Head First Python, 2nd Edition

PREV
10. Function Decorators: Wrapping Functions

NEXT
. 11¾ A Little Bit of Threading: Dealing with Waiting

Chapter 11. Exception Handling: What to Do When Things Go Wrong

I've tested this rope to destruction...what can possibly go wrong?

Things go wrong, all the time—no matter how good your code is.

You've successfully executed all of the examples in this book, and you're likely confident all of the code presented thus far works. But does this mean the code is robust? Probably not. Writing code based on the assumption that nothing bad ever happens is (at best) naive. At worst, it's dangerous, as unforeseen things do (and will) happen. It's much better if you're wary while coding, as opposed to trusting. Care is needed to ensure your code does what you want it to, as well as reacts properly when things go south. In this chapter, you'll not only see what can go wrong, but also learn what to do when (and, oftentimes, before) things do.

LONG EXERCISE

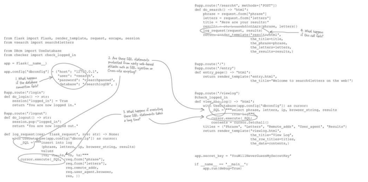
We're starting this chapter by diving right in. Presented below is the latest code to the `vsearch4web.py` webapp. As you'll see, we've updated this code to use the `check_logged_in` decorator from the last chapter to control when the information presented by the */viewlog* URL is (and isn't) visible to users.

Take as long as you need to read this code, then use a pencil to circle and annotate the parts you think might cause problems when operating within a production environment. Highlight *everything* that you think might cause an issue, not just potential runtime issues or errors.

```
from flask import Flask, render_template, request, escape, sess
from vsearch import search4letters

from DBcm import UseDatabase
from checker import check_logged_in

app = Flask(__name__)

app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearch',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }
@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'

@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'

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, ))

@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')
@check_logged_in
def view_the_log() -> 'html':
  with UseDatabase(app.config['dbconfig']) as cursor:
    _SQL = """select phrase, letters, ip, browser_string, resul
        from log"""
    cursor.execute(_SQL)
    contents = cursor.fetchall()
  titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', '
  return render_template('viewlog.html',
            the_title='View Log',
            the_row_titles=titles,
            the_data=contents,)

app.secret_key = 'YouWillNeverGuessMySecretKey'

if __name__ == '__main__':
  app.run(debug=True)
```

LONG EXERCISE SOLUTION

You were to take as long as you needed to read the code shown below (which is an updated version of the `vsearch4web.py` webapp). Then, using a pencil, you were to circle and annotate the parts you thought might cause problems when operating within a production environment. You were to highlight everything you thought might cause an issue, not just potential runtime issues or errors. (We've numbered our annotations for ease of reference.)



```
from flask import Flask, render_template, request, escape, session
from vsearch import search4letters

from DBcm import UseDatabase
from checker import check_logged_in

app = Flask(__name__)

app.config['dbconfig'] = {'host': '127.0.0.1',
                          'user': 'vsearch',
                          'password': 'vsearchpasswd',
                          'database': 'vsearchlogDB', }

@app.route('/login')
def do_login() -> str:
    session['logged_in'] = True
    return 'You are now logged in.'

@app.route('/logout')
def do_logout() -> str:
    session.pop('logged_in')
    return 'You are now logged out.'

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, ))

@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')
@check_logged_in
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,)

app.secret_key = 'YouWillNeverGuessMySecretKey'

if __name__ == '__main__':
    app.run(debug=True)
```

1. What happens if the database connection fails?

2. Are these SQL statements protected from nasty web-based attacks such as SQL injection or Cross-site scripting?

3. What happens if executing these SQL statements takes a long time?

4. What happens if this call fails?

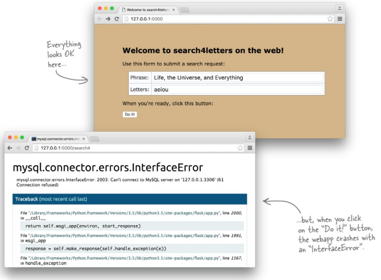
Databases Aren't Always Available

We've identified four potential issues with the `vsearch4web.py` code, and we concede that there may be many more, but we'll worry about these four issues for now. Let's consider each of the four issues in more detail (which we do here and on the next few pages, by simply describing the problems; *we'll work on solutions later in this chapter*). First up is worrying about the backend database:

1. **What happens if the database connection fails?**

   Our webapp blissfully assumes that the backend database is always operational and available, but it may not be (for any number of reasons). At the moment, it is unclear what happens when the database is down, as our code does not consider this eventuality.

Let's see what happens if we temporarily switch *off* the backend database. As you can see below, our webapp loads fine, but as soon as we do anything, an intimidating error message appears:



Welcome to search4letters on the web!

Use this form to submit a search request:

Phrase: Life, the Universe, and Everything

Letters: aeiou

When you're ready, click this button:

mysql.connector.errors.InterfaceError

mysql.connector.errors.InterfaceError: 2003: Can't connect to MySQL server on '127.0.0.1:3306' (61 Connection refused)

Traceback (most recent call last)

```
File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/flask/app.py", line 2000,
in __call__
  return self.wsgi_app(environ, start_response)
File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/flask/app.py", line 1991,
in wsgi_app
  response = self.make_response(self.handle_exception(e))
File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/flask/app.py", line 1567,
in handle_exception
```

Everything looks OK here...

...but, when you click on the "Do it!" button, the webapp crashes with an "InterfaceError".

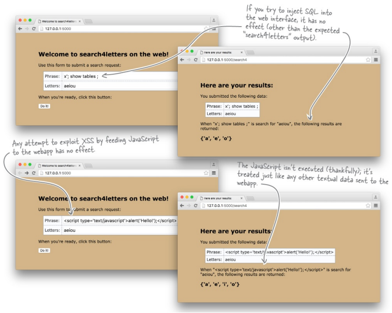
Web Attacks Are a Real Pain

As well as worrying about issues with your backend database, you also need to worry about nasty individuals trying to do nasty things to your webapp, which brings us to the second issue:

2. **Is our webapp protected from web attacks?**

   The phrases *SQL injection (SQLi)* and *Cross-site scripting (XSS)* should strike fear in the heart of every web developer. The former allows attackers to exploit your backend database, while the latter allows them to exploit your website. There are other web exploits that you'll need to worry about, but these are the "big two."

As with the first issue, let's see what happens when we try to simulate these exploits against our webapp. As you can see, it appears we're ready for both of them:



If you try to inject SQL into the web interface, it has no effect (other than the expected "search4letters" output).

Welcome to search4letters on the web!

