Table of Contents

Week 8

Intro to Python

- For many problems, C might not be the best language, because we need to implement many data structures and manage memory ourselves.
- Higher-level languages add features and abstractions to help programmers.
- Just as transitioning from Scratch to C led us to see many of the same concepts, so will our transition from C to Python.
- The first "hello, world" program we wrote in C could be written in Python with just:

```
print("hello, world")
```

- We can add a `main` function, even though it's not required in Python as it is in C:

```
def main():
    print("hello, world")

if __name__ == "__main__":
    main()
```

  - Notice that here we can define a function in Python with a line like `def main():`, without any curly braces, but indentation instead to indicate the hierarchy.
- `printf` in C is just `print` in Python, and we don't need to include a `\n` at the end, since it's automatically included for us.
- A loop to do something forever in Python looks like this:

```
while True:
    print("hello, world")
```

- The boolean `True` has to be capitalized.

- A loop to do something a certain number of times could be achieved with this:

```
for i in range(50):
    print("hello, world")
```

- The variable `i` is our placeholder again, but we don't need to declare it, and `range(50)` automatically creates a range of numbers (from `0` to `49` by default if we ask it for `50` numbers).

- In Python, too, we have types of variables but it is not a strongly-typed language like C. Instead it is loosely-typed, so we don't need to specify variable types each time.

- Instead, we can just write:

```
i = 0
```

- Boolean expressions have the same syntax: `i < 50`, `x < y`.

- Conditions are similar too:

```
if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
else:
    print("x is equal to y")
```

- Notice that we don't have parentheses around our expressions, anymore, and that instead of using curly braces, indentation is used to indicate which lines of code belong in which block.

- And `elif` is the Python keyword that means `else if`.

- Arrays in Python are called lists, and the language manages the memory for you.

- The syntax for lists are the same as arrays in C, where if we used `argv[0]` in C to get a command-line argument, we can use `sys.argv[0]` in Python.

- We'll also notice that while C was compiled, from source code to machine code, but Python is interpreted. This means that we can run our program with one command, which will then take care of everything that needs to be done to run the program.

- For example, we run a program we wrote with a command like `python hello.py`. This starts a program called `python`, which is then passed an argument, `hello.py`, that contains our program's source code.
- The `python` program is an interpreter that compiles our source code into something called bytecode that looks like this, and then runs it from top to bottom:

```
 2           0 LOAD_GLOBAL              0 (print)
             3 LOAD_CONST               1 ('hello,    world')
             6 CALL_FUNCTION            1 (1  positional, 0   keyword pair)
             9 POP_TOP
            10 LOAD_CONST               0 (None)
            13 RETURN_VALUE
```

- So Python is a language, but also a program that can compile and interpret that language. (Whereas C is a language, and `clang` is a compiler.)

First Programs

- We've also implemented some training wheels again, with functions in a library that we'll call like this:
  - `cs50.get_char`
  - `cs50.get_float`
  - `cs50.get_int`
  - `cs50.get_string`
  - …
- These functions are part of the `cs50` module, so we need to indicate that in Python.
- Python has familiar data types and features:
  - `bool`
  - `float`
  - `int`
  - `str` (a string, with functionality built-in to manage them easily)
  - `complex` (complex or imaginary numbers)
  - `list` (like arrays)
  - `tuple` (groups of values, like x, y coordinates)

- `range`
- `set` (collections of objects, like in math, with certain properties)
- `dict` (a dictionary, like a hash table)
- `...`

- So let's save a file, `hello.py`, with the following contents:

```
print("hello, world")
```

- Then, we can run `python hello.py` and see this:

```
$ python hello.py
hello, world
```

- We can translate this:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string name = get_string();
    printf("hello, %s\n", name);
}
```

- to this:

```
import cs50

s = cs50.get_string()
print("hello, {}".format(s))
```

  - The syntax for including a library is to use `import`.
  - Then we declare a variable called `s`, and not need to specify the type, and we call `cs50.get_string()` and store the return result into `s`.
  - Then we include `s` in what we print. Strings, or more generally objects, have built-in functions. We can call those functions with the syntax shown, like `"hello, {}".format(s)`, and by passing in the correct arguments, we can substitute variables the way we want.

- There are two main versions of Python, 2 and 3, which have enough differences such that programs written in one language will likely not work in the other.

- We'll use Python 3, but there might be lots of documentation or tutorials online that still use Python 2.
- Python also has an `input` function, which we can use instead of the CS50 library:

