labibi Documentation

Release 1.0

C. Titus Brown

February 14, 2017

Lecture/lab: Tu/Th 2:40-4pm, McDonel Hall 2

Instructor: C. Titus Brown, ctb@msu.edu, BPS 2228(c)

TA: Cait Pickens, picken19@msu.edu

Office hours: 6-8pm Tuesdays, or by arrangement; in 2228 BPS.

Objectives:

In this course, you will learn how the Web works by writing a Web server and some Web applications. More generally, we will discuss concepts in client-server and peer-to-peer architectures and how all of this technology works "under the hood" on today's Internet. We'll also discuss issues and approaches to developing software with an eye to maintainability, and learn about the practical separation of concerns in Web application stacks, from browser through server.

Read more in the syllabus

Class resources and homework:

CHAPTER 1

Day 25 - Th, April 25, 2013

1.1 Stuff I didn't teach, couldn't teach, or forgot to teach

CHAPTER 2

Cookies and logging in

See https://github.com/ctb/cse491-webz/blob/cookies/app.py for code for seting cookies.

Security

0. As a Web developer, how do you protect against guessing?

Answer: by not being dumb.

1. As a Web developer, how do you protect against eavesdropping?

Answer: HTTPS.

2. As a Web developer, how do you protect against server compromise?

Answer: you try not to write insecure servers.

3. How do you protect against client (browser-side) compromise?

Answer: you can't.

CHAPTER 4

Scaling; latency; throughput

How do you scale up to handle hundreds or thousands of concurrent sessions?

How do you minimize latency, the “turnaround time”?

How do you maximize throughput, the amount of “stuff” being delivered?

CHAPTER 5

What do real Web developers use?

All of the tech we used in the class is "real", including WSGI, Jinja2, JSON and JSON-RPC, CSS, JQuery, SQLite, etc. But people have built layers on top of WSGI and SQL (in particular) to automate as much of the manual stuff as possible.

Real Python Web developers use things like Django and Pyramid.

Personally, I have always had a soft spot for Quixote, which is a nice, simple, extensible Web framework. To take a look,

```
source env491/bin/activate.csh
wget http://quixote.ca/releases/Quixote-2.7.tar.gz
pip install Quixote-2.7.tar.gz
tar xzf Quixote-2.7.tar.gz

cd Quixote-2.7
python quixote/server/simple_server.py  --port=8000
```

See the files quixote/demo/root.ptl and quixote/demo/extras.ptl for the source for the examples.

CHAPTER 6

Day 24 - Tu, April 23, 2013

Please go fill out this form.

Homework #6 - the last one!

Due Wed, May 1st, at midnight. Office hours on Tuesday Apr 23rd, but then I'm out of town on Apr 30th; arrange something with Cait if you want help.

0. See point 0 in Homework #5, but fill out this form instead.
1. Implement a WSGI server that speaks POST well enough to handle a JSON-RPC request. Test it on your live Web app. (20/100)
2. Implement your feature from HW #5.2. Give me documentation, somewhere, on what it does and how – hey, maybe it can be on a Web page! (20/100)
3. Change db.py to use SQLite to save/load everything. (20/100)
4. Modify HW 4.2(c) with AJAX/JSON-RPC instead of an HTML form submit. (20/100)
5. Implement logging in with cookies. (20/100)

Hand in by tagging as hw6.

7.1 Honors option

Complete 4 or more points below. (sum score >= 4.0)

1. Install a NoSQL database and provide a branch on github where you use that instead of SQLite for back-end storage. E-mail me the branch info. (1 point)
2. Get your code running on a public Web site (virtual machine/rented host). E-mail me the URL and keep it running for 24 hours. (2 points)

4. Get your tests and code running on ShiningPanda continuous integration. E-mail me the URL for a public view of the passing tests. (2 points)

3. Implement Selenium/SauceLabs testing of the AJAX feature (#6.4) above. E-mail me a video. (3 points)

CHAPTER 8

Day 23 - Th, April 18, 2013

Schedule:

- Minute cards.
- AJAX and JSON-RPC
- More database: joins, transactions, multithreading, synch, etc.

8.1 Questions/TODO

(This is an example of using the jQuery JavaScript library with AJAX and JSON-RPC)

Grab the latest version of https://github.com/ctb/cse491-webz; for example, on the CSE cluster, you could do:

```
mkdir day23
cd day23
python2.7 -m virtualenv env
source env/bin/activate.csh
pip install simplejson
```

to get the environment running, and then

```
git clone https://github.com/ctb/cse491-webz.git
cd cse491-webz
python app.py
```

Now, go to the URL of the Web server, and take a look at the '/content' URL, which is loading from 'somefile.html'. (You can look at the source of 'somefile.html' on github, if you want.)

On the '/content' page, you'll see three input box regions

The value in the first is set dynamically by JavaScript using the jQuery library.

The value in the second is output in an alert box upon change.

The two values in the third are summed and displayed on the page.

TODO:

1. At your table, talk through the code for the first and second input boxes. What's going on? (One thing you can do to find out is edit the HTML to remove the first <script>, which loads jQuery; this will disable all the JavaScript.)

2. For the third input box ("retrieve from server"), write down a flowchart on your whiteboard that details, in order, what happens in terms of network traffic and function calls in Python (on the server side) and JavaScript (on the client side). Your flowchart should include
3. For the last text box ("retrieve from server"), why does an alert box pop up after you enter the first value?

8.2 SQL Multiple tables, and Transactions

Tables.

Transactions.

8.3 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

Day 22 - Tu, April 16, 2013

Schedule:

- Minute cards.
- Database foo; SQLite

9.1 Questions/TODO

Following the code in this sqlite IPython Notebook,

1. As a group at each table, discuss where in your code (specific code functionality) the database should be created, where the database should be opened, where it should be accessed (read only) and where the data in the database should be updated or modified.

   Write down your answers for discussion.

2. What would a database schema for your bottle types DB look like? Write a small script on the CSE cluster (to call with python2.6) that opens a new database and creates that schema.
3. Write an INSERT statement to add a new bottle type. Be sure to use '?' placeholders and variables. Add it to your script. Make sure the INSERT statement actually adds to the table (by calling SELECT * afterwards).
4. Write a SELECT statement to handle the query in _check_bottle_type_exists (in your drinkz/db.py). Add it to your script and print out the results (c.fetchall()).

NOTE, you have to use python2.6 on the CSE cluster in order to use sqlite3.

9.2 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

Homework #5

Due Friday, Apr 12th at 5pm. Note, no class on Thursday; no office hours on Tuesday. I'd suggest starting it BEFORE class on Tuesday so that you can ask questions in class.

0. To hand in your homework, tag the latest master as 'hw5' and fill out this form. If I don't see your code on github, I will not grade this homework and you will get a 0. So, verify by going to github and finding the tag and checking it, and then doing a clean clone and test run of that specific tag. Ask for help if you have trouble doing this. I mean it. I am serious. Don't screw around. Double-check your push. If you can't find it on github, I probably can't either.

1. Finish implementing recipes. (40/100)

   Specifically,

   (a) Implement feature #3

   (b) Implement bulk loading of recipes from the command line.

   (c) Write HTML forms to enter liquor types, liquor inventories, and recipes.

   (d) Write JSON-RPC functions for the same.

   (e) Write tests against everything but the HTML forms.

   Describe how to bulk-load recipes in RECIPES.txt in your top-level directory.

   Use Jinja2 templating so that your functions use template rendering to dynamically construct strings that are returned via the WSGI server (10 of the 35).

2. Write up a feature implementation. (20/100)

   Look through the use cases here and here and choose one to write up. You will be asked to implement this (or another one) for HW #6.

   Your feature should have –

   (a) a data component

   (b) an internal API for storing/retrieving/querying the relevant data

   (c) a Web interface for display and modifying the relevant data

   (d) a JSON-RPC set of functions

   So, write up a paragraph or three as in the recipes use case and put it in a text file called FEATURE.txt in the home directory of your drinkz repository.

   If you work collaboratively, please write up multiple stories (i.e. you're welcome to brainstorm, but if two people are brainstorming, make up two features).

3. If your last name starts with a letter whose ord in Python is odd, do this (40/100):

   Write an WSGI server that speaks HTTP well enough to receive a GET, call a WSGI app (as in http://www.python.org/dev/peps/pep-0333/), and return the status and HTML. Use only the 'socket' networking library; you should be speaking low-level TCP/IP, nothing else.

   A strong suggestion is to first write an HTTP server that returns just a static string (see 'get-page' from HW 4), and then slowly modify that to return the results of calling your WSGI app.

   See server.py from Day 20 - Th, April 2, 2013 for basics of TCP/IP networking and binding a socket.

   Describe how to run your server in SERVER.txt.

4. If your last name starts with a letter whose ord in Python is even, do this (40/100):

   Develop a simple WSGI app (e.g. see cse491-webz/app.py) that can be attached to a WSGI server (as in http://www.python.org/dev/peps/pep-0333/) and a set of client tests that use the 'socket' library to connect to the WSGI server and run the functions in the WSGI app, and test that you get back the right thing from the server.

   Your client & app should exercise:

   - a straight up GET
   - a form submission GET
   - image retrieval

   Describe how to run your client tests (they can be run using nose if you want, or just as a separate script) in CLIENT.txt.

Hint:

```
>>> print ord('a')
```

Also, ask questions :)

CHAPTER 11

Day 20 - Th, April 2, 2013

Schedule:

- Moar discussion of HTTP
- Work on questions from Day 19 - Th, Mar 28th, 2013 and today.
- Minute cards.

Note: no class on Thursday.

11.1 TCP/IP Networking

You can use this for a reference –

http://www.tutorialspoint.com/python/python_networking.htm

Take a look at the stuff under network/ in the latest cse491-webz –

> https://github.com/ctb/cse491-webz.git

This is a very simple client/server back-and-forth for TCP/IP.

Questions –

1. How can one side of a connection know how much data remains to be read from the other side of the connection?
2. Run the server, connect to it with the client, and then use CTRL-C to kill it. What signal does the client get?
3. Modify the client to send a file to the server upon connection. Make sure that both sides "clean up", that is, exit properly.