Use this form to submit a search request:

Phrase: x'; show tables ;
Letters: aeiou

When you're ready, click this button:

Here are your results:

You submitted the following data:

Phrase: x'; show tables ;
Letters: aeiou

When "x'; show tables ;" is search for "aeiou", the following results are returned:

{'a', 'e', 'o'}

Any attempt to exploit XSS by feeding JavaScript to the webapp has no effect.

Welcome to search4letters on the web!

Use this form to submit a search request:

Phrase: <script type='text/javascript'>alert('Hello!');</script>
Letters: aeiou

When you're ready, click this button:

The JavaScript isn't executed (thankfully); it's treated just like any other textual data sent to the webapp.

Here are your results:

You submitted the following data:

Phrase: <script type='text/javascript'>alert('Hello!');</script>
Letters: aeiou

When "<script type='text/javascript'>alert('Hello!');</script>" is search for "aeiou", the following results are returned:

{'a', 'e', 'i', 'o'}

Input-Output Is (Sometimes) Slow

At the moment, our webapp communicates with our backend database in an almost instantaneous manner, and users of our webapp notice little or no delay as the webapp interacts with the database. But imagine if the interactions with the backend database took some time, perhaps seconds:

3. **What happens if something takes a long time?**

   Perhaps the backend database is on another machine, in another building, on another continent...what would happen then?

Communications with the backend database may take time. In fact, whenever your code has to interact with something that's external to it (for example: a file, a database, a network, or whatever), the interaction can take any amount of time, the determination of which is usually beyond your control. Despite this lack of control, you do have to be cognizant that some operations may be lengthy.

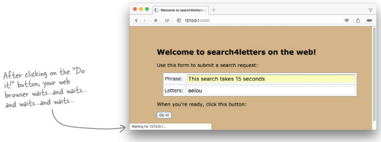
To demonstrate this issue, let's add an *artificial* delay to our webapp (using the `sleep` function, which is part of the standard library's `time` module). Add this line of code to the top of your webapp (near the other `import` statements):

```
from time import sleep
```

With the above `import` statement inserted, edit the `log_request` function and insert this line of code before the `with` statement:

```
sleep(15)
```

If you restart your webapp, then initiate a search, there's a very distinct delay while your web browser waits for your webapp to catch up. As web delays go, 15 seconds will feel like a lifetime, which will prompt most users of your webapp to believe something has *crashed*:



After clicking on the "Do it!" button, your web browser waits...and waits...and waits...and waits...

Welcome to search4letters on the web!

Use this form to submit a search request:

Phrase: This search takes 15 seconds
Letters: aeiou

When you're ready, click this button:

Waiting for 127.0.0.1...

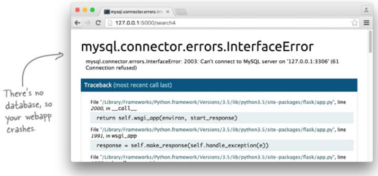
Your Function Calls Can Fail

The final issue identified during this chapter's opening exercise relates to the function call to `log_request` within the `do_search` function:

4. **What happens if a function call fails?**

   There's never a guarantee that a function call will succeed, especially if the function in question interacts with something external to your code.

We've already seen what can happen when the backend database is unavailable—the webapp crashes with an `InterfaceError`:



There's no database, so your webapp crashes.

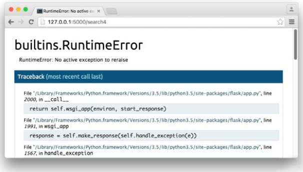
mysql.connector.errors.InterfaceError

mysql.connector.errors.InterfaceError: 2003: Can't connect to MySQL server on '127.0.0.1:3306' (61 Connection refused)

```
Traceback (most recent call last)

File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/flask/app.py", line 2000, in __call__
  return self.wsgi_app(environ, start_response)
File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/flask/app.py", line 1991, in wsgi_app
  response = self.make_response(self.handle_exception(e))
File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/flask/app.py", line
```

Other problems can surface, too. To simulate another error, find the **`sleep(15)`** line you added from the Issue 3 discussion, and replace it with a single statement:

`raise`. When executed by the interpreter, `raise` forces a runtime error. If you try your webapp again, a *different* error occurs this time:



builtins.RuntimeError

RuntimeError: No active exception to reraise

Traceback (most recent call last)

```
File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/flask/app.py", line 2000, in __call__
  return self.wsgi_app(environ, start_response)
File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/flask/app.py", line 1991, in wsgi_app
  response = self.make_response(self.handle_exception(e))
File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/flask/app.py", line 1567, in handle_exception
```

Something else went wrong, and your webapp crashes again.

NOTE

Before flipping the page, remove that call to "raise" from your code to ensure the webapp starts working again.

Considering the Identified Problems

We've identified four issues with the `vsearch4web.py` code. Let's revisit each and consider our next steps.

1. YOUR DATABASE CONNECTION FAILS

Errors occur whenever an external system your code relies on is unavailable. The interpreter reported an `InterfaceError` when this happened. It's possible to spot, then react to, these types of errors using Python's built-in exception-handling mechanism. If you can spot when an error occurs, you're then in a position to do something about it.

GEEK BITS

If you want to know more about *SQLi* and *XSS*, Wikipedia is a great place to start. See *https://en.wikipedia.org/wiki/SQL_injection* and *https://en.wikipedia.org/wiki/Cross-site_scripting*, respectively. And remember, there are all kinds of other types of attack that can cause problems for your app; these are just the two biggies.

2. YOUR APPLICATION IS SUBJECTED TO AN ATTACK

Although a case can be made that worrying about attacks on your application is only of concern to web developers, developing practices that improve the robustness of the code you write are always worth considering. With `vsearch4web.py`, dealing with the "big two" web attack vectors, *SQL injection (SQLi)* and *Cross-site scripting (XSS)*, appears to be well in hand. This is more of a happy accident than by design on your part, as the Jinja2 library is built to guard against *XSS* by default, escaping any potentially problematic strings (recall that the JavaScript we tried to trick our webapp into executing had no effect). As regards *SQLi*, our use of DB-API's parameterized SQL strings (with all those `%s` placeholders) ensures—again, thanks to the way these modules were designed—that your code is protected from this entire class of attack.

3. YOUR CODE TAKES A LONG TIME TO EXECUTE

If your code takes a long time to execute, you have to consider the impact on your user's experience. If your user doesn't notice, then you're likely OK. However, if your user has to wait, you may have to do something about it (otherwise, your user may decide the wait isn't worth it, and go elsewhere).

4. YOUR FUNCTION CALL FAILS

It's not just external systems that generate exceptions in the interpreter—your code can raise exceptions, too. When this happens, you need to be ready to spot the exception, then recover as needed. The mechanism you use to enable this behavior is the same one hinted at in the discussion of issue 1, above.

