

The Final Step: Handling Arguments

We are nearly there—the “guts” of the decorator’s code is in place. What remains is to ensure the decorator handles the decorated function’s arguments properly, no matter what they might be. Recall item 4 from the recipe:

4

A decorator maintains the decorated function’s signature

Your decorator needs to ensure that the function it returns takes the same number and type of arguments as expected by the decorated function.

When a decorator is applied to an existing function, any calls to the existing function are **replaced** by calls to the function returned by the decorator. As you saw in the solution on the previous page, to comply with item 3 of our decorator-creation recipe, we return a wrapped version of the existing function, which implements extra code as needed. This wrapped version *decorates* the existing function.

But there’s a problem with this, as doing the wrapping on its own is not enough; the *calling characteristics* of the decorated function need to be maintained, too. This means, for instance, that if your existing function accepts two arguments, your wrapped function also has to accept two arguments. If you could know ahead of time how many arguments to expect, then you could plan accordingly. Unfortunately, you can’t know this ahead of time because your decorator can be applied to any existing function, which could have—quite literally—any number and type of arguments.

What to do? The solution is to go “generic,” and arrange for the wrapper function to support any number and type of arguments. You already know how to do this, as you’ve already seen what `*args` and `**kwargs` can do.

Remember: `*args` and `kwargs` support any number and type of arguments.**



Sharpen your pencil

What do you
need to add to
the “wrapper”
function’s
signature?

Let’s adjust the wrapper function to accept any number and type of arguments. Let’s also ensure that when `func` is invoked, it uses the same number and type of arguments as were passed to `wrapper`. Add in the argument code in the spaces provided below:

```
def check_logged_in(func):
    def wrapper( ..... ):
        if 'logged_in' in session:
            return func( ..... )
        return 'You are NOT logged in.'
    return wrapper
```



Sharpen your pencil Solution

Using a generic signature does the trick here, as it supports any number and type of arguments. Note how we invoke "func" with the same arguments supplied to "wrapper", no matter what they are.

You were to adjust the `wrapper` function to accept any number and type of arguments, as well as ensure that, when `func` is invoked, it uses the same number and type of arguments as were passed to `wrapper`:

```
def check_logged_in(func):  
    def wrapper( ..... ):  
        if 'logged_in' in session:  
            return func( ..... )  
        return 'You are NOT logged in.'  
    return wrapper
```

We're done...or are we?

If you check our decorator-creating recipe, you'd be forgiven for believing that we're done. We are...almost. There are two issues that we still need to deal with: one has to do with all decorators, whereas the other has to do with this specific one.

Let's get the specific issue out of the way first. As the `check_logged_in` decorator is in its own module, we need to ensure that any modules its code refers to are also imported into `checker.py`. The `check_logged_in` decorator uses `session`, which has to be imported from Flask to avoid errors. Handling this is straightforward, as all you need to do is add this `import` statement to the top of `checker.py`:

```
from flask import session
```

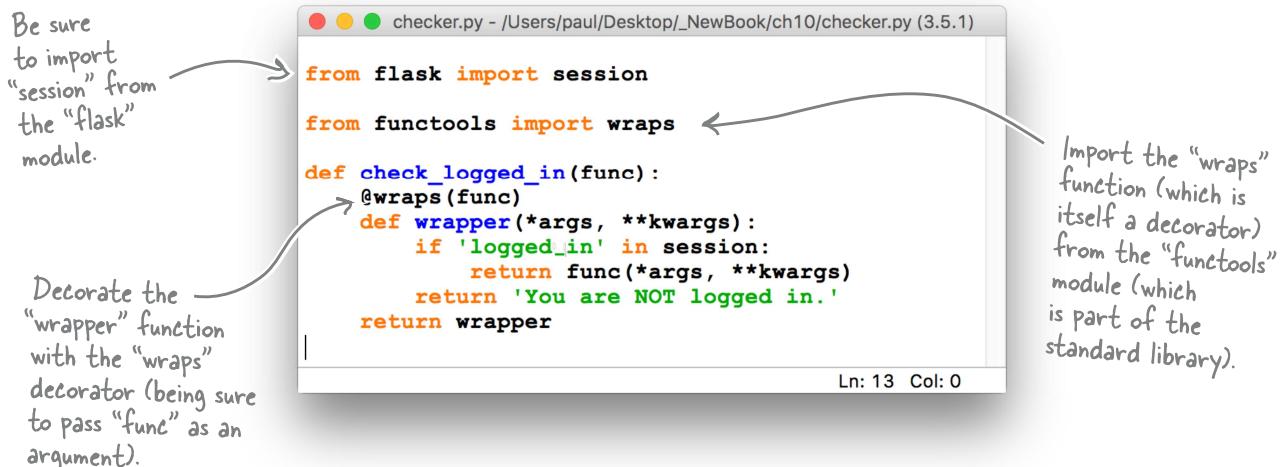
The other issue, which affects *all* decorators, has to do with how functions identify themselves to the interpreter. When decorated, and if due care is not taken, a function can forget its identity, which can lead to problems. The reason why this happens is very technical and a little exotic, and it requires a knowledge of Python's internals that most people don't need (or want) to know. Consequently, Python's standard library comes with a module that handles these details for you (so you need never worry about them). All you have to do is remember to import the required module (`functools`), then call a single function (`wraps`).

Perhaps somewhat ironically, the `wraps` function is implemented as a decorator, so you don't actually call it, but rather use it to decorate your `wrapper` function *inside* your own decorator. We've already gone ahead and done this for you, and you'll find the code to the completed `check_logged_in` decorator at the top of the next page.

**When creating
your own
decorators, always
import, then use,
the "functools"
module's "wraps"
function.**

Your Decorator in All Its Glory

Before continuing, make sure your decorator code *exactly* matches ours:



Now that the `checker.py` module contains a completed `check_logged_in` function, let's put it to use within `simple_webapp.py`. Here is the current version of the code to this webapp (which we're showing here over two columns):

```

from flask import Flask, session
app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello from the simple webapp.'

@app.route('/page1')
def page1() -> str:
    return 'This is page 1.'

@app.route('/page2')
def page2() -> str:
    return 'This is page 2.'

@app.route('/page3')
def page3() -> str:
    return 'This is page 3.'

@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.'

@app.route('/status')
def check_status() -> str:
    if 'logged_in' in session:
        return 'You are currently logged in.'
    return 'You are NOT logged in.'

app.secret_key = 'YouWillNeverGuess...'

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

```

Recall that our goal here is to restrict access to the `/page1`, `/page2`, and `/page3` URLs, which are currently accessible to any user's browser (based on this code).

Putting Your Decorator to Work

Adjusting the `simple_webapp.py` code to use the `check_logged_in` decorator is not difficult. Here's a list of what needs to happen:

1 Import the decorator

The `check_logged_in` decorator needs to be imported from the `checker.py` module. Adding the required `import` statement to the top of our webapp's code does the trick here.

2 Remove any unnecessary code

Now that the `check_logged_in` decorator exists, we no longer have any need for the `check_status` function, so it can be removed from `simple_webapp.py`.

3 Use the decorator as required

To use the `check_logged_in` decorator, apply it to any of our webapp's functions using the `@` syntax.

Here's the code to `simple_webapp.py` once more, with the three changes listed above applied. Note how the `/page1`, `/page2`, and `/page3` URLs now have two decorators associated with them: `@app.route` (which comes with Flask), and `@check_logged_in` (which you've just created):

Apply a decorator to an existing function using the `@` syntax.

```
from flask import Flask, session
from checker import check_logged_in

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello from the simple webapp.'

@app.route('/page1')
@check_logged_in
def page1() -> str:
    return 'This is page 1.'

@app.route('/page2')
@check_logged_in
def page2() -> str:
    return 'This is page 2.'
```

```
@app.route('/page3')
@check_logged_in
def page3() -> str:
    return 'This is page 3.'

@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.'

app.secret_key = 'YouWillNeverGuess...'

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

Don't forget to apply these highlighted edits to your webapp *before* continuing.



Test Drive

To convince ourselves that our login-checking decorator is working as required, let's take the decorator-enabled version of `simple_webapp.py` for a spin.

With the webapp running, use a browser to try to access `/page1` prior to logging in. After logging in, try to access `/page1` again and then, after logging out, try to access the restricted content once more. Let's see what happens:

1. When you first connect to the webapp, the home page appears.

2. Trying to access “/page1” results in refusal, as you are not logged in.

3. Accessing “/login” enables access to restricted URLs for your browser.

4. Now that your browser is logged in, you can see “/page1”—success!

5. You log out from the webapp.

6. As you are no longer logged in, you can’t see the restricted content on the “/page1” URL anymore.

gotta love decorators

The Beauty of Decorators

Take another look at the code for your `check_logged_in` decorator. Note how it abstracts the logic used to check if a user's browser is logged in, putting this (potentially complex) code in one place—*inside* the decorator—and then making it available throughout your code, thanks to the `@check_logged_in` decorator syntax:

A screenshot of a code editor window titled "checker.py - /Users/paul/Desktop/_NewBook/ch10/checker.py (3.5.1)". The code defines a `check_logged_in` decorator that wraps another function. The wrapped function checks if 'logged_in' is in the session; if so, it calls the original function with args and kwargs; if not, it returns a string 'You are NOT logged in.' A handwritten note on the right says: "This code looks freaky, but isn't really." An arrow points from this note to the wrapped function body.

```
from flask import session
from functools import wraps

def check_logged_in(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if 'logged_in' in session:
            return func(*args, **kwargs)
        return 'You are NOT logged in.'
    return wrapper
```

Ln: 13 Col: 0

Abstracting code in a decorator makes the code that uses it easier to read.

Consider this usage of our decorator on the `/page2` URL:

A screenshot of a code editor window titled "simple_webapp4.py - /Users/paul/Desktop/_NewBook/ch10/simple_webapp4.py (3.5.1)". It shows a `@app.route('/page2')` line followed by `@check_logged_in`. Below that is a `def page2():` definition. The code inside `page2` returns the string 'This is page 2.'. A handwritten note on the right says: "Using a decorator makes this code easier to read." An arrow points from this note to the `page2` function body.

```
@app.route('/page2')
@check_logged_in
def page2():
    return 'This is page 2.'
```

Ln: 1 Col: 0

Note how the `page2` function's code is only concerned with what it needs to do: display the `/page2` content. In this example, the `page2` code is a single, simple statement; it would be harder to read and understand if it *also* contained the logic required to check whether a user's browser is logged in or not. Using a decorator to separate out the login-checking code is a big win.

This “logic abstraction” is one of the reasons the use of decorators is popular in Python. Another is that, if you think about it, in creating the `check_logged_in` decorator, you’ve managed to write code that *augments an existing function with extra code, by changing the behavior of the existing function without changing its code*. When it was first introduced earlier in this chapter, this idea was described as “freaky.” But, now that you’ve done it, there’s really nothing to it, is there?

**Decorators
aren’t freaky;
they’re fun.**

Creating More Decorators

With the process of creating the `check_logged_in` decorator behind you, you can use its code as the basis of any new decorators you create from here on in.

To make your life easier, here's a generic code template (in the file `tmpl_decorator.py`) that you can use as the basis of any new decorators you write:

```
from functools import wraps

def decorator_name(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # 1. Code to execute BEFORE calling the decorated function.

        # 2. Call the decorated function as required, returning its
        #    results if needed.
        return func(*args, **kwargs)

        # 3. Code to execute INSTEAD of calling the decorated function.
    return wrapper
```

Ln: 15 Col: 0

This code template can be adjusted as needed to suit your needs. All you need to do is give your new decorator an appropriate name, then replace the three comments in the template with your decorator's specific code.

Replace these
comments with your
new decorator's
code.

If it makes sense for your new decorator to invoke the decorated function without returning its results, that's fine. After all, what you put in your `wrapper` function is your code, and you are free to do whatever you want to.

there are no
Dumb Questions

Q: Aren't decorators just like the last chapter's context manager in that they both let me wrap code with additional functionality?

A: That's a great question. The answer is: yes *and* no. Yes, both decorators and context managers augment existing code with additional logic. But no, they are not the same. Decorators are specifically concerned with augmenting existing functions with additional functionality, whereas context managers are more interested in ensuring your code executes within a specific context, arranging for code to run before a `with` statement as well as ensuring that code **always** executes after a `with` statement. You can do something similar with decorators, but most Python programmers would regard you as a little mad if you were to attempt this. Also, note that your decorator code is under no obligation to do anything after it invokes the decorated function (as is the case with the `check_logged_in` decorator, which does nothing). This decorator behavior is very different from the protocol that context managers are expected to adhere to.

[back to vsearch4web.py](#)

Back to Restricting Access to /viewlog



Now that you've created a mechanism that lets you restrict access to certain URLs in `simple_webapp.py`, it's a no-brainer to apply the same mechanism to any other webapp.

This includes `vsearch4web.py`, where you had a requirement to restrict access to the `/viewlog` URL. All you need to do is copy the `do_login` and `do_logout` functions from `simple_webapp.py` into `vsearch4web.py`, import the `checker.py` module, and then decorate the `view_the_log` function with `check_logged_in`. Granted, you may want to add some sophistication to `do_login` and `do_logout` (by, perhaps, checking user credentials against those stored in a database), but—as regards restricting access to certain URLs—the `check_logged_in` decorator does most of the heavy lifting for you.

What's Next?

Rather than spend a bunch of pages doing to `vsearch4web.py` what you've just spent a chunk of time doing to `simple_webapp.py`, we're going to leave adjusting `vsearch4web.py` for you to do *on your own*. At the start of the next chapter, we'll present an updated version of the `vsearch4web.py` webapp for you to compare with yours, as our updated code is used to frame the discussion in the next chapter.

To date, all of the code in this book has been written under the assumption that nothing bad ever happens, and nothing ever goes wrong. This was a deliberate strategy on our part, as we wanted to ensure you had a good grasp of Python before getting into topics such as error correction, error avoidance, error detection, exception handling, and the like.

We have now reached the point where we can no longer follow this strategy. The environments within which our code runs are real, and things can (and do) go wrong. Some things are fixable (or avoidable), and some aren't. If at all possible, you'll want your code to handle most error situations, only resulting in a crash when something truly exceptional happens that is beyond your control. In the next chapter, we look at various strategies for deciding what's a reasonable thing to do when stuff goes wrong.

Prior to that, though, here's a quick review of this chapter's key points.

BULLET POINTS

- When you need to store server-side state within a Flask webapp, use the `session` dictionary (and don't forget to set a hard-to-guess `secret_key`).
- You can pass a function as an argument to another function. Using the function's name (without the parentheses) gives you a **function object**, which can be manipulated like any other variable.
- When you use a function object as an argument to a function, you can have the receiving function **invoke** the passed-in function object by appending parentheses.
- A function can be **nested** inside an enclosing function's suite (and is only visible within the enclosing scope).
- In addition to accepting a function object as an argument, functions can **return** a nested function as a return value.
- `*args` is shorthand for "expand to a list of items."
- `**kwargs` is shorthand for "expand to a dictionary of keys and values." When you see "kw," think "keywords."
- Both `*` and `**` can also be used "on the way in," in that a list or keyword collection can be passed into a function as a single (expandable) argument.
- Using (`*args, **kwargs`) as a **function signature** lets you create functions that accept any number and type of arguments.
- Using the new function features from this chapter, you learned how to create a **function decorator**, which changes the behavior of an existing function without the need to change the function's actual code. This sounds freaky, but is quite a bit of fun (and is very useful, too).

Chapter 10's Code, 1 of 2

```
from flask import Flask, session

app = Flask(__name__)

app.secret_key = 'YouWillNeverGuess'

@app.route('/setuser/<user>')
def setuser(user: str) -> str:
    session['user'] = user
    return 'User value set to: ' + session['user']

@app.route('/getuser')
def getuser() -> str:
    return 'User value is currently set to: ' + session['user']

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

```
from flask import session

from functools import wraps

def check_logged_in(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if 'logged_in' in session:
            return func(*args, **kwargs)
        return 'You are NOT logged in.'
    return wrapper
```

This is "checker.py", which contains the code to this chapter's decorator: "check_logged_in".

This is "tmpl_decorator.py", which is a handy decorator-creating template for you to reuse as you see fit.

```
from functools import wraps

def decorator_name(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # 1. Code to execute BEFORE calling the decorated function.

        # 2. Call the decorated function as required, returning its
        #    results if needed.
        return func(*args, **kwargs)

        # 3. Code to execute INSTEAD of calling the decorated function.
    return wrapper
```

Chapter 10's Code, 2 of 2

```

from flask import Flask, session
from checker import check_logged_in
app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello from the simple webapp.'

@app.route('/page1')
@check_logged_in
def page1() -> str:
    return 'This is page 1.'

@app.route('/page2')
@check_logged_in
def page2() -> str:
    return 'This is page 2.'

@app.route('/page3')
@check_logged_in
def page3() -> str:
    return 'This is page 3.'

@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.'

app.secret_key = 'YouWillNeverGuessMySecretKey'

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

```

This is "simple_webapp.py", which pulls all of this chapter's code together. When you need to restrict access to specific URLs, base your strategy on this webapp's mechanism.

We think the use of decorators makes this webapp's code easy to read and understand. Don't you? 😊

11 exception handling

What to Do When Things 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, 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)

```



Long Exercise Solution

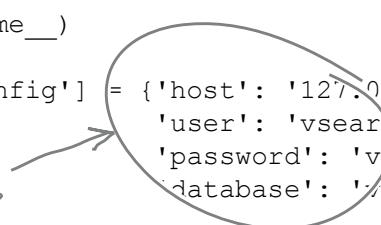
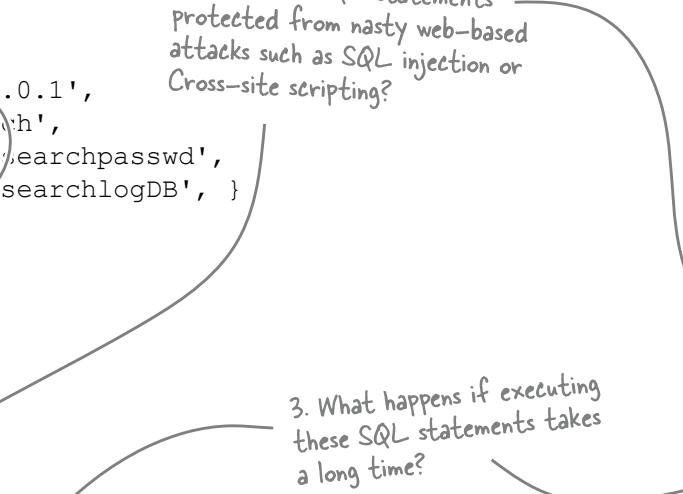
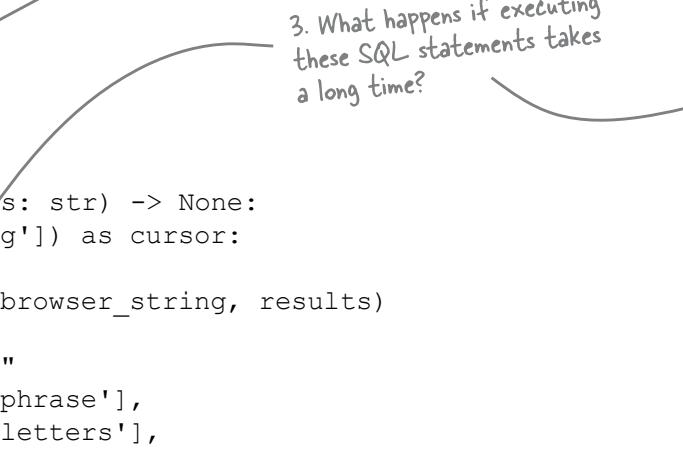
You were to take as long as you needed to read the code shown below (which is an updated version of the `vsearch4web.py` webapp). Then, using a pencil, you were to circle and annotate the parts you thought might cause problems when operating within a production environment. You were to highlight everything 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': 'vsearchh',
                        'password': 'vsearchpasswd',
                        'database': 'vsearchlogDB', }

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?
```

```
@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_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)

```

4. What happens if this call fails?

Databases Aren't Always Available

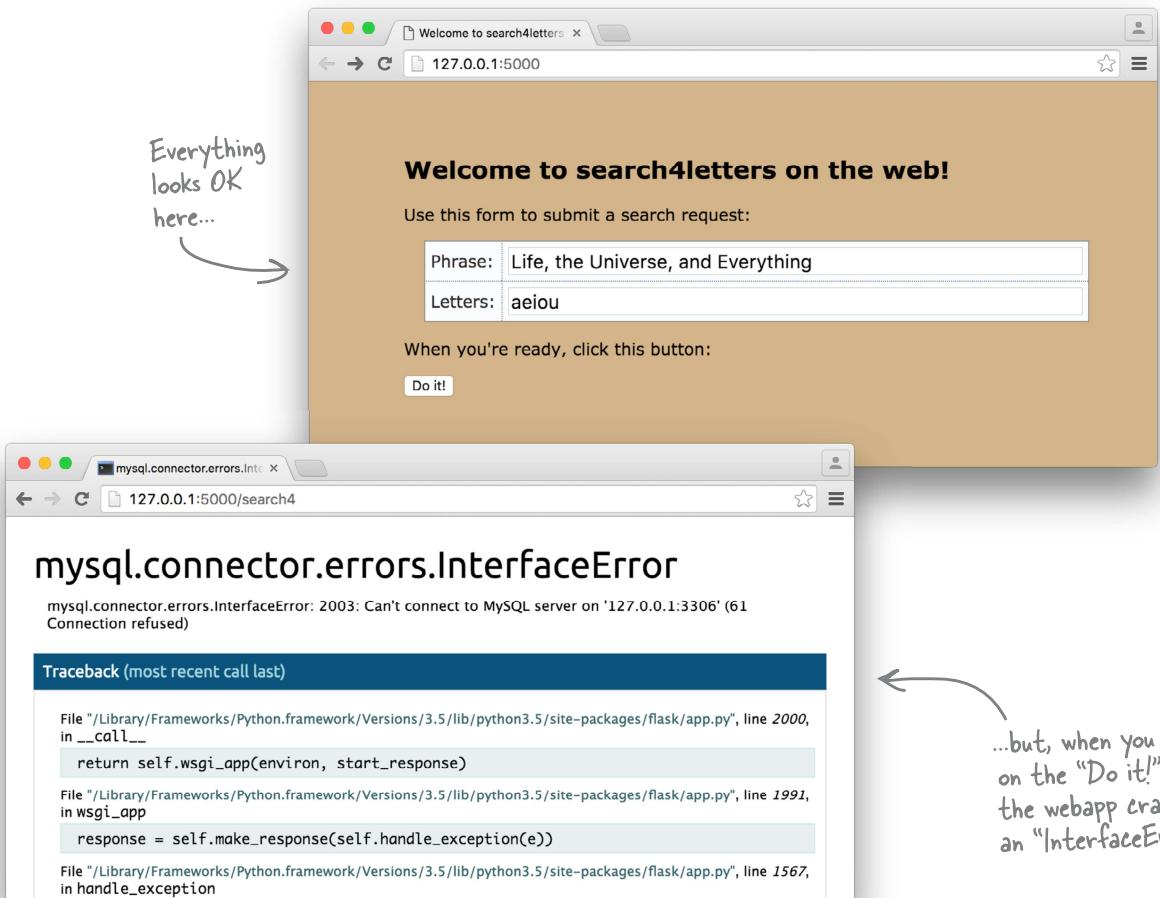
We've identified four potential issues with the `vsearch4web.py` code, and we concede that there may be many more, but we'll worry about these four issues for now. Let's consider each of the four issues in more detail (which we do here and on the next few pages, by simply describing the problems; *we'll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



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:

The figure consists of three screenshots of a web application. The top screenshot shows a search interface with fields for 'Phrase' (containing 'x'; show tables ;') and 'Letters' (containing 'aeiou'). A callout notes that SQL injection has no effect. The middle screenshot shows the results page with the submitted data and the response '{'a', 'e', 'o'}'. A callout notes that the JavaScript isn't executed. The bottom screenshot shows a similar setup with a more complex XSS payload (<script type='text/javascript'>alert('Hello!');</script>) in the 'Phrase' field, but the results page shows the same response as the middle one, indicating no execution.

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:

Do it!

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:

Do it!

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'}

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

Letters: aeiou

When you're ready, click this button:

Do it!

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'}

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

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):

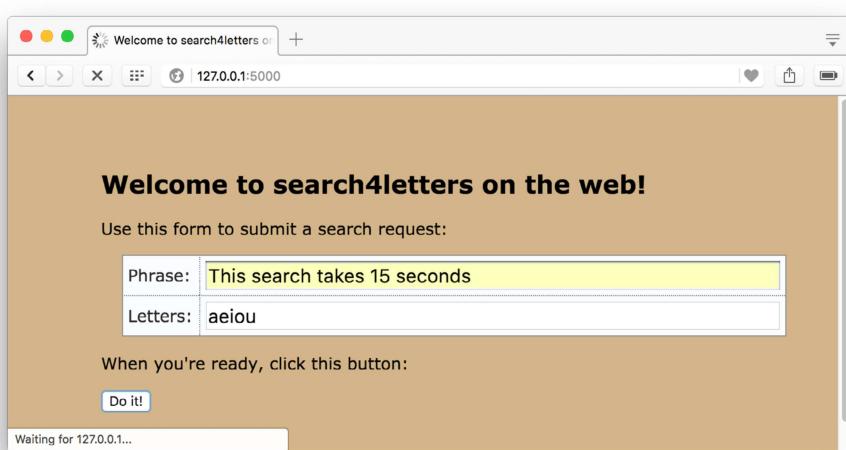
```
from time import sleep
```

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...and waits...



Your Function Calls Can Fail

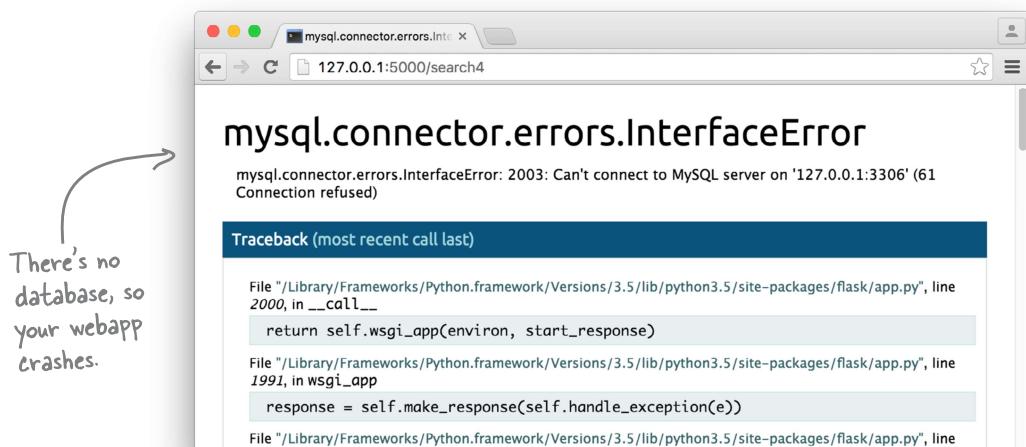
The final issue identified during this chapter's opening exercise relates to the function call to `log_request` within the `do_search` function:

4

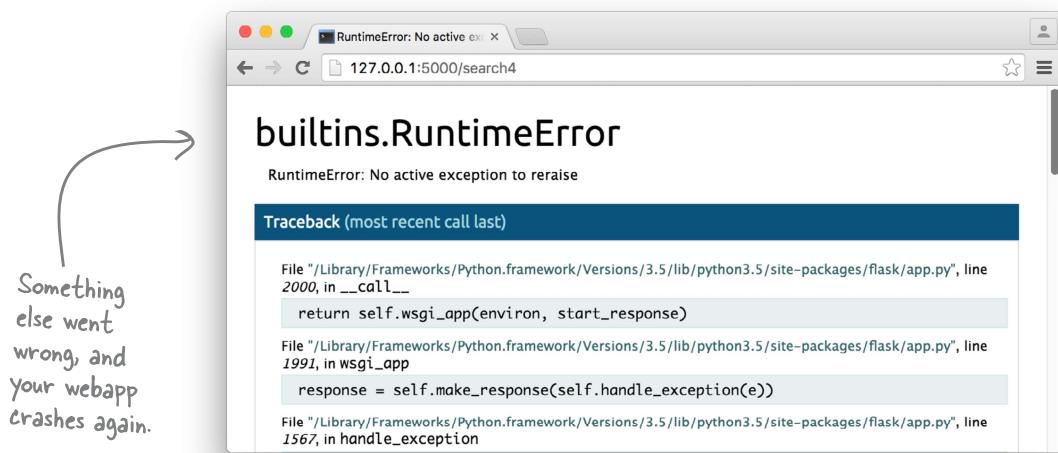
What happens if a function call fails?

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`:



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:



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

what to do?

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.

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 `?` 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.



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.

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>.

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

```
try_examples.py - /Users/paul/Desktop/_NewBo...
with open('myfile.txt') as fh:
    file_data = fh.read()
print(file_data)
Ln: 5 Col: 0
```

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:
FileNotFoundException: [Errno 2] No such file or directory: 'myfile.txt'
>>>
>>>
```

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

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

try try try

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 `FileNotFoundException` 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.

The diagram illustrates the evolution of Python code. On the left, a screenshot of a code editor shows a simple `try` block:

```
try_examples.py - /Users/paul/Desktop/_NewBo...
with open('myfile.txt') as fh:
    file_data = fh.read()
print(file_data)
Ln: 5 Col: 0
```

A callout points to the entire block with the text: "Note how the entire code snippet is indented under the 'try' statement."

An arrow points from this code to a second screenshot on the right, which shows the code with an `except` clause added:

```
try_examples2.py - /Users/paul/Desktop/_NewBook/ch11...
try:
    with open('myfile.txt') as fh:
        file_data = fh.read()
    print(file_data)
except FileNotFoundError:
    print('The data file is missing.')
Ln: 8 Col: 0
```

Two callouts point to the `except` block. One says: "The 'except' statement is indented to the same level as its associated 'try', and has its own suite." Another says: "This code is indented under the 'except' clause and only executes if the 'FileNotFoundException' 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.

```
Python 3.5.1 Shell
>>>
===== 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.
>>> |
Ln: 17 Col: 4
```

The new version of the code produces a much friendlier message 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.