   Note:

   ```
   data = open(filename).read()
   ```

   to read bytes in from a file, and

   ```
   fp = open(filename, 'w')
   fp.write(data)
   fp.close()
   ```

   to save data to a file.
4. Modify the server to send a file to the client as soon as the client connects. Make sure that both sides "clean up".
5. Modify the client to send a file containing text that the server then

(a) uppercases

(b) replaces all As with ZZs

Note,

```
s = t.replace('A', 'ZZ')
```

will do the latter.

11.2 Templating with Jinja2, round 2

Look at the Jinja2 stuff from Day 19 - Th, Mar 28th, 2013, and grab the latest cse491-webz. Try rendering 'test4.html' and 'test5.html'; look at templates/test4.html and templates/test5.html. What's going on?

Things to try –

- modify 'is_tuesday' in render.py to be True. What happens in test4.html?
- What does the |e do (see {{ name|e }}) in test5.html? Try removing it, and load stuff in a browser.
- Revisit test3.html. Do you understand what's going on here?

11.3 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

CHAPTER 12

Day 19 - Th, Mar 28th, 2013

Schedule:

- Discussion of HTTP
- Ask homework-related questions & clarifications.
- Work on questions from Day 18 - Tu, Mar 26th, 2013 and today.
- Minute cards.

12.1 More about HTTP

Read this document and connect it to what I show you in class today & your 'GET' work. Ask questions!

Topics I'll try to cover today:

- connecting to port 80 of 'lyorn.idyll.org' and 'www.google.com'
- structure of HTTP requests and responses
- "metadata" other than received and submitted content
- URLs at google.com

12.2 Using JSON-RPC in anger

Take a look in the 'twitter/' subdirectory of

```
https://github.com/ctb/cse491-webz
```

These scripts turn an address INTO a geolocation (latitude & longitude) using the Google Maps API, and turn a geolocation back into an address (using the Twitter geolocation API, from v1 of the API).

NOTE You'll need to use 'python2.6' instead of 'python' or 'python2.7'. Sorry!

Your mission –

1. Hook the scripts or code up so that you can take an address and see what happens when you geolocate it via Google Maps and decode it via Twitter.
2. Look through the latest Twitter API (v1.1) and work on gaining authenticated access to the API for your own Twitter account. Note, excessive googling for answers may be needed!

   Bonus: sign up for a Twitter account if you don't have one ;).

12.3 Templating with Jinja2

As you have probably already seen, there's lots of repetition in your various HTML files. This will get worse and worse as you add stylesheets, JavaScript, etc. Plus, you run the very real risk of lots of duplicate code, some of which will be right and some of which will be wrong.

The correct solution for this is *templating*. We'll be using jinja2 templating, which is similar in concept to many other templating systems.

The basic idea is that you have a template "source file" that you then "render", providing variables to fill in for whatever needs to be customized for that page.

12.3.1 Trying it out

Grab the latest:

```
https://github.com/ctb/cse491-webz
```

Enable your virtualenv:

```
source env491/bin/activate.csh
```

And install jinja2:

```
pip install jinja2
```

Now, go into cse491-webz/jinja2/, and try running:

```
python render.py test.html
```

and

```
python render.py test2.html
```

and

```
python render.py test3.html
```

Look at the files in 'templates/'. What's going on here?

12.4 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

CHAPTER 13

Day 18 - Tu, Mar 26th, 2013

Schedule:

- Ask homework-related questions.
- Work on questions from Day 17 - Th, Mar 21st, 2013 and today.
- Minute cards.

13.1 Style sheets and HTML

Go to http://www.oswd.org/ and find a design you like; download it, and unpack it (probably 'unzip' should work). Look at the index.html file and reverse engineer the various style sheet and HTML contributions to the pretty design.

Then, modify cse491-linkz (https://github.com/ctb/cse491-linkz/) to output files that incorporate the styling of the OSWD design you downloaded. Note the latest commit to cse491-linkz...

> https://github.com/ctb/cse491-linkz/commit/f303f182016ae30d7b503645d8e709bf8a1e8362

Ask questions as you have them.

13.2 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

CHAPTER 14

Day 17 - Th, Mar 21st, 2013

Schedule:

- Read over Homework #4 and ask questions.
- Read answers to questions from Day 13 – Tu, Feb 19th, 2013 and Day 14 – Th, Feb 21st, 2013.
- Work on questions from Day 15 - Tu, Mar 12th, 2013, Day 16 - Tu, Mar 19th, 2013, and today.
- Minute cards.

CHAPTER 15

Testing WSGI and Web apps, round 1

Update your cse491-webz branch with the latest master from https://github.com/ctb/cse491-webz, and run the tests in test_app using nose:

```
%% nosetests
```

Now go look at 'test_app.py' –

1. What is the call order of functions to get down to the 'index' function in app.py in test_index()?
2. What is the call order of functions to get down to the 'recv' function in app.py in test_recv()?
3. Why is the '/form' function not called in the tests at all?
4. Try refactoring test_app.py so that common code in test_index and test_form_recv is in one function that is then called in those two test functions.

CHAPTER 16

Slightly more advanced magic

Look at the 'magic' branch on https://github.com/ctb/cse491-webz, especially the calls under __main__:

```
https://github.com/ctb/cse491-webz/blob/magic/json-rpc-client.py#L52
```

This is an attempt to clean up the equivalent calls from the main branch:

```
https://github.com/ctb/cse491-webz/blob/master/json-rpc-client.py#L30
```

How does this work??

More specifically,

1. What is the chain of function calls that leads to and from 'call_remote' on the magic branch when you ask for 'magic.hello()' the first time? And the second time?
2. What does JSON_RPC_Magic do, and why is it separate from or different from MagicFunction?

16.1 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

Homework #4

Due Mar 27th at 11:59pm. I'd suggest starting it BEFORE class on Tuesday so that you can ask questions in class or at office hours.

(See Day 11 – Tu, Feb 12th, 2013 for collaboration instructions.)

0. Cleanup. I won't grade your homework if this isn't the case.

   (a) Merge with my master branch:

      https://github.com/ctb/cse491-drinkz.git

   such that 'git pull https://github.com/ctb/cse491-drinkz.git master' says "up to date", and make sure that your code includes everything through the end of HW 3. This includes fixes for anything that was graded as 'broken' at the end of HW 3.

   (a) The default branch should be master, so,

      https://github.com/<USERNAME/cse491-drinkz/blob/master/<filename>

   should be what's on the main page when you go to

```
https://github.com/<USERNAME>/cse491-drinkz
```

   and when you do a 'git clone' of your repository, you should have your latest code in default branch.

   (See https://github.com/blog/421-pick-your-default-branch.)

2. Refactor/fix. 10/100.

   (a) Put your unit conversion code in one file, consolidated into a single function. Your tests shouldn't need to change.

   (b) Make sure all your tests pass. This doesn't mean "delete them", this means "fix your code."

   Done.

3. Integrate a Web frontend to your drinkz code. 50/100.

   Copy or merge or otherwise integrate 'app.py' from cse491-webz (https://github.com/ctb/cse491-webz/blob/master/app.py) into the 'drinkz/' directory, and then –

   (a) write a 'bin/' script called 'run-web' that starts up a Web application as at the bottom of the current app.py.

   (b) dynamically generate the Web pages from HW as part of 'app.py'; for example, 'localhost:9786/' should show an index page that links to list of recipes, inventory, etc.

   (c) provide another link on your index page that connects to a page with a form on it that takes in some amount of liquid in a text box, and, when 'submit' is clicked, goes to a new Web page that shows the amount converted into ml. Make sure that there are links back to the index page, too.

   (d) create a new file 'drinkz/test_app.py' that exercises the WSGI interface by
       - initializing a clean database with some recipe data;
       - create a new SimpleApp object;
       - calls the __call_ function on the SimpleApp object with an 'environ' dictionary and a 'start_response' function of your own creation.
       - runs the code to generate the page that lists the recipes.
       - checks that the right recipe data is on that generated page.

4. Save to/load from disk. 15/100.

   Merge really trivial file saving/loading. More specifically,

   (a) merge the code in https://github.com/ctb/cse491-drinkz/tree/hw4-save-load into your repo, alter the db.load_db and db.save_db functions to save and load recipes as well, and make sure that you can "properly" save and load bottle types, inventory, and recipes.

       See bin/save-load for example usage/script creation.

   (b) Create a script 'bin/make-test-database' that adds some data (recipes, etc.) and then saves the database into a filename provided on the command line.

   (c) Modify make-html.py and app.py (from #2) to load that data in.

   Note, it would be really nice if you made sure that you could create a file with (b) that I could then load with (c), and if it were really obvious what the filename should be. Hint. HINT.

5. Simple CSS/JavaScript. 15/100. (You'll need to get #2 working first.)

   Integrate some really CSS and JavaScript into your HTML. Go check out the latest commit to cse491-linkz and then:

   (a) Modify all your Web pages in #2 to contain the html, head, and body regions, as well as a title. Make the title different on each page, please.

   (b) Modify all your Web pages in #2 to contain the <style> modification, and put an <h1> title on each page (which should now be red). The <h1> title should be different on each page.

   (c) On your index page, add the button to show an alert box with JavaScript. Make sure it works!

   Note: if you have other obvious CSS and JavaScript on your pages, e.g. because you've put together a style thing, you can skip this part of the homework. Just make sure it's obvious, m'kay?

6. JSON-RPC. 15/100. (You'll need to get #2 working first.)

   (a) Change the JSON-RPC functionality in app.py to provide the following functions:

```
convert_units_to_ml(amount) - given a str amount, returns ml
get_recipe_names() - returns a list of all recipe names
get_liquor_inventory() - returns a list of (mfg, liquor) tuples.
```

   (b) Create a new file 'drinkz/test_jsonrpc.py' that tests these functions through the 'app' interface. That is, you should have at least three tests, each of which
       - initializes a clean database with some data (if needed)
       - creates a new SimpleApp object
       - calls the __call__ function on the SimpleApp object with an 'environ' dictionary and a 'start_response' function of your own creation.
       - 'environ' should contain the information (PATH_INFO, CONTENT_LENGTH, etc.) that makes the SimpleApp run the relevant JSON-RPC accessible function (e.g. rpc_get_recipe_names).

