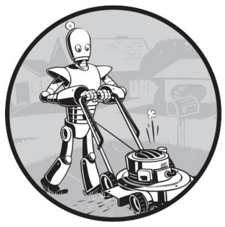
16

SQLITE DATABASES



You’re probably used to organizing information into spreadsheets such as Excel or Google Sheets, but most software stores its data in applications called *databases*. Databases make it easy for your programs to retrieve the specific data you want. If you had a spreadsheet or text file of cats and wanted to find the fur color of a cat named Zophie, you could press CTRL-F and enter “Zophie.” But what if you wanted to find the fur color of all cats that weighed between 3 and 5 kilograms and were born before October 2023? Even with the regular expressions from Chapter 9, this would be a tricky thing to code.

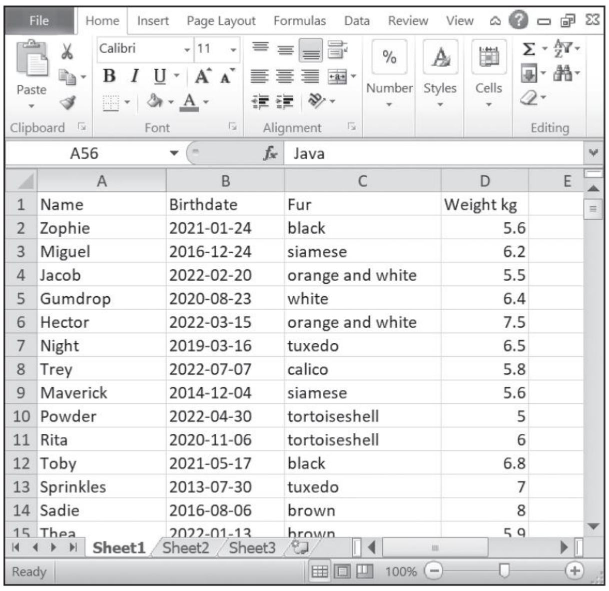
Databases allow you to perform complex queries like this one, written in the mini language of *Structured Query Language (SQL)*. You’ll see the term *SQL* used to refer to both a language for database operations and the databases that understand this language; it’s often pronounced “es-cue-el” but also sometimes “sequel.” This chapter introduces you to SQL and database concepts using *SQLite* (pronounced either “sequel-ite,” “es-cue-lite,” or “es-cue-el-ite”), a lightweight database included with Python.

SQLite is the most widely deployed database software, as it runs on every operating system and is small enough to embed within other applications. At the same time, SQLite’s simplifications make it notably different from other databases. While large database software such as PostgreSQL, MySQL, Microsoft SQL Server, and Oracle are intended to run on dedicated server hardware accessed over a network, SQLite stores the entire database in a single file on your computer.

Even if you’re already familiar with SQL databases, SQLite has enough of its own quirks that you should read this chapter to learn how to make the most of it. You can find the online SQLite documentation at *https://sqlite.org/docs.html* and the Python `sqlite3` module documentation at *https://docs.python.org/3/library/sqlite3.html*.

Spreadsheets vs. Databases

Let's consider the similarities and differences between spreadsheets and databases. In a spreadsheet, rows contain individual records, while columns represent the kind of data stored in the fields of each record. For example, Figure 16-1 is a spreadsheet of some of my cats. The columns list the name, birthday, fur color, and weight (in kilograms) of each cat.



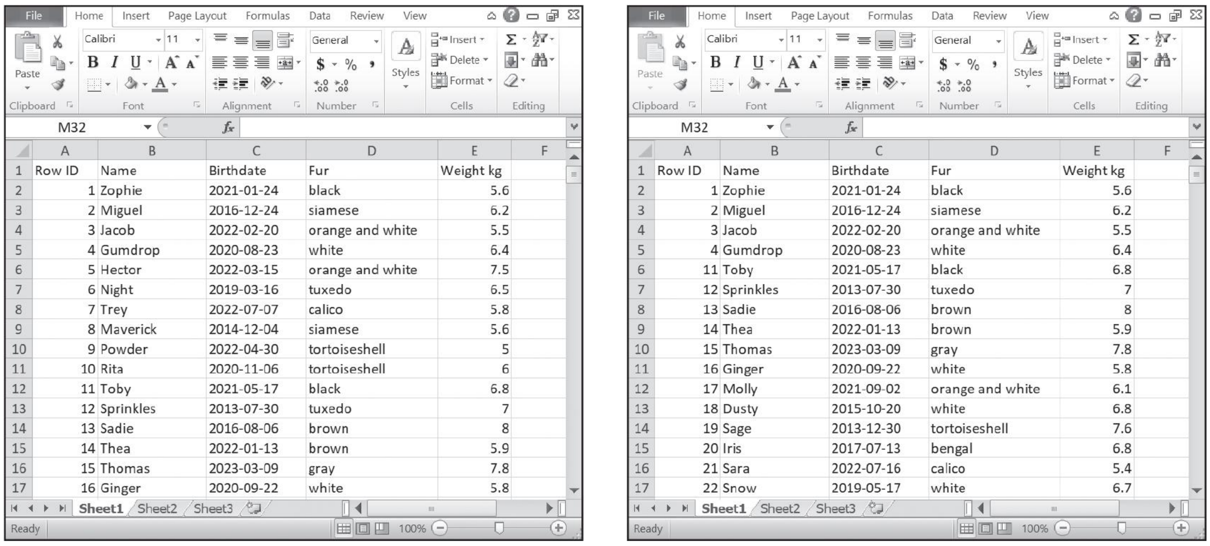
	A	B	C	D	E
1	Name	Birthdate	Fur	Weight kg	
2	Zophie	2021-01-24	black	5.6	
3	Miguel	2016-12-24	siamese	6.2	
4	Jacob	2022-02-20	orange and white	5.5	
5	Gumdrop	2020-08-23	white	6.4	
6	Hector	2022-03-15	orange and white	7.5	
7	Night	2019-03-16	tuxedo	6.5	
8	Trey	2022-07-07	calico	5.8	
9	Maverick	2014-12-04	siamese	5.6	
10	Powder	2022-04-30	tortoiseshell	5	
11	Rita	2020-11-06	tortoiseshell	6	
12	Toby	2021-05-17	black	6.8	
13	Sprinkles	2013-07-30	tuxedo	7	
14	Sadie	2016-08-06	brown	8	
15	Thea	2022-01-13	brown	5.9	

Figure 16-1: A spreadsheet stores data records as rows with a set column structure.

We can store this same information in a database. You can think of a database *table* as a spreadsheet, and a database can contain one or more tables. Tables have columns of different properties for each *record*, also called a *row* or *entry*. Databases like SQLite are called *relational databases*, where *relation* means that the database can contain multiple tables with relationships between them, as you'll later see.

Both spreadsheets and databases label the data they contain. A spreadsheet automatically labels the columns with letters and the rows with numbers. In addition, the example cat spreadsheet uses its first row to give the columns descriptive names. Each of the subsequent rows represents exactly one cat. In a SQL database, tables often have an ID column for each record's *primary key*: a unique integer that can unambiguously identify the record. In SQLite, this column is called `rowid`, and SQLite automatically adds it to your tables.

Deleting a spreadsheet row moves up all the rows underneath it, changing their row numbers. But a database record's primary key ID is unique and doesn't change. This is useful in many situations. What if a cat were renamed or had a change in weight? What if we wanted to reorder the rows to list the cats alphabetically by name? Each cat needs a unique identification number that remains constant no matter how the data changes. We could add a Row ID column to our spreadsheet to simulate a SQLite table's `rowid` column. This ID value would stay the same even if rows were deleted or moved around the spreadsheet, as shown in Figure 16-2, where the cats with the Row IDs of 5 to 10 are deleted.