So...where do we *start* when dealing with these four issues? It's possible to use the same mechanism to deal with issues 1 and 4, so that's where we'll begin.

Always Try to Execute Error-Prone Code

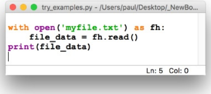
When something goes wrong with your code, Python raises a runtime **exception**. Think of an exception as a controlled program crash triggered by the interpreter.

As you've seen with issues 1 and 4, exceptions can be raised under many different circumstances. In fact, the interpreter comes with a whole host of built-in exception types, of which `RuntimeError` (from issue 4) is only one example. As well as the built-in exception types, it's possible to define your own custom exceptions, and you've seen an example of this too: the `InterfaceError` exception (from issue 1) is defined by the *MySQL Connector* module.

To spot (and, hopefully, recover from) a runtime exception, deploy Python's `try` statement, which can help you manage exceptions as they occur at runtime.

To see `try` in action, let's first consider a snippet of code that might fail when executed. Here are three innocent-looking, but potentially problematic, lines of code for you to consider:

For a complete list of the built-in exceptions, see https://docs.python.org/3/library/exceptions.html.

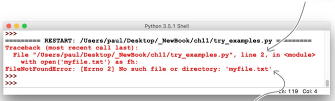


```
with open('myfile.txt') as fh:
    file_data = fh.read()
print(file_data)
```

There's nothing weird or wonderful going on here: the named file is opened, and its data is obtained and then displayed on screen.

There's nothing wrong with these three lines of code and—as currently written—they will execute. However, this code might fail if it can't access `myfile.txt`. Perhaps the file is missing, or your code doesn't have the necessary file-reading permissions. When the code fails, an exception is raised:

When a runtime error occurs, Python displays a "traceback", which details what went wrong, and where. In this case, the interpreter thinks the problem is on line 2.



```
>>>
========= RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples.py =========
Traceback (most recent call last):
  File "/Users/paul/Desktop/_NewBook/ch11/try_examples.py", line 2, in <module>
    with open('myfile.txt') as fh:
FileNotFoundError: [Errno 2] No such file or directory: 'myfile.txt'
>>>
>>>
```

Whoops!

Despite being ugly to look at, the traceback message is useful.

Let's start learning what `try` can do by adjusting the above code snippet to protect against this `FileNotFoundError` exception.

Catching an Error Is Not Enough

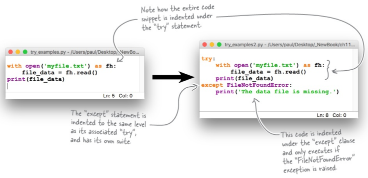
When a runtime error occurs, an exception is **raised**. If you *ignore* a raised exception it is referred to as **uncaught**, and the interpreter will terminate your code, then display a runtime error message (as shown in the example from the bottom of the last page). That said, raised exceptions can also be **caught** (i.e., dealt with) with the `try` statement. Note that it's not enough to catch runtime errors, you *also* have to decide what you're going to do next.

Perhaps you'll decide to deliberately ignore the raised exception, and keep going...with your fingers firmly crossed. Or maybe you'll try to run some other code in place of the code that crashed, and keep going. Or perhaps the best thing to do is to log the error before terminating your application as cleanly as possible. Whatever you decide to do, the `try` statement can help.

In its most basic form, the `try` statement allows you to react whenever the execution of your code results in a raised exception. To protect code with `try`, put the code within `try`'s suite. If an exception occurs, the code in the `try`'s suite terminates, and then the code in the `try`'s `except` suite runs. The `except` suite is where you define what you want to happen next.

Let's update the code snippet from the last page to display a short message whenever the `FileNotFoundError` exception is raised. The code on the left is what you had previously, while the code on the right has been amended to take advantage of what `try` and `except` have to offer:

When a runtime error is raised, it can be caught or uncaught: "try" lets you catch a raised error, and "except" lets you do something about it.


try_examples.py - /Users/paul/Desktop/_NewBo...
Ln: 5 Col: 0
try_examples2.py - /Users/paul/Desktop/_NewBook/ch11...
Ln: 8 Col: 0

```
with open('myfile.txt') as fh:
    file_data = fh.read()
print(file_data)
```

```
try:
    with open('myfile.txt') as fh:
        file_data = fh.read()
    print(file_data)
except FileNotFoundError:
    print('The data file is missing.')
```

Note how the entire code snippet is indented under the "try" statement.

The "except" statement is indented to the same level as its associated "try", and has its own suite.

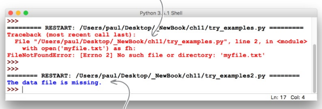
This code is indented under the "except" clause and only executes if the "FileNotFoundError" exception is raised.

Note that what was three lines of code is now six, which may seem wasteful, but isn't. The original snippet of code still exists as an entity; it makes up the suite associated with the `try` statement. The `except` statement and its suite is new code. Let's see what difference these amendments make.

TEST DRIVE

Let's take the `try`...`except` version of your code snippet for a spin. If `myfile.txt` exists and is readable by your code, its contents will appear on screen. If not, a run-time exception is raised. We already know that `myfile.txt` does not exist, but now, instead of seeing the ugly traceback message from earlier, the exception-handling code fires and we're presented with a friendlier message (even though our code snippet *still* crashed):

The first time you ran the code snippet, the interpreter generated this ugly traceback.



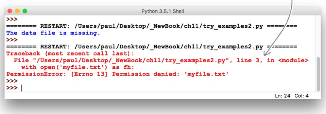
```
>>>
========= RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples.py =========
Traceback (most recent call last):
  File "/Users/paul/Desktop/_NewBook/ch11/try_examples.py", line 2, in <module>
    with open('myfile.txt') as fh:
FileNotFoundError: [Errno 2] No such file or directory: 'myfile.txt'
>>>
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples2.py ========
The data file is missing.
>>> |
```

The new version of the code produces a much friendlier messages thanks to "try" and "except".

THERE CAN BE MORE THAN ONE EXCEPTION RAISED...

This new behavior is better, but what happens if `myfile.txt` exists but your code does not have permission to read from it? To see what happens, we created the file, then set its permissions to simulate this eventuality. Rerunning the new code produces this output:

Yikes! We're back to seeing an ugly traceback message, as a "PermissionError" was raised.

