* [Introduction](https://cs50.harvard.edu/web/2020/notes/4/" \l "introduction)
* [SQL](https://cs50.harvard.edu/web/2020/notes/4/#sql)
  + [Databases](https://cs50.harvard.edu/web/2020/notes/4/#databases)
  + [Column Types](https://cs50.harvard.edu/web/2020/notes/4/#column-types)
* [Tables](https://cs50.harvard.edu/web/2020/notes/4/#tables)
* [SELECT](https://cs50.harvard.edu/web/2020/notes/4/#select)
  + [Working with SQL in the Terminal](https://cs50.harvard.edu/web/2020/notes/4/#working-with-sql-in-the-terminal)
  + [Functions](https://cs50.harvard.edu/web/2020/notes/4/#functions)
  + [UPDATE](https://cs50.harvard.edu/web/2020/notes/4/#update)
  + [DELETE](https://cs50.harvard.edu/web/2020/notes/4/#delete)
  + [Other Clauses](https://cs50.harvard.edu/web/2020/notes/4/#other-clauses)
* [Joining Tables](https://cs50.harvard.edu/web/2020/notes/4/#joining-tables)
  + [JOIN Query](https://cs50.harvard.edu/web/2020/notes/4/#join-query)
  + [Indexing:](https://cs50.harvard.edu/web/2020/notes/4/#indexing)
  + [SQL Vulnerabilities](https://cs50.harvard.edu/web/2020/notes/4/#sql-vulnerabilities)
* [Django Models](https://cs50.harvard.edu/web/2020/notes/4/#django-models)
* [Migrations](https://cs50.harvard.edu/web/2020/notes/4/#migrations)
* [Shell](https://cs50.harvard.edu/web/2020/notes/4/#shell)
  + [Starting our application](https://cs50.harvard.edu/web/2020/notes/4/#starting-our-application)
* [Django Admin](https://cs50.harvard.edu/web/2020/notes/4/#django-admin)
* [Many-to-Many Relationships](https://cs50.harvard.edu/web/2020/notes/4/#many-to-many-relationships)
* [Users](https://cs50.harvard.edu/web/2020/notes/4/#users)

[**Introduction**](https://cs50.harvard.edu/web/2020/notes/4/#introduction)

* So far, we’ve discussed how to build simple web pages using HTML and CSS, and how to use Git and GitHub in order to keep track of changes to our code and collaborate with others. We also familiarized ourselves with the Python programming language, and started using Django to create web applications.
* Today, we’ll learn about using SQL and Django models to efficiently store and access data.

## [SQL](https://cs50.harvard.edu/web/2020/notes/4/#sql)

: [SQL](https://www.w3schools.com/sql/), or Structured Query Language, is a programming language that allows us to update and query databases.

### [Databases](https://cs50.harvard.edu/web/2020/notes/4/#databases)

Before we get into how to use the SQL language, we should discuss how our data is stored. When using SQL, we’ll work with a [relational database](https://www.oracle.com/database/what-is-a-relational-database/#:~:text=A%20relational%20database%20is%20a,of%20representing%20data%20in%20tables.) where we can find all of our data stored in a number of [tables](https://www.essentialsql.com/what-is-a-database-table/). Each of these tables is made up of a set number of columns and a flexible number of rows.

To illustrate how to work with SQL, we’ll use the example of a website for an airline used to keep track of flights and passengers. In the following table, we see that we’re keeping track of a number of flights, each of which has an origin, a destination, and a duration.

There are several different relational database management systems that are commonly used to store information, and that can easily interact with SQL commands:

* [MySQL](https://www.mysql.com/)
* [PostgreSQL](https://www.postgresql.org/)
* [SQLite](https://www.sqlite.org/index.html)
* …

The first two, MySQL and PostgreSQL, are heavier-duty database management systems that are typically run on servers separate from those running a website. SQLite, on the other hand, is a lighter-weight system that can store all of its data in a single file. We’ll be using SQLite throughout this course, as it is the default system used by Django.