- checks to make sure that the WSGI app returns the right answer.

  Under no circumstances should you directly call anything other than __call__ on the WSGI SimpleApp object, i.e. this should mimic closely a “normal” call from the WSGI server into the app object.

  Protip: use simplejson as in https://github.com/ctb/cse491-webz/blob/master/json-rpc-client.py and a StringIO object (http://docs.python.org/2/library/stringio.html) to set up environ['wsgi.input'] to pass in the JSON necessary to make the call. You might also want to read http://en.wikipedia.org/wiki/JSON-RPC.

7. HTTP GET. 15/100. (You’ll need to get #2 working first.)

   Write a standalone Python script that does an HTTP GET using the ‘socket’ library in Python. More specifically, read http://effbot.org/zone/socket-intro.htm and then write a standalone script called ‘grab-page’ (not in bin, or drinkz/, and not with a .py on the end, and not with an underscore) in the root directory of your repository. This script should take two command line arguments, the hostname and the port for your running app.py server, and print out the results of submitting a GET request for ‘/’ on the app.py server.

   This might be worthwhile reading, too, if you’re confused or interested:

   http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

8. Hand in your homework by tagging it as tag ‘hw4’:

```
git tag hw4
git push origin hw4:hw4
```

   I strongly suggest that you make sure you can clone your repo into a new directory, check out ‘hw4’, and run all of your tests properly. You might want to double-check that everything above works, too...

   Note that it’s fairly easy to delete tags, so you should try this early on and tell me if it doesn’t work; then you can delete the first hw4 tag and update it to whatever you want to hand in.

CHAPTER 18

Day 16 - Tu, Mar 19th, 2013

- Read answers to questions from Day 13 – Tu, Feb 19th, 2013 and Day 14 – Th, Feb 21st, 2013.
- Work on questions from Day 15 - Tu, Mar 12th, 2013 and today.

18.1 In class

Work on the in-class exercises from Day 15 - Tu, Mar 12th, 2013, or work on the following –

18.1.1 JSON-RPC

Set up your virtualenv and install simplejson:

```
%% source env491/bin/activate.csh
%% pip install simplejson
```

Next, update your existing cse491-webz directory, or clone a new one (c.f. https://github.com/ctb/cse491-webz).

Open up two login windows to CSE.

In one, run 'app.py':

```
%% source ~/env491/bin/activate.csh
%% python app.py
```

In the other, run the 'json-rpc-client.py' file with the URL of your server:

```
%% python json-rpc-client.py http://arctic.cse.msu.edu:8231/
```

but replacing the last URL with your actual URL.

What is going on here??

Note, you can do this FROM anywhere to the CSE server... including your laptop, if you install simplejson there.

18.1.2 Questions

1. In the 'dispatch' machinery in app.py, why are the values also strings? Couldn't I just have used functions directly?
2. Why does the start_response make a copy of html_headers, e.g. why 'list()'?

```
    start_response('200 OK', list(html_headers))
```

3. What URL on the server is called for the ‘hello’ function?
4. What server-side function is called when the JSON-RPC client asks for ‘hello’? More generally, how does the JSON-RPC server-side machinery know what function to call?
5. What is the JSON format, and why do we need to use it?

18.2 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

Day 15 - Tu, Mar 12th, 2013

- Mini lecture
- Read answers to questions from Day 13 – Tu, Feb 19th, 2013 and Day 14 – Th, Feb 21st, 2013.
- Work on integrating webz, linkz,

19.1 Answers to questions from Day 13 and Day 14

Read through these questions and answers and make sure you grok both. If you have specific questions to ask about any of this, stop by the and ask 'em.

Note, we'll have a quizlet next Tuesday on WSGI and Web stuff to gauge absorption of the material. PANIC.

From Day 13 – Tu, Feb 19th, 2013:

1. How does your Web browser know how to contact *your* app.py instead of your neighbor's?

   Answer: the hostname and port (http://hostname:port/) are unique identifiers to your running application.

2. Where is the logic in app.py for what is returned?

   Answer: See app.py, 'if path == ' section. Here, 'path' is everything AFTER the hostname and port, above.

3. What role does content type play in what is returned?

   Answer: the browser (on the receiving end, i.e. your computer) uses content-type to decide how to interpret was is returned – HTML, an image, etc.

4. What variable type is 'data'?

   Answer: A string (potentially a binary string, but still a string).

5. What role does 'status' play?

   Answer: 'status' is essentially metadata ABOUT the connection, and it helps the browser decide what to do with the data it receives. For example, '200 OK' means "everything is fine! no worries!" while '403 Not Found' means "whatever I'm giving you is basically an error message." There are redirects and other things, too.

6. What happens if 'somefile.html' isn't present?

   Answer: An error! Because we don't trap exceptions within the SimpleApp class, the WSGI server catches them and returns a complaint.

7. How does the Web browser know what is being returned?

   Answer: See #3, above; 'content-type'.

8. Why do we generate a random number at the bottom of the script?

   Answer: that's the port. If you and another person try to bind the same network port, only one of you will get it and the other person will get an error. (Try it!)

9. How is PATH_INFO generated, ultimately?

   Answer: PATH_INFO is whatever's on the URL line after the hostname and port (and before any ? – more on that later)

10. What happens when you 'print' something from within app.py?

    Answer: It goes to the standard output of the process *running* app.py, which (if you followed all my instructions above) is arctic.cse.msu.edu.

11. When does __call__ in app.py get executed?

    Answer: when the WSGI server receives a request! Read more about that here and in days to come.

From Day 14 – Th, Feb 21st, 2013,

1. How many Web requests are made to the server to load '/'?

   Answer: two, technically, although you may see three (see favicon.ico, below). First, a Web request to load the HTML, and then a second one to load the image ('<img>' tag).

   Technically, only one "thing" is loaded at a time – one bundle of content-type and data. Each additional bundle (different content type or logical unit of content – think images, CSS, JavaScript, etc.) requires an additional request.

2. When you press 'submit' on the form page, what URL is received by the server?

   Answer: whatever is specified in the 'action' tag of the form.

3. What does 'formdata' look like, and what does urlparse do to it?

   Answer: it's a string that looks like 'key=value&key2=value2', and it encodes all of the data from the form. urlparse turns it into a dictionary-like data structure.

4. Why does the URL for 'recv' have a '/' before it when the form doesn't specify a '/'?

   Answer: the browser treats URLs like file paths, so if you are at a form from URL '/some/place', and submit to 'process', it will automatically set the URL to '/some/process'. You can use '../' and './' as you would on a file system.

5. What is 'favicon.ico'?

   Answer: it's a convention for the "site icon" – if the browser can load a site's /favicon.ico, then it uses the icon in the tool bar.

6. What determines that 'index.html' is served from the CSE Web server for URLs ending in '/'?

   Answer: as in 'app.py', whatever Web server is running determines that. It's a convention, not a requirement.

19.2 In class

Work on one of the following –

1. Integrating 'cse491-linkz' and 'cse491-webz',

   > https://github.com/ctb/cse491-linkz https://github.com/ctb/cse491-webz

   so that your 'webz' is serving the files created by 'linkz'.

2. Integrate the *dynamic* content generation from 'linkz' – i.e., the functions that produce HTML – into the 'webz' calls, so that instead of the linkz code writing a file that webz serves, the webz code calls a function that produces the HTML directly.
3. Design some forms for cse491-drinkz functionality, including –
    - Adding a bottle type
    - Adding to inventory
    - Adding a recipe

    In the latter case, how could you design a form to add multiple ingredients etc? Do you want to use pull down menus or select ingredients from a menu? Think about what you would want to see as a user.

    Strong suggestion – write Python code to generate the HTML in the forms. That way, if you want to get information from the db module, you can do so when generating your repository. Plus, extra abstraction is always good, right?
4. Clean up your github repository. For example,
    - Make sure that 'master' is the default branch when you clone your repo.
    - Make sure that your master branch contains everything up through HW 3.
    - Eliminate any unneeded branches and tags (please leave HW 3 alone :)
    - Eliminate and/or add to .gitignore any .pyc files
    - Make sure your code is up to date with HW 3, if it's already been graded.
5. Work on code cleanup. For example,
    - Change unit conversion over to use dictionaries, and have only a single function.
    - Eliminate or fix messy or inaccurate comments.

19.3 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

CHAPTER 20

Day 14 – Th, Feb 21st, 2013

- Group in pairs
- Complete in-class exercise about WSGI
- Minute cards

20.1 Pairs

Please distribute yourself as on Tuesday into the Day 13 groups (see Day 13 – Tu, Feb 19th, 2013).

20.2 WSGI, Web servers, and extra features

Grab the latest additions to cse491-webz:

```
% cd cse491-webz
% git pull https://github.com/ctb/cse491-webz.git master
```

This may require resetting your repository first – if you want to just revert to the last commit, do

```
% git checkout -f master
```

NOTE, this will eliminate any changes you have made. Make sure you want to do that :)

Now run app.py, and play with the new features (image display and forms). Then contemplate the following questions, in addition to the questions from Day 13 (see Day 13 – Tu, Feb 19th, 2013):

1. How many Web requests are made to the server in order to load the main (/) page, and why?
2. When you press 'submit' on the form page, what URL is received by the server?
3. What does 'formdata' look like, and what does urlparse do to it?
4. If you look at the URL for 'recv', why does it have a '/' before it when the form doesn't specify a '/'? Where does the '/' come from?
5. What is 'favicon.ico'
6. From earlier work, you know that 'index.html' is served from the CSE Web server when http://www.cse.msu.edu/~username/ is requested. What determines that?

20.3 Next steps

Please try to merge 'cse491-linkz' and 'cse491-webz' so that webz is serving the static files that were produced by cse491-linkz.

Note that you can merge two DIFFERENT repositories with 'git pull', too...

