Announcements

- Final project proposal due this Wed (11/25)
  - Submit it on Sakai

- Try to get help from me and tutors

- Second midterm on 12/3 in class
  - A sheet of paper, 8.5 x 11 (both sides, handwritten or typed) allowed but nothing electronic is allowed
  - Scope: focus on the material covered since first midterm excluding server-side programming although some concepts on server-side programming may be asked as T/F or multiple choice questions.

- Break around 10:15am
Optional and named parameters in python

- Python allows function arguments to have default values
- Named arguments can be specified in any order

```python
def info(a, b=20, c=30):
    print a, b, c

info(1)                   # 1 20 30
info(1, 2)               # 1 2 30
info(1, 2, 3)           # 1, 2, 3
info(1, c=3)           # 1, 20, 3
info(1, c=3, b=2)   # 1, 2, 3
```
The Request-Response Cycle
HTTP protocol

- Client and server communicates using a communication protocol called **HTTP protocol**
- HTTP protocol has **GET** and **POST** commands among others
- We will see how GET and POST commands are handled in server side programming
Request and response objects

- When Flask receives a request from a client, it needs to make a few objects available to the view function that will handle the request.
- `request` object encapsulates the HTTP request data (IP address, browser type, GET/POST, etc.) sent by the client.
- `response` object encapsulates the HTTP response sent by the server.
- Flask makes these objects globally accessible thus making view functions like the following possible:

```python
@app.route('/

def index():
    user_agent = request.headers.get('User-Agent')
    return '<p>Your browser is %s</p>' % user_agent
```
Request dispatching

- When an application receives a request from a client, it needs to find what view function to invoke to service it.

- Flask looks up the URL given in the request in the application’s URL map based on the `@app.route` decorators, e.g.,
  ```python
  @app.route('/index')
  ```
  is mapped to the view function `index()`.
Responses

- When Flask invokes a view function, it expects its return value to be the response to the request.
- In most cases the response is a simple string that is sent back to the client as an HTML page.
- HTTP protocol also requires a status code to be returned, e.g.,
  - **200**: successful
  - **400**: bad request
  - **404**: page not found
  - etc.
- Code can be added like this:

  ```python
  @app.route('/
  def index():
    return '<h1>Bad Request</h1>', 400
  ```
Response object

- Flask can also return a response object containing multiple values

- Use make_response() from flask module to create a response object

- Example: You can create a response object and set a cookie in it

```python
from Flask import make_response

@app.route('/
def index():
    response = make_response('<h1>A cookie returned</h1>')
    response.set_cookie('answer', '42')
    return response
```
Web Forms
Web forms

- Web forms are essential for a web app
- The request object exposes all the info sent by the client with a request
- For example, the following form obtains the name and hobby info from the user on a web browser and sends the info along with a POST request; the server side program identified by action handles the request.

```html
<form action="{{ url_for('hobby') }}" method=POST>
  <b>Your name:</b> <input type=text name=name><br>
  <b>Your hobby:</b> <input type=text name=hobby><br>
  <input type=submit value='Find Info'>
</form>
```
View function handling a form

- `request.form` provides access to the form data submitted in **POST** requests.
- A view function uses the request object to access the form data using `request.form[...]`, e.g., in `forms.py` you see the following:

  ```python
  @app.route('/hobby', methods=['POST'])
  def hobby():
      your_name = request.form['name']
      your_hobby = request.form['hobby']
      return render_template('find_info_for_hobby.html',
                              name=your_name,
                              hobby=your_hobby)
  ```
The data obtained from the form is passed to a template and is used there, i.e., in `find_info_for_hobby.html` (note that this creates another form of its own):

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

{% block body %}
  <h3><b style="color:red">{{ hobby }}, {{ name }}!?!</b>: Have fun!</h3>

  <form action="http://www.google.com/search"
       method="GET">
    <input type="text" name="q" value="{{ hobby }}" />
    <input type="submit" value="Google Search"/>
  </form>

{% endblock %}
```
A complete example

- Understand the **forms** example on the course web structured as follows

  - **forms**
    - forms.py
    - static
      - style.css
    - templates
      - demo_form.html
      - find_info_for_hobby.html
      - layout.html

- Indentation in the previous bullets shows the file folder hierarchy
Basic elements needed for Flask

• In the forms example in forms.py you will see the following and they are required for a Flask app. So, include them in your Flask app:

```python
# Configuration
SECRET_KEY = 'development key or any key of your own'

# Create an application
app = Flask(__name__)
app.config.from_object(__name__)
```
Clarification

- In a Flask app, things in the .py file(s) are executed on the server.
- The things in the static folder are downloaded to a client machine and get executed on the client machine.
- The things in the templates folder are executed on the server initially to create something to be executed on a client later.
  - See what the HTML file that server generates (by a call to the render_template function) looks like by “view page source” on your web browser.
  - Your web browser received the HTML file from the server which was generated by render_template!
- For example, in the forms example, the code in forms.py is executed on the server; the code in style.css, demo_form.html, find_info_for_hobby.html, and layout.html is executed on the server to create code (i.e., HTML) to be executed yet again on a client.