Lecture Notes:

- **Front-end** refers to the client side while **back-end** refers to the server side. The languages that we use for front-end include HTML, JavaScript and CSS. We will be using Python and Flask for the back-end.
- A HTTP Get request will show the query in the url.
  E.g. **/test/demo_form.php?name1=value1&name2=value2**
- An URL is generally in the form:
  scheme://host/some/path/to/file?query1=value&query2=value&...&queryn=value
  The ? separates the path from the query and the & separates 2 queries.
  I.e. The ? delimits the storage details from the query string and the & is used to delimit query string parameters.
- render_template is a function you can import from flask in Python that allows you to modify your HTML file with data from your Python file.
- request is another function you can import from flask in Python that allows you to do Get requests.
- g is another function you can import from flask in Python that allows flask to interact with the sqlite database.
- Here is a simple example of how you can use SQLite 3 with Flask:

```
// These functions were gotten from
// https://flask.palletsprojects.com/en/1.1.x/patterns/sqlite3/
import sqlite3
from flask import Flask, render_template, g

DATABASE = '/path/to/database.db'

def get_db():
    db = getattr(g, '_database', None)
    if db is None:
        db = g._database = sqlite3.connect(DATABASE)
    return db

def query_db(query, args=(), one=False):
    cur = get_db().execute(query, args)
    rv = cur.fetchall()
    cur.close()
    return (rv[0] if rv else None) if one else rv

def make_dicts(cursor, row):
    return dict((cursor.description[idx][0], value)
                for idx, value in enumerate(row))

@app.teardown_appcontext
def close_connection(exception):
    db = getattr(g, '_database', None)
    if db is not None:
        db.close()
```

- Here's what the get_db function does:
    1. The line "db=getattr(g, '_database', None)" creates a reference database called db and we're gonna use the getattr function to see if the attribute _database is defined on g. If g is not assigned to any value, we will assign it to None. Now, we know that db will be None if we are trying the database for the first time.
    2. The line "if db is None:" will run because of the above statement.
    3. The line "db=g._database=sqlite3.connect(DATABASE)" creates the connection to the database.

  In summary, get_db tries to make a connection with a database.
- The function query_db takes in an SQL query and will run it against the database. rv is an array that contains all the output rows of the query. The "one=False" argument means that all valid rows will be returned, not just the first.
- For each row in the database, the make_dict function will return it in dictionary form.
- Here's what the close_connection function does:
  For the line "db = getattr(g, '_database', None)", db is most likely not None as the user has connected it to a database in the get_db function, so this won't do anything. This is to double check that the user has opened a connection. In the case that the user did not connect to a database, db is set to None. Now, if db is not None, then we close the connection with the database.
  In summary, close_connection checks if db has been connected to a database (I.e. db is not None) and if db is not None, then we close the connection.
- Now, we'll build an actual application using Flask and SQLite.
  Assume that there is a database called database.db and that there's a table called employees. Furthermore, assume the employees table looks something like the following, but with more rows:

ID	First Name	Last Name	Position	Salary
1	Michael	Scott	Regional Manager	75, 000
2	Dwight	Schrute	Assistant to the Regional Manager	70, 000

Here will be the code:

```
import sqlite3
from flask import Flask, render_template, g

DATABASE = '/database.db'

def get_db():
    db = getattr(g, '_database', None)
    if db is None:
        db = g._database = sqlite3.connect(DATABASE)
    return db

def query_db(query, args=(), one=False):
    cur = get_db().execute(query, args)
    rv = cur.fetchall()
    cur.close()
    return (rv[0] if rv else None) if one else rv
```

```
def make_dicts(cursor, row):
   return dict((cursor.description[idx][0], value)
           for idx, value in enumerate(row))

app = Flask(__name__)

@app.teardown_appcontext
def close_connection(exception):
   db = getattr(g, '_database', None)
   if db is not None:
      db.close()

@app.route('/')
def root():
  db = get_db()
  db.row_factory = make_dicts

  employees = []
  for employee in query_db('select * from employees'):
    employees.append(employee)

 db.close()
 return employees.__str__()
```

This will get you an array of dictionaries where each dictionary represents a row in the database table.