20.4 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

CHAPTER 21

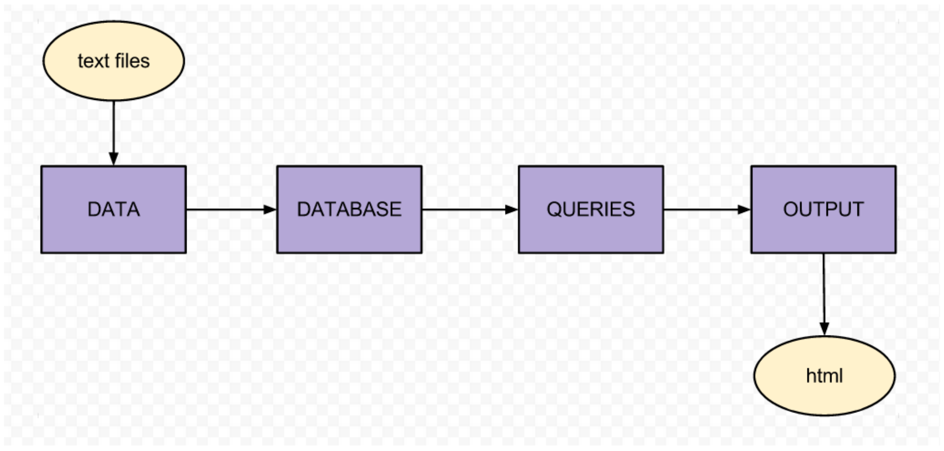
Day 13 – Tu, Feb 19th, 2013

Rough schedule for today, summary:

- Discuss the foundation for the drinkz web app
- Form new pairs
- Complete in-class exercise about confidence with topics covered in class
- Complete in-class exercise about WSGI

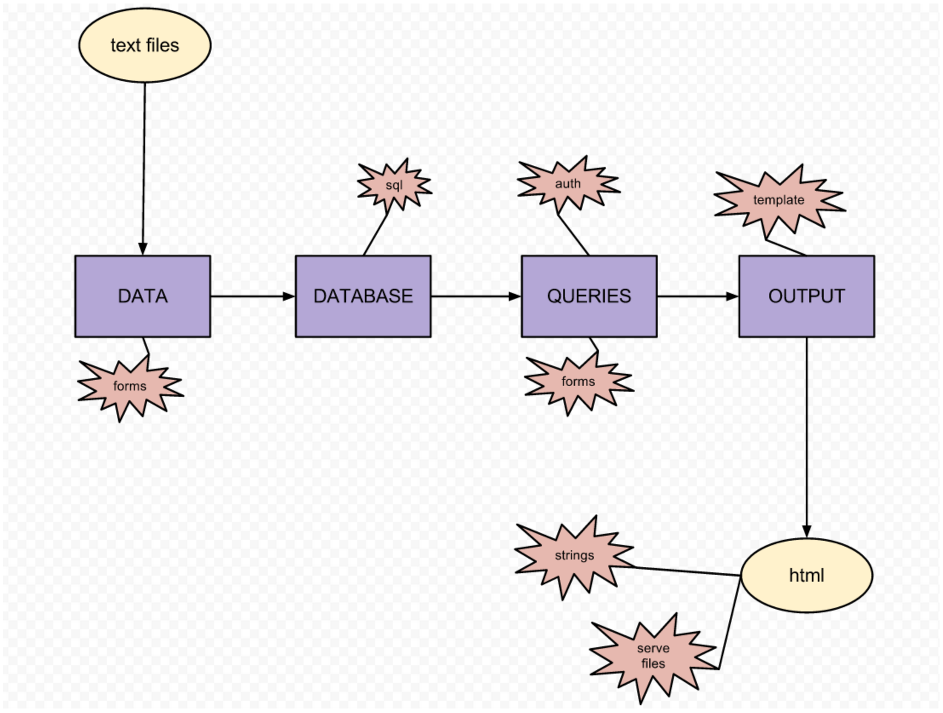
21.1 Web App Foundation

Thus far in class, we have laid a foundation for building a web app. We start from text files, turn it into data, collect the data into an in-memory database allow querying of the database, return query results, and output everything as html files:

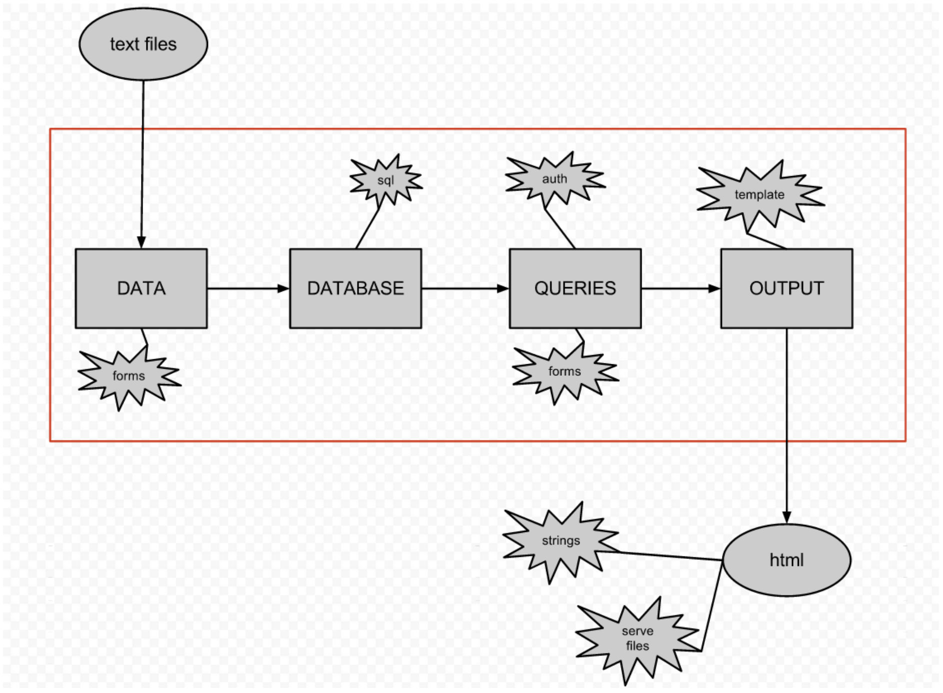


In order to develop the drinkz project, we need to expand on certain levels of this foundation. Paths from here are:

- Outputting strings or serving files
- Converting the data structures to a SQL database
- Using web forms to get data or query the database
- Including some form of security/authentication
- Incorporating templating



As we add these functionalities to the drinkz framework, we will see a web app emerge:

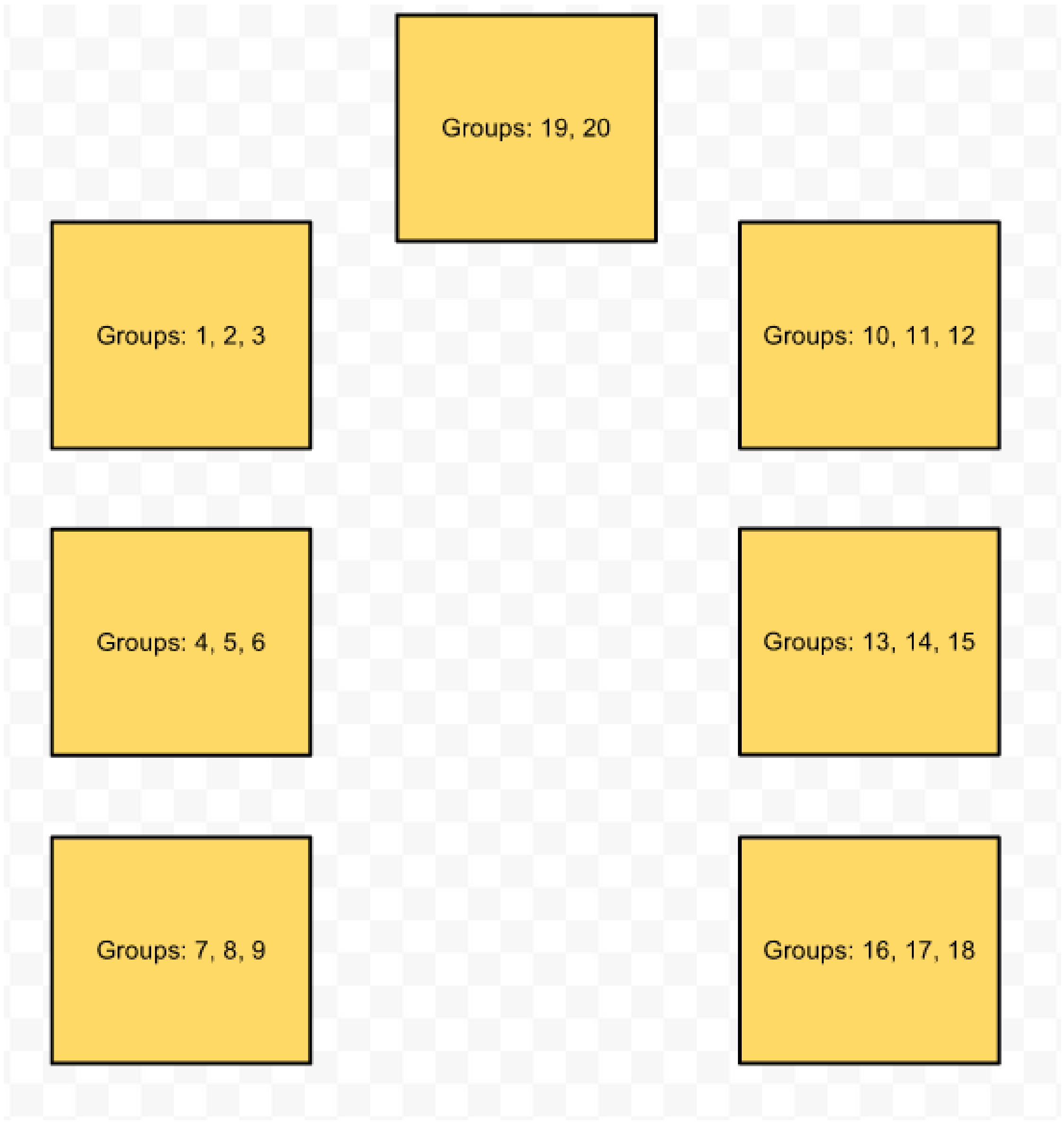


21.2 Pairs

Today's in-class activities will have assigned groups:

Group #		
1	Eric M	Michael M
2	Phil G	Marco B
3	Mike M	Mbozu L
4	Michelle T	Anthony C
5	Jacob W	Alex L
6	Jacob R	Adam P
7	Jon B	Ryan C
8	Ryan T	Adam K
9	Michelle V	Connor G
10	Nikhil A	Wes W
11	Jesus R	Austin H
12	Chris E	Eric Z
13	Marshal N	Anthony B
14	Jieping T	David J
15	James C	David W
16	Matt S	Chris E
17	Hassan A	Madalyn P
18	Yevgeny K	Aaron V
19	Garrett S	Connor G
20	Daniel S	

Groups will meet at the following tables:



21.3 WSGI

We're going to get started with a real, live Web server, built using internal libraries from Python.