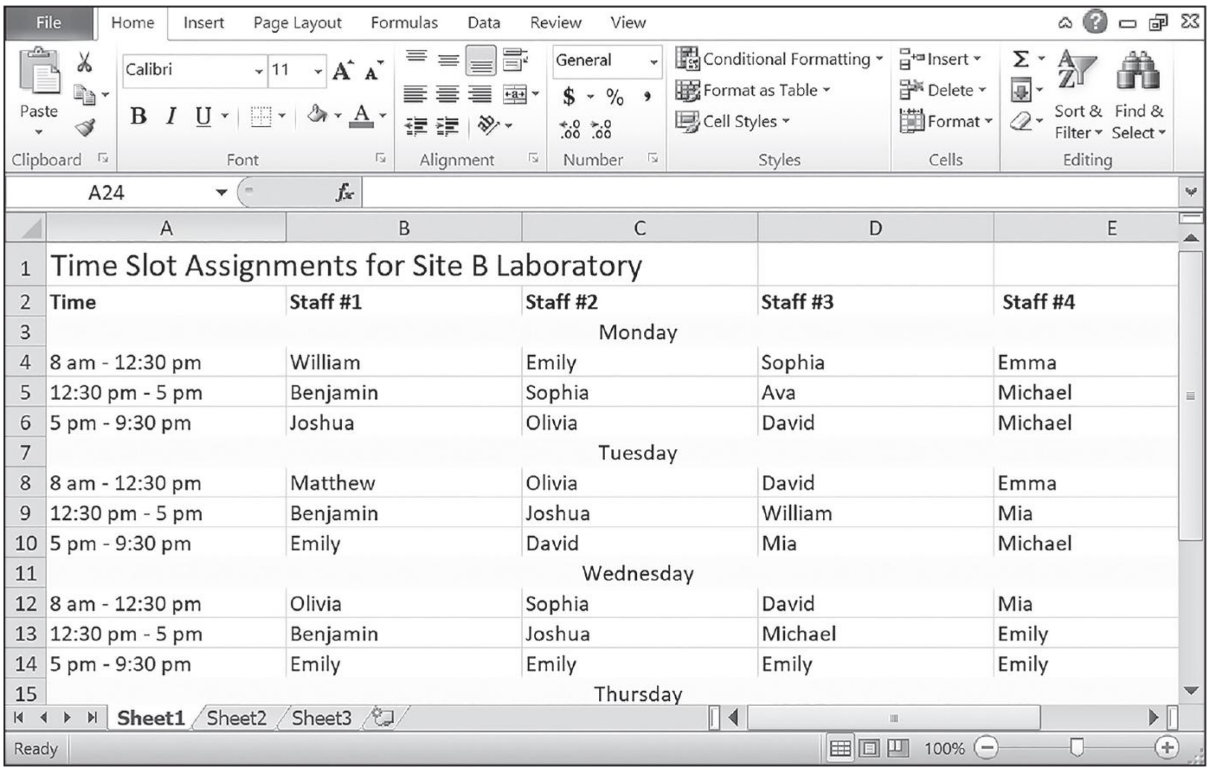


	A	B	C	D	E	F
1	Row ID	Name	Birthdate	Fur	Weight kg	
2	1	Zophie	2021-01-24	black	5.6	
3	2	Miguel	2016-12-24	siamese	6.2	
4	3	Jacob	2022-02-20	orange and white	5.5	
5	4	Gumdrop	2020-08-23	white	6.4	
6	5	Hector	2022-03-15	orange and white	7.5	
7	6	Night	2019-03-16	tuxedo	6.5	
8	7	Trey	2022-07-07	calico	5.8	
9	8	Maverick	2014-12-04	siamese	5.6	
10	9	Powder	2022-04-30	tortoiseshell	5	
11	10	Rita	2020-11-06	tortoiseshell	6	
12	11	Toby	2021-05-17	black	6.8	
13	12	Sprinkles	2013-07-30	tuxedo	7	
14	13	Sadie	2016-08-06	brown	8	
15	14	Thea	2022-01-13	brown	5.9	
16	15	Thomas	2023-03-09	gray	7.8	
17	16	Ginger	2020-09-22	white	5.8	

	A	B	C	D	E	F
1	Row ID	Name	Birthdate	Fur	Weight kg	
2	1	Zophie	2021-01-24	black	5.6	
3	2	Miguel	2016-12-24	siamese	6.2	
4	3	Jacob	2022-02-20	orange and white	5.5	
5	4	Gumdrop	2020-08-23	white	6.4	
6	11	Toby	2021-05-17	black	6.8	
7	12	Sprinkles	2013-07-30	tuxedo	7	
8	13	Sadie	2016-08-06	brown	8	
9	14	Thea	2022-01-13	brown	5.9	
10	15	Thomas	2023-03-09	gray	7.8	
11	16	Ginger	2020-09-22	white	5.8	
12	17	Molly	2021-09-02	orange and white	6.1	
13	18	Dusty	2015-10-20	white	6.8	
14	19	Sage	2013-12-30	tortoiseshell	7.6	
15	20	Iris	2017-07-13	bengal	6.8	
16	21	Sara	2022-07-16	calico	5.4	
17	22	Snow	2019-05-17	white	6.7	

Figure 16-2: The Row ID number, unlike the spreadsheet row numbers, offers a unique identifier for each record (left) even after cats with IDs 5 to 10 are deleted (right).

There is a second way people use spreadsheets that is entirely unlike how databases tend to store data. Spreadsheets can serve as templates for forms rather than as row-based data storage. You may have seen spreadsheets such as Figure 16-3.



	A	B	C	D	E
1	Time Slot Assignments for Site B Laboratory				
2	**Time**	**Staff #1**	**Staff #2**	**Staff #3**	**Staff #4**
3			Monday		
4	8 am - 12:30 pm	William	Emily	Sophia	Emma
5	12:30 pm - 5 pm	Benjamin	Sophia	Ava	Michael
6	5 pm - 9:30 pm	Joshua	Olivia	David	Michael
7			Tuesday		
8	8 am - 12:30 pm	Matthew	Olivia	David	Emma
9	12:30 pm - 5 pm	Benjamin	Joshua	William	Mia
10	5 pm - 9:30 pm	Emily	David	Mia	Michael
11			Wednesday		
12	8 am - 12:30 pm	Olivia	Sophia	David	Mia
13	12:30 pm - 5 pm	Benjamin	Joshua	Michael	Emily
14	5 pm - 9:30 pm	Emily	Emily	Emily	Emily
15			Thursday		

Figure 16-3: A spreadsheet with a lot of formatting and a fixed sized is generally not a good fit for a database.

These spreadsheets tend to have a lot of formatting, with background colors, merged cells, and different fonts, so that they look good to human eyes. While the row-based data spreadsheets can expand infinitely downward as new data is added, these spreadsheets usually have a fixed size and fill-in-the-blank design. They're often meant for humans to print out and look at, rather than for a Python program to extract data from them.

Databases aren't visually pleasing; they just contain raw data. More importantly, while spreadsheets give you the flexibility of putting any data into any cell, databases have a stricter structure to make data retrieval easier for software. If your data tends to look like the example in Chapter 14 and the EZSheets library from Chapter 15 and leaving it in an Excel or Google spreadsheet.

SQLite vs. Other SQL Databases

If you're used to working with other SQL databases, you might be wondering how SQLite compares. In short, SQLite strikes a balance between simplicity and power. It's a full relational database that uses SQL to read and write massive amounts of data, but it runs within your Python program and operates on a single file. Your program imports the `sqlite3` module just as it would import `sys`, `math`, or any other module in the Python standard library.

Here are the main differences between SQLite and other database software:

- SQLite databases are stored in a single file, which you can move, copy, or back up like any other file.
- SQLite can run on computers with few resources, such as embedded devices or decades-old laptops.
- SQLite is serverless; it doesn't require a background server application to constantly run on your laptop, or any dedicated server hardware. There are no network connections involved.
- From the perspective of users, SQLite doesn't require any installation or configuration. It's part of the Python program.
- For faster performance, SQLite databases can exist entirely in memory and be saved to a file before the program exits.
- While SQLite columns have data types, such as numbers and text, just as other SQL databases do, SQLite doesn't strictly enforce a column's data type.
- There are no permission settings or user roles in SQLite. SQLite has no `GRANT` or `REVOKE` statements like in other SQL databases.
- SQLite is public domain software; you can use it commercially or any way you want without restriction.

The main disadvantage of SQLite is that it can't efficiently handle hundreds or thousands of simultaneous write operations (say, from a social media web app). Aside from that, SQLite is just as powerful as any database, able to reliably handle GBs or even TBs of data, as well as simultaneous read operations, quickly and easily.

SQLite sells itself not so much as a competitor to other database software but as a competitor to using the `open()` function to work with text files (or the JSON, XML, and CSV files you'll learn about in Chapter 18). If your program needs the ability to store and quickly retrieve large amounts of data, SQLite is a better alternative to JSON or spreadsheet files.

Creating Databases and Tables

Let's begin by creating our first database and table using SQL. SQL is a mini language you can work with from within Python, much like regex for regular expressions. Like regex, SQL queries are written as Python string values. And just as you could write your own Python code to perform the text pattern-matching that regular expressions perform, you *could* write your own custom Python code to search for matching data in Python dictionaries and lists. But writing regular expressions and SQL database queries makes these tasks much simpler in the long run, even if they first require you to learn a new skill. Let's explore how to write queries that create tables in a new database.

We'll create a sample SQLite database in a file named *example.db* to store information about cats. To create a database, first import the `sqlite3` module. (The *3* is for SQLite major version 3, which is unrelated to Python 3.) A SQLite database lives in a single file. The name of the file can be anything, but by convention we give it a *.db* file extension. The extension *.sqlite* is also commonly used.

A database can contain multiple tables, and each table should store one particular type of data. For example, one table could contain records of cats, while another table could contain records of vaccinations given to specific cats in the first table. You can think of a table as a list of tuples, where each tuple is a table row. The `cats` table is essentially the same as `[('Zophie', '2021-01-24', 'black', 5.6), ('Colin', '2016-12-24', 'siamese', 6.2), ...]`, and so on.