```
s = input("name: ")
print("hello, {}".format(s))
```

  - We can pass in a prompt inside that function, and get the typed value back at the same time.
- Similarly, we can get a number:

```
import cs50

i = cs50.get_int()
print("hello, {}".format(i))
```

- We can print floating-point numbers with enough decimal places to see imprecision in Python, too:

```
print("{:.55f}".format(1 / 10))
```

  - The value we want to print is `1 / 10`, and to specify the format we place `:55f` inside the curly braces of the string.
- And if we run that, we see:

```
`0.1000000000000000055511151231257827021181583404541015625`
```

- In C, if we divided an `int` by another `int`, we get back another `int`. But Python automatically returns a floating-point value if one is needed.
- We can write a familiar program that uses various operators:

```
import cs50

# prompt user for x
print("x is ", end="")
x = cs50.get_int()

# prompt user for y
print("y is ", end="")
y = cs50.get_int()

# perform calculations for user
print("{} plus {} is {}".format(x, y, x + y))
print("{} minus {} is {}".format(x, y, x - y))
print("{} times {} is {}".format(x, y, x * y))
print("{} divided by {} is {}".format(x, y, x / y))
print("{} divided by {} (and floored) is {}".format(x, y, x // y))
print("remainder of {} divided by {} is {}".format(x, y, x % y))
```

- There is a special operator in Python, `//`, that divides two integers and returns an integer that's truncated (with everything after the decimal point removed).
- And comments in Python, instead of starting with `//`, will start with `#`.
- And we pass in `end=""` as an additional argument to `print` if we don't want a new line to be added for us automatically at the end.

- We can write a program to convert temperature:

```
import cs50

f = cs50.get_float()
c = 5.0 / 9.0 * (f - 32.0)
print("{:.1f}".format(c))
```

- We first get a float, `f`, apply the correct formula and save the result to `c`, and we want to format it to one decimal place so we use `:1f`.

Logical Programs

- We can add logic, too:

```
import cs50

c = cs50.get_char()
if c == "Y" or c == "y":
    print("yes")
elif c == "N" or c == "n":
    print("no")
else:
    print("error")
```

- We get a `char`, and compare it to `Y` or `y` or `N` or `n` to tell us if we said yes or no.
- We just say `or` and `and` in Python instead of `||` and `&&`.
- And in C, we needed to compare `char`s by using single quotes, but in Python single characters are also strings. The good news is, we can compare strings with a simple `==` and it will compare them the way we might expect, equalling `True` if the strings have the same contents. Even more mind-blowingly, in Python single quotes `'` and double quotes `"` can both be used to indicate strings, as long as we use the same one on both sides of the string.

- In C, we also once implemented a program to get a positive integer:

```
#include <cs50.h>
#include <stdio.h>

int get_positive_int();

int main(void)
{
    int i = get_positive_int();
    printf("%i is a positive integer\n", i);
}

int get_positive_int(void)
{
    int n;
    do
    {
        printf("n is ");
        n = get_int();
    }
    while (n < 1);
    return n;
}
```

- We needed to first delare the function, then a variable `n`, and then a `do while` loop.

- Now we can write:

```
import cs50

def main():
    i = get_positive_int()
    print("{} is a positive integer".format(i))

def get_positive_int():
    while True:
        print("n is ", end="")
        n = cs50.get_int()
        if n > 0:
            break
    return n

if __name__ == "__main__":
    main()
```

  - We don't need to declare `get_positive_int` before we call it, as long as it doesn't actually need to be run before we get to the part of the code that defines it. In this case, we call `get_positive_int` in `main`, but `main` itself isn't called until the very last line, so everything in our program should already be defined.
  - And we don't need to specify that `get_positive_int` takes no arguments, so we can just add a `()` instead of `(void)`.
  - Python also doesn't have a `do while` loop, so instead we use `while True`, but `break`, or stop the loop, `if n > 0`.
  - Then it returns `n`, but notice that we also didn't need to declare it outside the loop before we used it. `n` will be created the first time our loop runs, and then have the new value stored inside it every time after.
  - And finally, we need to call the `main` function with the last two lines.

- We could reimplement `cough`, to "cough" 3 times:

```
print("cough")
print("cough")
print("cough")
```

- To use a loop, we can:

```
for i in range(3):
    print("cough")
```

- And we can create a function:

```
def main():
    for i in range(3):
        cough()

def cough():
    print("cough")

if __name__ == "__main__":
    main()
```

- We can add an argument to our `cough` function:

```
def main():
    cough(3)

def cough(n):
    for i in range(n):
        print("cough")

if __name__ == "__main__":
    main()
```

  - Here `cough` takes in some argument `n`, which the language sets to an `int` automatically for us.

- And we can add multiple arguments to a function:

```
def main():
    cough(3)
    sneeze(3)

def cough(n):
    say("cough", n)

def sneeze(n):
    say("achoo", n)

def say(word, n):
    for i in range(n):
        print(word)

if __name__ == "__main__":
    main()
```

- Since we're only printing the `word` variable that's passed into our `say` function, we can just say `print(word)`.

More Complex Programs

- In Week 2, we implemented `strlen` ourselves:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string s = get_string();
    int n = 0;
    while (s[n] != '\0')
    {
        n++;
    }
    printf("%i\n", n);
}
```

- In Python, these implementation details are less and less visible, so we'll need to use documentation more frequently and rely more on built-in functions that are already written for us:

```
import cs50

s = cs50.get_string()
print(len(s))
```

- Let's see if we can convert characters to ASCII:

```
for i in range(65, 65 + 26):
    print("{} is {}".format(chr(i), i))
```

  - We can specify the starting number and the ending number in a range (including the starting number but not the ending number).
  - Then we print `chr(i)` first, and then `i`, using the `chr()` function in Python to convert an integer into a `char`.

- We can use command-line arguments too:

```
import sys

if len(sys.argv) == 2:
    print("hello, {}".format(sys.argv[1]))
else:
    print("hello, world")
```

  - We can check the length of the arguments with `len(sys.argv)`, and access the second one (recall that the first is the program's own name) with `sys.argv[1]`. Here `sys` is a module built into Python that has command-line arguments and others.

- We can print all of the arguments we get:

```
import sys

for i in range(len(sys.argv)):
    print(sys.argv[i])
```

- And we can print each character in each argument:

```
import sys

for s in sys.argv:
    for c in s:
        print(c)
    print()
```

- With `for s in sys.argv`, we are accessing element in `sys.argv`, and calling it `s`. And the type of each element will be a string.
- Then with `for c in s`, we are accessing each element in the string `s`, which we will call `c`, since each element is a character.

- We can also `exit` with some value, much like `return`ing some exit code in C:

```
import cs50
import sys

if len(sys.argv) != 2:
    print("missing command-line argument")
    exit(1)

print("hello, {}".format(sys.argv[1]))
exit(0)
```

- In Python, to end a program, since there might not always be a `main` function to `return` from, we call `exit` with some value.
- And recall that in the command line, we can type `echo $?` to see the return value of the last program that ran.

- We can compare two strings:

```
import cs50
import sys

print("s: ", end="")
s = cs50.get_string()

print("t: ", end="")
t = cs50.get_string()

if s != None and t != None:
    if s == t:
        print("same")
    else:
        print("different")
```

- Instead of `null`, since we don't need to worry about pointers as much anymore, there is a special value that `get_string` might return, `None`, that indicates there is nothing returned.
- In C, `s` and `t` would be two addresses that would not be the same, but in Python the contents of `s` and `t` would be compared automatically for us.

- To copy a string, we can do this:

```
import cs50
import sys

print("s: ", end="")
s = cs50.get_string()

if s == None:
    exit(1)

t = s.capitalize()

print("s: {}".format(s))
print("t: {}".format(t))

exit(0)
```

- Now we can run the program and see that `t` has a capitalized version of `s`, while `s` itself is unchanged.
- Recall that `s` is an object in Python, so it has built-in functions that we can call from that object with the `.` syntax, so we can use `s.capitalize()` that automatically takes the first character and capitalizes it.

- Furthermore, strings in Python are immutable, meaning that they can't be changed after they have been created. So `s.capitalize()` returns a copy of `s` that has been capitalized, which we then need to store somewhere. (Though, technically, we could store that right back into `s` with `s = s.capitalize()`, but it would be a "new" string.)