### [Column Types](https://cs50.harvard.edu/web/2020/notes/4/#column-types)

Just as we worked with several different variable types in Python, SQLite has [types](https://www.sqlite.org/datatype3.html) that represent different forms of information. Other management systems may have different data types, but all are fairly similar to those of SQLite:

* TEXT: For strings of text (Ex. a person’s name)
* NUMERIC: A more general form of numeric data (Ex. A date or boolean value)
* INTEGER: Any non-decimal number (Ex. a person’s age)
* REAL: Any real number (Ex. a person’s weight)
* BLOB (Binary Large Object): Any other binary data that we may want to store in our database (Ex. an image)

[**Tables**](https://cs50.harvard.edu/web/2020/notes/4/#tables)

Now, to actually get started with using SQL to interact with a database, let’s begin by creating a new table. The [command to create a new table](https://www.w3schools.com/sql/sql_create_table.asp) looks something like this:

CREATE TABLE flights(

id INTEGER PRIMARY KEY AUTOINCREMENT,

origin TEXT NOT NULL,

destination TEXT NOT NULL,

duration INTEGER NOT NULL

);

In the above command, we’re creating a new table that we’ve decided to call flights, and we’ve added four columns to this table:

1. id: It is often helpful to have an number that allows us to uniquely identify each row in a table. Here we have specified that id is an integer, and also that it is our [primary key](https://www.w3schools.com/sql/sql_primarykey.ASP), meaning it is our unique identifier. We have additionally specified that it will AUTOINCREMENT, which means we will not have to provide an id every time we add to the table because it will be done automatically.
2. origin: Here we’ve specified that this will be a text field, and by writing NOT NULL we have required that it have a value.
3. destination: Again we’ve specified that this will be a text field and prevented it from being null.
4. duration: Again this value cannot be null, but this time it is represented by an integer rather than as text.

We just saw the NOT NULL and PRIMARY KEY constraint when making a column, but there are several other [constraints](https://www.tutorialspoint.com/sqlite/sqlite_constraints.htm) available to us:

* CHECK: Makes sure certain constraints are met before allowing a row to be added/modified
* DEFAULT: Provides a default value if no value is given
* NOT NULL: Makes sure a value is provided
* PRIMARY KEY: Indicates this is the primary way of searching for a row in the database
* UNIQUE: Ensures that no two rows have the same value in that column.
* …

Now that we’ve seen how to create a table, let’s look at how we can add rows to it. In SQL, we do this using the INSERT command:

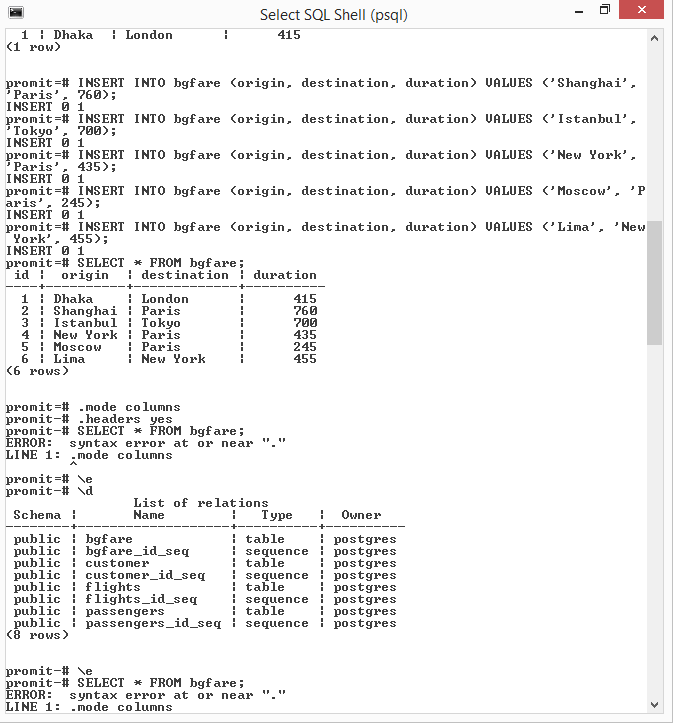
INSERT INTO flights

(origin, destination, duration)

VALUES (‘New York’, ‘London’, 415);

In the above command, we’ve specified the table name we wish to insert into, then provided a list of the column names we will be providing information on, and then specified the VALUES we would like to fill that row in the table, making sure the VALUES come in the same order as our corresponding list of columns. Note that we don’t need to provide a value for id because it is automatically incrementing.



