Lecture 13: Flask Models and Forms

Flask so far:

- Routes, and the request-response cycle
- MVC and templates
    - Flask uses Jinja2 templates to generate HTML that is passed to the browser
    - {{ ... }} and {% .... %} are the holes and logic that are rendered into a complete HTML page by Flask's `render_template` module.
    - Templates are the "View" part of MVC
        - Though, confusingly, the functions you assign to routes are called "view functions"

```
@app.route("/")
def index():        # This is a "view function". But it's really part of the controller
  return "hello"
```

Today:

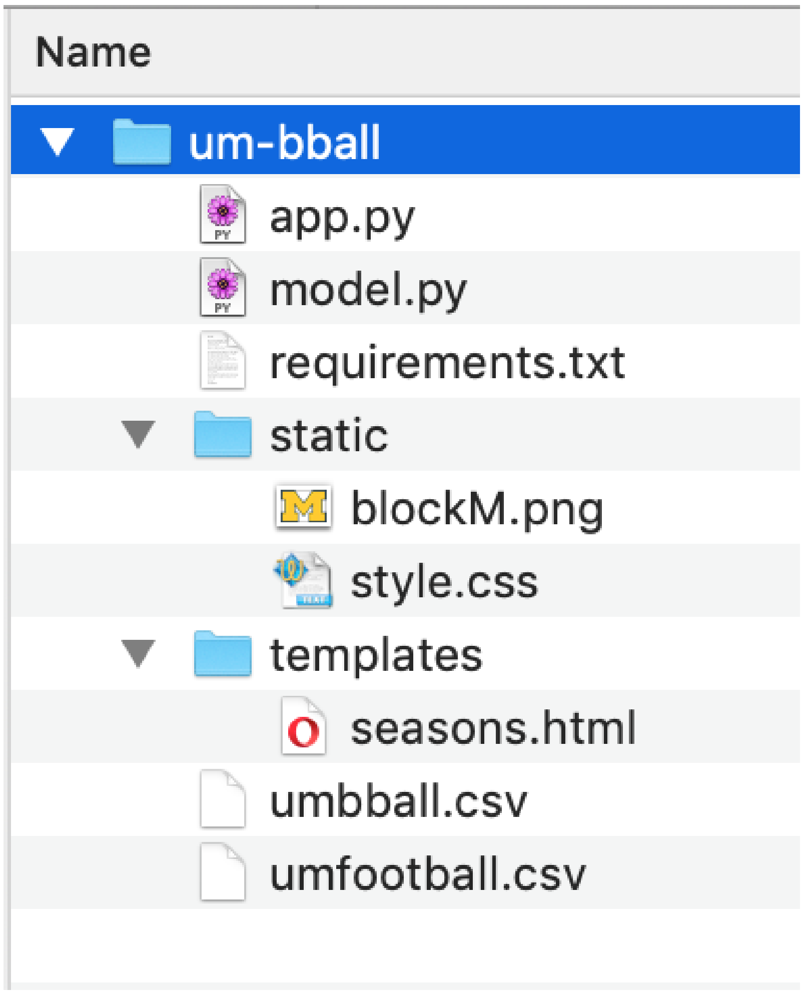
- Models
- Handling user input using Forms (and redirection)
- Creating a real(-ish) web app
- Making your web app available on the Internet

Model

The model is the part of an application that manages the data. To demonstrate, we will build a simple app to display historical information about the 🏀 University of Michigan Men's Basketball Team 🏀 .

To start, download the starting code from here. Unzip the zip file and move the resulting folder into the folder you created for today's lecture. Like last time, if you are on a Mac, make sure there are no spaces in your file path.

Now let's look at the contents:



Here's the stuff you've seen before:

- app.py — this is the main application program file
- templates/seasons.html — this is a template we'll be using. Take a look at the contents:

```html

<link rel="stylesheet" href="/static/style.css">

<h1> Michigan Men's Basketball </h1>

<table border=1>
    <tr>
        <th>Year</th>
        <th>Wins</th>
        <th>Losses</th>
        <th>Win %</th>
    </tr>
{% for s in seasons %}
    <tr>
        <td>{{s[1]}}</td>
        <td>{{s[3]}}</td>
        <td>{{s[4]}}</td>
        <td>{{s[5]}}</td>
    </tr>
{% endfor %}
</table>
```

    - The meat of this template is the table, and the meat of the table is the for loop that will print out one row `<tr>` with four cells `<td>` each. By looking at the table header you can figure out what each cell will contain (and also by looking at the csv file, which we'll get to shortly).
- requirements.txt — this contains the dependencies for this app.
    - Create a virtual environment called `bballenv` and use the requirements.txt file to set it up correctly.
- static/blockM.png and static/style.css — static files, referenced in `app.py:index()` and in `seasons.html`.