```
Python 3.5.1 Shell
>>>
===== 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'
>>>
>>> |
Ln: 24 Col: 4
```

multiple `except`s

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):

```
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.') ←
```

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

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:



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.

catching all exceptions

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:

A screenshot of a Mac OS X terminal window titled "Python 3.5.2 Shell". The window shows a Python traceback. The text is as follows:

```
>>>
=====
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'
>>>
```

Annotations:

- A curved arrow points from the text "Another exception has occurred." to the line "IsADirectoryError: [Errno 21] Is a directory: 'myfile.txt'".
- The line number "Ln: 15 Col: 4" is at the bottom right of the terminal window.

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:

A screenshot of a Mac OS X terminal window titled "try_examples4.py - /Users/paul/Desktop/_NewBook/ch11/tr...". The window shows a Python script with a modified `try` block. The text is as follows:

```
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.')}
```

Annotations:

- A curly brace on the left side of the code is labeled "This 'except' statement is 'bare': it does not refer to a specific exception."
- A curly brace on the right side of the code is labeled "This code provides a catch-all exception handler."
- The line number "Ln: 12 Col: 0" is at the bottom right of the terminal window.

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:

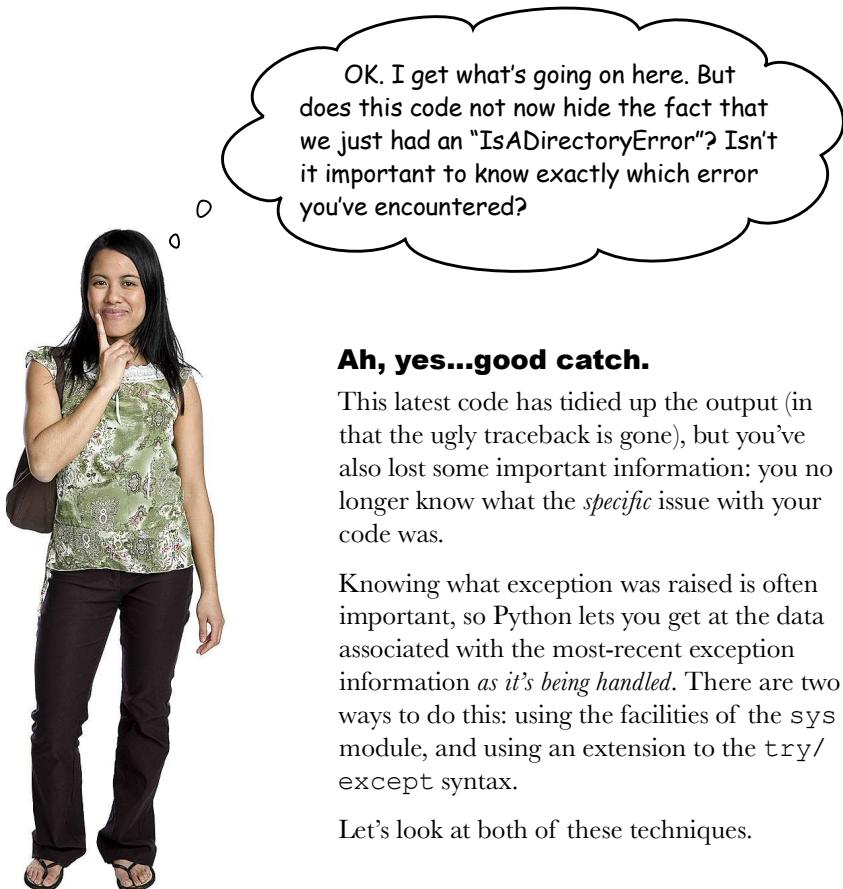
A screenshot of a Mac OS X terminal window titled "Python 3.5.2 Shell". The window shows a Python traceback followed by a friendly error message. The text is as follows:

```
>>>
=====
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.
>>>
```

Annotations:

- A curved arrow points from the text "This looks better." to the line "Some other error occurred."
- The line number "Ln: 16 Col: 4" is at the bottom right of the terminal window.

Haven't We Just Lost Something?



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

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 (presumably 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.

what went wrong?

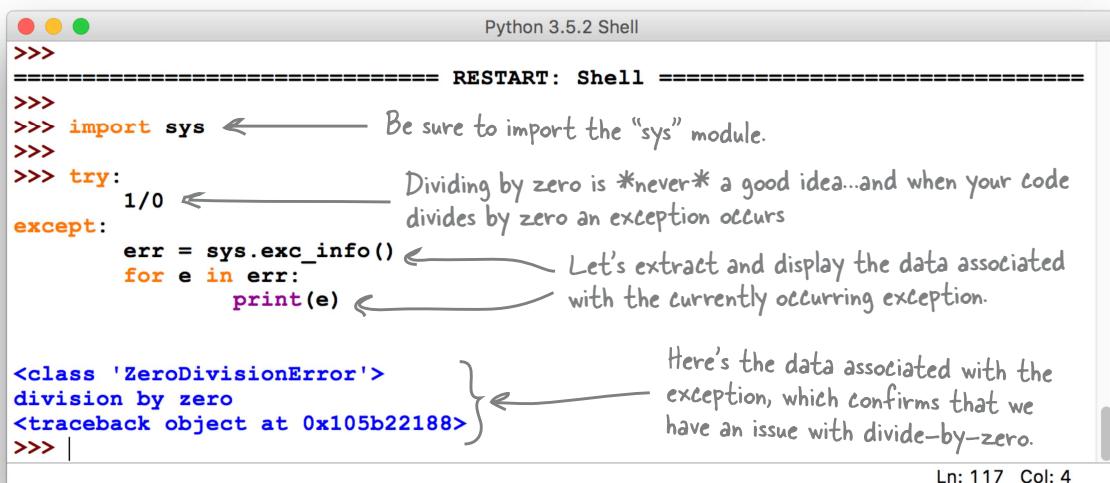
Learning About Exceptions from "sys"

The standard library comes with a module called `sys` that provides access to the interpreters, *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>.



The screenshot shows a Python 3.5.2 Shell window with the following interaction:

```
>>> ===== RESTART: Shell =====
>>> import sys
>>> try:
...     1/0
... except:
...     err = sys.exc_info()
...     for e in err:
...         print(e)
...
<class 'ZeroDivisionError'>
division by zero
<traceback object at 0x105b22188>
>>> |
```

Annotations explain the code:

- `Be sure to import the "sys" module.`
- `Dividing by zero is *never* a good idea...and when your code divides by zero an exception occurs`
- `Let's extract and display the data associated with the currently occurring exception.`
- `Here's the data associated with the exception, which confirms that we have an issue with divide-by-zero.`

Ln: 117 Col: 4

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 (`FileNotFoundException` and `PermissionError`), as well as provides a generic catch-all `except` suite (to handle everything else):

This code works, but
doesn't really tell
you much when some
unexpected exception
occurs.

```
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.')
|
```

Ln: 12 Col: 0

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.

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`:

Recall that all the
exceptions inherit
from “Exception”.

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

```
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 Exception as err:
    print('Some other error occurred:', str(err))
|
```

Ln: 12 Col: 0

Unlike the “bare” `except` catch-all shown above, this one arranges for the exception object to be assigned to the “`err`” variable.

The value of “`err`” is then used as part of the friendly message (as it's always a good idea to report all exceptions).

one last spin



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:

```
Python 3.5.2 Shell
>>>
===== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ======
The data file is missing.
>>>
Ln: 7 Col: 4
```

← "myfile.txt" doesn't exist.

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

```
Python 3.5.2 Shell
>>>
===== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ======
The data file is missing.
>>>
===== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ======
This is not allowed.
>>> |
Ln: 10 Col: 4
```

← The file exists, but you can't read it.

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

```
Python 3.5.2 Shell
>>>
===== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ======
The data file is missing.
>>>
===== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ======
This is not allowed.
>>>
===== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ======
Some other error occurred: [Errno 21] Is a directory: 'myfile.txt'
>>>
Ln: 23 Col: 4
```

Some other exception has occurred. In this case, what you thought was a file is in fact a folder.

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

```
Python 3.5.2 Shell
>>>
===== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ======
The data file is missing.
>>>
===== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ======
This is not allowed.
>>>
===== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ======
Some other error occurred: [Errno 21] Is a directory: 'myfile.txt'
>>>
===== RESTART: /Users/paul/Desktop/_NewBook/ch11/try_examples5.py ======
Empty (well... except for this line).
>>>
Ln: 27 Col: 4
```

Success! No exceptions occur, so the "try" suite runs to completion.

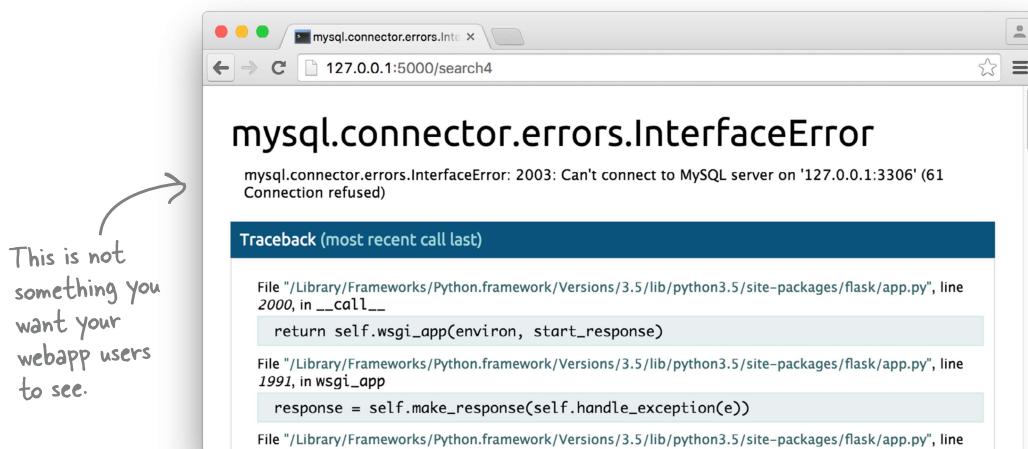
Getting Back to Our Webapp Code

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:

```
...
@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) ← 4. What happens if this call fails?
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results)
...

```

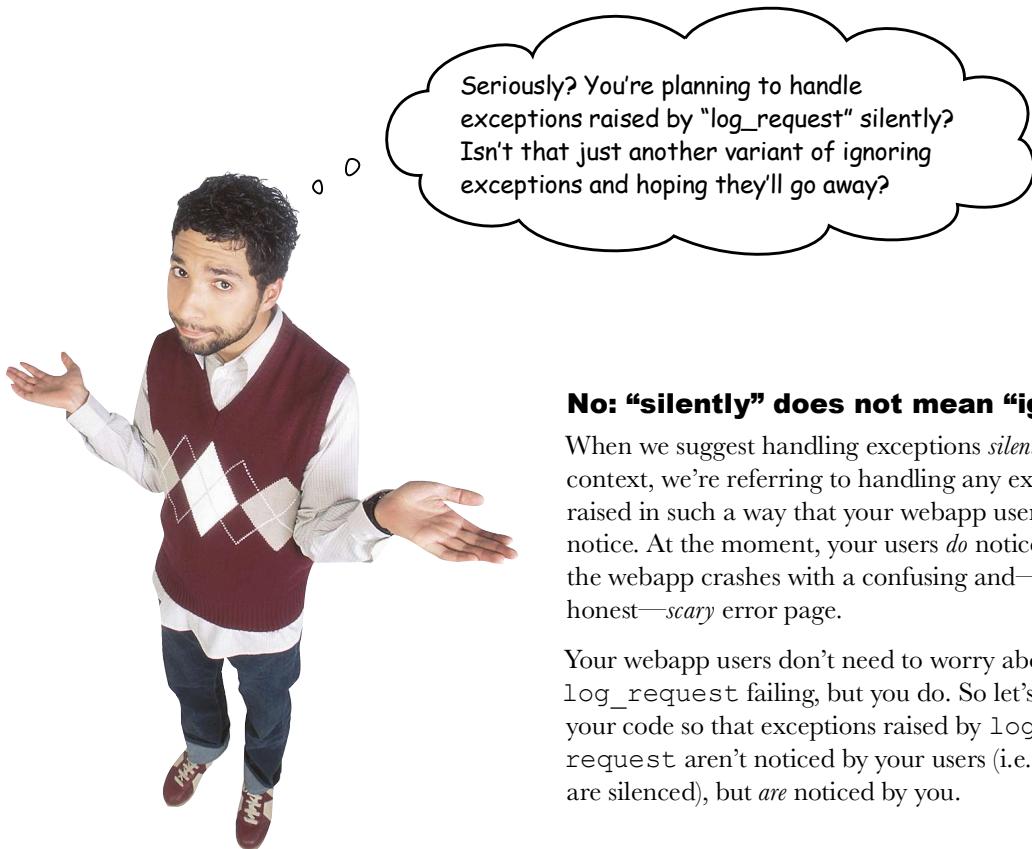
Based on our investigations, we learned that this call might fail if the backend database is unavailable, or if some other error occurs. When an error (of any type) occurs, the webapp responds with an unfriendly error page, which is likely to confuse (rather than enlighten) your webapp's users:



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*.

don't make a sound

Silently Handling Exceptions



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: Doesn’t all this `try/except` stuff just make my code harder to read and understand?

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:

```
@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,)
```

This line of code needs to be protected in case it fails (raising a runtime error).

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

```
@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))
    .....
    .....
    .....
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

Don't forget to call "log_request" as part of the code you add.



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`:

```
@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))
```

This is the catch-all.

`try:`

`.....
log_request(request, results)`

The call to "log_request" is moved into the suite associated with a new "try" statement.

`.....
except Exception as err:`

`.....
print('***** Logging failed with this error:', str(err))`

When a runtime error occurs, this message is displayed on screen for the admin only. Your user sees nothing.

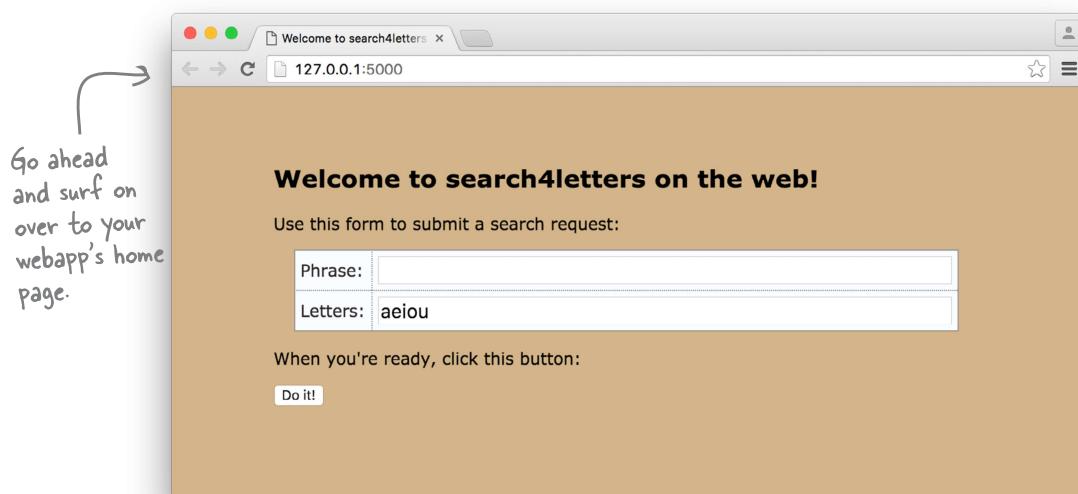


—(Extended) Test Drive, 1 OF 3—

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:

```
$ python3 vsearch4web.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with fsevents reloader
* Debugger is active!
* Debugger pin code: 184-855-980
```

Your webapp is up and running,
waiting to hear from a browser...



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

```
...
* Debugger pin code: 184-855-980
127.0.0.1 - - [14/Jul/2016 10:54:31] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [14/Jul/2016 10:54:31] "GET /static/hf.css HTTP/1.1" 200 -
127.0.0.1 - - [14/Jul/2016 10:54:32] "GET /favicon.ico HTTP/1.1" 404 -
```

These 200s confirm that your webapp is up and running (and serving up its home page). All is good at this point.

BTW: Don't worry about this 404...we haven't defined a "favicon.ico" file for our webapp (so it gets reported as not found when your browser asks for it).

test test test



—(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 `/search`:

Welcome to search4letters on the web!

Use this form to submit a search request:

Phrase:	Testing out catch all code
Letters:	testing

When you're ready, click this button:

Do it!

Here are your results:

You submitted the following data:

Phrase:	Testing out catch all code
Letters:	testing

When "Testing out catch all code" is search for "testing", the following results are returned:

{'e', 'n', 'g', 's', 't', 'i'}

```
...  
127.0.0.1 - - [14/Jul/2016 10:54:32] "GET /favicon.ico HTTP/1.1" 404 -  
***** Logging failed with this error: 2003: Can't connect to MySQL server on '127.0.0.1:3306'  
(61 Connection refused)  
127.0.0.1 - - [14/Jul/2016 10:55:55] "POST /search4 HTTP/1.1" 200 -
```

This message is generated by your catch-all exception-handling code. The webapp user doesn't see it.

Even though an error occurred, the webapp didn't crash. In other words, the search worked (but the webapp user isn't aware that the logging failed).



(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`:

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

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

```
...
***** Logging failed with this error: 1045 (28000): Access denied for user 'vsearchwrong'@'localhost' (using password: YES)
```

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`):

```
def log_request(req: 'flask_request', res: str) -> None:
    with UseDatabase(app.config['dbconfig']) as cursor:
        _SQL = """insert into logwrong
                  (phrase, letters, ip, browser_string, results)
                  values
                  (%s, %s, %s, %s, %s)"""
        ...
...
```

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

```
...
***** Logging failed with this error: 1146 (42S02): Table 'vsearchlogdb.logwrong' doesn't exist
```

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:

```
def log_request(req: 'flask_request', res: str) -> None:
    raise Exception("Something awful just happened.") ←
    with UseDatabase(app.config['dbconfig']) as cursor:
        ...
...
```

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

```
...
***** Logging failed with this error: Something awful just happened.
```

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:

```
...
@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,
                           )
...

```

All of this code
needs to be
protected, too.

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”?

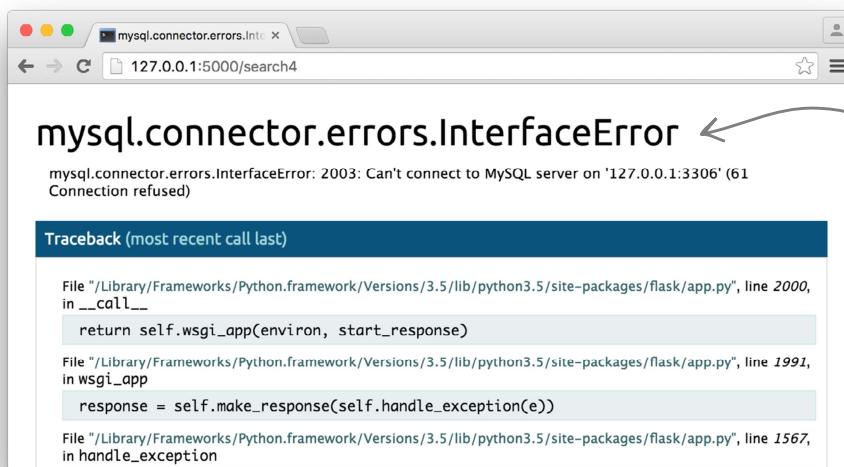
Knowing what we now know about `try/except`, we could add some code to the `view_the_log` function to protect the use of the `UseDatabase` context manager:

```

    ...
    @app.route('/viewlog')
    @check_logged_in
    def view_the_log() -> 'html':
        try:
            Another
            catch-all
            exception
            handler
            ↗
            with UseDatabase(app.config['dbconfig']) as cursor:
                ...
                ...
        except Exception as err:
            ↗
            print('Something went wrong:', str(err))
            ↘
            The rest of
            the function's
            code goes
            here.
            ↙

```

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:



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.

watch your imports

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:

```
...
@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    try:
        with UseDatabase(app.config['dbconfig']) as cursor:
            ...
    except mysql.connector.errors.InterfaceError as err:
        print('Is your database switched on? Error:', str(err))
    except Exception as err:
        print('Something went wrong:', str(err))
    ...

```

The rest of
the function's
code still goes
in here.

Add another "except"
statement to handle
a specific exception

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:

```
import mysql.connector

class UseDatabase:
    This is the
    context
    manager code
    in "DBcm.py".
    ↗
        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()
            ↘
                Now that you've
                seen "exc_info",
                it should be clear
                what these method
                arguments refer to:
                exception data.
```

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:

The screenshot shows a Python 3.5.2 Shell window with the following code and annotations:

```
>>>
>>> class ConnectionError(Exception):
>>>     pass
>>>
>>> raise ConnectionError('Cannot connect... is it time to panic?')
Traceback (most recent call last):
  File "<pyshell#74>", line 1, in <module>
    raise ConnectionError('Cannot connect... is it time to panic?')
ConnectionError: Cannot connect... is it time to panic?
>>>
>>>
>>> try:
>>>     raise ConnectionError('Whoops!')
>>> except ConnectionError as err:
>>>     print('Got:', str(err))
Got: Whoops!
>>>
>>> |
```

Annotations:

1. Create a new class called "ConnectionError" that inherits from the "Exception" class.
2. "pass" is Python's empty statement that creates the empty class.
3. Raising our new exception (with "raise") results in a traceback message.
4. Catch the "ConnectionError" exception using "try/except".
5. The "ConnectionError" was caught, allowing us to customize the error message.

Ln: 148 Col: 4

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.

Define your custom exception.

```
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()
```

Add code to "raise" a "ConnectionError".



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:

Use your pencil to show the changes you'd make to this code now that the "ConnectionError" exception exists.



```
from DBcm import UseDatabase
import mysql.connector
...
            the_row_titles=titles,
            the_data=contents,)

except mysql.connector.errors.InterfaceError as err:
    print('Is your database switched on? Error:', str(err))
except Exception as err:
    print('Something went wrong:', str(err))
return 'Error'
```

Sharpen your pencil Solution

Define the custom exception as an "empty" class that inherits from "Exception".

```
1 import mysql.connector

class ConnectionError(Exception):
    pass

class UseDatabase:

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

    def __enter__(self) -> 'cursor':
        try:
            self.conn = mysql.connector.connect(**self.configuration)
            self.cursor = self.conn.cursor()
            return self.cursor
        except mysql.connector.errors.InterfaceError as err:
            raise ConnectionError(err)

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

A new "try/except" construct protects the database connection code.

You don't need to import "mysql.connector" anymore (as "DBcm" does this for you).

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.

Within the "DBcm.py" code, refer to the backend database-specific exceptions by their full name.

Raise the custom exception.

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:

```
from DBcm import UseDatabase, ConnectionError
import mysql.connector

...
the_row_titles=titles,
the_data=contents,)

ConnectionError
except mysql.connector.errors.InterfaceError as err:
    print('Is your database switched on? Error:', str(err))
except Exception as err:
    print('Something went wrong:', str(err))
return 'Error'
```

Change the first "except" statement to look for a "ConnectionError" as opposed to an "InterfaceError".

Be sure to import the "ConnectionError" exception from "DBcm".



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 MySQL server on '127.0.0.1:3306'
(61 Connection refused)
127.0.0.1 - - [16/Jul/2016 21:21:51] "GET /viewlog HTTP/1.1" 200 -
```

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 MySQL server on '127.0.0.1:3306'
(61 Connection refused)
127.0.0.1 - - [16/Jul/2016 21:22:58] "GET /viewlog HTTP/1.1" 200 -
```



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:

```
import mysql.connector

class ConnectionError(Exception):
    pass

class UseDatabase:
    def __init__(self, config: dict):
        self.configuration = config

    def __enter__(self) -> 'cursor':
        try:
            self.conn = mysql.connector.connect(**self.configuration)
            self.cursor = self.conn.cursor()
            return self.cursor
        except mysql.connector.errors.InterfaceError as err:
            raise ConnectionError(err)

    def __exit__(self, exc_type, exc_value, exc_traceback):
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
```

This code
can raise a
“ProgrammingError”
exception.

But what about exceptions that occur
within the “with” suite? These happen
after the “`__enter__`” method ends
but *before* the “`__exit__`” method
starts.

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 `SQLLError` 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`:

```
import mysql.connector

class ConnectionError(Exception):
    pass

class CredentialsError(Exception):
    pass

class SQLLError(Exception):
    pass

class UseDatabase:
    def __init__(self, configuration: dict):
        self.config = configuration
    ...

```

Two additional
classes define your
two new exceptions.

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:

```
...
try:
    self.conn = mysql.connector.connect(**self.config)
    self.cursor = self.conn.cursor()
    return self.cursor
except mysql.connector.errors.InterfaceError as err:
    raise ConnectionError(err)
except mysql.connector.errors.ProgrammingError as err:
    raise CredentialsError(err)

def __exit__(self, exc_type, exc_value, exc_traceback):
    self.conn.commit()
    self.cursor.close()
    self.conn.close()
```

Add this code to
the "`__enter__`"
method to deal
with any login issues.

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.

catching more exceptions

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:

Be sure to import your new exception.

```
...  
from DBcm import UseDatabase, ConnectionError, CredentialsError  
...
```

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:

```
@app.route('/viewlog')  
@check_logged_in  
def view_the_log() -> 'html':  
    try:  
        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,)  
    except ConnectionError as err:  
        print('Is your database switched on? Error:', str(err))  
    except CredentialsError as err:  
        print('User-id/Password issues. Error:', str(err)) ←  
    except Exception as err:  
        print('Something went wrong:', str(err))  
    return 'Error'
```

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:

Add this code to "view_the_log" to catch when your code uses the wrong username or password with MySQL.

```
...  
User-id/Password issues. Error: 1045 (28000): Access denied for user 'vsearcherror'@'localhost'  
(using password: YES)  
127.0.0.1 - - [25/Jul/2016 16:29:37] "GET /viewlog HTTP/1.1" 200 -
```

Now that your code knows all about "CredentialsError", you generate an exception-specific error message.

Handling SQL Error 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:

```
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, ))
```

We need to worry about what happens if something goes wrong with this code (i.e., the code within the "with" suite).

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:

```
def __exit__(self, exc_type, exc_value, exc_traceback):
    self.conn.commit()
    self.cursor.close()
    self.conn.close()
```

The three exception arguments are ready for use.

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.

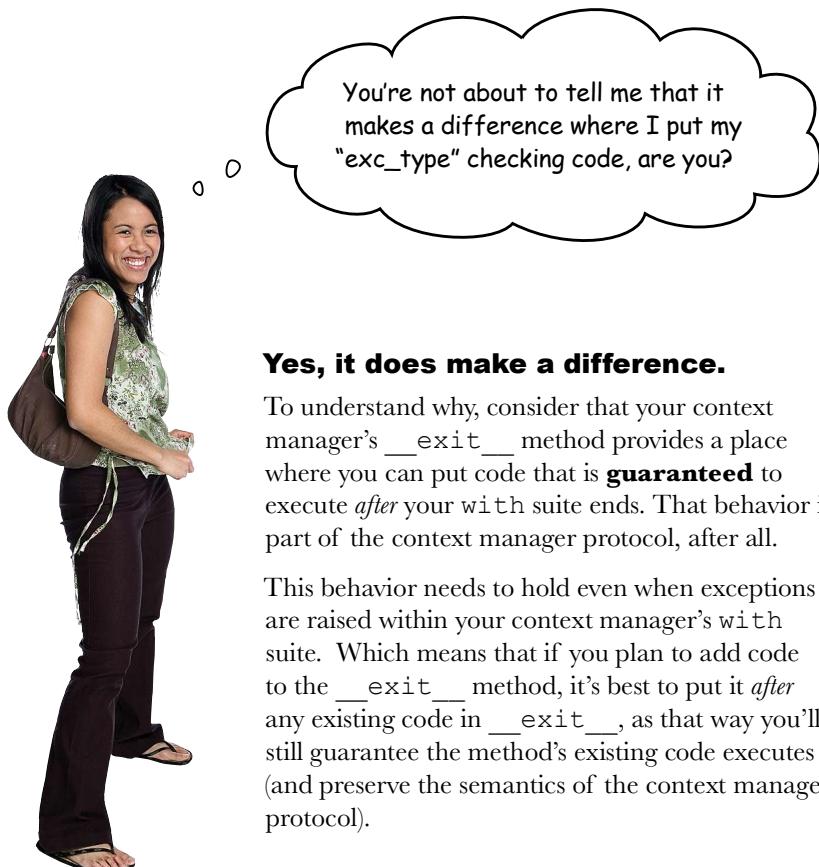
Let's exploit this behavior to raise an `SQLLError` whenever something goes wrong within the `UseDatabase` context manager's `with` suite.

"None" is Python's null value.

`exc_type exc_value exc_traceback`

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.



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:

```
def __exit__(self, exc_type, exc_value, exc_traceback):  
    self.conn.commit()  
    self.cursor.close()  
    self.conn.close()
```

If you add code in here, and that code raises an exception, the three existing lines of code won't be executed.

Adding code **after** the three existing lines of code ensures "`__exit__`" does its thing **before** any passed-in exception is dealt with.

Raising an SQLError

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

```
import mysql.connector

class ConnectionError(Exception):
    pass

class CredentialsError(Exception):
    pass

class SQLError(Exception):
    pass

class UseDatabase:
    def __init__(self, config: dict):
        self.configuration = config
    ...

```

Here's where
you added in
the "SQLError"
exception.



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:

If a "ProgrammingError"
occurs, raise an
"SQLError".



```
def __exit__(self, exc_type, exc_value, exc_traceback):
    self.conn.commit()
    self.cursor.close()
    self.conn.close()
    if exc_type is mysql.connector.errors.ProgrammingError:
        raise SQLError(exc_value)
```

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:

```
...
self.conn.close()
if exc_type is mysql.connector.errors.ProgrammingError:
    raise SQLError(exc_value)
elif exc_type:
    raise exc_type(exc_value)
```



This "elif" raises
any other exception
that might occur.



Test Drive

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

Add this code
into the "view_
the_log" function
within your
"vsearch4web.py"
webapp.

```
...  
except ConnectionError as err:  
    print('Is your database switched on? Error:', str(err))  
except CredentialsError as err:  
    print('User-id/Password issues. Error:', str(err))  
except SQLError as err:  
    print('Is your query correct? Error:', str(err))  
except Exception as err:  
    print('Something went wrong:', str(err))  
return 'Error'
```

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:

```
...  
Is your query correct? Error: 1146 (42S02): Table 'vsearchlogdb.logerror' doesn't exist  
127.0.0.1 - - [25/Jul/2016 21:38:25] "GET /viewlog HTTP/1.1" 200 -
```

No more generic "ProgrammingError" exceptions from MySQL Connector, as your custom exception-handling code catches these errors now.

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

```
...  
Something went wrong: Some unknown exception.  
127.0.0.1 - - [25/Jul/2016 21:43:14] "GET /viewlog HTTP/1.1" 200 -
```

If something unexpected happens, your code handles it.

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.



And the really nice thing about this is that this code takes the generic "ProgrammingError" exception provided by the MySQL Connector module and turns it into two custom exceptions that have specific meaning for our webapp.

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.

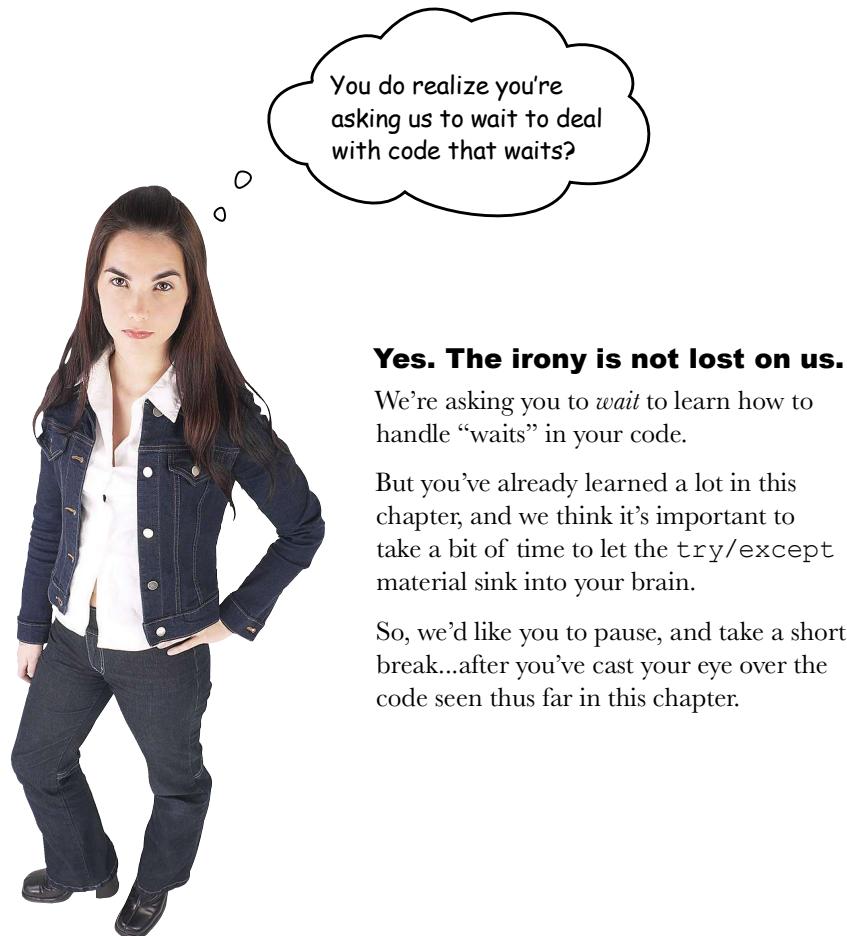
The question is: *what do you do if this write or read takes a long time?*

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?*



Chapter 11's Code, 1 of 3

This is "try_example.py".



```

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 Exception as err:
    print('Some other error occurred:', str(err))

```

```

import mysql.connector

class ConnectionError(Exception):
    pass

class CredentialsError(Exception):
    pass

class SQLLError(Exception):
    pass

class UseDatabase:
    def __init__(self, config: dict):
        self.configuration = config

    def __enter__(self) -> 'cursor':
        try:
            self.conn = mysql.connector.connect(**self.configuration)
            self.cursor = self.conn.cursor()
            return self.cursor
        except mysql.connector.errors.InterfaceError as err:
            raise ConnectionError(err)
        except mysql.connector.errors.ProgrammingError as err:
            raise CredentialsError(err)

    def __exit__(self, exc_type, exc_value, exc_traceback):
        self.conn.commit()
        self.cursor.close()
        self.conn.close()
        if exc_type is mysql.connector.errors.ProgrammingError:
            raise SQLLError(exc_value)
        elif exc_type:
            raise exc_type(exc_value)

```

This is the exception-savvy version of "DBcm.py".



Chapter 11's Code, 2 of 3

This is the version of "vsearch4web.py" that makes your users wait...