****

[**SELECT**](https://cs50.harvard.edu/web/2020/notes/4/#select)

Once a table has been populated with some rows, we’ll probably want a way to access data within that table. We do this using SQL’s [SELECT](https://www.w3schools.com/sql/sql_select.asp) query. The simplest SELECT query into our flights table might look something like this:

SELECT \* FROM flights;

The above command (\*) retrieves all of the data from our flights table

It may be the case though that we don’t really need all of the columns from the database, just origin and destination. To access just these columns, we can replace the \* with the column names we would like access to. The following query returns all of the origins and destinations.

SELECT origin, destination FROM flights;

As our tables get larger and larger, we will also want to narrow down which rows our query returns. We do this by adding a [WHERE](https://www.w3schools.com/sql/sql_where.asp) followed by some condition. For example, the following command selects only row with an id of 3:

SELECT \* FROM flights WHERE id = 3;

e can filter by any column, not just id!

SELECT \* FROM flights WHERE origin = ‘New York’;

### [Working with SQL in the Terminal](https://cs50.harvard.edu/web/2020/notes/4/#working-with-sql-in-the-terminal)

Now that we know some basic SQL commands, let’s test them out in the terminal! In order to work with SQLite on your computer, you must first [download SQLite](https://www.sqlite.org/download.html). (We won’t use it in lecture, but you can also [download DB Browser](https://sqlitebrowser.org/dl/) for a more user-friendly way to run SQL queries.)

We can start by creating a file for our database either by manually creating a new file, or running touch flights.sql in the terminal. Now, if we run sqlite3 flights.sql in the terminal, we’ll be brought to a SQLite prompt where we can run SQL commands:

# Entering into the SQLite Prompt

(base) % sqlite3 flights.sql

SQLite version 3.26.0 2018-12-01 12:34:55

Enter ‘.help’ for usage hints.

# Creating a new Table

CREATE TABLE flights(

id INTEGER PRIMARY KEY AUTOINCREMENT,

...> origin TEXT NOT NULL,

...> destination TEXT NOT NULL,

...> duration INTEGER NOT NULL

...> );

# Listing all current tables (Just flights for now)

sqlite> .tables

flights

# Querying for everything within flights (Which is now empty)

sqlite> SELECT \* FROM flights;

# Adding one flight

sqlite> INSERT INTO flights

...> (origin, destination, duration)

...> VALUES (‘New York’, ‘London’, 415);

# Checking for new information, which we can now see

sqlite> SELECT \* FROM flights;

1|New York|London|415

# Adding some more flights

sqlite> INSERT INTO flights (origin, destination, duration) VALUES (‘Shanghai’, ‘Paris’, 760);

sqlite> INSERT INTO flights (origin, destination, duration) VALUES (‘Istanbul’, ‘Tokyo’, 700);

sqlite> INSERT INTO flights (origin, destination, duration) VALUES (‘New York’, ‘Paris’, 435);

sqlite> INSERT INTO flights (origin, destination, duration) VALUES (‘Moscow’, ‘Paris’, 245);

sqlite> INSERT INTO flights (origin, destination, duration) VALUES (‘Lima’, ‘New York’, 455);

# Querying this new information

sqlite> SELECT \* FROM flights;

