Chapter 7. Using a Database: Putting Python's DB-API to Use

Storing data in a relational database system is handy.

In this chapter, you'll learn how to write code that interacts with the popular **MySQL** database technology, using a generic database API called **DB-API**. The DB-API (which comes standard with every Python install) allows you to write code that is easily transferred from one database product to the next...assuming your database talks SQL. Although we'll be using MySQL, there's nothing stopping you from using your DB-API code with your favorite relational database, whatever it may be. Let's see what's involved in using a relational database with Python. There's not a lot of new Python in this chapter, but using Python to talk to databases is a **big deal**, so it's well worth learning.

Database-Enabling Your Webapp

The plan for this chapter is to get to the point where you can amend your webapp to store its log data in a database, as opposed to a text file, as was the case in the last chapter. The hope is that in doing so, you can then provide answers to the questions posed in the last chapter: *How many requests have been responded to? What's the most common list of letters? Which IP addresses are the requests coming from? Which browser is being used the most?*

To get there, however, we need to decide on a database system to use. There are lots of choices here, and it would be easy to take a dozen pages or so to present a bunch of alternative database technologies while exploring the pluses and minuses of each. But we're not going to do that. Instead, we're going to stick with a popular choice and use *MySQL* as our database technology.

Having selected MySQL, here are the four tasks we'll work through over the next dozen pages:

1. **Install the MySQL server**
2. **Install a MySQL database driver for Python**
3. **Create our webapp's database and tables**
4. **Create code to work with our webapp's database and tables**

With these four tasks complete, we'll be in a position to amend the

THERE ARE NO DUMB QUESTIONS

Q:	**Q: Do we have to use MySQL here?**
A:	**A:** *If you want to follow along with the examples in the remainder of this chapter, the answer is yes.*
Q:	**Q: Can I use MariaDB instead of MySQL?**
A:	**A:** *Yes. As MariaDB is a clone of MySQL, we have no issue with you using MariaDB as your database system instead of the "official" MySQL. (In fact, over at Head First Labs, MariaDB is a favorite among the DevOps team.)*
Q:	**Q: What about PostgreSQL? Can I use that?**
A:	**A:** *Emm, eh...yes, subject to the following caveat: if you are already using PostgreSQL (or any other SQL-based database management system), you can try using it in place of MySQL. However, note that this chapter doesn't provide any specific instructions related to PostgreSQL (or anything else), so you may have to experiment on your own when something we show you working with MySQL doesn't work in quite the same way with your chosen database. There's also the standalone, single-user* **SQLite,** *which comes with Python and lets you work with SQL without the need for a separate server. That said, which database technology you use very much depends on what you're trying to do.*

Task 1: Install the MySQL Server

If you already have MySQL installed on your computer, feel free to move on to Task 2.

	Install MySQL on your computer.
	Install a MySQL Python driver.
	Create the database and tables.
	Create code to read/write data.

NOTE

We'll check off each completed task as we work through them.

How you go about installing MySQL depends on the operating system you’re using. Thankfully, the folks behind MySQL (and its close cousin, MariaDB) do a great job of making the installation process straightforward.

If you’re running *Linux*, you should have no trouble finding `mysql-server` (or `mariadb-server`) in your software repositories. Use your software installation utility (`apt`, `aptitude`, `rpm`, `yum`, or whatever) to install MySQL as you would any other package.

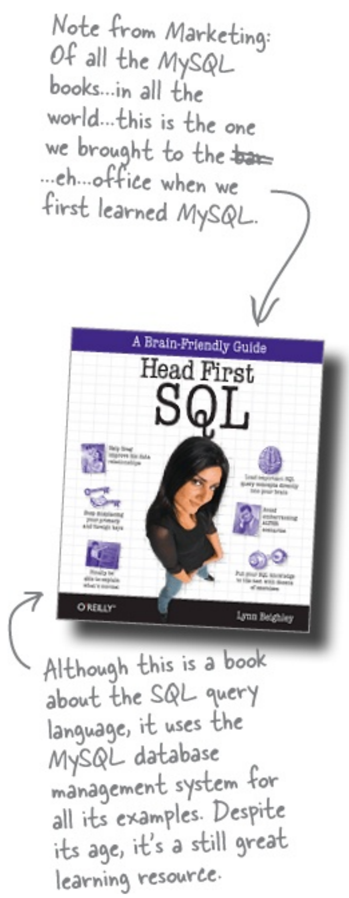
If you’re running *Mac OS X*, we recommend installing *Homebrew* (find out about Homebrew here: *http://brew.sh* (http://brew.sh)), then using it to install MariaDB, as in our experience this combination works well.

For all other systems (including all the various *Windows* versions), we recommend you install the **Community Edition** of the MySQL server, available from:

http://dev.mysql.com/downloads/mysql/ (http://dev.mysql.com/downloads/mysql/)

Or, if you want to go with MariaDB, check out:

https://mariadb.org/download/



Be sure to read the installation documentation associated with whichever version of the server your download and install.

Don't worry if this is new to you.

We don't expect you to be a MySQL whiz-kid while working through this material. We'll provide you with everything you need in order to get each of our examples to work (even if you've never used MySQL before).

If you want to take some time to learn more, we recommended *Lynn Beighley's* excellent *Head First SQL* as a wonderful primer.

