1. Echo hello PostgreSQL(PostgreSQL is a powerful, open source object-relational database system that uses and extends the SQL language)
2. .Your virtual machine comes with PostgreSQL installed. You will use the Psql terminal application to interact with it. Log in by typing `psql --username=freecodecamp --dbname=postgres` into the terminal and pressing enter.
3. Notice that the prompt changed to let you know that you are now interacting with PostgreSQL. First thing to do is see what databases are here. Type `\l` into the prompt to list them.
4. The databases you see are there by default. You can make your own like this:

```
CREATE DATABASE database_name;
```

The capitalized words are keywords telling PostgreSQL what to do. The name of the database is the lowercase word. Note that all commands need a semi-colon at the end. Create a new database named `first_database`.

5. CREATE DATABASE second_database;
6. You can connect to a database by entering `\c database_name`. You need to connect to add information. Connect to your `second_database`.
7. You should see a message that you are connected. Notice that the prompt changed to `second_database=>`. So the `postgres=>` prompt before must have meant you were connected to that database. A database is made of tables that hold your data. Enter `\d` to display the tables.
8. Looks like there's no tables or relations yet. Similar to how you created a database, you can create a table like this:
9.
```
CREATE TABLE table_name();
```

   Note that the parenthesis are needed for this one. It will create the table in the database you are connected to. Create a table named `first_table` in `second_database`.

10. View the tables in `second_database` again with the display command. You should see your new table there with a little meta data about it.
11. There should be two tables in this database now. Display them again to make sure.
12. You can view more details about a table by adding the table name after the display command like this: `\d table_name`. View more details about your `second_table`
13. Tables need columns to describe the data in them, yours doesn't have any yet. Here's an example of how to add one:
14.
```
ALTER TABLE table_name ADD COLUMN column_name DATATYPE;
```

    Add a column to `second_table` named `first_column`. Give it a data type of `INT`. `INT` stands for integer. Don't forget the semi-colon. 😄

15. Looks like it worked. Display the details of `second_table` again to see if your new column is there
16. Your column is there 😄 Use `ALTER TABLE` and `ADD COLUMN` to add another column to `second_table` named `id` that's a type of `INT`.
17. Your table should have an `id` column added. View the details of `second_table` to make sure.
18. Add another column to `second_table` named `age`. Give it a data type of `INT`.
19. Take a look at the details of `second_table` again.
20. ALTER TABLE second_table DROP COLUMN age; //drop age column from the table
21. It's gone. Use the `ALTER TABLE` and `DROP COLUMN` keywords again to drop `first_column`.
22. A common data type is `VARCHAR`. It's a short string of characters. You need to give it a maximum length when using it like this: `VARCHAR(30)`.

    Add a new column to `second_table`, give it a name of `name` and a data type of `VARCHAR(30)`.

23. you can see the `VARCHAR` type there. The `30` means the data in it can be a max of 30 characters. You named that column `name`, it should have been `username`. Here's how you can rename a column:

    ```
    ALTER TABLE table_name RENAME COLUMN column_name TO new_name;
    ```

    Rename the `name` column to `username`.

24. ake a look at the details of `second_table` again to see if it got renamed.
25. It worked. Rows are the actual data in the table. You can add one like this:

    ```
    INSERT INTO table_name(column_1, column_2) VALUES(value1, value2);
    ```

    Insert a row into `second_table`. Give it an `id` of `1`, and a `username` of `Samus`. The username column expects a `VARCHAR`, so you need to put Samus in single quotes like this: `'Samus'`.

    INSERT INTO second_table(id, username) VALUES(1, 'Samus');

26. You should have one row in your table. You can view the data in a table by querying it with the `SELECT` statement. Here's how it looks:

    ```
    SELECT columns FROM table_name;
    ```

    Use a `SELECT` statement to view all the columns in `second_table`. Use an asterisk (`*`) to denote that you want to see all the columns.