- We can swap variables, without having to dereference pointers. If we try to pass them in:

```
def main():
    x = 1
    y = 2

    print("x is {}".format(x))
    print("y is {}".format(y))
    print("Swapping...")
    swap(x, y)
    print("Swapped.")
    print("x is {}".format(x))
    print("y is {}".format(y))

def swap(a, b):
    tmp = a
    a = b
    b = tmp

if __name__ == "__main__":
    main()
```

  - The variables don't get swapped, since they are being passed in as copies again.

- But there's no way to get the pointers in Python, so the only way we can swap values is this:

```
x = 1
y = 2

print("x is {}".format(x))
print("y is {}".format(y))
print("Swapping...")
x, y = y, x
print("Swapped.")
print("x is {}".format(x))
print("y is {}".format(y))
```

  - In Python, we can actually swap variables with one line. The left side and right side, `x, y`, and `y, x` are both tuples, a data structure with multiple values, and we're setting the items inside `x, y` to what the items inside `y, x` are, which swaps the values.

- A function, too, can return multiple values, so we might need to save them with something like `a, b, c, d = foo()`
- Let's implement structures in Python:

```
import cs50
from student import Student

students = []
for i in range(3):

    print("name: ", end="")
    name = cs50.get_string()

    print("dorm: ", end="")
    dorm = cs50.get_string()

    students.append(Student(name, dorm))

for student in students:
    print("{} is in {}.".format(student.name, student.dorm))
```

  - First, we declare a `student` file that we'll soon write, and import the `Student` class from it.
  - Then we can create an empty list to store students called `students`, which we can add or remove things to.
  - Then we get a `name` and `dorm`, create a `Student` objects by passing those strings in as arguments, and `append` it, or add it, to the end of our list `students`. (Lists, too, have built-in functionality, one of which is `append`.)
  - Finally, for each `student`, we print the properties back with the `.` syntax.
- So to create our `student` module, we would:

```
class Student:
    def __init__(self, name, dorm):
        self.name = name
        self.dorm = dorm
```

  - We declare a `class` of objects called `Student`, which will only have one method, or built-in function, `__init__`, which we won't call directly but gets called when we create a `Student` as we did above with `Student(name, dorm)`.

- This function gets the object itself as an argument and the other arguments we want to be passed in when the object is created, in this case `name` and `dorm`. Then inside the function, we store the arguments to the object that's just been created.

- We can see another convenient feature:

```
import cs50
import csv
from student import Student

students = []
for i in range(3):

    print("name: ", end="")
    name = cs50.get_string()

    print("dorm: ", end="")
    dorm = cs50.get_string()

    students.append(Student(name, dorm))

file = open("students.csv", "w")
writer = csv.writer(file)
for student in students:
    writer.writerow((student.name, student.dorm))
file.close()
```

  - Now, instead of printing the students to the screen, we can write them to a file `students.csv` by opening it and using a built-in module, `csv`, that writes comma-separated values files.
  - With `csv.writer(file)`, we pass in the file we open to get back a `writer` object that will take in tuples, and write them to the file for us with just `writerow`.

- If we were to run this program without `import csv`, the interpreter would start the input, collecting input like `name` and `dorm` and creating `students`, but only when it reaches the line that calls for `csv` will it notice that it wasn't defined, and raise an exception (stop the program because there is an error).

- We can re-implement all the examples from weeks 1 through 5 in Python, and even the entire `speller` (http://cdn.cs50.net/2016/fall/lectures/8/src8/speller/) program.

- More interestingly, we can look at just the `dictionary.py` file:

```
class Dictionary:

    def __init__(self):
        self.words = set()

    def check(self, word):
        return word.lower() in self.words

    def load(self, dictionary):
        file = open(dictionary, "r")
        for line in file:
            self.words.add(line.rstrip("\n"))
        file.close()
        return True

    def size(self):
        return len(self.words)

    def unload(self):
        return True
```