Introducing Python's DB-API

With the database server installed, let's park it for a bit, while we add support for working with MySQL into Python.

	Install MySQL on your computer.
	Install a MySQL Python driver.
	Create the database and tables.
	Create code to read/write data.

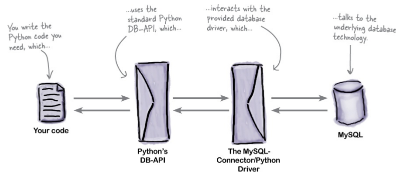
Out of the box, the Python interpreter comes with some support for working with databases, but nothing specific to MySQL. What's provided is a standard database API (application programmer interface) for working with SQL-based databases, known as *DB-API*. What's missing is the **driver** to connect the DB-API up to the actual database technology you're using.

The convention is that programmers use the DB-API when interacting with any underlying database using Python, no matter what that database technology happens to be. They do that because the driver shields programmers from having to understand the nitty-gritty details of interacting with the database's actual API, as the DB-API provides an abstract layer between the two. The idea is that, by programming to the DB-API, you can replace the underlying database technology as needed without having to throw away any existing code.

GEEK BITS

Python's DB-API is defined in PEP 0247. That said, don't feel the need to run off and read this PEP, as it's primarily designed to be used as a specification by database driver implementers (as opposed to being a how-to tutorial).

We’ll have more to say about the DB-API later in this chapter. Here’s a visualization of what happens when you use Python’s DB-API:



Some programmers look at this diagram and conclude that using Python’s DB-API must be hugely inefficient. After all, there are *two* layers of technology between your code and the underlying database system. However, using the DB-API allows you to swap out the underlying database as needed, avoiding any database “lock-in,” which occurs when you code *directly* to a database. When you also consider that no two SQL dialects are the same, using DB-API helps by providing a higher level of abstraction.

Task 2: Install a MySQL Database Driver for Python

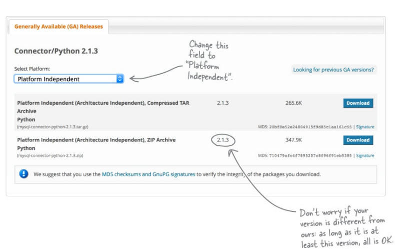
	Install MySQL on your computer.
	Install a MySQL Python driver.
	Create the database and tables.
	Create code to read/write data.

Anyone is free to write a database driver (and many people do), but it is typical for each database manufacturer to provide an *official driver* for each of the programming languages they support. *Oracle*, the owner of the MySQL technologies, provides the *MySQL-Connector/Python* driver, and that’s what we propose to use in this chapter. There’s just one problem: *MySQL-Connector/Python* can’t be installed with `pip`.

Does that mean we’re out of luck when it comes to using *MySQL-Connector/Python* with Python? No, far from it. The fact that a third-party module doesn’t use the `pip` machinery is rarely a show-stopper. All we need to do is install the module “by hand”—it’s a small amount of extra work (over using `pip`), but not much.

Let’s install the *MySQL-Connector/Python* driver by hand (bearing in mind there are *other* drivers available, such as *PyMySQL*; that said, we prefer *MySQL-Connector/Python*, as it’s the officially supported driver provided by the makers of MySQL).

Begin by visiting the *MySQL-Connector/Python* download page: *https://dev.mysql.com/downloads/connector/python/*. Landing on this web page will likely preselect your operating system from the *Select Platform* drop-down menu. Ignore this, and adjust the selection drop-down to read *Platform Independent*, as shown here:



Then, go ahead and click either of the *Download* buttons (typically, *Windows* users should download the ZIP file, whereas *Linux* and *Mac OS X* users can download the GZ file). Save the downloaded file to your computer, then double-click on the file to expand it within your download location.

Install MySQL-Connector/Python

With the driver downloaded and expanded on your computer, open a terminal window in the newly created folder (if you’re on *Windows*, open the terminal window with *Run as Administrator*).

	Install MySQL on your computer.
	Install a MySQL Python driver.
	Create the database and tables.
	Create code to read/write data.

On our computer, the created folder is called `mysql-connector-python-2.1.3` and was expanded in our `Downloads` folder. To install the driver into *Windows*, issue this command from within the `mysql-connector-python-2.1.3` folder:

```
py -3 setup.py install
```

On *Linux* or *Mac OS X*, use this command instead:

```
sudo -H python3 setup.py install
```

No matter which operating system you’re using, issuing either of the above commands results in a collection of messages appearing on screen, which should look similar to these:

These paths may be different on your computer. Don’t worry about it if they are.

```
running install
Not Installing C Extension
running build
running build_py
running install_lib
running install_egg_info
Removing /Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/
mysql_connector_python-2.1.3-py3.5.egg-info
Writing /Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/
mysql_connector_python-2.1.3-py3.5.egg-info
```

When you install a module with `pip`, it runs though this same process, but hides these messages from you. What you’re seeing here is the status messages that indicate that the installation is proceeding smoothly. If something goes wrong, the resulting error message should provide enough information to resolve the problem. If all goes well with the installation, the appearance of these messages is confirmation that *MySQL-Connector/Python* is ready to be used.

