Day 6

This is CS50 for MBAs. Harvard Busines School. Spring 2015.

Cheng Gong

Table of Contents

Questions

- Why would it be accurate to describe Python as a programming language (whereas HTML is not a programming language)?

  # Python has logical control over data, so we can do things to data that we can't in HTML.

- What's the difference between a compiled language and an interpreted language? Which is Python?

  # A compiled language has a compiler which turns our source code into object code that is executed directly on the processor. An interpreted language needs an interpreter, which is running on the processor, that reads in our source code and converts it to instructions as it is being run. A compiled program only needs to be compiled once into binary, and then you can run it over and over again, but an interpreted program needs to be interpreted constantly whenever it is running.

- How can we think of a language as being compiled or interpreted if the designation of compiled or interpreted depends on how the language is translated into machine code, not inherent in the language itself?

  # The creators of the language decided what the language will be, since they wrote an interpreter or compiler for that language. It's true that a compiler can be written for an interpreted language, so these designations might not be distinct, but there is a default that we would assume of how programs written in each language will be run.

- Will NLP (natural language processing) get so good that interpretive language will eventually be just a matter of writing good prose?
  - # Well all of this syntax helps make our commands specific and precise, and context too helps our conversations make sense, so it's uncertain how good computers might get in terms of understanding humans.
- When you define functions in python, does the order in which you use print and return matter? Seems like you do have to use both of them all the time. More broadly, why did the designers of python use this seemingly repetitive syntax?
  - # `print` prints something to the screen, and `return` gives a value from one function to another. For example, function-0.py[1]:

```
"""Prints a user's name. Demonstrates a function (not from a library)
 with a side effect."""

def printName(name):
    print("hello, {}".format(name))

s = raw_input("Your name: ")
printName(s)
```

    - # So it seems like we've defined a function called `printName` that's called (used) when we want later.
    - # Also, there are conventions for function names. In this example we use CamelCase[2] where the first letter is lowercase, but the first letters of following words are capitalized. Another way is to use underscores between words, as in `print_name`.
    - # Inside `def printName()` is a variable called `name`, and this is like an input within a block inside Scratch, where we're passing in an argument (input) to the function.
    - # There is another syntax, with `%s` and `% name`, but the latest version of Python uses the `.format` function.
- What are the most popular programming languages that tech companies use today for their websites? Have the most popular languages changed in the past ten years?

[1] http://cdn.cs50.net/2015/mba/classes/6/src6/cli/function-0.py
[2] http://en.wikipedia.org/wiki/CamelCase

And what are the factors people think about when deciding on a given language (e.g., choosing Ruby vs. Python)?

# A fun website, langpop.com[3], shows one method of comparison, by looking around various places on the internet. C is super popular, with other ones you might have heard of following, like PHP, Java, Python, or Ruby. And people choose languages based on their goal. iPhone apps need to be written in Objective-C or Swift, and other problems have similar constraints.

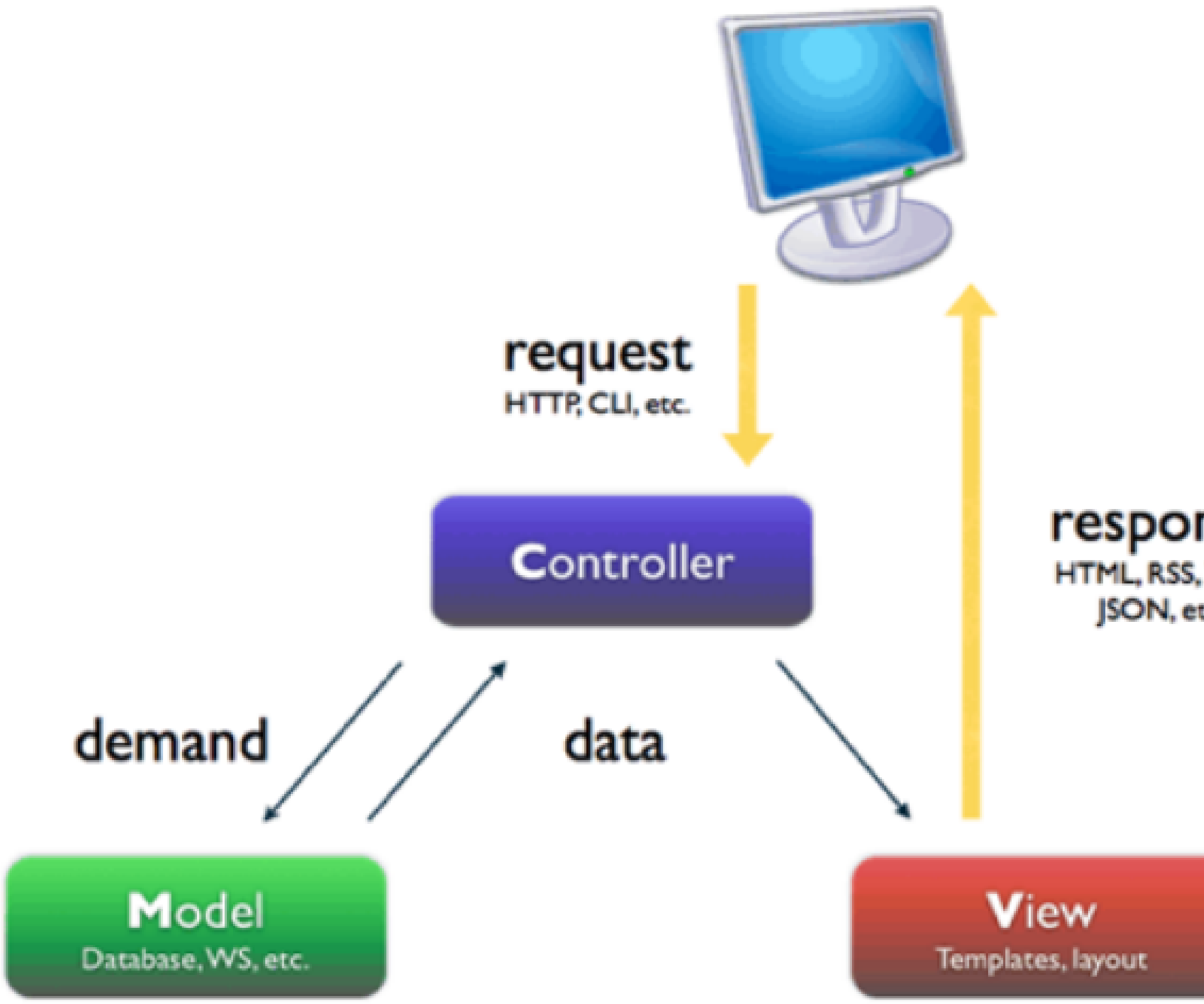
# To evaluate how good someone is, we'd want to see how they go about solving a particular problem and not just how they implement something in a particular language, though we would want to see how modern their skills are.

