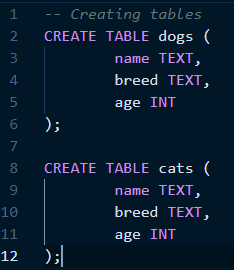
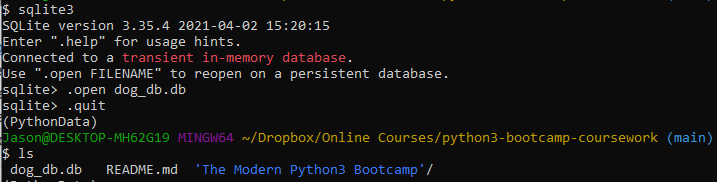
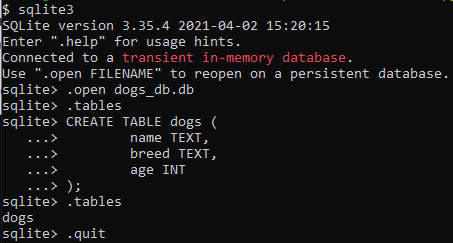
* There are a ton of different SQL databases out there and they are all a pain to install
* In this section we will be using **sqlite3**
  + Comes pre-installed on many machines
  + Very simple syntax, not overly complicated but also not particularly powerful
  + Goal is to focus on the Python portion
  + Python3 comes with the sqlite3 module as part of the standard library
  + Documentation: <https://sqlite.org/docs.html>
* Databases
  + The idea is that data is stored in tables
  + Tables are composed of rows and columns, which can contain different data types
* Data types (<https://sqlite.org/datatype3.html>)
  + **NULL** - The value is a NULL value.
  + **INTEGER -** The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.
  + **REAL** - The value is a floating point value, stored as an 8-byte IEEE floating point number.
  + **TEXT** - The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).
  + **BLOB** - The value is a blob of data, stored exactly as it was input.
  + Note that there are no Booleans or datetype datatypes in SQLite
* Creating tables in sqlite using basic syntax
  + Creating tables uses standard SQL syntax
  + Command: CREATE TABLE
    - Does not technically need to be capitalized, but the convention is to capitalize the command while not capitalizing the data input
    - This is followed by the name of the table, conventionally not capitalized
    - Finally, we have parentheses into which we enter the names of the columns that we want along with the datatypes
    - Just like any other SQL statements, the CREATE TABLE command must end with a semicolon (;)
  + In the example below, we are creating two different tables, one called dogs and one called cats. Both tables have three columns, one for name, breed, and age. The tables are initially empty. Note that the code is written in VS Code, and we copy and paste into SQLite. We can confirm the creation of the tables by using the **.tables** meta command within sqlite



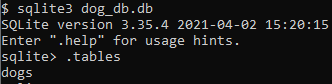
* An important aspect of sqlite3 is that it is part of a **transient, in-memory database**. It is NOT a client-server database engine, but rather is embedded within the end program
  + This means that if we exit out of sqlite, we lose our databases
  + In order to save our databases, we need to save them to files. The opening prompt of sqlite3 actually tells us as much
    - **.open** command is used to open or create a new database file



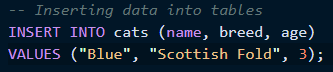
* + - Let’s re-open the database file and create a table called *dogs*.



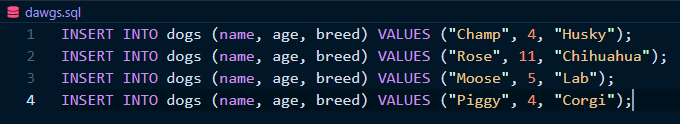
* + - We can then re-open this database within sqlite3 by running *sqlite3 database\_filename*.
      * This syntax can also be used to create a new database file if a file of that name does not yet exist.