THERE ARE NO DUMB QUESTIONS

Q:	**Q: Should I worry about that "Not Installing C Extension" message?**
A:	**A:** *No. Third-party modules sometimes include embedded C code, which can help improve computationally intensive processing. However, not all operating systems come with a preinstalled C compiler, so you have to specifically ask for the C extension support to be enabled when installing a module (should you decide you need it). When you don't ask, the third-party module installation machinery uses (potentially slower) Python code in place of the C code. This allows the module to work on any platform, regardless of the existence of a C compiler. When a third-party module uses Python code exclusively, it is referred to as being written in "pure Python." In the example above, we've installed the pure Python version of the MySQL-Connector/Python driver.*

Task 3: Create Our Webapp's Database and Tables

You now have the MySQL database server and the *MySQL-Connector/Python* driver installed on your computer. It's time for Task 3, which involves creating the database and the tables required by our webapp.

	Install MySQL on your computer.
	Install a MySQL Python driver.
	Create the database and tables.
	Create code to read/write data.

To do this, you're going to interact with the MySQL server using its command-line tool, which is a small utility that you start from your terminal window. This tool is known as the MySQL *console*. Here's the command to start the console, logging in as the MySQL database administrator (which uses the `root` user ID):

```
mysql -u root -p
```

If you set an administrator password when you installed the MySQL server, type in that password after pressing the *Enter* key. Alternatively, if you have no password, just press the *Enter* key twice. Either way, you'll be taken to the **console prompt**, which looks like this (on the left) when using MySQL, or like this (on the right) when using MariaDB:

```
mysql>                              MariaDB [None]>
```

Any commands you type at the console prompt are delivered to the MySQL server for execution. Let's start by creating a database for our webapp. Remember: we want to use the database to store logging data, so the database's name should reflect this purpose. Let's call our database `vsearchlogDB`. Here's the console command that creates our database:

```
mysql> create database vsearchlogDB;
```

Be sure to terminate each command you enter into the MySQL console with a semicolon.

The console responds with a (rather cryptic) status message: `Query OK, 1 row affected (0.00 sec)`. This is the console’s way of letting you know that everything is golden.

Let’s create a database user ID and password specifically for our webapp to use when interacting with MySQL as opposed to using the `root` user ID all the time (which is regarded as bad practice). This next command creates a new MySQL user called `vsearch`, uses “vsearchpasswd” as the new user’s password, and gives the `vsearch` user full rights to the `vsearchlogDB` database:

```
mysql> grant all on vsearchlogDB.* to 'vsearch' identified by 'vsearchpasswd';
```

You can use a different password if you like. Just remember to use yours as opposed to ours in the examples that follow.

A similar `Query OK` status message should appear, which confirms the creation of this user. Let’s now log out of the console using this command:

```
mysql> quit
```

You’ll see a friendly `Bye` message from the console before being returned to your operating system.

Decide on a Structure for Your Log Data

Now that you’ve created a database to use with your webapp, you can create any number of tables within that database (as required by your application). For our purposes, a single table will suffice here, as all we need to store is the data relating to each logged web request.

	Install MySQL on your computer.
	Install a MySQL Python driver.
	Create the database and tables.
	Create code to read/write data.

Recall how we stored this data in a text file in the previous chapter, with each line in the `vsearch.log` file conforming to a specific format:

We log the value of the “phrase”...

...as well as the value of “letters”.

The IP address of the computer that submitted the form data is also logged.

```
ImmutableMultiDict([('phrase', 'galaxy'), ('letters', 'xyz')])|127.0.0.1|Mozilla/5.0 (Macintosh; Intel
Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106 Safari/537.36|{'x', 'y'}
```

There’s a (rather large) string that describes the web browser being used.

Last—but not least—the actual results produced by searching for “letters” in “phrase” are also logged.

At the very least, the table you create needs five fields: for the phrase, letters, IP address, browser string, and result values. But let’s also include two other fields: a unique ID for each logged request, as well as a timestamp that records when the request was logged. As these two latter fields are so common, MySQL provides an easy way to add this data to each logged request, as shown at the bottom of this page.

You can specify the structure of the table you want to create within the console. Before doing so, however, let’s log in as our newly created `vsearch` user using this command (and supplying the correct password after pressing the *Enter* key):

```
mysql -u vsearch -p vsearchlogDB
```

Remember: we set this user’s password to “vsearchpasswd”.

Here’s the SQL statement we used to create the required table (called `log`). Note that the `->` symbol is not part of the SQL statement, as it’s added automatically by the console to indicate that it expects more input from you (when your SQL runs to multiple lines). The statement ends (and executes) when you type the terminating semicolon character, and then press the *Enter* key:

MySQL will automatically provide data for these fields.

```
mysql> create table log (
    -> id int auto_increment primary key,
    -> ts timestamp default current_timestamp,
    -> phrase varchar(128) not null,
    -> letters varchar(32) not null,
    -> ip varchar(16) not null,
    -> browser_string varchar(256) not null,
    -> results varchar(64) not null );
```

This is the console's continuation symbol.

These fields will hold the data for each request (as provided in the form data).

Confirm Your Table Is Ready for Data

With the table created, we're done with Task 3.

	Install MySQL on your computer.
	Install a MySQL Python driver.
	Create the database and tables.
	Create code to read/write data.