# To find out what the back-end is running, we might use Chrome's Developer Tools to look at the response headers. Sometimes the server will tell you, but other times it might not, since older versions might have security bugs, and telling that to the world might make a server a more likely target.

MVC

- Let's look at this picture, the theme of today:

[3] http://langpop.com/



- MVC stands for Model-view-controller, and it's just a way of organizing files for web applications so that various functionality is separated logically.
- The controller is like the brains that interact with all the components by looking at the incoming requests and decides what to send back to the user, the view is like the front-end that determine the aesthetics and presentation of information, and the model is where we store our data, which we'll talk more about next time.

- What we want to do is parse an HTTP request and return an HTML page, but we don't want to reinvent the wheel, so instead we'll use a framwork called Flask[4] that someone else has written, which just means that there are lots of files and functions that we can just use.
  - # A framework is like a wooden frame of a house, that has some structure or main components pre-built, but leaves flexibility for us to add the features we need in the ways we want.
- We'll also use a library, which is just another collection of code we can use in our own programs, called Flask-Mail[5].
  - # And to figure out which framework or libraries we might want to use, we just Google around and read articles or comments to see what fits our purposes best.

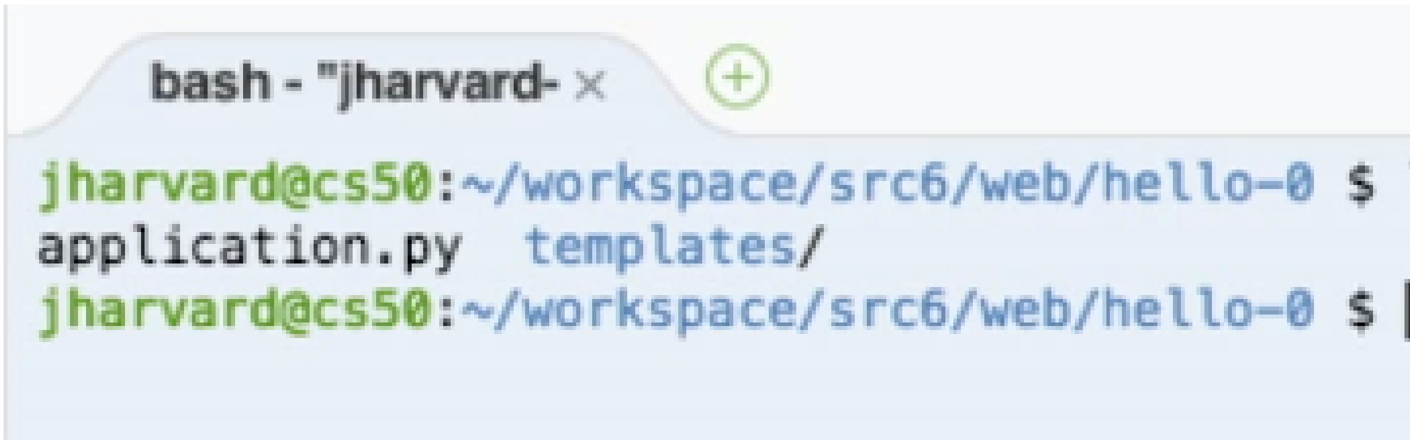
Python for the Web

- So let's jump in to our `hello-0` [6] source directory:

```
> templates
  application.py
```

