Lecture 12: Virtual Environments and Intro to Flask

Today

- Virtual environments: why, what, and how
- Intro to Flask
  - Getting started
  - View functions
- Some programming patterns
  - Decorators
  - The Request-Reponse cycle
- Frameworks and libraries
- Model View Controller (MVC) frameworks
- Using templates to create Flask views

Virtualenvs

What's the need?

Common problems:

- You what to try a python program someone else wrote. How do you figure out what packages you need to install in order to be able to run it?
- You are working on two (or more programs). They both require the same package, *but* they require different versions of the package. What do you do?
- We (instructors) will need to grade your final projects, many of which will use different packages. How can we do this efficiently?

All of these problems point to a need for a flexible package management. Virtualenvs aim to address this problem.

How do things work now?

Over the course of the last couple of semesters, you've (probably) installed lots of packages. How many? Let's see...

```
$ pip freeze
```

or

```
$ pip3 freeze
```

This is what I got:

```
beautifulsoup4==4.6.3
bs4==0.0.1
certifi==2018.8.24
chardet==3.0.4
decorator==4.3.0
idna==2.7
ipython-genutils==0.2.0
jsonschema==2.6.0
jupyter-core==4.4.0
nbformat==4.4.0
numpy==1.15.3
plotly==3.3.0
pytz==2018.6
requests==2.19.1
retrying==1.3.3
six==1.11.0
traitlets==4.3.2
urllib3==1.23
```

You probably got quite a few entries as well.

What happens when we do a pip(3) install?

- modules are downloaded to our computer and put... somewhere?
- Let's find out where.

```
$ python (or python3)
>>> import sys
>>> print (sys.path)
```

On my Mac I get:

```
['', '/Library/Frameworks/Python.framework/Versions/3.7/lib/python37.zip', '/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7', '/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/lib-dynload', '/Library/Fram
```

```
eworks/Python.framework/Versions/3.7/lib/python3.7/site-pac
kages']
```

On a PC, you would get something like this:

```
['', 'C:\\Program Files (x86)\\Python36-32\\python37.zip',
'C:\\Program Files (x86)\\Python37-32\\DLLs', 'C:\\Program
Files (x86)\\Python37-32\\lib', 'C:\\Program Files (x86)\\P
ython37-32', 'C:\\Program Files (x86)\\Python37-32\\lib\\si
te-packages']
```

- sys.path is the list of directories that python searches to find modules.
- Note that the first item is `''`--this means that Python always looks in the current directory (no path) first. The rest of the paths are *absolute* paths—they start from the root of your OS's file system (`/` on Mac, `C:` on Windows).
- The directory that pip uses is the one that ends in `site-packages`. Let's look at it:

On mac:

```
ls /Library/Frameworks/Python.framework/Versions/3.7/lib/py
thon3.7/site-packages'
```

On windows (note the quotes, and also the removal of extra back slashes; also, if the last directory in the output of sys.path is different for your machine, copy that one and remove extra slashes):

```
dir "C:\Program Files (x86)\Python37-32\lib\site-packages"
```

In Git Bash (if you happen to be using it) it works with the extra backslashes:

```
ls 'C:\\Program Files (x86)\\Python36-32', 'C:\\Program Fil
es (x86)\\Python37-32\\lib\\site-packages'
```

You should see some (a lot!) of similarity between what you saw in pip freeze and what you're seeing in this directory. That's where pip(3) puts things when you pip install a new package.

The problem with pip revisited

- Over time, you have created a personalized python environment that works for you. That's great!
- However, as we mentioned, there can be problems:
  - How can you share a program with someone else?
    - (e.g., your final project)

- Run a program on a different computer?
  - (e.g., building a Flask web app and installing it on a web host)
- Module dependencies
  - Example:
    - You write program A that depends on module B, which depends on module C
    - You write program D that depends on module E, which also depends on module C
    - A new version of module C is released. Module B is updated, but module E is not.
  - As you get deeper into python, this situation can start to arise somewhat regularly

The solution: Virtual Environments

A virtual environment is like a parallel universe where your programs can operate, with all the packages it needs, isolated from everything else.

- You can have as many of these parallel words as you need, each one of them with a completely different set of packages and/or versions of those packages.
- Each virtual environment can track what packages/versions it contains, so another instance (e.g., on another machine) can be easily created.
- These files that track package requirements can be shared with the code (they are just output of `pip freeze`!), so someone else can easily recreate the setup that's needed to run the code.

Let's create our first virtual environment.

Note that when working with virtual environments, we're going to be working in the Terminal/Command Prompt/git bash (i.e., not Python).

1. Install virtualenv

On all platforms:

```
$ pip install virtualenv
```

or

```
$ pip3 install virtualenv
```

2. create a new directory for your project.

Important Note: on a Mac the full path of your directory can't have any spaces in it!!!

```
$ cd <the parent of your new directory>
```

```
$ pwd
/Users/klasnja/Dropbox/Teaching/SI507/Examples
```

This is OK

```
$ pwd
/Users/klasnja/Dropbox/Teaching/SI507/Code examples
```

This is not OK. It won't work. Find a place to work that won't have any spaces. Note that this is not an issue in Windows.

```
$ mkdir flask
$ cd flask
```

3. create a virtual environment. I'll call it `flask1`

```
$ virtualenv flask1
```

Note: if this doesn't work (and you are on a Mac), type this instead:

```
/Library/Frameworks/Python.framework/Versions/3.7/bin/virtualenv flask1
```

4. look at what just got created. Mac/git bash:

```
$ ls
$ ls flask1
$ ls flask1/bin
```

Windows:

```
> dir
> dir flask1
> dir flask1\Scripts
```

5. Now we will "activate" our virtual environment.

On Mac:

```
$ source flask1/bin/activate
```

On Windows:

```
> flask1\Scripts\activate.bat
```

On git bash:

```
$ source flask1/Scripts/activate
```

This is the weirdest and least natural part of working with a virtual environment. You are invoking command line scripts here, and you're doing it using a relative path from your current directory. If you end up in a different directory and try this it won't work.