Let's confirm at the console that the table has indeed been created with the structure we require. While still logged into the MySQL console as user `vsearch`, issue the ***`describe log`*** command at the prompt:

```
mysql> describe log;
+----------------+--------------+------+-----+-------------------+-----
| Field          | Type         | Null | Key | Default           | Extr
+----------------+--------------+------+-----+-------------------+-----
| id             | int(11)      | NO   | PRI | NULL              | auto
| ts             | timestamp    | NO   |     | CURRENT_TIMESTAMP |
| phrase         | varchar(128) | NO   |     | NULL              |
| letters        | varchar(32)  | NO   |     | NULL              |
| ip             | varchar(16)  | NO   |     | NULL              |
| browser_string | varchar(256) | NO   |     | NULL              |
| results        | varchar(64)  | NO   |     | NULL              |
+----------------+--------------+------+-----+-------------------+-----
```

And there it is: proof that the `log` table exists and has a structure that fits with our web application's logging needs. Type ***`quit`*** to exit the console (as you are done with it for now).

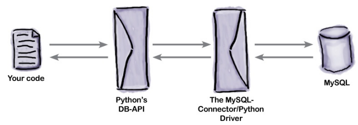
Yes, that's one possibility.

There's nothing stopping you from *manually* typing a bunch of SQL `INSERT` statements into the console to *manually* add data to your newly created table. But remember: we want our webapp to add our web request data to the `log` table **automatically**, and this applies to `INSERT` statements, too.

To do this, we need to write some Python code to interact with the `log` table. And to do *that*, we need to learn more about Python's DB-API.

DB-API UP CLOSE, 1 OF 3

Recall the diagram from earlier in this chapter that positioned Python's DB-API in relation to your code, your chosen database driver, and your underlying database system:



The promise of using DB-API is that you can replace the driver/database combination with very minor modifications to your Python code, so long as you limit yourself to only using the facilities provided by the DB-API.

Let's review what's involved in programming to this important Python standard. We are going to present six steps here.

DB-API Step 1: Define your connection characteristics

There are four pieces of information you need when connecting to MySQL: (1) the IP address/name of the computer running the MySQL server (known as the *host*), (2) the user ID to use, (3) the password associated with the user ID, and (4) the name of the database the user ID wants to interact with.

The *MySQL-Connector/Python* driver allows you to put these connection characteristics into a Python dictionary for ease of use and ease of reference. Let's do that now by typing the code in this *Up Close* into the `>>>` prompt. Be sure to follow along on your computer. Here's a dictionary (called `dbconfig`) that associates the four required "connection keys" with their corresponding values:

1. Our server is running on our local computer, so we use the localhost IP address for "host".

3. The "password" key is assigned the correct password to use with our user ID.

```
>>> dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }
```

2. The "vsearch" user ID from earlier in this chapter is assigned to the "user" key.

4. The database name—"vsearchlogDB" in this case—is assigned to the "database" key.

DB-API Step 2: Import your database driver

With the connection characteristics defined, it's time to `import` our database driver:

```
>>> import mysql.connector
```

Import the driver for the database you are using.

This import makes the MySQL-specific driver available to the DB-API.

DB-API Step 3: Establish a connection to the server

Let's establish a connection to the server by using the DB-API's `connect` function to establish our connection. Let's save a reference to the connection in a variable called `conn`. Here's the call to `connect`, which establishes the connection to the MySQL database server (and creates `conn`):

```
>>> conn = mysql.connector.connect(**dbconfig)
```

This call establishes the connection.

Pass in the dictionary of connection characteristics.

Note the strange `**` that precedes the single argument to the `connect` function. (If you're a C/C++ programmer, do **not** read `**` as "a pointer to a pointer," as Python has no notion of pointers.) The `**` notation tells the `connect` function that a dictionary of arguments is being supplied in a single variable (in this case `dbconfig`, the dictionary you just created). On seeing the `**`, the `connect` function expands the single dictionary argument into four individual arguments, which are then used within the `connect` function to establish the connection. (You'll see more of the `**` notation in a later chapter; for now, just use it as is.)

DB-API Step 4: Open a cursor

To send SQL commands to your database (via the just-opened connection) as well as receive results from your database, you need a *cursor*. Think of a cursor as the database equivalent of the *file handle* from the last chapter (which lets you communicate with a disk file once it was opened).

Creating a cursor is straightforward: you do so by calling the `cursor` method included with every connection object. As with the connection above, we save a reference to the created cursor in a variable (which, in a wild fit of imaginative creativity, we've named `cursor`):

```
>>> cursor = conn.cursor()
```
Create a cursor to send commands to the server, and to receive results.

We are now ready to send SQL commands to the server, and—hopefully—get some results back.

But, before we do that, let's take a moment to review the steps completed so far. We've defined the connection characteristics for the database, imported the driver module, created a connection object, and created a cursor. No matter which database you use, these steps are common to all interactions with MySQL (only the connection characteristics change). Keep this in mind as you interact with your data through the cursor.

DB-API UP CLOSE, 2 OF 3

With the cursor created and assigned to a variable, it’s time to interact with the data in your database using the SQL query language.

DB-API Step 5: Do the SQL thing!

The `cursor` variable lets you send SQL queries to MySQL, as well as retrieve any results produced by MySQL’s processing of the query.

