

Flask cheat sheet v4.1

Naming things is hard

First, come up with *two* names for your project: one which will be referred to as “the site name,” and one as “the app name.” The difference between these two names, and why there even *are* two names, is weird and confusing. Just do your best and come up with two names. (If your site has to do with basketball, for example, you could use “hoopsSite” for the site name and “basketball” for the app name.)

From this point forward in the instructions, whenever I say to type `mysite`, substitute your site name (e.g., `hoopsSite`). Whenever I say to type `myapp`, substitute your app name (e.g., `basketball`).

Install

Python virtual environment

It’s just good practice to use virtual environments.

```
$ mkdir mysite
$ cd mysite
$ python -m venv venv
$ source venv/bin/activate
```

git repo

It’s just good practice to use a git repo.

```
$ git init .
```

Also create a `.gitignore` file using vim, with these contents:

```
.gitignore
venv
__pycache__
```

Also create a `README.md` file, also using vim, with these contents:

give it these contents:

A sample Flask app. (...or whatever you want it to say...)

and commit it to your repo:

```
$ git add README.md
$ git commit -m "Initial commit."
```

Make sure you can run the “`git log`” and “`git status`” commands correctly before continuing.

Flask

```
$ pip install flask
```

Tip: when you create your alias to start up Flask, have it include “`source venv/bin/activate`” before “`export FLASK_APP=myapp`” and “`flask run`”. The alias in my `.bashrc` for starting the Ben & Jerry’s project looks like this:

```
alias run="cd /mydir ; source venv/bin/activate ; export FLASK_APP=bj ; pwd ; flask run"
```

After I’ve got all the stuff below setup, I will then be able to simply type “`run`” at the command line and it will start my Flask server at port 5000.

“Hello world”

Create scaffolding

Make a new directory `mysite` (if you didn’t just do that), and create these files inside it:

- `__init__.py` with contents:

```
from flask import Flask
myapp = Flask(__name__)
from mysite import routes
```

- `myapp.py` with one line: “`from mysite import myapp`”
- `routes.py` with contents:

```
from mysite import myapp
@myapp.route("/")
def index():
    return "Hello world!"
```

Run “hello world”

Either execute your alias (in my case, “`run`”), if you made one as suggested above, or else type:

```
$ export FLASK_APP=myapp
$ flask run
```

Then go to `http://localhost:5000`.

(Sometimes you’ll see people instead pip install `python-dotenv` and then add “`FLASK_APP=myapp.py`” in a new `.flaskenv` file in the `mysite` directory. This way, you can simply type “`flask run`” to start your app, without having to specify `FLASK_APP` each time. Either way works. I prefer my alias. If you do this `.flaskenv` thing, you’ll probably want to include “`.flaskenv`” in your `.gitignore` file.)

If you’re running your code on a server, then you’ll need to instead use this command on the server to start Flask:

```
$ flask run -h 0.0.0.0 -p yourPortNumber
```

Then go to `http://theServerYoureRunningOn:yourPortNumber`.

Basic dynamic web page

Flask uses the Jinja template engine, which has lots of powerful and convenient features. Here's the very very basics:

1. Create a `templates` directory in `mysite`.
2. In `templates`, create a `basic.html` with double-curly (“durly”) markup:

```
<HTML>
...
    I have {{ numPairs * 2 }} total shoes.
...
</HTML>
```

3. In `routes.py`:

```
from flask import render_template
...
def index():
    ...
    return render_template("basic.html", numPairs=17)
```

Now restart flask, refresh your browser, and see what that did.

-
4. Can iterate through collections in a template like this:

```
<HTML>
...
    {% for hobby in hobbies %}
        ...stuff in here with tables, lists, etc., using "hobby"...
    {% endfor %}
...
</HTML>
```

-
5. Can use if-statement-like syntax in a template like this:

```
<HTML>
...
    {% if hobby == "skydiving" %}
        ...stuff...
    {% elif hobby == "paperclip collecting" %}
        ...stuff...
    {% else %}
        ...stuff...
    {% endif %}
...
</HTML>
```

Generating URLs

A very common need in a dynamic web page is to generate a URL (often to another dynamic web page). The basic HTML syntax for this uses the `<a>` (for “anchor”) tag (`Go UMW!`).

