Lab 12: SQL **lab12.zip (lab12.zip)**

Due at 11:59pm on Friday, 4/26/2019.

Starter Files

Download lab12.zip (lab12.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

Submission

By the end of this lab, you should have submitted the lab with `python3 ok --submit`. You may submit more than once before the deadline; only the final submission will be graded. Check that you have successfully submitted your code on okpy.org (https://okpy.org/).

- To receive credit for this lab, you must complete Questions 2-4 in lab12.sql (lab12.sql), and submit through OK.
- Questions 5-6 are also considered **extra practice**. They can be found in the lab12_extra.sql (lab12_extra.sql) file. It is recommended that you complete these when you are finished with the required questions.

SQLite

Usage

First, check that a file named `sqlite_shell.py` exists alongside the assignment files. If you don't see it, or if you encounter problems with it, scroll down to the Troubleshooting section to see how to download an official precompiled SQLite binary before proceeding.

You can start an interactive SQLite session in your Terminal or Git Bash with the following command:

```
python3 sqlite3
```

While the interpreter is running, you can type `.help` to see some of the commands you can run.

To exit out of the SQLite interpreter, type `.exit` or `.quit` or press `Ctrl-C`. Remember that if you see `...>` after pressing enter, you probably forgot a `;`.

You can also run all the statements in a `.sql` file by doing the following:

1. Runs your code and then exits SQLite immediately afterwards.

```
python3 sqlite3 < lab12.sql
```

2. Runs your code and then opens an interactive SQLite session, which is similar to running Python code with the interactive `-i` flag.

```
python3 sqlite3 --init lab12.sql
```

Topics

SQL Basics

Creating Tables

You can create SQL tables either from scratch or from existing tables.

The following statement creates a table by specifying column names and values without referencing another table. Each `SELECT` clause specifies the values for one row, and `UNION` is used to join rows together. The `AS` clauses give a name to each column; it need not be repeated in subsequent rows after the first.

```
CREATE TABLE [table_name] AS
  SELECT [val1] AS [column1], [val2] AS [column2], ... UNION
  SELECT [val3]             , [val4]             , ... UNION
  SELECT [val5]             , [val6]             , ...;
```

Let's say we want to make the following table called `big_game` which records the scores for the Big Game each year. This table has three columns: `berkeley`, `stanford`, and `year`.

berkeley	stanford	year
30	7	2002
28	16	2003
17	38	2014

We could do so with the following `CREATE TABLE` statement:

```
CREATE TABLE big_game AS
  SELECT 30 AS berkeley, 7 AS stanford, 2002 AS year UNION
  SELECT 28,             16,            2003         UNION
  SELECT 17,             38,            2014;
```

Selecting From Tables

More commonly, we will create new tables by selecting specific columns that we want from existing tables by using a `SELECT` statement as follows:

```
SELECT [columns] FROM [tables] WHERE [condition] ORDER BY [columns] LIMIT [limit];
```

Let's break down this statement:

- `SELECT [columns]` tells SQL that we want to include the given columns in our output table; `[columns]` is a comma-separated list of column names, and `*` can be used to select all columns
- `FROM [table]` tells SQL that the columns we want to select are from the given table; see the joins section () to see how to select from multiple tables
- `WHERE [condition]` filters the output table by only including rows whose values satisfy the given `[condition]`, a boolean expression
- `ORDER BY [columns]` orders the rows in the output table by the given comma-separated list of columns
- `LIMIT [limit]` limits the number of rows in the output table by the integer `[limit]`

> *Note:* We capitalize SQL keywords purely because of style convention. It makes queries much easier to read, though they will still work if you don't capitalize keywords.

Here are some examples:

Select all of Berkeley's scores from the `big_game` table, but only include scores from years past 2002:

```
sqlite> SELECT berkeley FROM big_game WHERE year > 2002;
28
17
```

Select the scores for both schools in years that Berkeley won:

```
sqlite> SELECT berkeley, stanford FROM big_game WHERE berkeley > stanford;
30|7
28|16
```

Select the years that Stanford scored more than 15 points:

```
sqlite> SELECT year FROM big_game WHERE stanford > 15;
2003
2014
```

SQL operators

Expressions in the `SELECT`, `WHERE`, and `ORDER BY` clauses can contain one or more of the following operators:

- comparison operators: `=`, `>`, `<`, `<=`, `>=`, `<>` or `!=` ("not equal")
- boolean operators: `AND`, `OR`
- arithmetic operators: `+`, `-`, `*`, `/`
- concatenation operator: `||`

Here are some examples:

Output the ratio of Berkeley's score to Stanford's score each year:

```
sqlite> select berkeley * 1.0 / stanford from big_game;
0.447368421052632
1.75
4.28571428571429
```

Output the sum of scores in years where both teams scored over 10 points:

```
sqlite> select berkeley + stanford from big_game where berkeley > 10 and stanford > 10;
55
44
```

Output a table with a single column and single row containing the value "hello world":

```
sqlite> SELECT "hello" || " " || "world";
hello world
```

Joins

To select data from multiple tables, we can use joins. There are many types of joins, but the only one we'll worry about is the inner join. To perform an inner join on two on more tables, simply list them all out in the `FROM` clause of a `SELECT` statement:

```
SELECT [columns] FROM [table1], [table2], ... WHERE [condition] ORDER BY [columns] LIMIT |
```

We can select from multiple different tables or from the same table multiple times.

Let's say we have the following table that contains the names head football coaches at Cal since 2002:

```
CREATE TABLE coaches AS
  SELECT "Jeff Tedford" AS name, 2002 as start, 2012 as end UNION
  SELECT "Sonny Dykes"         , 2013         , 2016        UNION
  SELECT "Justin Wilcox"       , 2017         , null;
```

When we join two or more tables, the default output is a cartesian product (https://en.wikipedia.org/wiki/Cartesian_product). For example, if we joined `big_game` with `coaches`, we'd get the following:

berkeley	stanford	year
30	7	2002
28	16	2003
17	38	2014

name	start	end
Jeff Tedford	2002	2012
Sonny Dykes	2013	2016
Justin Wilcox	2017	null

berkeley	stanford	year	name	start	end
30	7	2002	Jeff Tedford	2002	2012
30	7	2002	Sonny Dykes	2013	2016
30	7	2002	Justin Wilcox	2017	null
28	16	2003	Jeff Tedford	2002	2012
28	16	2003	Sonny Dykes	2013	2016
28	16	2003	Justin Wilcox	2017	null
17	38	2014	Jeff Tedford	2002	2012
17	38	2014	Sonny Dykes	2013	2016
17	38	2014	Justin Wilcox	2017	null

If we want to match up each game with the coach that season, we'd have to compare columns from the two tables in the `WHERE` clause:

```
sqlite> SELECT * FROM big_game, coaches WHERE year >= start AND year <= end;
17|38|2014|Sonny Dykes|2013|2016
28|16|2003|Jeff Tedford|2002|2012
30|7|2002|Jeff Tedford|2002|2012
```

The following query outputs the coach and year for each Big Game win recorded in `big_game`:

```
sqlite> SELECT name, year FROM big_game, coaches
...>        WHERE berkeley > stanford AND year >= start AND year <= end;
Jeff Tedford|2003
Jeff Tedford|2002
```

In the queries above, none of the column names are ambiguous. For example, it is clear that the `name` column comes from the `coaches` table because there isn't a column in the `big_game` table with that name. However, if a column name exists in more than one of the tables being joined, or if we join a table with itself, we must disambiguate the column names using *aliases*.

For examples, let's find out what the score difference is for each team between a game in `big_game` and any previous games. Since each row in this table represents one game, in order to compare two games we must join `big_game` with itself:

```
sqlite> SELECT b.Berkeley - a.Berkeley, b.Stanford - a.Stanford, a.Year, b.Year
...>        FROM big_game AS a, big_game AS b WHERE a.Year < b.Year;
-11|22|2003|2014
-13|21|2002|2014
-2|9|2002|2003
```

In the query above, we give the alias `a` to the first `big_game` table and the alias `b` to the second `big_game` table. We can then reference columns from each table using dot notation with the aliases, e.g. `a.Berkeley`, `a.Stanford`, and `a.Year` to select from the first table.

Checkoff 8

Q1: Wild While-d Macro

In homework, you implemented a list comprehension macro in Scheme. However, what if we wanted to loop with a `while` loop instead of a `for`? Answer the following questions with an AI or a TA to get checked off.

