# Chapter 9. The Context Management Protocol: Hooking into Python's with Statement



It's time to take what you've just learned and put it to work.

<u>Chapter 7</u> discussed using a **relational database** with Python, while <u>Chapter 8</u> provided an introduction to using **classes** in your Python code. In this chapter, both of these techniques are combined to produce a **context manager** that lets us extend the with statement to work with relational database systems. In this chapter, you'll hook into the with statement by creating a new class, which conforms to Python's **context management protocol**.

# What's the Best Way to Share Our Webapp's Database Code?

During <u>Chapter 7</u> you created database code in your <code>log\_request</code> function that worked, but you had to pause to consider how best to share it. Recall the suggestions from the end of <u>Chapter 7</u>:



At the time, we proposed that each of these suggestions was valid, but believed Python programmers would be unlikely to embrace any of these proposed solutions *on their own*. We decided that a better strategy was to hook into the context management protocol using the with statement, but in order to do that, you needed to learn a bit about classes. They were the subject of the last chapter. Now that you know how to create a class, it's time to return to the task at hand: creating a context manager to share your webapp's database code.

# Consider What You're Trying to Do, Revisited

Below is our database management code from <u>Chapter 7</u>. This code is currently part of our Flask webapp. Recall how this code connected to our MySQL database, saved the details of the web request to the log table, committed any *unsaved* data, and then disconnected from the database:

### **HOW BEST TO CREATE A CONTEXT MANAGER?**

Before getting to the point where you can transform the above code into something that can be used as part of a with statement, let's discuss how this is achieved by conforming to the context management protocol. Although there is support for creating simple context managers in the standard library (using the contextlib module), creating a class that conforms to the protocol is regarded as the correct approach when you're using with to control some external object, such as a database connection (as is the case here).

to the "log" database table.

With that in mind, let's take a look at what's meant by "conforming to the context management protocol."

# **Managing Context with Methods**

The context management protocol sounds intimidating and scary, but it's actually quite simple. It dictates that any class you create must define at least two magic methods: \_\_enter\_\_ and \_\_exit\_\_. This is the protocol. When you adhere to the protocol, your class can hook into the with statement.

A protocol is an agreed procedure (or set of rules) that is to be adhered to.

### **DUNDER "ENTER" PERFORMS SETUP**

When an object is used with a with statement, the interpreter invokes the object's \_\_enter\_ method *before* the with statement's suite starts. This provides an opportunity for you to perform any required setup code within dunder enter.

The protocol further states that dunder enter can (but doesn't have to) return a value to the with statement (you'll see why this is important in a little bit).

### **DUNDER "EXIT" DOES TEARDOWN**

As soon as the with statement's suite ends, the interpreter *always* invokes the object's \_\_exit\_\_ method. This occurs *after* the with's suite terminates, and it provides an opportunity for you to perform any required teardown.

As the code in the with statement's suite may fail (and raise an exception), dunder exit has to be ready to handle this if it happens. We'll return to this issue when we create the code for our dunder exit method later in this chapter.

### If your class defines dunder "enter" and dunder "exit", it's a context manager.

If you create a class that defines \_\_enter\_\_ and \_\_exit\_\_, the class is automatically regarded as a context manager by the interpreter and can, as a consequence, hook into (and be used with) with. In other words, such a class *conforms* to the context management protocol, and *implements* a context manager.

# (AS YOU KNOW) DUNDER "INIT" INITIALIZES

In addition to dunder <code>enter</code> and dunder <code>exit</code>, you can add other methods to your class as needed, including defining your own <code>\_\_init\_</code> method. As you know from the last chapter, defining dunder <code>init</code> lets you perform additional object initialization.

Dunder <code>init</code> runs <code>before \_\_enter\_</code> (that is, <code>before your context manager's setup code executes)</code>.

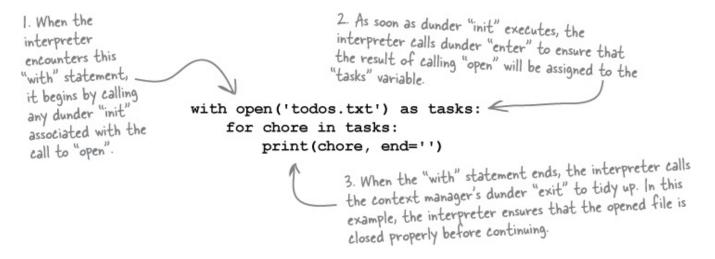
It's not an absolute requirement to define \_\_init\_\_ for your context manager (as \_\_enter\_\_ and \_\_exit\_\_ are all you really need), but it can sometimes be useful to do so, as it lets you separate any initialization activity from any setup activity. When we create a context manager for use with our database connections (later in this chapter), we define \_\_init\_\_ to initialize our database connection credentials. Doing so isn't absolutely necessary, but we think it helps to keep things nice and tidy, and makes our context manager class code easier to read and understand.