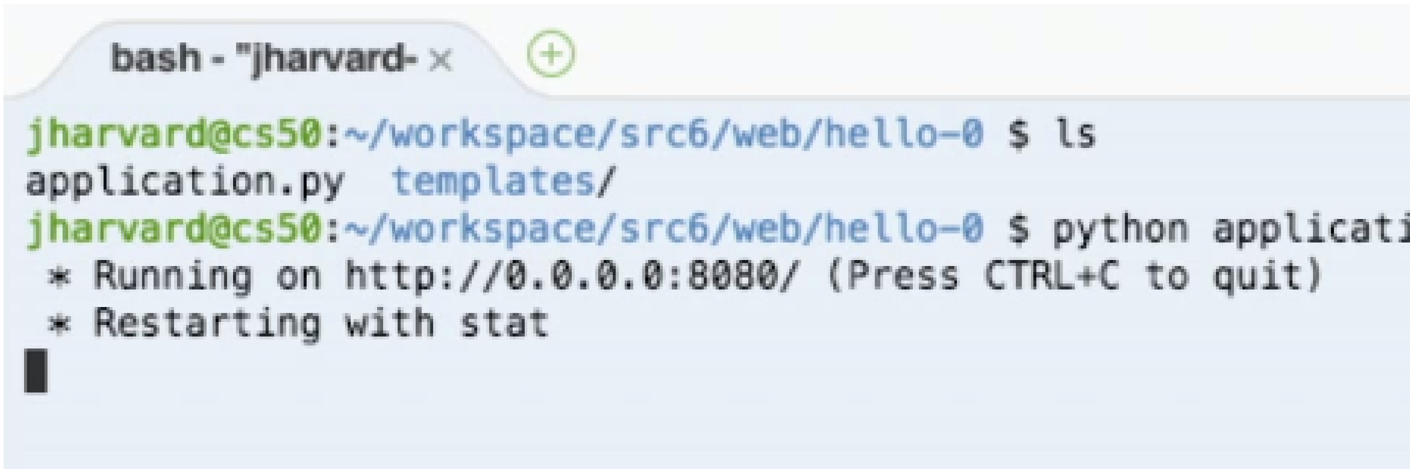
- We have these files, and we created them because we read the Flask documentation that told us to create a file called `application.py` (the controller) and a folder called `templates`, inside which our HTML files (the views) will live.
- We'll use the command line in our Cloud9 workspace, and first we'll type `ls`, which tells us what's in the folder we're currently looking at:

[4] http://flask.pocoo.org/
[5] https://pythonhosted.org/Flask-Mail/
[6] http://cdn.cs50.net/2015/mba/classes/6/src6/web/hello-0/



```
jharvard@cs50:~/workspace/src6/web/hello-0 $
application.py  templates/
jharvard@cs50:~/workspace/src6/web/hello-0 $
```

- Now we'll run `python application.py`:

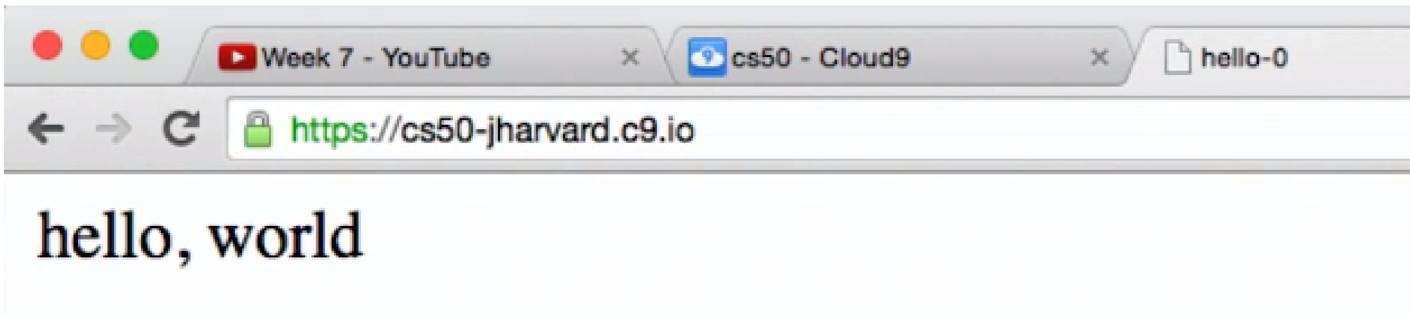


```
jharvard@cs50:~/workspace/src6/web/hello-0 $ ls
application.py  templates/
jharvard@cs50:~/workspace/src6/web/hello-0 $ python applicati
 * Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
 * Restarting with stat
```

# We started the Python interpreter, and gave it `application.py` as a source file.

# There's also a server running now, on port `8080`, at the IP `0.0.0.0`, which is a cryptic way of saying our server will accept any requests it gets.