```
from flask import Flask, render_template, request, escape, session
from flask import copy_current_request_context

from vsearch import search4letters

from DBcm import UseDatabase, ConnectionError, CredentialsError, SQLSError
from checker import check_logged_in

from time import sleep

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.'

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':

    @copy_current_request_context
    def log_request(req: 'flask_request', res: str) -> None:
        sleep(15) # This makes log_request really slow...
        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,))

    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
```

It's probably a good idea to protect this "with" statement in much the same way as you protected the "with" statement in "view_the_log" (on the next page).

The rest of "do_search" is at the top of the next page. →

Chapter 11's Code, 3 of 3

```

results = str(search4letters(phrase, letters))
try:
    log_request(request, results)
except Exception as err:
    print('***** Logging failed with this error:', str(err))
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':
    try:
        with UseDatabase(app.config['dbconfig']) as cursor:
            _SQL = """select phrase, letters, ip, browser_string, results
                      from log"""
            cursor.execute(_SQL)
            contents = cursor.fetchall()
        # raise Exception("Some unknown exception.")
        titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', 'Results')
        return render_template('viewlog.html',
                              the_title='View Log',
                              the_row_titles=titles,
                              the_data=contents,)
    except ConnectionError as err:
        print('Is your database switched on? Error:', str(err))
    except CredentialsError as err:
        print('User-id/Password issues. Error:', str(err))
    except SQLError as err:
        print('Is your query correct? Error:', str(err))
    except Exception as err:
        print('Something went wrong:', str(err))
    return 'Error'

app.secret_key = 'YouWillNeverGuessMySecretKey'

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

```

This is the rest of the "do_search" function.

11¾ a little bit of threading



*** Dealing with Waiting ***



Your code can sometimes take a long time to execute.

Depending on who notices, this may or may not be an issue. If some code takes 30 seconds to do its thing “behind the scenes,” the wait may not be an issue. However, if your user is waiting for your application to respond, and it takes 30 seconds, everyone notices. What you should do to fix this problem depends on what you’re trying to do (and who’s doing the waiting). In this short chapter, we’ll briefly discuss some options, then look at one solution to the issue at hand: *what happens if something takes too long?*

write wait read wait

Waiting: What to Do?

When you write code that has the potential to make your users wait, you need to think carefully about what it is you are trying to do. Let's consider some points of view.



Maybe it is the case that waiting for a write is *different* from waiting for a read, especially as it relates to how your webapp works?

Let's take another look at the SQL queries in `log_request` and `view_the_log` to see how you're using them.

How Are You Querying Your Database?

In the `log_request` function, we are using an SQL `INSERT` to add details of the request to our backend database. When `log_request` is called, it **waits** while the `INSERT` is executed by `cursor.execute`:

```
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, ))
```

At this point, the webapp "blocks" while it waits for the backend database to do its thing.

The same holds for the `view_the_log` function, which also **waits** whenever the SQL `SELECT` query is executed:

```
@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    try:
        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,
                                  )
    except ConnectionError as err:
        ...
        To save on space, we're not showing all of the code for "view_the_log". The exception-handling code still goes here.
```

Your webapp "blocks" here, too, while it waits for the database.



Geek Bits

Code that waits for something external to complete is referred to as "blocking code," in that the execution of your program is **blocked** from continuing until the wait is over. As a general rule, blocking code that takes a noticeable length of time is bad.

Both functions block. However, look closely at what happens *after* the call to `cursor.execute` in both functions. In `log_request`, the `cursor.execute` call is the last thing that function does, whereas in `view_the_log`, the results of `cursor.execute` are used by the rest of the function.

Let's consider the implications of this difference.

insert is not select

Database INSERTs and SELECTs Are Different

If you're reading the title to this page and thinking "Of course they are!", be assured that (this late in this book) we haven't lost our marbles.

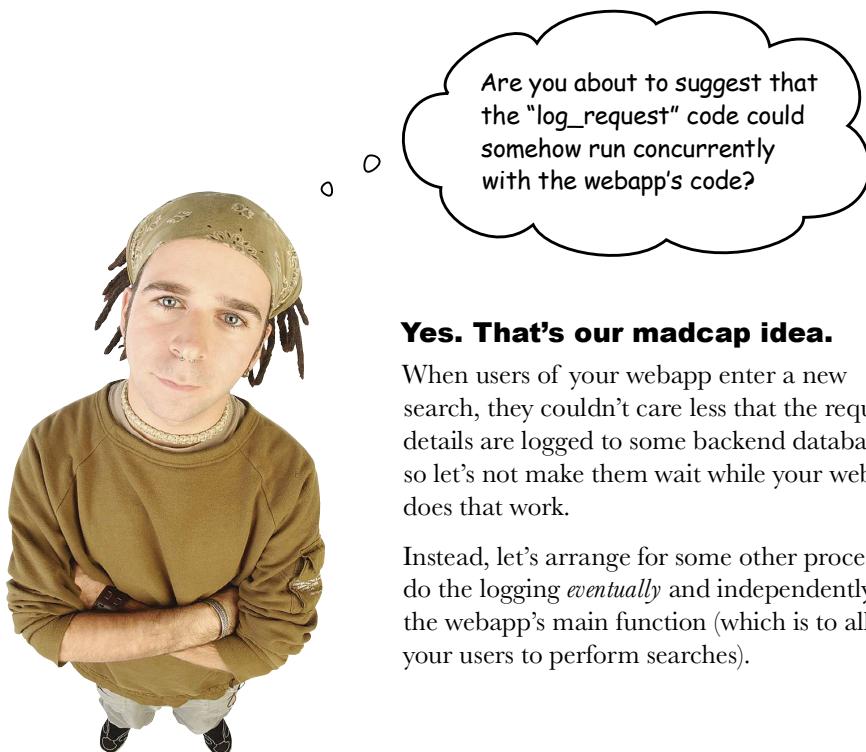
Yes: an `SQL INSERT` is *different* from an `SQL SELECT`, but, as it relates to your use of both queries in your webapp, it turns out that the `INSERT` in `log_request` doesn't need to block, whereas the `SELECT` in `view_the_log` does, which makes the queries *very* different.

This is a key observation.

If the `SELECT` in `view_the_log` doesn't wait for the data to return from the backend database, the code that follows `cursor.execute` will likely fail (as it'll have no data to work with). The `view_the_log` function **must** block, as it has to wait for data *before* proceeding.

When your webapp calls `log_request`, it wants the function to log the details of the current web request to the database. The calling code doesn't really care *when* this happens, just that it does. The `log_request` function returns no value, nor data; the calling code isn't waiting for a response. All the calling code cares about is that the web request is logged *eventually*.

Which begs the question: why does `log_request` force its callers to wait?



Yes. That's our madcap idea.

When users of your webapp enter a new search, they couldn't care less that the request details are logged to some backend database, so let's not make them wait while your webapp does that work.

Instead, let's arrange for some other process to do the logging *eventually* and independently of the webapp's main function (which is to allow your users to perform searches).

Doing More Than One Thing at Once

Here's the plan: you're going to arrange for the `log_request` function to execute independently of your main webapp. To do this, you're going to adjust your webapp's code so each call to `log_request` runs concurrently. This will mean that your webapp no longer has to wait for `log_request` to complete before servicing another request from another user (i.e., no more delays).

If `log_request` takes an instant, a few seconds, a minute, or even hours to execute, your webapp doesn't care (and neither does your user). What you care about is that the code eventually executes.

Concurrent code: you have options

When it comes to arranging for some of your application's code to run concurrently, Python has a few options. As well as lots of support from third-party modules, the standard library comes with some built-in goodies that can help here.

One of the most well known is the `threading` library, which provides a high-level interface to the threading implementation provided by the operating system hosting your webapp. To use the library, all you need to do is import the `Thread` class from the `threading` module near the top of your program code:

```
from threading import Thread
```

Go ahead and add this line of code near the top of your `vsearch4web.py` file.

Now the fun starts.

To create a new thread, you create a `Thread` object, assigning the name of the function you want the thread to execute to a named argument called `target`, and providing any arguments as a tuple to another named argument called `args`. The created `Thread` object is then assigned to a variable of your choosing.

As an example, let's assume that you have a function called `execute_slowly`, which takes three arguments, which we'll assume are three numbers. The code that invokes `execute_slowly` has assigned the three values to variables called `glacial`, `plodding`, and `leaden`. Here's how `execute_slowly` is invoked normally (i.e., without our worrying about concurrent execution):

```
execute_slowly(glacial, plodding, leaden)
```

If `execute_slowly` takes 30 seconds to do what it has to do, the calling code blocks and waits for 30 seconds before doing anything else. Bummer.

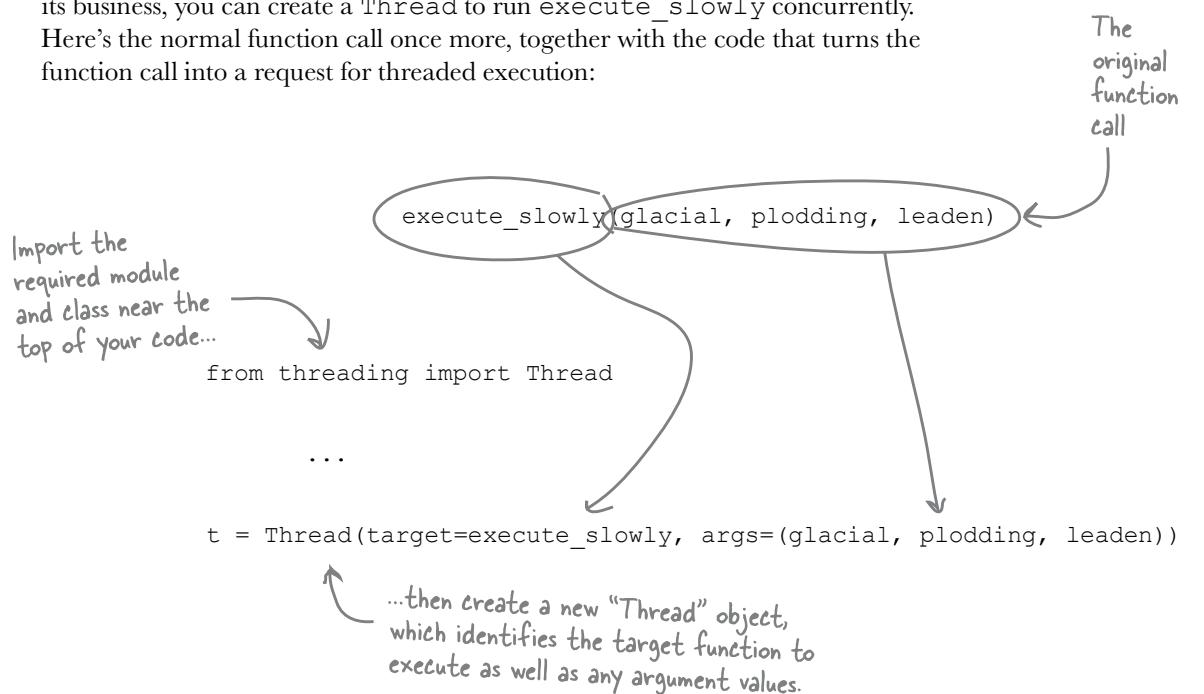
For the full list of (and all the details about) Python's standard library concurrency options, see <https://docs.python.org/3/library/concurrency.html>.

gotta love threads

Don't Get Bummed Out: Use Threads

In the big scheme of things, waiting 30 seconds for the `execute_slowly` function to complete doesn't sound like the end of the world. But, if your user is sitting and waiting, they'll be wondering what's gone wrong.

If your application can continue to run while `execute_slowly` goes about its business, you can create a `Thread` to run `execute_slowly` concurrently. Here's the normal function call once more, together with the code that turns the function call into a request for threaded execution:



Granted, this use of `Thread` looks a little strange, but it's not really. The key to understanding what's going on here is to note that the `Thread` object has been assigned to a variable (`t` in this example), and that the `execute_slowly` function has yet to execute.

Assigning the `Thread` object to `t` *prepares* it for execution. To ask Python's threading technology to run `execute_slowly`, start the thread like this:

`t.start()`

When you call "start", the function associated with the "t" thread is scheduled for execution by the "threading" module.

At this point, the code that called `t.start` continues to run. The 30-second wait that results from running `execute_slowly` has no effect on the calling code, as `execute_slowly`'s execution is handled by Python's `threading` module, not by you. The `threading` module conspires with the Python interpreter to run `execute_slowly` *eventually*.



Sharpen your pencil

When it comes to calling `log_request` in your webapp code, there's only one place you need to look: in the `do_search` function. Recall that you've already put your call to `log_request` inside a `try/except` to guard against unexpected runtime errors.

Note, too, that we've added a 15-second delay—using `sleep(15)`—to our `log_request` code (making it slow). Here's the current code to `do_search`:

```
@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))
    try:
        log_request(request, results)
    except Exception as err:
        print('***** Logging failed with this error:',
              str(err))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)
```

Here's how you currently invoke "log_request".



We are going to assume that you have already added `from threading import Thread` to the top of your webapp's code.

Grab your pencil, and in the space provided below, write the code you'd insert into `do_search` instead of the standard call to `log_request`.

Add the threading code you'd use to eventually execute "log_request".



.....

.....

.....

Sharpen your pencil Solution



When it comes to calling `log_request` in your webapp code, there's only one place you need to look: in the `do_search` function. Recall that you've already put your call to `log_request` inside a `try/except` to guard against unexpected run-time errors.

Note, too, that we've added a 15 second delay - using `sleep(15)` - to our `log_request` code (making it slow). Here's the current code to `do_search`:

```
@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))
    try:
        → log_request(request, results)
    except Exception as err:
        print('***** Logging failed with this error:',
              str(err))
    return render_template('results.html',
                          the_title=title,
                          the_phrase=phrase,
                          the_letters=letters,
                          the_results=results,)
```

Here's how you currently invoke "log_request".



We're keeping the "try" statement (for now).



We assumed that you had already added `from threading import Thread` to the top of your webapp's code.

In the space provided below, you were to write the code you'd insert into `do_search` instead of the standard call to `log_request`.

You were to use a `Thread` object to run `log_request`, just like we did with the recent `execute_slowly` example.

try:

t = Thread(target=log_request, args=(request, results))

t.start()

except ...

The "except" suite is unchanged, so we aren't showing it here.



Just like the earlier example, identify the target function to run, supply any arguments it needs, and don't forget to schedule your thread to run.



Test Drive

With these edits applied to `vsearch4web.py`, you are ready for another test run. What you're expecting to see here is next-to-no wait when you enter a search into your webapp's search page (as the `log_request` code is being run concurrently by the `threading` module).

Go ahead and give it a go.

Sure enough, the instant you click on the "Do it!" button, your webapp returns with your results. The assumption is that the `threading` module is now executing `log_request`, and waiting however long it takes to run that function's code to completion (approximately 15 seconds).

You're just about to give yourself a pat on the back (for a job well done) when, out of nowhere and after about 15 seconds, your webapp's terminal window erupts with error messages, not unlike these:

Take
a look
at this
message.

```
The last request was a success. →
...
127.0.0.1 - - [29/Jul/2016 19:43:31] "POST /search4 HTTP/1.1" 200 -
Exception in thread Thread-6:
Traceback (most recent call last):
  File "vsearch4web.not.slow.with.threads.but.broken.py", line 42, in log_request
    cursor.execute(_SQL, (req.form['phrase'],
  File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/site-packages/
werkzeug/local.py", line 343, in __getattr__
  ...
    raise RuntimeError(_request_ctx_err_msg)
RuntimeError: Working outside of request context. ← Whoops! An uncaught exception.
Lots (!!)  
more  
traceback  
messages  
here
This typically means that you attempted to use functionality that needed
an active HTTP request. Consult the documentation on testing for
information about how to avoid this problem.

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/3.5/lib/python3.5/threading.py",
line 914, in _bootstrap_inner
    self.run()
  ...
RuntimeError: Working outside of request context. ← And another one...yikes!
This typically means that you attempted to use functionality that needed
an active HTTP request. Consult the documentation on testing for
information about how to avoid this problem.
```

If you check your backend database, you'll learn that the details of your web request were **not** logged. Based on the messages above, it appears the `threading` module isn't at all happy with your code. A lot of the second group of traceback messages refer to `threading.py`, whereas the first group of traceback messages refer to code in the `werkzeug` and `flask` folders. What's clear is that adding in the `threading` code has resulted in a **huge mess**. What's going on?

what's going on?

First Things First: Don't Panic

Your first instinct may be to back out the code you added to run `log_request` in its own thread (and get yourself back to a known good state). But let's not panic, and let's **not** do that. Instead, let's take a look at that descriptive paragraph that appeared twice in the traceback messages:

```
...
This typically means that you attempted to use functionality that needed
an active HTTP request. Consult the documentation on testing for
information about how to avoid this problem.
...
```

This message is coming from Flask, not from the `threading` module. We know this because the `threading` module couldn't care less about what you use it for, and definitely has no interest in what you're trying to do with HTTP.

Let's take another look at the code that schedules the thread for execution, which we know takes 15 seconds to run, as that's how long `log_request` takes. While you're looking at this code, think about what happens during that 15 seconds:

```
@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))
    try:
        t = Thread(target=log_request, args=(request, results))
        t.start()
    except Exception as err:
        print('***** Logging failed with this error:', str(err))
    return render_template('results.html',
                          the_title=title,
                          the_phrase=phrase,
                          the_letters=letters,
                          the_results=results,)
```

What happens while this thread takes 15 seconds to execute?

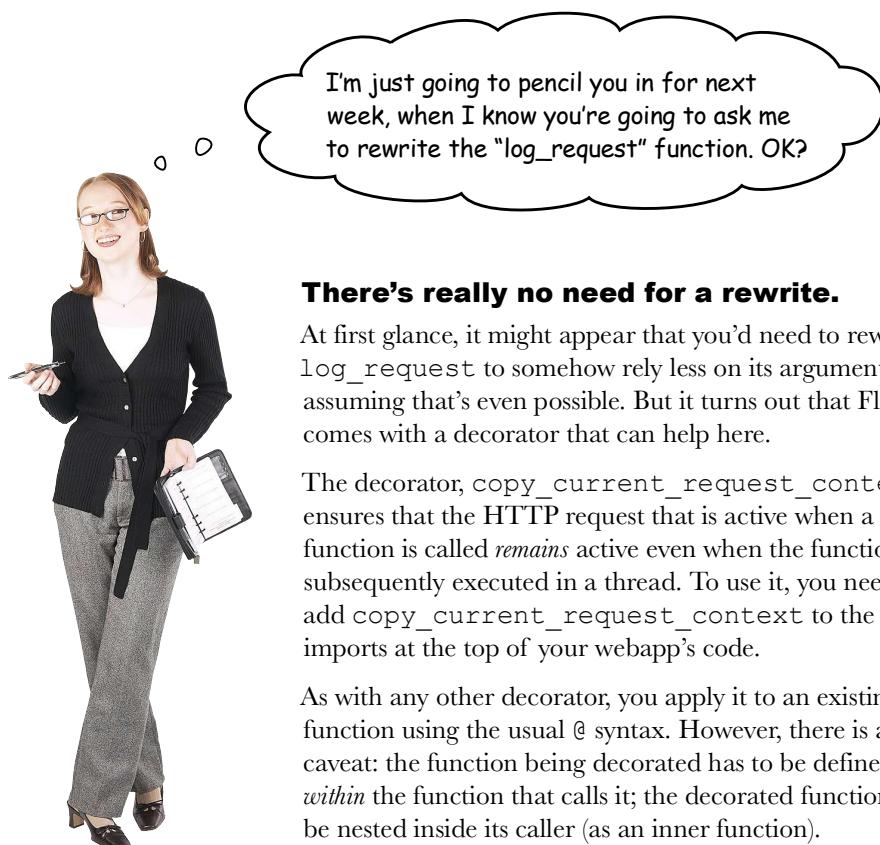
The instant the thread is scheduled for execution, the calling code (the `do_search` function) continues to execute. The `render_template` function executes (in the blink of an eye), and then the `do_search` function *ends*.

When `do_search` ends, all of the data associated with the function (its *context*) is reclaimed by the interpreter. The variables `request`, `phrase`, `letters`, `title`, and `results` cease to be. However, the `request` and `results` variables are passed as arguments to `log_request`, which tries to access them 15 seconds later. Unfortunately, at that point in time, the variables no longer exist, as `do_search` has ended. Bummer.

Don't Get Bummed Out: Flask Can Help

Based on what you've just learned, it appears the `log_request` function (when executed within a thread) can no longer "see" its argument data. This is due to the fact that the interpreter has long since cleaned up after itself, and reclaimed the memory used by these variables (as `do_search` has ended). Specifically, the `request` object is no longer active, and when `log_request` goes looking for it, it can't be found.

So, what can be done? Don't fret: help is at hand.



There's really no need for a rewrite.

At first glance, it might appear that you'd need to rewrite `log_request` to somehow rely less on its arguments... assuming that's even possible. But it turns out that Flask comes with a decorator that can help here.

The decorator, `copy_current_request_context`, ensures that the HTTP request that is active when a function is called *remains* active even when the function is subsequently executed in a thread. To use it, you need to add `copy_current_request_context` to the list of imports at the top of your webapp's code.

As with any other decorator, you apply it to an existing function using the usual `@` syntax. However, there is a caveat: the function being decorated has to be defined *within* the function that calls it; the decorated function must be nested inside its caller (as an inner function).



Here's what we want you to do (after updating the list of imports from Flask):

1. Take the `log_request` function and nest it inside the `do_search` function.
2. Decorate `log_request` with `@copy_current_request_context`.
3. Confirm that the runtime errors from the last *Test Drive* have gone away.



We asked you to do three things:

1. Take the `log_request` function and nest it inside the `do_search` function.
2. Decorate `log_request` with `@copy_current_request_context`.
3. Confirm that the runtime errors from the last *Test Drive* have gone away.

Here's what our `do_search` code looks like after we perform tasks 1 and 2 (note: we'll discuss task 3 over the page):

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    @copy_current_request_context
    def log_request(req: 'flask_request', res: str) -> None:
        sleep(15) # This makes log_request really slow...
        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, ))
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    try:
        t = Thread(target=log_request, args=(request, results))
        t.start()
    except Exception as err:
        print('***** Logging failed with this error:', str(err))
    return render_template('results.html',
                          the_title=title,
                          the_phrase=phrase,
                          the_letters=letters,
                          the_results=results,)
```

Task 1. The "log_request" function is now defined (nested) inside the "do_search" function.

All of the rest of this code remains unchanged.

Task 2. The decorator has been applied to "log_request".

there are no
Dumb Questions

Q: Does it still make sense to protect the threaded invocation of `log_request` with `try/except`?

A: Not if you are hoping to react to a runtime issue with `log_request`, as the `try/except` will have ended before the thread starts. However, your system may fail trying to create a new thread, so we figure it can't hurt to leave `try/except` in `do_search`.

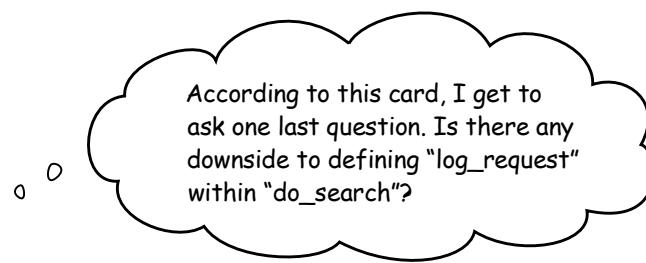


Test Drive

Task 3: Taking this latest version of `vsearch4web.py` for a spin confirms that the runtime errors from the last *Test Drive* are a thing of the past. Your webapp's terminal window confirms that all is well:

```
...  
127.0.0.1 - - [30/Jul/2016 20:42:46] "GET / HTTP/1.1" 200 -  
127.0.0.1 - - [30/Jul/2016 20:43:10] "POST /search4 HTTP/1.1" 200 -  
127.0.0.1 - - [30/Jul/2016 20:43:14] "GET /login HTTP/1.1" 200 -  
127.0.0.1 - - [30/Jul/2016 20:43:17] "GET /viewlog HTTP/1.1" 200 -  
127.0.0.1 - - [30/Jul/2016 20:43:37] "GET /viewlog HTTP/1.1" 200 -
```

No more scary runtime exceptions. All those 200s mean all is well with your webapp. And, 15 seconds after you submit a new search, your webapp eventually logs the details to your backend database WITHOUT requiring your webapp user to wait. ☺



No. Not in this case.

For this webapp, the `log_request` function was only ever called by `do_search`, so nesting `log_request`'s within `do_search` isn't an issue.

If you later decide to invoke `log_request` from some other function, you may have an issue (and you'll have to rethink things). But, for now, you're golden.

one last review

Is Your Webapp Robust Now?

Here are the four questions posed at the start of Chapter 11:

- 1 **What happens if the database connection fails?**
- 2 **Is our webapp protected from web attacks?**
- 3 **What happens if something takes a long time?**
- 4 **What happens if a function call fails?**

Your webapp now handles a number of runtime exceptions, thanks to your use of `try/except` and some custom exceptions that you can `raise` and catch as required.

When you know something can go wrong at runtime, fortify your code against any exceptions that might occur. This improves the overall robustness of your application, which is a good thing.

Note that there are other areas where robustness could be improved. You spent a lot of time adding `try/except` code to `view_the_log`'s code, which took advantage of the `UseDatabase` context manager. `UseDatabase` is *also* used within `log_request`, and should probably be protected, too (and doing so is left as a homework exercise for you).

Your webapp is more responsive due to your use of threading to handle a task that has to be performed eventually, but not right away. This is a good design strategy, although you do need to be careful not to go overboard with threads: the threading example in this chapter is very straightforward. However, it is very easy to create threading code that nobody can understand, and which will drive you mad when you have to debug it. **Use threads with care.**



In answering question 3—*what happens if something takes a long time?*—the use of threads improved the performance of the database write, but not the database read. It is a case of just having to wait for the data to arrive after the read, no matter how long it takes, as the webapp wasn't able to proceed without the data.

To make the database read go faster (assuming it's actually slow in the first place), you may have to look at using an alternative (faster) database setup. But that's a worry for another day that we won't concern ourselves with further in this book.

However, having said that, in the next and last chapter, we do indeed consider performance, but we'll be doing so while discussing a topic everyone understands, and which we've already discussed in this book: looping.

Chapter 11¾'s Code, 1 of 2

This is the latest and greatest
version of "vsearch4web.py".

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

from DBcm import UseDatabase, ConnectionError, CredentialsError, SQLSError
from checker import check_logged_in

from threading import Thread
from time import sleep

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.'

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':

    @copy_current_request_context
    def log_request(req: 'flask_request', res: str) -> None:
        sleep(15) # This makes log_request really slow...
        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,))

    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
```

The rest of "do_search" is
at the top of the next page. →

Chapter 11¾'s Code, 2 of 2

```
results = str(search4letters(phrase, letters))
try:
    t = Thread(target=log_request, args=(request, results))
    t.start()
except Exception as err:
    print('***** Logging failed with this error:', str(err))
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':
    try:
        with UseDatabase(app.config['dbconfig']) as cursor:
            _SQL = """select phrase, letters, ip, browser_string, results
from log"""
            cursor.execute(_SQL)
            contents = cursor.fetchall()
        # raise Exception("Some unknown exception.")
        titles = ('Phrase', 'Letters', 'Remote_addr', 'User_agent', 'Results')
        return render_template('viewlog.html',
                              the_title='View Log',
                              the_row_titles=titles,
                              the_data=contents,)
    except ConnectionError as err:
        print('Is your database switched on? Error:', str(err))
    except CredentialsError as err:
        print('User-id/Password issues. Error:', str(err))
    except SQLError as err:
        print('Is your query correct? Error:', str(err))
    except Exception as err:
        print('Something went wrong:', str(err))
    return 'Error'

app.secret_key = 'YouWillNeverGuessMySecretKey'

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

This is the rest of the "do_search" function.

12 advanced iteration



* Looping Like Crazy *

I've just had the most wonderful idea: what if I could make my loops go faster?



It's often amazing how much time our programs spend in loops.

This isn't a surprise, as most programs exist to perform something quickly a whole heap of times. When it comes to optimizing loops, there are two approaches: (1) improve the loop syntax (to make it easier to specify a loop), and (2) improve how loops execute (to make them go faster). Early in the lifetime of Python 2 (that is, a *long, long* time ago), the language designers added a single language feature that implements both approaches, and it goes by a rather strange name: **comprehension**. But don't let the strange name put you off: by the time you've worked through this chapter, you'll be wondering how you managed to live without comprehensions for so long.

Bahamas Buzzers Have Places to Go

To learn what loop comprehensions can do for you, you're going to take a look at some "real" data.

Operating out of Nassau on New Providence Island, *Bahamas Buzzers* provides island-hopping flights to some of the larger island airports. The airline has pioneered just-in-time flight scheduling: based on the previous day's demand, the airline predicts (which is just a fancy term for "guesses") how many flights they need the next day. At the end of each day, the *BB Head Office* generates the next day's flight schedule, which ends up in a text-based CSV (*comma-separated value*) file.

Here's what tomorrow's CSV file contains:

This is a standard CSV file, with the first line given over to header information. It all looks OK except for the fact that everything's **UPPERCASE** (which is a little "old school").

```
TIME,DESTINATION
09:35,FREEPOR
17:00,FREEPOR
09:55,WEST END
19:00,WEST END
10:45,TREASURE CAY
12:00,TREASURE CAY
11:45,ROCK SOUND
17:55,ROCK SOUND
```

We've places to go, people to see...



The header tells us to expect two columns of data: one representing times, the other destinations.

The rest of the CSV file contains the actual flight data.

Head Office calls this CSV file `buzzers.csv`.

If you were asked to read the data from the CSV file and display it on screen, you'd use a `with` statement. Here's what we did at IDLE's `>>>` prompt, after using Python's `os` module to change into the folder that contains the file:

Set this to the folder you're using.

```
>>> import os
>>> os.chdir('/Users/paul/buzzdata')
>>>
>>> with open('buzzers.csv') as raw_data:
    print(raw_data.read())
TIME,DESTINATION
09:35,FREEPOR
17:00,FREEPOR
09:55,WEST END
19:00,WEST END
10:45,TREASURE CAY
12:00,TREASURE CAY
11:45,ROCK SOUND
17:55,ROCK SOUND
>>> |
```

Ln: 59 Col: 4

The "read" method slurps up all of the characters in the file in one go.



Geek Bits

Learn more about the CSV format here: https://en.wikipedia.org/wiki/Comma-separated_values.

Reading CSV Data As Lists

The CSV data, in its raw form, is not very useful. It would be more useful if you could read and break apart each line at the comma, making it easier to get at the data.

Although it is possible to do this “breaking apart” with hand-crafted Python code (taking advantage of the string object’s `split` method), working with CSV data is such a common activity that the *standard library* comes with a module named `csv` that can help.

Here’s another small `for` loop that demonstrates the `csv` module in action.

Unlike the last example, where you used the `read` method to grab the entire contents of the file *in one go*, in the code that follows, `csv.reader` is used to read the CSV file *one line at a time* within the `for` loop. On each iteration, the `for` loop assigns each line of CSV data to a variable (called `line`), which is then displayed on screen:

This looks better:
each line of data
from the CSV
file has been
turned into a list.

The screenshot shows a Python 3.5.2 Shell window. The code reads a CSV file named 'buzzers.csv' using a 'with' statement to handle the file. It then iterates over the file using `csv.reader` to read each line. The output is a list of lists, where each inner list contains two items: a time and a destination. A callout points to the 'with' statement with the annotation 'Open the file using "with"...'. Another callout points to the `csv.reader` call with the annotation '...then read the data a line at a time with "csv.reader".' The output text is as follows:

```

Python 3.5.2 Shell
>>>
>>> import csv
>>>
>>> with open('buzzers.csv') as data:
    for line in csv.reader(data):
        print(line)

['TIME', 'DESTINATION']
['09:35', 'FREEPORT']
['17:00', 'FREEPORT']
['09:55', 'WEST END']
['19:00', 'WEST END']
['10:45', 'TREASURE CAY']
['12:00', 'TREASURE CAY']
['11:45', 'ROCK SOUND']
['17:55', 'ROCK SOUND']
>>>
>>> |

```

Ln: 77 Col: 4

The `csv` module is doing quite a bit of work here. Each line of raw data is being read from the file, then “magically” turned into a two-item list.