# You've Already Seen a Context Manager in Action

You first encountered a with statement back in <u>Chapter 6</u> when you used one to ensure a previously opened file was *automatically* closed once its associated with statement terminated. Recall how this code opened the todos.txt file, then read and displayed each line in the file one by one, before automatically closing the file (thanks to the fact that open is a context manager):

```
with open('todos.txt') as tasks:
    for chore in tasks:
        print(chore, end='')
Your first-ever
"with" statement
(borrowed from
Chapter b).
```

Let's take another look at this with statement, highlighting where dunder enter, dunder exit, and dunder init are invoked. We've numbered each of the annotations to help you understand the order the dunders execute in. Note that we don't see the initialization, setup, or teardown code here; we just know (and trust) that those methods run "behind the scenes" when needed:



### WHAT'S REQUIRED FROM YOU

Before we get to creating our very own context manager (with the help of a new class), let's review what the context management protocol expects you to provide in order to hook into the with statement. You must create a class that provides:

- 1. an \_\_init\_\_ method to perform initialization (if needed);
- 2. an enter method to do any setup; and
- 3. an \_\_exit\_\_ method to do any teardown (a.k.a. tidying-up).

Armed with this knowledge, let's now create a context manager class, writing these methods one by one, while borrowing from our existing database code as needed.

# **Create a New Context Manager Class**

To get going, we need to give our new class a name. Additionally, let's put our new class code into its own file, so that we can easily reuse it (remember: when you put Python code in a separate file it becomes a module, which can be imported into other Python programs as required).

Let's call our new file DBcm.py (short for *database context manager*), and let's call our new class UseDatabase. Be sure to create the DBcm.py file in the same folder that currently contains your webapp code, as it's your webapp that's going to import the UseDatabase class (once you've written it, that is).

# Remember: use CamelCase when naming a class in Python.

Using your favorite editor (or IDLE), create a new edit window, and then save the new, empty file as DBcm.py. We know that in order for our class to conform to the context management protocol it has to:

- 1. provide an \_\_init\_\_ method that performs initialization;
- 2. provide an \_\_enter\_\_ method that includes any setup code; and
- 3. provide an \_\_exit\_\_ method that includes any teardown code.

For now, let's add three "empty" definitions for each of these required methods to our class code. An empty method contains a single pass statement. Here's the code so far:

```
This is what our

    DBcm.py - /Users/paul/Desktop/_NewBook/ch09/webapp/DBcm.py (3.5.1)

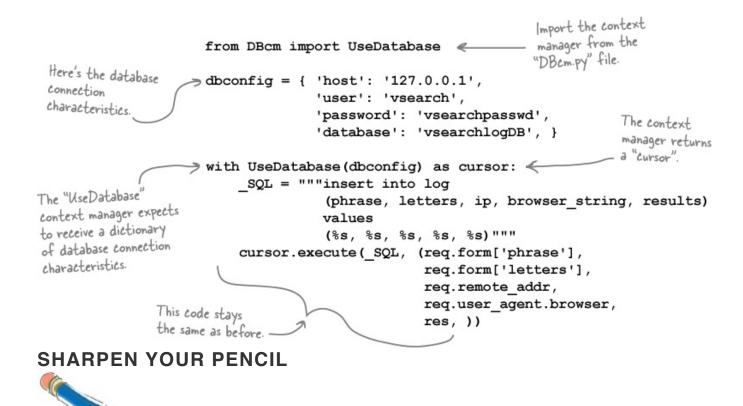
"DBem.py" file
                             import mysql.connector
looks like in IDL
At the moment,
                             class UseDatabase:
 it's made up from
 a single "import"
                                  def init (self):
 statement,
                                       pass
 together with
                                  def enter (self):
 a class called
                                       pass
 "UseDatabase"
 that contains
                                  def exit (self):
  three "empty"
                                       pass
  methods.
                                                                             Ln: 14 Col: 0
```

Note how at the top of the DBCm.py file we've included an import statement, which includes the *MySQL Connector* functionality (which our new class depends on).

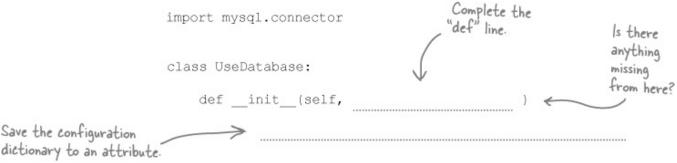
All we have to do now is move the relevant bits from the log\_request function into the correct method within the useDatabase class. Well...when we say we, we actually mean you. It's time to roll up your sleeves and write some method code.