As a general rule, the Python programmers over at *Head First Labs* like to code the SQL they intend to send to the database server in a triple-quoted string, then assign the string to a variable called `_SQL`. A triple-quoted string is used because SQL queries can often run to multiple lines, and using a triple-quoted string temporarily switches off the Python interpreter’s “end-of-line is the end-of-statement” rule. Using `_SQL` as the variable name is a convention among the *Head First Labs* programmers for defining constant values in Python, but you can use any variable name (and it doesn’t have to be all uppercase, nor prefixed within an underscore).

Let’s start by asking MySQL for the names of the tables in the database we’re connected to. To do this, assign the `show tables` query to the `_SQL` variable, and then call the `cursor.execute` function, passing `_SQL` as an argument:

Assign the SQL query to a variable.

```
>>> _SQL = """show tables"""
>>> cursor.execute(_SQL)
```

Send the query in the “_SQL” variable to MySQL for execution.

When you type the above `cursor.execute` command at the `>>>` prompt, the SQL query is sent to your MySQL server, which proceeds to execute the query (assuming it’s valid and correct SQL). However, any results from the query *don’t* appear immediately; you have to ask for them.

You can ask for results using one of three cursor methods:

- `cursor.fetchone` retrieves a **single** row of results.
- `cursor.fetchmany` retrieves the **number** of rows you specify.
- `cursor.fetchall` retrieves **all** the rows that make up the results.

For now, let’s use the `cursor.fetchall` method to retrieve all the results from the above query, assigning the results to a variable called `res`, then displaying the contents of `res` at the `>>>` prompt:

Get all the data returned from MySQL.

```
>>> res = cursor.fetchall()
>>> res
[('log',)]
```

Display the results.

The contents of `res` look a little weird, don’t they? You were probably expecting to see a single word here, as we know from earlier that our database (`vsearchlogDB`) contains a single table called `log`. However, what’s returned by `cursor.fetchall` is always a *list of tuples*, even when there’s only a single piece of data returned (as is the case above). Let’s look at another example that returns more data from MySQL.

Our next query, `describe log`, queries for the information about the `log` table as stored in the database. As you’ll see below, the information is shown *twice*: once in its raw form (which is a little messy) and then over multiple lines. Recall that the result returned by `cursor.fetchall` is a list of tuples.

Here’s `cursor.fetchall` in action once more:

Take the SQL query... ...then send it to the server... ...and then access the results.

It looks a little messy, but this is a list of tuples.

```
>>> _SQL = """describe log"""
>>> cursor.execute(_SQL)
>>> res = cursor.fetchall()
>>> res
[('id', 'int(11)', 'NO', 'PRI', None, 'auto_increment'), ('ts',
'timestamp', 'NO', '', 'CURRENT_TIMESTAMP', ''), ('phrase',
'varchar(128)', 'NO', '', None, ''), ('letters', 'varchar(32)',
'NO', '', None, ''), ('ip', 'varchar(16)', 'NO', '', None, ''),
('browser_string', 'varchar(256)', 'NO', '', None, ''), ('results',
'varchar(64)', 'NO', '', None, '')]
```

Take each row in the results... ...and display it on its own line.

Each tuple from the list of tuples is now on its own line.

```
>>> for row in res:
        print(row)

('id', 'int(11)', 'NO', 'PRI', None, 'auto_increment')
('ts', 'timestamp', 'NO', '', 'CURRENT_TIMESTAMP', '')
('phrase', 'varchar(128)', 'NO', '', None, '')
('letters', 'varchar(32)', 'NO', '', None, '')
('ip', 'varchar(16)', 'NO', '', None, '')
('browser_string', 'varchar(256)', 'NO', '', None, '')
('results', 'varchar(64)', 'NO', '', None, '')
```

Look closely. It’s the same data.

```
mysql> describe log;
+----------------+--------------+------+-----+-------------------+----------------+
| Field          | Type         | Null | Key | Default           | Extra          |
+----------------+--------------+------+-----+-------------------+----------------+
| id             | int(11)      | NO   | PRI | NULL              | auto_increment |
| ts             | timestamp    | NO   |     | CURRENT_TIMESTAMP |                |
| phrase         | varchar(128) | NO   |     | NULL              |                |
| letters        | varchar(32)  | NO   |     | NULL              |                |
| ip             | varchar(16)  | NO   |     | NULL              |                |
| browser_string | varchar(256) | NO   |     | NULL              |                |
| results        | varchar(64)  | NO   |     | NULL              |                |
+----------------+--------------+------+-----+-------------------+----------------+
```

The per-row display above may not look like much of an improvement over the raw output, but compare it to the output displayed by the MySQL console from earlier (shown below). What’s shown above is the same data as what’s shown below, only now the data is in a Python data structure called `res`:

DB-API UP CLOSE, 3 OF 3

Let's use an `insert` query to add some sample data to the `log` table.

It's tempting to assign the query shown below (which we've written over multiple lines) to the `_SQL` variable, then call `cursor.execute` to send the query to the server:

```
>>> _SQL = """insert into log
            (phrase, letters, ip, browser_string, results)
            values
            ('hitch-hiker', 'aeiou', '127.0.0.1', 'Firefox', "{'e', 'i'}")"""
>>> cursor.execute(_SQL)
```

When composing your query, use DB-API placeholders instead of actual data values.

```
>>> _SQL = """insert into log
            (phrase, letters, ip, browser_string, results)
            values
            (%s, %s, %s, %s, %s)"""
>>> cursor.execute(_SQL, ('hitch-hiker', 'xyz', '127.0.0.1', 'Safari', 'set()'))
```

Don't get us wrong, what's shown above does work. However, *hardcoding* the data values in this way is rarely what you'll want to do, as the data values you store in your table will likely change with every `insert`. Remember: you plan to log the details of each web request to the `log` table, which means these data values *will* change with every request, so hardcoding the data in this way would be a disaster.