6. You should now see a change to your command prompt, telling you that you're in a virtual environment. It should look like this (Mac):

```
(flask1) m-c02qw2c8fvh7:flask klasnja$
```

or, on Windows, something like this (or whatever directory you are in):

```
(flask1) C:\Users\m-mislem1\Code\si507\lec12>
```

Basically, you should see the name of your virtual environment pre-pended to your normal prompt, to remind you that you are running in a virtual environment.

Notes:

- The virtual environment is only active in the terminal/command prompt in which you activated it.
- Virtual environments are an OS thing, not a python thing, though the particular way they set up your OS is all about supporting python programs.
- Basically, a virtual environment sets the path for your OS to first search the directory that was created when you created the virtual environment, and there it puts copies and links/shortcuts to the various things python needs to work. It also changes python's search path to store/look for site packages in the virtual environment.

To explore this a bit more. Open a second terminal/prompt. Type the following into each:

```
$ pip freeze
$ echo $PATH (or echo %PATH% in Windows CMD)
$ python (NOTE: you don't need to say python3!!!)
>>> import sys
>>> print (sys.path)
```

Can you spot the differences? None of the packages you had previously installed are in the new virtual environment!

When you're done working with a virtual environment, you just deactivate it:

```
$ deactivate
```

We'll come back to virtualenvs in just a bit.

Introduction to Flask

What is Flask?

- Flask is a *micro web framework* for Python
- Web frameworks are software frameworks (i.e., software plumbing that allows you to more easily write code that does complex things—we'll come back to this later today) that handle many of the under-the-hood details that are required to run a web application.
- There are a number of other web frameworks. For python, the most popular full-featured web framework is Django. Other popular ones are Ruby on Rails (which uses the programming language Ruby), and ASP.NET, Microsoft's web framework built on their .NET development framework.
  - Web frameworks handle many things needed to run a web app, including the communication between the app and the browser and, often, database access (stuff we were doing directly in SQL this term).
- Flask is a *micro* framework, because it doesn't have some of the things that full-features web frameworks have, such as a database abstraction layer (so you can write the same code to talk to the database regardless of what actual database you are using) or form validation.
- But Flask can be extended with additional functionality through packages that integrate into it.
- Here are some resources on Flask for later reference:
  - Quick start guide
  - Tutorial that walks you through creating a simple blog using Flask
  - Wikipedia entry that gives you a little bit of history of Flask.

Getting started with Flask

Let's start by creating a simple "Hello, World!" web app in Flask.

- Activate your virtual environment. Verify that there are no modules installed.
- Install Flask

```
$ pip install flask (note: you don't need to say pip3!!)
```

- verify that it was installed correctly

```
$ python
(note that you don't need to say python3 either! in the virtual environment, python is a sim link to/shortcut for python3)
>>> import flask
>>> exit()
```

You should get no errors when you try to import flask.

Now create a new python file called `helloflask.py` and write:

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

@app.route('/')
def index():
    return '<h1>Hello World!</h1>'

if __name__ == '__main__':
    print('starting Flask app', app.name)
    app.run(debug=True)
```

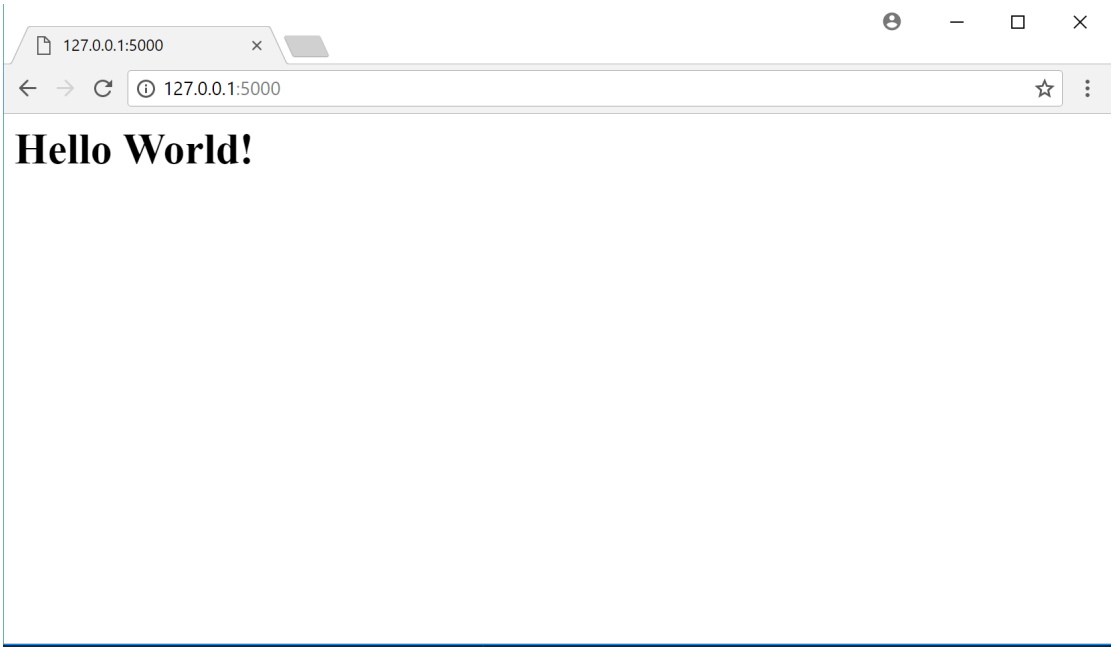
Then run it:

```
$ python helloflask.py

starting Flask app helloflask
 * Restarting with stat
starting Flask app helloflask
 * Debugger is active!
 * Debugger PIN: 212-671-065
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Hmmm. What happened?
You've just started a web server to run the app we just wrote!
It's "Running on http://127.0.0.1:5000/"

Open up a browser tab and type http://127.0.0.1:5000/ into the address bar. You should see:



What just happened?