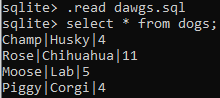


* **Inserting** data into tables
  + Syntax: INSERT INTO (table\_name) (column\_names – comma-separated) VALUES (*VALUES* – comma-separated)
  + The column names need to be declared in the same order as the inserted values, otherwise your data will get messed up

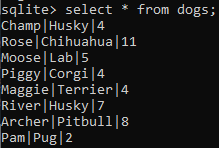


* + You can also execute code as SQL so that you don’t have to manually type or copy-and-paste things into sqlite. Instead, you can write your code into a .sql file, then use the .read command in sqlite to read and execute that code

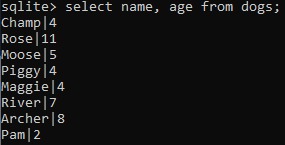




* **Selecting** data from tables
  + The “SELECT \*” command will select all columns from a table

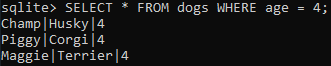


* + You can also specify individual columns – note that all rows will still be selected

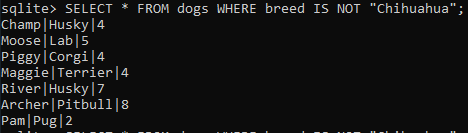


* + You can use the **where** clause to identify specific rows to select

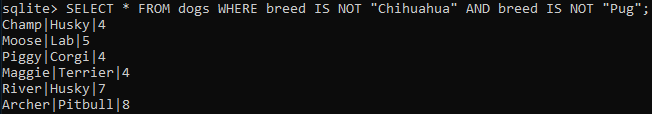




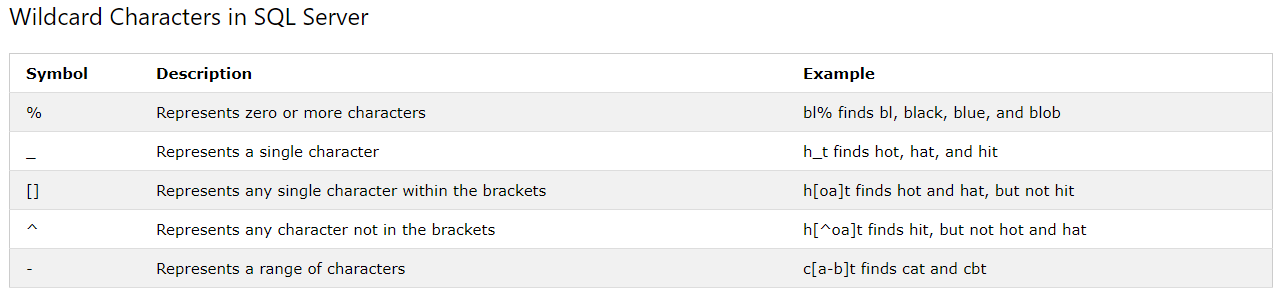
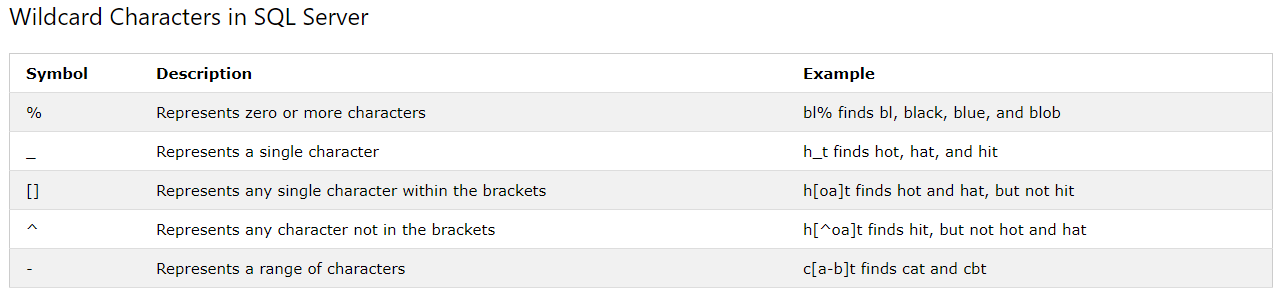
* + More on selection: <https://sqlite.org/lang_select.html>
  + You can use negation as well, using **NOT**