Here are the parts that are probably unfamiliar:
- model.py — this will be the file that contains data management code. We'll be looking at this shortly.
- umbball.csv and umfootball.csv — two CSV files that contain information about men's basketball and football seasons since each program began. You can see the information contained, but we'll just be focusing on a few of the columns: year, wins, losses, and win percentage.

Let's look at the app.py program. Everything in there should look familiar. But notice how the seasons parameter is being populated to get passed to the seasons.html template.

```
from flask import Flask, render_template
import model
```

```

app = Flask(__name__)

@app.route('/')
def index():
    return '''
        <img src="/static/blockM.png"/>
        <h1>Michigan Sports Info!</h1>
        <ul>
            <li><a href="/bball"> Men's Basketball </a></li>
        </ul>
    '''

@app.route('/bball')
def bball():
    return render_template("seasons.html", seasons=model.get_bball_seasons())

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

```

Let’s run the program and test the 2 routes. You’ll notice that nothing prints out in the table for the `bball` route. Why?

Let’s look at the model.py file:

```
#model.py
import csv

BB_FILE_NAME = 'umbball.csv'

bb_seasons = []

```

```
def init_bball(csv_file_name=BB_FILE_NAME):
    pass

def get_bball_seasons(sortby='year', sortorder='desc'):
    global bb_seasons
    return bb_seasons

```

Let's work on the model, so it actually sends to the view data it needs to populate the table.

Here's what we need to do:
1. Read the CSV from file into a program data structure (`bb_seasons`)
2. Implement the function `get_bball_seasons()` in `model.py` that provides access to that data structure. (For now it will just return the list.) — this is already done for you!

Read the CSV
Implement `init_bball()`. This should be just standard CSV reading. We'll also add some code to `app.py` that will run `init_bball()` .

To test it for now, we can add a line of code that will get the data using `get_bball_seasons()` and print the resulting list to the console (yes, you can still do this in Flask for debugging).

in model.py, we will modify the init_bball() function as follows:

```
def init_bball(csv_file_name=BB_FILE_NAME):
    global bb_seasons
    with open(csv_file_name) as f:
        reader = csv.reader(f)
        next(reader) # throw away headers
        next(reader) # throw away headers
        bb_seasons = [] # reset, start clean
        for r in reader:
            bb_seasons.append(r)
```

in app.py, we'll change the end of the file to call init_bball():

```python
if __name__ == '__main__':
    model.init_bball()
    app.run(debug=True)
```

Test the route.

You just made an MVC application!
To review:

- The "Model" manages the data. In our application, model.py is the only part that knows that the data came from a CSV or knows how to manage CSVs.
- The "View" manages the presentation of data. In our application, the template `seasons.html` is the only part that knows about HTML, tables, stylesheets, etc. (Well, actually the default route function `index()` has some hard-coded HTML, but we normally wouldn't do that).
- The controller maps user input (clicking on links) to output (what is displayed), accessing data from the model as needed. In our application, this happens when the `bball()` function gets the seasons list from the model and passes it to the view (the `seasons.html` template).

Forms

Next, we'll add some user interface elements to our application that will allow the user to control the displayed output by sorting the displayed stats in different ways. We will do that by implementing an HTML form in our application.

To understand how this is going to work, though, we need to go over a few things:

- HTTP request messages
- Defining an HTML Form
- Getting POST data from the `request` object

HTTP requests

Communication between browsers (or other web clients) and web servers is defined by the HTTP protocol. This communication follows a *request-response model*:

- the client reaches out to the server by sending a *request* message
- the server then responds to the client by sending the client a *response* message.
- how request and response messages are structured is defined by the HTTP protocol.

Usually, when you write applications, you don't have to craft request and response messages by hand. Libraries like the python `Requests` module or frameworks like Flask automate much of this work for you. However, there are a few details you *do* need to understand.