To avoid the need to hardcode data (as shown above), Python's DB-API lets you position "data placeholders" in your query string, which are filled in with the actual values when you call `cursor.execute`. In effect, this lets you reuse a query with many different data values, passing the values as arguments to the query just before it's executed. The placeholders in your query are stringed values, and are identified as `%s` in the code below.

Compare these commands below with those shown above:

There are two things to note above. First, instead of hardcoding the actual data values in the SQL query, we used the `%s` placeholder, which tells DB-API to expect a stringed value to be substituted into the query prior to execution. As you can see, there are five `%s` placeholders above, so the second thing to note is that `cursor.execute` call is going to expect five additional parameters when called. The only problem is that `cursor.execute` doesn't accept just *any* number of parameters; it accepts *at most* two.

How can this be?

Looking at the last line of code shown above, it's clear that `cursor.execute` accepts the *five* data values provided to it (without complaint), so what gives?

Take another, closer look at that line of code. See the pair of parentheses around the data values? The use of parentheses turns the five data values into a single tuple (containing the individual data values). In effect, the above line of code supplies two arguments to `cursor.execute`: the placeholder-containing query, as well as a single tuple of data values.

So, when the code on this page executes, data values are inserted into the `log` table, right? Well...not quite.

When you use `cursor.execute` to send data to a database system (using the `insert` query), the data may not be saved to the database immediately. This is because writing to a database is an **expensive** operation (from a processing-cycle perspective), so many database systems cache `inserts`, then apply them all at once later. This can sometimes mean the data you think is in your table isn't there *yet*, which can lead to problems.

For instance, if you use `insert` to send data to a table, then immediately use `select` to read it back, the data may not be available, as it is still in the database system's cache waiting to be written. If this happens, you're out of luck, as the `select` fails to return any data. Eventually, the data is written, so it's not lost, but this default caching behavior may not be what you desire.

If you are happy to take the performance hit associated with a database write, you can force your database system to commit all potentially cached data to your table using the `conn.commit` method. Let's do that now to ensure the two `insert` statements from the previous page are applied to the `log` table. With your data written, you can now use a `select` query to confirm the data values are saved:

"Force" any cached data to be written to the table.

```
>>> conn.commit()
>>> _SQL = """select * from log"""
>>> cursor.execute(_SQL)
>>> for row in cursor.fetchall():
      print(row)
```

Retrieve the just-written data.

Here's the "id" value MySQL automatically assigned to this row...

```
(1, datetime.datetime(2016, 3, ..., "{'e', 'i'}")
(2, datetime.datetime(2016, 3, ..., 'set()')
```

...and here's what it filled in for "ts" (timestamp).

We've abridged the output to make it fit on this page.

From the above you can see that MySQL has automatically determined the correct values to use for `id` and `ts` when data is inserted into a row. The data returned from the database server is (as before) a list of tuples. Rather than save the results of `cursor.fetchall` to a variable that is then iterated over, we’ve used `cursor.fetchall` directly in a `for` loop in this code. Also, don’t forget: a tuple is an immutable list and, as such, supports the usual square bracket access notation. This means you can index into the `row` variable used within the above `for` loop to pick out individual data items as needed. For instance, `row[2]` picks out the phrase, `row[3]` picks out the letters, and `row[-1]` picks out the results.

DB-API Step 6: Close your cursor and connection

With your data committed to its table, tidy up after yourself by closing the cursor as well as the connection:

```
>>> cursor.close()
True
>>> conn.close()
```

It’s always a good idea to tidy up.

Note that the cursor confirms successful closure by returning `True`, while the connection simply shuts down. It’s always a good idea to close your cursor and your connection when they’re no longer needed, as your database system has a finite set of resources. Over at *Head First Labs*, the programmers like to keep their database cursors and connections open for as long as required, but no longer.

Task 4: Create Code to Work with Our Webapp’s Database and Tables

With the six *steps* of the DB-API *Up Close* completed, you now have the code needed to interact with the `log` table, which means you’ve completed Task 4: *Create code to work with our webapp’s database and tables.*

	Install MySQL on your computer.
	Install a MySQL Python driver.
	Create the database and tables.
	Create code to read/write data.

NOTE

Our task list is done!

Let’s review the code you can use (in its entirety):

```
dbconfig = { 'host': '127.0.0.1',
             'user': 'vsearch',
             'password': 'vsearchpasswd',
             'database': 'vsearchlogDB', }

import mysql.connector

conn = mysql.connector.connect(**dbconfig)

cursor = conn.cursor()

_SQL = """insert into log
          (phrase, letters, ip, browser_string, results)
          values
          (%s, %s, %s, %s, %s)"""

cursor.execute(_SQL, ('galaxy', 'xyz', '127.0.0.1', 'Opera', "{'x', 'y'}"))

conn.commit()

_SQL = """select * from log"""

cursor.execute(_SQL)

for row in cursor.fetchall():
    print(row)

cursor.close()

conn.close()
```

Define your connection characteristics.