# Initialize the Class with the Database Config

Let's remind ourselves of how we intend to use the useDatabase context manager. Here's the code from the last chapter, rewritten to use a with statement, which itself uses the useDatabase context manager that you're about to write:



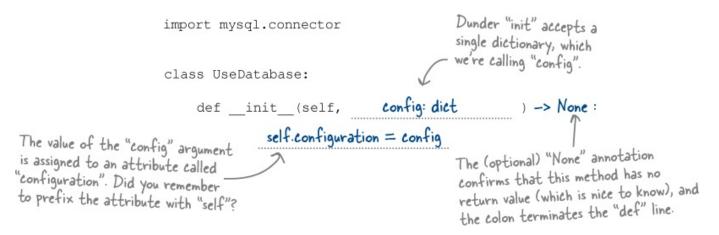
Let's start with the \_\_init\_\_ method, which we'll use to initialize any attributes in the UseDataBaseclass. Based on the usage shown above, the dunder init method accepts a single argument, which is a dictionary of connection characteristics called config (which you'll need to add to the def line below). Let's arrange for config to be saved as an attribute called configuration. Add the code required to save the dictionary to the configuration attribute to dunder init's code:



### SHARPEN YOUR PENCIL SOLUTION



You started with the \_\_init\_\_ method, which was to initialize any attributes in the useDataBase class. The dunder init method accepts a single argument, which is a dictionary of connection characteristics called config (which you needed to add to the def line below). You were to arrange for config to be saved to an attribute called configuration. You were to add the code required to save the dictionary to the configuration attribute in dunder init's code:



### YOUR CONTEXT MANAGER BEGINS TO TAKE SHAPE

With the dunder init method written, you can move on to coding the dunder enter method (\_enter\_). Before you do, make sure the code you've written so far matches ours, which is shown below in IDLE:

```
Make sure

your dunder

"init"

matches

ours.

def __init__(self, config: dict) -> None:
    self.configuration = config

def __enter__(self):
    pass

def __exit__(self):
    pass

Ln: 14 Col: 0
```

# **Perform Setup with Dunder "enter"**

The dunder enter method provides a place for you to execute the setup code that needs to be executed *before* the suite in your with statement runs. Recall the code from the log\_request function that handles this setup:

```
dbconfig = { 'host': '127.0.0.1',

'user': 'vsearch',

'password': 'vsearchpasswd',

'database': 'vsearchlogDB', }

"log_request"

function:

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

cursor = conn.cursor()

_SQL = """insert into log

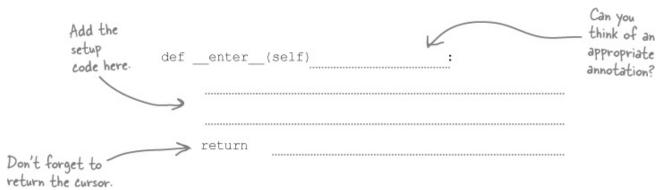
(phrase, letters, ip, browser_string, results)
...
```

This setup code uses the connection characteristics dictionary to connect to MySQL, then creates a database cursor on the connection (which we'll need to send commands to the database from our Python code). As this setup code is something you'll do every time you write code to talk to your database, let's do this work in your context manager class instead so that you can more easily reuse it.

### SHARPEN YOUR PENCIL



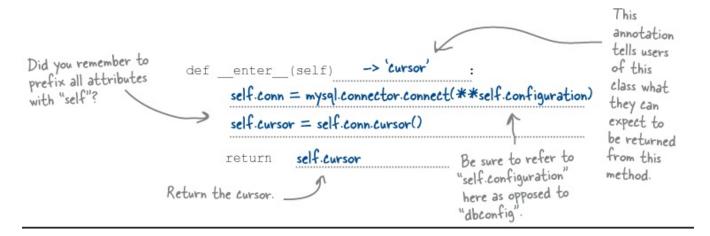
The dunder enter method (\_\_enter\_\_) needs to use the configuration characteristics stored in self.configuration to connect to the database and create a cursor. Other than the mandatory self argument, dunder enter takes no other arguments, but needs to return the cursor. Complete the code for the method below:



### SHARPEN YOUR PENCIL SOLUTION



The dunder enter method (\_\_enter\_\_) uses the configuration characteristics stored in self.configuration to connect to the database and create a cursor. Other than the mandatory self argument, dunder enter takes no other arguments, but needs to return the cursor. You were to complete the code for the method below:



### DON'T FORGET TO PREFIX ALL ATTRIBUTES WITH "SELF"

You may be surprised that we designated conn and cursor as attributes in dunder enter (by prefixing each with self). We did this in order to ensure both conn and cursor survive when the method ends, as both variables are needed in the \_\_exit\_\_ method. To ensure this happens, we added the self prefix to both the conn and cursor variables; doing so adds them to the class's attribute list.

Before you get to writing dunder exit, confirm that your code matches ours:

```
DBcm.py - /Users/paul/Desktop/_NewBook/ch09/webapp/DBcm.py (3.5.1)*

import mysql.connector

class UseDatabase:

def __init__ (self, config: dict) -> None:
    self.configuration = config

def __enter__ (self) -> 'cursor':
    self.conn = mysql.connector.connect(**self.configuration)
    self.cursor = self.cursor()
    return self.cursor

def __exit__ (self):
    pass

Ln:16 Col:0
```

# **Perform Teardown with Dunder "exit"**

The dunder exit method provides a place for you to execute the teardown code that needs to be run when your with statement terminates. Recall the code from the log\_request function that handles teardown:

The teardown code commits any data to the database, then closes the cursor and the connection. This teardown happens *every* time you interact with the database, so let's add this code to your context manager class by moving these three lines into dunder exit.