- And now if we go to our server's URL, we'll see this:



hello, world

- We can do this easily in HTML, but behind-the-scenes we have code that generates this dynamically.
- We'll open `application.py`[7] to look inside:

```
"""Says hello.

Demonstrates templates.

David J. Malan
malan@harvard.edu
"""

# import Flask, plus support for rendering templates
from flask import Flask, render_template

# create an instance of Flask
app = Flask(__name__)

@app.route("/")
def index():
    """says hello"""
    return render_template("index.html")

# if the script is executed directly from the Python interpreter and not
 used as an imported module
if __name__ == "__main__":
    app.run(debug=True, host="0.0.0.0", port=8080)
```

  # So we have some comment for ourself about what this program does, and then we have another one that tells us we're going to "import Flask". This is just telling our program to use the framework Flask by means of line 10, and then in line 13 we create a variable called `app` that holds our application.

  # Then in line 15, we're telling our web server that our default page, `/`, in our URL, will be the function that follows, `index()`.

  # The function `index()`, in turn, will return the file `index.html` by calling a function `render_template` (all that function does is open the file and get us the contents). Notice that no HTML is actually in this file, but separated.

[7] http://cdn.cs50.net/2015/mba/classes/6/src6/web/hello-0/application.py

  - The `@` syntax in line 15 takes the following lines of code (the following function, to be precise) and acts like a label that we can find whenever it's needed.
  - And the last two lines are fairly standard, just running a server with configuration options that let us run it in different settings.

- We got Flask by running the `sudo pip install flask` command, which runs a program called `pip`, which installs Python packages, or bundles of files, that we can refer to when we run Python in the future.

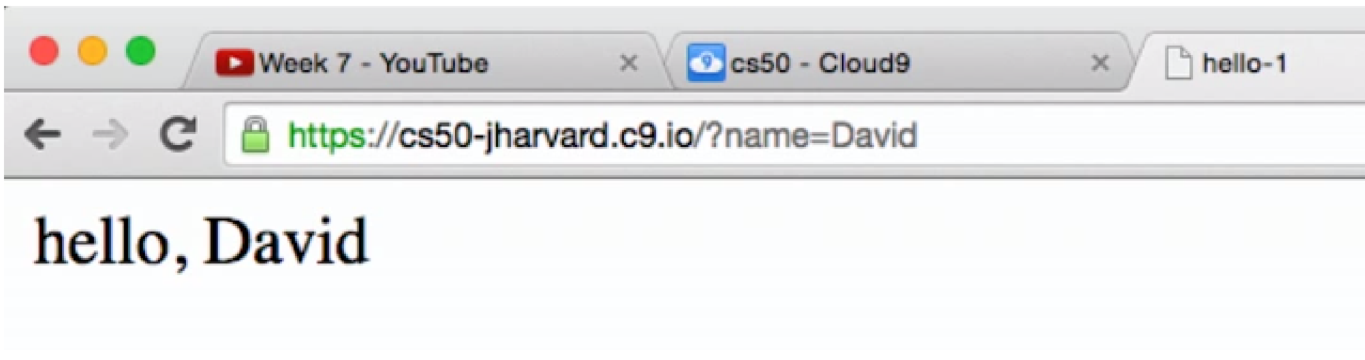
- So let's open `application.py`[8] from `hello-1` to see what else we can do:

```
"""Says hello.

Demonstrates templates with parameterization.

David J. Malan
malan@harvard.edu
"""

# import Flask, plus support for rendering templates and for accessing
 requests' parameters
from flask import Flask, render_template, request

# create an instance of Flask
app = Flask(__name__)

@app.route("/")
def index():
    """says hello"""
    return render_template("index.html", name=request.args.get("name"))

# if the script is executed directly from the Python interpreter and not
 used as an imported module
if __name__ == "__main__":
    app.run(debug=True, host="0.0.0.0", port=8080)
```