Flask Basic Authentication:

- Python/Flask Code:

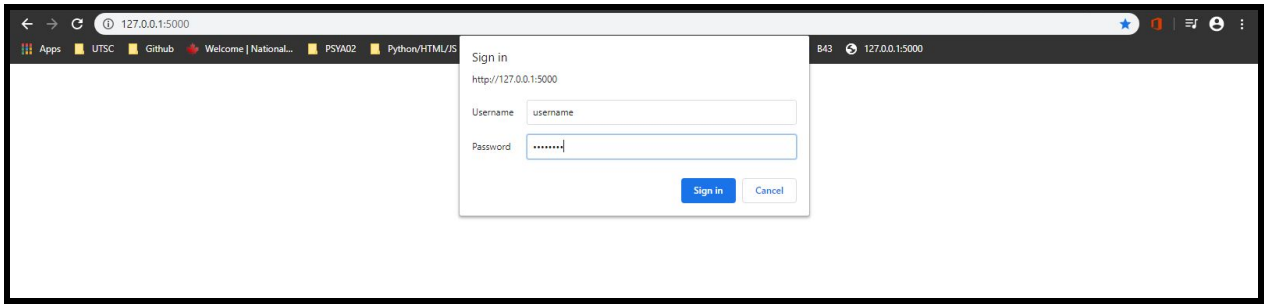
```python
from flask import Flask, request, make_response

app = Flask(__name__)

@app.route('/')
def index():
    #request.authorization.username gets the username the user entered.
    #request.authorization.password gets the password the user entered.

    # If the authorization exists inside the request and
    # both the username and password is correct, then the words "You are logged in" will appear.
    # Otherwise, a custom message will be displayed.
    if (request.authorization and request.authorization.username=="username1" and request.authorization.password=="password"):
        return "<h1> You are logged in. </h1>"

    # make_response takes 3 inputs:
    # 1. The message to the user.
    # 2. An HTTP Status Code. (Note: This is optional).
    # 3. Headers that are sent over that tells the browser that the route requires
    # HTTP basic authentication.
    return make_response('Could not verify', 401, {'WWW-Authenticate': 'Basic realm="Login Required"'})

@app.route('/page')
def page():
    return "<h1> You are in another page. </h1>"


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

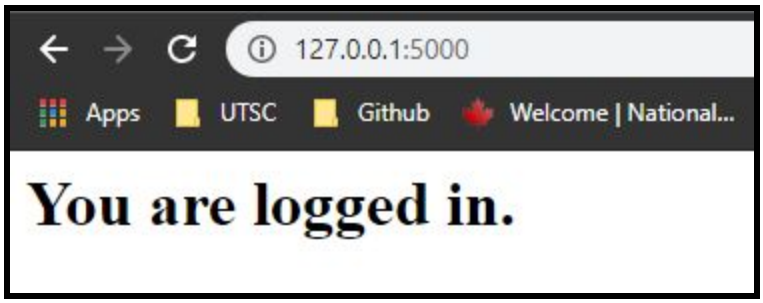
- Once you log in, the browser automatically sends the login information with each request afterwards. Furthermore, if only the homepage has the authentication requirement, if you log in and change your username/password, if you go to any page other than the homepage, you don't need to log in again. This is shown below:

  Original (Username is username and Password is password):

```
if (request.authorization and request.authorization.username=="username" and request.authorization.password=="password"):
    return "<h1> You are logged in. </h1>"
```

  Home Page (Needed to log in):





  Other Page (Didn't need to log in):

  Now, let's say I change my username to username1:

```
if (request.authorization and request.authorization.username=="username1" and request.authorization.password=="password"):
    return "<h1> You are logged in. </h1>"
```

When I refresh http://127.0.0.1:5000/page, I don't need to log in a second time.

However, if I go back to the home page, I do need to log in again.

- If you do want the user to enter their username and password at any page if they change their username or password, we can use decorators to do this. The code is shown below:

app.py

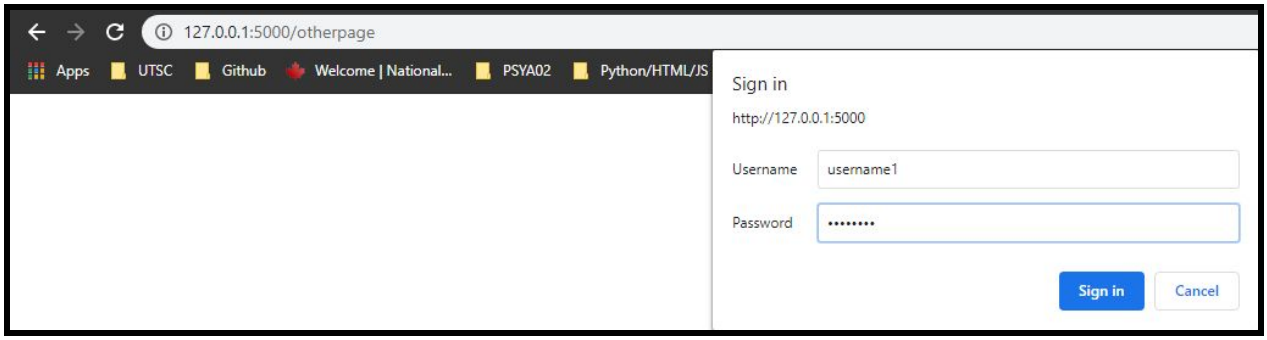
```python
from flask import Flask, request, make_response
from functools import wraps

app = Flask(__name__)

# Creates the decorator.
# It takes a function, which is the function that is being decorated.
def auth_required(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        auth = request.authorization
        if (auth and auth.username == "username1" and auth.password == "password"):
            return f(*args, **kwargs)
        return make_response('Could not verify', 401, {'WWW-Authenticate': 'Basic realm="Login Required"'})
    return decorated

@app.route('/')
@auth_required
def index():
    return "<h1> You are logged in to the home page. </h1>"

@app.route('/page')
@auth_required
def page():
    return "<h1> You are logged in to another page. </h1>"

@app.route('/otherpage')
@auth_required
def other_page():
    return "<h1> You are logged in to the other page. </h1>"


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

- When I first log in, I need to enter the username and password:

- Now, if I go to page or other page, I don't need to reenter the login info:

- Now, if I change the username as such, I have to enter the new username/password on any of the 3 pages.

```
def auth_required(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        auth = request.authorization
        if (auth and auth.username == "username1" and auth.password == "password"):
            return f(*args, **kwargs)
        return make_response('Could not verify', 401, {'WWW-Authenticate': 'Basic realm="Login Required"'})
    return decorated
```



- **make_response:**
  - This function can be called instead of using a return and you will get a response object which you can use to attach headers.
  - The HTTP WWW-Authenticate response header defines the authentication method that should be used to gain access to a resource.
  - The WWW-Authenticate header is sent along with a 401 Unauthorized response.
  - Syntax: **WWW-Authenticate: <type> realm=<realm>**
    <type>: This is the aAuthentication type. The most common authentication scheme is the "Basic" authentication scheme. The "Basic" HTTP authentication scheme transmits credentials as user ID/password pairs, encoded using base64. As the user ID and password are passed over the network as clear text (it is base64 encoded, but base64 is a reversible encoding), the basic authentication

scheme is not secure.
realm=<realm>: A description of the protected area. If no realm is specified, clients often display a formatted hostname instead.