A request message has several parts. Among these are:

- request header
- message body

It's a little more complicated than that, but these are the ones we normally need to worry about. If you are interested in the gory details, you can find them here: https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html

There are several types of request messages, corresponding to what the request is intended to do:

- A request might initialize a communication session between the client and the server
- A request might ask the server for some information (i.e., it can ask for a resource)
- A request might send information to the server to create or update a resource.
- A request might tell the server to delete a resource

The type of the request is called its method, You can read a summary of the various HTTP request methods here:
https://www.w3schools.com/tags/ref_httpmethods.asp

The two methods we need to work with are GET and POST:

- GET method is used to request a resource from the server.
  - All information about the resource that is requested is passed to the server in the request header. In other words, message body of a GET request does not contain anything.

    - When we used python's `Request` module to get data from various websites (via an API or to scrap a page), we were sending those API end points and web sites GET requests.
- POST method is used to send information to the server so it can create or update a resource
    - For example, when you're filling in your name, address, and billing information on an e-commerce site, when you hit Submit that information is packaged as a POST request and sent to the server.
    - POST requests contain information in the message body. That's how a POST request packages up the information that the user typed into a web form.

Creating forms

To see how this all works, let's create some forms. We'll work with a simple example first.

We will do the following:

1. Create a template that includes an HTML Form.
2. Create a view function that gets data from the HTML Form and passes it back to the template for display.

Create a template

In the templates folder, create a new template called `hello.html`. It should have this content:

```
<h2> What is your name? </h2>

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

    First Name: <input type="text" name="firstname"/> <br/>
    Last Name: <input type="text" name="lastname"/> <br/>

    <input type="submit" value="submit"/>

</form>
```

What is happening here?

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

- The `<form>` tag defines an HTML form inside an HTML page
- The `action` attribute defines what happens when the form is submitted—specifically what URL the data should be sent to.
- The `method` attribute defines how the form data should be sent to the URL. As we discussed, there are two main options:
    - GET doesn't send anything inside the message body. The only information inside a GET request is the URL itself, which might contain various kinds of parameters.
        - In the case of Form data send via GET, the data would be sent to the server in the form of a query string.
    - POST sends the form data "inside" the request message body. The server has to extract the contents of the message to make use of it.
        - For forms, the entered data is typically sent via a POST request. That's what we'll be doing here.

```
    First Name: <input type="text" name="firstname"/> <br/>
    Last Name: <input type="text" name="lastname"/> <br/>
```

- We are defining two form elements inside the form. They are both "input" elements, meaning they accept input from the user. They are both of type "text", meaning they are text boxes (there are other types, like radio buttons and check boxes). Each input element has a "name", which we will need later to extract the data that the user put in. (you can think of this as if the user's input is being assigned to a variable name.)

```
    <input type="submit" value="submit"/>
```

- We are defining one more element: a submit button. The "submit" type is a special input type that, when clicked, submits the form to the URL specified in the form's `action` attribute.

Render the template

For now, we'll just display the template and ignore the form data. We'll add a new route to do this:

```
@app.route('/hello', methods=['GET', 'POST'])
def hello():
    return render_template("hello.html")
```

- The only thing new here is `methods=['GET', 'POST']`. This means that this route can handle both GET and POST requests. We'll see why this matters shortly.

Update the template to handle variables we will initialize from the form
We are not actually handling the POST data yet, we're just getting ready to do so.

Replace

```
<h2> What is your name? </h2>
```

with

```
{% if firstname == '' %}
<h2> What is your name? </h2>
{% else %}
<h2> Hello {{firstname}} {{lastname}} </h2>
Would you like to change your name? <br/>
{% endif %}
```

This is doing two things:

1. Changing the greeting string based on whether or not the firstname is present. This allows this template to work the first time, when the user hasn't entered a name.
2. Displaying the name that the user entered and prompting them to enter another name.

Update the "view" function to handle form data
To get data from the form, we'll need to access Flask's `request` object. The `request` object contains information about the request that triggered the route. It contains information such as the URL of the page the user came from, the type of browser the user is using, and a bunch of other stuff.

For our purposes, what we care about is the fact that the `request` object contains the form data that was submitted by the user.

We can access the form data like this:

```
    firstname = request.form['firstname']
    lastname = request.form['lastname']
```

So you can infer from this that `form` is a dictionary inside of the `request` object that contains key-value pairs where they key is the `name` of the form element and the value is the data that was entered by the user.