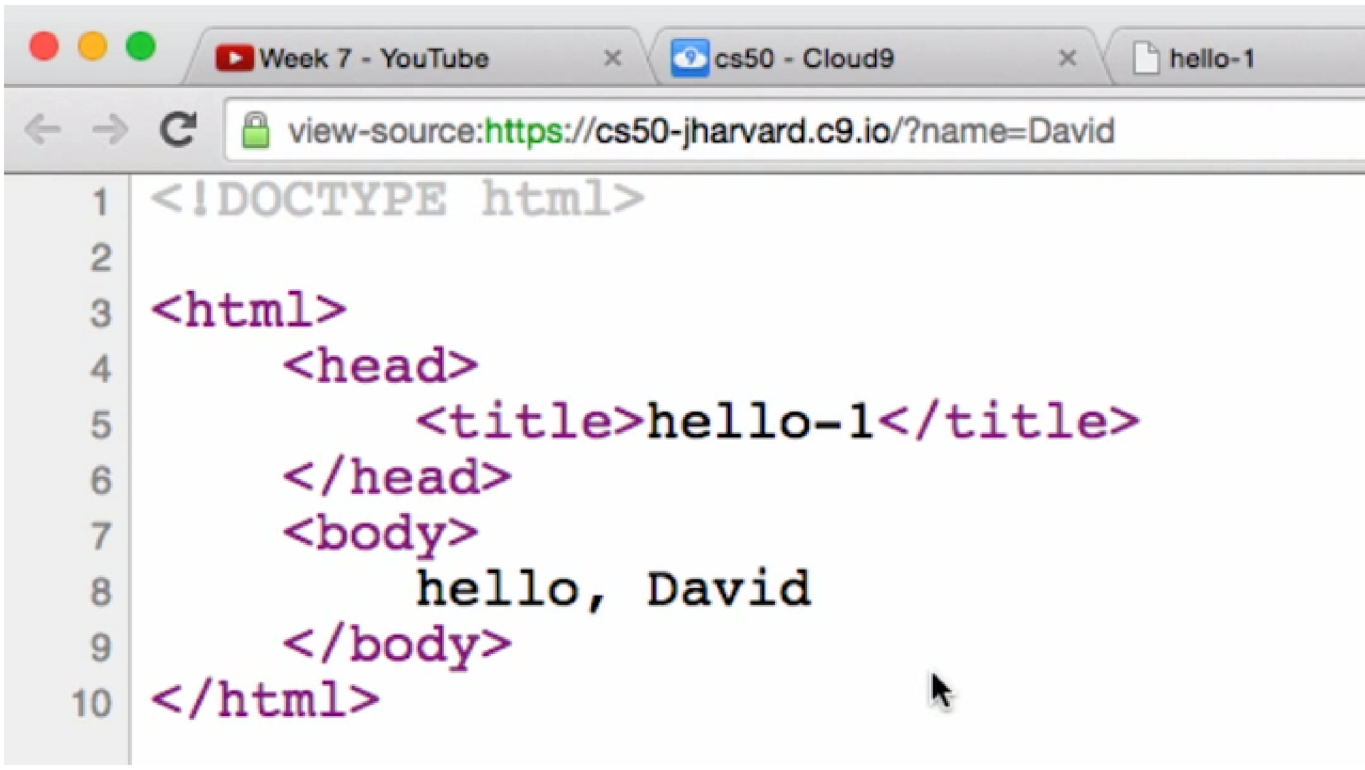
  - This all looks the same, except we've imported another component, `request`, which allows us to take HTML parameters as input.

[8] http://cdn.cs50.net/2015/mba/classes/6/src6/web/hello-1/application.py

# We've also changed `render_template` to take in another argument, `name`, which we've set to `request.args.get("name")`, which sounds like it's getting the `name` input from the request:



hello, David



```
<!DOCTYPE html>

<html>
    <head>
        <title>hello-1</title>
    </head>
    <body>
        hello, David
    </body>
</html>
```

# We can change the URL we send to anything, and the source will be updated seemingly magically.

# By reading the documentation for Flask, we know to change our HTML source file to include placeholders (which would otherwise be invalid HTML) for values that we pass in, so it can generate the appropriate valid HTML:

```
<!DOCTYPE html>

<html>
    <head>
        <title>hello-1</title>
    </head>
    <body>
        hello, {{ name }}
    </body>
</html>
```

- The `{{ name }}` is our placeholder, and we could have called it anything, as long as it was consistent with our `application.py`.
- These pages also aren't stored anywhere once they're generated, which means we have to do this processing over and over again. But a technique called caching can be enabled, where the most frequently generated pages are saved, so we can save some time.
- Now let's look at `application.py`[9] from `hello-2`:

[9] http://cdn.cs50.net/2015/mba/classes/6/src6/web/hello-2/application.py

```
"""Says hello.

Demonstrates variable rules.

David J. Malan
malan@harvard.edu
"""

# import Flask, plus support for rendering templates and for accessing
 requests' parameters
from flask import Flask, render_template, request

# create an instance of Flask
app = Flask(__name__)

@app.route("/<name>")
def index(name):
    """says hello"""
    return render_template("index.html", name=name)

# if the script is executed directly from the Python interpreter and not
 used as an imported module
if __name__ == "__main__":
    app.run(debug=True, host="0.0.0.0", port=8080)
```