- First, assume we are trying to print `hi` 4 times. What expression would you type on the command line? There could be multiple answers (hint: use `begin`)
- Now recall that a `while` loop has a condition it checks on each iteration. Let's not worry about incrementing variables or restarting the loop for now. How would you incorporate a condition `(< x 4)` into the above expression? (hint: don't overthink this)
- There is a slight problem with this. We are printing `hi` 4 times if `x` is less than 4, so we print `hi` 4 times regardless of whether `x` is 3, 2, 1, 0, -1, -2, or anything smaller! Ignore this problem for now--we will handle it later. Now remember that a `while` loop will usually have some sort of variable incrementation. In this case, let's say `x` increases by 1 on each loop, and that `x` exists in the global frame (i.e., before you enter a loop in Python, you would first set `x = 0`). We can use the `define` special form to help us with this. How might you incorporate `define` into the previous expression to implement incrementing behavior?
- Hmm interesting. Each time we increment our `x`, we check if `(< x 4)`, print, and then increment just as we would in Python. This suggests a recursive structure! We have implemented the recursive behavior for you below (note: it is not too important that you understand how this recursion works--the point of this exercise is just to walk you through a cool use of macros). Fill in the blank to make this `while` macro work, using the above 3 parts, assuming we have defined `x` as 0 in the global frame and called the macro with `(while (< x 4) (print 'hi) (define x (+ x 1)))`.

.

```
scm> (define-macro (while cond body change) `(if __________ (while ,cond ,body ,change)))
while
```

Required Questions

Getting to Know Your Fellow 61A Students

Last week, we asked you and your fellow students to complete a brief online survey through Google Forms, which involved relatively random but fun questions. In this lab, we will interact with the results of the survey by using SQL queries to see if we can find interesting things in the data.

First, take a look at `sp19data.sql` and examine the table defined in it. Note its structure. You will be working with:

- `students` : The main results of the survey. Each column represents a different question from the survey, except for the first column, which is the time of when the result was submitted. This time is a unique identifier for each of the rows in the table.

Column Name	**Question**
`time`	The unique timestamp that identifies the submission
`number`	What's your favorite number between 1 and 100?
`color`	What is your favorite color?
`seven`	Choose the number 7 below. Options: ◦ 7 ◦ You're not the boss of me! ◦ Choose this option instead ◦ seven ◦ the number 7 below.
`song`	If you could listen to only one of these songs for the rest of your life, which would it be? Options: ◦ "Shelter" by Porter Robinson ◦ "Crab Rave" by Noisestorm ◦ "All I want for Christmas is you" by Mariah Carey ◦ "All Star" by Smash Mouth ◦ "Big enough" by Callinan ◦ "Never gonna give you up" by Rick Astley
`date`	Pick a day of the year!
`pet`	If you could have any animal in the world as a pet, what would it be?
`animal`	Choose your favorite photo of a staff member's pet cat or dog! (Options shown under Question 2)
`smallest`	Try to guess the smallest unique positive INTEGER that anyone will put!

- `checkboxes` : The results from the survey in which students could select more than one option from the numbers listed, which ranged from 0 to 10 and included 2018, 9000, and 9001. Each row has a time (which is again a unique identifier) and has the value

`'True'` if the student selected the column or `'False'` if the student did not. The column names in this table are the following strings, referring to each possible number: `'0'`, `'1'`, `'2'`, `'4'`, `'5'`, `'6'`, `'7'`, `'8'`, `'9'`, `'10'`, `'2018'`, `'9000'`, `'9001'`.

Since the survey was anonymous, we used the timestamp that a survey was submitted as a unique identifier. A time in `students` matches up with a time in `checkboxes`. For example, the row in `students` whose `time` value is `"11/9/2018 18:02:33"` matches up with the row in `checkboxes` whose `time` value is `"11/9/2018 18:02:33"`. These entries come from the same Google form submission and thus belong to the same student.

> *Note*: If you are looking for your personal response within the data, you may have noticed that some of your answers are slightly different from what you had input. In order to make SQLite accept our data, and to optimize for as many matches as possible during our joins, we did the following things to clean up the data:
>
> - `color` and `pet`: We converted all the strings to be completely lowercase.
> - For some of the more "free-spirited" responses, we escaped the special characters so that they could be properly parsed.