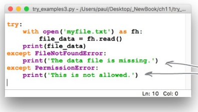


```
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples2.py ========
The data file is missing.
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples2.py ========
Traceback (most recent call last):
  File "/Users/paul/Desktop/_NewBook/ch11/try_examples2.py", line 3, in <module>
    with open('myfile.txt') as fh:
PermissionError: [Errno 13] Permission denied: 'myfile.txt'
>>>
>>> |
```

try Once, but except Many Times

To protect against another exception being raised, simply add another `except` suite to your `try` statement, identifying the exception you're interested in and providing whatever code you deem necessary in the new `except`'s suite. Here's another updated version of the code that handles the `PermissionError` exception (should it be raised):

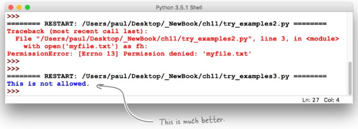
In addition to "FileNotFoundError" exceptions, this code also handles a "PermissionError".


try_examples3.py - /Users/paul/Desktop/_NewBook/ch11/try_...
Ln: 10 Col: 0

```
try:
    with open('myfile.txt') as fh:
        file_data = fh.read()
    print(file_data)
except FileNotFoundError:
    print('The data file is missing.')
except PermissionError:
    print('This is not allowed.')
```

The code in the "except" suites can do anything. For now, each displays a friendly message.

Executing this amended code still results in the `PermissionError` exception being raised. However, unlike before, the ugly traceback has been replaced by a much friendlier message:



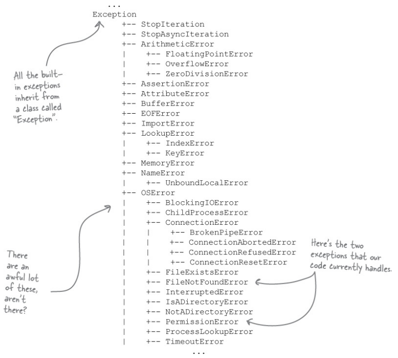
```
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples2.py ========
Traceback (most recent call last):
  File "/Users/paul/Desktop/_NewBook/ch11/try_examples2.py", line 3, in <module>
    with open('myfile.txt') as fh:
PermissionError: [Errno 13] Permission denied: 'myfile.txt'
>>>
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples3.py ========
This is not allowed.
>>>
```

This is much better.

This is looking good: you've managed to adjust what happens whenever the file you're hoping to work with isn't there (it doesn't exist), or is inaccessible (you don't have the correct permissions). But what happens if an exception is raised that you weren't expecting?

A Lot of Things Can Go Wrong

Before answering the question posed at the bottom of the last page—*what happens if an exception is raised that you weren't expecting?*—take a look at some of Python 3's built-in exceptions (which are copied directly from the Python documentation). Don't be surprised if you're struck by just how many there are:



```
          ...
     Exception
          +-- StopIteration
          +-- StopAsyncIteration
          +-- ArithmeticError
          |    +-- FloatingPointError
          |    +-- OverflowError
          |    +-- ZeroDivisionError
          +-- AssertionError
          +-- AttributeError
          +-- BufferError
          +-- EOFError
          +-- ImportError
          +-- LookupError
          |    +-- IndexError
          |    +-- KeyError
          +-- MemoryError
          +-- NameError
          |    +-- UnboundLocalError
          +-- OSError
          |    +-- BlockingIOError
          |    +-- ChildProcessError
          |    +-- ConnectionError
          |    |    +-- BrokenPipeError
          |    |    +-- ConnectionAbortedError
          |    |    +-- ConnectionRefusedError
          |    |    +-- ConnectionResetError
          |    +-- FileExistsError
          |    +-- FileNotFoundError
          |    +-- InterruptedError
          |    +-- IsADirectoryError
          |    +-- NotADirectoryError
          |    +-- PermissionError
          |    +-- ProcessLookupError
          |    +-- TimeoutError
                       ...
```

All the built-in exceptions inherit from a class called "Exception".

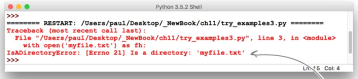
There are an awful lot of these, aren't there?

Here's the two exceptions that our code currently handles.

It would be crazy to try to write a separate `except` suite for each of these runtime exceptions, as some of them may never occur. That said, some *might* occur, so you do need to worry about them a little bit. Rather than try to handle each exception *individually*, Python lets you define a **catch-all** `except` suite, which fires whenever a runtime exception occurs that you haven't specifically identified.

THE CATCH-ALL EXCEPTION HANDLER

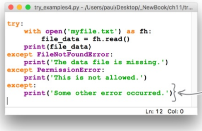
Let's see what happens when some other error occurs. To simulate just such an occurrence, we've changed `myfile.txt` from a file into a folder. Let's see what happens when we run the code now:



```
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples3.py ========
Traceback (most recent call last):
  File "/Users/paul/Desktop/_NewBook/ch11/try_examples3.py", line 3, in <module>
    with open('myfile.txt') as fh:
IsADirectoryError: [Errno 21] Is a directory: 'myfile.txt'
>>>
```

Another exception has occurred.

Another exception is raised. You could create an extra `except` suite that fires when this `IsADirectoryError` exception occurs, but let's specify a catch-all runtime error instead, which fires whenever *any* exception (other than the two we've already specified) occurs. To do this, add a catch-all `except` statement to the end of the existing code:

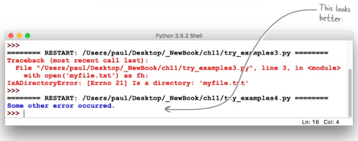


```
try:
    with open('myfile.txt') as fh:
        file_data = fh.read()
    print(file_data)
except FileNotFoundError:
    print('The data file is missing.')
except PermissionError:
    print('This is not allowed.')
except:
    print('Some other error occurred.')
```

This "except" statement is "bare": it does not refer to a specific exception.

This code provides a catch-all exception handler.

Running this amended version of your code gets rid of the ugly traceback, displaying a friendly message instead. No matter what other exception occurs, this code handles it thanks to the addition of the catch-all `except` statement:



```
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples3.py ========
Traceback (most recent call last):
  File "/Users/paul/Desktop/_NewBook/ch11/try_examples3.py", line 3, in <module>
    with open('myfile.txt') as fh:
IsADirectoryError: [Errno 21] Is a directory: 'myfile.txt'
>>>
======== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples4.py ========
Some other error occurred.
>>> |
```

This looks better.

Haven’t We Just Lost Something?

image with no caption

Ah, yes...good catch.

This latest code has tidied up the output (in that the ugly traceback is gone), but you’ve also lost some important information: you no longer know what the *specific* issue with your code was.

Knowing what exception was raised is often important, so Python lets you get at the data associated with the most-recent exception information *as it’s being handled*. There are two ways to do this: using the facilities of the `sys` module, and using an extension to the `try`/`except` syntax.