Import the database driver.

Establish a connection and create a cursor.

Assign a query to a string (note the five placeholder arguments).

Send the query to the server, remembering to provide values for each of the required arguments (in a tuple).

Force the database to write your data.

Retrieve the (just written) data from the table, displaying the output row by row.

Tidy up when you’re done.

With each of the four tasks now complete, you’re ready to adjust your webapp to log the web request data to your MySQL database system as opposed to a text file (as is currently the case). Let's start doing this now.

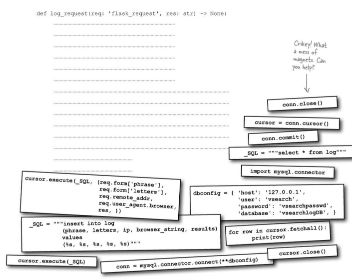
DATABASE MAGNETS

Take another look at the `log_request` function from the last chapter.

Recall that this small function accepts two arguments: a web request object, and the results of the vsearch:

```
def log_request(req: 'flask_request', res: str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req.form, req.remote_addr, req.user_agent, res, fi
```

Your job is to replace this function's suite with code that logs to your database (as opposed to the text file). The `def` line is to remain unchanged. Decide on the magnets you need from those scattered at the bottom on this page, then position them to provide the function's code:

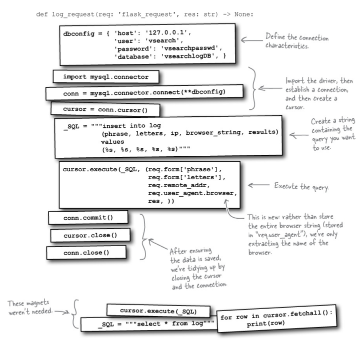


DATABASE MAGNETS SOLUTION

You were to take another look at the `log_request` function from the last chapter:

```
def log_request(req: 'flask_request', res: str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req.form, req.remote_addr, req.user_agent, res, fi
```

Your job was to replace this function's suite with code that logs to your database. The `def` line was to remain unchanged. You were to decide which magnets you needed from those scattered at the bottom on the page.



TEST DRIVE

Change the code in your `vsearch4web.py` file to replace the original `log_request` function's code with that from the last page. When you have saved your code, start up this latest version of your webapp at a command prompt. Recall that on *Windows*, you need to use this command:

```
C:\webapps> py -3 vsearch4web.py
```

While on *Linux* or *Mac OS X*, use this command:

```
$ python3 vsearch4web.py
```

Your webapp should start running at this web address:

http://127.0.0.1:5000/ (http://127.0.0.1:5000/)

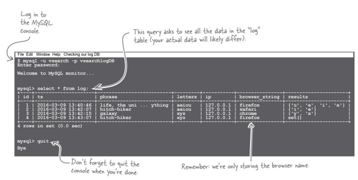
Use your favorite web browser to perform a few searches to confirm that your webapp runs fine.

There are two points we'd like to make here:

- Your webapp performs exactly as it did before: each search returns a "results page" to the user.
- Your users have no idea that the search data is now being logged to a database table as opposed to a text file.

Regrettably, you can't use the */viewlog* URL to view these latest log entries, as the function associated with that URL (`view_the_log`) only works with the `vsearch.log` text file (not the database). We'll have more to say about fixing this over the page.

For now, let's conclude this *Test Drive* by using the MySQL console to confirm that this newest version of `log_request` is logging data to the `log` table. Open another terminal window and follow along (note: we've reformatted and abridged our output to make it fit on this page):



Storing Data Is Only Half the Battle

Having run though the *Test Drive* on the last page, you've now confirmed that your Python DB-API–compliant code in `log_request` does indeed store the details of each web request in your `log` table.

Take a look at the most recent version of the `log_request` function once more (which includes a docstring as its first line of code):

```
def log_request(req: 'flask_request', res: str) -> None:
    """Log details of the web request and the results."""
    dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }

    import mysql.connector

    conn = mysql.connector.connect(**dbconfig)
    cursor = conn.cursor()
    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
    conn.commit()
    cursor.close()
    conn.close()
```

Experienced Python programmers may well look at this function's code and let out a gasp of disapproval. You'll learn why in a few pages' time.

THIS NEW FUNCTION IS A BIG CHANGE

There's a lot more code in the `log_request` function now than when it operated on a simple text file, but the extra code is needed to interact with MySQL (which you're going to use to answer questions about your logged data at the end of this chapter), so this new, bigger, more complex version of `log_request` appears justified.

However, recall that your webapp has another function, called `view_the_log`, which retrieves the data from the `vsearch.log` log file and displays it in a nicely formatted web page. We now need to update the `view_the_log` function's code to retrieve its data from the `log` table in the database, as opposed to the text file.

The question is: what's the best way to do this?

How Best to Reuse Your Database Code?

You now have code that logs the details of each of your webapp's requests to MySQL. It shouldn't be too much work to do something similar in order to retrieve the data from the `log` table for use in the `view_the_log` function. The question is: what's the best way to do this? We asked three programmers our question...and got three different answers.

In its own way, each of these suggestions is valid, if a little suspect (especially the first one). What may come as a surprise is that, in this case, a Python programmer would be unlikely to embrace any of these proposed solutions *on their own*.