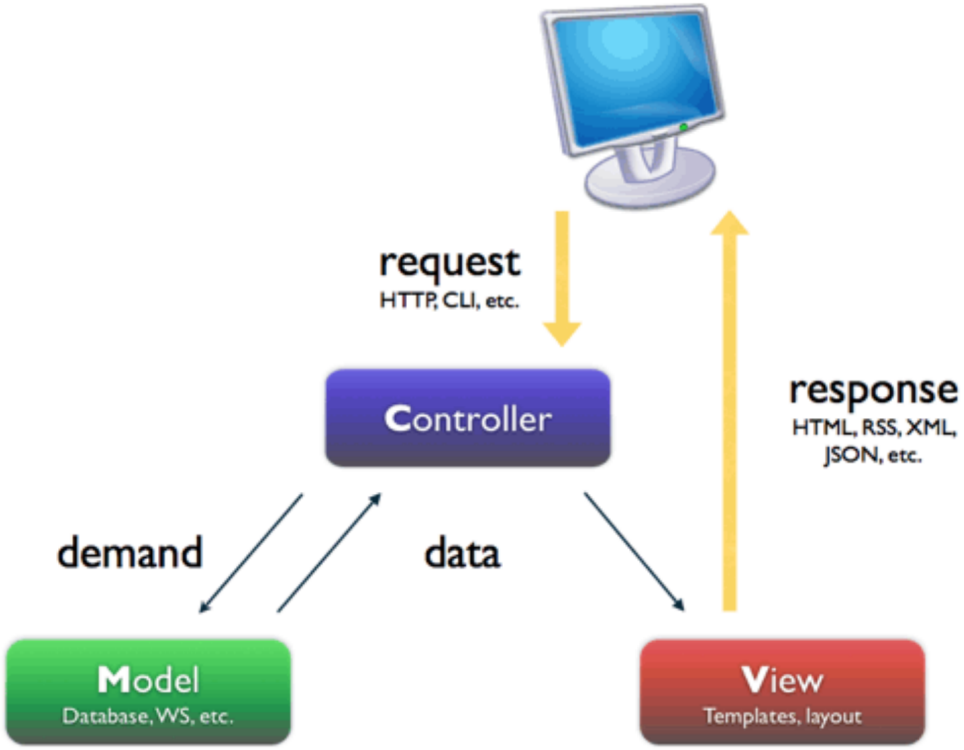
- Here, we create a `words` property when each Dictionary is initialized, and set it to an empty `set`. In Python, sets are abstracted (so we don't know anything about how it's implemented in memory anymore, or whether it's a hash table, or trie, or something else entirely) but we can easily operate with it.
- We can add items to `self.words` with `self.words.add()`, check if a word is in it with `word in self.words()`, and get the size with `len(self.words)`.
- And since Python mananges our memory for us, we don't even need to worry about unloading it or freeing it.

- As we see above, a higher-level language like Python, which has implemented many lower-level features that would take dozens of lines in C, allows us to write more and more sophisticated programs without having to worry about all of the details.

Web Servers

- In Week 6 we talked about how servers and browsers communicate, but we just opened HTML files that we wrote in our own workspaces, without talking to a server that could generate a dynamic response.

- Our goal will be to implement a web server in Python, that can take in an HTTP request, and respond with some response and some generated HTML content.
- But before we get there, we need a mental model:



  - This popular paradigm, or design pattern, for web software is called MVC, Model-View-Controller.
  - The Controller has the logic for determining what the code does in response to requests, such as checking if a user is logged in.
  - The View has the look of the site, with HTML templates and CSS styles.
  - The Model has the data that the controller uses to fill in Views, which will then form what the user gets back.
- Today we'll focus on the V and the C.
- Python comes with built-in web server capabilities, that starts a program on your computer that listens to requests from the internet to your computer, and responds to them. We'll create our own `HTTPServer_RequestHandler` that inherits (takes the methods of) the `BaseHTTPRequestHandler` class that comes with Python:

```
from http.server import BaseHTTPRequestHandler, HTTPServer

# HTTPRequestHandler class
class HTTPServer_RequestHandler(BaseHTTPRequestHandler):

    # GET
    def do_GET(self):
        # send response status code
        self.send_response(200)

        # send headers
        self.send_header('Content-type','text/html')
        self.end_headers()

        # determine message to send to client
        if self.path == "/":
            message = "Hello, world!"
        else:
            name = self.path[1:]
            message = "Hello, {}!".format(name)

        # write message
        self.wfile.write(bytes(message, "utf8"))
        return
...
```

- We'll write our own `do_GET` function for the server that is called when a `GET` request is received. We'll always send back the response code `200`, send a header, and write a message back.
- And all of these functions and features we'd learn about from reading Python documentation online.