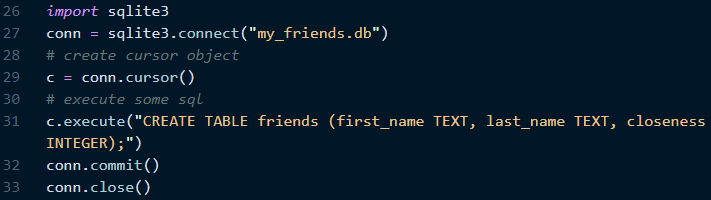
* + You can also chain together negations



* + Wildcards can be used to make selections



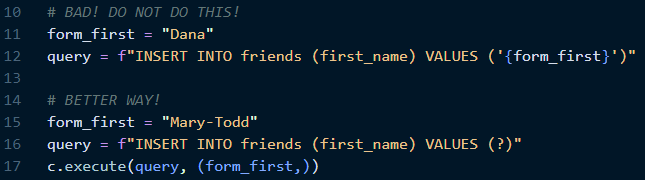
* Using Python with SQL
  + Python comes with a built-in library, **sqlite3**, to communicate with sqlite3. It acts a connector between sqlite and Python
  + To create a connection, we use sqlite3**.connect(‘database\_filename’)**. This is usually saved as a variable
    - If the database does not exist already, it will be created by the .connect() command
  + Then the **.cursor()** command is called on the connection variable to create a **cursor object**. Again this is usually saved as a variable
    - A cursor is a temporary work area/space for SQL commands, with a dedicated bit of memory allocated for the task
  + We can then use **.execute()** command on the cursor object to write SQL code.
    - Within the execute command, we can use direct SQL syntax
    - Watch for typos, and don’t forget the semicolon! You also need to enclose the entire SQL command in quotes
  + Next, we use the **.commit()** method to commit the SQL statement
  + Finally, we close the connection using the **.close()**
  + Here is an example of connecting, creating the cursor object, creating a table, committing the table, and closing the connection



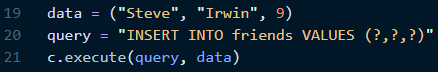
* How about **inserting data into tables** with Python? We can do that too!
  + We can use .execute() and use regular SQL code for your insertion. However, this kind of defeats the purpose of using Python because it would be simpler just to go straight to SQL and made the insertion
    - Example:



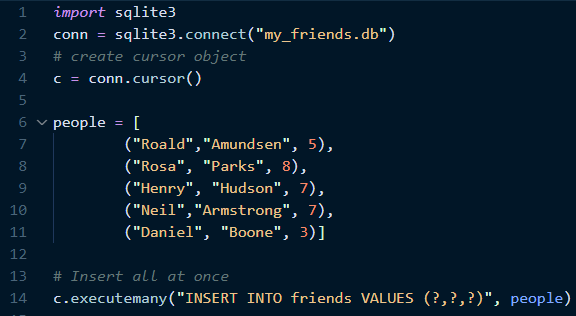
* The power of Python comes in when you have some *dynamic data*. For example, maybe you have a bunch of lines from a CSV file that you want to convert into a SQL database!
  + Don’t try to use f-strings to write SQL commands, there’s another way to do it
  + Instead, use a question mark to indicate values that you want to sub in. Each question mark serves as a placeholder for a variable that you want to insert into the query string. Finally, use the .execute() command again, passing in the query string and a tuple containing the values that you want to insert as arguments to the method. Yes, it has to be a tuple!
    - If it’s a single item tuple, don’t forget the comma after that item