Before you do this, however, you need to know that there's a complication with dunder exit, which has to do with handling any exceptions that might occur within the with's suite. When something goes wrong, the interpreter *always* notifies \_\_exit\_\_ by passing three arguments into the method: exc\_type, exc\_value, and exc\_trace. Your def line needs to take this into account, which is why we've added the three arguments to the code below. Having said that, we're going to *ignore* this exception-handling mechanism for now, but will return to it in a later chapter when we discuss what can go wrong and how you can handle it (so stay tuned).

### SHARPEN YOUR PENCIL



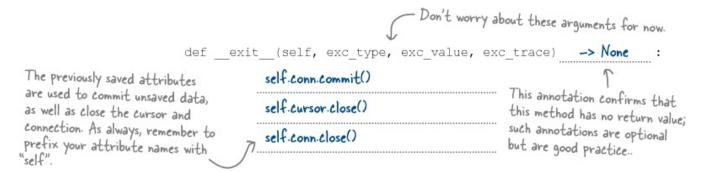
The teardown code is where you do your tidying up. For this context manager, tidying up involves ensuring any data is committed to the database prior to closing both the cursor and the connection. Add the code you think you need to the method below.

1	Don't worry about these arguments for now.
Add the teardown code here.	defexit(self, exc_type, exc_value, exc_trace):
code noto	

### SHARPEN YOUR PENCIL SOLUTION

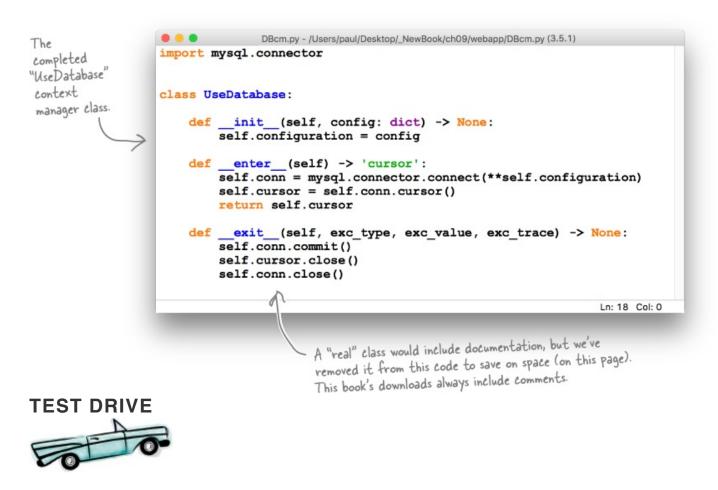


The teardown code is where you do your tidying up. For this context manager, tidying up involves ensuring any data is committed to the database prior to closing both the cursor and the connection. You were to add the code you think you need to the method below.



### YOUR CONTEXT MANAGER IS RE ADY FOR TESTING

With the dunder exit code written, it's now time to test your context manager prior to integrating it into your webapp code. As has been our custom, we'll first test this new code at Python's shell prompt (the >>>). Before doing this, perform one last check to ensure your code is the same as ours:



With the code for DBcm.py in an IDLE edit window, press F5 to test your context manager:

```
Import the
context manager
class from
the "DBcm.py"
module file.
                                                 Python 3.5.1 Shell
                     >>>
                     >>> from DBcm import UseDatabase
                     >>>
                     >>> dbconfig = { 'host': '127.0.0.1',
                                                                                                Put the
                                           'user': 'vsearch',
                                                                                                connection
                                           'password': 'vsearchpasswd',
'database': 'vsearchlogDB', }
Use the
                                                                                                characteristics
context
                     >>>
                                                                                                in a dictionary.
manager to
                     >>> with UseDatabase(dbconfig) as cursor:
                                SQL = """show tables"""
send some
                                cursor.execute ( SQL)
SQL to the
                                data = cursor.fetchall()
server and
get some
data back.
                     >>> data
                     [('log',)] <
                     >>>
                     >>>
                                                                            Ln: 45 Col: 4
                                             The returned data may look a little strange...until you
                                              remember that the "cursor.fetchall" call returns a list of
                                              tuples, with each tuple corresponding to a row of results
                                              (as returned from the database).
```

# THERE'S NOT MUCH CODE HERE, IS THERE?

Hopefully, you're looking at the code above and deciding there's not an awful lot to it. As you've successfully moved some of your database handling code into the usedatabase class, the initialization, setup, and teardown are now handled "behind the scenes" by your context manager. All you have to do is provide the connection characteristics and the SQL query you wish to execute—the context manager does all the rest. Your setup and teardown code is reused as part of the context manager. It's also clearer what the "meat" of this code is: getting data from the database and processing it. The context manager hides the details of connecting/disconnecting to/from the database (which are always going to be the same), thereby leaving you free to concentrate on what you're trying to do with your data.