- Then we need to start our server:

```
...
def run():
  print('starting server...')

  # set up server
  port = 8080
  server_address = ('127.0.0.1', port)
  httpd = HTTPServer(server_address, HTTPServer_RequestHandler)

  # run server
  print('running server on port {}...'.format(port))
  httpd.serve_forever()


run()
```

  - We specify the port that we want to listen to messages from, the address of the server (`127.0.0.1` is always our own computer, create an `HTTPServer` that's built-into Python, but giving it our own `HTTPServer_RequestHandler`. And finally we run it with the `serve_forever` function.

- We start the program, and nothing seems to happen. But if we open our browser and visit `http://127.0.0.1:8080 (http://127.0.0.1:8080)` (on the CS50 IDE, the address will be different), we'll see:

```
hello, world
```

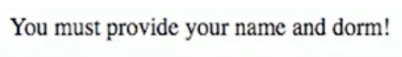
- But this is just text, and to write out code that generates HTML from scratch would be a lot of work.
- We can use a framework, a collection of code that contains even more functionality that we can use to build projects on top of.
- One such framework is Flask, which has some basic functionality we can use. A basic application will look something like this:

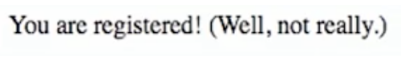
```
from flask import Flask, redirect, render_template, request, session, url

app = Flask(__name__)

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

@app.route("/register", methods=["POST"])
def register():
    if request.form["name"] == "" or request.form["dorm"] == "":
        return render_template("failure.html")
    return render_template("success.html")
```

- We first `import` lots of functionality `from flask`, and create an `app`.
- Then we have a line `@app.route("/")` that says the next function should be called whenever that path on the web server is requested. In this case, the function will return `render_template("index.html")`, or whatever the file `index.html` looks like.
- Then if we see `@app.route("/register", methods=["POST"])`, someone sending a `POST` request to `/register`, we'll call the `register` function underneath. That function, if we don't have certain elements in the `form`, will return the template `failure.html`. Otherwise, it'll return `success.html`.

- So we'll run this app by going to the directory with our `application.py` file, and run `$ flask run --host=0.0.0.0 --port=8080`. With `--host=0.0.0.0`, we're specifying that it listens to requests for all addresses.
- Now if we visit `http://127.0.0.1:8080 (http://127.0.0.1:8080)` in our browser, we get back a form we've implemented in `index.html`:

- And if we fill out that form, we'll see an error or success message, depending on how much we've filled in:





- Even if we fill in the form, we aren't really registered for anything since this application doesn't have a database, or a place to store the data that we just entered.

- But in any case, let's look at `failure.html`:

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

{% block title %}
Registration Failed
{% endblock %}

{% block body %}
You must provide your name, comfort, and dorm!
{% endblock %}
```

  - There's not much logic here but it looks like we're extending a file called `layout.html` that probably has a basic page structure, and then with `{% block title %}` and `{% block body %}` we're indicating what should go into the title and body.

- `success.html` has something similar:

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

{% block title %}
Registration Successful
{% endblock %}

{% block body %}
You are registered!  (Well, not really.)
{% endblock %}
```

- And `layout.html` is familiar:

```
<!DOCTYPE html>

<html>
    <head>
        <meta content="initial-scale=1, width=device-width" name="viewpor
        <title>{% block title %}{% endblock %}</title>
    </head>
    <body>
        {% block body %}
        {% endblock %}
    </body>
</html>
```

- We see the same `{% block title %}` and `{% block body %}` magic words in this file, which works not because of HTML or Python but because of the Flask framework (the `render_template` function) that builds pages with these templates.