    SELECT * FROM second_table;

27. There's your one row. Insert another row into `second_table`. Fill in the `id` and `username` columns with the values `2` and `'Mario'`.

    INSERT INTO second_table(id, username) VALUES(2, 'Mario');

28. INSERT INTO second_table(id, username) VALUES(3, 'Luigi');
29. DELETE FROM second_table WHERE username='Luigi'
30. DELETE FROM second_table WHERE username='Mario';
31. DELETE FROM second_table WHERE username='Samus';

32. \d second_table
33. There's two columns. You won't need either of them for the Mario database. Alter the table `second_table` and drop the column `username`.

    ALTER TABLE second_table DROP username;

34. ALTER TABLE second_table DROP id;
35. Still two. You won't need either of those for the new database either. Drop `second_table` from your database. Here's an example:

    ```
    DROP TABLE table_name;
    ```

36. ```
    DROP TABLE first_table;
    ```
37. Rename `first_database` to `mario_database`. You can rename a database like this:
38. ```
    ALTER DATABASE database_name RENAME TO new_database_name;
    ```

    ```
    ALTER DATABASE first_database RENAME TO mario_database;
    ```

39. ```
    \c mario_database
    ```

    ```
    You are now connected to database "mario_database" as user
    "freecodecamp".//connect to mario new database
    ```

40. Now that you aren't connected to `second_database`, you can drop it. Use the `DROP DATABASE` keywords to do that.

    DROP DATABASE second_database;

41. Create a new table named `characters`, it will hold some basic information about Mario characters.

    CREATE TABLE characters();

42. The `SERIAL` type will make your column an `INT` with a `NOT NULL` constraint, and automatically increment the integer when a new row is added. View the details of the `characters` table to see what `SERIAL` did for you.
43. Add a column to `characters` called `name`. Give it a data type of `VARCHAR(30)`, and a constraint of `NOT NULL`. Add a constraint by putting it right after the data type.

44. ALTER TABLE characters ADD COLUMN name VARCHAR(30) NOT NULL;
45. You can make another column for where they are from. Add another column named `homeland`. Give it a data type of `VARCHAR` that has a max length of `60`.

    ALTER TABLE characters ADD COLUMN homeland VARCHAR(60);

46. ALTER TABLE characters ADD COLUMN favorite_color VARCHAR(30);
47. INSERT INTO characters(name, homeland, favorite_color) VALUES('Mario','Mushroom Kingdom','Red');
48. Mario should have a row now and his `character_id` should have been automatically added. View all the data in your `characters` table with `SELECT` to see this.

    SELECT * FROM characters;

49. ```
    INSERT INTO characters(name, homeland, favorite_color)
    ```

```
VALUES('Mario', 'Mushroom Kingdom', 'Red'),
('Luigi', 'Mushroom Kingdom', 'Green'),
('Peach', 'Mushroom Kingdom', 'Pink');
```

50. `INSERT INTO characters(name, homeland, favorite_color) VALUES('Toadstool', 'Mushroom Kingdom', 'Red'),('Bowser', 'Mushroom Kingdom', 'Green');`
51. If you don't get a message after a command, it is likely incomplete. This is because you can put a command on multiple lines. Add two more rows. Give the first one the values: `Daisy`, `Sarasaland`, and `Yellow`. The second: `Yoshi`, `Dinosaur Land`, and `Green`. Try to do it with one command.
52. INSERT INTO characters(name, homeland, favorite_color) VALUES('Daisy', 'Sarasaland', 'Yellow'),('Yoshi', 'Dinosaur Land', 'Green');
53. It looks good, but there's a few mistakes. You can change a value like this:
54. `UPDATE table_name SET column_name=new_value WHERE condition;`

    You used `username='Samus'` as a condition earlier. `SET` Daisy's `favorite_color` to `Orange`. You can use the condition `name='Daisy'` to change her row.