Let’s look at both of these techniques.

THERE ARE NO DUMB QUESTIONS

Q:

Q: *Is it possible to create a catch-all exception handler that does nothing?*

A:

A: *Yes. It is often tempting to add this* `except` *suite to the bottom of a* `try` *statement:*

```
except:
    pass
```

Please try not to do this. *This* `except` *suite implements a catch-all that ignores any other exception (presumedly in the misguided hope that if something is ignored it might go away). This is a dangerous practice, as—at the very least—an unexpected exception should result in an error message appearing on screen. So, be sure to always write error-checking code that handles exceptions, as opposed to ignores them.*

Learning About Exceptions from “sys”

The standard library comes with a module called `sys` that provides access to the interpreter’s *internals* (a set of variables and functions available at runtime).

One such function is `exc_info`, which provides information on the exception currently being handled. When invoked, `exc_info` returns a three-valued tuple where the first value indicates the exception’s **type**, the second details the exception’s **value**, and the third contains a **traceback object** that provides access to the traceback message (should you need it). When there is no currently available exception, `exc_info` returns the Python null value for each of the tuple values, which looks like this:`(None, None, None)`.

Knowing all of this, let’s experiment at the `>>>` shell. In the IDLE session that follows, we’ve written some code that’s always going to fail (as dividing by zero is *never* a good idea). A catch-all `except` suite uses the `sys.exc_info` function to extract and display data relating to the currently firing exception:

To learn more about “sys”, see https://docs.python.org/3/library/sys.html.

image with no caption

It’s possible to delve deeper into the traceback object to learn more about what just happened, but this already feels like too much work, doesn’t it? All we really want to know is what *type* of exception occurred.

To make this (and your life) easier, Python extends the `try`/`except` syntax to make it convenient to get at the information returned by the `sys.exc_info` function, and it does this without you having to remember to import the `sys` module, or wrangle with the tuple returned by that function.

Recall from a few pages back that the interpreter arranges exceptions in a hierarchy, with each exception inheriting from one called `Exception`. Let’s take advantage of this hierarchical arrangement as we rewrite our catch-all exception handler.

Recall the exception hierarchy from earlier.

```
   ...
Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
          ...
```

The Catch-All Exception Handler, Revisited

Consider your current code, which explicitly identifies the two exceptions you want to handle (`FileNotFoundError` and `PermissionError`), as well as provides a generic catch-all `except` suite (to handle everything else):

image with no caption

Note how, when referring to a *specific* exception, we’ve identified the exception by name after the `except` keyword. As well as identifying specific exceptions after `except`, it’s also possible to identify *classes* of exceptions using any of the names in the hierarchy.

For instance, if you’re only interested in knowing that an arithmetic error has occurred (as opposed to—specifically—a divide-by-zero error), you could specify `except ArithmeticError`, which would then catch a `FloatingPointError`, an `OverflowError`, and a `ZeroDivisionError` should they occur. Similarly, if you specify `except Exception`, you’ll catch *any* error.

image with no caption

But how does this help...surely you’re already catching all errors with a “bare” `except` statement? It’s true: you are. But you can extend the `except Exception` statement with the `as` keyword, which allows you to assign the current exception object to a variable (with `err` being a very popular name in this situation) and create more informative error message. Take a look at another version of the code, which uses `except Exception as`:

image with no caption

TEST DRIVE

With this—the last of the changes to your `try`/`except` code—applied, let’s confirm that everything is working as expected before returning to `vsearch4web.py` and applying what you now know about exceptions to your webapp.

Let’s start with confirming that the code displays the correct message when the file is missing:

image with no caption

If the file exists, but you don’t have permission to access it, a different exception is raised:

image with no caption

Any other exception is handled by the catch-all, which displays a friendly message:

image with no caption

Finally, if all is OK, the `try` suite runs without error, and the file’s contents appear on screen:

image with no caption

Getting Back to Our Webapp Code

Recall from the start of this chapter that we identified an issue with the call to `log_request` within `vsearch4web.py`’s `do_search` function. Specifically, we’re concerned about what to do when the call to `log_request` fails:

image with no caption

Based on our investigations, we learned that this call might fail if the backend database is unavailable, or if some other error occurs. When an error (of any type) occurs, the webapp responds with an unfriendly error page, which is likely to confuse (rather than enlighten) your webapp’s users:

image with no caption

Although it is important to us, the logging of each web request is not something that our webapp users really care about; all they want to see is the results of their search. Consequently, let’s adjust the webapp’s code so that it deals with errors within `log_request` by handling any raised exceptions *silently*.

Silently Handling Exceptions

image with no caption

No: “silently” does not mean “ignore.”

When we suggest handling exceptions *silently* in this context, we’re referring to handling any exceptions raised in such a way that your webapp users don’t notice. At the moment, your users *do* notice, as the webapp crashes with a confusing and—let’s be honest—*scary* error page.

Your webapp users don’t need to worry about `log_request` failing, but you do. So let’s adjust your code so that exceptions raised by `log_request` aren’t noticed by your users (i.e., they are silenced), but *are* noticed by you.

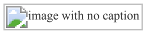
THERE ARE NO DUMB QUESTIONS

Q:	**Q: Doesn't all this `try`/`except` stuff just make my code harder to read and understand?**
A:	**A:** *It's true that the example code in this chapter started out as three easy-to-understand lines of Python code, and then we added seven lines of code, which—on the face of things—have nothing to do with what the first three lines of code are doing. However, it is important to protect code that can potentially raise an exception, and `try`/`except` is generally regarded as the best way to do this. Over time, your brain will learn to spot the important stuff (the code actually doing the work) that lives in the `try` suite, and filter out the `except` suites that are there to handle exceptions. When trying to understand code that uses `try`/`except`, always read the `try` suite first to learn what the code does, then look at the `except` suites if you need to understand what happens when things go wrong.*

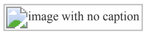
SHARPEN YOUR PENCIL

Let's add some `try/except` code to `do_search`'s invocation of the `log_request` function. To keep things straightforward, let's add a catch-all exception handler around the call to `log_request`, which, when it fires, displays a helpful message on standard output (using a call to the `print` BIF). In defining a catch-all exception handler, you can suppress your webapp's standard exception-handling behavior, which currently displays the unfriendly error page.

Here's `log_request`'s code as it's currently written:



In the spaces below, provide the code that implements a catch-all exception handler around the call to `log_request`:



SHARPEN YOUR PENCIL SOLUTION

The plan was to add some `try/except` code to `do_search`'s invocation of the `log_request` function. To keep things straightforward, we decided to add a catch-all exception handler around the call to `log_request`, which, when it fires, displays a helpful message on standard output (using a call to the `print` BIF).