- And the form in `index.html` is even more familiar:

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

{% block title %}
Frosh IMs
{% endblock %}

{% block body %}
<div style="text-align:center">
    <h1>Register for Frosh IMs</h1>
    <form action="{{ url_for('register') }}" method="post">
        Name: <input name="name" type="text"/>
        <br/>
        <input name="captain" type="checkbox"/> Captain?
        <br/>
        <input name="comfort" type="radio" value="less"/> Less Comfortabl
        <input name="comfort" type="radio" value="more"/> More Comfortabl
        <br/>
        Dorm:
        <select name="dorm">
            <option value=""></option>
            <option value="Apley Court">Apley Court</option>
            <option value="Canaday">Canaday</option>
            <option value="Grays">Grays</option>
            <option value="Greenough">Greenough</option>
            <option value="Hollis">Hollis</option>
            <option value="Holworthy">Holworthy</option>
            <option value="Hurlbut">Hurlbut</option>
            <option value="Lionel">Lionel</option>
            <option value="Matthews">Matthews</option>
            <option value="Mower">Mower</option>
            <option value="Pennypacker">Pennypacker</option>
            <option value="Stoughton">Stoughton</option>
            <option value="Straus">Straus</option>
            <option value="Thayer">Thayer</option>
            <option value="Weld">Weld</option>
            <option value="Wigglesworth">Wigglesworth</option>
        </select>
        <br/>
        <input type="submit" value="Register"/>
    </form>
</div>
{% endblock %}
```

- The `{% block body %}` here has more HTML, a header and a form. The form also has `{{ url_for('register') }}` for its `action`, which calls a function that gets the `register` route in our app, rather than hardcodes it.

- Going back to our app where that `register` route is,

```
@app.route("/register", methods=["POST"])
def register():
    if request.form["name"] == "" or "captain" not in request.form or "co
        return render_template("failure.html")
    return render_template("success.html")
```

  - we see again the controller logic that checks whether the form is complete, and returns the correct view.

- We can demonstrate another simple app that lets us select how many `foo`, `bar`, and `baz` we want:

- And dynamically generates a page that tells us how many we've added to our cart:

- And if we closed the window and opened it, it would remember how many of each item we've added.

- The HTML for the form is simple, using elements we've seen before …

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

{% block title %}
store
{% endblock %}

{% block body %}
<h1>Store</h1>
<form action="{{ url_for('store') }}" method="post">
    <input name="foo" type="number" value="0"/> Foo
    <br/>
    <input name="bar" type="number" value="0"/> Bar
    <br/>
    <input name="baz" type="number" value="0"/> Baz
    <br/>
    <input type="submit" value="Purchase"/>
</form>
<p>
    View your <a href="{{ url_for('cart') }}">shopping cart</a>.
</p>
{% endblock %}
```

- … as is the template for the cart:

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

{% block title %}
Cart
{% endblock %}

{% block body %}

<h1>Cart</h1>

{% for item in cart %}
    {{ item["quantity"] }} : {{ item["name"] }}
    <br/>
{% endfor %}

{% endblock %}
```

    - Notice that we can list each `item` in the `cart` variable with a `for` loop, and access fields of each of the `item` object, two of which at least are `quantity` and `name`.

- And we can look in `application.py` to find out what `cart` is:

```
from flask import Flask, redirect, render_template, request, session, url

app = Flask(__name__)
app.secret_key = "shhh"

@app.route("/", methods=["GET", "POST"])
def store():
    if request.method == "POST":
        for item in ["foo", "bar", "baz"]:
            if item not in session:
                session[item] = int(request.form[item])
            else:
                session[item] += int(request.form[item])
        return redirect(url_for("cart"))
    return render_template("store.html")

@app.route("/cart")
def cart():
    cart = []
    for item in ["foo", "bar", "baz"]:
        cart.append({"name":item.capitalize(), "quantity":session[item]})
    return render_template("cart.html", cart=cart)
```

  - `shhh` is just some secret value that we use to keep our session, or shopping cart, secure. (Though it should be longer, and harder for someone to guess!)
  - In the `store` method, we first check if the request was a `POST`. If so, meaning a user submitted the form, then we'll add the quantity indicated in the form for each `item` into `session`, if it's not in that object already, or increment its quantity if it is. And `session` is an object we get from the framework, that acts like a shopping cart, where we can save information for each user connected to our web server.
  - The `cart` method creates an empty list, and stores a dictionary for each item with its name and quantity, using the session to get that value. Then we pass that into `render_template` as an argument, so we can use it to build `cart.html`, which we saw above.
- Phew, that was a lot! Yet that was barely the surface of what we have access to with the language of Python and the many many frameworks out there.