But in a Jinja template, it is convenient to call the Flask `url_for()` function to accomplish this. Its first argument is the *function* in your `routes.py` that you want to URL to go to. Other (keyword) arguments may follow, in which case `url_for()` will automatically escape any necessary characters and glom them on as HTTP parameters to the end of the URL.

For example, if you had this in your `routes.py`:

```
def show_ingr():
    the_recipe = request.args['recipe']
    ...
```

and this in a Flask template:

```
Love Mom's <a href="{url_for(show_ingr,recipe='lemon pie')}}">lemon pie</a>!
```

then when rendered, the template will produce this output:

```
Love Mom's <a href="/show_ingr?recipe=lemon+pie">lemon pie</a>!
```

which is exactly what you want.

Beyond the basics

There’s lots, lots more [Jinja features](#) to explore.

Using template inheritance

Suppose you have an HTML file called “`duck.html`” in your `templates` directory that looks like this:

```
<html>
    ...stuff...
    {% block silly %} some default silly text {% endblock %}
    ...more stuff...
    {% block biscuit %} some default biscuit text {% endblock %}
    ...yet more stuff...
</html>
```

and another one called “`daffy.html`” that looks like this:

```
{% extends "duck.html" %}
    {% block silly %} special, daffy-specific silly text {% endblock %}
```

If a browser accesses the `daffy.html` page, it will receive this HTML payload:

```
<html>
    ...stuff...
    special, daffy-specific silly text
    ...more stuff...
    some default biscuit text
    ...yet more stuff...
```

```
</html>
```

You'll see that most of the content comes from the `duck.html` template, not the `daffy.html` template. In fact, the page returned is literally the `duck.html` template, with the `{% block X %}` and `{% endblock %}` Jinja tags removed, and with `daffy`'s special "silly" block content substituted for `duck`'s. (The "biscuit" block's text remains unchanged from the "duck" template, because it was not overridden by `daffy`.)

This is called "**template inheritance**" because it bears a vague resemblance to inheritance in object-oriented programming. The "duck" template in this example is referred to as a "**parent template**" (a.k.a. "base template," a.k.a. "superclass template") and "daffy" is called a "**child template**" (a.k.a. "derived template," a.k.a. "subclass template.") Note that the child template can choose to provide substituted content for all of the parent's `blocks`, or only some of them.

HTML forms

The simplest way to get user input to a web page is through an **HTML form**. This is a special element with sub-elements, and generally looks like this:

```
<form action="theHandlerURL" method="GET" >

  What's your name? <input type="text" name="nickname" /><br/>

  What's your favorite color? (choose one)
  chartreuse <input type="radio" name="faveColor" value="chartreuse" /><br/>
  fuschia <input type="radio" name="faveColor" value="fuschia" />
  indigo <input type="radio" name="faveColor" value="indigo" />

  Which foods do you like? (check all that apply)
  haggis <input type="checkbox" name="likedFood" value="haggis" /><br/>
  hummus <input type="checkbox" name="likedFood" value="hummus" />
  okra <input type="checkbox" name="likedFood" value="okra" />

  Choose an adjective to describe you:
  <select name="adjective">
    <option value="mellifluous">mellifluous</option>
    <option value="mendacious">mendacious</option>
    <option value="perspicacious">perspicacious</option>
  </select>

  <input type="submit" name="submit" value="That's my stuff!" />

</form>
```

This example shows the four basic types of interactive form widgets: **text boxes** for typing open-ended text, **radio buttons** for choose-one-of-these situations, **check boxes** for choose-some-of-these situations, and **drop-downs** which are also normally used for choose-one-of-these situations (when there are a lot of options).

Note that all of these are `<input>` elements *except* drop-downs, which have their own nested HTML structure.

The last `<input>` element, of type “submit”, presents a button for the user to click on when they want to submit their information. When the user clicks it, the browser will pack all of user’s inputs - no matter what kind of widget was involved - as parameters to the URL given in the “action” parameter, and call that URL. These can be retrieved on Flask using `request.args`, a dictionary-like object available in any route. (Be sure that `routes.py` imports the `request` object from the Flask package.) You can give the name of the input widget as a key to `request.args` (for example, `request.args["faveColor"]`) and get the value the user chose.