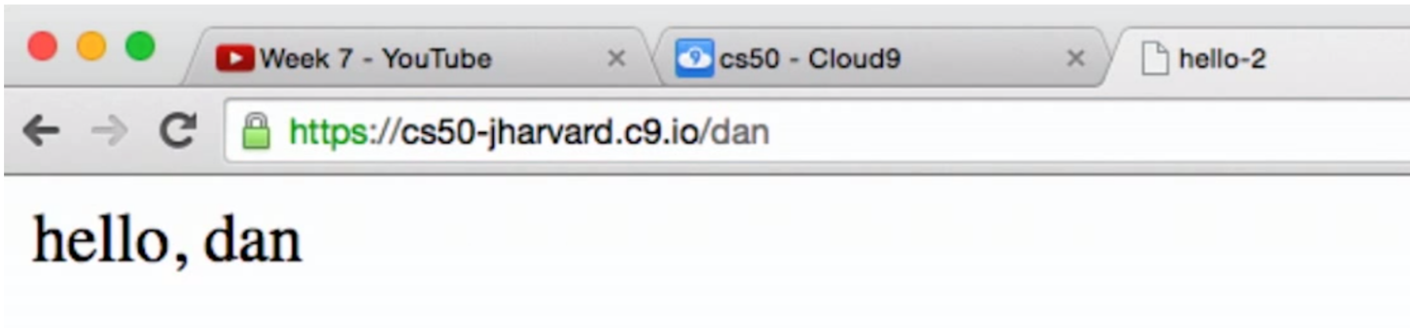
# Not too much is different, except we've changed line 15, 16, and 18.

# In line 15, our route is just `/<name>`, which is just saying that we'll take whatever is after the `/` in the URL and save it as a variable called `name`.

# In line 16, we take that variable and give it to our `index` function.

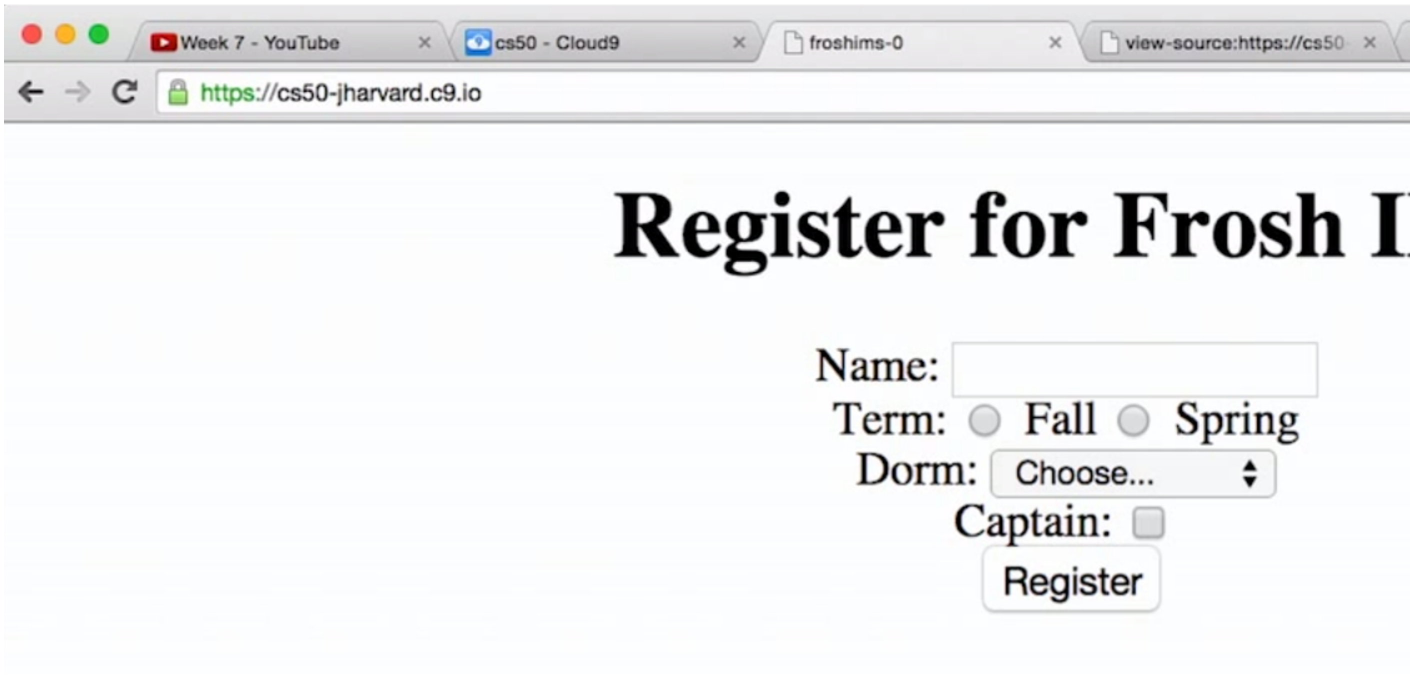
# In line 18, we return the same page as before, but with the `name` passed in to the function (which was originally from the URL) as the `name` variable as input to the template.