1|New York|London|415

2|Shanghai|Paris|760

3|Istanbul|Tokyo|700

4|New York|Paris|435

5|Moscow|Paris|245

6|Lima|New York|455

# Changing the settings to make output more readable

sqlite> .mode columns

sqlite> .headers yes

# Querying all information again

sqlite> SELECT \* FROM flights;

id origin destination duration

---------- ---------- ----------- ----------

1 New York London 415

2 Shanghai Paris 760

3 Istanbul Tokyo 700

4 New York Paris 435

5 Moscow Paris 245

6 Lima New York 455

# Searching for just those flights originating in New York

sqlite> SELECT \* FROM flights WHERE origin = ‘New York’;

id origin destination duration

---------- ---------- ----------- ----------

1 New York London 415

4 New York Paris 435

We can also use more than just equality to filter out our flights. For integer and real values, we can use greater than or less than:

use other logic ([AND, OR](https://www.w3schools.com/sql/sql_and_or.asp)) like in Python:

SELECT \* FROM SELECT \* FROM flights WHERE duration > 500;

And we can also flights WHERE duration > 500 AND destination = ‘Paris’;

SELECT \* FROM flights WHERE duration > 500 OR destination = ‘Paris’;

We can also use the keyword [IN](https://www.w3schools.com/sql/sql_in.asp) to see if a bit of data is one of several options:

SELECT \* FROM flights WHERE origin IN (‘New York’, ‘Lima’);

We can even use regular expressions to search words more broadly using the [LIKE](https://www.w3schools.com/sql/sql_like.asp) keyword. The below query finds all results with an a in the origin, by using % as a wildcard character.

SELECT \* FROM flights WHERE origin LIKE ‘%a%’;

### [Functions](https://cs50.harvard.edu/web/2020/notes/4/#functions)

There are also a number of SQL functions we can apply to the results of a query. These can be useful if we don’t need all of the data returned by a query, but just some summary statistics of the data.

* [AVERAGE](https://www.w3schools.com/sql/sql_count_avg_sum.asp)
* [COUNT](https://www.w3schools.com/sql/sql_count_avg_sum.asp)
* [MAX](https://www.w3schools.com/sql/sql_min_max.asp)
* [MIN](https://www.w3schools.com/sql/sql_min_max.asp)
* [SUM](https://www.w3schools.com/sql/sql_count_avg_sum.asp)
* …

### [UPDATE](https://cs50.harvard.edu/web/2020/notes/4/#update)

We’ve now seen how to add to and search tables, but we may also want to be able update rows of a table that already exist. We do this using the [UPDATE](https://www.w3schools.com/sql/sql_update.asp) command as shown below. As you may have guessed by reading this out loud, the command finds any flights that go from New York to London, and then sets their durations to 430.

UPDATE flights

SET duration = 430

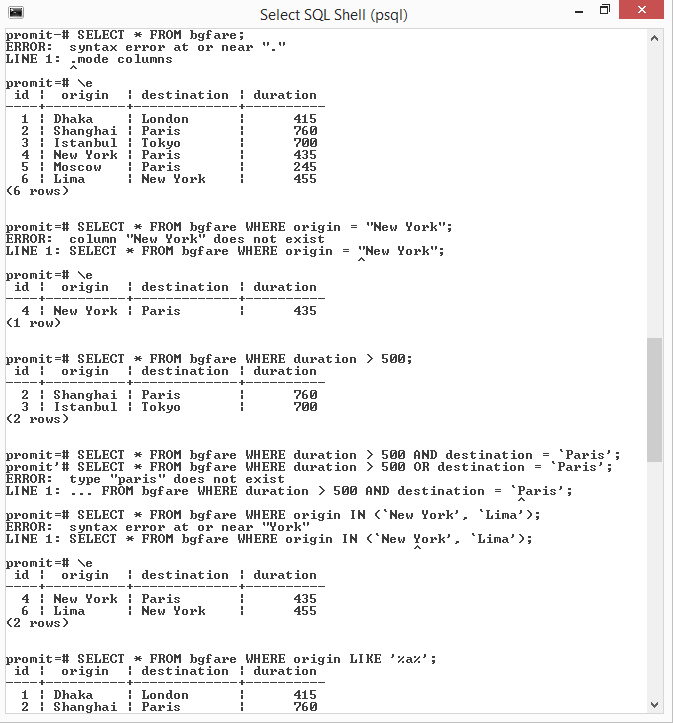
WHERE origin = ‘New York’