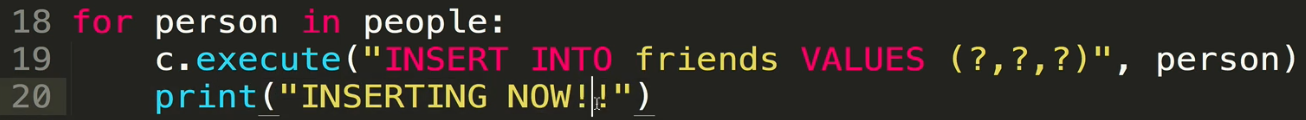
* + You wouldn’t really use the above approach when you’re just inserting one piece of data. The true value of this approach comes when you want to insert more than that
    - In the example below, you’ve got three pieces of data to pass into a table with three columns. The data will be inserted into the SQL query according the positions of the question marks!
    - This is useful because SQL does not allow you to save data to variables in this manner. This illustrates the utility of Python
    - Note that we don’t need to specify the column names because the order of the data matches the order of the columns that they are being inserted into



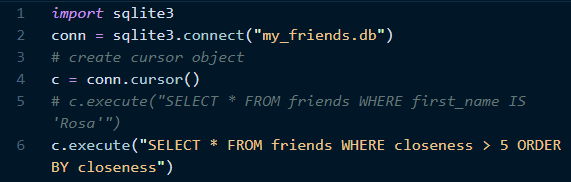
* One of the advantages of using Python for SQL is the ability to do bulk insertions
  + Sqlite3 has a method called **executemany()**, which you call on your cursor object, and pass in as arguments the SQL command and a list containing tuples of data that you want to pass in



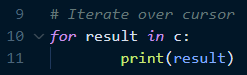
* + But what if we wanted to do something else along with inserting the values, and we want to do this different thing with each person in the list? In this example, we iterate over the list and insert each person one at a time, printing the fact that we are making an insertion

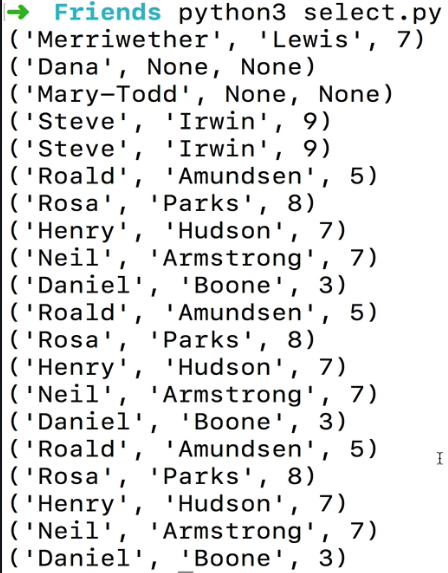


* How do we SELECT information from a SQL database using Python? What makes this different from insertion is that we’re receiving data back from the database! So how do we handle that?
  + Syntactically, the approach is similar to INSERT