- Now we can just visit a URL like this:



hello, dan

- Back in the day, there was this freshman intramural sports program, which didn't have a website, and long story short David implemented an online registration form for them.
- `froshims-0`[10] looks like this:

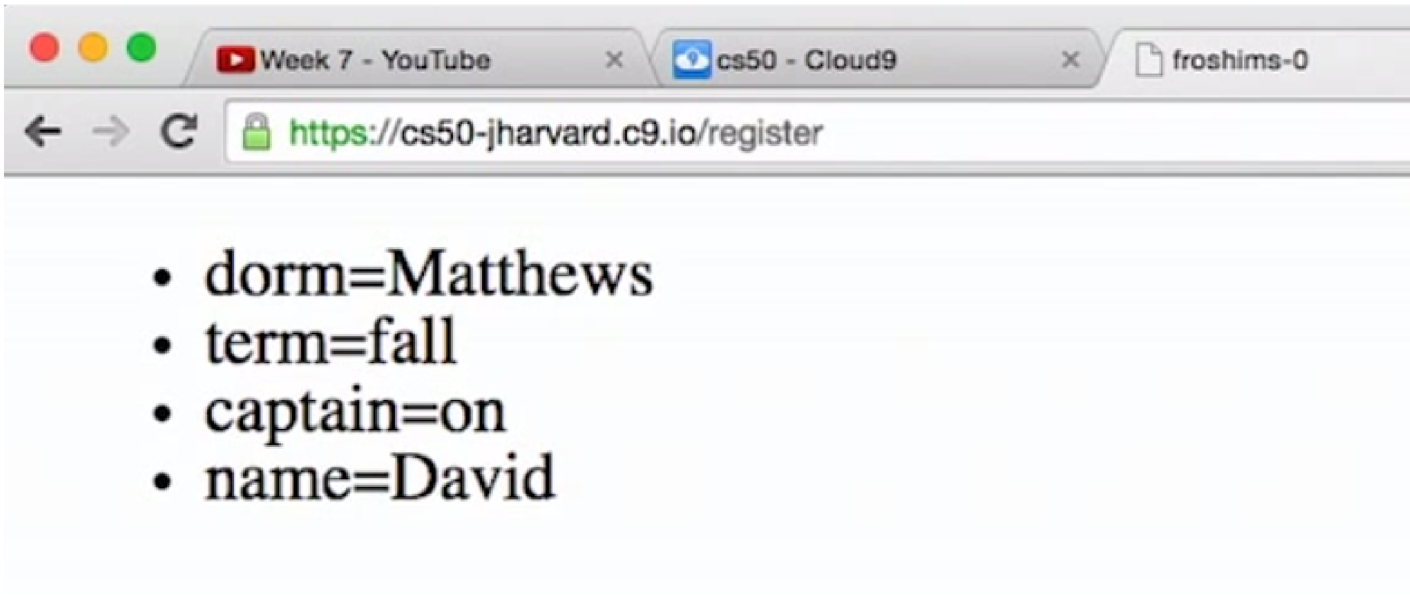


Register for Frosh I

Name:
Term: Fall Spring
Dorm: Choose...
Captain:

- When we submit the form, all we get back is this:

[10] http://cdn.cs50.net/2015/mba/classes/6/src6/web/froshims-0/



- dorm=Matthews
- term=fall
- captain=on
- name=David

- It doesn't seem to be doing anything useful, but does demonstrate how we might get input from a user. `application.py`[11] looks like:

[11] http://cdn.cs50.net/2015/mba/classes/6/src6/web/froshims-0/application.py

```
""" """

# import Flask, plus support for rendering templates and for accessing
 requests' parameters
from flask import Flask, render_template, request

# create an instance of Flask
app = Flask(__name__)

@app.route("/")
def index():
    """displays registration form"""
    return render_template("index.html")

@app.route("/register", methods=["POST"])
def register():
    """displays values of any parameters submitted via POST"""
    return render_template("register.html", form=request.form)

# if the script is executed directly from the Python interpreter and not
 used as an imported module
if __name__ == "__main__":
    app.run(debug=True, host="0.0.0.0", port=8080)
```

- It's not much more complicated, apart from the second route in line 14 on, where we have a new route called `/register` that seems to accept POST instead of GET as its method.
- The `register` function is rendering the template `register.html`, but passing in the entire form as a variable this time (which the `request` library created for us from the request). So we need to go deeper and look at `register.html`[12]:

[12] http://cdn.cs50.net/2015/mba/classes/6/src6/web/froshims-0/templates/register.html

```
<!DOCTYPE html>

<html>
    <head>
        <title>froshims-0</title>
    </head>
    <body>
        <ul>
            {% for key, value in form.items() %}
                <li>{{ key }}={{ value }}</li>
            {% endfor %}
        </ul>
    </body>
</html>
```

# Flask has its own template language, which is simple and not quite Python, and again we just Googled around and looked at documentation to figure out the syntax.

# In this example, it seems like we're accessing the `form`'s fields with the `items` function of the variable, and for each of those fields there's something called a `key` and a `value`, where a `key` is the name of the field, like `Dorm`, and the `value` is what's inside, like `Matthews`.