AND destination = ‘London’;

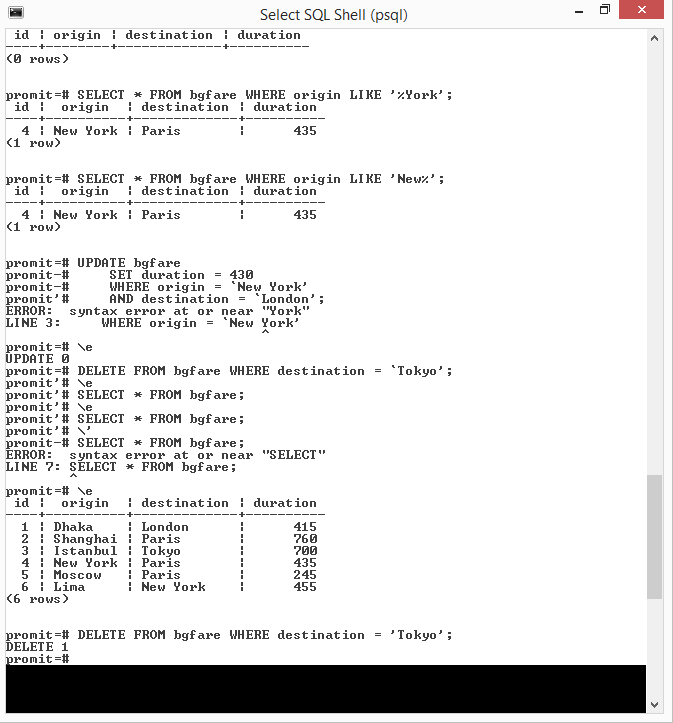
### [DELETE](https://cs50.harvard.edu/web/2020/notes/4/#delete)

We also might want the ability to delete rows from our database, and we can do this using the [DELETE](https://www.w3schools.com/sql/sql_delete.asp) command. The following code will remove all flights that land in Tokyo:

DELETE FROM flights WHERE destination = ‘Tokyo’;



N



### [Other Clauses](https://cs50.harvard.edu/web/2020/notes/4/#other-clauses)

There are a number of additional clauses we can use to control queries coming back to us

* [LIMIT](https://www.w3schools.com/sql/sql_top.asp): Limits the number of results returned by a query
* [ORDER BY](https://www.w3schools.com/sql/sql_orderby.asp): Orders the results based on a specified column
* [GROUP BY](https://www.w3schools.com/sql/sql_groupby.asp): Groups results by a specified column
* [HAVING](https://www.w3schools.com/sql/sql_having.asp): Allows for additional constraints based on the number of results

## [Joining Tables](https://cs50.harvard.edu/web/2020/notes/4/#joining-tables)

So far, we’ve only been working with one table at a time, but many databases in practice are populated by a number of tables that all relate to each other in some way. In our flights example, let’s imagine we also want to add an airport code to go with the city. The way our table is currently set up, we would have to add two more columns to go with each row. We would also be repeating information, as we would have to write in multiple places that city X is associated with code Y.

One way we can solve this problem is by deciding to have one table that keeps track of flights, and then another table keeping track of airports. The second table might look something like this

Now we have a table relating codes and cities, rather than storing an entire city name in our flights table, it will save storage space if we’re able to just save the id of that airport. Therefore, we should rewrite the flights table accordingly. Since we’re using the id column of the airports table to populate origin\_id and destination\_id, we call those values [Foreign Keys](https://www.w3schools.com/sql/sql_foreignkey.asp)

In addition to flights and airports, an airline might also want to store data about its passengers, like which flight each passenger will be on. Using the power of relational databases, we can add another table that stores first and last names, and a foreign key representing the flight they are on

We can do even better than this though, as the same person may be on more than one flight. To account for this, we can create a people table that stores first and last names, and a passengers table that pairs people with flights

ecause in this case a single person can be on many flights and a single flight can have many people, we call the relationship between flights and people a **Many to Many** relationship. The passengers table that connects the two is known as an **association table**.