Here's `log_request`'s code as currently written:

```
@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,)
```

In the spaces below, you were to provide the code that implements a catch-all exception handler around the call to `log_request`:

image with no caption

(EXTENDED) TEST DRIVE, 1 OF 3

With the catch-all exception-handling code added to `vsearch4web.py`, let's take your webapp for an extended spin (over the next few pages) to see the difference this new code makes. Previously, when something went wrong, your user was greeted with an unfriendly error page. Now, however, the error is handled "silently" by the catch-all code. If you haven't done so already, run `vsearch4web.py`, then use any browser to surf to your webapp's home page:

image with no caption

image with no caption

On the terminal that's running your code, you should see something like this:

image with no caption

(EXTENDED) TEST DRIVE, 2 OF 3

In order to simulate an error, we've switched off our backend database, which should result in an error occurring whenever the webapp tries to interact with the database. As our code silently catches all errors generated by `log_request`, the webapp user isn't aware that the logging hasn't occurred. The catch- all code has arranged to generate a message on screen describing the problem. Sure enough, when you enter a phrase and click on the "Do it!" button, the webapp displays the results of your search in the browser, whereas the webapp's terminal screen displays the "silenced" error message. Note that, despite the runtime error, the webapp continues to execute and successfully services the call to */search4*:

image with no caption

image with no caption

(EXTENDED) TEST DRIVE, 3 OF 3

In fact, no matter what error occurs when `log_request` runs, the catch-all code handles it.

We restarted our backend database, then tried to connect with an incorrect username. You can raise this error by changing the `dbconfig` dictionary in `vsearch4web.py` to use `vsearchwrong` as the value for `user`:

image with no caption

When your webapp reloads and you perform a search, you'll see a message like this in your terminal:

Change the value for user back to `vsearch`, and then let's try to access a nonexistent table, by changing the name of the table in the SQL query used in the `log_request` function to be `logwrong` (instead of `log`): Change the name of the table back to `log` and then, as a final example, let's add a `raise` statement to the `log_request` function (just before the `with` statement), which generates a custom exception:

image with no caption

When your webapp reloads one last time, and you perform one last search, you'll see the following message in your terminal:

Change the name of the table back to `log` and then, as a final example, let's add a `raise` statement to the `log_request` function (just before the `with` statement), which generates a custom exception:

image with no caption

When your webapp reloads one last time, and you perform one last search, you'll see the following message in your terminal:

Handling Other Database Errors

The `log_request` function makes use of the `UseDatabase` context manager (as provided by the `DBcm` module). Now that you've protected the call to `log_request`, you can rest easy, safe in the knowledge that any issues relating to problems with the database will be caught (and handled) by your catch-all exception-handling code.

However, the `log_request` function isn't the only place where your webapp interacts with the database. The `view_the_log` function grabs the logging data from the database prior to displaying it on screen.

Recall the code for the `view_the_log` function:

image with no caption

This code can fail, too, as it interacts with the backend database. However, unlike `log_request`, the `view_the_log` function is not called from the code in `vsearch4web.py`; it's invoked by Flask on your behalf. This means you can't write code to protect the invocation of `view_the_log`, as it's the Flask framework that calls the function, not you.

If you can't protect the invocation of `view_the_log`, the next best thing is to protect the code in its suite, specifically the use of the `UseDatabase` context manager. Before considering how to do this, let's consider what can go wrong:

- The backend database may be unavailable.
- You may not be able to log in to a working database.
- After a successful login, your database query might fail.
- Something else (unexpected) might happen.

This list of problems is similar to those you had to worry about with `log_request`.

Does "More Errors" Mean "More excepts"?

Knowing what we now know about `try`/`except`, we could add some code to the `view_the_log` function to protect the use of the `UseDatabase` context manager:

image with no caption

This catch-all strategy certainly works (after all, that's what you used with `log_request`). However, things can get complicated if you decide to do something other than implement a catch-all exception handler. What if you decide you need to react to a specific database error, such as "Database not found"? Recall from the beginning of this chapter that MySQL reports an `InterfaceError` exception when this happens:

image with no caption

You could add an `except` statement that targets the `InterfaceError` exception, but to do this your code also has to import the `mysql.connector` module, which defines this particular exception.

On the face of things, this doesn't seem like a big deal. But it is.

Avoid Tightly Coupled Code

Let's assume you've decided to create an `except` statement that protects against your backend database being unavailable. You could adjust the code in `view_the_log` to look something like this:

image with no caption

If you also remember to add `import mysql.connector` to the top of your code, this additional `except` statement works. When your backend database can't be found, this additional code allows your webapp to remind you to check that your database is switched on.

This new code works, and you can see what's going on here...what's not to like?

The issue with approaching the problem in this way is that the code in `vsearch4web.py` is now very **tightly coupled** to the *MySQL* database, and specifically the use of the *MySQL Connector* module. Prior to adding this second `except` statement, your `vsearch4web.py` code interacted with your backend database via the `DBcm` module (developed earlier in this book). Specifically, the `UseDatabase` context manager provides a convenient **abstraction** that decouples the code in `vsearch4web.py` from the backend database. If, at some time in the future, you needed to replace *MySQL* with *PostgreSQL*, the only changes you'd need to make would be to the `DBcm` module, *not* to all the code that uses `UseDatabase`. However, when you create code like that shown above, you tightly bind (i.e., couple) your webapp code to the MySQL backend database because of that `import mysql.connector` statement, in addition to your new `except` statement's reference to `mysql.connector.errors.InterfaceError`.

If you need to write code that tightly couples to your backend database, always consider putting that code in the `DBcm` module. This way, your webapp can be written to use the generic interface provided by `DBcm`, as opposed to a specific interface that targets (and locks you into) a specific backend database.

Let's now consider what moving the above `except` code into `DBcm` does for our webapp.

The DBcm Module, Revisited

You last looked at `DBcm` in Chapter 9, when you created that module in order to provide a hook into the `with` statement when working with a *MySQL* database. Back then, we sidestepped any discussion of error handling (by conveniently ignoring the issue). Now that you've seen what the `sys.exc_info` function does, you should have a better idea of what the arguments to `UseDatabase`'s `__exit__` method mean:

image with no caption

Recall that `UseDatabase` implements three methods:

- `__init__` provides a configuration opportunity *prior* to `with` executing,
- `__enter__` executes as the `with` statement *starts*, and
- `__exit__` is guaranteed to execute whenever the `with`'s suite *terminates*.

At least, that's the expected behavior whenever everything goes to plan. When things go wrong, this behavior **changes**.

For instance, if an exception is raised while `__enter__` is executing, the `with` statement terminates, and any subsequent processing of `__exit__` is *cancelled*.