Let's update your webapp to use your context manager.

# **Reconsidering Your Webapp Code, 1 of 2**

It's been quite a while since you've considered your webapp's code.

The last time you worked on it (in <u>Chapter 7</u>), you updated the log\_request function to save the webapp's web request to the MySQL database. The reason we started down the path to

learning about classes (in <u>Chapter 8</u>) was to determine the best way to share the database code you added to <code>log\_request</code>. We now know that the best way (for this situation) is to use the just-written <code>useDatabase</code> context manager class.

# Your webapp's code is in the "vsearch4web.py" file in your "webapp" folder.

In addition to amending <code>log\_request</code> to use the context manager, the other function in the code that we need to amend work with the data in the database is called <code>view\_the\_log</code> (which currently works with the <code>vsearch.log</code> text file). Before we get to amending both of these functions, let's remind ourselves of the current state of the webapp's code (on this page and the next). We've highlighted the bits that need to be worked on:

```
from flask import Flask, render template, request, escape
                   from vsearch import search4letters
                   import mysql.connector <
                                                      import "DBcm"
                   app = Flask( name )
                   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', }
                       conn = mysql.connector.connect(**dbconfig)
                       cursor = conn.cursor()
This code has to
                        SQL = """insert into log
be amended to use
                                  (phrase, letters, ip, browser string, results)
the "UseDatabase"
context manager.
                                  (%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()
```

# Reconsidering Your Webapp Code, 2 of 2

```
@app.route('/search4', methods=['POST'])
def do search() -> 'html':
    """Extract the posted data; perform the search; return results."""
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log request (request, results)
    return render template('results.html',
                            the title=title,
                            the phrase=phrase,
                            the letters=letters,
                            the results=results,)
@app.route('/')
@app.route('/entry')
def entry page() -> 'html':
    """Display this webapp's HTML form."""
    return render template('entry.html',
                            the title='Welcome to search4letters on the web!')
@app.route('/viewlog')
def view the log() -> 'html':
    """Display the contents of the log file as a HTML table."""
    contents = []
                                                             This code needs to be amended
    with open('vsearch.log') as log:
                                                             to use the data in the
        for line in log:
                                                             database via the "UseDatabase"
            contents.append([])
                                                             context manager.
            for item in line.split('|'):
                contents[-1].append(escape(item))
    titles = ('Form Data', 'Remote addr', 'User agent', 'Results')
    return render_template('viewlog.html',
                            the title='View Log',
                            the row titles=titles,
                            the data=contents,)
if __name__ == '__main_
    app.run (debug=True)
```

# Recalling the "log\_request" Function

When it comes to amending the log\_request function to use the useDatabase context manager, a lot of the work has already been done for you (as we showed you the code we were shooting for earlier).

Take a look at log\_request once more. At the moment, the database connection characteristics dictionary (dbconfig in the code) is defined within log\_request. As you'll want

to use this dictionary in the other function you have to amend (view\_the\_log), let's move it out of the log\_request's function so that you can share it with other functions as needed:

```
def log request(req: 'flask request', res: str) -> None:
                     dbconfig = { 'host': '127.0.0.1',
                                   'user': 'vsearch',
                                   'password': 'vsearchpasswd',
Let's move this
                                   'database': 'vsearchlogDB', }
dictionary out
of the function
                     conn = mysql.connector.connect(**dbconfig)
so it can be
                     cursor = conn.cursor()
shared with
                      SQL = """insert into log
other functions
                                 (phrase, letters, ip, browser_string, results)
as required.
                                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()
```

Rather than move <code>dbconfig</code> into our webapp's global space, it would be useful if we could somehow add it to our webapp's internal configuration.

As luck would have it, Flask (like many other web frameworks) comes with a built-in configuration mechanism: a dictionary (which Flask calls app.config) allows you to adjust some of your webapp's internal settings. As app.config is a regular Python dictionary, you can add your own keys and values to it as needed, which is what you'll do for the data in dbconfig.

The rest of  $log_request$ 's code can then be amended to use  $log_request$ .

Let's make these changes now.

# Amending the "log\_request" Function

Now that we've applied the changes to our webapp, our code looks like this:

```
vsearch4web.py - /Users/paul/Desktop/_NewBook/ch09/webapp/vsearch4web.py (3.5.1)
                  from flask import Flask, render template, request, escape
We changed
                  from vsearch import search4letters
the old
                                                                            We added the
"import"
                 from DBcm import UseDatabase
                                                                            connection
statement
                                                                            characteristics
                  app = Flask( name )
to this
                                                                            dictionary to the
updated one.
                                                                           webapp's configuration.
                  app.config['dbconfig'] = {'host': '127.0.0.1',
                                                'user': 'vsearch',
                                                'password': 'vsearchpasswd',
                                                'database': 'vsearchlogDB', }
                  def log request(req: 'flask_request', res: str) -> None:
                       """Log details of the web request and the results."""
                     with UseDatabase(app.config['dbconfig']) as cursor:
                           SQL = """insert into log
                                      (phrase, letters, ip, browser string, results)
                                      values
We adjusted
                                      (%s, %s, %s, %s, %s)"""
the code to use
                           cursor.execute(_SQL, (req.form['phrase'],
"UseDatabase",
                                                    req.form['letters'],
being sure to pass
                                                    req.remote addr,
in the database
                                                    req.user agent.browser,
                                                    res, ))
configuration
from "app.config"
                                                                                    Ln: 10 Col: 0
```