To add this to our application we need to first add `request` to our imports:

```
from flask import Flask, render_template, request
```

And then add the code to access the form data and pass it to the template:

```
@app.route('/hello', methods=['GET', 'POST'])
def hello():
    if request.method == 'POST':
        firstname = request.form['firstname']
        lastname = request.form['lastname']
    else:
        firstname = ''
        lastname = ''

    return render_template("hello.html", firstname=firstname, lastname=lastname)
```

Now try it out!

Adding a form to our Basketball app

We are going to give the user the ability to sort the basketball stats in different ways. We will need to:

1. Add a form to the template
2. Update the `bball()` function in app.py to deal with the form data and extract the right information from the model.
3. Update the model (specifically `get_bball_seasons()`) to return the list sorted in different ways.

Adding a form

We'll use this form:

```
<form action="/bball" method="POST">
    Sort By:<br/>
    <input type="radio" name="sortby" value="year"/> Year <br/>
    <input type="radio" name="sortby" value="wins"/> Wins <br/>
    <input type="radio" name="sortby" value="pct"/> Win Percentage <br/>
    <br/>
    Sort Order:<br/>
    <input type="radio" name="sortorder" value="asc"/> Ascending <br/>
    <input type="radio" name="sortorder" value="desc"/> Descending <br/>
    <br/>
    <input type="submit" name="update" value="Update"/>
    <br/>
    <br/>
</form>
```

Add this to `seasons.html` *under* the <h1> and *above* the <table>.

You can test this out to see what it looks like.

A new thing here, for those of you new to HTML forms, is the "radio" type for <input> elements. You can see how this works by playing with it in the HTML page.

Notes:

- Radio buttons have to be in "groups" that all have the same "name" but different values.
- Only one radio button for a group can be selected a time
- The name=value pair that will be submitted with the form corresponds to the selected button when the form is submitted.
- This form has two radio button groups: `sortby` and `sortorder`. It has a submit button too (though we've changed the name and value to

change what is displayed on the button).

Updating `bball()`

We need to extract the data from the form and use it to change the data we ask the model for.

```python
@app.route('/bball', methods=['GET', 'POST'])
def bball():
    if request.method == 'POST':
        sortby = request.form['sortby']
        sortorder = request.form['sortorder']
        seasons = model.get_bball_seasons(sortby, sortorder)
    else:
        seasons = model.get_bball_seasons()

    return render_template("seasons.html", seasons=seasons)
```

Updating `model.py`

The code we just wrote for `bball()` will break the application, because we added two parameters to `model.get_bball_seasons()` that the function isn't expecting. Let's go and fix that now.

```python
def get_bball_seasons(sortby='year', sortorder='desc'):
    if sortby == 'year':
        sortcol = 1
    elif sortby == 'wins':
        sortcol = 3
    elif sortby == 'pct':
        sortcol = 5
    else:
        sortcol = 0

    rev = (sortorder == 'desc')
    sorted_list = sorted(bb_seasons, key=lambda row: row
[sortcol], reverse=rev)
    return sorted_list
```

You may not have seen the `sorted` function in a while, so may be fuzzy on what is happening with `key=lambda row: row[sortcol]` . Basically this is just telling the `sorted` function which element within the row to use for sorting. `lambda` is just a nifty way to define a "temporary" function within another python statement.

Note that we have provided default values for `sortby` and `sortorder` so that this function will work even when the user hasn't specified a sort order (as will happen the first time the page is accessed).

Try it out!

You'll notice that things don't quite work, especially when you sort by wins. This is because when we imported our CSV, everything was imported as a string. We need to change this so that wins is an integer, so that the sorting will work as expected.

Easy enough. Add some code to `init_bball` to fix the data types when we're reading things in. Here's a complete implementation of `init_bball`, just take the parts you need to update your version:

```
    with open(csv_file_name) as f:
        reader = csv.reader(f)
        next(reader) # throw away headers
        next(reader) # throw away headers
        global bb_seasons
        bb_seasons = [] # reset, start clean
        for r in reader:
            r[3] = int(r[3])
            r[4] = int(r[4])
            r[5] = float(r[5])
            bb_seasons.append(r)
```