1. You created a flask app by instantiating an instance of the Flask class.
2. You gave it the name `__name__`
    a. since you doing this from the main program (i.e., you din't create a flask app in a program that's being called from another program), the `__name__` variable will take the name of the main program, in our case `helloflask.`

3. You created a "view" function called "index()" and assigned it to the route '/'.

```
@app.route('/')
def index():
    return '<h1>Hello World!</h1>'
```

We'll come back to this in a second.

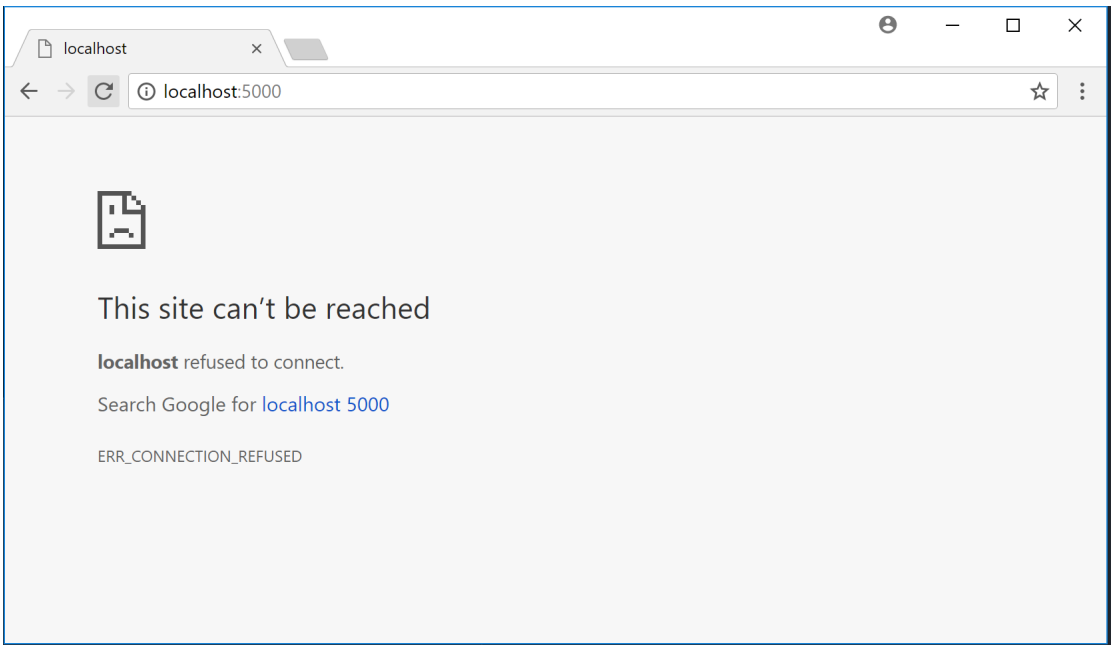
4. You told flask to run your app.

```
if __name__ == '__main__':
    print('starting Flask app', app.name)
    app.run(debug=True)
```

Flask obliged and started up a local server to host your app. It started up that server on port 5000 of the "localhost" (i.e., your computer) which is always assigned the special IP address 127.0.0.1.

4. You used a web browser to look at your app, running on the Flask server.
    a. Note: Most browsers will let you also type in 'localhost:5000' to get to this server, as well as 127.0.0.1:5000.
    b. Another note: the format `IP address:port` is how you tell your browser to use a non-default port. If you leave it blank, it will default to port 80 for http or port 443 for https. Flask intentionally uses a non-standard port so as not to conflict with any actual web servers you might be running on your computer.

Now kill Flask by typing Ctrl-C in the terminal/prompt. Refresh your browser. It shouldn't work anymore.



View Functions

Let's look at this more closely:

```
@app.route('/')
def index():
    return '<h1>Hello World!</h1>'
```

To understand what's happening here, we need to understand two programming patterns, at least a little bit:

- The "decorator" pattern
- The "request response" cycle

Decorators

- A "decorator" is a function that "wraps" another function. Usually this is done to add some functionality to the "wrapped" function without

changing it.

- A common use: you might want to add some special functionality to a particular instance of a class, without creating a whole new subclass just for that instance.
- For example, f you have a instance of the Dog class, Bo, who's a medical service dog, you might want to wrap the dog instance Bo in a decorator that would give Bo some abilities that other dogs don't have (e.g., wake up owner if her blood sugar is low at night). By using a decorator for that, you don't affect any other instances of the Dog class, just Bo.

The details for how this is done are beyond the scope of this course (though I encourage you to look it up), but the basic idea is this:

Say I have a function:

```
def my_func():
  print('hello')
```

And I want to wrap it with a decorator, I can create another function that will do some stuff, then call my function, then maybe do some other stuff:

```
def my_func():
  print("hello")

def my_decorator(dec_func):
  print('calling the decorated function')
  dec_func()
  print('called the decorated function')

my_decorator(my_func)
```

You'll notice that I passed the name of my function `my_func` as an argument to the decorator `my_decorator`, and then (indirectly) called it from within `my_decorator`.