Let's create a database, then create a table for the cat data, insert some cat records into it, read the data from the database, and close the database connection.

Connecting to Databases

The first step of writing SQLite code is getting a `Connection` object for the database file by calling `sqlite3.connect()`. Enter the following into the interactive shell:

```
>>> import sqlite3
>>> conn = sqlite3.connect('example.db',
isolation_level=None)
```

The first argument to the function can be either a string of a filename or a `pathlib.Path` object for the database file. If this filename doesn't belong to an existing SQLite database, the function creates a new file containing an empty database. For example, `sqlite3.connect('example.db', isolation_level=None)` connects a database file named *example.db* in the current working directory. If this file doesn't exist, the function creates an empty one.

If the file you connect to exists but isn't a SQLite database file, Python raises a `sqlite3.DatabaseError: file is not a database` exception once you try to execute queries. "Checking Path Validity" in Chapter 10 explains how to use the `exists() Path` method and `os.path.exists()` function, which can tell your program if a file exists or not.

The `isolation_level=None` keyword argument causes the database to use autocommit mode. This relieves you of having to write `commit()` method calls after each `execute()` method call.

The `sqlite3.connect()` function returns a `Connection` object, which we store in a variable named `conn` for these examples. Each `Connection` object connects to one SQLite database file. You can, of course, choose any variable name you'd like for this `Connection` object, and you should use more descriptive variable names if your program opens multiple databases at the same time. But for small programs that connect to only one database at a time, the name `conn` is easy to write and descriptive enough. (The name `con` would be even shorter, but is easy to misunderstand as "*con*sole," "*con*tent," or "*con*fusing name for a variable.")

When your program is done with the database, call `conn.close()` to close the connection. The program also closes the connection automatically when it terminates.

Creating Tables

After connecting to a new, blank database, create a table with a `CREATE TABLE` SQL query. To run SQL queries, you must call the `execute()` method of `Connection` objects. Pass this `conn.execute()` method a string of the query:

```
>>> import sqlite3
>>> conn = sqlite3.connect('example.db',
isolation_level=None)
>>> conn.execute('CREATE TABLE IF NOT EXISTS
cats (name TEXT NOT NULL,
birthdate TEXT, fur TEXT, weight_kg REAL)
STRICT')
<sqlite3.Cursor object at 0x00000201B2399540>
```

By convention, SQL keywords, such as `CREATE` and `TABLE`, are written using uppercase letters. However, SQLite doesn't enforce this; the query `'create table if not exists cats (name text not null, birthdate text, fur text, weight_kg real) strict'` runs just fine. Table and column names are also case-insensitive, but the convention is to make them lowercase and to separate multiple words with underscores, as in `weight_kg`.

The `CREATE TABLE` statement raises a `sqlite3.OperationalError: table cats already exists` exception if you try to create a table that already exists without the `IF NOT EXISTS` part. Including this part is a quick way to avoid tripping over this exception, and you'll almost always want to add it to your `CREATE TABLE` queries.

In our example, we follow the `CREATE TABLE IF NOT EXISTS` keywords with the table name `cats`. After the table name is a set of parentheses containing the column names and data types.

Defining Data Types

There are six data types in SQLite:

`NULL` Analogous to Python's `None`

`INT` or `INTEGER` Analogous to Python's `int` type

`REAL` A reference to the mathematics term *real number*; analogous to Python's `float` type

`TEXT` Analogous to Python's `str` type

`BLOB` Short for *Binary Large Object*; analogous to Python's `bytes` type and useful for storing entire files in a database

SQLite has its own data types because it wasn't built just for Python; other programming languages can interact with SQLite databases as well.

Unlike other SQL database software, SQLite isn't strict about the data types of its columns. This means SQLite will, by default, gladly store the string `'Hello'` in an `INTEGER` column without raising an exception. But SQLite's data types aren't entirely cosmetic, either; SQLite automatically *casts* (that is, changes) data to the column's data type if possible, a feature called *type affinity*. For example, if you add the string `'42'` to an `INTEGER` column, SQLite automatically stores the value as the integer `42`, because the column has a type affinity for integers. However, if you add the string `'Hello'` to an `INTEGER` column, SQLite will store `'Hello'` (without error), because despite the integer type affinity, `'Hello'` cannot be converted to an integer.

The `STRICT` keyword enables *strict mode* for this table. Under strict mode, every column must be given a data type, and SQLite will raise a `sqlite3.IntegrityError` exception if you try to insert data of the wrong type into the table. SQLite will still automatically cast data to the column's data type; inserting `'42'` into an `INTEGER` column would insert the integer `42`. However, the string `'Hello'` cannot be cast to an integer, so attempting to insert it would raise an exception. I highly recommend using strict mode; it can give you an early warning about bugs caused by inserting incorrect data into your table.

SQLite added the `STRICT` keyword in version 3.37.0, which is used by Python 3.11 and later. Earlier versions don't know about strict mode and will report a syntax error if you attempt to use it. You can always check the version of SQLite that Python is using by examining the `sqlite3.sqlite _version` variable, which will look something like this:

```
>>> import sqlite3
>>> sqlite3.sqlite_version
'3.xx.xx'
```

SQLite doesn't have a Boolean data type, so use `INTEGER` for Boolean data instead; you can store a `1` to represent `True` and a `0` to represent `False`. SQLite also doesn't have a date, time, or datetime data type. Instead, you can use the `TEXT` data type to store a string in a format listed in Table 16-1.

Table 16-1: Recommended Formats for Dates, Times, and Datetimes in SQLite

Format	Example
`YYYY-MM-DD`	`'2035-10-31'`
`YYYY-MM-DD HH:MM:SS`	`'2035-10-31 16:30:00'`
`YYYY-MM-DD HH:MM:SS.SSS`	`'2035-10-31 16:30:00.407'`
`HH:MM:SS`	`'16:30:00'`
`HH:MM:SS.SSS`	`'16:30:00.407'`

The `NOT NULL` part of `name TEXT NOT NULL` specifies that the Python `None` value cannot be stored in the `name` column. This is a good way to make a table column mandatory.

SQLite tables automatically create a `rowid` column containing a unique primary key integer. Even if your `cats` table has two cats that coincidentally have the same name, birthday, fur color, and weight, the `rowid` allows you to distinguish between them.

Listing Tables and Columns

All SQLite databases have a table named `sqlite_schema` that lists metadata about the database, including all of its tables. To list the tables in the SQLite database, run the following query:

```
>>> import sqlite3
>>> conn = sqlite3.connect('example.db',
isolation_level=None)
>>> conn.execute('SELECT name FROM
sqlite_schema WHERE type="table"').fetchall()
[('cats',)]
```

The output shows the `cats` table we just created. (I explain the syntax of the `SELECT` statement in “Reading Data from the Database” on page 394.) To obtain information about the columns in the `cats` table, run the following query:

```
>>> import sqlite3
>>> conn = sqlite3.connect('example.db',
isolation_level=None)
>>> conn.execute('PRAGMA
TABLE_INFO(cats)').fetchall()
[(0, 'name', 'TEXT', 1, None, 0), (1,
'birthdate', 'TEXT', 0, None, 0), (2,
'fur', 'TEXT', 0, None, 0), (3, 'weight_kg',
'REAL', 0, None, 0)]
```

This query returns a list of tuples that each describe a column of the table. For example, the `(1, 'birthdate', 'TEXT', 0, None, 0)` tuple provides the following information about the `birthdate` column:

Column position The `1` indicates that the column is second in the table. Column numbers are zero based, like Python list indexes, so the first column is at position 0.

Name `'birthdate'` is the name of the column. Remember that SQLite column and table names are case insensitive.

Data type `'TEXT'` is the SQLite data type of the `birthdate` column.

Whether the column is `NOT NULL` The `0` means `False` and that the column is not `NOT NULL` (that is, you can put `None` values in this column).

Default value `None` is the default value inserted if no other value is specified.

Whether the column is the primary key The `0` means `False`, meaning this column is not a primary-key column.

Note that the `sqlite_schema` table itself isn't listed as a table. You'll never need to modify the `sqlite_schema` table yourself, and doing so will likely corrupt the database, making it unreadable.

CRUD Database Operations

CRUD is an acronym for the four basic operations that databases carry out: creating data, reading data, updating data, and deleting data. In SQLite, we perform these operations with `INSERT`, `SELECT`, `UPDATE`, and `DELETE` statements, respectively. Here are examples of each statement, which we'll later pass as strings to `conn.execute()`:

- `INSERT INTO cats VALUES ("Zophie", "2021-01-24", "black", 5.6)`
- `SELECT rowid, * FROM cats ORDER BY fur`
- `UPDATE cats SET fur = "gray tabby" WHERE rowid = 1`
- `DELETE FROM cats WHERE rowid = 1`

Most applications and social media websites are really just fancy user interfaces for a CRUD database. When you post a photo or reply, you're creating a record in a database somewhere. When you scroll through a social media timeline, you're reading records from the database. And when you edit or delete a post, you're performing an update or a deletion operation. Whenever you're learning a new app, programming language, or query language, use the CRUD acronym to remind yourself of which basic operations you should find out about.