    UPDATE characters SET favorite_color='Orange' Where name='Daisy';

55. UPDATE characters SET name='Toad' Where favorite_color='Red';
56. UPDATE characters SET name='Mario' Where character_id=1;
57. UPDATE characters SET favorite_color='Blue' Where character_id=4;
58. Bowser's `favorite_color` is wrong. He likes `Yellow`. Why don't you update it without changing any of the other rows?

    UPDATE characters SET favorite_color='Yellow' Where character_id=5;

59. UPDATE characters SET homeland='Koopa Kingdom' Where character_id=5;
60. Actually, you should put that in order. Here's an example:
61. `SELECT columns FROM table_name ORDER BY column_name;`

    View all the data again, but put it in order by `character_id`.

    SELECT * FROM characters ORDER BY character_id;

```
+--------------+--------+----------------+---------------+
| character_id |  name  |    homeland    | favorite_color |
+--------------+--------+----------------+---------------+
|            1 | Mario  | Mushroom Kingdom | Red          |
|            2 | Luigi  | Mushroom Kingdom | Green        |
|            3 | Peach  | Mushroom Kingdom | Pink         |
```

```
|          4 | Toad   | Mushroom Kingdom | Blue          |
|          5 | Bowser | Koopa Kingdom    | Yellow        |
|          6 | Daisy  | Sarasaland       | Orange        |
|          7 | Yoshi  | Dinosaur Land    | Green         |
+------------+--------+------------------+---------------+
```

62. It looks good. Next, you are going to add a primary key. It's a column that uniquely identifies each row in the table. Here's an example of how to set a `PRIMARY KEY`:

    ```
    ALTER TABLE table_name ADD PRIMARY KEY(column_name);
    ```

    The `name` column is pretty unique, why don't you set that as the primary key for this table.

    ALTER TABLE characters ADD PRIMARY KEY(name);

63. You can see the key for your `name` column at the bottom. It would have been better to use `character_id` for the primary key. Here's an example of how to drop a constraint:

    ```
    ALTER TABLE table_name DROP CONSTRAINT constraint_name;
    ```

    Drop the primary key on the `name` column. You can see the constraint name is `characters_pkey`.

    ALTER TABLE characters DROP CONSTRAINT characters_pkey;

64. ALTER TABLE characters ADD PRIMARY KEY(character_id);
65. That's better. The table looks complete for now. Next, create a new table named `more_info` for some extra info about the characters.

    CREATE TABLE more_info();

66. I wonder what that third one is. It says `characters_character_id_seq`. I think I have a clue. View the details of the `characters` table.
67. That is what finds the next value for the `character_id` column. Add a column to your new table named `more_info_id`. Make it a type of `SERIAL`.
68. ALTER TABLE more_info ADD COLUMN more_info_id SERIAL;
69. ALTER TABLE more_info ADD PRIMARY KEY(more_info_id);
70. ALTER TABLE more_info ADD COLUMN birthday DATE;
71. ALTER TABLE more_info ADD COLUMN height INT;
72. ALTER TABLE more_info ADD COLUMN weight NUMERIC(4,1);
73. There’s your four columns and the primary key you created at the bottom. To know what row is for a character, you need to set a foreign key so you can relate rows from this table to rows from your `characters` table. Here's an example that creates a column as a foreign key:

```
ALTER TABLE table_name ADD COLUMN column_name DATATYPE REFERENCES
referenced_table_name(referenced_column_name);
```

That's quite the command. In the `more_info` table, create a `character_id` column. Make it an `INT` and a foreign key that references the `character_id` column from the `characters` table. Good luck.

ALTER TABLE more_info ADD COLUMN character_id INT REFERENCES characters(character_id);

74. There's your foreign key at the bottom. These tables have a "one-to-one" relationship. One row in the `characters` table will be related to exactly one row in `more_info` and vice versa. Enforce that by adding the `UNIQUE` constraint to your foreign key. Here's an example:
75. ```
    ALTER TABLE table_name ADD UNIQUE(column_name);
    ```

    Add the `UNIQUE` constraint to the column you just added.