The actual decorator pattern gets a little bit more complicated than this, but this is the basic idea.

The decorator pattern turns out to be so useful that it is supported natively by Python using the `@` syntax.

So in this code:

```
@app.route('/')
def index():
    return '<h1>Hello World!</h1>'
```

`app.route( )` is *decorating* `index( )`. In particular, it is adding to the `index()` function a mapping to the web route `/`, which is the "root" of the web server (i.e., the page that you get if you don't specify a page or a path).

To see this in a bit more detail, let's add another route:

```
@app.route('/about')
def about():
    html = '''
      <h1>About this Site </h1>
      <p> This is my first ever Flask website! </p>
      <a href='/'> Go back home </a>
      '''
    return html
```

Try it out! Go to http://localhost:5000/about and see what you get.

Note that functions that are tied to routes (in our examples, to `/` and `/about`) return a string that contains the HTML that the browser should display when you go to the corresponding page.

Of course if all we ever did was print out static HTML there would be no point in using Flask (you could just shove HTML files onto a server). Let's add some simple logic:

```
ctr = 0
@app.route('/counter')
def counter():
  global ctr
  ctr += 1
  return '<h3>' + str(ctr) + '</h3>'
```

Try this out. Go to http://localhost:5000/counter and hit refresh a bunch of times!

Why does this work? To understand this, we need to get a basic understanding of another programming pattern, the request-response cycle.

The Request Response Cycle

Whenever a client (e.g., browser) makes a request to a web server, a bunch of stuff happens:

1. client sends request to server
2. server receives the request
3. server parses different parts of the request
4. server separates the route from the parameters
    a. "route" is the web page that is being requested (e.g., /section/science is a route on the www.nytimes.com website)
    b. Parameters are the information that is being submitted to a route (e.g., you sent a bunch of parameters to the various API endpoints you were using in this class)
5. server looks for the requested route
6. if it finds a route, it does one of the following:
    a. returns to the client the web page corresponding to the route (for a static site)
    b. delegates handling to the web application, passing along the parameters (for a web app)
7. In the case of a web app, the application processes the request and returns a response
8. server packages the response and sends it to the client as a web page

Flask does everything for you but #7 (and #1, which is the browser's job).

Flask applications you write have the job of providing handlers for particular types of requests (i.e., requests to different "paths" on the server). They do that that by registering those handlers to *routes*.

The app logic and state is being handled by your program.

- In our example, the state is simple, it's just the value of the global variable `ctr`
- The important thing to note is that your web browser knows nothing about how many times you visited the counter page.
    - When you refresh the page, browser just send another request to the server for that same route/path

- Server passes it to the appropriate function in your app (this is Flask's job)
- Your app is keeping track of the number of times visited, it augments the counter variable and then sends back (via flask) a string containing html that displays the current count.
- Server passes this static html to the browser that displays it to you. (this is again flask's job)

Lecture exercise

Create a Flask app to introduce yourself. Call the app file `me.py`

- the index (`/`) should show your name and some info about you (e.g., your hometown, year in school, etc.) and optionally a photo (use the `img` tag to point to a picture of you on the web, if there is one)
- add at least two other pages with other bits of information (classes you're taking, your family, your hobbies, your resume?). Link all of the pages together so that you can get to each of the other pages from each page.
- Add a counter to the home page (index) that displays:

  This site has been visited X times!

Virtualenvs, continued

requirements.txt

To share information about what modules need to be installed for your application to run, you create a file called `requirements.txt`. You will very often see these documents in python github repositories that you come across.

To generate requirements.txt, make sure you are working in an activated virtual environment (in our case, the one where you installed flask)

Now you do:

```
(flask1) $ pip freeze > requirements.txt
```

This writes the output of `pip freeze` into a file. The `>` operator in the command prompt is the "redirect" operator, and it tells the shell to write its output to a file instead of to the screen.

Look at the contents. On a Mac:

```
$ more requirements.txt
```

On Windows:

```
> type requirements.txt
```

setting up an environment using requirements.txt
So let's say you want to run someone else's program. Here's how you'd do that:
(before you do this, deactivate your current environment by typing `deactivate` in the shell)

1. Create a new virtual environment (go ahead! you can make as many as you want! they're free!)

```
(flask1) $ deactivate
$ virtualenv flask2
```

2. Activate the virtualenv

```
$ source flask2/bin/activate
```

or on Windows

```
> flask2\Scripts\activate.bat
```

3. Verify that there are no modules installed by pip, and then do a pip install from requirements.txt:

```
(flask2) $ pip freeze
(flask2) $ pip install -r requirements.txt
(flask2) $ pip freeze

click==6.7
Flask==0.12.2
itsdangerous==0.24
Jinja2==2.10
MarkupSafe==1.0
```

```
10 Werkzeug==0.14.1
```

... is what you should see.

Further reading: Just in case you want to know more about how pip deals with requirements and how you can have more control over what goes into requirements.txt, you can read the spec.

Announcement

You need to generate a requirements.txt file for your final project! We will need this in order to be able to run your project!

To sum up, here are the steps:

1. Create and activate a virtual environment
2. pip install the modules you need for your project
   a. If you started working on the project before you created the virtual environment, just keep trying to run it and read the error messages to figure out which modules you still need to install
3. When all the needed modules are installed, create the requirements.txt file (as described above)
4. Be sure to upload `requirements.txt` along with your final project to Canvas.
5. If you add more modules, you'll need to repeat this to create an updated requirements.txt. It is recommended that you re-generate your requirements.txt as a final step before submitting your project. Just in case.