Inserting Data into the Database

Now that we've created the database and a `cats` table, let's insert records for my pet cats. I have about 300 cats in my home, and using a SQLite database helps me keep track of them. An `INSERT` statement can add new records to a table. Enter the following code into the interactive shell:

```
>>> import sqlite3
>>> conn = sqlite3.connect('example.db',
isolation_level=None)
>>> conn.execute('CREATE TABLE IF NOT EXISTS
cats (name TEXT NOT NULL, birthdate TEXT,
fur TEXT, weight_kg REAL) STRICT')
<sqlite3.Cursor object at 0x00000201B2399540>
>>> conn.execute('INSERT INTO cats VALUES
("Zophie", "2021-01-24", "black", 5.6)')
<sqlite3.Cursor object at 0x00000162842E78C0>
```

This `INSERT` query adds a new row to the `cats` table. Inside the parentheses are the comma-separated values for its columns. The parentheses are mandatory for `INSERT` queries. Note that when inserting `TEXT` values, I've used double quotation marks (`"`) because I'm already using single quotation marks (`'`) for the query's string. The `sqlite3` module uses either single or double quotes for its `TEXT` values.

Transactions

An `INSERT` statement begins a *transaction*, which is a unit of work in a database. Transactions must pass the *ACID test*, a database concept meaning that transactions are:

Atomic The transaction is carried out either completely or not at all.

Consistent The transaction doesn't violate constraints, such as `NOT NULL` rules for columns.

Isolated One transaction doesn't affect other transactions.

Durable If committed, the transaction results are written to persistent storage, such as the hard drive.

SQLite is an ACID-compliant database; it has even passed tests that simulate the computer losing power in the middle of a transaction, so you have high assurance that the database file won't be left in a corrupt, unusable state. A SQLite query will either completely insert data into the database or not insert it at all.

SQL Injection Attacks

A category of hacking techniques called *SQL injection attacks* can change your queries to do things you didn't intend. These techniques are beyond the scope of this book, and they mostly likely are not an issue for your code if your program isn't accepting data from strangers on the internet. But to prevent them, use the `?` question mark syntax whenever you reference variables when inserting or updating data in your database.

For example, if I want to insert a new cat record based on data stored in variables, I shouldn't insert these variables directly into the query string using Python, like this:

```
>>> cat_name = 'Zophie'
>>> cat_bday = '2021-01-24'
>>> fur_color = 'black'
>>> cat_weight = 5.6
>>> conn.execute(f'INSERT INTO cats VALUES
("{cat_name}", "{cat_bday}",
"{fur_color}", {cat_weight})')
<sqlite3.Cursor object at 0x0000022B91BB7C40>
```

If the values of these variables came from user input such as a web app form, a hacker could potentially specify strings that changed the meaning of the query. Instead, I should use a `?` in the query string, then pass the variables in a list argument following the query string:

```
>>> conn.execute('INSERT INTO cats VALUES
(?, ?, ?, ?)', [cat_name, cat_bday,
fur_color, cat_weight])
<sqlite3.Cursor object at 0x0000022B91BB7C40>
```

The `execute()` method replaces the `?` placeholders in the query string with the variable values after making sure they won't cause a SQL injection attack. While such attacks are unlikely to apply to your code, it's a good habit to use the `?` placeholders instead of formatting the query string yourself.

Reading Data from the Database

Once there's data inside the database, you can read it with a `SELECT` query. Enter the following into the interactive shell to read data from the *example.db* database:

```
>>> import sqlite3
>>> conn = sqlite3.connect('example.db',
isolation_level=None)
>>> conn.execute('SELECT * FROM
cats').fetchall()
[('Zophie', '2021-01-24', 'black', 5.6)]
```

The `execute()` method call for the `SELECT` query returns a `Cursor` object. To obtain the actual data, we call the `fetchall()` method on this `Cursor` object. Each record is returned as a tuple in the list of tuples. The data in each tuple appears in the order of the table's columns.

Instead of writing Python code to sort through this list of tuples yourself, you can make SQLite extract the specific information you want. The example `SELECT` query has four parts:

- The `SELECT` keyword
- The columns you want to retrieve, where `*` means "all columns except `rowid`"
- The `FROM` keyword
- The table to retrieve data from; in this case, the `cats` table

If you wanted just the `rowid` and `name` columns of records in the `cats` table, your query would look like this:

```
>>> conn.execute('SELECT rowid, name FROM
cats').fetchall()
[(1, 'Zophie')]
```

You can also use SQL to filter the query results, as you'll learn in the next section.

Looping over Query Results

The `fetchall()` method returns your `SELECT` query results as a list of tuples. A common coding pattern is to use this data in a `for` loop to perform some operation for each tuple. For example, download the *sweigartcats.db* file from *https://nostarch.com/automate-boring-stuff-python-3rd-edition*, then enter the following into the interactive shell to process its data:

```
>>> import sqlite3
>>>
conn = sqlite3.connect('sweigartcats.db',
isolation_level=None)
>>> for row in conn.execute('SELECT * FROM
cats'):
...     print('Row data:', row)
...     print(row[0], 'is one of my favorite
cats.')
...
Row data: ('Zophie', '2021-01-24', 'gray
tabby', 5.6)
Zophie is one of my favorite cats.
Row data: ('Miguel', '2016-12-24', 'siamese',
6.2)
Miguel is one of my favorite cats.
Row data: ('Jacob', '2022-02-20', 'orange and
white', 5.5)
Jacob is one of my favorite cats.
--snip--
```

The `for` loop can iterate over the tuples of row data returned by `conn.execute()` without calling `fetchall()`, and the code in the body of the `for` loop can operate on each row individually, because the `row` variable populates with a tuple of row data from the query. The code can then access the columns using the tuple's integer index: index `0` for the name, index `1` for the birthdate, and so on.

Filtering Retrieved Data

Our `SELECT` queries have been retrieving every row in the table, but we might want just the rows that match some filter criteria. Using the *sweigartcats.db* file, add a `WHERE` clause to the `SELECT` statement to provide search parameters, such as having black fur:

```
>>> import sqlite3
>>>
```

```
conn = sqlite3.connect('sweigartcats.db',
isolation_level=None)
>>> conn.execute('SELECT * FROM cats WHERE
fur = "black"').fetchall()
[❶('Zophie', '2021-01-24', 'black', 5.6),
('Toby', '2021-05-17', 'black',
6.8), ('Thor', '2013-05-14', 'black', 5.2),
('Sassy', '2017-08-20', 'black',
7.5), ('Hope', '2016-05-22', 'black', 7.6)]
```

In this example, the `WHERE` clause `WHERE fur = "black"` will retrieve data only for records that have `"black"` in the `fur` column.

SQLite defines its own operators for use in the `WHERE` clause, but they're similar to Python's operators: =, !=, <, >, <=, >=, `AND`, `OR`, and `NOT`. Note that SQLite uses the = operator to mean "is equal to," while Python uses the == operator for that purpose. On either side of the operator, you can put a column name or a literal value.

The comparison will occur for each row in the table. For example, for `WHERE fur = "black"`, SQLite makes the following comparisons:

- Because `fur` is `'black'` and `'black' = 'black'` is true, SQLite includes the row at ❶ in the results.
- For the row `(2, 'Miguel', '2016-12-24', 'siamese', 6.2)`, `fur` is `'siamese'` and `'siamese' = 'black'` is false, so it doesn't include the row in the results.
- For the row `(3, 'Jacob', '2022-02-20', 'orange and white', 5.5)`, `fur` is `'orange and white'` and `'orange and white' = 'black'` is false, so it doesn't include the row in the results.

... and so on, for every row in the `cats` table.

Let's continue the previous example with a more complicated `WHERE` clause: `WHERE fur = "black" OR birthdate >= "2024-01-01"'`. Let's also use the `pprint.pprint()` function to "pretty print" the returned list:

```
>>> import pprint
>>> matching_cats = conn.execute('SELECT *
FROM cats WHERE fur = "black"
OR birthdate >= "2024-01-01"').fetchall()
>>> pprint.pprint(matching_cats)
[('Zophie', '2021-01-24', 'black', 5.6),
 ('Toby', '2021-05-17', 'black', 6.8),
 ('Taffy', '2024-12-09', 'white', 7.0),
 ('Hollie', '2024-08-07', 'calico', 6.0),
 ('Lewis', '2024-03-19', 'orange tabby',
5.1),
 ('Thor', '2013-05-14', 'black', 5.2),
 ('Shell', '2024-06-16', 'tortoiseshell',
6.5),
 ('Jasmine', '2024-09-05', 'orange tabby',
6.3),
 ('Sassy', '2017-08-20', 'black', 7.5),
 ('Hope', '2016-05-22', 'black', 7.6)]
```

All of the cats in the resulting `matching_cats` list have either black fur or a birthdate after January 1, 2024. Note that the birthdate is just a string. While comparison operators like >= typically perform alphabetical comparisons on strings, they can also perform temporal comparisons, as long as the birthdate format is *YYYY-MM-DD*.