    ALTER TABLE more_info ADD UNIQUE(character_id);

76. The column should also be `NOT NULL` since you don't want to have a row that is for nobody. Here's an example:`ALTER TABLE table_name ALTER COLUMN column_name SET NOT NULL;`

    Add the `NOT NULL` constraint to your foreign key column.

    ALTER TABLE more_info ALTER COLUMN character_id SET NOT NULL;

77. The structure is set, now you can add some rows. First, you need to know what `character_id` you need for the foreign key column. You have viewed all columns in a table with `*`. You can pick columns by putting in the column name instead of `*`. Use `SELECT` to view the `character_id` column from the `characters` table.

    SELECT character_id FROM characters;

    +-------------+

    | character_id |

    +-------------+

    | 2 |

    | 3 |

    | 7 |

    | 6 |

    | 1 |

    | 4 |

    | 5 |

    +-------------+

    (7 rows)
    ```

78. INSERT INTO more_info(birthday, height, weight, character_id) VALUES('1981-07-09', 155, 64.5, 1);
79. INSERT INTO more_info(birthday, height, weight, character_id) VALUES('1983-07-14', 175, 48.8, 2);
80. INSERT INTO more_info(birthday, height, weight, character_id) VALUES('1985-10-18', 173, 52.2, 3);
81. Toad is next. Instead of viewing all the rows to find his id, you can just view his row with a `WHERE` condition. You used several earlier to delete and update rows. You can use it to view rows as well. Here's an example:
82. `SELECT columns FROM table_name WHERE condition;`

    A condition you used before was `username='Samus'`. Find Toad's id by viewing the `character_id` and `name` columns from `characters` for only his row.

    SELECT character_id,name FROM characters WHERE name='Toad';

83. INSERT INTO more_info(birthday, height, weight, character_id) VALUES('1950-01-10', 66, 35.6, 4);
84. INSERT INTO more_info(birthday, height, weight, character_id) VALUES('1990-10-29', 258, 300, 5);
85. INSERT INTO more_info(birthday, height, weight, character_id) VALUES('1989-07-31', NULL, NULL, 6);
86. SELECT character_id,name FROM characters WHERE name='Yoshi';
87. INSERT INTO more_info(birthday, height, weight, character_id) VALUES('1990-04-13', 162, 59.1, 7);
88. ALTER TABLE more_info RENAME COLUMN height TO height_in_cm;
89. ALTER TABLE more_info RENAME COLUMN weight TO weight_in_kg;
90. CREATE TABLE sounds(sound_id SERIAL PRIMARY KEY);
91. There's your `sounds` table. Add a column to it named `filename`. Make it a `VARCHAR` that has a max length of `40` and with constraints of `NOT NULL` and `UNIQUE`. You can put those contraints at the end of the query to add them all.

    ALTER TABLE sounds ADD COLUMN filename VARCHAR(40) NOT NULL UNIQUE;

92. ALTER TABLE sounds ADD COLUMN character_id INT NOT NULL REFERENCES characters(character_id);
93. SELECT * FROM characters ORDER BY character_id;

    ```
    +--------------+--------+-----------------+---------------+
    | character_id |  name  |     homeland    | favorite_color |
    +--------------+--------+-----------------+---------------+
    |            1 | Mario  | Mushroom Kingdom | Red           |
    |            2 | Luigi  | Mushroom Kingdom | Green         |
    |            3 | Peach  | Mushroom Kingdom | Pink          |
    |            4 | Toad   | Mushroom Kingdom | Blue          |
    |            5 | Bowser | Koopa Kingdom    | Yellow        |
    |            6 | Daisy  | Sarasaland       | Orange        |
    |            7 | Yoshi  | Dinosaur Land    | Green         |
    +--------------+--------+-----------------+---------------+
    ```