You will write all of your solutions in the starter file `lab12.sql` provided. As with other labs, you can test your solutions with OK. In addition, you can use either of the following commands:

```
python3 sqlite3 < lab12.sql
python3 sqlite3 --init lab12.sql
```

Q2: What Would SQL print?

> Note: there is no submission for this question

First, load the tables into sqlite3.

```
$ python3 sqlite3 --init lab12.sql
```

Before we start, inspect the schema of the tables that we've created for you:

```
sqlite> .schema
```

This tells you the name of each of our tables and their attributes.

Let's also take a look at some of the entries in our table. There are a lot of entries though, so let's just output the first 20:

```
sqlite> SELECT * FROM students LIMIT 20;
```

If you're curious about some of the answers students put into the Google form, open up `su18data.sql` in your favorite text editor and take a look!

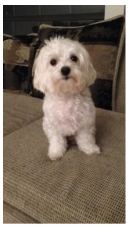
For each of the SQL queries below, think about what the query is looking for, then try running the query yourself and see!

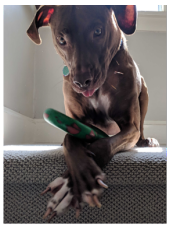
```
sqlite> SELECT * FROM students LIMIT 30; -- This is a comment. * is shorthand for all colu
------

sqlite> SELECT color FROM students WHERE number = 16;
------

sqlite> SELECT song, pet FROM students WHERE color = "blue" AND date = "12/25";
------
```

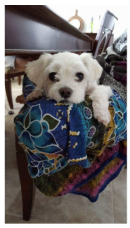
Remember to end each statement with a `;`! To exit out of SQLite, type `.exit` or `.quit` or hit `Ctrl-C`.

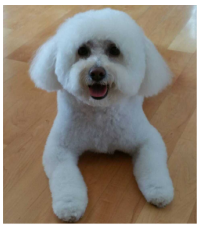
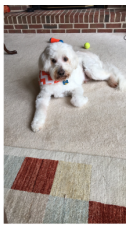
Q3: Obedience

To warm-up, let's ask a simple question related to our data: Is there a correlation between whether students do as they're told and their favorite images of the staff's animal's?


1


2

3

4

5

6

7

8

Write an SQL query to create a table that contains the columns `seven` (this column representing "obedience") and `animal` (the image of an animal students selected) from the `students` table.

You should get the following output:

```
sqlite> SELECT * FROM obedience LIMIT 10;
7|9 (Jacqueline Tiffany Yeung)
the number 7 below.|7 (Kevin Yu)
7|3 (Tim Foster)
7|5 (Chae Park)
seven|4 (Chae Park)
7|2 (Rachel De Jaen)
Choose this option instead|7 (Kevin Yu)
the number 7 below.|7 (Kevin Yu)
7|5 (Chae Park)
the number 7 below.|4 (Chae Park)
```

```
CREATE TABLE obedience AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

Use Ok to test your code:

```
python3 ok -q obedience
```

Q4: The Smallest Unique Positive Integer

Who successfully managed to guess the smallest unique positive integer value? Let's find out!

Unfortunately we have not learned how to do aggregations in SQL, which can help us count the number of times a specific value was selected, just yet. As such, we can only hand inspect our data to determine it. However, an anonymous elf has informed us that the smallest unique positive value is greater than 2!

Write an SQL query to create a table with the columns `time` and `smallest` that we can inspect to determine what the smallest integer value is. In order to make it easier for us to inspect these values, use `WHERE` to restrict the answers to numbers greater than 2, `ORDER BY` to sort the numerical values, and `LIMIT` your result to the first 20 values that are greater than the number 2.

The first 5 lines of your output should look like this:

```
sqlite> SELECT * FROM smallest_int LIMIT 5;
4/19/2019 22:00:36|3
4/19/2019 23:55:08|3
4/20/2019 0:27:37|3
4/20/2019 11:09:57|3
4/20/2019 14:43:04|3
```

```
CREATE TABLE smallest_int AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

Use Ok to test your code:

```
python3 ok -q smallest-int
```

After you've successfully passed the Ok test, take a look at the table `smallest_int` that you just created and manually find the smallest unique integer value! If you're curious how to do this with aggregations, check out Question 7.