- ***args and **kwargs:**
  - *args and *kwargs are special keywords which allow functions to take variable length arguments.
  - The special syntax *args in function definitions in python is used to pass a variable number of arguments to a function. It is used to pass a non-keyworded, variable-length argument list.
  - The syntax is to use the symbol * to take in a variable number of arguments; by convention, it is often used with the word args.
  - What *args allows you to do is take in more arguments than the number of formal arguments that you previously defined. With *args, any number of extra arguments can be tacked on to your current formal parameters (including zero extra arguments).
  - The special syntax **kwargs in function definitions in python is used to pass a keyworded, variable-length argument list. We use the name kwargs with the double star. The reason is because the double star allows us to pass through keyword arguments (and any number of them).
  - A **keyword argument** is where you provide a name to the variable as you pass it into the function. Keyword arguments do not care about the position of their arguments.
    E.g.

```
def num_comparator(num1, num2):
    if(num1 > num2):
        print(str(num1) + " is bigger than " + str(num2))
    elif(num1 < num2):
        print(str(num1) + " is smaller than " + str(num2))
    else:
        print(str(num1) + " is equal to " + str(num2))

# All 3 function calls will print out: "1 is less than 2"
num_comparator(1, 2)
num_comparator(num1=1, num2=2) # This is a keyword argument.
num_comparator(num2=2, num1=1) # This is also a keyword argument.
```

  - One can think of the kwargs as being a dictionary that maps each keyword to the value that we pass alongside it. That is why when we iterate over the kwargs there doesn't seem to be any order in which they were printed out.
    E.g.

```
def myFun(**kwargs):
    for key, value in kwargs.items():
        print (key + ": " + value)

# Driver code
myFun(first ='Hi', mid ='World')
```

These are the outputs:

```
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop/CSCB20 Code/Args and Kwargs$ python3 kwargs.py
mid: World
first: Hi
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop/CSCB20 Code/Args and Kwargs$ python3 kwargs.py
first: Hi
mid: World
ricklan@DESKTOP-148J53H:/mnt/c/Users/rick/Desktop/CSCB20 Code/Args and Kwargs$
```

Flask-Login:

- **Note:** You need to download the flask-login package to use it. To install it, you can do pip install flask-login.
- Flask-Login provides user session management for Flask. It handles the common tasks of logging in, logging out, and remembering your users' sessions over extended periods of time.
- It will:
  - Store the active user's ID in the session, and let you log them in and out easily.
  - Let you restrict views to logged-in (or logged-out) users.
  - Handle the normally-tricky "remember me" functionality.
  - Help protect your users' sessions from being stolen by cookie thieves.
- However, it does not:
  - Impose a particular database or other storage method on you. You are entirely in charge of how the user is loaded.
  - Restrict you to using usernames and passwords, OpenIDs, or any other method of authenticating.
  - Handle permissions beyond "logged in or not."
  - Handle user registration or account recovery.
- The most important part of an application that uses Flask-Login is the LoginManager class. You should create one for your application somewhere in your code, like this:
  login_manager = LoginManager()
- The login manager contains the code that lets your application and Flask-Login work together, such as how to load a user from an ID, where to send users when they need to log in, and the like.
  I.e. LoginManager is used to hold the settings used for logging in.
- Once the actual application object has been created, you can configure it for login with:
  login_manager.init_app(app)
- By default, Flask-Login uses sessions for authentication. This means you must set the secret key on your application, otherwise Flask will give you an error message telling you to do so.
- The User class is the class that you use to represent users needs to implement the following properties and methods. You can inherit the methods from UserMixin.
- These are the methods:
  - **is_authenticated:** This property should return True if the user is authenticated. I.e. they have provided valid credentials.
  - **is_active:** This property should return True if this is an active user, in addition to being authenticated, they also have activated their account, not been suspended, or any condition your application has for rejecting an account.
  - **is_anonymous:** This property should return True if this is an anonymous user. Actual users should return False instead.
  - **get_id():** This method must return a unicode that uniquely identifies this user, and can be used to load the user from the user_loader callback. Note that this

must be a unicode. If the ID is natively an int or some other type, you will need to convert it to unicode.

- Views that require your users to be logged in can be decorated with the **login_required** decorator:
  E.g.

```
@app.route("/settings")
@login_required
def settings():
    pass
```

  E.g. When the user is ready to log out:

```
@app.route("/logout")
@login_required
def logout():
    logout_user()
    return redirect(somewhere)
```