21.3.1 Running the Web server

Log into arctic.cse.msu.edu, and clone the cse491-webz repository from github/ctb:

```
% git clone https://github.com/ctb/cse491-webz.git
```

This repo contains a simple Web server built using the wsgiref module, with a Web application that follows the WSGI specification. Basically, wsgiref handles all of the network stuff, while the WSGI application – here, the 'SimpleApp' class in 'app.py' – follows the WSGI application spec and serves up all the content. We'll be writing both a WSGI app (the drinkz stuff) and a WSGI server in this class.

To run the Web server, do:

```
% cd cse491-webz
% python2.7 app.py
```

and use your browser to go to the URL that is printed out.

Use CTRL-C to exit the app.py Web server.

21.3.2 In-class TODO

In pairs, please

- read through the app.py source code
- answer the below questions and be prepared to discuss them with ctb
- merge your cse491-linkz work (from Day 12) into the cse491-webz repository, and serve the cse491-linkz stuff via cse491-webz.

Questions:

1. Stop the Web server, modify the HTML printed out next to the top page in the Web server ('Visit:' ...), and re-run it. How does your Web browser know how to contact *your* app.py instead of your neighbor's?
2. Note that the only content being returned to your Web browser is sent from app.py. Where is the logic in app.py for what is returned, and what is returned by default from this logic (i.e. if nothing specific is matched, what's the default?)
3. What role does content type play in what is returned? What happens if you return the "wrong" content-type?
4. What variable type is 'data'?
5. What role does 'status' play? What if you return the "wrong" status?
6. What happens if 'somefile.html' isn't present?
7. How does the Web browser know that 'somefile.html' is HTML, and that the GIF file content is an image?
8. Why do we generate a random number at the bottom of the script? What happens if you make 'port' a fixed number?
9. How is PATH_INFO generated, ultimately? (What do you have to change in the Web browser to change PATH_INFO?)
10. What happens when you 'print' something from within app.py – where does it print out, and why?
11. When does __call__ in app.py get executed, and how does the server know when to execute it?

21.4 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

CHAPTER 22

Stories

`use-cases`

These documents are all written in reStructuredText; see a quickstart primer.

All stories:

22.1 Story: Recipes, round 1

Summary:

We want to support storing mixed-drink recipes and integrating them with our inventory to figure out what we need to buy to make a particular mixed drink.

We also want to be able to ask, given an inventory, what drink recipes could be made.

Group: Titus B., Cait P.

22.1.1 Stories

A. Timmy is hosting a party, and has several favorite recipes for mixed drinks. He wants to know what he's missing if he wants to make a given number and type of mixed drinks, so that he can go buy it.

B. Suzy is hosting a party, and has several favorite recipes for mixed drinks. She wants to know which mixed drinks (and how many of each) she can make with the current contents of her liquor cabinet.

22.1.2 Features

1. Store named recipes.
2. Given a recipe, find out what we don't have in inventory.
3. Given an inventory and a list of recipes, find out which recipes we can make.

22.1.3 Data storage functionality

Recipes will consist of lists of ingredients (liquor type, amount). They will also have a free text name.

We will need to be able to add them individually.

We will need bulk input functionality, to load multiple recipes.

We will need to be able to retrieve recipes (name & ingredients) individually.

22.1.4 More complex queries

We will want to be able to get all recipes that can be made with a given set of ingredients.

22.2 Recipes

Premise: we want to support storing mixed-drink recipes and integrating them with our inventory to figure out what we need to buy.

Story: Recipes, round 1

CHAPTER 23

Day 12 – Th, Feb 14th, 2013

Rough schedule for today, summary:

- review classes in Python, below
- read about refactoring, below
- read and work through HTML example, below (optional)
- read hw3 carefully and ask questions!
- work on examples and HW; ask questions; run about, scream and shout
- minute cards

23.1 Classes and objects in Python

Classes can be defined like so:

```
class MyClass(object):
   def __init__(self, ...):
      ...

   def method1(self, ...):
      ...
```

and then instantiated like so:

```
x = MyClass(...) # <-- __init__ arguments; no self needed.
```

Instantiating a class creates an empty object and then runs __init__ on it. You call methods on objects like so:

```
x.method1(...)       # no self needed; it's passed in explicitly.
```

Note that the method1 call, above, is identical to:

```
MyClass.method1(x, ...)
```

where ‘x’ must be an object of type MyClass.

Objects are, in essence, namespaces with behavior: that is, they contain other objects (self.x, self.y, self.z, etc.) and behavior on them, in the form of methods. See cse491-numberz/series_iter/series.py for a really simple example of this.

See tutorialspoint for more info, or use the Google.

23.2 Refactoring

Refactoring is a really valuable buzzword to know.

Refactoring is the art of making code nicer (more modular, better organized, etc.) *without* changing functionality. Most students think it's a waste of time because they only work on one homework at a time; they're wrong. If you consistently refactor your code to be cleaner, more modular, and better tested, then over time your underlying project will get much easier to modify and extend without fear.

Automated tests are a really valuable component of this.

In fact, one of the most important uses of tests is to support refactoring – when refactoring, you should only *add* tests to test newly factored out bits of functionality, and never change existing tests or remove them (unless you remove a function, but then you should refactor the test so it still tests the same underlying functionality).

For an example of refactoring, see: `day12-refactoring`

23.3 Basis HTML output and linking - discussion and exercise

The 10-second introduction to HTML: HTML, which is how most Web pages are formatted, consists of display instructions inside of angle brackets, e.g.

> <b>Make this text bold</b>, and leave this text not bold.

If you put that in a file named 'bold.html' and opened it in a Web browser, you would see the first bit bold and the second bit not bold. The tags <b> and </b> would not be visible. If you forget to "close" the <b> tag with </b>, then all of the text would be bold.

Let's try it out.

Go grab github.com/ctb/cse491-linkz, clone it, and run 'python make-html.py'. Now open html/index.html in a Web browser (see below for instructions on working remotely on the CSE cluster) and click around.

Now, go into make-html.py and read through it, and then:

1. Create a new file 'link.html' that contains 'Hello, world' in bold.
2. At the bottom of table.html, add a link to ../index.html to go back "up" in the site hierarchy. Note the connection between the directory structure and URLs.
3. Fix the output of 'catastrophe.html' by closing the h3 tags correctly, and then link catastrophe.html into the bottom of index.html as 'check out the catastrophe!'

Could people post HTML tutorials that they found useful, please? Either in the comments on this page, or to the mailing list?

23.3.1 Posting Web pages on your CSE account

If you have a CSE account and are working remotely on the CSE machines, something like the following should work to post HTML pages at www.cse.msu.edu/~username. (Also read: http://www.cse.msu.edu/facility/howto/mywebpage/)

```
python make-html.py
mkdir ~/web
chmod a+rx ~/web

rm -fr ~/web/cse491-linkz
```

```
cp -r html ~/web/cse491-linkz
chmod -R a+rx ~/web/cse491-linkz
```

You can then update this with your HTML diretory by doing:

```
rm -fr ~/web/cse491-linkz
cp -r html ~/web/cse491-linkz
chmod -R a+rx ~/web/cse491-linkz
```

23.4 Homework 3 - readme

Read through Homework #3; note that there are several branches to merge for various bits of the homework. You can do this all at once (recommended) or do it as you implement the functionality.

Especially check out the test format I used on the hw3-recipe-tests branch in test_recipes.py, and read about test fixtures to understand the setUp and tearDown functionality in hw3-recipe-tests.

23.4.1 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

CHAPTER 24

Day 11 – Tu, Feb 12th, 2013

Rough schedule for today, summary:

- brief discussion of group work
- work on HW; ask questions; etc.
- minute cards

24.1 Group work

The homeworks are only going to get longer and more difficult. Wouldn't it be nice to work on it with others?

For this next homework (and hopefully for the rest), we would like to encourage you to get together with other people, both in and out of class. We expect to see three different kinds of work behavior –

1. Lone wolf. This individual prefers to not work with others, and except for explicit group projects, can do so.
2. Pair programming. Two individuals working through the homework together, but submitting commits separately through github. If only one person is doing the commits, that's fine; just make sure to note that you were working in pairs.
3. Group work. Here, individuals can work independently on the projects, and then merge afterwards. As long as the authors are noted properly on the commits (you should see them with 'git log', in other words) I encourage this mode of work. Please, however, do the merges by yourself.

24.2 Work on the homework

See HW #3 in Homework #3; go to town. Feel free to ask questions.

24.3 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

CHAPTER 25

Day 10 – Th, Feb 7th, 2013

Rough schedule for today, summary:

- Python data structures
- minute cards

25.1 Python data structures

Work individually or in groups to write up code examples in response to the questions on form 1 and then on form 2.

We will go over the answers in class at 3:35, and I will post them here after class.

Update: here are the answers.

Once you're done, go consider Homework #3.

25.1.1 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

CHAPTER 26

Homework #3

Due Feb 20th at 11:59pm.

See Day 10 – Th, Feb 7th, 2013 for HW 3.1, and Day 12 – Th, Feb 14th, 2013 for HW 3.3 and 3.4. Also see Day 11 – Tu, Feb 12th, 2013 for collaboration instructions.

—

1. In your latest cse491-drinkz, reimplement the data structures to be more efficient than lists.

   Specifically, reimplement _bottle_types_db as a Python 'set' containing tuples of (mfg, liquor, typ); and reimplement _inventory_db as a dictionary, keyed by tuples of (mfg, liquor) with the value in the dictionary being the total quantity.