94. INSERT INTO sounds(filename, character_id) VALUES('its-a-me.wav', 1);
95. INSERT INTO sounds(filename, character_id) VALUES('yippee.wav', 1);
96. INSERT INTO sounds(filename, character_id) VALUES('ha-ha.wav', 2);
97. INSERT INTO sounds(filename, character_id) VALUES('oh-yeah.wav', 2);
98. INSERT INTO sounds(filename, character_id) VALUES('yay.wav', 3), ('woo-hoo.wav', 3);
99. INSERT INTO sounds(filename, character_id) VALUES('mm-hmm.wav', 3), ('yahoo.wav', 1);
100. mario_database=> SELECT * FROM sounds;

    mario_database=>

    +----------+-------------+-------------+

    | sound_id | filename | character_id |

    +----------+-------------+-------------+

    | 1 | its-a-me-wav | 1 |

    | 2 | its-a-me.wav | 1 |

    | 3 | yippee.wav | 1 |

    | 4 | ha-ha.wav | 2 |

    | 5 | oh-yeah.wav | 2 |

    | 6 | yay.wav | 3 |

    | 7 | woo-hoo.wav | 3 |

    | 8 | mm-hmm.wav | 3 |

    | 9 | yahoo.wav | 1 |

    +----------+-------------+-------------+

101. CREATE TABLE actions(action_id SERIAL PRIMARY KEY);
102. ALTER TABLE actions ADD COLUMN action VARCHAR(20) NOT NULL UNIQUE;
103. The actions table won't have any foreign keys. It's going to have a "many-to-many" relationship with the characters table. This is because many of the characters can perform many actions. You will see why you don't need a foreign key later. Insert a row into the `actions` table. Give it an `action` of `run`.

    INSERT INTO actions(action) VALUES('run');

104. INSERT INTO actions(action) VALUES('jump');
105. INSERT INTO actions(action) VALUES('duck');
106. It looks good. "Many-to-many" relationships usually use a junction table to link two tables together, forming two "one-to-many" relationships. Your `characters` and `actions` table will be linked using a junction table. Create a new table called `character_actions`. It will describe what actions each character can perform.

    CREATE TABLE character_actions();

107. ALTER TABLE character_actions ADD COLUMN character_id INT NOT NULL;
108. ALTER TABLE character_actions ADD FOREIGN KEY(character_id) REFERENCES characters(character_id);
109. ALTER TABLE character_actions ADD COLUMN action_id INT NOT NULL;
110. ALTER TABLE character_actions ADD FOREIGN KEY(action_id) REFERENCES actions(action_id);
111. ALTER TABLE character_actions ADD PRIMARY KEY(character_id, action_id);
112. This table will have multiple rows with the same `character_id`, and multiple rows the same `action_id`. So neither of them are unique. But you will never have the same `character_id` and `action_id` in a single row. So the two columns together can be used to uniquely identify each row. View the details of the `character_actions` table to see your composite key.
113. INSERT INTO character_actions(character_id,action_id) VALUES(7,1),(7,2),(7,3);
114. INSERT INTO character_actions(character_id,action_id) VALUES(6,1),(6,2),(6,3);
115. INSERT INTO character_actions(character_id,action_id) VALUES(5,1),(5,2),(5,3);
116. INSERT INTO character_actions(character_id,action_id) VALUES(4,1),(4,2),(4,3);
117. INSERT INTO character_actions(character_id,action_id) VALUES(3,1),
118. INSERT INTO character_actions(character_id,action_id) VALUES(2,1),(2,2),(2,3);
119. INSERT INTO character_actions(character_id,action_id) VALUES(1,1),(1,2),(1,3);
120. You can see the `character_id` there so you just need to find the matching id in the `characters` table to find out who it's for. Or... You added that as a foreign key, that means you can get all the data from both tables with a `JOIN` command:
121. ```
SELECT columns FROM table_1 FULL JOIN table_2 ON table_1.primary_key_column = table_2.foreign_key_column;
```

    Enter a join command to see all the info from both tables. The two tables are `characters` and `more_info`. The columns are the `character_id` column from both tables since those are the linked keys.

    SELECT * FROM characters FULL JOIN more_info ON characters.character_id= more_info.character_id;

    +-------------+--------+----------------+--------------+-------------+-----------+-------------+-------------+-------------+
    | character_id | name | homeland | favorite_color | more_info_id | birthday | height_in_cm | weight_in_kg | character_id |
    +-------------+--------+----------------+--------------+-------------+-----------+-------------+-------------+-------------+
    | 2 | Luigi | Mushroom Kingdom | Green | 2 | 1983-07-14 | 175 | 48.8 | 2 |
    | 3 | Peach | Mushroom Kingdom | Pink | 3 | 1985-10-18 | 173 | 52.2 | 3 |
    | 7 | Yoshi | Dinosaur Land | Green | 7 | 1990-04-13 | 162 | 59.1 | 7 |
    | 6 | Daisy | Sarasaland | Orange | 6 | 1989-07-31 | | | 6 |
    | 1 | Mario | Mushroom Kingdom | Red | 1 | 1981-07-09 | 155 | 64.5 | 1 |
    | 4 | Toad | Mushroom Kingdom | Blue | 4 | 1950-01-10 | 66 | 35.6 | 4 |
```