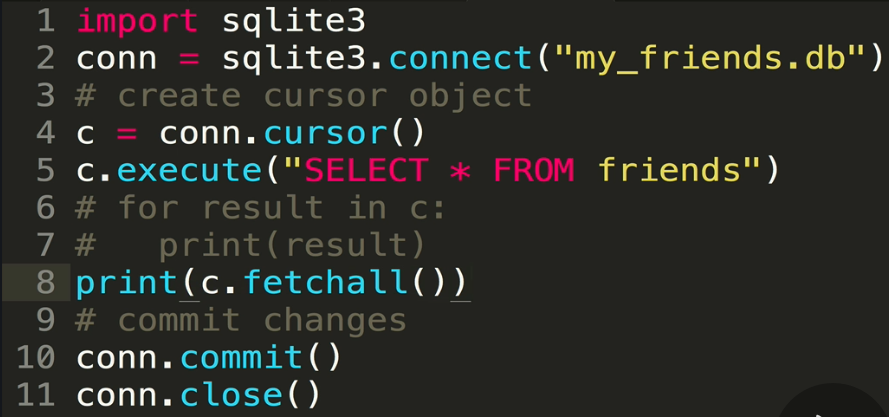


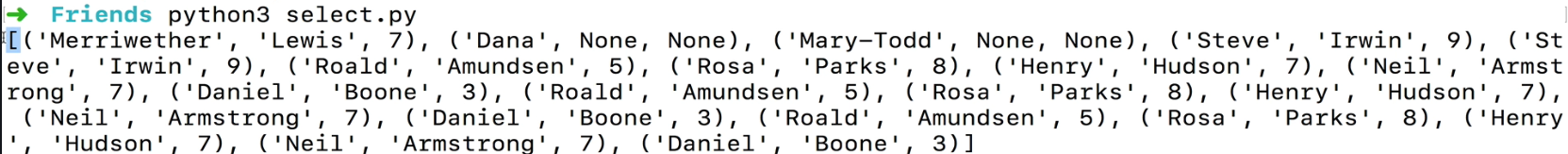
* + We can’t just simply print **c** in this case, because that will just give us the cursor object. However, the cursor object is an iterator that we can iterate over! Using a for loop, we can get all of the data that has been selected and print the result of each iteration.





* + There is another option as well, which is the **.fetchall()** function. We call this as a method on the cursor object, and the result comes back as a single self-contained list. We can then use that list as we would any other Python list



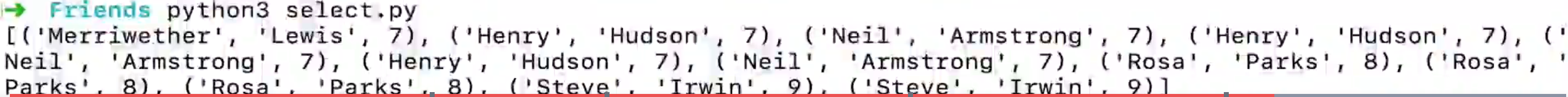


* + We don’t need to select \* all the time either! We can just select the first item that matches the query conditions. This is done using the **.fetchone()** method, and it does not require iterating over the entire cursor