To do this, try the following:

```
$ python3 sqlite3 --init lab12.sql
sqlite> SELECT * FROM smallest_int; -- No LIMIT this time!
```

Q5: Matchmaker, Matchmaker

Did you take 61A with the hope of finding your soul mate? Well you're in luck! With all this data in hand, it's easy for us to find your perfect match. If two students want the same pet and have the same taste in music, they are clearly meant to be together! In order to provide some more information for the potential lovebirds to converse about, let's include the favorite colors of the two individuals as well!

In order to match up students, you will have to do a join on the `students` table with itself. When you do a join, SQLite will match every single row with every single other row, so make sure you do not match anyone with themselves, or match any given pair twice!

> **Important Note:** When pairing the first and second person, make sure that the first person responded first (i.e. they have an earlier `time`). This is to ensure your output matches our tests.

> *Hint:* When joining table names where column names are the same, use dot notation to distinguish which columns are from which table: `[table_name].[column name]`. This sometimes may get verbose, so it's stylistically better to give tables an alias using the `AS` keyword. The syntax for this is as follows:
>
> ```
> SELECT <[alias1].[column name1], [alias2].[columnname2]...>
>     FROM <[table_name1] AS [alias1],[table_name2] AS [alias2]...> ...
> ```
>
> The query in the football example from earlier uses this syntax.

Write a SQL query to create a table that has 4 columns:

- The shared preferred `pet` of the couple
- The shared favorite `song` of the couple
- The favorite `color` of the first person
- The favorite `color` of the second person

You should get the following output:

```
sqlite> SELECT * FROM matchmaker LIMIT 10;
wolf|Never gonna give you up|red|blue
dragon|All Star|greem|purple
rabbit|All I want for Christmas|pink|pink
cat|Never gonna give you up|something dark|blue
cat|Never gonna give you up|something dark|artisan color
cat|All I want for Christmas|pink|blue
cat|All I want for Christmas|pink|maroon
cat|All I want for Christmas|pink|black
cat|All I want for Christmas|pink|blue
cat|All I want for Christmas|pink|red
```

```
CREATE TABLE matchmaker AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

Use Ok to test your code:

```
python3 ok -q matchmaker
```

Optional Questions

> The following questions are for **extra practice** -- they can be found in the lab12_extra.sql (lab12_extra.sql) file. It is recommended that you complete these problems, but you do not need to turn them in for credit.

The COUNT Aggregator

> Note: We haven't covered aggregation yet (as of 4/23), but you can come back tomorrow (4/24) and do these problems then, or you can read ahead and try them now!

How many people liked each pet? What is the biggest date chosen this semester? How many obedient people chose Image 1 for the Staff member's animal? Is there a difference between last semester's average favorite number and this semester's?

To answer these types of questions, we can bring in SQL aggregation, which allows us to accumulate values across rows in our SQL database!

In order to perform SQL aggregation, we can group rows in our table by one or more attributes. Once we have groups, we can aggregate over the groups in our table and find things like:

- the maximum value (`MAX`),
- the minimum value (`MIN`),
- the number of rows in the group (`COUNT`),
- the average over all of the values (`AVG`),

and more! `SELECT` statements that use aggregation are usually marked by two things: an aggregate function (`MAX` , `MIN` , `COUNT` , `AVG` , etc.) and a `GROUP BY` clause. `GROUP BY [column(s)]` groups together rows with the same value in each column(s). In this section we will only use `COUNT` , which will count the number of rows in each group, but feel free to check out this link (http://www.sqlcourse2.com/agg_functions.html) for more!

For example, the following query will print out the top 10 favorite numbers with their respective counts:

```
sqlite> SELECT number, COUNT(*) AS count FROM students GROUP BY number
  ORDER BY count DESC LIMIT 10;
69|34
7|24
3|21
1|15
42|14
17|12
27|12
4|11
9|11
21|11
```

This `SELECT` statement first groups all of the rows in our table `students` by `number`. Then, within each group, we perform aggregation by `COUNT`ing all the rows. By selecting `number` and `COUNT(*)`, we then can see the highest `number` and how many students picked that number. We have to order by our `COUNT(*)`, which is saved in the alias `count`, by `DESC` ending order, so our highest count starts at the top, and we limit our result to the top 10.

Q6: Let's Count

Let's have some fun with this! For each query below, we created its own table in `lab12_extra.sql`, so fill in the corresponding table and run it using Ok. Try working on this on your own or with a neighbor before toggling to see the solutions.

> *Hint:* You may find that there isn't a particular attribute you should have to perform the `COUNT` aggregation over. If you are only interested in counting the number of rows in a group, you can just say `COUNT(*)`.

What are the top 10 pets this semester?