In addition to the header information (from the first line of the file) being returned as a list, each individual flight time and destination pair also gets its own list. Take note of the *type* of the individual data items returned: everything is a string, even though the first item in each list (clearly) represents a time.

The `csv` module has a few more tricks up its sleeve. Another interesting function is `csv.DictReader`. Let’s see what that does for you.

Reading CSV Data As Dictionaries

Here's code similar to the last example, but for the fact that this new code uses `csv.DictReader` as opposed to `csv.reader`. When `DictReader` is used, the data from the CSV file is returned as a collection of dictionaries, with the keys for each dictionary taken from the CSV file's header line, and the values taken from each of the subsequent lines. Here's the code:

The screenshot shows the Python 3.5.2 Shell window. The code reads a CSV file named 'buzzers.csv' and prints each row as a dictionary. The output is a list of dictionaries, where each dictionary has 'DESTINATION' as the key and 'TIME' as the value. Arrows point from the text 'The keys' to the header 'DESTINATION' and from 'The values' to the 'TIME' key in the dictionaries. A callout bubble points to the right, stating: 'Using "csv.DictReader" is a simple change, but it makes a big difference. What was lines of lists (last time) are now lines of dictionaries.' To the right of the shell window, the raw CSV data is shown in a table format.

	TIME, DESTINATION
0	09:35, FREEPORT
1	17:00, FREEPORT
2	09:55, WEST END
3	19:00, WEST END
4	10:45, TREASURE CAY
5	12:00, TREASURE CAY
6	11:45, ROCK SOUND
7	17:55, ROCK SOUND

Recall: the raw data in the file looks like this.

There is no doubt that this is powerful stuff: with a single call to `DictReader`, the `csv` module has transformed the raw data in your CSV file into a collection of Python dictionaries.

But imagine you've been tasked with converting the raw data in the CSV file based on the following requirements:

- 1 Convert the flight times from 24-hour format to AM/PM format**
- 2 Convert the destinations from UPPERCASE to Titlecase**

In and of themselves, these are not difficult tasks. However, when you consider the raw data as a collection of lists or a collection of dictionaries, they can be. So, let's write a custom `for` loop to read the data into a single dictionary that can then be used to perform these conversions with a lot less fuss.

Let's Back Up a Little Bit

Rather than use `csv.reader` or `csv.DictReader`, let's roll our own code to convert the raw data in the CSV file into a *single* dictionary, which we can then manipulate to perform the required conversions.

We've had a chat with the *Head Office* folks over at *Bahamas Buzzers*, and they've told us they're very happy with the conversions we have in mind, but would still like the data kept in its "raw form," as that's how their antiquated departures board expects its data to arrive: 24-hour format for flight times, and all **UPPERCASE** for destinations.

You could perform conversions on the raw data in your single dictionary, but let's ensure that the conversions are performed on *copies* of the data, not the actual raw data as read in. Although it's not totally clear at the moment, the noises coming out of *Head Office* seem to indicate that whatever code you create may have to interface with some existing systems. So, rather than face the prospect of converting the data back into its raw form, let's read it into a single dictionary as is, then convert to copies as required (while leaving the raw data in the original dictionary *untouched*).

It's not an awful lot of work (over and above what you had to do with the `csv` module) to read the raw data into a dictionary. In the code below, the file is opened, and the first line is read and ignored (as we don't need the header info). A `for` loop then reads each line of raw data, splitting it in two at the comma, with the flight time being used as your dictionary *key*, and the destination used as your dictionary *value*.

The raw data

```
TIME, DESTINATION
09:35, FREEPORT
17:00, FREEPORT
09:55, WEST END
19:00, WEST END
10:45, TREASURE CAY
12:00, TREASURE CAY
11:45, ROCK SOUND
17:55, ROCK SOUND
```

Can you break each line in two, using the comma as the delimiter?

```

Python 3.5.2 Shell
>>> with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.split(',')
        flights[k] = v
>>> flights
{'12:00': 'TREASURE CAY\n', '09:35': 'FREEPORT\n', '17:00': 'FREEPORT\n', '19:00': 'WEST END\n', '17:55': 'ROCK SOUND\n', '10:45': 'TREASURE CAY\n', '09:55': 'WEST END\n', '11:45': 'ROCK SOUND\n'}
>>> import pprint
>>> pprint.pprint(flights)
{'09:35': 'FREEPORT\n',
 '09:55': 'WEST END\n',
 '10:45': 'TREASURE CAY\n',
 '11:45': 'ROCK SOUND\n',
 '12:00': 'TREASURE CAY\n',
 '17:00': 'FREEPORT\n',
 '17:55': 'ROCK SOUND\n',
 '19:00': 'WEST END\n'}
>>>

```

Annotations:

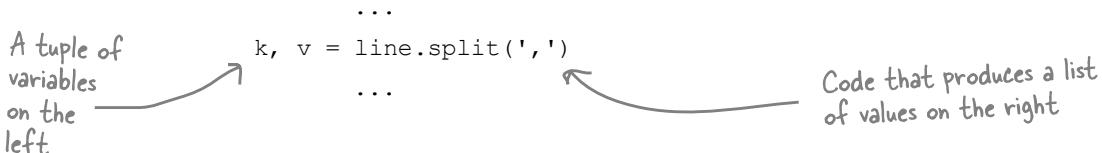
- Open the file as before.
- Create a new, empty dictionary called "flights".
- Display the contents of the dictionary, which looks a little messed up until...
- ...the "pretty-printing" library produces more human-friendly output.
- Ignore the header info.
- Process each line.
- Assign destination to flight time.
- The inclusion of the newline character looks a little strange, doesn't it?

Break apart the line at the comma, which returns two values: the key (flight time) and value (destination).

no to newlines

Stripping, Then Splitting, Your Raw Data

The latest `with` statement used the `split` method (included with all string objects) to break the line of raw data in two. What's returned is a list of strings, which the code individually assigns to the `k` and `v` variables. This multivariable assignment is possible due to the fact that you have a tuple of variables on the left of the assignment operator, as well as code that produces a list of values on the right of the operator (remember: tuples are *immutable* lists):



Another string method, `strip`, removes whitespace from the beginning and end of an existing string. Let's use it to remove the unwanted trailing newline from the raw data *before* performing the `split`.

Here's one final version of our data-reading code. We create a dictionary called `flights`, which uses the flight times as keys and the destinations (without the newline) as values:

```
Python 3.5.2 Shell
>>>
>>> with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.strip().split(',')
        flights[k] = v
    >>> pprint.pprint(flights)
{'09:35': 'FREEPORT',
 '09:55': 'WEST END',
 '10:45': 'TREASURE CAY',
 '11:45': 'ROCK SOUND',
 '12:00': 'TREASURE CAY',
 '17:00': 'FREEPORT',
 '17:55': 'ROCK SOUND',
 '19:00': 'WEST END'}
>>>
>>> |
```

This code strips the line, then splits it, to produce the data in the format required.



Whitespace: the following characters are considered whitespace in strings: `space`, `\t`, `\n`, and `\r`.

You may not have spotted this, but the order of the rows in the dictionary different from what's in the data file. This happens because dictionaries do NOT maintain insertion order. Don't worry about this for now.

TIME, DESTINATION
09:35, FREEPORT
17:00, FREEPORT
09:55, WEST END
19:00, WEST END
10:45, TREASURE CAY
12:00, TREASURE CAY
11:45, ROCK SOUND
17:55, ROCK SOUND

What if you switched the order of the methods in your code, like so:

`line.split(',') .strip()`

What do you think would happen?

When you string methods together like this, it's called a "method chain."

Be Careful When Chaining Method Calls

Some programmers don't like the fact that Python's method calls can be chained together (as `strip` and `split` were in the last example), because such chains can be hard to read the first time you see them. However, method chaining is popular among Python programmers, so you'll likely run across code that uses this technique "in the wild." Care is needed, however, as the order of the method calls is *not* interchangeable.

As an example of what can go wrong, consider this code (which is very similar to what came before). Whereas before the order was `strip`, then `split`, this code calls `split` first, then tries to call `strip`. Look what happens:

```

Python 3.5.2 Shell
>>>
>>>
>>> with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.split(',').strip()
        flights[k] = v

Traceback (most recent call last):
  File "<pyshell#194>", line 5, in <module>
    k, v = line.split(',').strip()
AttributeError: 'list' object has no attribute 'strip'
>>>
>>>

```

The interpreter is not happy, and crashes with an "AttributeError".

The order of this method chain has flipped from what you used before.

To understand what's going on here, consider the *type* of the data to the right of the assignment operator as the above method chain executes.

Before anything happens, `line` is a string. Calling `split` on a string returns a list of strings, using the argument to `split` as a delimiter. What started out as a *string* (`line`) has dynamically morphed into a *list*, which then has another method invoked against it. In this example, the next method is `strip`, which expects to be invoked on a *string*, *not* a *list*, so the interpreter raises an `AttributeError`, as lists don't have a method called `strip`.



The method chain from the previous page does not suffer from this issue:

```

...
line.strip().split(',')
...

```

With this code, the interpreter starts out with a string (in `line`), which has any leading/trailing whitespace removed by `strip` (yielding another string), which is then `split` into a list of strings based on the comma delimiter. There's no `AttributeError`, as the method chain doesn't violate any typing rules.

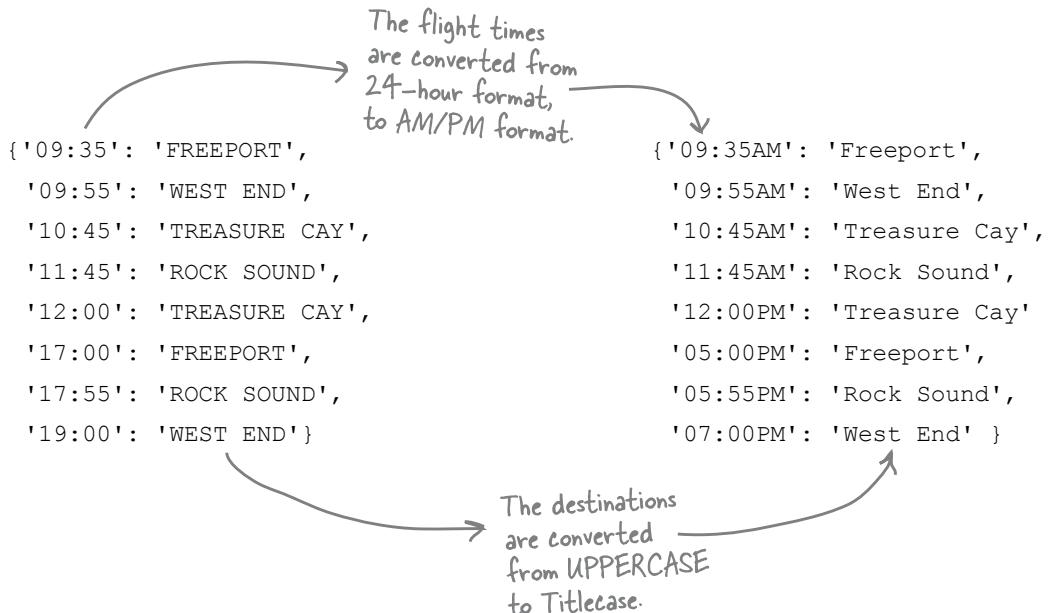
Transforming Data into the Format You Need

Now that the data is in the `flights` dictionary, let's consider the data manipulations *BB Head Office* has asked you to perform.

The first is to perform the two conversions identified earlier in this chapter, creating a new dictionary in the process:

- ➊ Convert the flight times from 24-hour format to AM/PM format
- ➋ Convert the destinations from UPPERCASE to Titlecase

Applying these two transformations to the `flights` dictionary allows you to turn the dictionary on the left into the one on the right:

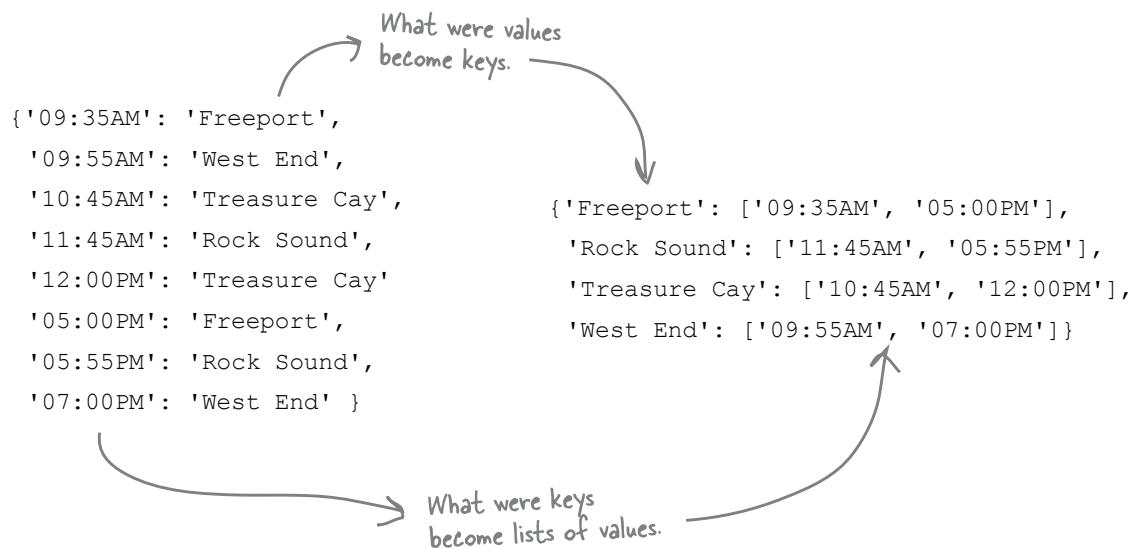


Note that the data in both dictionaries has the same meaning, it's just the representation that's changed. *Head Office* needs the second dictionary, as they feel that its data is more universally understandable, as well as friendlier; *Head Office* thinks all-UPPERCASE is akin to shouting.

At the moment, the data in both dictionaries has a single line for each flight time/destination combination. Although *Head Office* will be happy when you've transformed the dictionary on the left into the dictionary on the right, they've also suggested that it would be really useful if the data could be presented with single destinations as keys and a list of flight times as values—that is, a single row of data for each destination. Let's look at how *that* dictionary would appear before embarking on coding the required manipulations.

Transforming into a Dictionary Of Lists

Once the data in `flights` has been transformed, *Head Office* wants you to perform this second manipulation (discussed at the bottom of the last page):



Think about the data wrangling that's needed here...

There's a bit of work required to get from the raw data in the CSV file to the dictionary of lists shown above on the right. Take a moment to think about how you'd go about doing this using the Python you already know.

If you're like most programmers, it won't take you long to work out that the `for` loop is your friend here. As Python's main looping mechanism, `for` has already helped you extract the raw data from the CSV file and populate the `flights` dictionary:

This is a classic use of
"for", and a hugely popular
programming idiom in Python.

```

with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.strip().split(',')
        flights[k] = v
  
```

It's tempting to suggest that this code be amended to perform the transformations to the raw data as it is read from the CSV file—that is, *prior* to adding rows of data to `flights`. But recall the *Head Office*'s request that the raw data remain untouched in `flights`: any transformations need to be applied to a **copy** of the data. This makes things more complex, but not by much.

basic converts first

Let's Do the Basic Conversions

At the moment, the `flights` dictionary contains flight times in 24-hour format as its keys, with UPPERCASE strings representing destinations as its values. You have two initial conversions to perform:

- ➊ Convert the flight times from 24-hour format to AM/PM format
- ➋ Convert the destinations from UPPERCASE to Titlecase

Conversion #2 is easy, so let's do that one first. Once data is in a string, simply call the string's `title` method, as this IDLE session demonstrates:

```
>>> s = "I DID NOT MEAN TO SHOUT."
>>> print(s)
I DID NOT MEAN TO SHOUT.
>>> t = s.title()
>>> print(t)
I Did Not Mean To Shout.
```

The “title”
method returns
a copy of the
data in “s”. →

This is much friendlier
than before.

Conversion #1 involves a bit more work.

If you think about it for a minute, things get quite involved when it comes to converting 19:00 into 7:00PM. However, this is only the case when you look at the 19:00 data as a string. You'd need to write a lot of code to do the conversion.

If you instead consider that 19:00 is a time, you can take advantage of the `datetime` module that is included as part of Python's *standard library*. This module's `datetime` class can take a string (like 19:00) and convert it to its equivalent AM/PM format using two prebuilt functions and what's known as *string format specifiers*. Here's a small function, called `convert2ampm`, which uses the facilities of the `datetime` module to perform the conversion you need:

For more on string format specifiers,
see <https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior>.



**Ready Bake
Code**

```
from datetime import datetime

def convert2ampm(time24: str) -> str:
    return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')
```

Given a time in 24-hour format (as a string), this method chain converts it into a string in AM/PM format.



Sharpen your pencil

Let's put the conversion techniques from the last page to work.

Below is the code that reads the raw data from the CSV file, populating the `flights` dictionary as it goes. The `convert2ampm` function is also shown.

Your job is to write a `for` loop that takes the data in `flights` and converts the keys to AM/PM format, and the values to *Titlecase*. A new dictionary, called `flights2`, is created to hold the converted data. Use your pencil to add the `for` loop code in the space provided.

Hint: when processing a dictionary with a `for` loop, recall that the `items` method returns the key and value for each row (as a tuple) on each iteration.

Define the conversion function.

```
from datetime import datetime
import pprint

def convert2ampm(time24: str) -> str:
    return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')
```

Grab the data from the file.

```
with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.strip().split(',')
        flights[k] = v
    pprint.pprint(flights)
    print()

flights2 = {} ← The new dictionary, called "flights2", starts out empty.
```

Pretty-print the "flights" dictionary prior to performing the conversions.

Add your "for" loop here. →

.....

```
pprint.pprint(flights2)
```

Pretty-print the "flights2" dictionary to confirm that the conversions are working.

convert then run



Sharpen your pencil Solution

We saved all
of this code
in a file called
"do_convert.py".

```
from datetime import datetime
import pprint

def convert2ampm(time24: str) -> str:
    return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')

with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.strip().split(',')
        flights[k] = v

    pprint.pprint(flights)
    print()

    flights2 = {}
    for k, v in flights.items():
        flights2[convert2ampm(k)] = v.title()
    pprint.pprint(flights2)
```

The "items"
method returns
each row from
the "flights"
dictionary.

On each iteration, the
key (in "k") is converted
to AM/PM format,
then used as the new
dictionary's key.

The value (in "v") is
converted to Titlecase,
then assigned to the
converted key.



Test DRIVE

If you execute the above program, two dictionaries are displayed on screen (which we're showing below, side by side). The conversions work, although the ordering in each dictionary differs, as the interpreter does **not** maintain *insertion order* when you populate a new dictionary with data:

This is
"flights".

```
{'09:35': 'FREEPORT',
'09:55': 'WEST END',
'10:45': 'TREASURE CAY',
'11:45': 'ROCK SOUND',
'12:00': 'TREASURE CAY',
'17:00': 'FREEPORT',
'17:55': 'ROCK SOUND',
'19:00': 'WEST END'}
```

```
{'05:00PM': 'Freeport',
'05:55PM': 'Rock Sound',
'07:00PM': 'West End',
'09:35AM': 'Freeport',
'09:55AM': 'West End',
'10:45AM': 'Treasure Cay',
'11:45AM': 'Rock Sound',
'12:00PM': 'Treasure Cay'}
```

This is
"flights2".

The raw data is
transformed.

Did You Spot the Pattern in Your Code?

Take another look at the program you've just executed. There's a very common programming pattern used *twice* in this code. Can you spot it?

```
from datetime import datetime
import pprint

def convert2ampm(time24: str) -> str:
    return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')

with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.strip().split(',')
        flights[k] = v

pprint.pprint(flights)
print()

flights2 = {}
for k, v in flights.items():
    flights2[convert2ampm(k)] = v.title()

pprint.pprint(flights2)
```

If you answered: “the `for` loop,” you’re only half-right. The `for` loop *is* part of the pattern, but take another look at the code that *surrounds* it. Spot anything else?

```
from datetime import datetime
import pprint

def convert2ampm(time24: str) -> str:
    return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')

with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.strip().split(',')
        flights[k] = v

    pprint.pprint(flights)
    print()

    flights2 = {}
    for k, v in flights.items():
        flights2[convert2ampm(k)] = v.title()

    pprint.pprint(flights2)
```

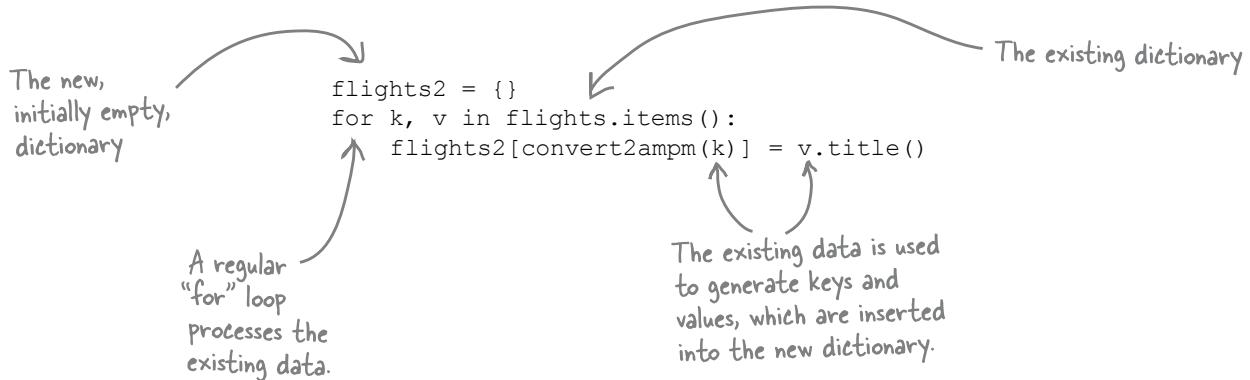
Each of the “`for`” loops is preceded by the creation of a new, empty data structure (e.g., a dictionary).

Each of the “`for`” loop’s suites contains code that adds data to the new data structure, based on the processing of some existing data.

spot the pattern

Spotting the Pattern with Lists

The examples on the last page highlighted the programming pattern as it relates to dictionaries: start with a new, empty dictionary, then use a `for` loop to process an existing dictionary, generating data for a new dictionary as you go:



This pattern also makes an appearance with lists, where it is easier to spot. Take a look at this IDLE session, where the keys (i.e., the flight times) and the values (i.e., the destinations) are extracted from the `flights` dictionary as lists, then converted into new lists using the programming pattern (numbered 1 through 4 in the annotations):

```
>>>
>>>
>>> flight_times = []
>>> for ft in flights.keys():
...     flight_times.append(convert2ampm(ft))
...
>>> print(flight_times)
['05:00PM', '09:55AM', '11:45AM', '10:45AM', '07:00PM', '05:55PM',
 '12:00PM', '09:35AM']
>>>
>>> destinations = []
>>> for dest in flights.values():
...     destinations.append(dest.title())
...
>>> print(destinations)
['Freeport', 'West End', 'Rock Sound', 'Treasure Cay', 'West End',
 'Rock Sound', 'Treasure Cay', 'Freeport']
>>>
>>> |
```

Annotations numbered 1 through 4 point to specific parts of the code:

1. Start with a new, empty list.
2. Iterate through each of the flight times.
3. Append the converted data to the new list.
4. View the new list's data.

This pattern is used so often that Python provides a convenient shorthand notation for it called the **comprehension**. Let's see what's involved in creating a comprehension.

Converting Patterns into Comprehensions

Let's take the most recent `for` loop that processed the destinations as our example. Here it is again:

```
destinations = []
for dest in flights.values():
    destinations.append(dest.title())

```

The diagram illustrates the conversion process with three numbered steps:

1. Start with a new, empty list.
2. Iterate through each of the destinations.
3. Append the converted data to the new list.

Python's built-in **comprehension** feature lets you rework the above three lines of code as a single line.

To convert the above three lines into a comprehension, we're going to step through the process, building up to the complete comprehension.

Begin by starting with a new, empty list, which is assigned to a new variable (which we're calling `more_dests` in this example):

```
more_dests = []
```

Step 1: Start with a new, empty list (and give it a name).

Specify how the existing data (in `flights` in this example) is to be iterated over using the familiar `for` notation, and place this code within the new list's square brackets (note the *absence* of the colon at the end of the `for` code):

```
more_dests = [for dest in flights.values()]
```

Step 2: Iterate through each of the destinations.

Note that there's NO colon here.

To complete the comprehension, specify the transformation to be applied to the data (in `dest`), and put this transformation *before* the `for` keyword (note the *absence* of the call to `append`, which is assumed by the comprehension):

```
more_dests = [dest.title() for dest in flights.values()]
```

Step 3: Append the converted data to the new list, without actually calling "append".

And that's it. The single line of code at the bottom of this page is functionally equivalent to the three lines of code at the top. Go ahead and run this line of code at your `>>>` prompt to convince yourself that the `more_dests` list contains the same data as the `destinations` list.

alternative to for

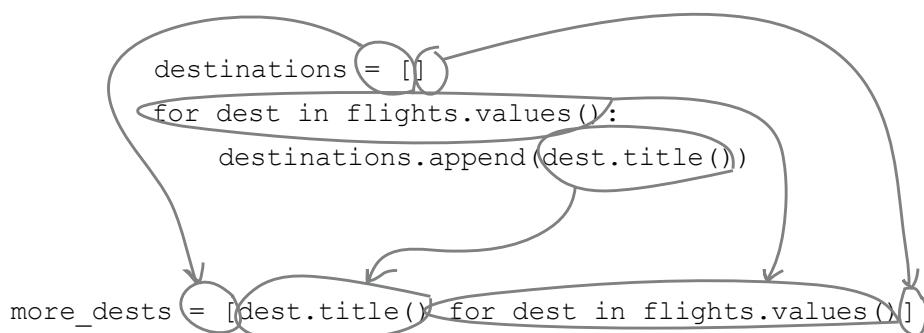
Take a Closer Look at the Comprehension

Let's look at the comprehension in a little more detail. Here's the original three lines of code as well as the single-line comprehension that performs the same task.

Remember: both versions produce new lists (`destinations` and `more_dests`) that have exactly the same data:

```
destinations = []
for dest in flights.values():
    destinations.append(dest.title())
 $\downarrow$ 
more_dests = [dest.title() for dest in flights.values()]
```

It's also possible to pick out the parts of the original three lines of code and see where they've been used in the comprehension:

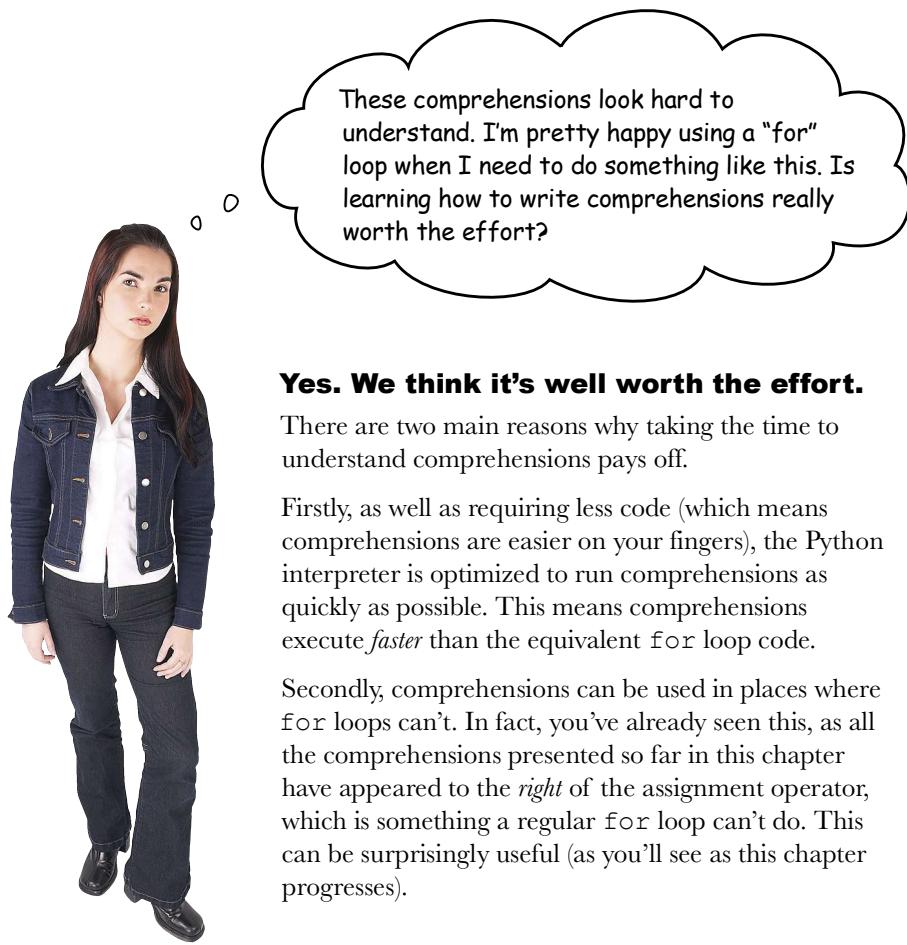


If you spot this pattern in other code, you can easily turn it into a comprehension. For example, here's some code from earlier (which produces the list of AM/PM flight times) reworked as a comprehension:

```
flight_times = []
for ft in flights.keys():
    flight_times.append(convert2ampm(ft))
 $\downarrow$ 
fts2 = [convert2ampm(ft) for ft in flights.keys()]
```

A handwritten note on the right side of the diagram says: "These do the *same thing*."

What's the Big Deal?



Yes. We think it's well worth the effort.

There are two main reasons why taking the time to understand comprehensions pays off.

Firstly, as well as requiring less code (which means comprehensions are easier on your fingers), the Python interpreter is optimized to run comprehensions as quickly as possible. This means comprehensions execute *faster* than the equivalent `for` loop code.

Secondly, comprehensions can be used in places where `for` loops can't. In fact, you've already seen this, as all the comprehensions presented so far in this chapter have appeared to the *right* of the assignment operator, which is something a regular `for` loop can't do. This can be surprisingly useful (as you'll see as this chapter progresses).

Comprehensions aren't just for lists

The comprehensions you've seen so far have created new lists, so each is known as a **list comprehension** (or *listcomp* for short). If your comprehension creates a new dictionary, it's known as a **dictionary comprehension** (*dictcomp*).

And, so as not to leave any data structure out, you can also specify a **set comprehension** (*setcomp*).

There's no such thing as a *tuple comprehension*; we'll explain why later in this chapter.

First, though, let's take a look at a dictionary comprehension.

dictionary comprehensions

Specifying a Dictionary Comprehension

Recall the code from earlier in this chapter that read the raw data from the CSV file into a dictionary called `flights`. This data was then transformed into a new dictionary called `flights2`, which is keyed by AM/PM flight times and uses “titlecased” destinations as values:

...

```
flights2 = {}
for k, v in flights.items():
    flights2[convert2ampm(k)] = v.title()
...
}
```

This code conforms to the “comprehension pattern.”

Let’s rework these three lines of code as a dictionary comprehension.

Start by assigning a new, empty dictionary to a variable (which we are calling `more_flights`):

```
more_flights = {}
```

1. Start with a new, empty dictionary.

Specify how the existing data (in `flights`) is to be iterated over using the `for` loop notation (being sure not to include the usual trailing colon):

```
more_flights = {for k, v in flights.items()}
```

2. Iterate through each of the keys and values from the existing data.

Note that there's NO colon here.

To complete the dictcomp, specify how the new dictionary’s keys and values relate to each other. The `for` loop at the top of the page produces the key by converting it to an AM/PM flight time using the `convert2ampm` function, while the associated value is turned into titlecase thanks to the string’s `title` method. An equivalent dictcomp can do the same thing and, as with listcomps, this relationship is specified to the *left* of the dictcomp’s `for` keyword. Note the inclusion of the colon separating the new key from the new value:

```
more_flights = {convert2ampm(k): v.title() for k, v in flights.items()}
```

3. Associate the converted key with its “titlecased” value (and note the use of the colon here).

And there it is: your first dictionary comprehension. Go ahead and take it for a spin to confirm that it works.

Extend Comprehensions with Filters

Let's imagine you need only the converted flight data for *Freeport*.

Reverting to the original `for` loop, you'd likely extend the code to include an `if` statement that filters based on the current value in `v` (the destination), producing code like this:

```
just_freeport = {}
for k, v in flights.items():
    if v == 'FREEPORT':
        just_freeport[convert2ampm(k)] = v.title()
```

The flight data is only converted and added to the "just_freeport" dictionary if it relates to the Freeport destination.