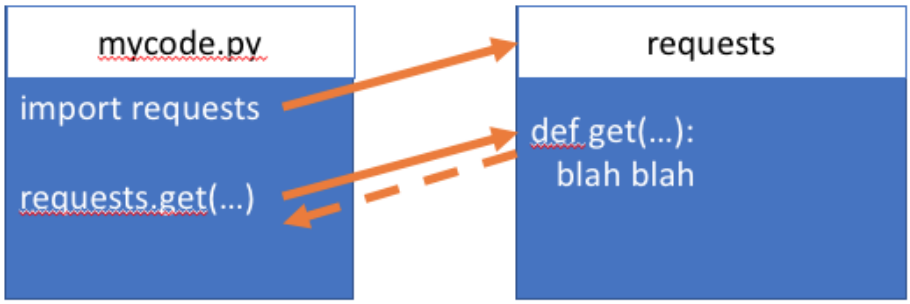
Frameworks and Libraries

and modules and packages

Brief sidebar:

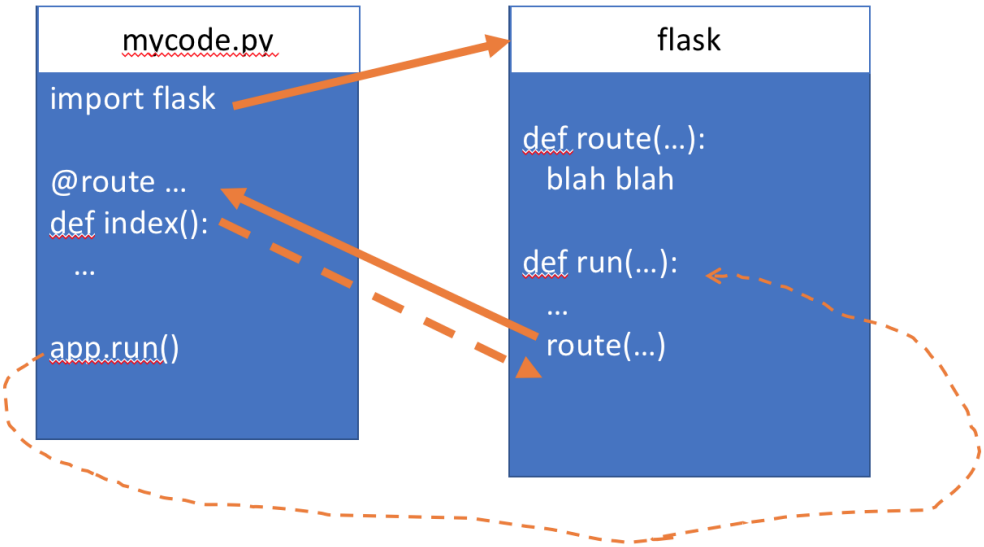
- When you add functionality to Python using an `import` statement, you are including functionality from a *module* or a *package*
  - A package has multiple modules in it. A module is a single python file.
- But a module or package can be related to your code in different ways. So far, we have been importing modules and packages as *libraries.*

Libraries: your code calls into the library. Your code is in charge of the overall control flow.



There's another way. Sometimes you interact with 3rd party code where the code provides a *framework*.

Unlike with libraries, where you call into the library, a framework calls into your code. The framework is in charge of the control flow.



In the case of Flask, Flask spins up a web server and handles all interactions with the browser. It just calls your code when the browser asks for a page that your app has registered to handle.

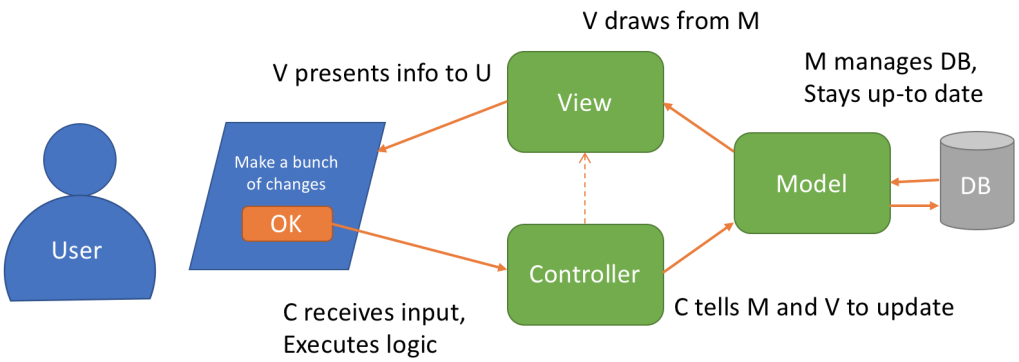
Model-View-Controller

Early on in the development of graphical user interfaces (Smalltalk-76, to be exact), it was recognized that code gets pretty complex pretty quickly. A typical interaction looks like this:

Each of these steps can get complicated (think about some of the sql you've been writing recently). Mixing the code for the database (update and fetch) and control flow (which screen to show next) and user interface (configuring and displaying the particular screen contents) tends to make programs hard to read, maintain, and debug.

Also, it's often the case that the people writing the UI code are not the same as the people writing the database access code, so you want to keep those parts separate to support collaborative coding.

The Model-View-Controller architectural pattern is one way to separate these concerns.



In a nutshell:

- The Model is responsible for keeping the underlying program data up-to-date, including updating the database. It is the only part that knows about how the data is stored or accessed. In addition, the model typically encodes the overall application business logic (what the app can do and how)
- The View is responsible for displaying the data to the user (i.e., providing output). It doesn't need to worry about keeping the data up to date, it just

trusts the Model that it will get the right data to display. It is the only part that knows about details of the UI, like the layout, fonts, graphics, etc.
- The Controller handles input and decides when data needs to be updated or a view needs to be changed.