```
sqlite> SELECT * FROM sp19favpets;
cat|19
dog|19
panda|7
dragon|6
elephant|3
hedgehog|3
lion|3
red panda|3
baby elephant|2
dolphin|2
```

```
CREATE TABLE sp19favpets AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

How many people marked exactly the word 'dog' as their ideal pet this semester?

```
sqlite> SELECT * FROM sp19dog;
dog|19
```

```
CREATE TABLE sp19dog AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

Although close, our query doesn't give us an entirely accurate picture of what people's favorite pets are. For example, `a dog` would not be counted the same as `dog`. Let's see how many people actually want a dog this semester by using `LIKE`, which compares substrings. We can use it inside `WHERE`, as in `WHERE [column_name] LIKE '%[word]%'` to find how many people would like some type of dog.

```
sqlite> SELECT * from sp19alldogs;
dog|24
```

```
CREATE TABLE sp19alldogs AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

We can find the student's favorite for any column (try it yourself in the interpreter), but let's go back to our Obedience question. Let's see how many obedient students this semester picked each image of an animal. We can do this by selecting only the rows that have `seven = '7'` then `GROUP BY animal`, and finally we can `COUNT` them.

```
sqlite> SELECT * FROM obedienceimages;
7|1 (Karthik Bharathala)|13
7|2 (Rachel De Jaen)|3
7|3 (Tim Foster)|6
7|4 (Chae Park)|16
7|5 (Chae Park)|10
7|6 (Rachel De Jaen catsits)|5
7|7 (Kevin Yu)|7
7|8 (Jade Singh)|1
7|9 (Jacqueline Tiffany Yeung)|3
```

```
CREATE TABLE obedienceimages AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

The possibilities are endless, so have fun experimenting!

Use Ok to test your code:

```
python3 ok -q lets-count
```

Q7: The Smallest Unique Positive Integer (Part 2)

Now, let's revisit the previous problem of finding the smallest positive integer that anyone chose, and take a closer look at the `COUNT` aggregate.

Write a SQL query that uses the `COUNT` aggregate to create a table that pairs the attribute `smallest` with the number of times it was chosen by a student (this is the aggregation part).

> *Hint:* Think about what attribute you need to `GROUP BY`.

After you've defined your table, you should get something like:

```
sqlite> SELECT * FROM smallest_int_count LIMIT 25;
1|60
2|9
3|6
4|6
5|1
6|3
7|3
8|1
9|4
10|1
11|5
12|1
13|3
14|4
15|3
17|2
19|4
21|1
22|3
23|4
29|2
30|1
31|3
32|1
34|3
```

```
CREATE TABLE smallest_int_count AS
  SELECT "REPLACE THIS LINE WITH YOUR SOLUTION";
```

Use Ok to test your code:

```
python3 ok -q smallest-int-count
```

It looks like the number `5` only had one person choose it! Were you the lucky student that put it down?

Troubleshooting/Advanced SQLite

Troubleshooting

Python already comes with a built-in SQLite database engine to process SQL. However, it doesn't come with a "shell" to let you interact with it from the terminal. Because of this, until now, you have been using a simplified SQLite shell written by us. However, you may find the shell is old, buggy, or lacking in features. In that case, you may want to download and use the official SQLite executable.

If running `python3 sqlite_shell.py` didn't work, you can download a precompiled sqlite directly by following the following instructions and then use `sqlite3` and `./sqlite3` instead of `python3 sqlite_shell.py` based on which is specified for your platform.

Setup

CS 61A (/)

Weekly Schedule (/weekly.html)

Office Hours (/office-hours.html)

Staff (/staff.html)

Resources (/resources.html)

Studying Guide (/articles/studying.html)

Debugging Guide (/articles/debugging.html)

Composition Guide (/articles/composition.html)

Policies (/articles/about.html)

Assignments (/articles/about.html#assignments)

Exams (/articles/about.html#exams)

Grading (/articles/about.html#grading)