Near the top of the file, we've replaced the import mysql.connector statement with an import statement that grabs useDatabase from our DBcm module. The DBcm.py file itself includes the import mysql.connector statement in its code, hence the removal of import mysql.connector from this file (as we don't want to import it twice).

We've also moved the database connection characteristics dictionary into our webapp's configuration. And we've amended log request's code to use our context manager.

After all your work on classes and context managers, you should be able to read and understand the code shown above.

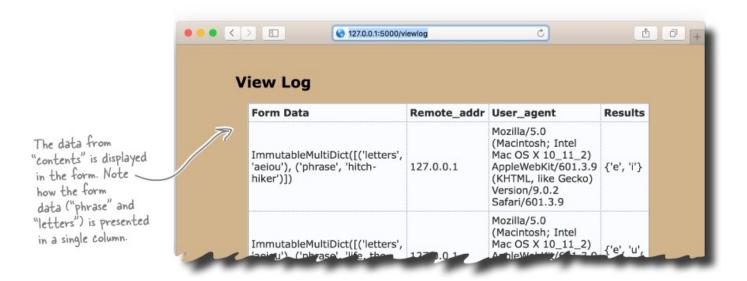
Let's now move onto amending the view\_the\_log function. Make sure your webapp code is amended to be exactly like ours above before turning the page.

# Recalling the "view\_the\_log" Function

Let's take a long, hard look at the code in view\_the\_log, as it's been quite a while since you've considered it in detail. To recap, the current version of this function extracts the logged data from the vsearch.log text file, turns it into a list of lists (called contents), and then sends the data to a template called viewlog.html:

```
@app.route('/viewlog')
Grab each line of
                  def view the log() -> 'html':
data from the file,
and then transform
                       contents = []
it into a list of
                       with open('vsearch.log') as log:
escaped items, which
                        for line in log:
are appended to the
                                contents.append([])
"contents" list.
                                for item in line.split('|'):
                                     contents[-1].append(escape(item))
                       titles = ('Form Data', 'Remote addr', 'User agent', 'Results')
                       return render template ('viewlog.html',
                                                 the title='View Log',
                     The processed log -
                                                 the row titles=titles,
                     data is sent to the
                                               the data=contents,)
                     template for display.
```

Here's what the output looks like when the <code>viewlog.html</code> template is rendered with the data from the <code>contents</code> list of lists. This functionality is currently available to your webapp via the /viewlog URL:



# It's Not Just the Code That Changes

Before diving in and changing the code in <code>view\_the\_log</code> to use your context manager, let's pause to consider the data as stored in the <code>log</code> table in your database. When you tested your initial <code>log\_request</code> code in <a href="Chapter 7">Chapter 7</a>, you were able to log into the MySQL console, then check that the data was saved. Recall this MySQL console session from earlier:

If you consider the above data in relation to what's currently stored in the vsearch.log file, it's clear that some of the processing view\_the\_log does is no longer needed, as the data is now stored in a table. Here's a snippet of what the log data looks like in the vsearch.log file:

Some of the code currently in <code>view\_the\_log</code> is only there because the log data is currently stored as a collection of long strings (delimited by vertical bars) in the <code>vsearch.log</code> file. That format worked, but we did need to write extra code to make sense of it.

This is not the case with data in the log table, as it is "structured by default." This should mean you don't need to perform any additional processing within <code>view\_the\_log</code>: all you have to do is extract the data from the table, which—happily—is returned to you as a list of tuples (thanks to DB-API's <code>fetchall</code> method).

On top of this, the data in the log table separates the value for phrase from the value for letters. If you make a small change to your template-rendering code, the output produced can display five columns of data (as opposed to the current four), making what the browser displays even more useful and easier to read.

# Amending the "view\_the\_log" Function

Based on everything discussed on the last few pages, you've two things to do to amend your current view\_the\_log code:

- 1. Grab the log data from the database table (as opposed to the file).
- 2. Adjust the titles list to support five columns (as opposed to four).

If you're scratching your head and wondering why this small list of amendments doesn't include adjusting the <code>viewlog.html</code> template, wonder no more: you don't need to make any changes to *that* file, as the current template quite happily processes any number of titles and any amount of data you send to it.