MVC and Flask

- MVC, or its extensions, is extremely prevalent in application development frameworks, and most modern web frameworks follow this pattern.
  - Flask is no exception, though Flask does not take a particularly strong stance in terms of forcing you into this model (unlike, say, RoR).
- In Flask, the controller and model are python files
  - Your controller is your main program file. It sets up the routes and contains the logic that maps input events to changes in the model and the view.
  - Your model will typically be another python file that deals with the data and provides functions that allow the view and controller to update and access the data. Flask leaves up to you how you want to implement this. We'll look at models next week.
- In Flask, the *view* is handled by *templates*

Templates

Modern Web UIs are built using HTML, CSS, and JavaScript. They can get very complex, as you saw when you inspected pages you were trying to scrape.

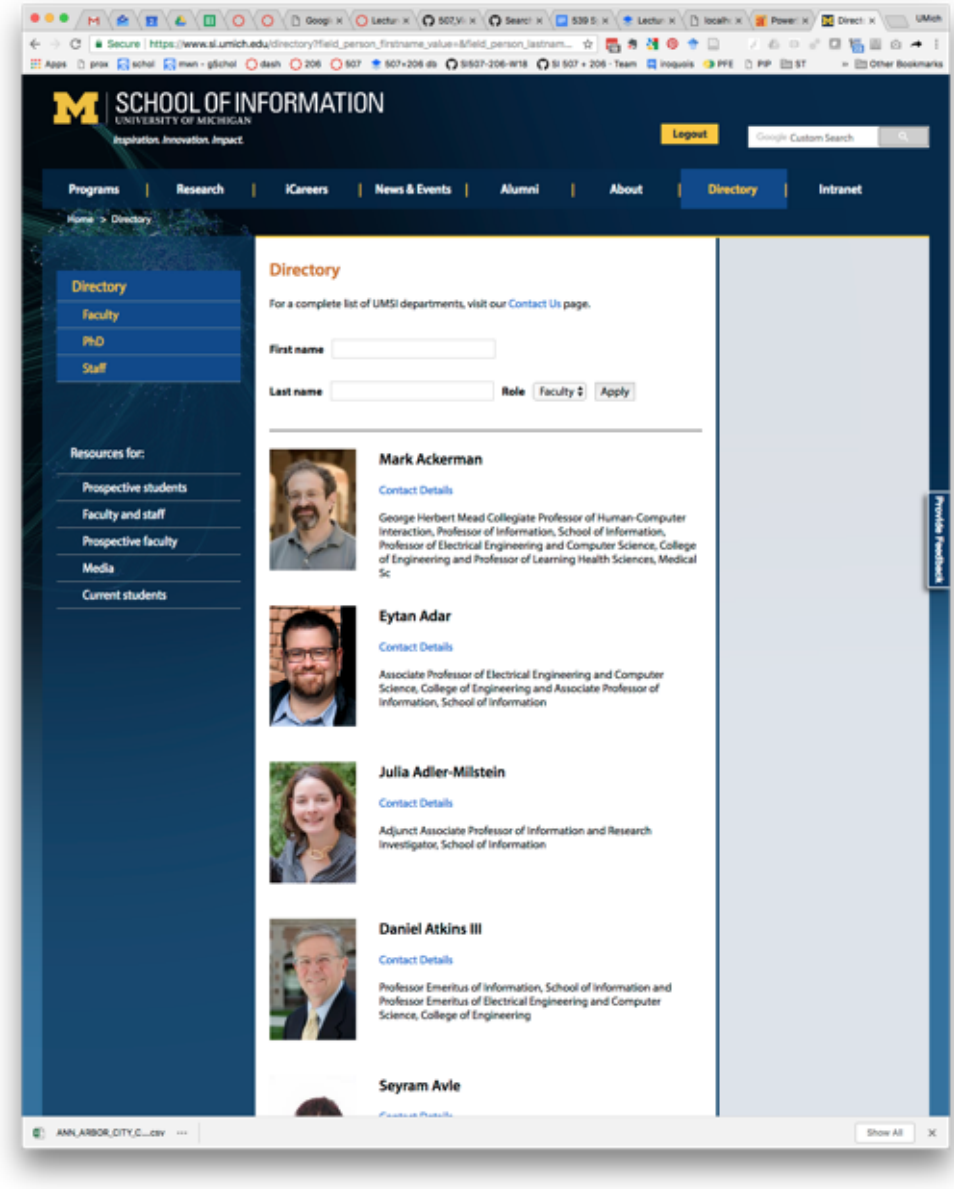
Writing modern web UIs as python strings (e.g., "<h1>Hello World!</h1>") inside of view functions would be totally insane.

Templates let you write real UI code, that is controlled by, and gets its data from, your controller and model.