Lecture Exercise

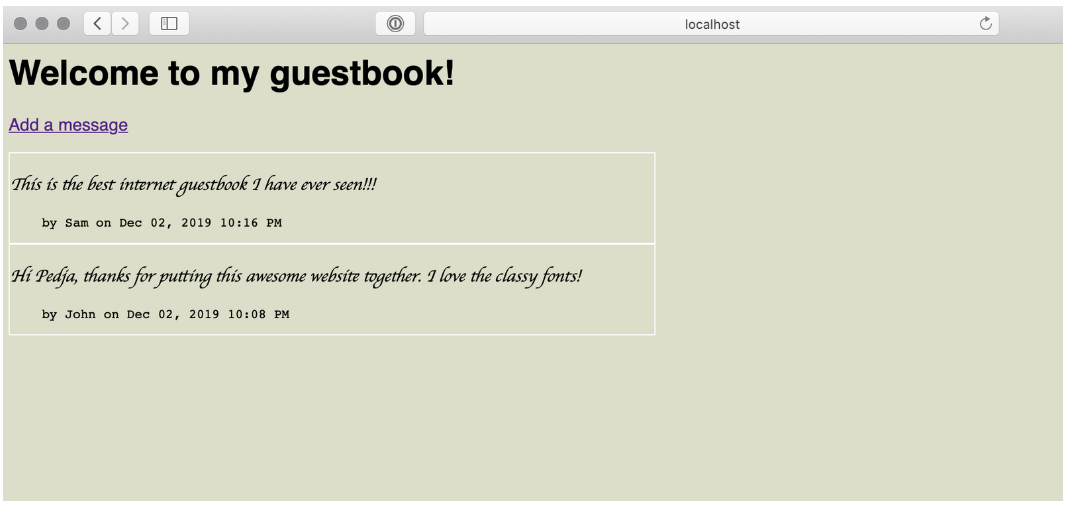
- Change the app to also allow the user to display and sort football stats.
    - `localhost:5000/football`
    - Can you get it to work so that both football and basketball stats are using the same template, seasons.html?

Updates and Persistent Storage

So far, the web apps we built either pulled the data from the web (the word rhyming app) or displayed data from a fixed/unchanging data file. Most apps you would want to build, however, would have more complex functionality where data is both read and written/saved in response to user interaction. in the next example we'll see how we would handle such cases.

A Guestbook App

To illustrate this functionality, we'll create a guestbook app that allows people to leave you a message. It will look like this (don't you love the tacky fonts!?):



Welcome to my guestbook!

Add a message

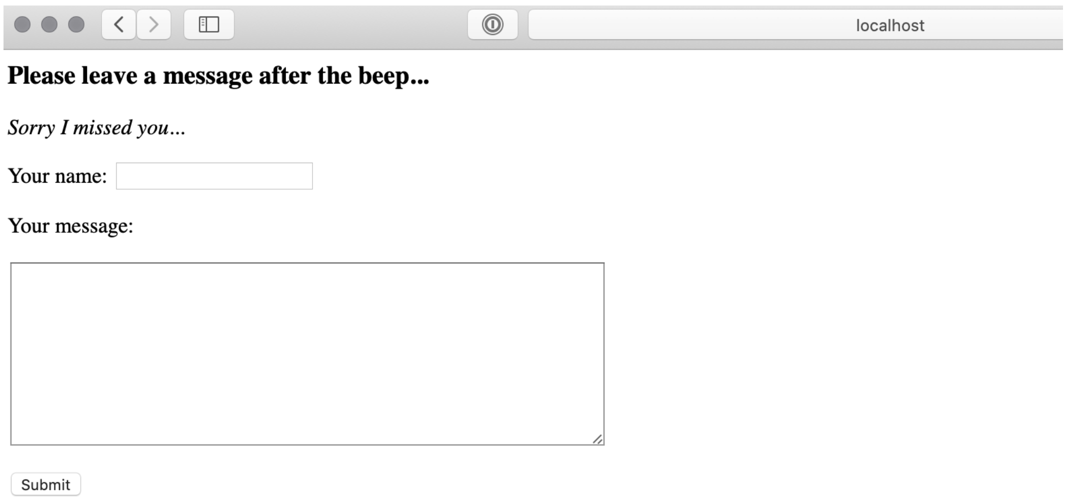
This is the best internet guestbook I have ever seen!!!

by Sam on Dec 02, 2019 10:16 PM

Hi Pedja, thanks for putting this awesome website together. I love the classy fonts!

by John on Dec 02, 2019 10:08 PM

If you click the "Add a new message" link, you'll be shown a form that allows you to create a new guestbook entry:



Please leave a message after the beep...