Here's the view\_the\_log function's current code, which you are about to amend:

```
@app.route('/viewlog')
                def view the log() -> 'html':
As a result of
task #1 above,
                     contents = []
this code
                     with open('vsearch.log') as log:
needs to be
                         for line in log:
replaced.
                              contents.append([])
                              for item in line.split('|'):
                                  contents[-1].append(escape(item))
                     titles = ('Form Data', 'Remote addr', 'User agent', 'Results')
     As a result
                     return render template('viewlog.html',
     of task #2
                                               the title='View Log',
     above, this
                                               the row titles=titles,
     line needs to
                                               the data=contents,)
      be amended
```

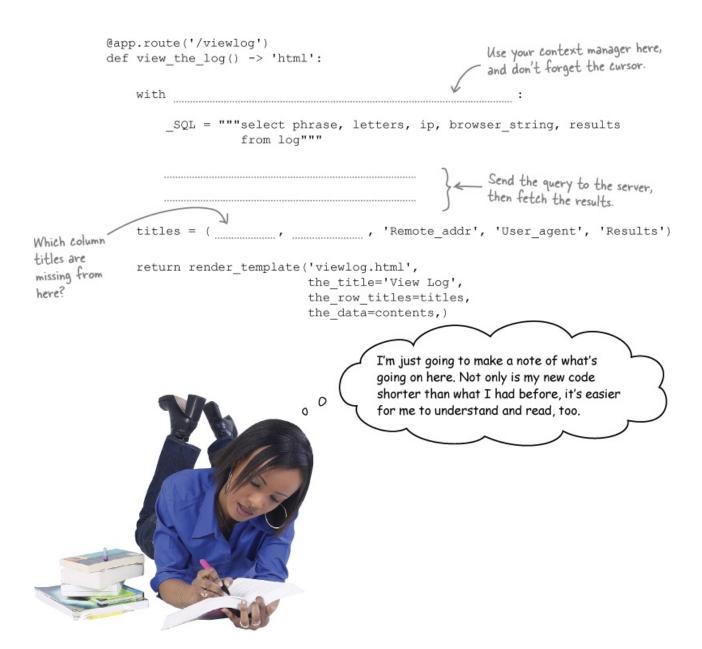
### HERE'S THE SQL QUERY YOU'LL NEED

Ahead of the next exercise (where you'll update the <code>view\_the\_log</code> function), here's an SQL query that, when executed, returns all the logged data stored in the webapp's MySQL database. The data is returned to your Python code from the database as a list of tuples. You'll need to use this query in the exercise on the next page:

```
select phrase, letters, ip, browser_string, results
from log
SHARPEN YOUR PENCIL
```



Here's the view\_the\_log function, which has to be amended to use the data in the log table. Your job is to provide the missing code. Be sure to read the annotations for hints on what you need to do:



# Yep—that was our goal all along.

By moving the log data into a MySQL database, you've removed the requirement to create, and then process, a custom text-based file format.

Also, by reusing your context manager, you've simplified your interactions with MySQL when working in Python. What's not to like?

### SHARPEN YOUR PENCIL SOLUTION



Here's the view\_the\_log function, which has to be amended to use the data in the log table. Your job was to provide the missing code.

```
This is the same line of code
            @app.route('/viewlog')
                                                                           from the "log request" function.
            def view the log() -> 'html':
                with UseDatabase(app.config['dbconfig']) as cursor:
                      SQL = """select phrase, letters, ip, browser_string, results
                                  from log"""
                                                      Send the query to the server, then fetch the results. Note the assignment of the fetched data to "contents".
                      cursor.execute(SQL)
                                            'Letters'
                                                         , 'Remote addr', 'User agent', 'Results')
Add in the
correct
                 return render template ('viewlog.html',
column names.
                                              the title='View Log',
                                              the row titles=titles,
                                              the data=contents,)
```

### IT'S NEARLY TIME FOR ONE LAST TEST DRIVE

Before taking this new version of your webapp for a spin, take a moment to confirm that your view\_the\_log function is the same as ours:

### **TEST DRIVE**



It's time to take your database-ready webapp for a spin.

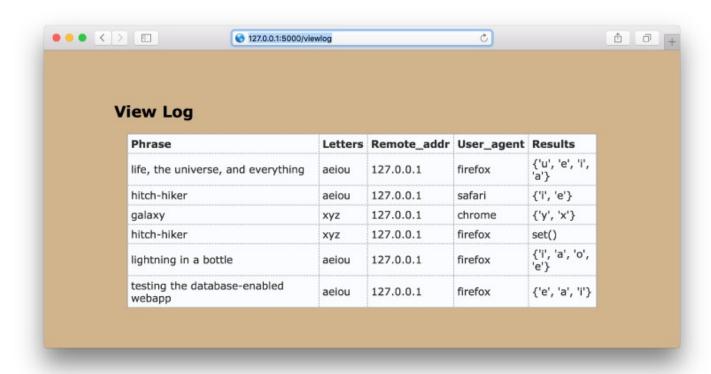
Be sure the DBcm.py file is in the same folder as your vsearch4web.py file, then start your webapp in the usual way on your operating system:

• Use python3 vsearch4web.py on Linux/Mac OS X

• Use py -3 vsearch4web.py on Windows.