If you execute the above loop code at the `>>>` prompt, you'll end up with just two rows of data (representing the two scheduled flights to *Freeport* as contained in the raw data file). This shouldn't be surprising, as using an `if` in this way to filter data is a standard technique. It turns out that such filters can be used with comprehensions, too. Simply take the `if` statement (minus the colon) and tack it onto the end of your comprehension. Here's the dictcomp from the bottom of the last page:

```
more_flights = {convert2ampm(k): v.title() for k, v in flights.items()}
```

And here's a version of the same dictcomp with the filter added:

```
just_freeport2 = {convert2ampm(k): v.title() for k, v in flights.items() if v == 'FREEPORT'}
```

If you execute this filtered dictcomp at your `>>>` prompt, the data in the newly created `just_freeport2` dictionary is identical to the data in `just_freeport`. Both `just_freeport` and `just_freeport2`'s data is a **copy** of the original data in the `flights` dictionary.

Granted, the line of code that produces `just_freeport2` looks intimidating. Many programmers new to Python complain that comprehensions are **hard to read**. However, recall that Python's usual end-of-line-means-end-of-statement rule is switched off whenever code appears between a bracket pair, so you can rewrite any comprehension over multiple lines to make it easier to read, like so:

```
just_freeport3 = {convert2ampm(k): v.title()
                  for k, v in flights.items()
                  if v == 'FREEPORT'}
```

TIME, DESTINATION
09:35, FREEPORT
17:00, FREEPORT
09:55, WEST END
19:00, WEST END
10:45, TREASURE CAY
12:00, TREASURE CAY
11:45, ROCK SOUND
17:55, ROCK SOUND

The raw data

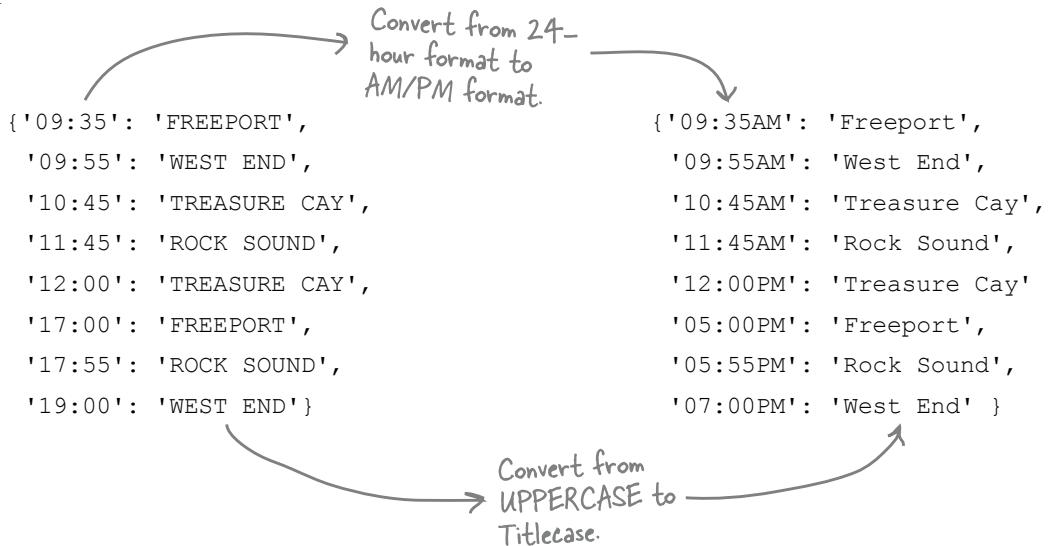
The flight data is only converted and added to the "just_freeport2" dictionary if it relates to the Freeport destination.

You'll need to get used to reading those one-line comprehensions. That said, Python programmers are increasingly writing longer comprehensions over multiple lines (so you'll see this syntax, too).

a quick review

Recall What You Set Out to Do

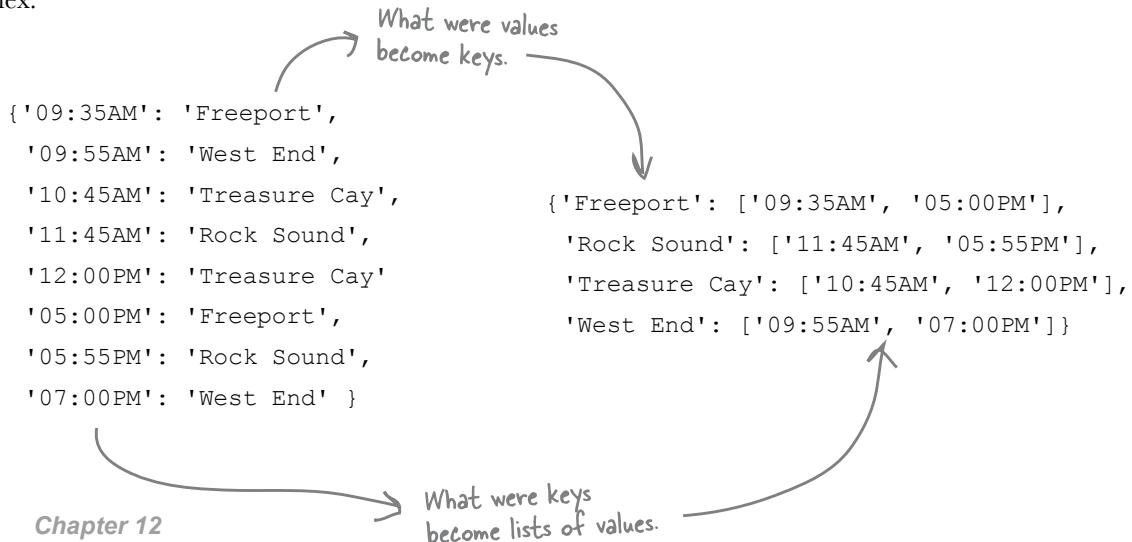
Now that you've seen what comprehensions can do for you, let's revisit the required dictionary manipulations from earlier in this chapter to see how we're doing. Here's the first requirement:



Given the data in the `flights` dictionary, you've seen that the following dictionary comprehension performs the above conversions *in one line of code*, assigning the copied data to a new dictionary called `fts` here:

```
fts = {convert2ampm(k): v.title() for k, v in flights.items()}
```

The second manipulation (listing flight times per destination) is a little more involved. There's a bit more work to do due to the fact that the data manipulations are more complex:





Sharpen your pencil

Before starting to work on the second manipulation, let's pause for a bit to see how well the comprehension material is seeping into your brain.

You've been tasked with transforming the three `for` loops on this page into comprehensions. As you do, don't forget to test your code in IDLE (before flipping the page and peeking at our solutions). In fact, before you try to write the comprehensions, execute these loops and see what they do. Write your comprehension solutions in the spaces provided.

```
1 data = [ 1, 2, 3, 4, 5, 6, 7, 8 ]
evens = []
for num in data:
    if not num % 2: ←
        evens.append(num)
```

The `%` operator is Python's modulo operator, which works as follows: given two numbers, divide the first by the second, then return the remainder.

```
2 data = [ 1, 'one', 2, 'two', 3, 'three', 4, 'four' ]
words = []
for num in data:
    if isinstance(num, str): ←
        words.append(num)
```

The "isinstance" BIF checks to see whether a variable refers to an object of a certain type.

```
3 data = list('So long and thanks for all the fish'.split())
title = []
for word in data:
    title.append(word.title())
```

comprehending the solutions



Sharpen your pencil Solution

You were to grab your pencil, and pop your thinking cap on. For each of these three `for` loops, you were tasked with transforming them into comprehensions, being sure to test your code in IDLE.

1 `data = [1, 2, 3, 4, 5, 6, 7, 8]`

`evens = []`

`for num in data:`

`if not num % 2:`

`evens.append(num)`

}

These four lines of loop code
(which populate "evens") become
one line of comprehension.

`evens = [num for num in data if not num % 2]`

2 `data = [1, 'one', 2, 'two', 3, 'three', 4, 'four']`

`words = []`

`for num in data:`

`if isinstance(num, str):`

`words.append(num)`

}

Again, this four-line loop
is reworked as a one-line comprehension.

`words = [num for num in data if isinstance(num, str)]`

3 `data = list('So long and thanks for all the fish'.split())`

`title = []`

`for word in data:`

`title.append(word.title())`

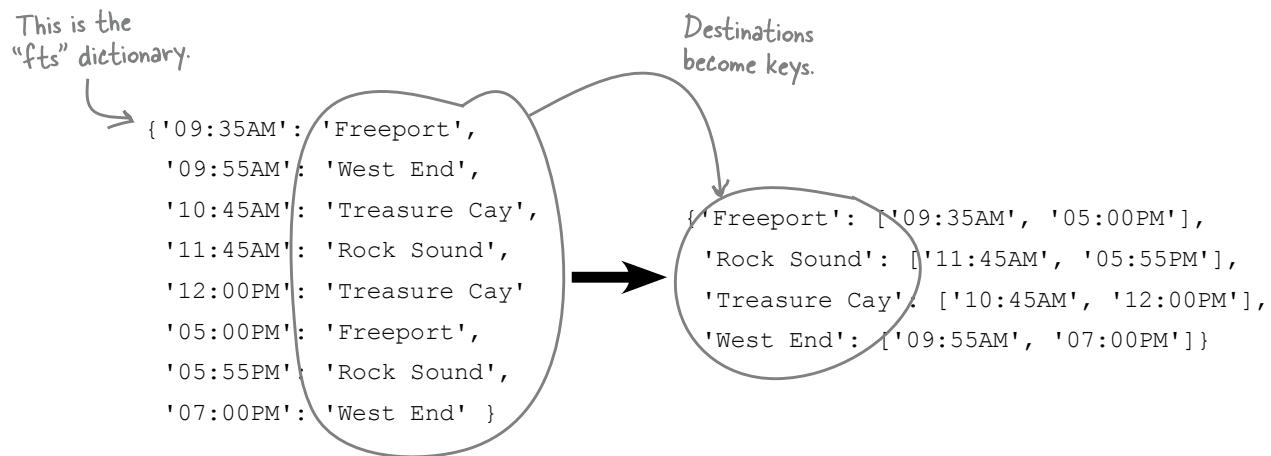
You should find this
one the easiest of the
three (as it contains no
filter).

`title = [word.title() for word in data]`

Deal with Complexity the Python Way

With your comprehension practice session behind you, let's experiment at the >>> prompt to work out what has to happen to the data in the `fts` dictionary in order to transform it into what's required.

Before writing any code, take another look at the required transformation. Notice how the keys in the new dictionary (on the right) are a list of unique destinations taken from the values in the `fts` dictionary (on the left):



It turns out that producing those four unique destinations is very straightforward.

Given that you have the data on the left in a dictionary called `fts`, you can access all of the values using `fts.values`, then feed that to the `set` BIF to remove duplicates. Let's store the unique destinations in a variable called `dests`:

```
>>> dests = set(fts.values())
>>> print(dests)
{'Freeport', 'West End', 'Rock Sound', 'Treasure Cay'}
```

Grab all of the values in "fts", then feed them to the "set" BIF. This gets you the data you need.

Here are the four unique destinations, which you can use as the new dictionary's keys.

Now that you have a way to get the unique destinations, it's time to grab the flight times associated with those destinations. This data is also in the `fts` dictionary.

Before turning the page, have a think about how you'd go about extracting the flight times given each unique destination.

In fact, don't worry about extracting all the flight times for *every* destination; just work out how to do it for *West End* first.

west end only

Extract a Single Destination's Flight Times

Let's start by extracting the flight time data for a single destination, namely *West End*. Here's the data you need to extract:

```
{'09:35AM': 'Freeport',
 '09:55AM': 'West End',
 '10:45AM': 'Treasure Cay',
 '11:45AM': 'Rock Sound',
 '12:00PM': 'Treasure Cay'
 '05:00PM': 'Freeport',
 '05:55PM': 'Rock Sound',
 '07:00PM': 'West End'}
```

You need to turn these keys in a list of values.

As before, pull up the >>> prompt and get to work. Given the `fts` dictionary, you can extract the *West End* flight times using code like this:

```
>>> wests = []
>>> for k, v in fts.items():
...     if v == 'West End':
...         wests.append(k)
...
>>> print(wests)
['09:55AM', '07:00PM']
```

1. Start with a new, empty list.

2. Extract the keys and values from the "fts" dictionary.

3. Filter the data on destination "West End".

4. Append the "West End" flight times to the "wests" list.

It worked! Here's the data you need.

On seeing this code, you should hear little alarm bells ringing in your brain, as this `for` loop is surely a candidate for reworking as a list comprehension, right?

That `for` loop becomes this equivalent listcomp:

What was four lines of code is now one, thanks to your use of a listcomp.

It also worked!
Here's the data you need.

```
>>> wests2 = [k for k, v in fts.items() if v == 'West End']

>>> print(wests2)
['09:55AM', '07:00PM']
```

Now that you know how to extract this data for one specific destination, let's do it for all the destinations.

Extract Flight Times for All Destinations

You now have this code, which extracts the set of unique destinations:

```
dests = set(fts.values())
```

And you also have this listcomp, which extracts the list of flight times for a given destination (in this example, that destination is *West End*):

```
wests2 = [k for k, v in fts.items() if v == 'West End']
```

To extract the list of flights times for *all* of the destinations, you need to combine these two statements (within a `for` loop).

In the code that follows, we've dispensed with the need for the `dests` and `wests2` variables, preferring to use the code *directly* as part of the `for` loop. We no longer hardcode *West End*, as the current destination is in `dest` (within the listcomp):

```
>>> for dest in set(fts.values()):
    print(dest, '->', [k for k, v in fts.items() if v == dest])
```

Treasure Cay -> ['10:45AM', '12:00PM']
 West End -> ['07:00PM', '09:55AM']
 Rock Sound -> ['05:55PM', '11:45AM']
 Freeport -> ['09:35AM', '05:00PM']

The fact that we've just written a `for` loop that appears to conform to our comprehension pattern starts our brain's little bell ringing again. Let's try to suppress that ringing for now, as the code you've just experimented with at your `>>>` prompt *displays* the data we need...but what you really need is to *store* the data in a new dictionary. Let's create a new dictionary (called `when`) to hold this newly extracted data. Head back to your `>>>` prompt and adjust the above `for` loop to use `when`:

```
1. Start with a new, empty dictionary.
>>> when = {}
>>> for dest in set(fts.values()):
    when[dest] = [k for k, v in fts.items() if v == dest]
```

```
>>> pprint.pprint(when)
{'Freeport': ['09:35AM', '05:00PM'],
 'Rock Sound': ['05:55PM', '11:45AM'],
 'Treasure Cay': ['10:45AM', '12:00PM'],
 'West End': ['07:00PM', '09:55AM']}
```

If you're like us, your little brain bell (that you've been trying to suppress) is likely ringing loudly and driving you crazy as you look at this code.

gotta love comprehensions

That Feeling You Get...

...when a single line of code starts to look like *magic*.

Switch off your brain bell, then take another look at the code that makes up your most recent `for` loop:

```
when = {}
for dest in set(fts.values()):
    when[dest] = [k for k, v in fts.items() if v == dest]
```

This code conforms to the pattern that makes it a potential target for reworking as a comprehension. Here's the above `for` loop code reworked as a dictcomp that extracts a *copy* of the data you need into a new dictionary called `when2`:

```
when2 = {dest: [k for k, v in fts.items() if v == dest] for dest in set(fts.values())}
```

It looks like *magic*, doesn't it?

This is the most complex comprehension you've seen so far, due mainly to the fact that the *outer* dictcomp contains an *inner* listcomp. That said, this dictcomp showcases one of the features that set comprehensions apart from the equivalent `for` loop code: you can put a comprehension almost anywhere in your code. The same does not hold for `for` loops, which can only appear as statements in your code (that is, not as part of expressions).

Of course, that's not to say you should *always* do something like this:

```
when = {}
for dest in set(fts.values()):
    when[dest] = [k for k, v in fts.items() if v == dest]
```



These do the
same thing.

```
when2 = {dest: [k for k, v in fts.items() if v == dest] for dest in set(fts.values())}
```

Be warned: a dictionary comprehension containing an embedded list comprehension is hard to read *the first time you see it*.

However, with repeated exposure, comprehensions do get easier to read and understand, and—as stated earlier in this chapter—Python programmers use them *a lot*. Whether you use comprehensions is up to you. If you are happier with the `for` loop code, use that. If you like the look of comprehensions, use them...just don't feel you *have to*.



Test Drive

Before moving on, let's put all of this comprehension code into our `do_convert.py` file. We can then run the code in this file (using IDLE) to see that the conversions and transformations that are required by *Bahamas Buzzers* are occurring as required. Confirm that your code is the same as ours, then execute the code to confirm that everything is working to specification.

```
do_convert.py - /Users/paul/Desktop/_NewBook/ch12/do_convert.py (3.5.2)
from datetime import datetime
import pprint

def convert2ampm(time24: str) -> str:
    return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')

with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.strip().split(',')
        flights[k] = v

pprint.pprint(flights)
print()

fts = {convert2ampm(k): v.title() for k, v in flights.items()}

pprint.pprint(fts)
print()

when = {dest: [k for k, v in fts.items() if v == dest] for dest in set(fts.values())}

pprint.pprint(when)
print()
```

Python 3.5.2 Shell

```
/ch12/do_convert.py ======
{'09:35': 'FREEPORT',
'09:55': 'WEST END',
'10:45': 'TREASURE CAY',
'11:45': 'ROCK SOUND',
'12:00': 'TREASURE CAY',
'17:00': 'FREEPORT',
'17:55': 'ROCK SOUND',
'19:00': 'WEST END'}

{'05:00PM': 'Freeport',
'05:55PM': 'Rock Sound',
'07:00PM': 'West End',
'09:35AM': 'Freeport',
'09:55AM': 'West End',
'10:45AM': 'Treasure Cay',
'11:45AM': 'Rock Sound',
'12:00PM': 'Treasure Cay'}

{'Freeport': ['05:00PM', '09:35AM'],
'Rock Sound': ['05:55PM', '11:45AM'],
'Treasure Cay': ['10:45AM', '12:00PM'],
'West End': ['07:00PM', '09:55AM']}
```

>>>

you are here ▶ 503

Ln: 214 Col: 4

1. The original, raw data, as read in from the CSV data file. This is "flights".

2. The raw data, copied and transformed into AM/PM format and Titlecase. This is "fts".

3. The list of flight times per destination (extracted from "fts"). This is "when".

We're flying now!

there are no
Dumb Questions

Q: So...let me get this straight: a comprehension is just syntactic shorthand for a standard looping construct?

A: Yes, specifically the `for` loop. A standard `for` loop and its equivalent comprehension do the same thing. It's just that the comprehension tends to execute considerably faster.

Q: When will I know when to use a list comprehension?

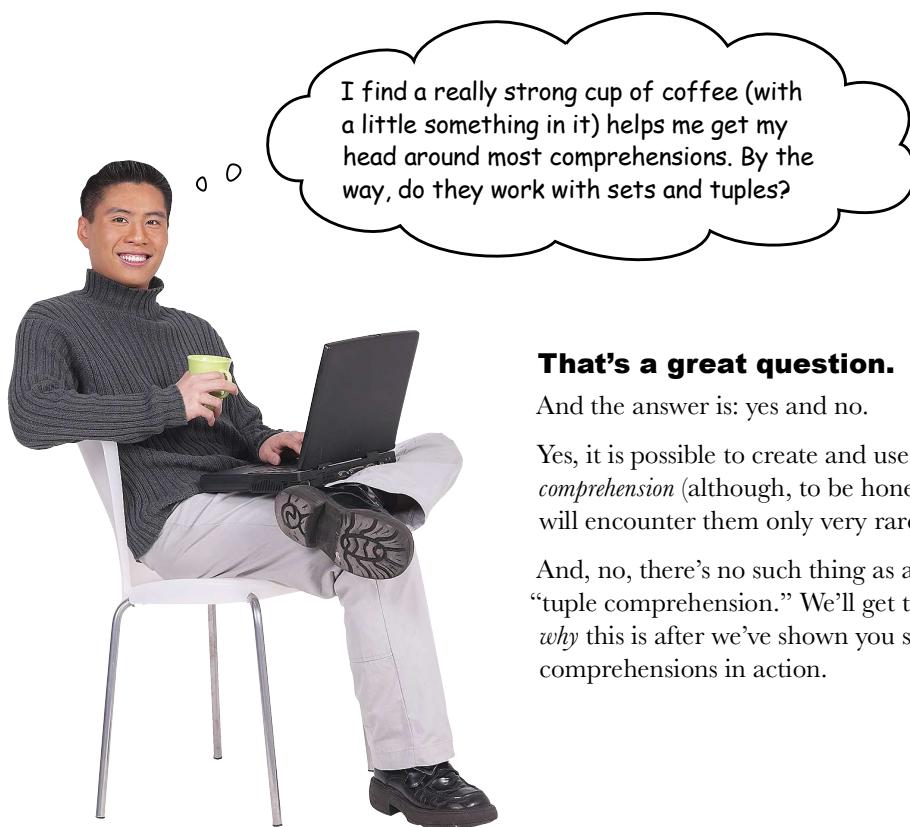
A: There are no hard and fast rules here. Typically, if you are producing a new list from an existing one, have a good look at your loop code. Ask yourself if the loop is a candidate for conversion to an equivalent comprehension. If the new list is “temporary” (that is, used once, then thrown away), ask yourself if an *embedded* list comprehension would be better for the task at hand. As a general rule, you should avoid introducing temporary variables into your code, especially if they're only used once. Ask yourself if a comprehension can be used instead.

Q: Can I avoid comprehensions altogether?

A: Yes, you can. However, they tend to see quite a bit of use within the wider Python community, so unless your plan is to never look at anyone else's code, we'd suggest taking the time to become familiar with Python's comprehension technology. Once you get used to seeing them, you'll wonder how you ever lived without them. Did we mention that they are *fast*?

Q: Yes, I get that, but is speed such a big deal nowadays? My laptop is super-fast and it runs my `for` loops quick enough.

A: That's an interesting observation. It's true that today we have computers that are vastly more powerful than anything that's come before. It's also true that we spend a lot less time trying to eke out every last CPU cycle from our code (because, let's face it: we don't have to anymore). However, when presented with a technology that offers a performance boost, why not use it? It's a small bit of effort for a big return in performance.



That's a great question.

And the answer is: yes and no.

Yes, it is possible to create and use a *set comprehension* (although, to be honest, you will encounter them only very rarely).

And, no, there's no such thing as a “tuple comprehension.” We'll get to *why* this is after we've shown you set comprehensions in action.

The Set Comprehension in Action

A set comprehension (or *setcomp* for short) allows you to create a new set in one line of code, using a construct that's very similar to the list comprehension syntax.

What sets a *setcomp* apart from a *listcomp* is that the set comprehension is surrounded by curly braces (unlike the square brackets around a *listcomp*). This can be confusing, as *dictcomps* are surrounded by curly braces, too. (One wonders what came over the Python core developers when they decided to do this.)

A literal set is surrounded by curly braces, as are literal dictionaries. To tell them apart, look for the colon character used as a delimiter in dictionaries, as the colon has no meaning in sets. The same advice applies to quickly determining whether a curly-braced comprehension is a *dictcomp* or a *setcomp*: look for the colon. If it's there, you're looking at a *dictcomp*. If not, it's a *setcomp*.

Here's a quick set comprehension example (which hearkens back to an earlier example in this book). Given a set of letters (*in vowels*), and a string (*in message*), the `for` loop as well as its equivalent *setcomp* produce the same result—a set of the vowels found in *message*:

```
vowels = {'a', 'e', 'i', 'o', 'u'}
message = "Don't forget to pack your towel."

found = set()
for v in vowels:
    if v in message:
        found.add(v)
```

The *setcomp* follows
the same pattern as
the *listcomp*.



```
found2 = { v for v in vowels if v in message }
```

Note the use of curly braces here,
as this comprehension produces a set
when executed by the interpreter

Take a few moments to experiment with the code on this page at your `>>>` prompt. Because you already know what *listcomps* and *dictcomps* can do, getting your head around set comprehensions isn't that tricky. There's really nothing more to them than what's on this page.

spot that comprehension

How to Spot a Comprehension

As you become more familiar with the look of comprehension code, they become easier to spot and understand. Here's a good general rule for spotting list comprehensions:

If you spot code surrounded by [and], then you are looking at a list comprehension.

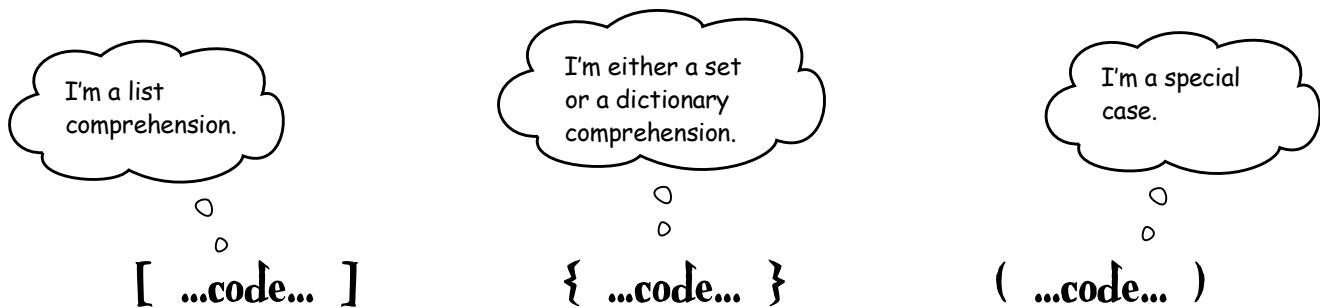
This rule can be generalized as follows:

If you spot code surrounded by brackets (curly or square), then you are likely looking at a comprehension.

Why the use of the word “likely”?

In addition to code being surrounded by [], comprehensions can also, as you've seen, be surrounded by { }. When code is surrounded by [and], you are looking at a **list** comprehension. When code is surrounded by { and }, you are looking at either a **set** or a **dictionary** comprehension. A dictcomp is easy to spot thanks to its use of the colon character as a delimiter.

However, code can also appear between (and), which is a *special case*, even though you'd be forgiven for suggesting that code surrounded by parentheses must surely be a *tuple comprehension*. You'd be forgiven, but wrong: “tuple comprehensions” don't exist, even though you can put code between (and). After the “fun” you've been having with comprehensions so far in this chapter, you may be thinking: *could this get any weirder?*



Let's conclude this chapter (and this book) by exploring what's going on with code that appears between (and). It's not a “tuple comprehension,” but it is obviously allowed, so what is it?

What About “Tuple Comprehensions”?

Python’s four built-in data structures (tuples, lists, sets, and dictionaries) can be put to many uses. However, all but tuples can be created via a comprehension.

Why is this?

It turns out that the idea of a “tuple comprehension” doesn’t really make sense. Recall that tuples are *immutable*: once a tuple is created, it cannot be changed. This also means that it’s not possible to generate a tuple’s values in code, as this short IDLE session shows:

The screenshot shows a Python 3.5.2 Shell window. The code entered is:

```
>>> names = ()
>>>
>>> for n in ('John', 'Paul', 'George', 'Ringo'):
...     names.append(n)
```

A handwritten note on the left says "Create a new, empty tuple." An arrow points from this note to the first line of code. Another handwritten note on the right says "Try to dynamically add data to the tuple." An arrow points from this note to the line "names.append(n)". A third handwritten note on the right says "You can't append to an existing tuple, as it is immutable." An arrow points from this note to the error message in the traceback.

Traceback (most recent call last):

```
  File "<pyshell#17>", line 2, in <module>
    names.append(n)
AttributeError: 'tuple' object has no attribute 'append'
```

Ln: 44 Col: 4

There’s nothing weird or wonderful going on here, as this is the behavior expected from tuples: once one exists, it *cannot* be changed. This fact alone should be enough to rule out using a tuple within any sort of comprehension. But take a look at this interaction at the >>> prompt. The second loop differs from the first in the smallest of ways: the square brackets around the listcomp (in the first loop) have been replaced with parentheses (in the second):

The screenshot shows a Python 3.5.2 Shell window. The code entered is:

```
>>>
>>> for i in [x*3 for x in [1,2,3,4,5]]:
...     print(i)
```

A handwritten note on the left says "What gives? Both loops generate the same results." An arrow points from this note to the first loop. A handwritten note on the right says "This for loop and list comprehension combination displays each of the list's values tripled. You know this is a listcomp, as that's code inside square brackets." An arrow points from this note to the first loop. Another handwritten note on the right says "But look at this. The parentheses makes this look like a ‘tuple comprehension’—but you know such a thing is impossible. Yet the loop still produces the expected output. Weird, eh?" An arrow points from this note to the second loop.

```
3
6
9
12
15
```

```
>>>
>>> for i in (x*3 for x in [1,2,3,4,5]):
...     print(i)
```

```
3
6
9
12
15
```

Ln: 46 Col: 4

generate your data

Parentheses Around Code == Generator

When you come across something that looks like a listcomp but is surrounded by parentheses, you're looking at a **generator**:

This looks like a listcomp, but isn't: `for i in (x*3 for x in [1, 2, 3, 4, 5]):
 print(i)`

A generator can be used anywhere a listcomp is used, and produces the same results.

As you saw at the bottom of the last page, when you replace a listcomp's surrounding square brackets with parentheses, the results are the same; that is, the generator and the listcomp produce the same data.

However, they do not execute in the same way.

If you're scratching your head at the previous sentence, consider this: when a listcomp executes, it produces **all** of its data prior to any other processing occurring. Taken in the context of the example at the top of this page, the `for` loop doesn't start processing *any* of the data produced by the listcomp until the listcomp is done. This means that a listcomp that takes a long time to produce data delays any other code from running *until the listcomp concludes*.

With a small list of data items (as shown above), this is not a big issue.

But imagine your listcomp is required to work with a list that produces 10 million items of data. You've now got two issues: (1) you have to wait for the listcomp to process those 10 million data items *before doing anything else*, and (2) you have to worry that the computer running your listcomp has enough RAM to hold all that data in memory while the listcomp executes (**10 million** individual pieces of data). If your listcomp runs out of memory, the interpreter terminates (and your program is toast).

Generators produce data items one at a time...

When you replace your listcomp's square brackets with parentheses, the listcomp becomes a **generator**, and your code behaves differently.

Unlike a listcomp, which must conclude before any other code can execute, a generator releases data as soon as the data is produced by the generator's code. This means if you generate 10 million data items, the interpreter only needs enough memory to hold **one** data item (at a time), and any code that's waiting to consume the data items produced by the generator executes immediately; that is, *there's no waiting*.

There's nothing quite like an example to understand the difference using a generator can make, so let's perform a simple task twice: once with a listcomp, then again with a generator.

Listcomps and generators produce the same results, but operate in a very different way.

Using a Listcomp to Process URLs

To demonstrate the difference using a generator can make, let's perform a task using a listcomp (before rewriting the task as a generator).

As has been our practice throughout this book, let's experiment with some code at the >>> prompt that uses the `requests` library (which lets you programmatically interact with the Web). Here's a small interactive session that imports the `requests` library, defines a three-item tuple (called `urls`), and then combines a `for` loop with a listcomp to request each URL's landing page, before processing the web response returned.

To understand what's going on here, you need to follow along on your computer.

Define a tuple of URLs. Feel free to substitute your own URLs here. Just be sure to define at least three.

```
>>>
>>> import requests
>>>
>>> urls = ('http://headfirstlabs.com', 'http://oreilly.com', 'http://twitter.com')
>>>
>>> for resp in [requests.get(url) for url in urls]:
...     print(len(resp.content), '->', resp.status_code, '->', resp.url)
...
31590 -> 200 -> http://headfirstlabs.com/
78722 -> 200 -> http://www.oreilly.com/
128244 -> 200 -> https://twitter.com/
>>> |
```

The "for" loop contains a listcomp, which, for each of the URLs in "urls", gets the website's landing page.

Nothing weird or wonderful here. The output produced is exactly what's expected.

With each response received, display the size of the returned landing page (in bytes), the HTTP status code, and the URL used.

If you're following along on your computer, you will experience a noticeable delay between entering the `for` loop code and seeing the results. When the results appear, they are displayed in one go (all at once). This is because the listcomp works through each of the URLs in the `urls` tuple before making any results available to the `for` loop. The outcome? You have to wait for your output.

