBI</title>

{% endblock %}
</head>
<body>

<div id="content">
{% block content %}{% endblock %}
</div>
</body>
</html>
```

```
/* Inhereit.html */

{% extends "base.html" %}
{% block title %}Inheritance{% endblock %}
{% block content %}
   <h2>An Example of Inheritance</h2>
{% endblock %}
```

Forms

HTML Forms

- An HTML form can send data to the server. Suppose that the form has inputs:
    - Name
    - Age
    - Telephone Number
- Examples
    - 004_form.py
    - form.html

```
<form method="GET" action="/sayhi/">

<label for="name">Your Name</label>
<input name="name" value="Bob">

<label for="age">Your Age</label>
<input name="age" value="25">

<label for="age">Your Tel</label>
<input name="age" value=”+44 123456789">

<input type=”submit" value="Submit">

<form>
```

```
@app.route('/sayhi/', methods=['GET','POST'])
def sayhi():
   if request.method == 'GET’:
      name = request.args['name']
      age = request.args['age’]
      tel = request.args['tel’]
   return '[GET] - {}, {}, and
                    {}’.format(name,age,tel)
```

GET and POST

- HTML Forms can be sent using GET or POST
  - GET – Form values sent via the URL
  - POST – Form values sent in the body of the HTTP request
- Flask can access the data either way it's sent, but in different places
  - Flask puts GET data in request.args
  - Flask puts POST data in request.form
- Both objects are accessible as Python dictionaries
- The default way that a request comes in is as a GET request.
- We need to import the “requests” module in our Flask import statement
- We explicitly state in the app.route that our function accepts both forms of submission - `methods=['GET','POST'])`

Flask + SQLite

Python + SQLite

- Python has an in-built support for **SQlite**.
- SQlite3 module is shipped with Python distribution as a part of the standard library
  - **import** sqlite3
- We can use the SQL skills we've acquired so far in the course and embed them within the logic of our web application
- This will allow us to dynamically generate pages using data stored in the databases
- Our web app can also serve as the interface for inserting, updating and maintaining data in our SQLite database.

Connect + Execute

1. Import the SQLite module
2. Establish a connection
3. Specify how we want the results to be returned
4. Set up an cursor which allows us to query the DB
5. Execute our query
6. Get (all) the results

```
import sqlite3

# Establish a connection with our Student DB file
conn = sqlite3.connect('Students.db’)

# Converts the plain tuple result into a more
# useful object.
conn.row_factory = sqlite3.Row

# sqlite3.Cursor allows us to execute SQLite
# statements, fetch data from the result sets of
# the queries.
cur = conn.cursor()

# Here we say - execute this SQL query
cur.execute("SELECT * FROM Student LEFT OUTER JOIN
Grade ON Student.ID = Grade.ID;")

# Fetch all the results from the above query.
# We may alternatively request a single - fetchone
rows = cur.fetchall();
```

Controlling Changes

- The sqlite3 module may not automatically commit an INSERT, UPDATE, DELETE, REPLACE statement
- We can control this by explicitly using .commit()
- The module is attempting to automatically manage “transactions”, but often causes trouble …
- If you want autocommit mode, then set isolation_level to None.

```
# Using "with" will automatically close the
connection to the
# sqlite DB after the inner block completes
with sqlite3.connect("Students.db") as conn:
    cur = conn.cursor()
    cur.execute("INSERT INTO Student (ID,First,Last)
                 VALUES (?,?,?)", (id, first, last))
    conn.commit()
```

```
conn.isolation_level = None
```

Handling Errors and Failure

- Web Applications should be reliable and available
- A Web App should not crash as a result of an error interacting with a DB
- We use Python’s **try** and **except** constructs to trap interactions which may cause errors.
- We can think of **try/catch** as ”try this, if there’s an error, do that (except)”

```
try:
     cur.execute("""INSERT INTO People(Name,Age)
                    VALUES (?,?)", (name, age))
     conn.commit()
except sqlite3.Error as e:
    return print("Oh no, something went wrong!")
```

Next Steps

- We’ll practice this in the next lab session (which we’ll dedicate 2 weeks to)
- It’s very important you practice
  - Don’t just read the slides
- Coursework 2 will be based on Python, Flask and SQL