This makes sense: if `__enter__` runs into trouble, `__exit__` can no longer assume that the execution context is initialized and configured correctly (so it's prudent not to run the `__exit__` method's code).

The big issue with the `__enter__` method's code is that the backend database may not be available, so let's take some time to adjust `__enter__` for this possibility, generating a custom exception when the database connection cannot be established. Once we've done this, we'll adjust `view_the_log` to check for our custom exception instead of the highly database-specific `mysql.connector.errors.InterfaceError`.

Creating Custom Exceptions

Creating your own custom exceptions couldn't be any easier: decide on an appropriate name, then define an empty class that inherits from Python's built-in `Exception` class. Once you've defined a custom exception, it can be raised with the `raise` keyword. And once an exception is raised, it's caught (and dealt with) by `try`/`except`.

A quick trip to IDLE's `>>>` prompt demonstrates custom exceptions in action. In this example, we're creating a custom exception called `ConnectionError`, which we then raise (with `raise`), before catching with `try`/`except`. Read the annotations in numbered order, and (if you're following along) enter the code we've typed at the `>>>` prompt:

image with no caption

The empty class isn't quite empty...

In describing the `ConnectionError` class as "empty," we told a little lie. Granted, the use of `pass` ensures that there's no *new* code associated with the `ConnectionError` class, but the fact that `ConnectionError` **inherits** from Python's built-in `Exception` class means that all of the attributes and behaviors of `Exception` are available in `ConnectionError` too (making it anything but empty). This explains why `ConnectionError` works just as you'd expect it to with `raise` and `try`/`except`.

SHARPEN YOUR PENCIL

Let's adjust the `DBcm` module to `raise` a custom `ConnectionError` whenever a connection to the backend database fails.

1. Here's the current code to `DBcm.py`. In the spaces provided, add in the code required to `raise` a `ConnectionError`.

   image with no caption

2. With the code in the `DBcm` module amended, use your pencil to detail any changes you'd make to this code from `vsearch4web.py` in order to take advantage of the newly defined `ConnectionError` exception:

   image with no caption

SHARPEN YOUR PENCIL SOLUTION

1. You were to adjust the `DBcm` module to `raise` a custom `ConnectionError` whenever a connection to the backend database fails. You were to adjust the current code to `DBcm.py` to add in the code required to `raise` a `ConnectionError`.

image with no caption

2. With the code in the `DBcm` module amended, you were to detail any changes you'd make to this code from `vsearch4web.py` in order to take advantage of the newly defined `ConnectionError` exception:

image with no caption

TEST DRIVE

Let's see what difference this new code makes. Recall that you've moved the MySQL-specific exception-handling code from `vsearch4web.py` into `DBcm.py` (and replaced it with code that looks for your custom `ConnectionError` exception). Has this made any difference?

Here are the messages that the previous version of `vsearch4web.py` generated whenever the backend database couldn't be found:

```
        ...
Is your database switched on? Error: 2003: Can't connect to MyS
(61 Connection refused)
127.0.0.1 - - [16/Jul/2016 21:21:51] "GET /viewlog HTTP/1.1" 20
```

And here are the messages that the most recent version of `vsearch4web.py` generates whenever the backend database can't be found:

```
        ...
Is your database switched on? Error: 2003: Can't connect to MyS
(61 Connection refused)
127.0.0.1 - - [16/Jul/2016 21:22:58] "GET /viewlog HTTP/1.1" 20
```

image with no caption

Yes. On the face of things, these are the same.

However, although the output from the current and previous versions of `vsearch4web.py` appears identical, behind the scenes things are *very different*.

If you decide to change the backend database from *MySQL* to *PostgreSQL*, you no longer have to worry about changing any of the code in `vsearch4web.py`, as all of your database-specific code resides in `DBcm.py`. As long as the changes you make to `DBcm.py` maintain the same *interface* as previous versions of the module, you can change SQL databases as often as you like. This may not seem like a big deal now, but if `vsearch4web.py` grows to hundreds, thousands, or tens of thousands of lines of code, its really is a big deal.

What Else Can Go Wrong with “DBcm”?

Even if your backend database is up and running, things can still go wrong.

For example, the credentials used to access the database may be incorrect. If they are, the `__enter__` method will fail again, this time with a `mysql.connector.errors.ProgrammingError`.

Or, the suite of code associated with your `UseDatabase` context manager may raise an exception, as there's never a guarantee that it executes correctly. A `mysql.connector.errors.ProgrammingError` is *also* raised whenever your database query (the SQL you're executing) contains an error.

The error message associated with an SQL query error is different than the message associated with the credentials error, but the exception raised is the same: `mysql.connector.errors.ProgrammingError`. Unlike with credentials errors, errors in your SQL results in an exception being raised while the `with` statement is executing. This means that you'll need to consider protecting against this exception in more than one place. The question is: where?

To answer this question, let's take another look at `DBcm`'s code:

image with no caption

You might be tempted to suggest that exceptions raised within the `with` suite should be handled with a `try`/`except` statement *within* the `with`, but such a strategy gets you right back to writing tightly coupled code. But consider this: when an exception is raised within `with`'s suite and *not* caught, the `with` statement arranges to pass details of the uncaught exception into your context manager's `__exit__` method, where you have the option of doing something about it.

Creating More Custom Exceptions

Let's extend `DBcm.py` to report two additional, custom exceptions.

The first is called `CredentialsError` and is raised when a `ProgrammingError` occurs within the `__enter__` method. The second is called `SQLError` and is raised when a `ProgrammingError` is reported to the `__exit__` method.

Defining these new exceptions is easy: add two new, empty exception classes to the top of `DBcm.py`:

image with no caption

A `CredentialsError` can occur during `__enter__`, so let's adjust that method's code to reflect this. Recall that an incorrect MySQL username or password results in a `ProgrammingError` being raised:

image with no caption

These code changes adjust `DBcm.py` to raise a `CredentialsError` exception when you provide either an incorrect username or password from your code to your backend database (MySQL). Adjusting `vsearch4web.py`'s code is your next task.

Are Your Database Credentials Correct?

With these latest changes made to `DBcm.py`, let's now adjust the code in `vsearch4web.py`, paying particular attention to the `view_the_log` function. However, before doing anything else, add `CredentialsError` to the list of imports from `DBcm` at the top of your `vsearch4web.py` code:

image with no caption

With the `import` line amended, you next need to add a new `except` suite to the `view_the_log` function. As when you added support for a `ConnectionError`, this is a straightforward edit:

image with no caption

There's really nothing new here, as all you're doing is repeating what you did for `ConnectionError`. Sure enough, if you try to connect to your backend database with an incorrect username (or password), your webapp now displays an appropriate message, like this:

image with no caption

Handling SQLError Is Different

Both `ConnectionError` and `CredentialsError` are raised due to problems with the `__enter__` method's code executing. When either exception is raised, the corresponding `with` statement's suite is **not** executed.