Note that there's *nothing* wrong with this code: it does what you want it to, and the output is correct. However, let's rework this listcomp as a generator to see the difference it makes. As mentioned above, be sure to follow along on your computer as you work through the next page (so you can see what happens).

**Download
"requests" from
PyPI using the
"pip" command.**

gotta love generators

Using a Generator to Process URLs

Here's the example from the last page reworked as a generator. Doing so is easy; simply replace the listcomp's square brackets with parentheses:

The screenshot shows a Python 3.5.2 Shell window. The code entered is:

```
>>>
>>>
>>>
>>>
>>> for resp in (requests.get(url) for url in urls):
    print(len(resp.content), '->', resp.status_code, '->', resp.url)
```

A handwritten note with a bracket and arrow points to the square brackets in the list comprehension, indicating they should be replaced by parentheses.

Ln: 151 Col: 1

A short moment after entering the above `for` loop, the first result appears:

The screenshot shows the Python shell continuing to run the code. The output is:

```
>>>
>>>
>>> for resp in (requests.get(url) for url in urls):
    print(len(resp.content), '->', resp.status_code, '->', resp.url)

31590 -> 200 -> http://headfirstlabs.com/ ← The first URL's response
```

Ln: 153 Col: 0

Then, a moment later, the next line of results appear:

The screenshot shows the Python shell continuing to run the code. The output is:

```
>>>
>>> for resp in (requests.get(url) for url in urls):
    print(len(resp.content), '->', resp.status_code, '->', resp.url)

31590 -> 200 -> http://headfirstlabs.com/
78722 -> 200 -> http://www.oreilly.com/ ← The second URL's response
```

Ln: 154 Col: 0

Then—finally—a few moments later, the last results line appears (and the `for` loop ends):

The screenshot shows the Python shell continuing to run the code. The output is:

```
>>>
>>> for resp in (requests.get(url) for url in urls):
    print(len(resp.content), '->', resp.status_code, '->', resp.url)

31590 -> 200 -> http://headfirstlabs.com/
78722 -> 200 -> http://www.oreilly.com/
128244 -> 200 -> https://twitter.com/ ← The third, and final, URL's response
>>> |
```

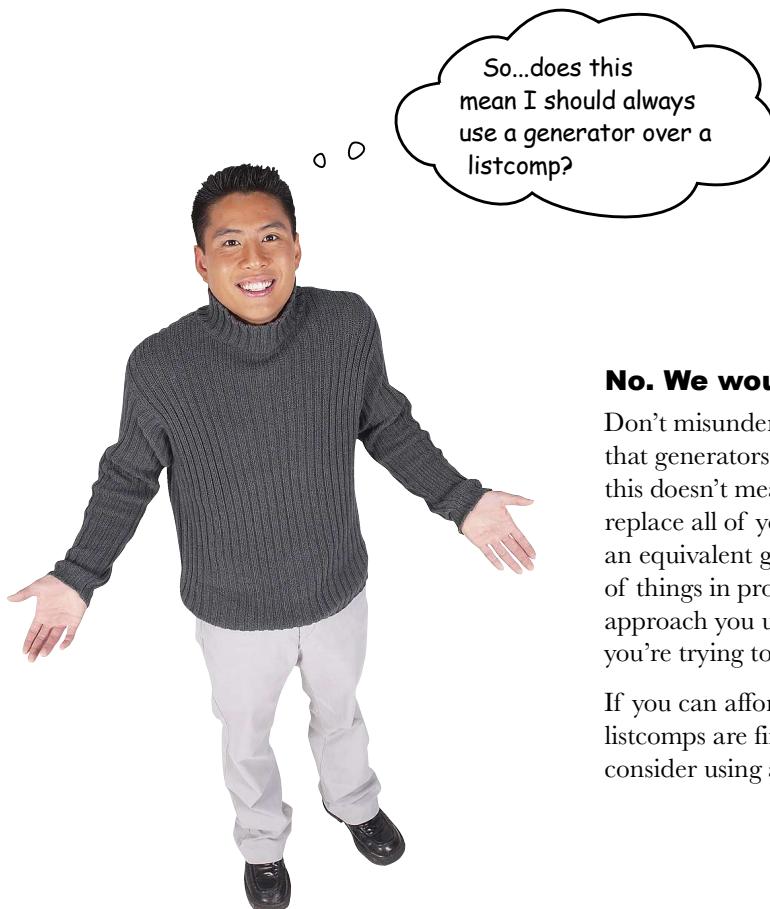
Ln: 156 Col: 4

Using a Generator: What Just Happened?

If you compare the results produced by your listcomp to those produced by your generator, they are *identical*. However, the behavior of your code isn't.

The listcomp **waits** for all of its data to be produced before feeding any data to the waiting `for` loop, whereas the generator **releases** data as soon as it becomes available. This means the `for` loop that uses the generator is much more responsive, as opposed to the listcomp (which makes you wait).

If you're thinking this isn't really that big a deal, imagine if the URLs tuple was defined with one hundred, one thousand, or one million URLs. Further, imagine that the code processing the response is feeding the processed data to another process (perhaps a waiting database). As the number of URLs increases, the listcomp's behavior becomes worse compared to that of the generator.



No. We wouldn't say that.

Don't misunderstand: the fact that generators exist is *great*, but this doesn't mean you'll want to replace all of your listcomps with an equivalent generator. Like a lot of things in programming, which approach you use depends on what you're trying to do.

If you can afford to wait, then listcomps are fine; otherwise, consider using a generator.

One interesting usage of generators is to embed them within a function. Let's take a look at encapsulating your just-created generator in a function.

generator functions rock

Define What Your Function Needs to Do

Let's imagine that you want to take your `requests` generator and turn it into a function. You've decided to package the generator within a small module you're writing, and you want other programmers to be able to use it without having to know or understand generators.

Here's your generator code once more:

```
import requests ← Import any required libraries.  
  
urls = ('http://headfirstlabs.com', 'http://oreilly.com', 'http://twitter.com') ← Define a tuple of URLs.  
  
for resp in (requests.get(url) for url in urls): ←  
    print(len(resp.content), '->', resp.status_code, '->', resp.url)  
  
↑  
Process the generated data.  
  
The generator (remember:  
looks like a listcomp, but is  
surrounded by parentheses)
```

Let's create a function that encapsulates this code. The function, which is called `gen_from_urls`, takes a single argument (a tuple of URLs), and returns a tuple of results for each URL. The returned tuple contains three values: the length of the URL's content, the HTTP status code, and the URL the response came from.

Assuming `gen_from_urls` exists, you want other programmers to be able to execute your function as part of a `for` loop, like this:

```
from url_utils import gen_from_urls ← Import the function from your module.  
  
urls = ('http://headfirstlabs.com', 'http://oreilly.com', 'http://twitter.com') ← Define a tuple of URLs.  
  
for resp_len, status, url in gen_from_urls(urls): ←  
    print(resp_len, status, url)  
  
↑  
Process the data.  
  
Call the function on each iteration of the "for" loop.
```

Although this new code does not look all that different from the code at the top of the page, note that programmers using `gen_from_urls` have no clue (nor do they need to know) that you're using `requests` to talk to the Web. Nor do they need to know that you're using a generator. All of your implementation details and choices are hidden behind that easy-to-understand function call.

Let's see what's involved in writing `gen_from_urls` so that it can generate the data you need.

Yield to the Power of Generator Functions

Now that you know what the `gen_from_urls` function needs to do, let's go about writing it. Begin by creating a new file called `url_utils.py`. Edit this file, then add `import requests` as its first line of code.

The function's `def` line is straightforward, as it takes a single tuple on the way in, and returns a tuple on output (note how we've included type annotations to make this explicit for users of our generator function). Go ahead and add the function's `def` line to the file, like so:

```
import requests

def gen_from_urls(urls: tuple) -> tuple:
```

After
importing
"requests",
define your
new function.

The function's suite is the generator from the last page, and the `for` line is a simple copy-and-paste:

```
import requests

def gen_from_urls(urls: tuple) -> tuple:
    for resp in (requests.get(url) for url in urls):
```

Add in your "for"
loop line with the
generator.

The next line of code needs to "return" the result of that GET request as performed by the `requests.get` function. Although it's tempting to add the following line as the `for`'s suite, **please don't do this**:

```
    return len(resp.content), resp.status_code, resp.url
```



When a function executes a `return` statement, the function *terminates*. You don't want this to happen here, as the `gen_from_urls` function is being called as part of a `for` loop, which is expecting a *different* tuple of results *each time the function's called*.

But, if you can't execute `return`, what are you to do?

Use `yield` instead. The `yield` keyword was added to Python to support the creation of **generator functions**, and you can use it anywhere a `return` is used. When you do, your function morphs into a generator function that can be "called" from any iterator, which, in this case, is from within your `for` loop:

```
import requests

def gen_from_urls(urls: tuple) -> tuple:
    for resp in (requests.get(url) for url in urls):
        yield len(resp.content), resp.status_code, resp.url
```

Use "yield" to return
each line of results
from the GET
response to the waiting
"for" loop. Remember:
DON'T use "return".

Let's take a closer look at what's going on here.

Tracing Your Generator Function, 1 of 2

To understand what happens when your generator function runs, let's trace the execution of the following code:

```
Import your
generator
function.
from url_utils import gen_from_urls

urls = ('http://talkpython.fm', 'http://pythonpodcast.com', 'http://python.org')

for resp_len, status, url, in gen_from_urls(urls):
    print(resp_len, '->', status, '->', url)

Define a
tuple of
URLs.

Use your generator
function as part of a
"for" loop.
```

The first two lines of code are simple enough: the function is imported, and a tuple of URLs is defined.

The fun starts on the next line of code, when the `gen_from_urls` generator function is invoked. Let's refer to this `for` loop as "the calling code":

```
for resp_len, status, url, in gen_from_urls(urls): ←
The calling code's "for"
loop communicates
with the generator
function's "for" loop.

def gen_from_urls(urls: tuple) -> tuple:
    for resp in (requests.get(url) for url in urls):
        yield len(resp.content), resp.status_code, resp.url
```

The `for` loop contains the generator, which takes the first URL in the `urls` tuple and sends a GET request to the identified server. When the HTTP response is returned from the server, the `yield` statement executes.

This is where things get interesting (or weird, depending on your point of view).

Rather than executing, then moving on to the next URL in the `urls` tuple (i.e., continuing with the next iteration of `gen_from_urls`'s `for` loop), `yield` passes its three pieces of data back to the calling code. Rather than terminating, the `gen_from_urls` function generator now *waits*, as if in *suspended animation*...

Tracing Your Generator Function, 2 of 2

When the data (as passed back by `yield`) arrives at the calling code, the `for` loop's suite executes. As the suite contains a single call to the `print` BIF, that line of code executes and displays the results from the first URL on screen:

```
print(resp_len, '->', status, '->', url)
```

```
34591 -> 200 -> https://talkpython.fm/
```

The calling code's `for` loop then iterates, calling `gen_from_urls` again...sort of.

This is *almost* what happens. What actually happens is that `gen_from_urls` is awakened from its suspended animation, then continues to run. The `for` loop within `gen_from_urls` iterates, takes the next URL from the `urls` tuple, and contacts the server associated with the URL. When the HTTP response is returned from the server, the `yield` statement executes, passing its three pieces of data back to the calling code (which the function accesses via the `resp` object):

```
yield len(resp.content), resp.status_code, resp.url
```



The three yielded pieces of data are taken from the "resp" object returned by the "requests" library's "get" method.

As before, rather than terminating, the `gen_from_urls` generator function now *waits* once more, as if in *suspended animation*...

When the data (as passed back by `yield`) arrives at the calling code, the `for` loop's suite executes `print` once more, displaying the second set of results on screen:

```
34591 -> 200 -> https://talkpython.fm/
19468 -> 200 -> http://pythonpodcast.com/
```

The calling code's `for` loop iterates, "calling" `gen_from_urls` once more, which results in your generator function awakening again. The `yield` statement is executed, results are returned to the calling code, and the display updates again:

```
34591 -> 200 -> https://talkpython.fm/
19468 -> 200 -> http://pythonpodcast.com/
47413 -> 200 -> https://www.python.org/
```

At this point, you've exhausted your tuple of URLs, so the generator function and the calling code's `for` loop both terminate. It's as if the two pieces of code were taking turns to execute, passing data between themselves on each turn.

Let's see this in action at the `>>>` prompt. It's now time for one last *Test Drive*.

don't be sad



Test DRIVE

In this, the last *Test Drive* in this book, let's take your generator function for a spin. As has been our practice all along, load your code into an IDLE edit window, then press F5 to exercise the function at the >>> prompt. Follow along with our session (below):

Here's the "gen_from_urls" generator function in the "url_utils.py" module.

```
import requests

def gen_from_urls(urls: tuple) -> tuple:
    for resp in (requests.get(url) for url in urls):
        yield len(resp.content), resp.status_code, resp.url
```

Ln: 8 Col: 0

The first example below shows `gen_from_urls` being called as part of a `for` loop. As expected, the output is the same as that obtained a few pages back.

The second example below shows `gen_from_urls` being used as part of a `dictcomp`. Note how the new dictionary only needs to store the URL (as a key) and the size of the landing page (as the value). The HTTP status code is *not* needed in this example, so we tell the interpreter to ignore it using Python's **default variable name** (which is a single underscore character):

Each line of results appears, after a short pause, as the data is generated by the function.

This dictcomp associates the URL with the length of its landing page.

```
>>>
>>>
>>> for resp_len, status, url in gen_from_urls(urls):
...     print(resp_len, '>', status, '>', url)
...
31590 -> 200 -> http://headfirstlabs.com/
78722 -> 200 -> http://www.oreilly.com/
128244 -> 200 -> https://twitter.com/
...
>>> urls_res = {url: size for size, _, url in gen_from_urls(urls)}
>>>
>>> import pprint
>>>
>>> pprint.pprint(urls_res)
{'http://headfirstlabs.com/': 31590,
 'http://www.oreilly.com/': 78722,
 'https://twitter.com/': 128244}
...
>>>
```

Pass the tuple of URLs to the generator function.

The underscore tells the code to ignore the yielded HTTP status code value.

Pretty-printing the "url_res" dictionary confirms that the generator function can be used within a dictcomp (as well as within a "for" loop).

Ln: 271 Col: 0

Concluding Remarks

The use of comprehensions and generator functions is often regarded as an advanced topic in the Python world. However, this is mainly due to the fact that these features are missing from other mainstream programming languages, which means that programmers moving to Python sometimes struggle with them (as they have no existing point of reference).

That said, over at *Head First Labs*, the Python programming team *loves* comprehensions and generators, and believes that with repeated exposure, specifying the looping constructs that use them becomes second nature. They can't imagine having to do without them.

Even if you find the comprehension and generator syntax weird, our advice is to stick with them. Even if you dismiss the fact that they are more performant than the equivalent `for` loop, the fact that you can use comprehensions and generators in places where you cannot use a `for` loop is reason enough to take a serious look at these Python features. Over time, and as you become more familiar with their syntax, opportunities to exploit comprehensions and generators will present themselves as naturally as those that tell your programming brain to use a function here, a loop there, a class over here, and so on. Here's a review of what you were introduced to in this chapter:

BULLET POINTS

- When it comes to working with data in files, Python has options. As well as the standard `open` BIF, you can use the facilities of the standard library's `csv` module to work with CSV-formatted data.
- Method **chains** allow you to perform processing on data in one line of code. The `string.strip().split()` chain is seen a lot in Python code.
- Take care with how you order your method chains. Specifically, pay attention to the type of data returned from each method (and ensure type compatibility is maintained).
- A `for` loop used to transform data from one format to another can be reworked as a **comprehension**.
- Comprehensions can be written to process existing lists, dictionaries, and sets, with list comprehensions being the most popular variant "in the wild." Seasoned Python programmers refer to these constructs as *listcomps*, *dictcomps*, and *setcomps*.
- A **listcomp** is code surrounded by square brackets, while a **dictcomp** is code surrounded by curly braces (with colon delimiters). A **setcomp** is also code surrounded by curly braces (but without the dictcomp's colon).
- There's no such thing as a "tuple comprehension," as tuples are immutable (so it makes no sense to try to dynamically create one).
- If you spot comprehension code surrounded by parentheses, you're looking at a **generator** (which can be turned into a function that itself uses `yield` to generate data as needed).

As this chapter concludes (and, by definition, the core content of this book), we have one final question to ask you. Take a deep breath, then flip the page.

gotta love whitespace

One Final Question

OK. Here goes, our final question to you: *at this stage in this book, do you even notice Python's use of significant whitespace?*

The most common complaint heard from programmers new to Python is its use of whitespace to signify blocks of code (instead of, for instance, curly braces). But, after a while, your brain tends not to notice anymore.

This is not an accident: Python's use of significant whitespace was intentional on the part of the language's creator.

It was deliberately done this way, because **code is read more than it's written**. This means code that conforms to a consistent and well-known look and feel is easier to read. This also means that Python code written 10 years ago by a complete stranger is still readable by you *today* because of Python's use of whitespace.

This is a big win for the Python community, which makes it a big win for *you*, too.

Chapter 12's Code

 This is "do_convert.py".

```
from datetime import datetime
import pprint

def convert2ampm(time24: str) -> str:
    return datetime.strptime(time24, '%H:%M').strftime('%I:%M%p')

with open('buzzers.csv') as data:
    ignore = data.readline()
    flights = {}
    for line in data:
        k, v = line.strip().split(',')
        flights[k] = v

pprint.pprint(flights)
print()

fts = {convert2ampm(k): v.title() for k, v in flights.items()}

pprint.pprint(fts)
print()

when = {dest: [k for k, v in fts.items() if v == dest] for dest in set(fts.values())}

pprint.pprint(when)
print()
```

 This is "url_utils.py".

```
import requests

def gen_from_urls(urls: tuple) -> tuple:
    for resp in (requests.get(url) for url in urls):
        yield len(resp.content), resp.status_code, resp.url
```

so long (and thanks for all the fish)

It's Time to Go...



You're on your way!

We're sad to see you leave, but nothing would make us happier than you taking what you've learned about Python in this book and *putting it to use*. You're at the start of your Python journey, and there's always more to learn. Of course, you're not quite done with this book just yet. There's the five (yes: five!) appendixes to work through. We promise they're not that long, and are well worth the effort. And, of course, there's the index—let's not forget about the index!

We hope you've had as much fun learning about Python as we've had writing this book for you. It's been a blast. Enjoy!

appendix a: installation



Installing Python



First things first: let's get Python installed on your computer.

Whether you're running on *Windows*, *Mac OS X*, or *Linux*, Python's got you covered. How you install it on each of these platforms is specific to how things work on each of these operating systems (we know...a shocker, eh?), and the Python community works hard to provide installers that target all the popular systems. In this short appendix, you'll be guided through installing Python on your computer.

works on windows

Install Python 3 on Windows

Unless you (or someone else) has installed the Python interpreter onto your Windows PC, it is unlikely to be preinstalled. Even if it is, let's install the latest and greatest version of Python 3 into your *Windows* computer now.

If you already have a version of Python 3 installed, it'll be upgraded. If you have Python 2 installed, Python 3 will install alongside it (but won't interfere with your Python 2 in any way). And if you don't have any version of Python yet, well, you soon will!

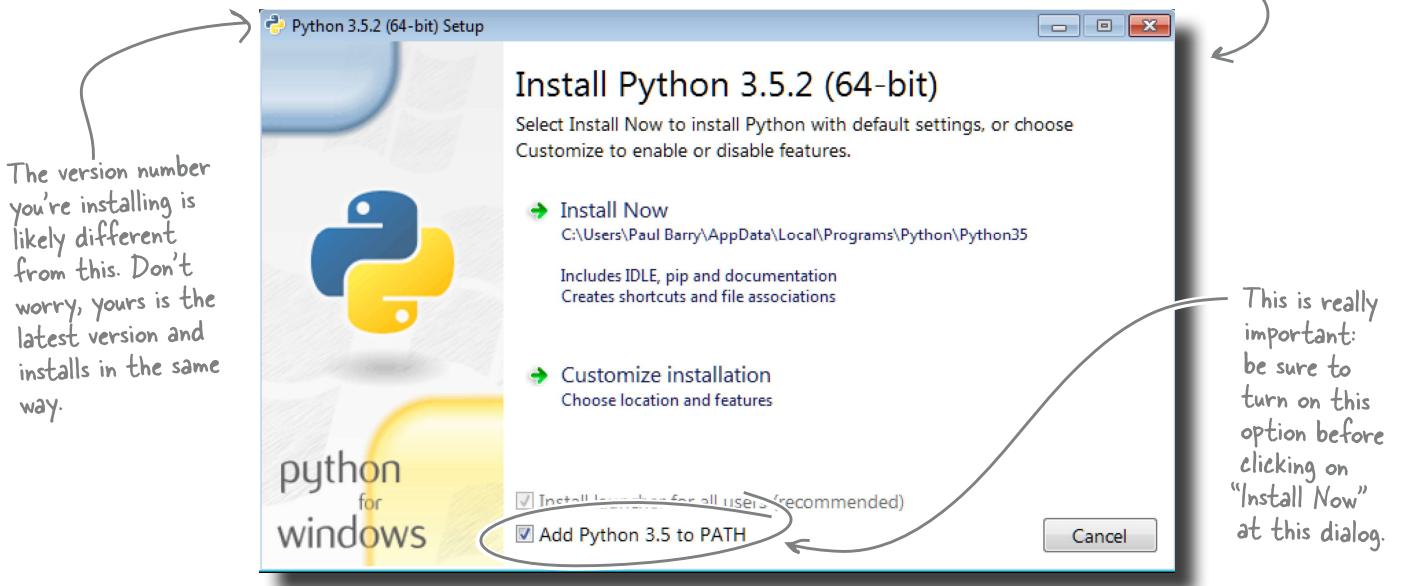
Download, then install

Point your browser to www.python.org, and then click the *Downloads* tab.

Two large buttons will appear, offering the choice of the latest version of Python 3 or Python 2. Click on the *Python 3* button. Go ahead and save the file for download when prompted. After a little while, the download will complete. Locate the downloaded file in your *Downloads* folder (or wherever you saved it), then double-click on the file to start the install.

A standard *Windows* installation process begins. By and large, you can click on *Next* at each of the prompts, except for this one (shown below), where you'll want to pause to make a configuration change to ensure *Add Python 3.5 to Path* is selected; this ensures *Windows* can find the interpreter whenever it needs to:

Note: As this book hurtles toward its date with the printing press, the next version of Python 3 (release 3.6) is due out. As this won't be until the end of 2016 (a mere handful of weeks *after* this book publishes), we're showing 3.5 in these screenshots. Don't worry about matching the version we have here. Go ahead and download/install the latest.



Check Python 3 on Windows

Now that the Python interpreter is installed on your *Windows* machine, let's run a few checks to confirm all is OK.

For starters, you should have a new group on your *Start* menu under *All Programs*. We've included what it looks like on one of the *Head First Labs' Windows 7* machines. Yours should look similar. If it doesn't, you may need to redo the installation. Windows 8 users (or higher) should also have a new group similar to this.

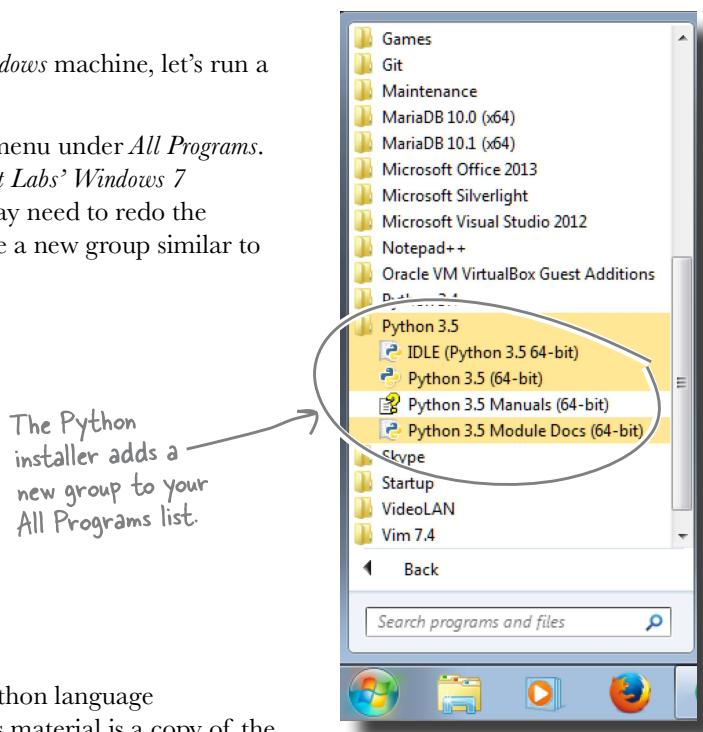
Let's examine the items in the Python 3.5 group from the bottom up.

The *Python 3.5 Modules Docs* option provides access to all of the documentation included with all of the installed modules that are available within your Python system. You'll be learning lots about modules as you work through this book, so you don't need to worry about doing anything with this option right now.

The *Python 3.5 Manuals* option opens the entire set of Python language documentation in the standard *Windows* help utility. This material is a copy of the Python 3 documentation available on the Web.

The *Python 3.5* option fires up a text-based interactive command prompt, `>>>`, which is used to experiment with code as you write it. We'll have more to say about the `>>>` prompt starting from Chapter 1. If you have clicked on this option to try it out and are now at a loss as to what to do, type `quit()` to escape back to *Windows*.

The final option, *IDLE (Python 3.5)*, runs the Python integrated development environment, which is called *IDLE*. This is a very simple IDE that provides access to Python's `>>>` prompt, a passable text editor, the Python debugger, and the Python documentation. We'll be using *IDLE* a lot in this book, starting in Chapter 1.



It's Python 3 on Windows, sort of...

Python's heritage is on Unix and Unix-like systems, and this can sometimes come through when you're working in *Windows*. For instance, some software that is assumed to exist by Python isn't always available by default on *Windows*, so to get the most out of Python, programmers on *Windows* often have to install a few extra bits and pieces. Let's take a moment to install one such bonus piece to demonstrate how these missing bits can be added when needed.

windows need pyreadline

Add to Python 3 on Windows

Sometimes programmers using the *Windows* version of Python feel like they are being short-changed: some of the features assumed (by Python) on those other platforms are “missing” from *Windows*.

Thankfully, some enterprising programmers have written third-party modules that can be installed *into* Python, thus providing the missing functionality. Installing any of these modules involves only a little bit of work at the *Windows* command prompt.

As an example, let’s add Python’s implementation of the popular `readline` functionality to your *Windows* version of Python. The `pyreadline` module provides a Python version of `readline`, effectively plugging this particular hole in any default *Windows* installation.

Open up a *Windows* command prompt and follow along. Here, we’re going to use a software installation tool (included in Python 3.5) to install the `pyreadline` module. The tool is called `pip`, short for “Python Index Project,” named after the work that spawned `pip`’s creation.

At the *Windows* command prompt, type **`pip install pyreadline`**:



Geek Bits

The `readline` library implements a set of functions that provide interactive text-editing facilities (typically at command lines). The `pyreadline` module provides a Python interface to `readline`.

```
File Edit Window Help InstallingPyReadLine
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Users\Head First>
C:\Users\Head First> pip install pyreadline
Downloading/unpacking pyreadline
...
...
Successfully installed pyreadline
Cleaning up...
C:\Users\Head First>
```

This is what you need to type into the command prompt.

You'll see lots of messages here.

If you see this message, all is OK.

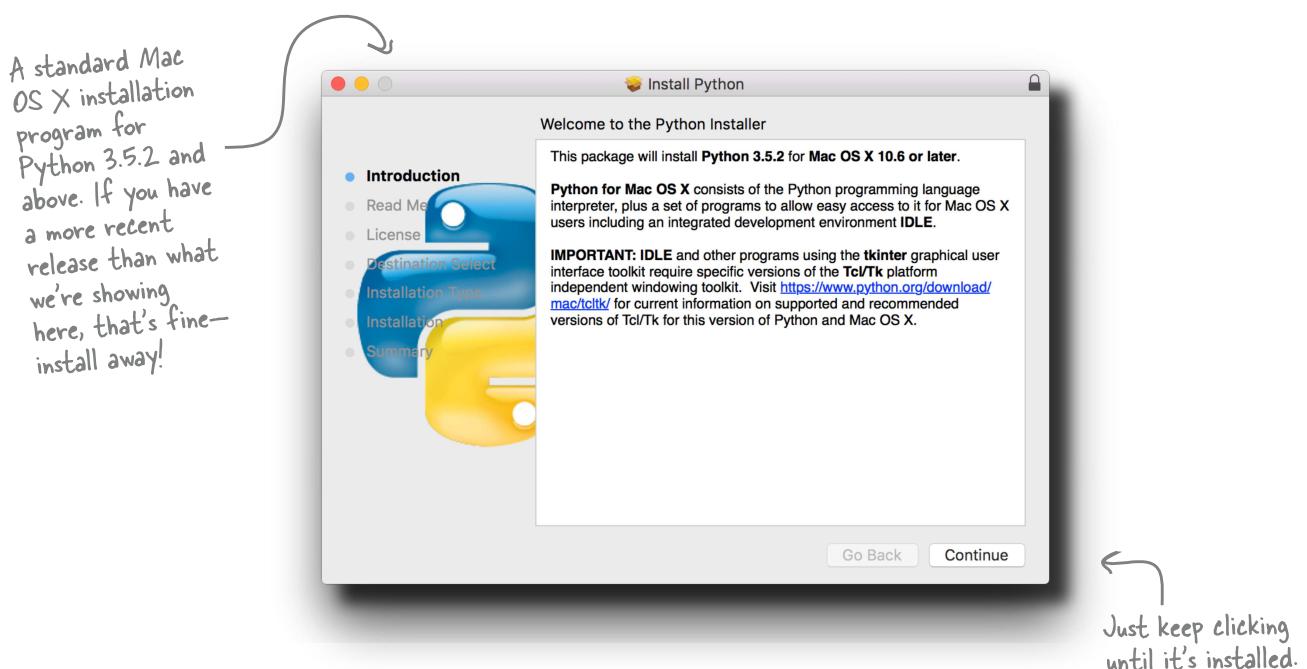
Make sure you are connected to the Internet before issuing this command.

And with that, `pyreadline` is installed and ready to go on *Windows*.

You can now flip back to Chapter 1 to get started with some sample Python code.

Install Python 3 on Mac OS X (macOS)

Python 2 comes preinstalled on *Mac OS X* by default. But this is no use to us, as we want to use Python 3 instead. Thankfully, when you visit the Python website (<http://www.python.org>), it is smart enough to work out that you're using a Mac. Hover your mouse over the *Download* tab, then click the 3.5.x button to download the Mac installer for Python. Select the latest version of Python 3, download its package, and then install in the usual "Mac way."



Using a package manager

On Macs, it is also possible to use one of the popular open source *package managers*, namely *Homebrew* or *MacPorts*. If you have never used either of these package managers, feel free to skip this little section and jump over to the top of the next page. If, however, you are already using either of these package managers, here are the commands you need to install Python 3 on your Mac from inside a terminal window:

- On *Homebrew*, type **brew install python3**.
- On *MacPorts* type **port install python3**.

And that's it: you're golden. Python 3 is ready for action on *Mac OS X*—let's take a look at what gets installed.

set your path

Check and Configure Python 3 on Mac OS X

To see if the install succeeded on *Mac OS X*, click on the *Applications* icon on your dock, then look for the *Python 3* folder.

Click on the *Python 3* folder and you'll see a bunch of icons (below).

The Python 3 folder on Mac OS X

The first option, *IDLE*, is by far the most useful, and it is how you will interact with Python 3 most of the time while learning the language. Choosing this option opens Python's integrated development environment called *IDLE*. This is a very simple IDE that provides access to Python's >>> interactive prompt, a passable text editor, the Python debugger, and the Python documentation. We'll be using *IDLE* a lot in this book.

The *Python Documentation.html* option opens a local copy of Python's entire documentation in HTML within your default browser (without requiring you to be online).