Some odds and ends:

- You won’t normally be hardcoding a specific URL into the “action” parameter, but rather asking Flask to create a URL for you, with the syntax “`{{ url_for('nameOfPythonHandlerFunction') }}`”. So, this is an appropriate form tag:

```
<form action="{{ url_for('personalDataHandler') }}" method="GET" >
```

..... assuming you have this in `routes.py`:

```
@myapp.route("/personalDataHandler")
def personalDataHandler():
    ...
    ...do stuff with request.args...
```

- For radio buttons, you have a bunch of widgets with the *same* name. (That’s how the browser knows to mutually-exclusive-ize them.) To get those values on the server, use `request.args.getlist("nameOfRadioButtonInputs")` instead of treating `request.args` like a dict.
- If you don’t want the inputs to appear in the URL, you can use `method="POST"` instead of `method="GET"`. If you do this, you’ll need to (1) add a `methods=["POST"]` argument to your `@myapp.route()` annotation, and (2) use `request.form` instead of `request.args` in your route code to get the values. In other words, these things happily go together:

```
<form action="{{ url_for('someOtherDataHandler') }}" method="POST" >
```

..... assuming you have this in `routes.py`:

```
@myapp.route("/someOtherDataHandler", methods=["POST"])
def someOtherDataHandler():
    ...
    ...do stuff with request.form...
```

“Hidden form variables”

Sometimes a situation arises when you need the browser to send additional information to the server beyond what’s in the form widgets you explicitly present. (In class, this came up when we wanted to let the user order ice cream from the `recipedetails` page, but the form only had the number of cartons, not the recipe name itself.) To address this, you can “smuggle in” another `<input>` element, of type `hidden`:

```
<form action="theHandlerURL" method="GET" >
```

```
    ...input widgets, text, submit button, etc...
```

```
    <input type=hidden name=recipe value="{{ recipename }}" />
```

...

</form>

In this example, we're having the Jinja template engine replace the `value` attribute of this `<input>` element with whatever the `recipe` variable has in it. Now, when the user presses submit, the server will get not only the stuff they entered in the widgets, but also a `recipe/Chunky Monkey` key/value pair in `request.args`.

Redirects

A common pattern in Web programming is handling input from a form, doing something with it (like updating a database), and then *sending the browser to a different page*. This is best accomplished with the Flask `redirect()` function, which works like this:

```
@myapp.route("/yetAnotherDataHandler")
def yetAnotherDataHandler():
    ...
    ...do stuff...
    return redirect(url_for('thePythonFunctionForThePageToGoTo'))
```

This is called a **redirect**. There's also another technique for doing a similar kind of thing, called a **forward**, but I don't like it as well so I'll make you look it up if you want to know about it.

Session management

Client-side storage

In encrypted cookie value, stored in each browser, dutifully returned to the server whenever another request is sent.

1. In order to enable this, you need to have this one line somewhere in your `__init__.py`:

```
myapp.secret_key = "Some random string better than this!"
```

2. Import `session` from `flask`, and then this variable is available to you as a dict-like object holding key/value pairs *only* for the current user. (All other users have their own key/value session pairs which do not interfere with each other.) This magic is achievable through cookie passing.

Server-side storage

Coming soon!

An important weirdness with sessions “knowing” they’ve been updated.

The issue is this. If you change one of the key/value pairs in the session — and by that I mean you change **which object a key is referring to** — then Flask “knows” you changed the session and dutifully sends the updated version to the client for storage, as it should.

However, if you only modify what's **in** one of the object values, rather than changing which object that key points to, Flask plays dumb and thinks you haven't changed the session.

For this reason, you'll need to add this line of Python code (see p. line 25 of routes.py in the class github repo):

```
session.modified = True
```

on the line immediately after changing merely the *contents* of a collection, rather than substituting an entirely different collection object some session value. (And if that's confusing, realize that `session.modified = True` is always a safe operation.)