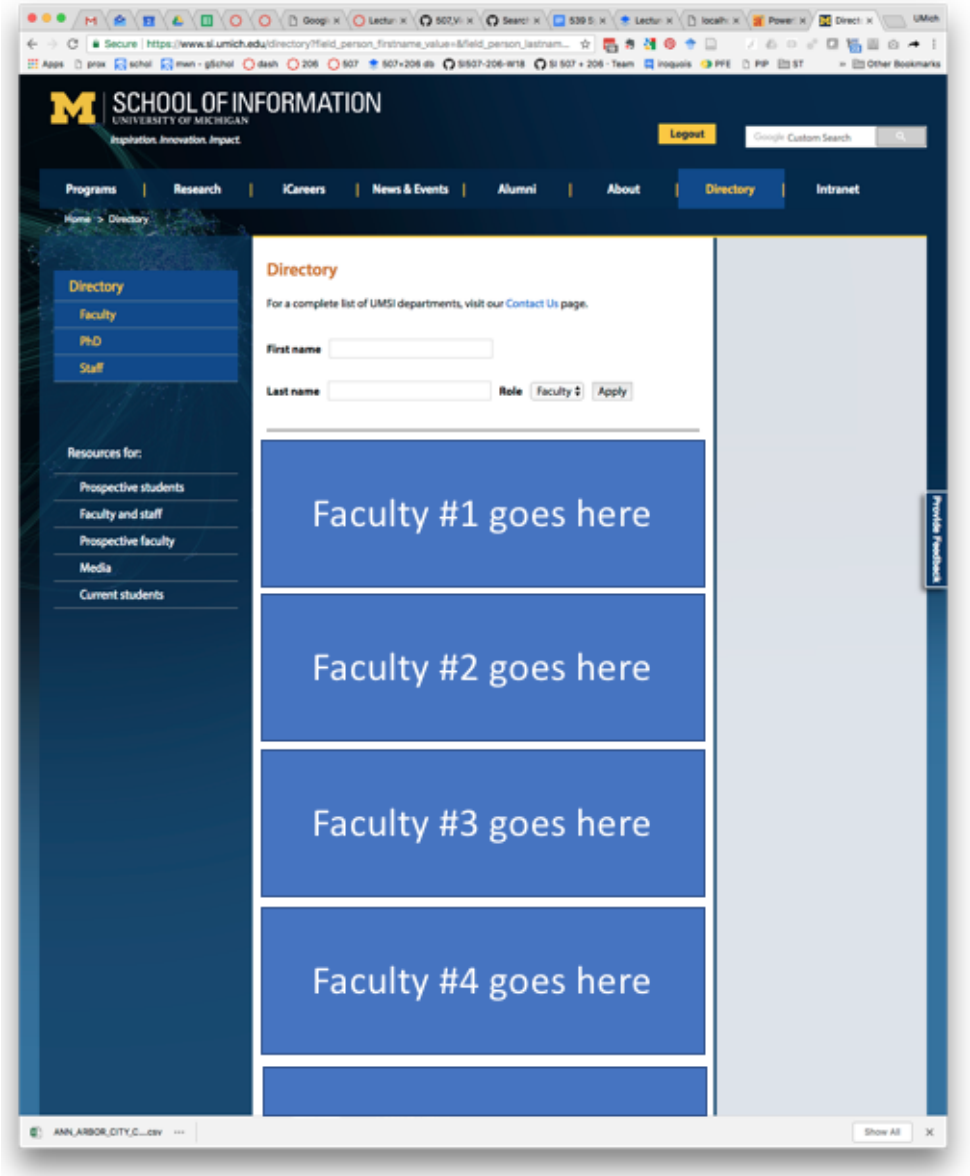
We're not going to get too deep into this. Just enough to get a taste.

A Template is basically a web page with some holes in it.

If this is what will get presented to the user,



the template might (logically) look something like this:



To define such templates, Flask uses a templating language called jinja2. The package for jinja2 was installed automatically when you installed Flask (you might have noticed it when you typed `pip freeze`.

Let's look at an example. We can start by creating a file called 'name.html' that contained this code:

```
<!-- name.html -->
<h1> Hello my name is {{name}} </h1>
```

Simple enough. So, how would we use it?

In our python file, we could add a piece of code like this:

```
@app.route('/name/<nm>')
```

```
def hello_name(nm):
    return render_template('name.html', name=nm)
```

Let's unpack this.

```
@app.route('/name/<nm>')
```

The route part you've seen before. The <nm> part is new. This is how you define a variable in a path. Whatever you put after `/name/` in your browser's address bar will be passed into the variable `nm`, which in turn will be passed as a parameter for the `hello_name` function:

```
def hello_name(nm):
```

(note: the variable names need to match)

The `hello_name` function then calls Flask's `render_template` function to produce the html we want to display for this path:

- the first parameter is the template name. This has to be the name of the file in the `templates` subdirectory. We'll come back to this.
- after the first parameter you can have as many *named parameters* as you want. These will be passed into the template where they can be used to fill in the holes.

So if you went to the URL http://localhost:5000/name/Sam the `render_template` would be called with the arguments `"name.html"`, `name=Sam`, and the resulting html that would be returned to display to the user would be

```
<!-- name.html -->
<h1> Hello Sam! Nice to see you! </h1>
```

Let's step through this all.

1. in your working directory where your app code is for this lecture, create a new directory called "templates"
2. create a new file there called `name.html`, with the contents:

```
<!-- name.html -->
<h1> Hello my name is {{name}} </h1>
```

3. Create your name.py file (*not* in templates—one level up) and put into it the following code. It looks pretty similar to what we did earlier today, doesn't it?

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/name/<nm>')
def hello_name(nm):
    return render_template('name.html', name=nm)

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

4. run it, and visit `localhost:5000/name/<yourname>`

Jinja2 Templates

- Jinja(2) is a templating language for python.
- Documentation is here: https://jinja.palletsprojects.com/en/2.10.x/
- It includes "python-like expressions" (e.g., `for` loops), so you can build complex output more easily
- It is the default templating language for Flask
- There are other templating languages like this (e.g., Django has a similar one), and templating languages can be mixed and matched with application frameworks

How to use:

- `{% … %}` syntax is used for control flow statements, which can be interwoven with HMTL to generate HTML output that will be shown to the user. For example, you could write this:
  - `{% if name == "Fred" %}`
    `<h1> Hi Fred </h1>`
    `{% endif %}`
  - `{% for i in range(10) %}`
    `<p>hey</p>`
    `{% endfor %}`
- `{{…}}` is used to print variables or expressions as part of page text.
  - `<h1>My name is {{name}}</h1>`

  or
  - `{% for item in my_list %}`
    `<p> {{item}} </p>`

`{% endfor %}`

Note that when you use a variable name as part of a control structure, you don't enclose it in {{}}. You do that only when it's part of the HTML that will be displayed in the browser.