The `LIKE` operator lets you match just the start or end of a value, treating the percent sign (`%`) as a wildcard. For example, `name LIKE "%y"` matches all the names that end with `'y'`, while `name LIKE "Ja%"` matches all the names that start with `'Ja'`:

```
>>> conn.execute('SELECT rowid, name FROM
cats WHERE name LIKE "%y"').fetchall()
[(5, 'Toby'), (11, 'Molly'), (12, 'Dusty'),
(17, 'Mandy'), (18, 'Taffy'), (25, 'Rocky'),
(27,
'Bobby'), (30, 'Misty'), (34, 'Mitsy'), (38,
'Colby'), (40, 'Riley'), (46, 'Ruby'), (65,
'Daisy'), (67, 'Crosby'), (72, 'Harry'), (77,
'Sassy'), (85, 'Lily'), (93, 'Spunky')]
>>> conn.execute('SELECT rowid, name FROM
cats WHERE name LIKE "Ja%"').fetchall()
[(3, 'Jacob'), (49, 'Java'), (75, 'Jasmine'),
(80, 'Jamison')]
```

You can also put percent signs at the start and end of a string to match text anywhere in the middle. For example, `name LIKE "%ob%"` matches all names that have `'ob'` anywhere in them:

```
>>> conn.execute('SELECT rowid, name FROM
cats WHERE name LIKE "%ob%"').fetchall()
[(3, 'Jacob'), (5, 'Toby'), (27, 'Bobby')]
```

The `LIKE` operator does a case-insensitive match, so `name LIKE "%ob%"` also matches `'%OB%'`, `'%Ob%'`, and `'%oB%'`. To do a case-sensitive match, use the `GLOB` operator and `*` as the wildcard characters:

```
>>> conn.execute('SELECT rowid, name FROM
cats WHERE name GLOB "*m*"').fetchall()
[(4, 'Gumdrop'), (9, 'Thomas'), (44, 'Sam'),
(63, 'Cinnamon'), (75, 'Jasmine'),
(79, 'Samantha'), (80, 'Jamison')]
```

While `name LIKE "%m%"` matches either a lowercase or uppercase *m*, `name GLOB "*m*"` matches only the lowercase *m*.

SQLite's wide set of operators and functionality rivals that of any full programming language. You can read more about it in the SQLite documentation at *https://www.sqlite.org/lang_expr.html*.

Ordering the Results

While you can always sort the list returned by `fetchall()` by calling Python's `sort()` method, it's easier to have SQLite sort the data for you by adding an `ORDER BY` clause to your `SELECT` query. For example, if I wanted to sort the cats by fur color, I could enter the following:

```
>>> import sqlite3, pprint
>>>
conn = sqlite3.connect('sweigartcats.db',
isolation_level=None)
>>>
pprint.pprint(conn.execute('SELECT * FROM
cats ORDER BY fur').fetchall())
[('Iris', '2017-07-13', 'bengal', 6.8),
 ('Ruby', '2023-12-22', 'bengal', 5.0),
 ('Elton', '2020-05-28', 'bengal', 5.4),
 ('Toby', '2021-05-17', 'black', 6.8),
 ('Thor', '2013-05-14', 'black', 5.2),
--snip--
 ('Celine', '2015-04-18', 'white', 7.3),
 ('Daisy', '2019-03-19', 'white', 6.0)]
```

If there is a `WHERE` clause in your query, the `ORDER BY` clause must come after it. You can also order the rows based on multiple columns. For example, if you want to first sort the rows by fur color and then sort the rows within each fur color by birthdate, run the following:

```
>>> cur = conn.execute('SELECT * FROM cats
ORDER BY fur, birthdate')
>>> pprint.pprint(cur.fetchall())
[('Iris', '2017-07-13', 'bengal', 6.8),
 ('Elton', '2020-05-28', 'bengal', 5.4),
 ('Ruby', '2023-12-22', 'bengal', 5.0),
 ('Thor', '2013-05-14', 'black', 5.2),
 ('Hope', '2016-05-22', 'black', 7.6),
--snip--
 ('Ginger', '2020-09-22', 'white', 5.8),
 ('Taffy', '2024-12-09', 'white', 7.0)]
```

The ORDER BY clause lists the fur column first, followed by the birthdate column, separated by a comma. By default, these sorts are in ascending order: the smallest values come first, followed by larger values. To sort in descending order, add the DESC keyword after the column name. You can also use the ASC keyword to specify ascending order if you want your query to be explicit and readable. To practice using these keywords, enter the following into the interactive shell:

```
>>> cur = conn.execute('SELECT * FROM cats
ORDER BY fur ASC, birthdate DESC')
>>> pprint.pprint(cur.fetchall())
[('Ruby', '2023-12-22', 'bengal', 5.0),
 ('Elton', '2020-05-28', 'bengal', 5.4),
 ('Iris', '2017-07-13', 'bengal', 6.8),
 ('Toby', '2021-05-17', 'black', 6.8),
 ('Sassy', '2017-08-20', 'black', 7.5),
--snip--
 ('Mitsy', '2015-05-29', 'white', 5.0),
 ('Celine', '2015-04-18', 'white', 7.3)]
```

The output lists the cats by fur color in ascending order (with 'bengal' coming before 'white'). Within each fur color, the cats are sorted by birthdate in descending order (with '2023-12-22' coming before '2020-05-28').

Limiting the Number of Results

If you're interested in viewing only the first few rows returned by your SELECT query, you might try to use Python list slices to limit the results. For example, use the [:3] slice to show only the first three rows in the cats table:

```
>>> import sqlite3
>>>
conn = sqlite3.connect('sweigartcats.db',
isolation_level=None)
>>> conn.execute('SELECT * FROM
cats').fetchall()[:3]  # This is inefficient.
```

```
[('Zophie', '2021-01-24', 'gray tabby', 5.6),
('Miguel', '2016-12-24',
'siamese', 6.2), ('Jacob', '2022-02-20',
'orange and white', 5.5)]
```

This code works, but it’s inefficient; it first fetches all of the rows from the table, then discards everything except for the first three. It would be faster for your program to fetch just the first three rows from the database. You can do this with a `LIMIT` clause:

```
>>> conn.execute('SELECT * FROM cats LIMIT
3').fetchall()
[('Zophie', '2021-01-24', 'gray tabby', 5.6),
('Miguel', '2016-12-24',
'siamese', 6.2), ('Jacob', '2022-02-20',
'orange and white', 5.5)]
```

This code runs faster than the code that fetches all the rows, especially for tables with a large number of rows. The `LIMIT` clause must come after the `WHERE` and `ORDER BY` clauses if your `SELECT` query includes them, as in the following example:

```
>>> conn.execute('SELECT * FROM cats WHERE
fur="orange" ORDER BY birthdate LIMIT
4').fetchall()
[('Mittens', '2013-07-03', 'orange', 7.4),
('Piers', '2014-07-08', 'orange', 5.2),
('Misty', '2016-07-08', 'orange', 5.2),
('Blaze', '2023-01-16', 'orange', 7.4)]
```

There are a few other clauses you can add to your `SELECT` queries, but they are beyond the scope of this chapter. You can learn more about them in the SQLite documentation.

Creating Indexes for Faster Data Reading

In a previous section, we ran a `SELECT` query to find records based on matching names. You could speed up this search by creating an index on the `name` column. A *SQL index* is a data structure that organizes a column's data. As a result, queries with `WHERE` clauses that use these columns will perform better. The downside is that the index takes up a little bit more storage, so queries that insert or update data will be slightly slower, because SQLite must also update the data's index. If your database is large, and you read data from it more often than you insert or update its data, creating an index may be worthwhile. However, you should conduct testing to verify that the index actually improves performance.

To create indexes on, say, the `names` and `birthdate` columns in the `cats` table, run the following `CREATE INDEX` queries:

```
>>> import sqlite3
>>>
conn = sqlite3.connect('sweigartcats.db',
isolation_level=None)
>>> conn.execute('CREATE INDEX idx_name ON
cats (name)')
<sqlite3.Cursor object at 0x0000013EC121A040>
>>> conn.execute('CREATE INDEX idx_birthdate
ON cats (birthdate)')
<sqlite3.Cursor object at 0x0000013EC121A040>
```

Indexes require names, and by convention, we name them after the column to which they apply, along with the `idx_` prefix. Index names are global across the entire database, so if the database contains multiple tables with columns named `birthdate`, you may also want to include the table in the index name, like `idx_cats_birthdate`. To see all the indexes that exist for a table, check the built-in `sqlite_schema` table with a `SELECT` query:

```
>>> conn.execute('SELECT name FROM
sqlite_schema WHERE type = "index" AND
tbl_name = "cats"').fetchall()
[('idx_name',), ('idx_birthdate',)]
```

If you change your mind or find that the indexes aren't improving performance, you can delete them with a `DROP INDEX` query:

```
>>> conn.execute('SELECT name FROM
sqlite_schema WHERE type = "index" AND
tbl_name = "cats"').fetchall()
[('idx_birthdate',) ('idx_name',)]
>>> conn.execute('DROP INDEX idx_name')
<sqlite3.Cursor object at 0x0000013EC121A040>
>>> conn.execute('SELECT name FROM
sqlite_schema WHERE type = "index" AND
tbl_name = "cats"').fetchall()
[('idx_birthdate',)]
```

For small databases with only a few thousand records, you can safely ignore indexes, as the benefits they provide are negligible. However, if you find that your database queries are taking a noticeable amount of time, creating indexes could improve their performance.