   Note, the function calls in db.py should not change at all, and all of the tests should still pass without any change.

   For reference, you might check out the answers to the day 10 exercises here:

   > http://msu-web-dev.readthedocs.org/en/latest/day10.html

   (15/100 points)

2. I've provided a bunch of tests for cse491-drinkz on the branch 'hw3-tests' on the ctb github repository,

   > https://github.com/ctb/cse491-drinkz

   They are all in the file 'drinkz/tests_by_ctb.py', with some test data. Please *merge* this commit into *your* master branch, and fix your code so that all of the tests pass. Note that you should, in general, *avoid* changing the test code – the tests should be correct ;).

   Note, you can run just the tests in that file by typing:

```
nosetests drinkz/tests_by_ctb.py
```

   A few pointers –

   - change db.get_liquor_amount() to return floating point values indicating ml. (Yes, you'll need to change your own tests in test_drinkz.)
   - make sure 'bin/load-liquor-inventory' is the name of the bulk loading script, and that it takes two parameters: a list of bottle types, and a list of bottle amounts.

   (15/100 points)

3. Implement the first part of recipes, as in Story: Recipes, round 1.

   (a) Make a new file 'recipes.py' that can be imported as 'drinkz.recipes', and that defines a class 'Recipe'.

   The constructor for Recipe should take a name (a string) and a list of ingredient 2-tuples, (liquor type, amount). For example,

```
    Recipe('vodka martini', [('vodka', '6 oz'), ('vermouth', '1 oz')])
```

should create a recipe object for, well, you know!

(b) Define three new functions in db.py, 'add_recipe(r)', 'get_recipe(name)', and 'get_all_recipes()'. Decide if you should use a list, a dictionary, or a set to store 'em; implement it that way; and justify your decision in correctly-spelled and mostly grammatical English in the docstring (the “”“information”“” at the top of the file) in db.py.

Hint: don't use a list, Michelle!

(c) Define a method on the Recipe class, 'r.need_ingredients()', that returns a list of what is *missing* in order to make this recipe. The list of what is missing should be a list of 2-tuples, [('vodka', amount_in_ml), ...]; if it's empty, that means that you have everything you need.

See the tests in 'drinkz/test_recipes.py' on branch 'hw3-recipe-tests' for a few useful details.

Please be sure to merge ctb branch 'hw3-recipe-tests' and check that *all* of your tests pass.

Strong suggestion – make a new function, say, 'convert_to_ml', that takes a string amount and converts it to ml; then factor out common functionality in db.py and recipes.py. Extra kudos for writing tests just for that function, 'cause you know you should.

Another strong suggestion – write a new function in db.py, say, 'check_inventory_for_type', that checks to see if you have a generic type (like 'blended scotch') and if so returns a list or set of mfg/liquor tuples. You'll find it handy for (c).

(50/100 points)

4. Write an HTML output script named 'make-html.py' in the top-level directory (above 'drinkz' and 'bin').

This script should create a directory 'html', and four files within it: 'index.html', 'recipes.html', 'inventory.html', and 'liquor_types.html'. They should be created by the script *from the information in your database* using the db.py functions!

The index file should link to the other three files.

The liquor_types file should contain a list, in whatever form you want (ul, ol, table, etc.), of all of the liquor types in your database.

The inventory file should contain a list of all of the liquor *amounts* in your database.

The recipes file should contain a list of all of the recipes by name, with an entry next to each recipe that indicates whether or not you have all of the ingredients for that recipe (just a “yes” or a “no” (or a “heck yeah”, Wes) is fine).

Each of the files should link to the other three files with an a href tag, i.e. a clickable link.

Be sure to populate your database with at least two examples of recipes, liquor types, and liquor inventory, and make sure that your output . Check out test_recipes for inspiration – you can just copy the data in their for your examples.

Finally, make the output HTML files available in a directory on the CSE cluster under your account, as in Day 12 – Th, Feb 14th, 2013.

(20/100 points)

5. Finish up the HW, and tag it as 'hw3' by doing

```
git tag hw3
git push origin hw3:hw3
```

Then fill out this form.

(As usual, make sure all of your tests pass.)

Day 9 – Tu, Feb 5th, 2013

Rough schedule for today, summary:

- use case presentations
- use case groups: brainstorming => storyboard
- use case groups: storyboard => features, data storage functionality,
- write up and add to the msu-cse491-2013 Web site
- minute cards

27.1 Use case stories

The end goal of today is to have six sets of functionality fleshed out as in Story: Recipes, round 1 (which is one set of functionality...)

Note that the Web site is on github: https://github.com/ctb/msu-cse491-2013/

27.2 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

CHAPTER 28

Homework #2

Due Feb 6th at 11:59pm. If you work in a group, please work in a group distinct from those you work with in the class.

—

0. Clone my cse491-drinkz repository into your own directory tree and set up your own github repo as origin, and mine as ‘ctb’:

```
git clone https://github.com/ctb/cse491-drinkz.git
git remote rm origin
git remote add origin https://github.com/YOURUSERNAME/cse491-drinkz.git
git remote add ctb https://github.com/ctb/cse491-drinkz.git
```

   (replace YOURUSERNAME with, well, your user name :)

   Next, remove your master branch and replace it with mine:

```
git push origin :master  # this deletes your master branch
git push origin master:master  # this replaces it with mine
```

   (After all of this, ‘git diff origin/master’ should return no differences. If you’re not sure, check with someone!)

1. Using load_bulk_data.load_bottle_types as inspiration, modify the bulk loading functions to ignore empty lines (note: ‘if not line.strip()’) and ‘#’-commented lines. Write tests for this functionality to make sure it works; the tests should check each piece of functionality separately.

2. Next, write a generic generator wrapper around the ‘csv.reader’ that eliminates commented lines and whitespace lines. You should have one function that can be used by all the bulk loading functions; for example,

      new_reader = data_reader(fp)

      for mfg, name, typ in new_reader: ...

   should now work. Make sure that all your tests still pass!

3. Implement try/except wrappers around each individual line so that malformed lines in the bulk loading functions simply print out an error rather than failing. (See ‘test_add_to_inventory_2’ in test_drinkz.py for the basic formatting of exception handling.)

4. Fix get_liquor_amount to correctly sum different types of liquor in oz and ml, and report in ml. Write tests to make sure it works properly!

5. Add a script in the main directory called ‘show-liquor-amounts’ that reports the amounts of liquor that you have, as well as the types. (You can base it off the script ‘show-liquor-types’.) Don’t bother writing any tests for this one.

6. Add a script under bin/ called load-liquor-inventory that runs the bulk loading function on a text file; model it on load-liquor-types. Write tests for it, based on the tests for load-liquor-inventory.

7. Push all your homework to your github repo and send me a pull request.

CHAPTER 29

Day 8 – Th, Jan 31, 2013

Rough schedule for today, summary:

- reviewing git & taking questions
- version control principles
- back to drinkzing
- use cases!!
- minute cards

Note: mailing list archive is at http://lists.idyll.org/pipermail/cse491-spring-2013/

29.1 Reviewing git and taking questions

See Day 6 – Th, Jan 24, 2013, “Resolving conflicts during merge”, and Day 7 – Tu, Jan 29, 2013.

At this point, I’m assuming wrt git –

- you know enough git to handle many things
- you can work through most problems
- you’ll ask questions if you need help

Also see the various e-mails recommending git tutorials - here, here, and here.

29.2 Some version control principles

1. Commit often

Divide your work up into small discrete chunks, and every time you complete a chunk, commit. This is simply good practice.

In collaborative work, remember: the last person to merge has the most work to do, so if you’re working collaboratively, try to get complete “chunks” of work (new features) committed at a high level of granularity, so that other people have to do the work of merging.

2. Only add/commit files that aren’t automatically generated.

For example, Python compiles .py files into .pyc files; don’t commit those to your repository! (Also see .gitignore, in the root directory; this will tell git which files it should ignore when you do ‘git status’.)

3. Don't commit any unnecessary files or changes.

I use 'git status' and 'git diff' liberally to make sure that only intentional changes get committed. This should not include "whitespace" changes, commented out code, etc.

4. Run your tests before committing, and again before pushing.

Never commit/communicate code that you *know* is broken, unless you have a reason to do so.

29.3 New functionality in drinkz

What do the merges you merged last week actually do?

29.4 Use cases!!!

Check out the README for drinkz:

> https://github.com/ctb/cse491-drinkz/blob/master/README.md

To guide development of new features, software developers often design what *use cases*. These are, at a high level, little stories about what users or customers want to do. For example,

Brian has a liquor cabinet and a bunch of drink recipes that use various types and amounts of liquor. He wants the 'drinkz' site to output a shopping list for him based on a set of drink recipes he selects.

This use case drives a number of design considerations for the site database:

1. The site has to be able to store Brian's drink inventory.
2. The site has to be able to store Brian's recipe inventory.
3. The site has to be able to correlate Brian's drink inventory with the recipe.

It also drives a number of user interface considerations, but we'll get to those in the new weeks.

29.4.1 Brainstorming

Individually or in a group, brainstorm a bunch of use cases that center around activies based on one or more liquor cabinets, and one or more recipes. Remember that "customers" in this case can also include business-to-business customers like markets that want to advertise on your site, offer discounts, etc. Any way that you can make a buck ;). (Ignore all privacy and ethical concerns for now.)

Prepare to present three use cases as a table. They should be different from the use cases presented by the other tables ;).

29.4.2 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

CHAPTER 30

Day 7 – Tu, Jan 29, 2013

Rough schedule for today, summary:

- git branches and repositories
- use cases and brainstorming
- minute cards

30.1 Git - setting up

First, clone a new copy of the cse491-drinkz repository from github. You can either delete your existing copy (which you won't need for anything in particular) and create a new one under the same name:

```
rm -fr cse491-drinkz
git clone https://github.com/ctb/cse491-drinkz.git
```

OR you can clone a copy into a new directory,

```
rm -fr cse491-drinkz
git clone https://github.com/ctb/cse491-drinkz.git new-drinkz
```

(There are ways to clean up an existing repository, but in some cases it's easier to just trash it and start fresh, if there's nothing precious in it!)