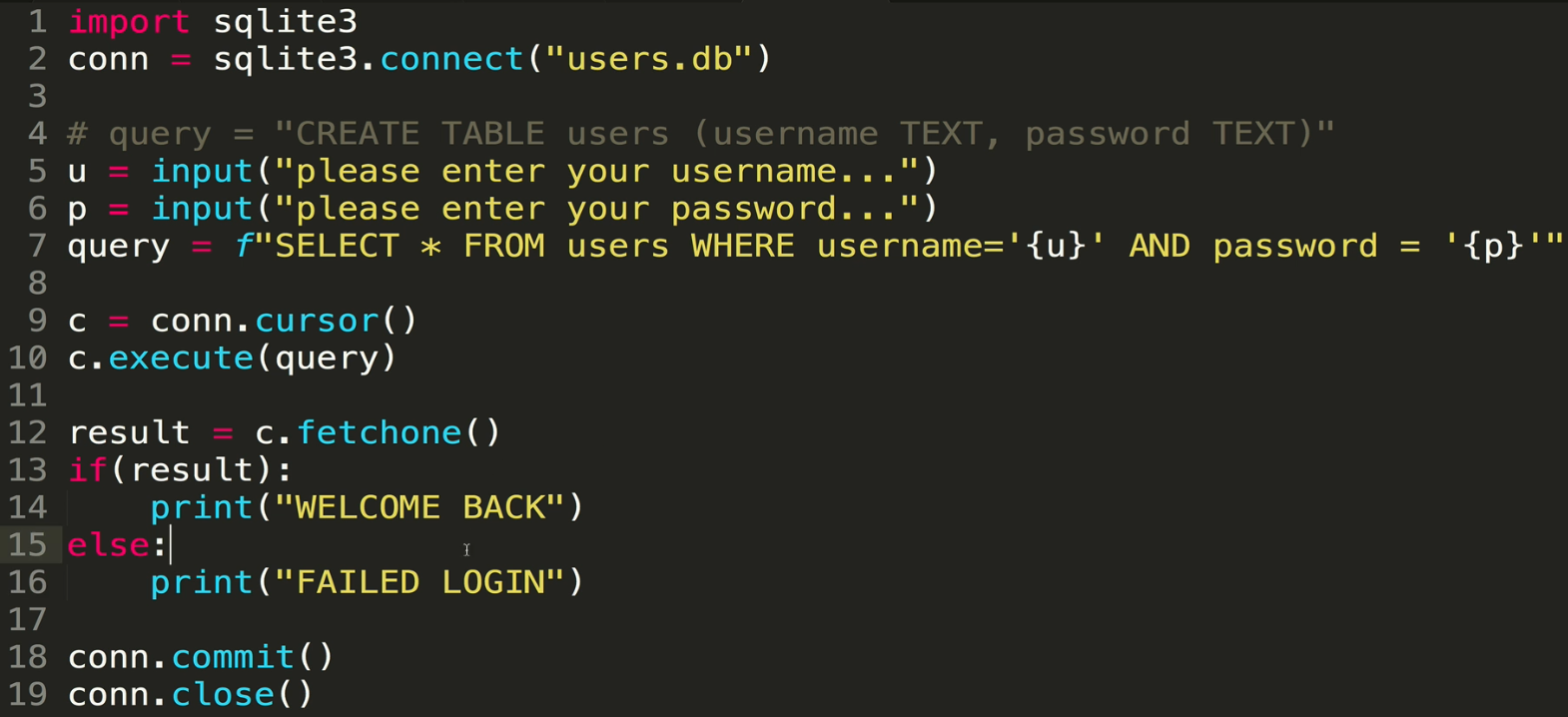


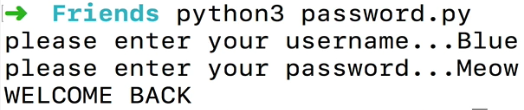
* + Another example of selection where we apply a condition and order the selection. In the example below, we’ll select only friends who have a closeness score of greater than 5, and they will be ordered by closeness

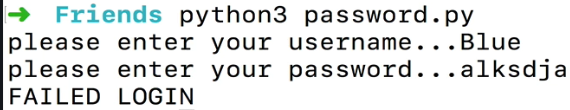




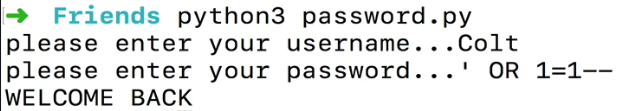
* **SQL Injection** – an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution
  + https://www.w3schools.com/sql/sql\_injection.asp
  + The idea is that users write pure SQL and inject it into a query
  + This is the reason we didn’t want to use f-strings to make SQL commands
  + Example with username and password using f-strings (DO NOT DO THIS!)







* + Because we’re just directly inserting our variables into a SQL query, a malicious user could just write SQL as part of their inputs. In this example, instead of providing a password, we end the password with a single quote and then pass in an OR statement and a condition that will always resolve to true. This transforms the query into “select everything where 1=1”, which is always true, and everything in the table will be selected!



* + - The double dashes at the end is comment in SQL. By including it, we eliminate the quote at the end that is supposed to go into the query



* + Here is a much safer approach that does not use f-strings, but rather passes the inputs into the query using the SQL “?” syntax. Instead of passing the input directly into the query, we pass the information as a tuple into the .execute() function along with the query. This means the sqlite library will take care of data interpolation, and we shield ourselves from malicious SQL injections