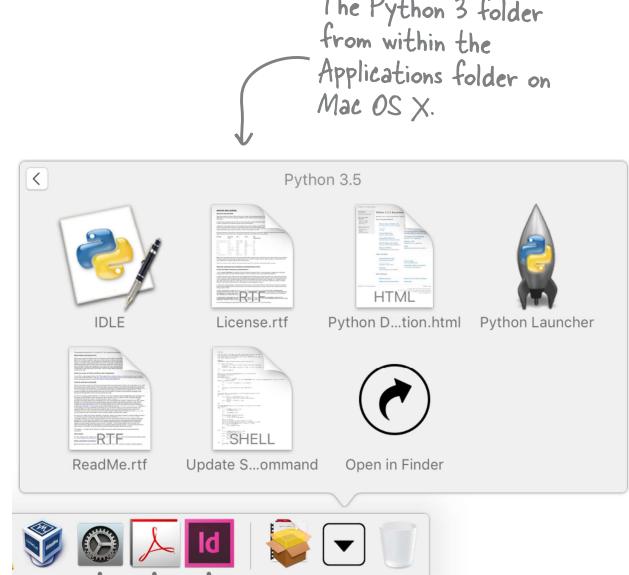
The *Python Launcher* option is automatically run by *Mac OS X* whenever you double-click on an executable file containing Python code. Although this may be useful for some, at *Head First Labs* we rarely use it, but it's still nice to know it's there if we ever do need it.

The last option, *Update Shell Profile.command*, updates the configuration files on *Mac OS X* to ensure the location of the Python interpreter and its associated utilities are correctly added to your operating system's path. You can click on this option now to run this command, then forget about ever having to run it again—once is enough.

You're ready to run on Mac OS X

And with that, you're all set on *Mac OS X*.

You can now skip back to Chapter 1 and get started.

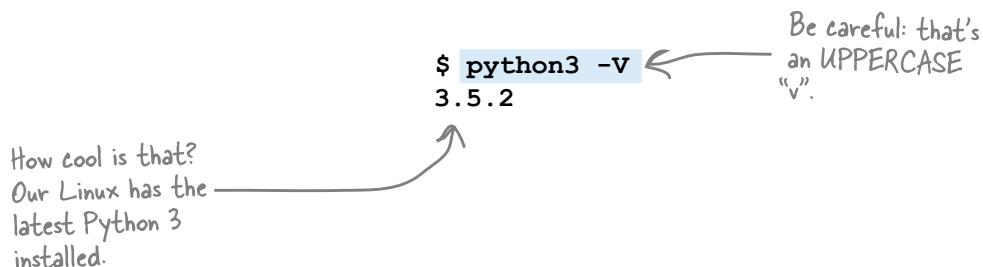


In the Dock, click the Applications icon, and then select the Python 3 folder.

Install Python 3 on Linux

If you are running a recent distribution of your favorite *Linux*, the *really great news* is that you most likely have Python 2 *and* Python 3 already installed.

Here's a quick way to ask the Python interpreter to fess up its currently installed version number; open up a command line and type:



If, after you issue this command, *Linux* complains that it can't find `python3`, you need to install a copy. How you do this depends on the *Linux* distribution you are running.

If your *Linux* is one based on the popular *Debian* or *Ubuntu* distribution (as is the one we use at *Head First Labs*), you can use the `apt-get` utility to install Python 3. Here's the command to use:

```
$ sudo apt-get install python3 idle3
```

If you are running a *yum*-based or *rpm*-based distribution, use the equivalent command for those systems. Or fire up your favorite *Linux* GUI and use your distribution's GUI-based package manager to select `python3` *and* `idle3` for installation. On many *Linux* systems, the *Synaptic Package Manager* is a popular choice here, as are any number of GUI-based software installers.

After installing Python 3, use the command from the top of this page to check that all is OK.

No matter which distribution you use, the `python3` command gives you access to the Python interpreter at the command line, whereas the `idle3` command gives you access to the GUI-based integrated development environment called *IDLE*. This is a very simple IDE that provides access to Python's `>>>` interactive prompt, a passable text editor, the Python debugger, and the Python documentation.

We'll be using the `>>>` prompt and *IDLE* a lot in this book, starting in Chapter 1, which you can flip back to now.

Be sure to select the "python3" and "idle3" packages for installation on Linux.

appendix b: pythonanywhere



* Deploying Your Webapp *

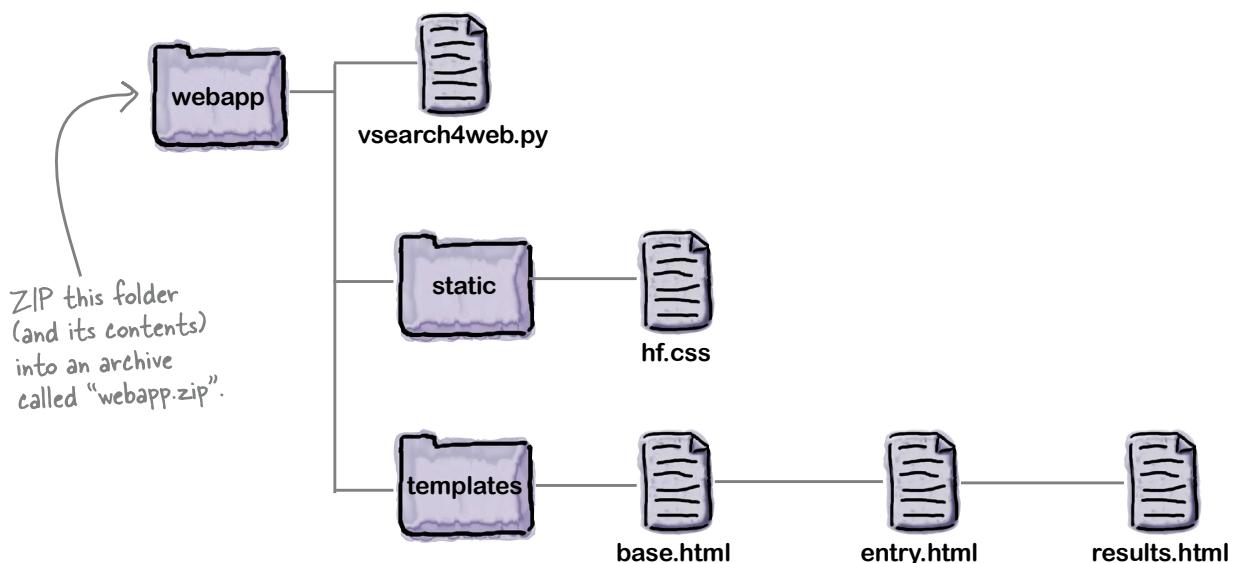


At the end of Chapter 5, we claimed that deploying your webapp to the cloud was only 10 minutes away.

It's now time to make good on that promise. In this appendix, we are going to take you through the process of deploying your webapp on *PythonAnywhere*, going from zero to deployed in about 10 minutes. *PythonAnywhere* is a favorite among the Python programming community, and it's not hard to see why: it works exactly as you'd expect it to, has great support for Python (and Flask), and—best of all—you can get started hosting your webapp at no cost. Let's check out *PythonAnywhere*.

Step 0: A Little Prep

At the moment, you have your webapp code on your computer in a folder called `webapp`, which contains the `vsearch4web.py` file and the `static` and `templates` folders (as shown below). To prepare all this stuff for deployment, create a ZIP archive file of everything in your `webapp` folder, and call the archive file `webapp.zip`:



In addition to `webapp.zip`, you also need to upload and install the `vsearch` module from Chapter 4. For now, all you need to do is locate the distribution file that you created back then. On our computer, the archive file is called `vsearch-1.0.tar.gz` and it's stored in our `mymodules/vsearch/dist` folder (on Windows, the file is likely called `vsearch-1.0.zip`).

You don't need to do anything with either archive file right now. Just make a note of where both archive files are on your computer so that they are easy to find when you upload them to *PythonAnywhere*. Feel free to grab a pencil and scribble down each archive file's location here:

Recall from Chapter 4 that Python's "setuptools" module creates ZIPs on Windows, and .tar.gz files on everything else.

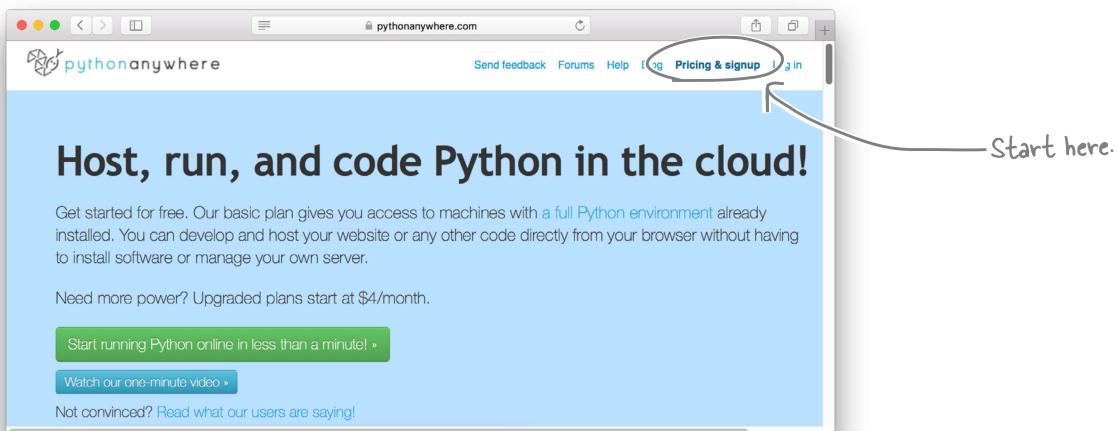
`webapp.zip`

`vsearch-1.0.tar.gz`

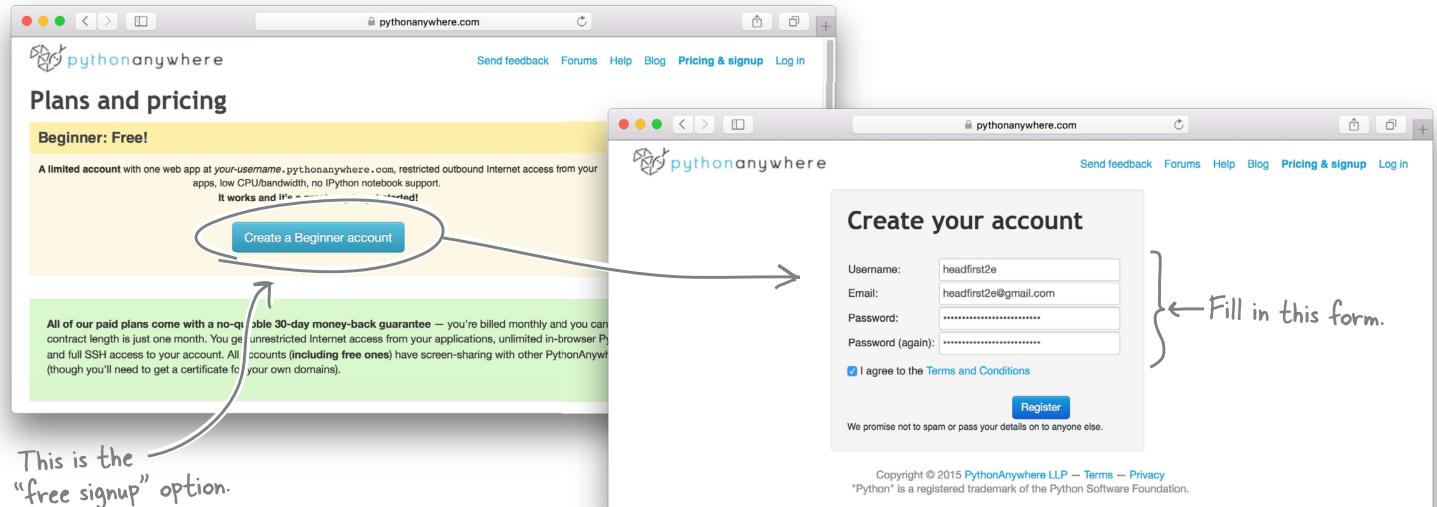
This is "vsearch.zip" instead if you're on Windows.

Step 1: Sign Up for PythonAnywhere

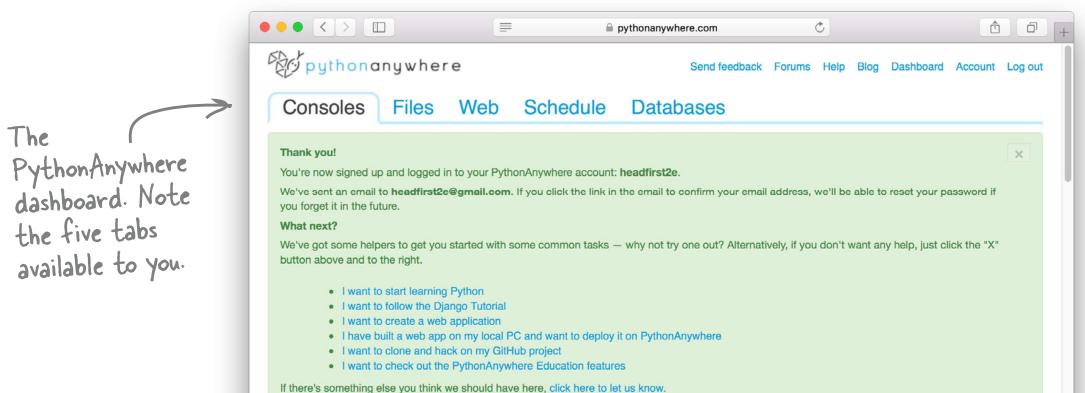
This step couldn't be any easier. Surf over to pythonanywhere.com, then click on the **Pricing & signup** link:



Click on the big, blue button to create a *Beginner account*, then fill in the details on the signup form:



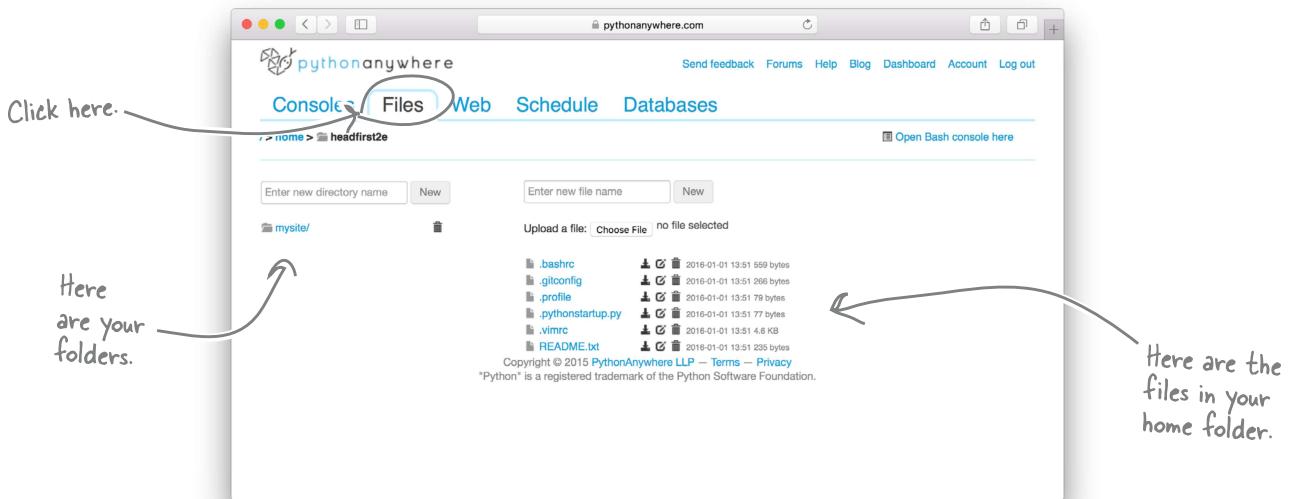
If all is well, the PythonAnywhere dashboard appears. Note: you are both registered *and* signed in at this point:



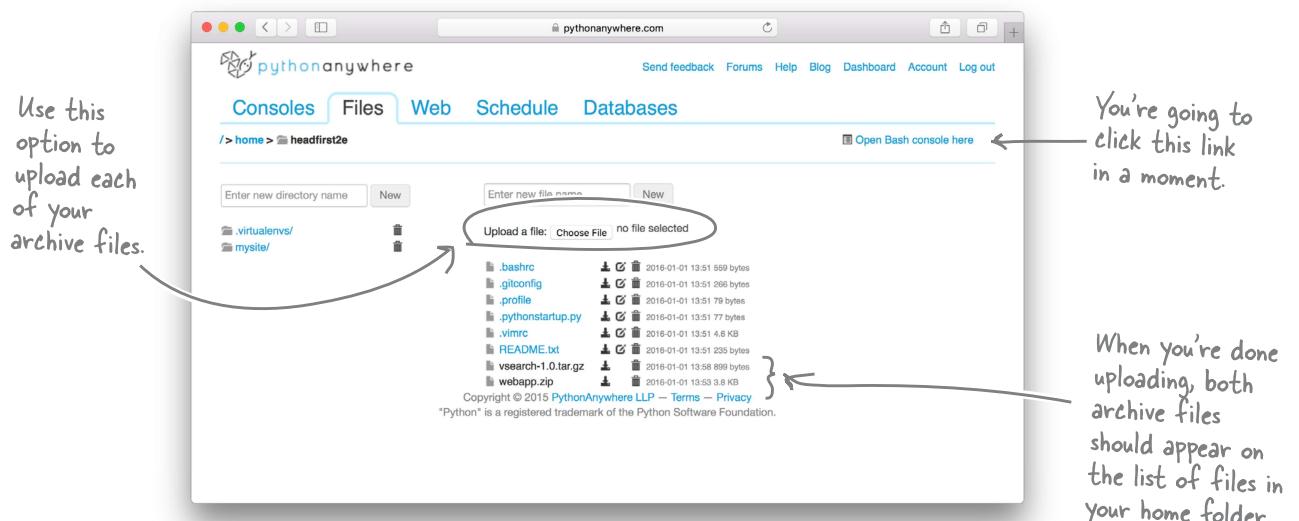
upload your code

Step 2: Upload Your Files to the Cloud

Click on the **Files** tab to view the folders and files available to you:



Use the *Upload a file* option to locate and upload the two archive files from **Step 0**:



You're now ready to extract and install these two uploaded archive files, and you'll do that during **Step 3**. To get ready, click the *Open a bash console here* link at the top right of the above page. This opens up a terminal window in your browser window (on *PythonAnywhere*).

Step 3: Extract and Install Your Code

When you click the *Open a bash console here* link, *PythonAnywhere* responds by replacing the *Files* dashboard with a browser-based Linux console (command prompt). You're going to issue a few commands to extract and install the `vsearch` module as well as your webapp's code within this console. Begin by installing `vsearch` into Python as a "private module" (i.e., just for your use) using this command (be sure to use `vsearch-1.0.zip` if you're on *Windows*):

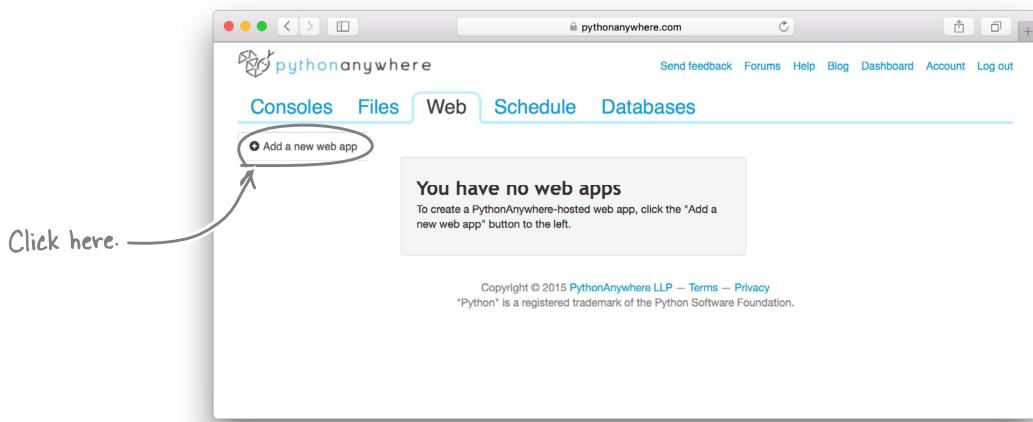
```
python3 -m pip install vsearch-1.0.tar.gz --user
Run the command.
Success!
" --user " ensures
the "vsearch"
module is installed
for your use only.
PythonAnywhere
does not allow you
to install a module
for everyone's use
(just your own).
13:59 ~ $ ls
README.txt mysite vsearch-1.0.tar.gz webapp.zip
14:00 ~ $ python3 -m pip install vsearch-1.0.tar.gz --user
Processing ./vsearch-1.0.tar.gz
Building wheels for collected packages: vsearch
  Running setup.py bdist_wheel for vsearch
    Stored in directory: /home/headfirst2e/.cache/pip/wheels/eb/fc/ad/734c9fb2d6fcf97f96f045b42e63d536eb2a1f47
191735fce
Successfully built vsearch
Installing collected packages: vsearch
Successfully installed vsearch
14:00 ~ $
```

With the `vsearch` module successfully installed, it's time to turn your attention to your webapp's code, which has to be installed into the `mysite` folder (which already exists on your *PythonAnywhere* home folder). To do this, you need to issue two commands:

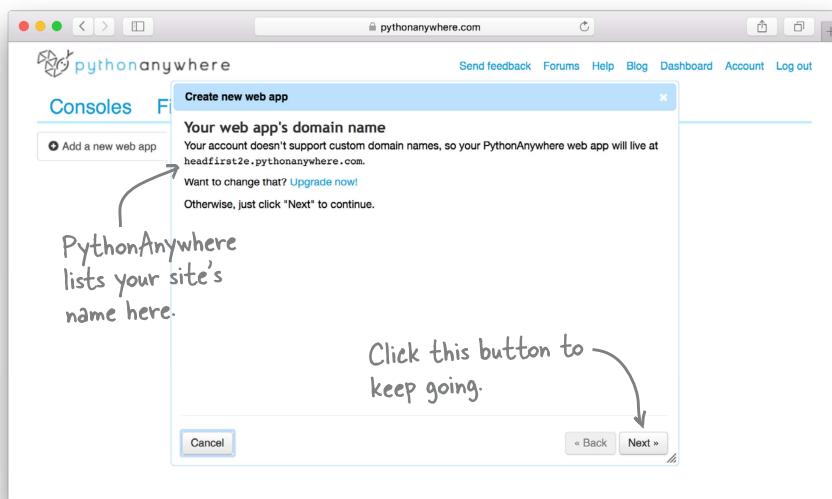
```
Unpack your webapp's code...
You should see messages similar to these.
...then move the code into the "mysite" folder.
unzip webapp.zip
mv webapp/* mysite
13:59 ~ $ Building wheels for collected packages: vsearch
  Running setup.py bdist_wheel for vsearch
    Stored in directory: /home/headfirst2e/.cache/pip/wheels/eb/fc/ad/734c9fb2d6fcf97f96f045b42e63d536eb2a1f47
191735fce
Successfully built vsearch
Installing collected packages: vsearch
Successfully installed vsearch
14:00 ~ $ unzip webapp.zip
Archive: webapp.zip
  creating: webapp/
    inflating: webapp/.DS_Store
    creating: __MACOSX/
    creating: __MACOSX/webapp/
    inflating: __MACOSX/webapp/._DS_Store
    creating: webapp/static/
    inflating: webapp/static/hf.css
    creating: webapp/templates/
    inflating: webapp/templates/base.html
    inflating: webapp/templates/entry.html
    inflating: webapp/templates/results.html
    inflating: webapp/vsearch4web.py
14:00 ~ $ mv webapp/* mysite/
14:00 ~ $
```

Step 4: Create a Starter Webapp, 1 of 2

With **Step 3** done, return to the *PythonAnywhere* dashboard and select the **Web** tab, where *PythonAnywhere* invites you to create a new starter webapp. You'll do this, then swap out the starter's webapp code for your own. Note that each *Beginner account* gets one webapp for free; if you want more, you'll have to upgrade to a paid account. Luckily—for now—you only need the one, so let's keep going by clicking *Add a new web app*:



As you are using a free account, your webapp is going to run on the site name shown on the next screen. Click the *Next* button to proceed with *PythonAnywhere*'s suggested site name:



Click *Next* to continue with this step.

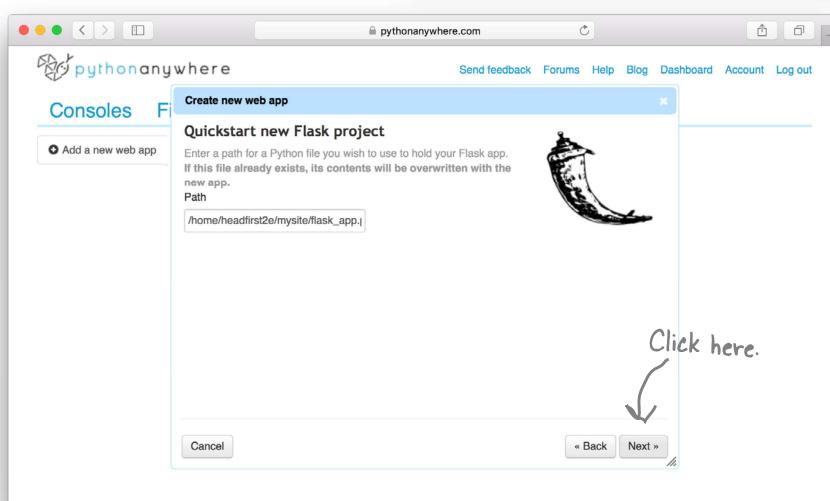
Step 4: Create a Starter Webapp, 2 of 2

PythonAnywhere supports more than one Python web framework, so the next screen offers you a choice among the many supported systems. Pick Flask, then select the version of Flask and Python you wish to deploy to. As of this writing, Python 3.4 and Flask 0.10.1 are the most up-to-date versions supported by *PythonAnywhere*, so go with that combination unless a newer combination is offered (in which case, pick the newer one instead):

Select "Flask" for your webapp, then choose the most up-to-date Python/Flask combination.

You don't need to click "Next" here. As soon as you choose the combination you want, this screen appears.

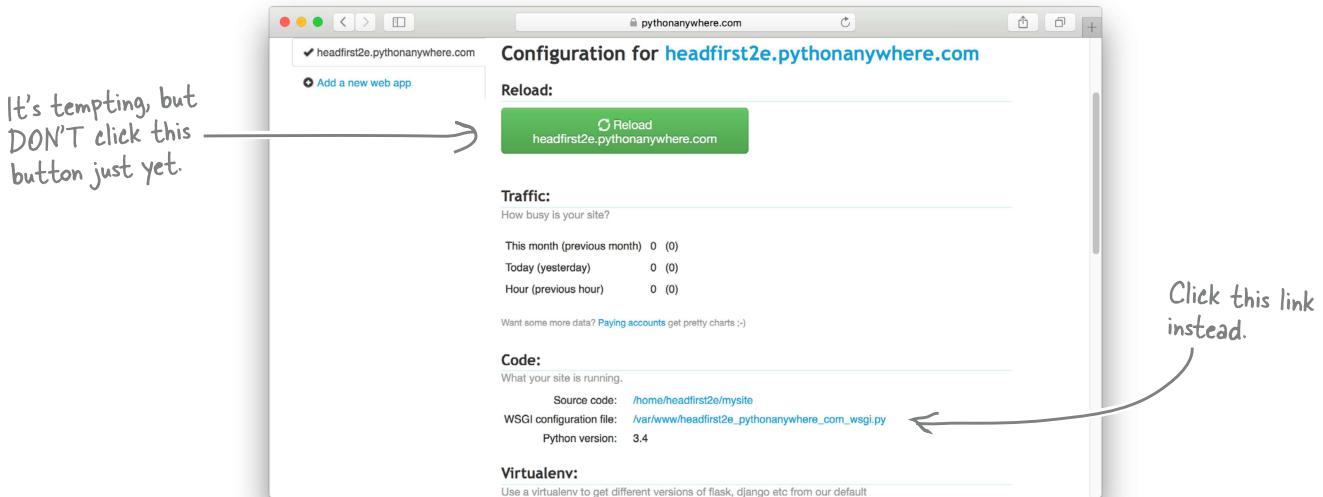
You're nearly there. The next screen offers to create a quickstart Flask webapp. Go ahead and do that now by accepting the values on this page and clicking on the *Next* button to continue:



import your module

Step 5: Configure Your Webapp

With **Step 4** complete, you are presented with the **Web** dashboard. Don't be tempted to click that big, green button just yet—you haven't told *PythonAnywhere* about your code yet, so hold off on running anything for now. Instead, click in the long link to the right of the *WSGI configuration file* label:



Clicking that long link loads your newly created Flask webapp's configuration file into *PythonAnywhere*'s web-based text editor. At the end of Chapter 5, we told you that *PythonAnywhere* imports your webapp code before invoking `app.run()` for you. This is the file that supports that behavior. However, it needs to be told to reference *your* code, not the code in the starter app, so you need to edit the last line of this file (as shown below), and then click *Save*:

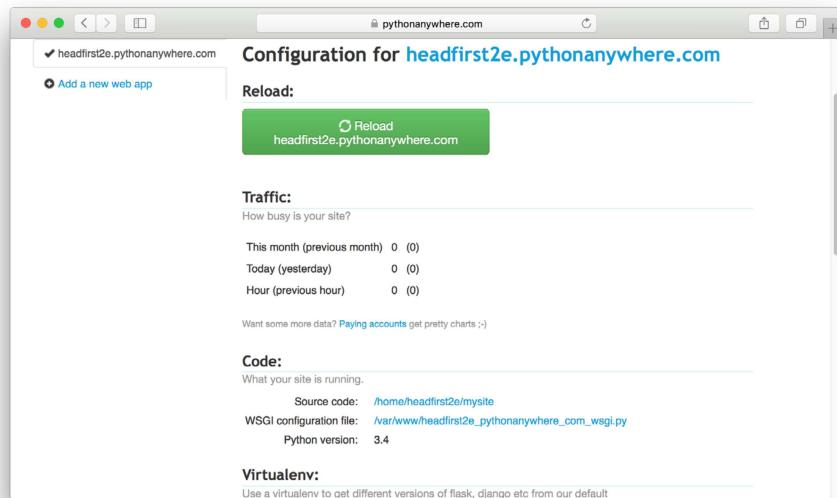
```
1 # This file contains the WSGI configuration required to serve up your
2 # web application at http://<your-username>.pythonanywhere.com/
3 # It works by setting the variable 'application' to a WSGI handler of some
4 # description.
5 #
6 # The below has been auto-generated for your Flask project
7
8 import sys
9
10 # add your project directory to the sys.path
11 project_home = u'/home/headfirst2e/mysite'
12 if project_home not in sys.path:
13     sys.path = [project_home] + sys.path
14
15 # import flask app but need to call it "application" for WSGI to work
16 from flask import app as application
17 |
```

The "Save" button

Change the last line of this file to reference your "vsearch4web" module.

Step 6: Take Your Cloud-Based Webapp for a Spin!

Be sure to save your changed configuration file, then return to the **Web** tab on the dashboard. It is now time to click on that big, tempting, green button. Go for it!



After but a brief moment, your webapp appears in your browser, and it works exactly as it did when you ran it locally, only now anybody with an Internet connection and a web browser can use it too:

And with that, you're done. The webapp you developed in Chapter 5 has been deployed to *PythonAnywhere*'s cloud (in less than 10 minutes). There's lots more to *PythonAnywhere* than what's shown in this short appendix, so feel free to explore and experiment. At some point, remember to return the *PythonAnywhere* dashboard and log out. Note that, despite your logging out, your webapp keeps running in the cloud until you tell it otherwise. That's pretty cool, isn't it?

appendix c: top ten things we didn't cover

There's Always More to Learn

I think we have a problem here.
There are a whole bunch of
things they didn't cover.



It was never our intention to try to cover everything.

This book's goal was always to show you enough Python to get you up to speed as quickly as possible. There's a lot more we could've covered, but didn't. In this appendix, we discuss the top 10 things that—given another 600 pages or so—we would've eventually gotten around to. Not all of the 10 things will interest you, but quickly flip through them just in case we've hit on your sweet spot, or provided an answer to that nagging question. All the programming technologies in this appendix come baked in to Python and its interpreter.

1. What About Python 2?

As of this book's publication date (late 2016) there are two mainstream flavors of Python in widespread use. You already know quite a bit about **Python 3**, as that's the flavor you've used throughout this book.

All new language developments and enhancements are being applied to Python 3, which is on a 12- to 18-month minor release cycle. Release 3.6 is due before 2016 ends, and you can expect 3.7 to arrive late in 2017 or early in 2018.

Python 2 has been “stuck” at release 2.7 for some time now. This has to do with the fact that the *Python core developers* (the people who guide the development of Python) decided that Python 3 was the future, and that Python 2 should quietly go away. There were solid technical reasons for this approach, but no one really expected things to take so long. After all, Python 3—the future of the language—first appeared in late 2008.