Sorry I missed you...

Your name:

Your message:

You can download the full code here. Unzip the file and then follow along as we walk through what we are doing here.

Here is our checklist of how this will work:

View

- Template for showing the guest book
- Template for allowing a visitor add a message
- A style sheet to make it look amazing

Model

- A list of entries
- Init function
- Add entries
- Access function (to get the list of current entries)

Controller

- / route — display the guest book
- /add route — display message entry form
- /postentry route — receive post data from the /add route

Setup

- create a virtual environment
- activate it
- install flask

View

In /templates:

- index.html
- addentry.html

index.html

```
<link rel="stylesheet" href="/static/style.css">

<h1> Welcome to my guestbook! </h1>

<p><a href="/add">Add a message</a></p>

{% for e in entries %}
    <div class="entry">
        <p class="entrytext"> {{e['text']}} </p>
        <p class="metadata"> by {{e['author']}} on {{e['timestamp']}} </p>
    </div>
{% endfor %}
```

addentry.html

```
<h3> Please leave a message after the beep...</h3>
<i>Sorry I missed you...</i>
<form action="/postentry" method="POST">
    <p>Your name: <input type="text" name="name"/></p>
    <p>Your message: </p>
    <p><textarea name="message" rows=10 cols=60></textarea></p>
    <p><input type="submit" value="Submit"/></p>
</form>
```

In /static

- style.css

style.css

```
body {
    background-color: rgb(220, 222, 200);
    font-family: Helvetica, Arial, sans-serif;
}

.entry {
    border-width: 1px;
    border-color: white;
    border-style: solid;
    width: 600px;
}

.entrytext {
    font-family: cursive;
    font-style: oblique;
    font-size: 16;
}

.metadata {
    font-family: monospace;
    font-size: 12;
    padding-left: 30;
}
```

Model

Create model.py. We're going to make it so that we write the list of entries to a file, so that it will persist across application runs.

model.py

```
import json
from datetime import datetime

GUESTBOOK_ENTRIES_FILE = "entries.json"
```

```python
entries = []

def init():
    global entries
    try:
        f = open(GUESTBOOK_ENTRIES_FILE)
        entries = json.loads(f.read())
        f.close()
    except:
        entries = []

def get_entries():
    global entries
    return entries

def add_entry(name, text):
    global entries, GUESTBOOK_ENTRIES_FILE
    now = datetime.now()
    time_string = now.strftime("%b %d, %Y %-I:%M %p")
    # if you have an error using this format, just use
    # time_string = str(now)
    entry = {"author": name, "text": text, "timestamp": 
time_string}
    entries.insert(0, entry) ## add to front of list
    try:
        f = open(GUESTBOOK_ENTRIES_FILE, "w")
        dump_string = json.dumps(entries)
        f.write(dump_string)
        f.close()
    except:
        print("ERROR! Could not write entries to file.")
```

Note: dates and time can get kinda complicated. Python has a package called `datetime` to deal with them. Here's how we're using it here:

```
from datetime import datetime
```

Seems kinda repetitive, doesn't it? datetime is a class (or a submodule, I'm not sure) inside of the datetime package. There are several other modules/classes inside datetime, so we need to import it like this.

```
    now = datetime.now()
```

gives you a datetime object representing the current time

```
    time_string = now.strftime("%b %d, %Y %-I:%M %p")
```

datetime.strftime() converts a datetime object to a string. It uses formatting specifiers. You can read more about the options here: https://docs.python.org/3/library/datetime.html

The rest of the model file should be fairly self-explanatory.