Updating Data in the Database

Once you've inserted rows into a table, you can change a row with an `UPDATE` statement. For example, let's update the record `(1, 'Zophie', '2021-01-24', 'black', 5.6)` to change the fur color from `'black'` to `'gray tabby'` in the *sweigartcats.db* file:

```
>>> import sqlite3
>>>
conn = sqlite3.connect('sweigartcats.db',
isolation_level=None)
>>> conn.execute('SELECT * FROM cats WHERE
rowid = 1').fetchall()
[('Zophie', '2021-01-24', 'black', 5.6)]
>>>
```

```
conn.execute('UPDATE cats SET fur = "gray
tabby" WHERE rowid = 1')
<sqlite3.Cursor object at 0x0000013EC121A040>
>>> conn.execute('SELECT * FROM cats WHERE
rowid = 1').fetchall()
[('Zophie', '2021-01-24', 'gray tabby', 5.6)]
```

The `UPDATE` statement has the following parts:

- The `UPDATE` keyword
- The name of the table containing the rows to update
- The `SET` clause, which specifies the column to update, as well as the value to update it to
- The `WHERE` clause, which specifies which rows to update

You can update multiple columns at a time by separating them with commas. For example, the query `'UPDATE cats SET fur = "black", weight_kg = 6 WHERE rowid = 1'` updates the value in the `fur` and `weight` columns to `"black"` and `6`, respectively.

The `UPDATE` query updates every row in which the `WHERE` clause is true. If you ran the query `'UPDATE cats SET fur = "gray tabby" WHERE name = "Zophie"'`, the updates would apply for every cat named Zophie. That might be more cats than you intended! This is why, in most update queries, the `WHERE` clause uses the primary key from the `rowid` column to specify an individual record to update. The primary key uniquely identifies a row, so using it in the `WHERE` clause ensures that you update only the one row you intended.

It's a common bug to forget the `WHERE` clause when updating data. For example, if you wanted to do a find-and-replace to change every cat with `'white and orange'` fur to `'orange and white'` fur, you would run the following:

```
>>> conn.execute('UPDATE cats SET fur =
"orange and white" WHERE fur = "white and
orange"')
```

If you forgot to include the `WHERE` clause, the updates would apply to every row in the table. and suddenly all of your cats would have orange and white fur!

To avoid this bug, always include a `WHERE` clause in your `UPDATE` queries, even if you intend to apply a change to every row. In that case, you can use `WHERE 1`. Since `1` is the value that SQLite uses for a Boolean `True`, this tells SQLite to apply the change to every row. It may seem silly to have a superfluous `WHERE 1` at the end of your query, but it lets you avoid dangerous bugs that could easily wipe out real data.

Deleting Data from the Database

You can delete rows from a table with a `DELETE` query. For example, to remove Zophie from the `cats` table, run the following on the *sweigartcats.db* file:

```
>>> import sqlite3
>>>
conn = sqlite3.connect('sweigartcats.db',
isolation_level=None)
>>> conn.execute('SELECT rowid, * FROM cats
WHERE rowid = 1').fetchall()
[(1, 'Zophie', '2021-01-24', 'gray tabby',
5.6)]
>>> conn.execute('DELETE FROM cats WHERE
rowid = 1')
<sqlite3.Cursor object at 0x0000020322D183C0>
>>> conn.execute('SELECT * FROM cats WHERE
rowid = 1').fetchall()
[]
```

The `DELETE` statement has the following parts:

- The `DELETE FROM` keywords
- The name of the table containing the rows to delete
- The `WHERE` clause, which specifies which rows to delete

As with the `INSERT` statement, it’s vital to always include a `WHERE` clause in your `DELETE` statements; otherwise, you’ll delete every row from the table. If you intend to delete every row, use `WHERE 1` so that you can identify any `DELETE` statement without a `WHERE` clause as a bug.

Rolling Back Transactions

You may sometimes want to run several queries all together, or else not run those queries at all, but you won't know which you want to do until you've run at least some of the queries. One way to handle this situation is to begin a new transaction, execute the queries, and then either *commit* all of the queries to the database to complete the transaction or *roll them back* so that the database looks as if none of them were made.

Normally, a new transaction starts and finishes every time you call `conn.execute()` when connected to the SQLite database in autocommit mode. However, you can also run a `BEGIN` query to start a new transaction; then, you can either complete the transaction by calling `conn.commit()` or undo all the queries by calling `conn.rollback()`.

For example, let's add two new cats to the `cats` table, then roll back the transaction so that the table remains unchanged:

```
>>> import sqlite3
>>>
conn = sqlite3.connect('sweigartcats.db',
isolation_level=None)
>>> conn.execute('BEGIN')
<sqlite3.Cursor object at 0x00000219C8BF7C40>
>>> conn.execute('INSERT INTO cats VALUES
("Socks", "2022-04-04", "white", 4.2)')
<sqlite3.Cursor object at 0x00000219C8BF7C40>
>>> conn.execute('INSERT INTO cats VALUES
("Fluffy", "2022-10-30", "gray", 4.5)')
<sqlite3.Cursor object at 0x00000219C8BF7C40>
>>>
conn.rollback()  # This undoes the INSERT
statements.
>>> conn.execute('SELECT * FROM cats WHERE
name = "Socks"').fetchall()
[]
>>> conn.execute('SELECT * FROM cats WHERE
name = "Fluffy"').fetchall()
[]
```

The new cats, Socks and Fluffy, were not inserted into the database.

On the other hand, if you want to apply all of the queries you've run, call `conn.commit()` to commit the changes to the database:

```
>>> conn.execute('BEGIN')
<sqlite3.Cursor object at 0x00000219C8BF7C40>
>>> conn.execute('INSERT INTO cats VALUES
("Socks", "2022-04-04", "white", 4.2)')
<sqlite3.Cursor object at 0x00000219C8BF7C40>
>>> conn.execute('INSERT INTO cats VALUES
("Fluffy", "2022-10-30", "gray", 4.5)')
<sqlite3.Cursor object at 0x00000219C8BF7C40>
>>> conn.commit()
>>> conn.execute('SELECT * FROM cats WHERE
name = "Socks"').fetchall()
[('Socks', '2022-04-04', 'white', 4.2)]
>>> conn.execute('SELECT * FROM cats WHERE
name = "Fluffy"').fetchall()
[('Fluffy', '2022-10-30', 'gray', 4.5)]
```

Now the cats Socks and Fluffy have records in the database.

Backing Up Databases

A friend of mine was once making changes to a database used by an e-commerce site specializing in collectible sports cards. She had to correct some naming mistakes in a few cards, and had just typed `UPDATE cards SET name = 'Chris Clemons'` when her cat walked on her keyboard, pressing ENTER. Without a `WHERE` clause, the query updated every one of the thousands of cards for sale on the website.

Fortunately, she had backups of the database, so she could restore it to its previous state. (This was especially useful because the same thing happened again in the exact same way, making her suspect the cat was doing it on purpose.)

If a program isn't currently accessing the SQLite database, you can back it up by simply copying the database file. A Python program might do this by calling `shutil.copy('sweigartcats.db', 'backup.db')`, as described in Chapter 11. If your software is constantly reading or updating the database's contents, however, you'll need to use the `Connection` object's `backup()` method instead. For example, enter the following into the interactive shell:

```
>>> import sqlite3
>>>
conn = sqlite3.connect('sweigartcats.db',
isolation_level=None)
>>> backup_conn =
sqlite3.connect('backup.db',
isolation_level=None)
>>> conn.backup(backup_conn)
```

The `backup()` method safely backs up the contents of the *sweigartcats.db* database to the *backup.db* file in between the other queries being run on it. Now that your data is safely backed up, your cat is free to step on your keyboard as much as it wants.

Altering and Dropping Tables

After creating a table in a database and inserting rows into it, you may want to rename the table or its columns. You may also wish to add or delete columns in the table, or even delete the entire table itself. You can use an `ALTER TABLE` query to perform these actions.