Another example
Remember datamuse? Let's create a route that will take a word and show the rhymes.

Start with a template that will print any list. (this should be a new file in the `templates` directory.)

```
<!-- list.html -->
<h1>{{title}}!</h1>
<ul>
{% for item in my_list %}
    <li>{{item}}</li>
{% endfor %}
</ul>
```

Now, for a warm up, let's create a new file, rhymes.py, and define the default route:

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

@app.route('/')
def hello_world():
    numbers = ['one', 'two', 'three']
    return render_template('list.html', title="Numbers from 1 to 3", my_list=numbers)

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

```

Note, your import line will need to look like this, since render_template is another flask module:

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

And try it out. http://localhost:5000

Now let's add the code that will get the rhymes from datamuse and display them on a page the user goes to. To do this, add another route to your program:

```
@app.route('/rhyme/<word>')
def find_rhymes(word):
    base_url = 'https://api.datamuse.com/words'
    params = { 'rel_rhy': word }
    results = requests.get(base_url, params).json()
    rhy_words = []
    for r in results:
        rhy_words.append(r['word'])
    return render_template('list.html',
        title="Rhymes with " + word, my_list=rhy_words)
```

You'll need to install `requests` in your virtual environment, and import it at the top of your file (along with `Flask` and `render_template` modules from the `flask` package).

Take a minute to look through what's happening here.

- What parameters are getting passed to the template, and how are they being used?
- How is this "view function" getting the word to rhyme? How is it finding the matches?
- How is the view function preparing the data it needs to send to the template?

Static Files in Flask

So far we've looked at how to generate HTML from within Python, and how to use Jinja templates in conjunction with Python code to assemble HTML that can be displayed.

We've also seen one example of how Flask looks for particular kinds of files (templates) that it needs for certain operations.

Web apps generally have other resources that they need, and these are usually unchanging, or "static" resources. These include image files, CSS files, static HTML, and JavaScript files). By convention, Flask apps usually put these in a subdirectory called "static". In fact, the "static" route is a special, built-in route (like templates) that Flask understands by default. Let's look at how we would include an image. First, let's generate HTML in Python that includes an image. I'll be working with this image. You can download it here.

Save this image into a new directory within your project directory called "static".

Now let's add a route that will display our image:

```
@app.route('/m')
def m():
    return '<img src="/static/bigM.png" />'
```

Test this out. Note how we define the image location—starting from the "root" of the Flask server.

This *only* works because `/static` is a defined route in Flask. It won't work for files that are not in the static directory, and it won't work if you call your static directory something other than 'static'.

OK, How about adding an image to a template?

Add this to the top of `list.html`:

```
{% for i in range(5) %}
    <img src="/static/bigM.png" height=100 />
{% endfor %}
```

And try it out.

url_for

One problem with what we've just done is that it's a bit fragile. In particular, it depends on Flask being configured with this particular 'static' route. This is the default and it will work just fine for what we're doing, but it could break if you ever want to install your Flask app onto another server that may be configured differently (as will often be the case if it's an industrial strength server that hosts lots of different apps). To ensure that our app will work in different environments, we should use the `url_for` function built into Flask to generate the correct path to the resources we need.

First, we need to add this to our imports

```
from flask import Flask, render_template, url_for
```

And then we use it:

```
@app.route('/m')
def m():
    html = url_for('static', filename='bigM.png')
    return '<img src='+html+'/>'
```

The `url_for` function will return a string representing the path to the filename we want to reference, based on where the server is configured to look for the static resources.

We can do something similar within the template. In `list.html`:

```
{% for i in range(5) %}
```

```
    <img src="{{ url_for('static', filename='bigM.png') }}"
height=100 />
{% endfor %}
```

(Note how `{{ … }}` just prints the output of the url_for function the same way it prints the content of a variable. Basically `{{ … }}` evaluates the expression inside the braces and prints it out.

The reason all of this works is that Jinja templates are not actual html files. They get compiled down into python code at run time, so they are able to call functions (like url_for) like a regular python program. When that code is executed, it produces an html output (a big string containing all generated tags) that can be passed back to the browser for displaying to the user.

Finally, we can also do this for stylesheets. Let's create a file called `style.css` in your static folder and add some styles:

```
/* style.css */
body {
    background-color: lightblue;
}

h1 {
    color: brown;
    font-family: fantasy;
    font-size: 44;
}

li {
    color: yellow;
    font-family: monospace;
}
```

Now, let's add a line to our template file (list.html) to link to the stylesheet:

```
<link rel="stylesheet" type="text/css"
  href="{{ url_for('static', filename='style.css') }}"/>
```

Try going to a rhyme page for a new word. It should be all styled up.

Lecture Exercise

1. Change the `rhyme` route so that it only prints out the first 10 rhyming words.
2. Add another route to your Flask app: `/similar/<word>`. Using the datamuse API, have your app print out a list of up to 10 words that are *similar in meaning* to the chosen word.
3. Add a route `/wordinfo/<word>` that displays a page that lets the user choose between two links: one that will show rhymes and one that will show similar words.
4. Add some images and style. Make it zing!