Controller

Our app.py will look like this:

```
from flask import Flask, render_template, request, redirect
import model

app = Flask(__name__)

@app.route("/")
def index():
    ## print the guestbook
    return render_template("index.html", entries=model.get_entries())

@app.route("/add")
def addentry():
    ## add a guestbook entry
    return render_template("addentry.html")
```

```

@app.route("/postentry", methods=["POST"])
def postentry():
    name = request.form["name"]
    message = request.form["message"]
    model.add_entry(name, message)
    return redirect("/")

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

The only thing really new here is `redirect`. This basically allows one route to redirect to another route. Here, after we add the entry to the list, we redirect back to the main application page, which will display the updated guestbook.

Now run it!

Lecture exercise (if time permits)
Modify the guestbook app to include a route /images that will show a list of gifs and images that visitors have linked to. You will need to make a few changes:

- Modify the index.html view to add a link to the /images page. You can do that above or below the “Add an entry” link at the top of the page
- Create a view images.html that will be shown when a visitor goes to /images. This template should show linked images one at a time, at a reasonable size, along with the name of the visitors that posted the links. In essence, you are creating an image version of the main guestbook page.
- Create an addimage.html template that visitors can use to provide their name and a url of an image on the internet
- Modify the model.py file to read/write `images.json` file that will be storing entered names and URLs.
- Modify the app.py to add the view functions for the required routes: /images /addimage and any helper route you may need.

You can model all of this on the code you already have for the guestbook app!

Some notes on deployment

So far, we have been just developing our web apps on our laptops. If you wanted to put your app (e.g., the guestbook app) on the internet, so anyone could access it through a web browser, you would need to deploy it to one of the web app hosting services. There are quite a few hosting services, including Amazon Web Service, Google App Engine, Heroku, Microsoft Azure, etc. Each of these services has its own requirements for deploying, and the procedures don't generalize in any meaningful way.

The flask website has links to instructions for various services that can be found here:
https://flask.palletsprojects.com/en/1.1.x/deploying/

In previous semesters, 507 students were asked to deploy their apps to Heroku, and Mark Newman wrote up the documentation for how to do that. You can find that documentation here:
+Deploying Flask Apps with Heroku

Going through the steps in class doesn't have a whole lot of pedagogical value, though, since the steps are different for each hosting environment and it comes down to following the instructions on the web page for the environment you want to use. So, we'll skip going through this process.

Instead, we'll finish by going over a few basic concepts that you will need to understand in order to deploy python-based web apps:

Each hosting environment for web applications uses the same general components, and for some of those components it will give you a choice of which of the various options to use. The components include the following:

- **Web server.** This is the software that receives web requests from the internet and sends requested web pages back to the browsers of users who are accessing your site. Some hosting services will let you choose which web server you want to use, while others will have a standard configuration for this. The most common options for this component are:

- Apache: a hugely popular open source web server. It's used by something like half of all web sites on the internet.
- NginX: A highly scalable web server used by Netflix, Bank of America, and a whole bunch of other big web sites. Heroku uses this as its standard web server.
- Microsoft IIS: Microsoft's web server that works best with web apps written in .NET framework, but it can support other types of web applications as well.

- **Web Server Gateway Interface (WSGI) server.** This is software that sits between a web server and a web app written using a python-based web framework (flask, django). Standard web servers don't understand how to deal with python applications. WSGI servers understand how python frameworks work, so they handle the handoff back and forth between a python app and the web server facing the internet. Sometimes WSGI servers are implemented as an extension to a web server, but today the norm is to use a dedicated WSGI server. You can read more about this, if you are interested, here: https://www.fullstackpython.com/wsgi-servers.html Note that the little server Flask uses for debugging is a combination of a WSGI server and a very simple web server. Some common WSGI servers used by web hosting environments are:
  - Green Unicorn (http://gunicorn.org). This server is used by Heroku, so if you are deploying your flask app there, you need to use this server.
  - uWSGI (https://uwsgi-docs.readthedocs.io/en/latest/). A new, high-performing WSGI server.
- **Database server.** As we discussed a few weeks ago, this is the server that hosts your database so you can have a fast, scalable way of reading and writing data. For python-based web apps written using flask or django frameworks, the most common options include:
  - MySQL (http://www.mysql.com). A scalable, open source relational database system commonly used with web applications.
  - PostgreSQL (https://www.postgresql.org). Another open-source option commonly used with python.
  - MongoDB (https://www.mongodb.com). Mongo is a "no-SQL" database that uses a document-oriented storage model, rather than being a rational database. For some situations, this is preferable (read: faster). Details are outside the scope of this class, but I wanted you to know that no-sql databases exist.

Deploying to a hosting environment involves moving your code to your account on AWS, Heroku, etc., and, as necessary, configuring your app to work with the servers you chose to use (or need to use). You can see how that looks for Heroku in the example linked above. Here is a link to what's involved in running your flask app on AWS: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create-deploy-python-flask.html

Again, it's all a matter of carefully following the directions and troubleshooting things until you get it to work.

And with that, go forth and build some (web) apps! Thanks for spending the semester with us!