The following interactive shell examples start with a fresh copy of the *sweigartcats.db* database file. Run an `ALTER TABLE RENAME` query to rename the `cats` table to `felines`:

```
>>> import sqlite3
>>>
conn = sqlite3.connect('sweigartcats.db',
isolation_level=None)
>>> conn.execute('SELECT name FROM
sqlite_schema WHERE type="table"').fetchall()
```

```
[('cats',)]
>>> conn.execute('ALTER TABLE cats RENAME TO
felines')  # Rename the table.
<sqlite3.Cursor object at 0x000001EDDB477C40>
>>> conn.execute('SELECT name FROM
sqlite_schema WHERE type="table"').fetchall()
[('felines',)]
```

To rename a column in a table, run an `ALTER TABLE RENAME COLUMN` query. For example, let's rename the `fur` column to `description`:

```
>>> conn.execute('PRAGMA
TABLE_INFO(felines)').fetchall()[2]  # List
the third column.
(2, 'fur', 'TEXT', 0, None, 0)
>>> conn.execute('ALTER TABLE felines RENAME
COLUMN fur TO description')
<sqlite3.Cursor object at 0x000001EDDB477C40>
>>> conn.execute('PRAGMA
TABLE_INFO(felines)').fetchall()[2]  # List
the third column.
(2, 'description', 'TEXT', 0, None, 0)
```

To add a new column to the table, run an `ALTER TABLE ADD COLUMN` query. For example, let's add a new `is_loved` column to the `felines` table containing a Boolean value. SQLite uses `0` for false values and `1` for true values; we'll set the default value for `is_loved` to `1`:

```
>>> conn.execute('ALTER TABLE felines ADD
COLUMN is_loved INTEGER DEFAULT 1')
<sqlite3.Cursor object at 0x000001EDDB477C40>
>>> import pprint
>>>
```

```
pprint.pprint(conn.execute('SELECT * FROM
felines LIMIT 3').fetchall())
[('Zophie', '2021-01-24', 'gray tabby', 5.6,
1),
 ('Miguel', '2016-12-24', 'siamese', 6.2, 1),
 ('Jacob', '2022-02-20', 'orange and white',
5.5, 1)]
```

It turns out the `is_loved` column isn't needed, since I store a `1` in it for all my cats, so I can remove the column with an `ALTER TABLE DROP COLUMN` query:

```
>>> conn.execute('PRAGMA
TABLE_INFO(felines)').fetchall()  # List all
columns.
[(0, 'name', 'TEXT', 1, None, 0), (1,
'birthdate', 'TEXT', 0, None, 0), (2,
'description', 'TEXT',
0, None, 0), (3, 'weight_kg', 'REAL', 0,
None, 0), (4, 'is_loved', 'INTEGER', 0, '1',
0)]
>>> conn.execute('ALTER TABLE felines DROP
COLUMN is_loved')  # Delete the column.
<sqlite3.Cursor object at 0x000001EDDB477C40>
>>> conn.execute('PRAGMA
TABLE_INFO(felines)').fetchall()  # List all
columns.
[(0, 'name', 'TEXT', 1, None, 0), (1,
'birthdate', 'TEXT', 0, None, 0), (2,
'description', 'TEXT',
0, None, 0), (3, 'weight_kg', 'REAL', 0,
None, 0)]
```

Any data stored in the deleted column will also be deleted.

If you want to delete the entire table, run a DROP TABLE query:

```
>>> conn.execute('SELECT name FROM
sqlite_schema WHERE type="table"').fetchall()
[('felines',)]
>>> conn.execute('DROP TABLE felines')  #
Delete the entire table.
<sqlite3.Cursor object at 0x000001EDDB477C40>
>>> conn.execute('SELECT name FROM
sqlite_schema WHERE type="table"').fetchall()
[]
```

Try to limit how often you change your tables and columns, as you'll also have to update the queries in your program to match.

Joining Multiple Tables with Foreign Keys

The structure of SQLite tables is rather strict; for example, each row has a set number of columns. But real-world data is often more complicated than a single table can capture. In relational databases, we can store complex data across multiple tables, and we can create links between them called *foreign keys*.

Say we want to store information about the vaccinations our cats receive. We can't just add columns to our `cats` table, as each cat could have one vaccination or many. Also, for each vaccination, we'd also want to list the vaccination date and the name of the doctor who administered it. SQL tables are not good at storing a list of columns. You would not want to have columns named `vaccination1`, `vaccination2`, `vaccination3`, and so on, for the same reason you wouldn't want variables named `vaccination1` and `vaccination2`. If you create too many columns or variables, your code becomes a verbose, unreadable mess. If you create too few, you will have to constantly update your program to add more as needed.

Whenever you have a varying amount of data to add to a row, it makes more sense to list the added data as rows in a separate table, then have those rows refer back to the rows in the main table. In our *sweigartcats.db* database, add a second `vaccinations` table by entering the following into the interactive shell:

```
>>> import sqlite3
>>>
conn = sqlite3.connect('sweigartcats.db',
isolation_level=None)
>>> conn.execute('PRAGMA foreign_keys = ON')
<sqlite3.Cursor object at 0x000001E730AD03C0>
>>> conn.execute('CREATE TABLE IF NOT EXISTS
vaccinations (vaccine TEXT,
date_administered TEXT, administered_by TEXT,
cat_id INTEGER,
FOREIGN KEY(cat_id) REFERENCES cats(rowid))
STRICT')
<sqlite3.Cursor object at 0x000001CA42767D40>
```

The new `vaccinations` table has a column named `cat_id` with an `INTEGER` type. The integer values in this column matches the `rowid` values of the rows in the `cats` table. We call the `cat_id` column a *foreign key* because it refers to the primary key column of another table.

In the `cats` table, the cat Zophie has a `rowid` of `1`. To record her vaccinations, we insert new rows into the `vaccinations` table with a `cat_id` value of `1`:

```
>>> conn.execute('INSERT INTO vaccinations
VALUES ("rabies", "2023-06-06", "Dr. Echo",
1)')
<sqlite3.Cursor object at 0x000001CA42767D40>
>>> conn.execute('INSERT INTO vaccinations
VALUES ("FeLV", "2023-06-06", "Dr. Echo",
1)')
<sqlite3.Cursor object at 0x000001CA42767D40>
>>> conn.execute('SELECT * FROM
vaccinations').fetchall()
[('rabies', '2023-06-06', 'Dr. Echo', 1),
('FeLV', '2023-06-06', 'Dr. Echo', 1)]
```

We could record vaccinations for other cats by using their `rowid`. If we wanted to add vaccination records for Mango, we could find Mango's `rowid` in the `cats` table and then add records to the `vaccinations` table using that value for the `cat_id` column:

```
>>> conn.execute('SELECT rowid, * FROM cats
WHERE name = "Mango"').fetchall()
[(23, 'Mango', '2017-02-12', 'tuxedo', 6.8)]
>>> conn.execute('INSERT INTO vaccinations
VALUES ("rabies", "2023-07-11", "Dr. Echo",
23)')
<sqlite3.Cursor object at 0x000001CA42767D40>
```

We can also perform a type of `SELECT` query called an *inner join*, which returns the linked rows from both tables. For example, enter the following into the interactive shell to retrieve the `vaccinations` rows joined with the data from the `cats` table:

```
>>> conn.execute('SELECT * FROM cats INNER
JOIN vaccinations ON cats.rowid =
vaccinations.cat_id').fetchall()
[('Zophie', '2021-01-24', 'gray tabby', 5.6,
'rabies', '2023-06-06', 'Dr. Echo', 1),
 ('Zophie', '2021-01-24', 'gray tabby', 5.6,
'FeLV', '2023-06-06', 'Dr. Echo', 1),
 ('Mango', '2017-02-12', 'tuxedo', 6.8,
'rabies', '2023-07-11', 'Dr. Echo', 23)]
```

Note that while you could make `cat_id` an `INTEGER` column and use it as a foreign key without actually setting up the `FOREIGN KEY(cat_id) REFERENCES cats(rowid)` syntax, foreign keys have several safety features to ensure that your data remains consistent. For example, you can't insert or update a vaccination record using a `cat_id` for a nonexistent cat. SQLite also forces you to delete all vaccination records for a cat before deleting the cat so as to not leave behind "orphaned" vaccination records.

These safety features are disabled by default. You can enable them by running the `PRAGMA` query after calling `sqlite3.connect()`:

```
>>> conn.execute('PRAGMA foreign_keys = ON')
```

Foreign keys and joins have additional features, but they are outside the scope of this book.

In-Memory Databases and Backups

If your program is making a large number of queries, you can significantly improve the speed of your database by using an *in-memory database*. These databases are stored entirely in the computer's memory rather than in a file on the computer's hard drive. This makes changes incredibly fast. However, you'll need to remember to save the in-memory database to a file using the `backup()` method. If your program crashes in the middle of running, you'll lose the entire in-memory database, just as you would the values in the program's variables.