| 5 | Bowser | Koopa Kingdom | Yellow | 5 | 1990-10-29 | 258 | 300.0 | 5 |

+------------+-------+----------------+--------------+------------+-----------+------------+------------+

122. SELECT * FROM characters FULL JOIN sounds ON characters.character_id= sounds.character_id;

+------------+-------+----------------+--------------+---------+------------+------------+

| character_id | name | homeland | favorite_color | sound_id | filename | character_id |

+------------+-------+----------------+--------------+---------+------------+------------+

| 1 | Mario | Mushroom Kingdom | Red | 1 | its-a-me.wav | 1 |

| 1 | Mario | Mushroom Kingdom | Red | 2 | yippee.wav | 1 |

| 2 | Luigi | Mushroom Kingdom | Green | 3 | ha-ha.wav | 2 |

| 2 | Luigi | Mushroom Kingdom | Green | 4 | oh-yeah.wav | 2 |

| 3 | Peach | Mushroom Kingdom | Pink | 5 | yay.wav | 3 |

| 3 | Peach | Mushroom Kingdom | Pink | 6 | woo-hoo.wav | 3 |

| 3 | Peach | Mushroom Kingdom | Pink | 7 | mm-hmm.wav | 3 |

| 1 | Mario | Mushroom Kingdom | Red | 8 | yahoo.wav | 1 |

| 5 | Bowser | Koopa Kingdom | Yellow | | | |

| 6 | Daisy | Sarasaland | Orange | | | |

| 4 | Toad | Mushroom Kingdom | Blue | | | |

| 7 | Yoshi | Dinosaur Land | Green | | | |

+------------+-------+----------------+--------------+---------+------------+------------+

123. This shows the "one-to-many" relationship. You can see that some of the characters have more than one row because they have many sounds. How can you see all the info from the `characters`, `actions`, and `character_actions` tables? Here's an example that joins three tables:

124. ```
SELECT columns FROM junction_table
```

125. ```
FULL JOIN table_1 ON junction_table.foreign_key_column =
table_1.primary_key_column
```

126. ```
FULL JOIN table_2 ON junction_table.foreign_key_column =
table_2.primary_key_column;
```

127.

128. Congratulations on making it this far. This is the last step. View all the data from `characters`, `actions`, and `character_actions` by joining all three tables. When you see the data, be sure to check the "many-to_many" relationship. Many characters will have many actions.

129. Get A Hint
```

130.

131.

132.