### [JOIN Query](https://cs50.harvard.edu/web/2020/notes/4/#join-query)

Although our data is now more efficiently stored, it seems like it may be harder to query our data. Thankfully, SQL has a [JOIN](https://www.w3schools.com/sql/sql_join.asp) query where we can combine two tables for the purposes of another query.

For example, let’s say we want to find the origin, destination, and first name of every trip a passenger is taking. Also for simplicity in this table, we’re going to be using the unoptimized passengers table that includes the flight id, first name, and last name. The first part of this query looks fairly familiar:

SELECT first, origin, destination

FROM ...

But we run into a problem here because first is stored in the passengers table, while origin and destination are stored in the flights table. We solve this by joining the two tables using the fact that flight\_id in the passengers table corresponds to id in the flights table:

SELECT first, origin, destination

FROM flights JOIN passengers

ON passengers.flight\_id = flights.id;

We’ve just used something called an [INNER JOIN](https://www.w3schools.com/sql/sql_join_inner.asp), which means we are ignoring rows that have no matches between the tables, but there are other types of joins, including [**LEFT JOIN**s](https://www.w3schools.com/sql/sql_join_left.asp), [**RIGHT JOIN**s](https://www.w3schools.com/sql/sql_join_right.asp), and [**FULL OUTER JOIN**s](https://www.w3schools.com/sql/sql_join_full.asp), which we won’t discuss here in detail.

### [Indexing:](https://cs50.harvard.edu/web/2020/notes/4/#indexing)

One way we can make our queries more efficient when dealing with large tables is to create an index similar to the index you might see in the back of a textbook. For example, if we know that we’ll often look up passengers by their last name, we could create an index from last name to id using the command:

CREATE INDEX name\_index ON passengers (last);

### [SQL Vulnerabilities](https://cs50.harvard.edu/web/2020/notes/4/#sql-vulnerabilities)

**Now that we know the basics of using SQL to work with data, it’s important to point out the main vulnerabilities associated with using SQL. We’ll start with**[**SQL Injection**](https://www.w3schools.com/sql/sql_injection.asp)**.**

A SQL injection attack is when a malicious user enters SQL code as input on a site in order to bypass the sites security measures. For example, let’s say we have a table storing usernames and passwords, and then a login form on the home site of a page. We may search for the user using a query such as:

SELECT \* FROM users

WHERE username = username AND password = password;

A user named Harry might go to this site and type harry as a username and 12345 as a password, in which case the query would look like this:

SELECT \* FROM users

WHERE username = ‘harry’ AND password = ‘12345’;

A hacker, on the other hand, might type harry’ -- as a username and nothing as a password. It turns out that -- stands for a comment in SQL, meaning the query would look like:

SELECT \* FROM users

WHERE username = ‘harry’--’ AND password = ‘12345’;

Because in this query the password checking has been commented out, the hacker can log into Harry’s account without knowing their password. To solve this problem, we can use:

* Escape characters to make sure SQL treats the input as plain text and not as SQL code.
* An abstraction layer on top of SQL which includes its own escape sequence, so we don’t have to write SQL queries ourselves.

**The other main vulnerability when it comes to SQL is known as a**[**Race Condition**](https://searchstorage.techtarget.com/definition/race-condition#:~:text=A%20race%20condition%20is%20an,sequence%20to%20be%20done%20correctly.)**.**

A race condition is a situation that occurs when multiple queries to a database occur simultaneously. When these are not adequately handled, problems can arise in the precise times that databases are updated. For example, let’s say I have $150 in my bank account. A race condition could occur if I log into my bank account on both my phone and my laptop, and attempt to withdraw $100 on each device. If the bank’s software developers did not deal with race conditions correctly, then I may be able to withdraw $200 from an account with only $150 in it. One potential solution for this problem would be locking the database. We could not allow any other interaction with the database until one transaction has been completed. In the bank example, after clicking navigating to the ‘Make a