If all is well, your `with` suite executes as normal.

Recall this `with` statement from the `log_request` function, which uses the `UseDatabase` context manager (provided by `DBcm`) to insert data into the backend database:

image with no caption

If (for some reason) your SQL query contains an error, the *MySQL Connector* module generates a `ProgrammingError`, just like the one raised during your context manager's `__enter__` method. However, as this exception occurs *within* your context manager (i.e., within the `with` statement) and is *not* caught there, the exception is passed back to the `__exit__` method as three arguments: the *type* of the exception, the *value* of the exception, and the *traceback* associated with the exception.

If you take a quick look at `DBcm`'s existing code for `__exit__`, you'll see that the three arguments are ready and waiting to be used:

image with no caption

When an exception is raised within the `with` suite and not caught, the context manager terminates the `with` suite's code, and jumps to the `__exit__` method, which then executes. Knowing this, you can write code that checks for exceptions of interest to your application. However, if no exception is raised, the three arguments (`exc_type`, `exc_value`, and `exc_traceback`) are all set to `None`. Otherwise, they are populated with details of the raised exception.

"None" is Python's null value.

Let's exploit this behavior to raise an `SQLError` whenever something goes wrong within the `UseDatabase` context manager's `with` suite.

Be Careful with Code Positioning

To check whether an uncaught exception has occurred within your code's `with` statement, check the `exc_type` argument to the `__exit__` method within `__exit__`'s suite, being careful to consider exactly where you add your new code.

image with no caption

Yes, it does make a difference.

To understand why, consider that your context manager's `__exit__` method provides a place where you can put code that is **guaranteed** to execute *after* your `with` suite ends. That behavior is part of the context manager protocol, after all.

This behavior needs to hold even when exceptions are raised within your context manager's `with` suite. Which means that if you plan to add code to the `__exit__` method, it's best to put it *after* any existing code in `__exit__`, as that way you'll still guarantee the method's existing code executes (and preserve the semantics of the context manager protocol).

Let's take another look at the existing code in the `__exit__` method in light of this code placement discussion. Consider that any code we add needs to raise an `SQLError` exception if `exc_type` indicates a `ProgrammingError` has occurred:

image with no caption

Raising an SQLError

At this stage, you've already added the `SQLError` exception class to the top of the `DBcm.py` file:

image with no caption

With the `SQLError` exception class defined, all you need to do now is add some code to the `__exit__` method to check whether `exc_type` is the exception you're interested in, and if it is, `raise` an `SQLError`. This is so straightforward that we are resisting the usual *Head First* urge to turn creating the required code into an exercise, as no one wants to insult anyone's intelligence at this stage in this book. So, here's the code you need to append to the `__exit__` method:

image with no caption

If you want to be **extra safe**, and do something sensible with any other, unexpected exception sent to `__exit__`, you can add an `elif` suite to the end of the `__exit__` method that reraises the unexpected exception to the calling code:

image with no caption

TEST DRIVE

With support for the `SQLError` exception added to `DBcm.py`, add another `except` suite to your `view_the_log` function to catch any `SQLError`s that occur:

image with no caption

Once you save `vsearch4web.py`, your webapp should reload and be ready for testing. If you try to execute an SQL query that contains errors, the exception is handled by the above code:

image with no caption

Equally, if something unexpected happens, your webapp's catch-all code kicks into gear, displaying an appropriate message:

image with no caption

With exception-handling code added to your webapp, no matter what runtime error occurs, your webapp continues to function without displaying a scary or confusing error page to your users.

image with no caption

Yes, it does. And this is very powerful.

A Quick Recap: Adding Robustness

Let's take a minute to remind ourselves of what we set out to do in this chapter. In attempting to make our webapp code more robust, we had to answer four questions relating to four identified issues. Let's review each question and note how we did:

1. **What happens if the database connection fails?**

   You created a new exception called `ConnectionError` that is raised whenever your backend database can't be found. You then used `try`/`except` to handle a `ConnectionError` were it to occur.

2. **Is our webapp protected from web attacks?**

   It was a "happy accident," but your choice of *Flask* plus *Jinja2*, together with Python's DB-API specification, protects your webapp from the most notorious of web attacks. So, yes, your webapp is protected from *some* web attacks (but not all).

3. **What happens if something takes a long time?**

   We still haven't answered this question, other than to demonstrate what happens when your webapp takes 15 seconds to respond to a user request: your web user has to wait (or, more likely, your web user gets fed up waiting and leaves).

4. **What happens if a function call fails?**

   You used `try`/`except` to protect the function call, which allowed you to control what the user of your webapp sees when something goes wrong.

WHAT HAPPENS IF SOMETHING TAKES A LONG TIME?

When you did the initial exercise at the start of this chapter, this question resulted from our examination of the `cursor.execute` calls that occurred in the `log_request` and `view_the_log` functions. Although you've already worked with both of these functions in answering questions 1 and 4, above, you're not done with them quite yet.

Both `log_request` and `view_the_log` use the `UseDatabase` context manager to execute an SQL query. The `log_request` function **writes** the details of the submitted search to the backend database, whereas the `view_the_log` function **reads** from the database.

The question is: *what do you do if this write or read takes a long time?*

Well, as with a lot of things in the programming world, it depends.

How to Deal with Wait? It Depends...

How you decide to deal with code that makes your users wait—either on a read, or on a write—can get complex. So we're going to pause this discussion and defer a solution until the next, short chapter.

In fact, the next chapter is so short that it doesn't warrant its own chapter number (as you'll see), but the material it presents is complex enough to justify splitting it off from this chapter's main discussion, which presented Python's `try`/`except` mechanism. So, let's hang on for a bit before putting to rest issue 3: *what happens if something takes a long time?*

image with no caption

Yes. The irony is not lost on us.

We're asking you to *wait* to learn how to handle "waits" in your code.

But you've already learned a lot in this chapter, and we think it's important to take a bit of time to let the `try`/`except` material sink into your brain.

So, we'd like you to pause, and take a short break...after you've cast your eye over the code seen thus far in this chapter.

Chapter 11's Code, 1 of 3

image with no caption

image with no caption

Chapter 11's Code, 2 of 3

image with no caption

Chapter 11's Code, 3 of 3

image with no caption

PREV
10. Function Decorators: Wrapping Functions

NEXT
12. 11¾ A Little Bit of Threading: Dealing with Waiting