All of the next commands will take place in this directory, so change into it:

```
cd new-drinkz
```

30.2 Branches in git

Repositories – what you create with 'clone' – contain one or more *branches* of source code.

Branches are essentially named collections of commits, which in turn are a set of changes to one or more files (including creating and deleting them).

Each branch will have multiple commits, and you can add commits to a branch by either making changes and committing them yourself, OR by merging in commits from other branches.

Your repository will generally start with one branch, 'master'. This is the default. Check it out:

```
git branch
```

The ‘*’ next to master says this is the current branch. You can also use ‘git status’ to see your current branch:

```
git status
```

this should tell you that you have no changes in your repository.

If you type ‘git log’, you’ll see the list of commits in your current branch:

```
git log
```

The loooong commit number after the word ‘commit’ uniquely identifies this commit within your repository, and generally within all repositories for the project. (Obviously this can’t be guaranteed across *all* repositories, as there is no central naming authority for commits.)

30.3 Fetching new branches

You can generally divide git commands into things that work on local branches – branches that are local to your repository – and things that communicate between repositories, generally by communicating branches.

Commands like ‘commit’, ‘log’, ‘add’, and ‘merge’ work on local branches.

Commands like ‘clone’, ‘fetch’, and ‘push’ copy branches between repositories.

(Commands like ‘pull’ combine commands for convenience sake – pull, for example, is a combined fetch+merge.)

If you want to fetch a branch from another repository, you use a command like this:

```
git fetch <repository address> <remote branch>:<local branch>
```

So, for example, you can run:

```
git fetch https://github.com/ctb/cse491-drinkz.git script:script
```

to grab the set of changes associated with the ‘script’ commit in my github repository and copy it the local branch named ‘script’, which you can then see with ‘git branch’:

```
git branch
```

To switch to the ‘script’ branch, type ‘git checkout script’:

```
git checkout script
```

And voila, all of your files will be switched over to the contents of the ‘script’ branch. And you can type:

```
git checkout master
```

to switch back to master. In both cases you should see ‘git branch’ switch your default branch (the one with the ‘*’* next to it).

One **very important note** is that ‘git checkout’ will refuse to delete or overwrite files that it doesn’t know about or have saved. It’s a pretty safe command, therefore, unless you say ‘git checkout -f’ in which case it will feel free to forcibly (-f!) wipe out uncommitted changes.

Note that ‘git fetch’ and ‘git checkout’ can both be run multiple times in a row with no ill effects.

If you want to compare the branch you’re on with another branch, type ‘git diff <branchname>’:

```
git checkout master
git diff script
```

The '+' are lines that need to be added to the *other* branch to make it look like *this* branch, and the '-' are lines that need to be added to *this* branch to make it look like the *other* branch. In this case, there should only be '-' lines – but you can see the other side of things by doing

```
git checkout script
git diff master
```

and now the only differences should be lines starting with '+'.

If you want to merge, you can do it in two ways – you can go to 'script' and merge from master:

```
git checkout script
git merge master
```

and, because 'master' is an ancestor of 'script', you will see "Already up-to-date". git merging only *adds* commits, so since 'script' contains all of the commits present in master (and then some) nothing happens.

You can also go the other way:

```
git checkout master
```

Here, let's not "contaminate" master just yet with the commits in 'script'; instead of merging directly into master, do a trial merge into a new branch, which we'll call 'merge_script'

```
git checkout -b merge_script
git merge script
```

Here the '-b' says, 'take my current branch and make a new copy, called merge_script'. And then 'merge', of course, merges in the changes from the *other* branch. When you do this merge, you should see the message 'Fast-forward', because 'script' is a descendant of 'master' and so there's no real "merging" being done – you're just updating the commit that the name master points at to be the same as the commit that 'script' points at. As we saw on Thursday, things only get tricky when merges are between branches with conflicting commits.

Side note: the ease of creating new branches in 'git' is one of its most powerful attributes, because it lets you easily name and save a set of changes.

30.4 In-class work

Three different workers have added three different features (with tests!) to drinkz. They are located in the following repositories and branches:

https://github.com/ctb/cse491-drinkz.git, amounts

https://github.com/picken19/cse491-drinkz.git, output_types

https://github.com/picken19/cse491-drinkz.git, script

Your mission, should you choose to accept it, is:

1. Fetch the branches so they're all local to your repository.
2. Figure out what the set of changes in each branch does by using 'diff' and 'log'.
3. Merge all three branches together, and make sure all the tests work, by running 'nosetests'.

   (There should be 10, I think.)
4. Push back to your master branch on YOUR repository.

30.5 When you're done with that: write/brainstorm *use cases*

Check out the README for drinkz:

https://github.com/ctb/cse491-drinkz/blob/master/README.md

To guide development of new features, software developers often design what *use cases*. These are, at a high level, little stories about what users or customers want to do. For example,

Brian has a liquor cabinet and a bunch of drink recipes that use various types and amounts of liquor. He wants the 'drinkz' site to output a shopping list for him based on a set of drink recipes he selects.

Individually or in a group, brainstorm a bunch of use cases that center around activies based on one or more liquor cabinets, and one or more recipes. Remember that "customers" in this case can also include business-to-business customers like markets that want to advertise on your site, offer discounts, etc. Any way that you can make a buck ;). (Ignore all privacy and ethical concerns for now.)

Write the uses cases down somewhere – you won't have to hand them in but I'd like to get a class-wide list.

30.5.1 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

Day 6 – Th, Jan 24, 2013

Rough schedule for today, summary:

- testing, code paths, and branching paths
- in class exercise on testing
- git: resolving conflicts
- in class exercise on resolving conflicts
- minute cards (3:50)

Join the online chat for Q&A at: https://www.hipchat.com/gpAMmlQ4v

31.1 Testing and code paths

Suppose I give you a function:

```
def fut(a, b):
   "Return 0 if a and b are equal, 1 if a is bigger, and -1 if b is bigger."
   if a == b:
      return 0
   elif a > b:
      return 1
   else:
      return -1
```

How many different combinations of a and b should you pass into make sure that every line of code in this function is tested?

Here's another:

```
def corrected_fraction(a, b):
   "returns a/(b - 1) if b > 0, a/b otherwise."
   a = float(a)
   b = float(b)

   if b > 0:
      b -= 1

   return a/b
```

Here's a third:

```
def slow_mod(a, b):
   "returns a mod b"
   while a > b:
      a -= b

   return a
```

What conditions should you feed into these functions in order to test them thoroughly? (Ignore out of bounds errors, please, like 'this number is too big', etc.)

Answer here, please.

31.2 git: Resolving conflicts during merge

The main thing I need to show you all before we move forward with all the exciting features that git has to offer is the question of how to resolve conflicts.

One of the primary purposes of version control is *collaboration* – it's supposed to help multiple people work on the same project without colliding. This is easy (at a technical level) when the people are working on different files or different lines of different files. But what about when their changes collide, e.g. Emily makes a change to the same line as Beth?

In that case, git forces you to *manually* resolve things.

Let's take a look at the cse491-ex-rezolve repository on github:

https://github.com/ctb/cse491-ex-rezolve

This contains a bunch of text quotes that have been edited by a mad professor and need to be fixed. Specifically, the mad professor made everything lowercase and replaced all of the 'e's with 'z's. Your job will be to fix one of these problems (say, lowercase), for one of the quotes files, and then merge your fixes with someone else's fixes for the OTHER problem (say, e->z).

Let's take a look at example-quote.txt:

```
no synonym for god is so pzrfzct as bzauty. whzthzr as szzn carving thz linzs
of thz mountains with glacizrs, or gathzring mattzr into stars, or planning thz
movzmznts of watzr, or gardzning - still all is bzauty!

...
THIS IS A BIG BLOCK OF STUFF THAT DOESN'T CHANGE
...

zvzry crzator painfully zxpzriznczs thz chasm bztwzzn his innzr vision and its ultimatz zxprzssion.
```

Suppose someone comes in and fixes all of the e->z problems, and pushes them to their repository on github (see, for example, this file on the ze_fix branch: https://github.com/ctb/cse491-ex-rezolve/blob/ze_fix/example-quote.txt). And you come along and fix all of the capitalization problems, and push them to a *different* repository on github (see this file on the cap_fix branch: https://github.com/ctb/cse491-ex-rezolve/blob/cap_fix/example-quote.txt). There's no way for the computer to know how to merge these branches.... But you want to merge them!

First, what are the mechanics of merging?

Start by forking a copy of my repository,

https://github.com/ctb/cse491-ex-rezolve.git

and then cloning it:

```
git clone https://github.com/USERNAME/cse491-ex-rezolve.git
```

If you look at example-quote.txt, you'll see the completely unfixed version. Let's pull in the fixes from both branches in my repository:

```
cd cse491-ex-rezolve
git fetch https://github.com/ctb/cse491-ex-rezolve.git cap_fix:cap_fix
git fetch https://github.com/ctb/cse491-ex-rezolve.git ze_fix:ze_fix
```

Check out the differences between them and the 'master' branch, which is the default one when you clone a repository:

```
git diff cap_fix
```

and

> git diff ze_fix

(The '+' at the beginning of the line means this line has been changed.)

Now let's try merging in the ze_fix:

```
git merge ze_fix
```

You should see something like this:

```
Updating 30d05e4..153c8d9
Fast-forward
 example-quote.txt |    8 ++++----
 1 files changed, 4 insertions(+), 4 deletions(-)
```

and you'll see that all the 'z's in example-quote.txt have been fixed. Wait – how did it merge this without asking us anything!?

Well, git knows that the z->e changes are safe to apply to the master branch, because it kept track of me making those changes ;). (Edit, commit, basically.) git *also* knows that it's safe to apply the capitalization fixes directly to master, too. What git does *not* know is how to *combine* the two.

So... now what?

Try merging in the cap_fix branch also:

```
setenv EDITOR nano
git merge cap_fix
```

Uh-oh!

```
Auto-merging example-quote.txt
CONFLICT (content): Merge conflict in example-quote.txt
Automatic merge failed; fix conflicts and then commit the result.
```

'git status' will tell you that the problem is 'example-quote.txt'. What does this file look like?