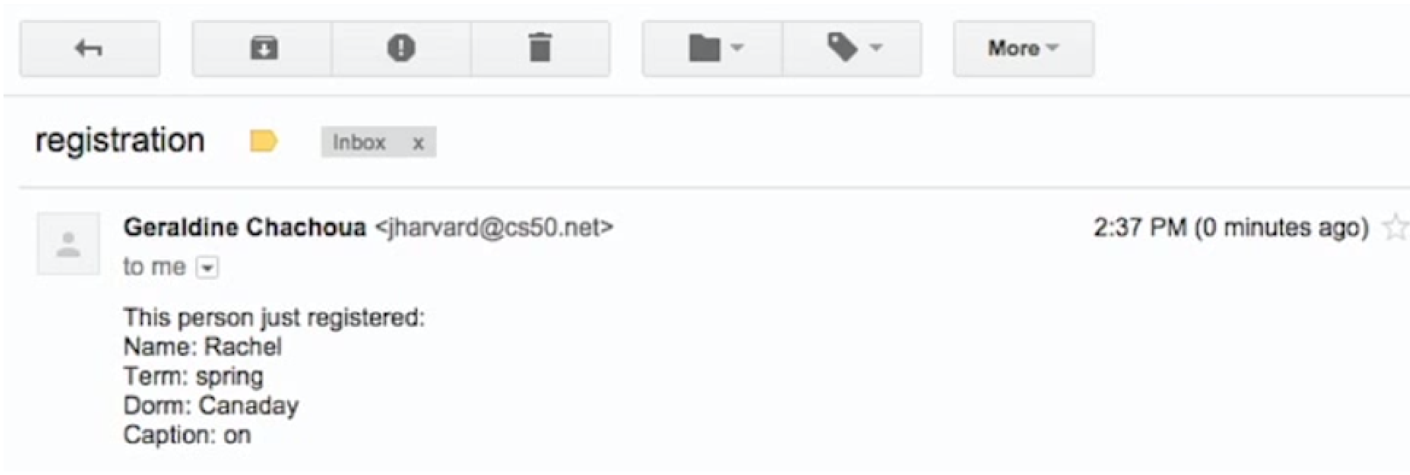
- Let's fast-forward to `froshims-3`[13], with lots of code, but it looks like we have some email functionality now:

[13] http://cdn.cs50.net/2015/mba/classes/6/src6/web/froshims-3/application.py

```
"""Implements a registration form for Frosh IMs; reports registration via
 email.

David J. Malan
malan@harvard.edu
"""

# import Flask, plus support for redirects, rendering templates, accessing
 requests' parameters, and generating URLs
from flask import Flask, redirect, render_template, request, url_for

# import Flask-Mail for sending email
from flask_mail import Mail, Message

# create an instance of Flask
app = Flask(__name__)

# configure Flask-Mail
app.config["MAIL_DEFAULT_SENDER"] = ("TODO", "TODO")
app.config["MAIL_PASSWORD"] = "TODO"
app.config["MAIL_PORT"] = 465
app.config["MAIL_SERVER"] = "smtp.gmail.com"
app.config["MAIL_USE_SSL"] = True
app.config["MAIL_USERNAME"] = "TODO"
mail = Mail(app)

@app.route("/")
def index():
    """displays registration form"""
    return render_template("index.html")

@app.route("/register", methods=["POST"])
def register():
    """sends email upon each registration"""

    # ensure user provides name, term, and dorm when registering
    if not (request.form.get("name") and request.form.get("term") and
 request.form.get("dorm")):
        return redirect(url_for("index"))

    # prepare body of email
    body = """This person just registered:
Name: {}
Term: {}
```

```
Dorm: {}
Caption: {}
""".format(request.form.get("name"), request.form.get("term"),
```

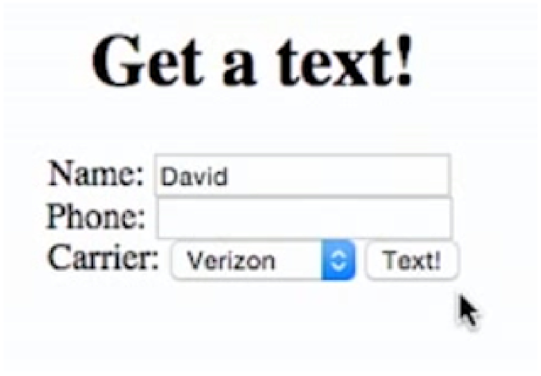
- Now if we fill out the form and submit it, we'll get an email from `jharvard@cs50.net`, but the name that we set:



Geraldine Chachoua <jharvard@cs50.net>
to me
2:37 PM (0 minutes ago)

This person just registered:
Name: Rachel
Term: spring
Dorm: Canaday
Caption: on

- So we see how we might set the sender's email address to something that isn't us, and it would depend on the receiving server to trust whether the email is legitimate or not.
- We can also use a Python program to send texts, since most carriers have a feature where emails sent to a particular address will be sent as a text to the corresponding cell phone number.
- We have another demo where we fill out a form that looks like this, and it sends us a text:



Get a text!

Name:	David
Phone:	
Carrier:	Verizon