Consider What You're Trying to Reuse

Let's take another look our database code in the `log_request` function.

It should be clear that there are parts of this function we can reuse when writing additional code that interacts with a database system. Thus, we've annotated the function's code to highlight the parts we think are reusable, as opposed to the parts that are specific to the central idea of what the `log_request` function actually does:

Based on this simple analysis, the `log_request` function has three groups of code statements:

- statements that can be easily reused (such as the creation of `conn` and `cursor`, as well as the calls to `commit` and `close`);
- statements that are specific to the problem but still need to be reusable (such as the use of the `dbconfig` dictionary); and
- statements that cannot be reused (such as the assignment to `_SQL` and the call to `cursor.execute`). Any further interactions with MySQL are very likely to require a different SQL query, as well as different arguments (if any).

What About That Import?

Nope, we didn't forget.

The `import mysql.connector` statement wasn't forgotten when we considered reusing the `log_request` function's code.

This omission was deliberate on our part, as we wanted to call out this statement for special treatment. The problem isn't that we don't want to reuse that statement; it's that it shouldn't appear in the function's suite!

BE CAREFUL WHEN POSITIONING YOUR IMPORT STATEMENTS

We mentioned a few pages back that experienced Python programmers may well look at the `log_request` function's code and let out a gasp of disapproval. This is due to the inclusion of the `import mysql.connector` line of code in the function's suite. And this disapproval is in spite of the fact that our most recent *Test Drive* clearly demonstrated that this code works. So, what's the problem?

The problem has to do with what happens when the interpreter encounters an `import` statement in your code: the imported module is read in full, then executed by the interpreter. This behavior is fine when your `import` statement occurs *outside of a function*, as the imported module is (typically) only read *once*, then executed *once*.

However, when an `import` statement appears *within* a function, it is read *and* executed **every time the function is called**. This is regarded as an extremely wasteful practice (even though, as we've seen, the interpreter won't stop you from putting an `import` statement in a function). Our advice is simple: think carefully about where you position your `import` statements, and don't put any inside a function.

Consider What You're Trying to Do

In addition to looking at the code in `log_request` from a reuse perspective, it's also possible to categorize the function's code based on *when* it runs.

The "guts" of the function is the assignment to the `_SQL` variable and the call to `cursor.execute`. Those two statements most patently represent *what* the function is meant to **do**, which—to be honest—is the most important bit. The function's initial statements define the connection characteristics (in `dbconfig`), then create a connection and cursor. This **setup** code always has to run *before* the guts of the function. The last three statements in the function (the single `commit` and the two `close`s) execute *after* the guts of the function. This is **teardown** code, which performs any required tidying up.

With this *setup, do, teardown* pattern in mind, let's look at the function once more. Note that we've repositioned the `import` statement to execute outside of the `log_request` function's suite (so as to avoid any further disapproving gasps):

Wouldn't it be neat if there were a way to reuse this setup, do, teardown pattern?

You've Seen This Pattern Before

Consider the pattern we just identified: setup code to get ready, followed by code to do what needs to be done, and then teardown code to tidy up. It may not be immediately obvious, but in the previous chapter, you encountered code that conforms to this pattern. Here it is again:

Recall how the `with` statement *manages the context* within which the code in its suite runs. When you're working with files (as in the code above), the `with` statement arranges to open the named file and return a variable representing the file stream. In this example, that's the `tasks` variable; this is the **setup** code. The suite associated with the `with` statement is the **do** code; here that's the `for` loop, which does the actual work (a.k.a. "the important bit"). Finally, when you use `with` to open a file, it comes with the promise that the open file will be closed when the `with`'s suite terminates. This is the **teardown** code.

It would be neat if we could integrate our database programming code into the `with` statement. Ideally, it would be great if we could write code like this, and have the `with` statement take care of all the database setup and teardown details:

The *good news* is that Python provides the **context management protocol**, which enables programmers to hook into the `with` statement as needed. Which brings us to the *bad news*...

The Bad News Isn't Really All That Bad

At the bottom of the last page, we stated that the *good news* is that Python provides a context management protocol that enables programmers to hook into the `with` statement as and when required. If you learn how to do this, you can then create a context manager called `UseDatabase`, which can be used as part of a `with` statement to talk to your database.

The idea is that the setup and teardown "boilerplate" code that you've just written to save your webapp's logging data to a database can be replaced by a single `with` statement that looks like this:

The *bad news* is that creating a context manager is complicated by the fact that you need to know how to create a Python class in order to successfully hook into the protocol.

Consider that up until this point in this book, you've managed to write a lot of usable code without having to create a class, which is pretty good going, especially when you consider that some programming languages don't let you do *anything* without first creating a class (we're looking at *you*, Java).

However, it's now time to bite the bullet (although, to be honest, creating a class in Python is nothing to be scared of).

As the ability to create a class is generally useful, let's deviate from our current discussion about adding database code to our webapp, and dedicate the next (short) chapter to classes. We'll be showing you just enough to enable you to create the `UseDatabase` context manager. Once that's done, in the chapter after that, we'll return to our database code (and our webapp) and put our newly acquired class-writing abilities to work by writing the `UseDatabase` context manager.

Chapter 7's Code

PREV
6. Storing and Manipulating Data: Where to Put Your Data

NEXT
8. A Little Bit of Class: Abstracting Behavior and State