Use your browser to go to your webapp's home page (running at <a href="http://127.0.0.1:5000">http://127.0.0.1:5000</a>), then enter a handful of searches. Once you've confirmed that the search feature is working, use the /viewlog URL to view the contents of your log in your browser window.

Although the searches you enter will very likely differ from ours, here's what we saw in our browser window, which confirms that everything is working as expected:



This browser output confirms the logged data is being read from the MySQL database when the /viewlog URL is accessed. This means the code in view\_the\_log is working—which, incidentally, confirms the log\_request function is working as expected, too, as it's putting the log data in the database as a result of every successful search.

Only if you feel the need, take a few moments to log into your MySQL database using the MySQL console to confirm that the data is safely stored in your database server. (Or just trust us: based on what our webapp is displaying above, it is.)

# All That Remains...

It's now time to return to the questions first posed in <a>Chapter 7</a>:

- 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?

Although it *is* possible to write Python code to answer these questions, we aren't going to in this case, even though we've just spent this and the previous two chapters looking at how Python and databases work together. In our opinion, creating Python code to answer these types of questions is nearly always a bad move...



SQL is definitely the way to go.

These types of "data questions" are best answered by your database technology's querying mechanism (which, in MySQL, is SQL). As you'll see on the next page, you'd be unlikely to produce Python code as quickly as writing the SQL queries you need.

Knowing when to use Python and when *not* to is important, as is knowing what sets Python apart from many other programming technologies. While most mainstream languages support classes and objects, few provide anything close to Python's context management protocol. (In the next chapter, you'll meet another feature that sets Python apart from many other languages: function decorators.)

Before we get to the next chapter, let's take a quick (one page) look at those SQL queries...

# **Answering the Data Questions**

Let's take the questions first posed in <u>Chapter 7</u> one by one, answering each with the help of some database queries written in SQL.

### HOW MANY REQUESTS HAVE BEEN RESPONDED TO?

If you're already a SQL dude (or dudette), you may be scoffing at this question, seeing as it doesn't really get much simpler. You already know that this most basic of SQL queries displays all the data in a database table:

```
select * from log;
```

To transform this query into one that reports how many rows of data a table has, pass the \* into the SQL function count, as follows:



### WHAT'S THE MOST COMMON LIST OF LETTERS?

The SQL query that answers this question looks a little scary, but isn't really. Here it is:

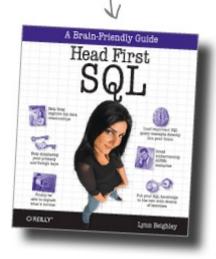
```
select count(letters) as 'count', letters
from log
group by letters
order by count desc
limit 1;
```

# Which IP addresses are the requests coming from?

The SQL dudes/dudettes out there are probably thinking "that's almost too easy":

```
select distinct ip from log;
```

As suggested in Chapter 7, we always recommend this book when someone's first learning SQL (as well as updating previous knowledge that might be a bit rusty).



# Which browser is being used the most?

The SQL query that answers this question is a slight variation on the query that answered the second question:

```
select browser_string, count(browser_string) as 'count'
from log
group by browser_string
order by count desc
limit 1;
```

So there you have it: all your pressing questions answered with a few simple SQL queries. Go ahead and try them at your mysql> prompt before starting in on the next chapter.

# Chapter 9's Code, 1 of 2

This is the context manager code in "DBcm.py".

```
import mysql.connector

class UseDatabase:

    def __init__(self, config: dict) -> None:
        self.configuration = config

def __enter__(self) -> 'cursor':
        self.conn = mysql.connector.connect(**self.configuration)
        self.cursor = self.conn.cursor()
        return self.cursor

def __exit__(self, exc_type, exc_value, exc_trace) -> None:
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
```

This is the first half of the webapp code in "vsearch 4 web.py

```
from flask import Flask, render template, request, escape
from vsearch import search4letters
from DBcm import UseDatabase
app = Flask( name )
app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearch',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }
def log_request(req: 'flask_request', res: str) -> None:
   with UseDatabase(app.config['dbconfig']) as 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, ))
```

# Chapter 9's Code, 2 of 2

```
This is the second half of the webapp code in "vsearch4web.py".
```

```
@app.route('/search4', methods=['POST'])
def do search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log request (request, results)
    return render template ('results.html',
                           the title=title,
                           the phrase=phrase,
                           the letters=letters,
                           the results=results,)
@app.route('/')
@app.route('/entry')
def entry page() -> 'html':
    return render template ('entry.html',
                           the title='Welcome to search4letters on the web!')
@app.route('/viewlog')
def view_the_log() -> 'html':
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""
        cursor.execute(SQL)
        contents = cursor.fetchall()
    titles = ('Phrase', 'Letters', 'Remote addr', 'User agent', 'Results')
    return render template ('viewlog.html',
                           the title='View Log',
                           the row titles=titles,
                           the data=contents,)
if __name__ == '__main ':
    app.run(debug=True)
```