You should see blocks like this:

```
<<<<<<< HEAD
no synonym for god is so perfect as beauty. whether as seen carving the lines
of the mountains with glaciers, or gathering matter into stars, or planning the
movements of water, or gardening - still all is beauty!
=======
No synonym for god is so pzrfzct as bzauty. Whzthzr as szzn carving thz linzs
of thz mountains with glacizrs, or gathzring mattzr into stars, or planning thz
movzmznts of watzr, or gardzning - still all is bzauty!
>>>>>>> cap_fix
```

This is git's way of saying, "everything between <<<< and === is from the current branch, everything from === to >>> is from the branch you're telling me to merge, and I can't resolve it. HELP."

To fix this, just merge them manually; make it look like:

```
No synonym for god is so perfect as beauty. Whether as seen carving the lines
of the mountains with glaciers, or gathering matter into stars, or planning the
movements of water, or gardening - still all is beauty!
```

and delete all the >>> === and <<< stuff. Then save, and at the command line say:

```
git add example-quote.txt
git commit -am "merged ze fix and cap fix"
```

The first line is an explicit direction to git that you fixed the merge problems in that file; the second line tells git that you're all done.

31.3 Sharing branches with others

So, let's say I tell you to EITHER fix the capitals OR the 'z's in one of the quotes files, and that I want you to share these changes via github. How do you do this?

First, make the changes.

Then, commit them with 'git commit -am "commit message"'.

Then *push* them to your github repo:

```
git push origin master:remote_branch_name
```

You can now *fetch* other people's branches like so:

```
git fetch <their github URL> remote_branch_name:local_branch_name
```

and use 'git merge' as above.

So I would like you to fix one or the other of the capitals and the z/es in one of the quotes (as per handed-out directions), push them to github as either 'ze_fix' or 'cap_fix' branches (see: remote_branch_name, above), go find a compadre who has worked on the *opposite* problem, and then merge their changes (and have them merge your changes), and then push the merge to your master branch.

31.3.1 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

CHAPTER 32

Day 5 – Tu, Jan 22, 2013

Rough schedule for today, summary:

- announcement: syllabus updated; office hours Tuesday, 6-8pm, 2228 BPS
- in-class project
- minute cards (3:45)
- pre-survey

Join the online chat for Q&A at: https://www.hipchat.com/gpAMmlQ4v

32.1 In-class Project: drinkz testing

On the CSE cluster, do the following:

```
cd
python2.7 -m virtualenv env491
source env491/bin/activate.csh
easy_install nose
```

This creates a 'virtual' installation of Python 2.7 in the directory 'env491' that is under your own control – see http://pypi.python.org/pypi/virtualenv – and then activates it so that it is the first Python install in your PATH. Then 'easy_install nose' installs the nose testing framework under that environment.

If you're running this on your own computer, you can skip the virtualenv and activate steps, and just 'easy_install nose' as superuser. Or you can install virtualenv as superuser, and following everything as normal.

Next, go to https://github.com/ctb/cse491-drinkz and fork this repository into your own github account. On the CSE cluster, do

```
git clone https://github.com/FAKEUSERNAME/cse491-drinkz.git
```

but replace 'FAKEUSERNAME' with your own github username. If you do:

```
cd cse491-drinkz
ls -R
```

you should see the following files:

```
.:
README.md  drinkz
```

```
./drinkz:
__init__.py  db.py  load_bulk_data.py  test_drinkz.py
```

Use ‘less’ to read the README, and

```
nosetests -v
```

to run the tests.

Your mission: fix the tests by implementing the various functions, then commit your changes and push to your github repo.

Once you have fixed all the tests, to commit and push do:

```
git commit -am "fixed tests"
git push origin master
```

Note that you can fix one test at a time (and then commit and push), and share work, and thereby work in parallel on different tests and functions; if you want to do this talk to me and I’ll show you how to pull from other people’s repositories and merge. We’ll be discussing this on Thursday anyway, however.

32.1.1 Pre Survey

If you finish the in-class project, please take this pre-survey about your experience with topics we will be studying in the course.

32.1.2 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

CHAPTER 33

Homework #1

Due January 24th at 11:59pm. If you work in a group, please work in a group distinct from those you work with in the class.

—

0. Fork my cse491-numberz repository into your own github account, and clone it into your account. Base the homework off of this.
1. Look at my implementation of Eratosthenes' Sieve, sieve-fn.py, and implement this in module, iterator, and generator form (as you saw above with series_mod, series_iter, and series_gen, etc.). Add, commit, and push back to your github repository.

   Use the same naming conventions for your implementation as we did for 'series' and 'fib' so we know where to look for your solutions.

   Hint: if it is not viewable through the github Web interface, you did not successfully add/commit/push!

   Warning: code with syntax errors => 0 points.

   Collect bad and/or buggy examples of code and submit them at the end (see form in #5, below).
2. Write at least six tests (three each) as in iter_bug/test.py for your iterator and generator implementations of the Sieve, and then add/commit/push them to your github repo. These should be individual scripts named 'test1.py' etc. that are standalone, can be executed by the command line, and test something about your implementation in 'sieve.py'. The tests *can* be similar (or even the same) for iterator and generators, but should be checked into each directory.
3. Hand in your homework by filling in this form:

   https://docs.google.com/spreadsheet/viewform?formkey=dHl3c1REa1V3ZHFGNTRmMUYtR3RJU2c6MQ

   You can fill it in at any time, but I can't grade your homework if I don't have this information :)
4. Enter a pull request from your homework in your cse491-numberz to mine.
5. Please fill out this form, too:

   https://docs.google.com/spreadsheet/viewform?formkey=dHo3R3M3Z3Z4RGZ3dThqMmluaDlncUE6MQ

(Note, the last question was added late. So if you've already done the homework, don't worry about filling it in.)

Day 4 – Th, Jan 17, 2013

Rough schedule for today, summary:

- fill in office hours doodle
- look at source code for exercise #2 from Tuesday
- answer quizlet questions (2:50)
- discuss more broadly
- do exercise #3, below (debugging, git add, git push) (3:15)
- discussion (3:45)
- minute cards (3:55)

Join the online chat for Q&A at: https://www.hipchat.com/gpAMmlQ4v

34.1 Office hours

Please go here:

http://whenisgood.net/82igkrb

and fill in your availability/interest for office/TA hours.

34.2 Today!

Look at the source code for exercise #2 from Day 3 – Tu, Jan 15, 2013, and prepare to answer some questions about it.

Quizlet

34.2.1 Iterators

The example in series_iter.py has the following code:

```
for i in series.adder():
   ...
```

This code expands to the following set of calls:

```
x = series.adder()
y = iter(x)
while 1:
   i = y.next()
   ...
```

Which expands or simplifies to this:

```
x = series.adder()
y = x.__iter()__
while 1:
   i = y.next()
```

which in turn simplifies to:

```
x = series.adder()
while 1:
   y = x.next()
```

You can read up online about other ways that iterators can be manipulated, stopped, understood, etc; see, for example, StackOverflow or the Python docs themselves.

34.2.2 Exercise #3

In your 'cse491-numberz' directory (the one you cloned from my github URL, that contains all the source?) execute:

```
git pull origin master
```

This will retrieve updates and new files. You should now have a directory 'iter_bug'.

Try running it:

```
cd iter_bug
python2.7 test.py
```

Make 'iter_bug/test.py' work by editing both test.py and fib.py. Bear in mind that by "work" I mean that 'fib.fib()' should produce an accurate and correct Fibonacci series, and all of the code in test.py should run properly when 'test.py' is executed from the command line.

Once you have everything working, commit the changes:

```
git commit -am "fixed the bug"
```

Next, push them to github.com. Specifically, go to github.com, log in, and go to

> https://github.com/ctb/cse491-numberz

Click the 'fork' button to fork this project into your own github account. Then, select the 'https' URL from the middle of the page of your own copy of the cse491-numberz project – it should look something like this:

```
https://github.com/YOUR_USERNAME/cse491-numberz.git
```

and, in your *local* copy of the repository, do:

```
git remote rm origin
git remote add origin https://github.com/YOUR_USERNAME/cse491-numberz.git
git remote add ctb https://github.com/ctb/cse491-numberz.git
```

And, finally, do:

```
git push origin master
```

You should now see the bugfix changes to your LOCAL files (on CSE or wherever) up on the github Web site under your account.

...and this is how you will hand in homework :).

34.2.3 Minute Cards

In the last 5 minutes of class, please fill out this minute card survey.

CHAPTER 35

Day 3 – Tu, Jan 15, 2013

Join the online chat for Q&A at: https://www.hipchat.com/gpAMmlQ4v

---

Go look at https://github.com/ctb/cse491-numberz, either in your Web browser or on

you can check this out on the CSE cluster by doing this:

```
cd ~/
git clone https://github.com/ctb/cse491-numberz.git
```

This will create a directory 'cse491-numberz', into which you should change:

```
cd cse491-numberz
```

35.1 Demo

Run series-fn.py, and mentally or actually annotate each line with a number indicating the order in which each line is run; also indicate when 'n' changes and what the new value is.

35.2 Exercise #1

Run, discuss, and annotate the order of execution and the state changes in variables for series_mod, series_iter, and series_gen amongst yourselves (around the table, or in your own little hermited existence). You can run these like so:

```
cd series_mod
python2.7 example.py
```

... Survey link

35.3 Exercise #2

Run, discuss, and annotate the order of execution and the state changes in variables for fib-fn.py, fib_mod, fib_iter, and fib_gen amongst yourselves (around the table, or in your own little hermited existence).

(At the end of this exercise, we'll give you a link to some questions to answer in singleton or in pairs.)

—

Things to discuss at end of class, if time:

- introducing the Web site
- commenting
- editing and pull requests
- forking things into your own account explicitly (for the HW)

CHAPTER 36

Useful websites

36.1 git/SVN

A useful reference for git commands. If you know SVN you can search by the SVN command to find the git equivalent. If you don't know SVN just search for the normal bash commands to find how to do it in git.

http://git.or.cz/course/svn.html

Explanation of how git works and how it is different from SVN and other version controls

http://git-scm.com/book/en/Getting-Started-Git-Basics

CHAPTER 37

Indices and tables

- genindex
- modindex
- search