The following example creates an in-memory database and then saves it to a database in the file *test.db*:

```
>>> import sqlite3
>>> memory_db_conn =
sqlite3.connect(':memory:',
isolation_level=None)  # Create an in-memory
database.
>>>
memory_db_conn.execute('CREATE TABLE test
(name TEXT, number REAL)')
<sqlite3.Cursor object at 0x000001E730AD0340>
>>> memory_db_conn.execute('INSERT INTO test
VALUES ("foo", 3.14)')
<sqlite3.Cursor object at 0x000001D9B0A07EC0>
>>>
file_db_conn = sqlite3.connect('test.db',
isolation_level=None)
```

```
>>> memory_db_conn.backup(file_db_conn)  #
Save the database to test.db.
```

You can load a SQLite database file into memory with the `backup()` method as well:

```
>>> import sqlite3
>>> file_db_conn =
sqlite3.connect('sweigartcats.db',
isolation_level=None)
>>> memory_db_conn =
sqlite3.connect(':memory:',
isolation_level=None)
>>> file_db_conn.backup(memory_db_conn)
>>> memory_db_conn.execute('SELECT * FROM
cats LIMIT 3').fetchall()
[('Zophie', '2021-01-24', 'gray tabby', 5.6),
('Miguel', '2016-12-24',
'siamese', 6.2), ('Jacob', '2022-02-20',
'orange and white', 5.5)]
```

There are some downsides to using in-memory databases. If your program crashes from an unhandled exception, you'll lose the database. You can mitigate this risk by wrapping your code in a `try` statement that catches any unhandled exceptions and then uses an `except` statement to save the file to a database. Chapter 4 covers exception handling with the `try` and `except` statements.

Copying Databases

You can obtain a copy of a database by calling the `iterdump()` method on the `Connection` object. This method returns an iterator that generates the text of the SQLite queries needed to re-create the database. You can use iterators in `for` loops or pass them to the `list()` function to convert them to a list of strings. For example, to get the SQLite queries needed to re-create the *sweigartcats.db* database, enter the following into the interactive shell:

```
>>> import sqlite3
>>>
conn = sqlite3.connect('sweigartcats.db',
isolation_level=None)
>>> with open('sweigartcats-queries.txt',
'w', encoding='utf-8') as fileObj:
...     for line in conn.iterdump():
...         fileObj.write(line + '\n')
```

This code creates a *sweigartcats-queries.txt* file with the following SQLite queries, which can re-create the database:

```
BEGIN TRANSACTION;
CREATE TABLE "cats" (name TEXT NOT NULL,
birthdate TEXT, fur TEXT, weight_kg REAL)
STRICT;
INSERT INTO "cats"
VALUES('Zophie','2021-01-24','gray
tabby',5.6);
INSERT INTO "cats"
VALUES('Miguel','2016-12-24','siamese',6.2);
INSERT INTO "cats"
VALUES('Jacob','2022-02-20','orange and
white',5.5);
--snip--
INSERT INTO "cats"
VALUES('Spunky','2015-09-04','gray',5.9);
INSERT INTO "cats"
VALUES('Shadow','2021-01-18','calico',6.0);
COMMIT;
```

The text of these queries will almost certainly be larger than the original database, but the queries have the advantage of being human readable and easy to edit before copying and pasting into your Python code or into a SQLite app, as we’ll cover next.

SQLite Apps

At times, you may want to investigate a SQLite database directly without having to write all of this extraneous Python code. You can do so by installing the `sqlite3` command, which runs from a terminal command line window and is documented at *https://sqlite.org/cli.html*.

On Windows, download the files labeled “A bundle of command line tools for managing SQLite database files” from *https://sqlite.org/download.html* and place the *sqlite3.exe* program in a folder on the system `PATH`. (See Chapter 12 for information about the `PATH` environment variable and terminal windows.) The `sqlite3` command is preinstalled on macOS. For UbuntuLinux, run `sudo apt install sqlite3` to install it.

Next, in a terminal window, run **sqlite3 example.db** to connect to the database in *example.db*. If this file doesn’t exist, `sqlite3` creates this file with an empty database. You can enter SQL queries into this tool, though unlike the queries passed to `conn.execute()` in Python, they must end with a semicolon.

For example, enter the following into the terminal window:

```
C:\Users\Al>sqlite3 example.db
SQLite version 3.xx.xx
Enter ".help" for usage hints.
sqlite>
CREATE TABLE IF NOT EXISTS cats (name TEXT
NOT NULL,
birthdate TEXT, fur TEXT, weight_kg REAL)
STRICT;
sqlite> INSERT INTO cats VALUES ('Zophie',
'2021-01-24', 'gray tabby', 4.7);
sqlite> SELECT * from cats;
Zophie|2021-01-24|gray tabby|4.7
```

As you can see in this example, the `sqlite3` command line tool provides a sort of SQLite interactive shell for you to enter queries at its `sqlite>` prompt. The `.help` command displays additional commands, such as `.tables` (which shows the tables in the database) and `.headers` (which lets you turn column headers on or off):

```
sqlite> .tables
cats
sqlite> .headers on
sqlite> SELECT * from cats;
name|birthdate|fur|weight_kg
Zophie|2021-01-24|gray tabby|4.7
```

If the command line tool is too sparse for you, there are also free, open source apps for displaying SQLite databases in a graphical user interface (GUI) on Windows, macOS, and Linux:

- DB Browser for SQLite (*https://sqlitebrowser.org*)
- SQLite Studio (*https://sqlitestudio.pl*)
- DBeaver Community (*https://dbeaver.io*)

While these GUI apps make it easier to work with SQLite databases, it's still worth learning the text-based syntax of SQLite queries.

Summary

Computers make it possible to deal with large amounts of data, but simply putting data into a text file, or even a spreadsheet, might not organize it well enough for you to make efficient use of it. SQL databases such as SQLite offer an advanced way to not only store large amounts of information but also retrieve the precise data you want through the SQL language.

SQLite is an impressive database, and Python comes with the `sqlite3` module in its standard library. SQLite's version of SQL is different from that used in other relational databases, but it's similar enough that learning SQLite provides a good introduction to databases in general.

SQLite databases live in a single file without a dedicated server. They can contain multiple tables (which you can think of as analogous to spreadsheets), and each table can contain multiple columns. To edit a table's values, you can perform the CRUD operations (for create, read, update, and delete) with the `INSERT`, `SELECT`, `UPDATE`, and `DELETE` queries. To change tables and columns themselves, you can use the `ALTER TABLE` and `DROP TABLE` queries. Lastly, foreign keys allow you to link records in multiple tables together using a technique called joins.

There's a lot more to SQLite and databases than can be covered in one chapter. If you'd like to learn more about SQL databases in general, I recommend *Practical SQL,* 2nd edition (No Starch Press, 2022) by Anthony DeBarros.

Practice Questions

1. What Python instructions will obtain a `Connection` object for a SQLite database in a file named *example.db*?
2. What Python instruction will create a new table named `students` with `TEXT` columns named `first_name`, `last_name`, and `favorite_color`?
3. How do you connect to a SQLite database in autocommit mode?
4. What's the difference between the `INTEGER` and `REAL` data types in SQLite?
5. What does strict mode add to a table?
6. What does the `*` in the query `'SELECT * FROM cats'` mean?
7. What does CRUD stand for?
8. What does ACID stand for?
9. What query adds new records to a table?
10. What query deletes records from a table?
11. What happens if you don't specify the `WHERE` clause in an `UPDATE` query?
12. What is an index? What code would create an index for a column named `birthdate` in a table named `cats`?
13. What is a foreign key?
14. How can you delete a table named `cats`?
15. What "filename" do you specify to create an in-memory database?
16. How can you copy a database to another database?

Practice Programs

For practice, write programs to do the following tasks.

Cat Vaccination Checker

Download the *sweigartcats.db* database of my cats from the book's resources at *https://nostarch.com/automate-boring-stuff-python-3rd-edition*. Write a program that opens this database and lists all cats that don't have vaccines named `'rabies'`, `'FeLV'`, and `'FVRCP'`. Also, check the database for errors by finding all vaccines that were administered on a date before the cat's birthday.

Meal Ingredients Database

Write a program that creates two tables, one for meals and one for ingredients, using these SQL queries:

```
CREATE TABLE IF NOT EXISTS meals (name TEXT)
STRICT
CREATE TABLE IF NOT EXISTS ingredients (name
TEXT,
meal_id INTEGER, FOREIGN KEY(meal_id)
REFERENCES meals
(rowid)) STRICT
```

Then, write a program that prompts the user for input. If the user enters `'quit'`, the program should exit. The user can also enter a new meal name, followed by a colon and a comma-delimited list of ingredients: `'meal:ingredient1,ingredient2'`. Save the meal and its ingredients in the `meals` and `ingredients` tables.

Finally, the user can enter the name of a meal or ingredient. If the name appears in the `meals` table, the program should list the meal’s ingredients. If the name appears in the `ingredients` table, the program should list every meal that uses this ingredient. For example, the output of the program could look like this:

```
> onigiri:rice,nori,salt,sesame seeds
Meal added: onigiri
> chicken and rice:chicken,rice,cream of
chicken soup
Meal added: chicken and rice
> onigiri
Ingredients of onigiri:
  rice
  nori
  salt
  sesame seeds
> chicken
Meals that use chicken:
  chicken and rice
> rice
Meals that use rice:
  onigiri
chicken and rice
> quit
```