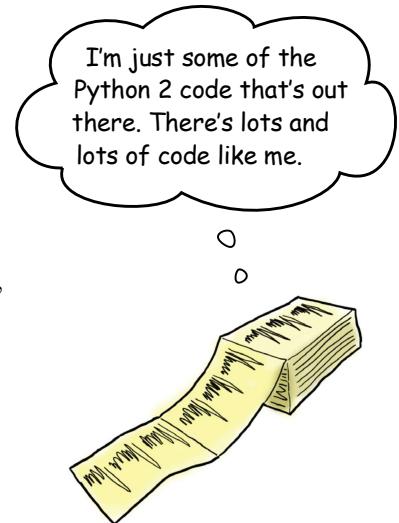
An entire book could be written on what's happened since late 2008 until now. Suffice it to say, Python 2 stubbornly refused to go away. There was (and still is) a huge installed base of Python 2 code and developers, with some domains dragging their heels when it comes to upgrading. There's a very simple reason for why this is: Python 3 introduced a handful of enhancements that broke backward compatibility. Put another way: there's lots of Python 2 code that will not run *unchanged* in Python 3 (even though, at a first glance, it can be hard to tell Python 2 code from Python 3 code). Also, many programmers simply believed Python 2 was “good enough,” and didn't upgrade.

Recently (over the last year), there's been a sea change. The switching rate from 2 to 3 appears to be increasing. Some very popular third-party modules have released Python 3-compatible versions, and this is having a positive effect on Python 3 adoption. Additionally, the *Python core developers* keep adding extra goodness to Python 3, making it a more attractive programming language over time. The practice of “backporting” the cool new features from 3 to 2 has stopped with 2.7, and although bug and security fixes are still being applied, the *Python core developers* have announced that this activity will stop in 2020. The clock is ticking for Python 2.

Here's the common advice offered when you're trying to decide whether 3 or 2 is right for you:

If you're starting a new project, use Python 3.

You need to resist the urge to create more legacy code in Python 2, especially if you're starting with a blank slate. If you have to maintain some existing Python 2 code, what you know about 3 carries over: you'll certainly be able to read the code and understand it (it's still Python, regardless of the major version number). If there are technical reasons why the code has to remain running in Python 2, then so be it. If, however, the code can be ported to Python 3 without too much fuss, then we believe the gain is worth the pain, as Python 3 *is* the better language, and *is* the future.



2. Virtual Programming Environments

Let's imagine you have two clients, one with Python code that relies on one version of a third-party module, and another that relies on a *different* version of the same third-party module for their code. And, of course, you're the poor soul who has to maintain both projects' code.

Doing so on one computer can be problematic, as the Python interpreter doesn't support the installation of different versions of third-party modules.

That said, help is at hand thanks to Python's notion of virtual environments.

A **virtual environment** lets you create a new, clean Python environment within which you can run your code. You can install third-party modules into one virtual environment without impacting another, and you can have as many virtual environments as you like on your computer, switching between them by *activating* the one you want to work on. As each virtual environment can maintain its own copy of whatever third-party modules you wish to install, you can use two different virtual environments, one for each of your client projects discussed above.

Before doing so, however, you have to make a choice: use the virtual environment technology, called `venv`, that ships with Python 3's *standard library*, or install the `virtualenv` module from PyPI (which does the same thing as `venv`, but has more bells and whistles). It's best if you make an informed choice.

To learn more about `venv`, check out its documentation page:

<https://docs.python.org/3/library/venv.html>

To find out what `virtualenv` offers over and above `venv`, start here:

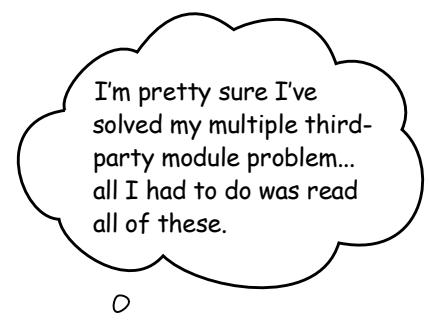
<https://pypi.org/project/virtualenv/>

Whether you use virtual environments for your projects is a personal choice. Some programmers swear by them, refusing to write any Python code unless it's within a virtual environment. This may be a bit of an extreme stance, but to each their own.

We chose not to cover virtual environments in the main body of this book. We feel virtual environments are—if you need them—a total godsend, but we don't yet believe every Python programmer needs to use one for everything they do.

We recommend you slowly back away from people who say that you aren't a proper Python programmer *unless* you use `virtualenv`.

All he had to do was use
a virtual environment.



everything's an object

3. More on Object Orientation

If you've read through this entire book, by now you'll (hopefully) appreciate what's meant by this phrase: "In Python, everything's an object."

Python's use of objects is great. It generally means that things work the way you expect them to. However, the fact that everything's an object does **not** mean that everything has to belong to a class, especially when it comes to your code.

In this book, we didn't learn how to create our own class until we needed one in order to create a custom context manager. Even then, we only learned as much as was needed, and nothing more. If you've come to Python from a programming language that insists all your code resides in a class (with *Java* being the classic example), the way we've gone about things in this book may be disconcerting. Don't let this worry you, as Python is much less strict than *Java* (for instance) when it comes to how you go about writing your programs.

If you decide to create a bunch of functions to do the work you need to do, then have at it. If your brain thinks in a more functional way, Python can help here too with the comprehension syntax, tipping its hat to the world of functional programming. And if you can't get away from the fact that your code needs to reside in a class, Python has full-featured object-oriented-programming syntax built right in.

If you do end up spending a lot of time creating classes, check out the following:

- `@staticmethod`: A decorator that lets you create a static function within a class (which does not receive `self` as its first argument).
- `@classmethod`: A decorator that lets you create a class method that expects a class as its first object (usually referred to as `cls`), not `self`.
- `@property`: A decorator that allows you to redesignate and use a method as if it were an attribute.
- `__slots__`: A class directive that (when used) can greatly improve the memory efficiency of the objects created from your class (at the expense of some flexibility).

To learn more about any of these, consult the Python docs (<https://docs.python.org/3/>). Or check out some of our favorite Python books (discussed in the next appendix).

OK, chaps...let's think about this for a moment. Does that code really need to be in a class?



4. Formats for Strings and the Like

The recurring example application used in this book displayed its output in a web browser. This allowed us to defer any output formatting to HTML (specifically, we used the *Jinja2* module included with *Flask*). In doing so, we sidestepped one area where Python shines: text-based string formatting.

Let's say you have a string that needs to contain values that won't be known until your code runs. You want to create a message (`msg`) that contains the values so you can perform some later processing (perhaps you're going to print the message on screen, include the message within an HTML page you're creating with *Jinja2*, or tweet the message to your 3 million followers). The values your code generates at runtime are in two variables: `price` (the price of the item in question) and `tag` (a catchy marketing tagline). You have a few options here:

- Build the message you need using concatenation (the `+` operator).
- Use old-style string formats (using the `%` syntax).
- Take advantage of every string's `format` method to build your message.

Here's a short `>>>` session showing each of these techniques in action (bearing in mind that you, having worked through this book, already concur with what the generated message is telling you):

```

Python 3.5.2 Shell
>>>
>>> price = 49.99
>>> tag = 'is a real bargain!'
>>>
>>> msg = 'At ' + str(price) + ', Head First Python ' + tag
>>> msg
'At 49.99, Head First Python is a real bargain!'
>>>
>>> msg = 'At %2.2f, Head First Python %s' % (price, tag)
>>> msg
'At 49.99, Head First Python is a real bargain!'
>>>
>>> msg = 'At {}, Head First Python {}'.format(price, tag)
>>> msg
'At 49.99, Head First Python is a real bargain!'
>>> |

```

You already knew this, right? 😊

The `%s` and `%f` format specifiers are as old as the hills...but, hey, like me, they still work.



Which of these techniques you use is a personal preference, although there's a bit of a push on to encourage the use of the `format` method over the other two (see *PEP 3101* at <https://www.python.org/dev/peps/pep-3101/>). You'll find code in the wild that uses one technique over the other, and sometimes (and not at all helpfully) mixes all three. To learn more, start here:

<https://docs.python.org/3/library/string.html#format-specification-mini-language>

the sorted BIF rocks

5. Getting Things Sorted

Python has wonderful built-in sorting capabilities. Some of the built-in data structures (lists, for example) contain `sort` methods that can be used to perform in-place ordering of your data. However, it is the `sorted` BIF that makes Python truly special (as this BIF works with *any* of the built-in data structures).

In the IDLE session below, we first define a small dictionary (`product`), which we then process with a succession of `for` loops. The `sorted` BIF is exploited to control the order in which each `for` loop receives the dictionary's data. Follow along on your computer while you read the annotations:

**BIF is short
for "built-in
function."**

The screenshot shows a Python 3.5.2 Shell window with four code snippets demonstrating different ways to sort a dictionary.

- Example 1:** Prints the raw dictionary. The output is:
PDF -> 43.99
Book -> 49.99
Video -> 199.99
- Example 2:** Prints the dictionary sorted by key. The output is:
Book -> 49.99
PDF -> 43.99
Video -> 199.99
- Example 3:** Prints the dictionary sorted by value using `key=product.get`. The output is:
PDF -> 43.99
Book -> 49.99
Video -> 199.99
- Example 4:** Prints the dictionary sorted by value using `key=product.get, reverse=True`. The output is:
Video -> 199.99
Book -> 49.99
PDF -> 43.99

Annotations explain the behavior:

- "Print out the dictionary on screen." points to the first example.
- "The raw data looks sorted by value, but isn't really (this is more luck than anything else)." points to the second example.
- "The output is ordered from lowest to highest price." points to the third example.
- "The output is now ordered from highest to lowest price." points to the fourth example.
- "Define the dictionary (remember: insertion order is *not* maintained)" points to the first line of the first example.
- "Adding a call to "sorted" sorts the dictionary by key, which may or may not be what you want here." points to the second example.
- ""B" comes before "P", which comes before "V". Sorting by keys—the default—works." points to the third example.
- "The addition of the "key" argument lets you sort by value." points to the fourth example.
- "Adding the "reverse" argument flips the sort order." points to the fourth example.

Ln: 284 Col: 4

Learn more about how to sort with Python from this wonderful *HOWTO*:

<https://docs.python.org/3/howto/sorting.html#sortinghowto>

6. More from the Standard Library

Python's *standard library* is full of goodness. It's always a worthy exercise to take 20 minutes every once in a while to review what's available, starting here:

<https://docs.python.org/3/library/index.html>

If what you need is in the *standard library*, don't waste your precious time rewriting it. Use (and/or extend) what's already available. In addition to the Python docs, *Doug Hellmann* has ported his popular *Module of the Week* material over to Python 3. Find Doug's excellent material here:

<https://pymotw.com/3/>

We've reviewed a few of our favorite *standard library* modules below. Note that we can't stress enough how important it is to know what's in the *standard library*, as well as what all the provided modules can do for you.

collections

This module provides importable data structures, over and above the built-in list, tuple, dictionary, and set. There's lots to like in this module. Here's an abbreviated list of what's in `collections`:

- `OrderedDict`: A dictionary that maintains insertion order.
- `Counter`: A class that makes counting things almost too easy.
- `ChainMap`: Combines one or more dictionaries and makes them appear as one.



itertools

You already know Python's `for` loop is great, and when reworked as a comprehension, looping is crazy cool. This module, `itertools`, provides a large collection of tools for building custom iterations. This module has a lot to offer, but be sure to also check out `product`, `permutations`, and `combinations` (and once you do, sit back and thank your lucky stars you didn't have to write any of that code).

functools

The `functools` library provides a collection of higher-order functions (functions that take function objects as arguments). Our favorite is `partial`, which lets you "freeze" argument values to an existing function, then invoke the function with a new name of your choosing. You won't know what you're missing until you try it.

doing many things

7. Running Your Code Concurrently

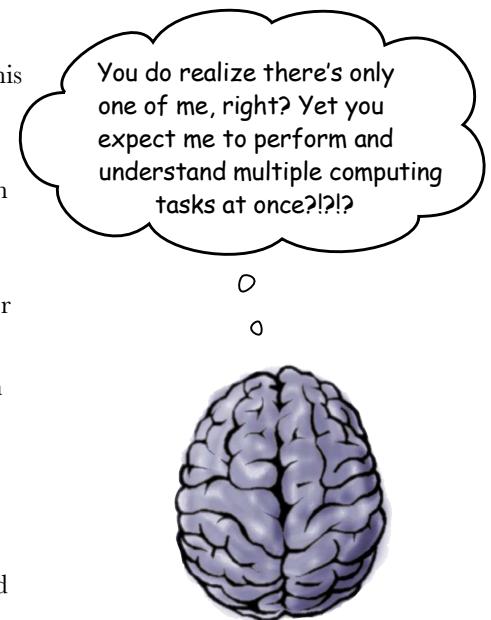
In Chapter 11^{3/4}, you used a thread to solve a waiting problem. Threads are not the only game in town when it comes to running code concurrently within your programs, although, to be honest, threads are the most used and abused of all of the available techniques. In this book, we deliberately kept our use of threads as simple as possible.

There are other technologies available to you when you find yourself in a situation where your code has to do more than one thing at once. Not every program needs these types of services, but it is nice to know that Python has a bunch of choices in this area should the need arise.

In addition to the `threading` module, here are some modules worth checking out (and we also refer you back one page to #6, as *Doug Hellmann* has some great posts on some of these modules):

- `multiprocessing`: This module allows you to spawn multiple Python processes, which—if you have more than one CPU core—can spread your computational load across many CPUs.
- `asyncio`: Lets you specify concurrency via the creation and specification of coroutines. This is a relatively new addition to Python 3, so—for many programmers—it's a very new idea (and the jury is still out).
- `concurrent.futures`: Lets you manage and run a collection of tasks concurrently.

Which of these is right for you is a question you'll be able to answer once you've tried each of them with some of your code.



New keywords: `async` and `await`

The `async` and `await` keywords were added in Python 3.5, and provide a standard way to create coroutines.

The `async` keyword can be used in front of the existing `for`, `with`, and `def` keywords (with the `def` usage receiving the most attention to date). The `await` keyword can be used in front of (almost) any other code. As of the end of 2016, `async` and `await` are very new, and Python programmers the world over are only just beginning to explore what they can do with them.

The Python docs have been updated with information on these new keywords, but, for our money, you'll find the best descriptions of their use (and the craziness that using them induces) by searching *YouTube* for anything on the topic by *David Beazley*.

Be warned: David's talks are always excellent, but do tend to lean toward the more advanced topics in the Python language ecosystem.

David's talks on Python's *GIL* are regarded as classics by many, and his books are great too; more on this in *Appendix E*.



8. GUIs with Tkinter (and Fun with Turtles)

Python comes with a complete library called `tkinter` (the *Tk interface*) for building cross-platform GUIs. You may not realize it, but you've been using an application from the very first chapter of this book that is built with `tkinter`: IDLE.

What's neat about `tkinter` is that it comes preinstalled (and ready for use) with every Python installation that includes IDLE (i.e., nearly all of them). Despite this, `tkinter` doesn't receive the use (and love) it deserves, as many believe it to be unnecessarily clunky (compared to some third-party alternatives). Nevertheless, and as IDLE demonstrates, it is possible to produce useful and usable programs with `tkinter`. (Did we mention that `tkinter` comes preinstalled and ready for use?)

One such usage is the `turtle` module (which is also part of the *standard library*). To quote the Python docs: *Turtle graphics is a popular way for introducing programming to kids. It was part of the original Logo programming language developed by Wally Feurzig and Seymour Papert in 1966.* Programmers (i.e., mainly kids, but fun for newbies, too) can use commands like `left`, `right`, `pendown`, `penup`, and so on to draw on a GUI canvas (provided by `tkinter`).

Here's a small program, which has been adapted ever so slightly from the example that comes with the `turtle` docs:

As well as showing "turtle" in action, this small program also demonstrates the use of Python's "while" loop and "break" statement. They work exactly as you'd expect them to, but don't see nearly as much action as the "for" loop and comprehensions.

```
tksample.py - /Users/paul/Documents/tksample.py (3.5.2)

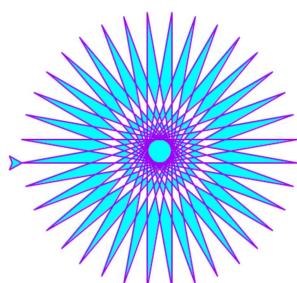
from turtle import *
color('purple', 'cyan')
begin_fill()

while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break

end_fill()
done()

Ln: 13 Col: 0
```

And when this small `turtle` program is executed, a thing of beauty is drawn and appears on screen:



We know you can do better than this, so why not give "turtle" a try?

test test test

9. It's Not Over 'Til It's Tested

This book has barely mentioned automated testing, aside from a passing nod to the `py.test` tool for checking conformance to *PEP 8* (at the end of Chapter 4). This is not because we think automated testing isn't important. **We think automated testing is very important.** It is such an important topic that entire books are dedicated to it.

That said, in this book, we avoided automated testing tools on purpose. This has nothing to do with how we feel about automated testing (it really *is* very important). However, when you are first learning to program in a new programming language, introducing automated testing can confuse more than it clarifies, as the creation of tests assumes a good understanding of the thing being tested, and if that "thing" happens to be a new programming language that you're learning...well, you can see where we're going with this, can't you? It's a bit like the chicken and the egg. Which comes first: learning to code, or learning how to test the code you're learning?

Of course, now that you're a bona-fide Python programmer, you can take the time to understand how Python's *standard library* makes it easy to test your code. There are two modules to look at (and consider):

- `doctest`: This module lets you embed your tests in your module's docstrings, which isn't as weird as it sounds and *is* very useful.
- `unittest`: You may have already used a "unittest" library with another programming language, and Python comes with its very own version (which works exactly as you'd expect it to).

The `doctest` module is adored by those who use it. The `unittest` module works like most other "unittest" libraries in other languages, and a lot of Python programmers complain that it's not *pythonic* enough. This has led to the creation of the hugely popular `py.test` (which we talk more about in the next appendix).



10. Debug, Debug, Debug

You'd be forgiven for thinking that the vast majority of Python programmers revert to adding `print` calls to their code when something goes wrong. And you wouldn't be far off: it's a popular debugging technique.

Another is experimenting at the `>>>` prompt, which—if you think about it—is very like a debugging session *without* the usual debugging chores of watching traces and setting up breakpoints. It is impossible to quantify how productive the `>>>` prompt makes Python programmers. All we know is this: if a future release of Python decides to remove the interactive prompt, things will get ugly.

If you have code that's not doing what you think it should, and the addition of `print` calls as well as experimenting at the `>>>` prompt have left you none the wiser, consider using Python's included debugger: `pdb`.

It's possible to run the `pdb` debugger directly from your operating system's terminal window, using a command like this (where `myprog.py` is the program you need to fix):

```
python3 -m pdb myprog.py
```

It's also possible to interact with `pdb` from the `>>>` prompt, which is as close an instantiation of "the best of both worlds" as we think you'll ever come across. The details of how this works, as well as a discussion of all the usual debugger commands (set a breakpoint, skip, run, etc.) are in the docs:

<https://docs.python.org/3/library/pdb.html>

The `pdb` technology is not an "also ran," nor was it an afterthought; it's a wonderfully feature-full debugger for Python (and it comes built-in).

You can learn all about traces and breakpoints by working through the "pdb" docs.

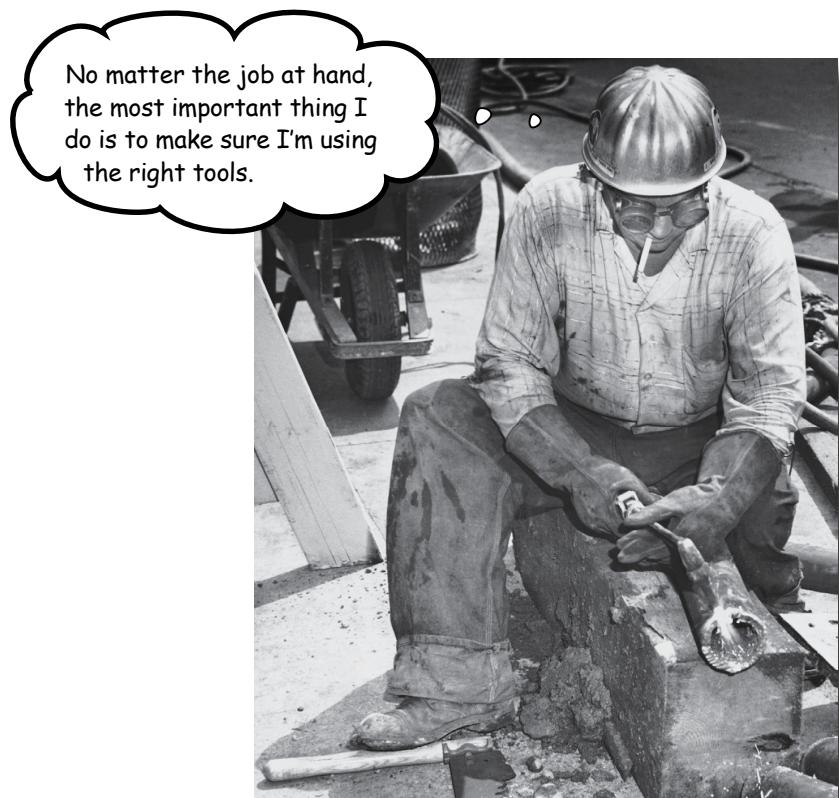
As always, Windows users need to use "py -3" instead of "Python3". (That's "py", space, then minus 3).

Make sure a working understanding of Python's "pdb" debugger is part of your toolkit.



appendix d: top ten projects not covered

Even More Tools, Libraries, and Modules



We know what you're thinking as you read this appendix's title.

Why on Earth didn't they make the title of the last appendix: *The Top Twenty Things We Didn't Cover?* Why another 10? In the last appendix, we limited our discussion to stuff that comes baked in to Python (part of the language's "batteries included"). In this appendix, we cast the net much further afield, discussing a whole host of technologies that are available to you because Python exists. There's lots of good stuff here and—just like with the last appendix—a quick perusal won't hurt you *one single bit*.

ipython ipython ipython

1. Alternatives to >>>

Throughout this book we've happily worked at Python's built-in >>> prompt, either from within a terminal window or from within IDLE. In doing so, we hope we've demonstrated just how effective using the >>> prompt can be when you're experimenting with ideas, exploring libraries, and trying out code.

There are lots of alternatives to the built-in >>> prompt, but the one that gets the most attention is called `ipython`, and if you find yourself wishing you could do more at the >>> prompt, `ipython` is worth a look. It is very popular with many Python programmers, but is *especially* popular within the scientific community.

To give you an idea of what `ipython` can do compared to the plain ol' >>> prompt, consider this short interactive `ipython` session:

The screenshot shows a Mac OS X terminal window titled "paul — IPython: Users/paul — ipython — 79x19". The session starts with the system login information and the Python version. Then, it shows the help documentation for IPython 5.1.0. Following this, three lines of code are entered and executed:

```
[In [1]:] nums = [1, 2, 3]
[In [2]:] squares = [n*n for n in nums]
[In [3]:] squares
Out[3]: [1, 4, 9]
```

Annotations with arrows point to specific parts of the session:

- An arrow points from the text "With 'ipython' installed, start it at your operating system's command line." to the terminal window title bar.
- An arrow points from the text "Your code is color-coded." to the color-coded syntax of the input and output lines.
- An arrow points from the text "You can easily tell which output goes with which input (thanks to numbered prompts)." to the numbered prompts [1], [2], and [3].

Find out more about `ipython` at <https://ipython.org>.

There are other >>> alternatives, but the only other one that's a match (in our view) for what `ipython` has to offer is `ptpython` (more information can be found here: <https://pypi.org/project/ptpython/>). If you like working within a text-based terminal window, but are looking for something a bit more "full screen" than `ipython`, take a look at `ptpython`. You won't be disappointed.

As with all third-party modules, you can use "pip" to install both "ipython" and "ptpython".

2. Alternatives to IDLE

We're not afraid to state this: we have a soft spot for *IDLE*. We really like the fact that Python not only comes with a capable >>> prompt, but also ships with a passable cross-platform GUI-based editor and debugger. There are few other mainstream programming languages that provide anything similar as part of their default install.

Regrettably, *IDLE* gets a fair amount of flack in the Python community, as it stacks up poorly against some of the more capable “professional” offerings. We think this is an *unfair* comparison, as *IDLE* was never designed to compete in that space. *IDLE*'s main goal is to get new users up and going as quickly as possible, and it does this *in spades*. Consequently, we feel *IDLE* should be celebrated more in the Python community.

IDLE aside, if you need a more professional IDE, you have choices. The most popular in the Python space include:

- *Eclipse*: <https://www.eclipse.org>
- *PyCharm*: <https://www.jetbrains.com/pycharm/>
- *WingWare*: <https://wingware.com>

Eclipse is a completely open source technology, so won't cost you more than the download. If you're already an *Eclipse* fan, its support for Python is very good. But, if you aren't currently using *Eclipse*, we wouldn't recommend its use to you, due to the existence of *PyCharm* and *WingWare*.

Both *PyCharm* and *WingWare* are commercial products, with “community versions” available for download at no cost (but with some restrictions). Unlike *Eclipse*, which targets many programming languages, both *PyCharm* and *WingWare* target Python programmers specifically and, like all IDEs, have great support for project work, links to source code management tools (like *git*), support for teams, links to the Python docs, and so on. We encourage you to try both, then make your choice.

If IDEs aren't for you, fear not: all of the world's major text editors offer excellent language support to Python programmers.

What does Paul use?

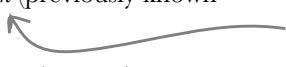
Paul's text editor of choice is *vim* (Paul uses *MacVim* on his development machines). When working on Python projects, Paul supplements his use of *vim* with *ptpython* (when experimenting with code snippets), and he's also a fan of *IDLE*. Paul uses *git* for local version control.

For what it's worth, Paul doesn't use a full-featured IDE, but his students love *PyCharm*. Paul also uses (and recommends) *Jupyter Notebook*, which is discussed next.



jupyter jupyter jupyter

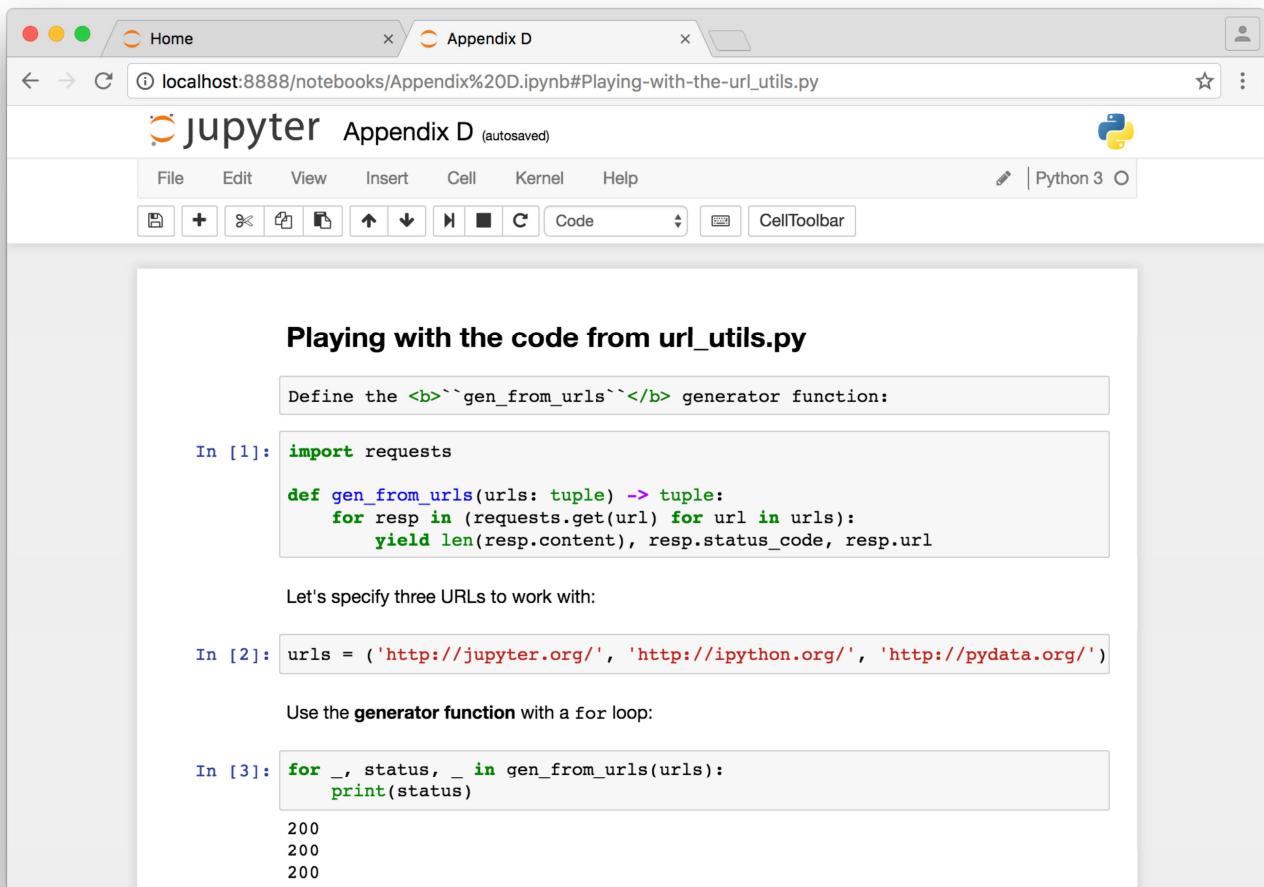
3. Jupyter Notebook: The Web-Based IDE

In item #1, we drew your attention to ipython (which is an excellent >>> alternative). From the same project team comes *Jupyter Notebook* (previously known as *iPython Notebook*). 

Jupyter Notebook can be described as the power of ipython in an interactive web page (which goes by the generic name of “notebook”). What’s amazing about *Jupyter Notebook* is that your code is editable and runnable from within the notebook, and—if you feel the need—you can add text and graphics, too.

Here’s some code from Chapter 12 running within a *Jupyter Notebook*. Note how we’ve added textual descriptions to the notebook to indicate what’s going on:

The next generation of Jupyter Notebook is called Jupyter Lab, and it was in “alpha” as work on this book was concluding. Keep an eye out for the Jupyter Lab project: it’s going to be something rather special.



The screenshot shows a Jupyter Notebook interface running in a web browser. The title bar says "localhost:8888/notebooks/Appendix%20D.ipynb#Playing-with-the-url_utils.py". The main window has a toolbar with File, Edit, View, Insert, Cell, Kernel, Help, and a Python 3 icon. Below the toolbar is a CellToolbar with various icons. The notebook content starts with a text cell titled "Playing with the code from url_utils.py" containing the instruction "Define the **gen_from_urls** generator function:". This is followed by a code cell (In [1]) with the following Python code:

```
import requests

def gen_from_urls(urls: tuple) -> tuple:
    for resp in (requests.get(url) for url in urls):
        yield len(resp.content), resp.status_code, resp.url
```

Next is a text cell asking "Let's specify three URLs to work with:" followed by a code cell (In [2]) with the URL list:

```
urls = ('http://jupyter.org/', 'http://ipython.org/', 'http://pydata.org/')
```

Then is a text cell asking "Use the generator function with a for loop:" followed by a code cell (In [3]) with the code:

```
for _, status, _ in gen_from_urls(urls):
    print(status)
```

The output of this cell is:

```
200  
200  
200
```

Learn more about *Jupyter Notebook* from its website (<http://jupyter.org>), and use pip to install it onto your computer, then start exploring. You will be glad you did. *Jupyter Notebook* is a killer Python application.

4. Doing Data Science

When it comes to Python adoption and usage, there's one domain that continues to experience explosive growth: the world of **data science**.

This is not an accident. The tools available to data scientists using Python are world class (and the envy of many other programming communities). What's great for non-data scientists is that the tools favored by the data folks have wide applicability outside the *Big Data* landscape.

Entire books have been (and continue to be) written about using Python within the *data science* space. Although you may think this advice biased, the books on this subject from *O'Reilly Media* are excellent (and plentiful). *O'Reilly Media* has made a business out of spotting where the technology industry is heading, then ensuring there's plenty of great, high-quality learning material available to those wanting to learn more.

Here's just a selection of some of the libraries and modules available to you if you do data science (or any other science calculations, for that matter). If *data science* isn't your thing, check out this stuff anyway—there's lots to like here:

- **bokeh**: A set of technologies for publishing interactive graphics on web pages.
- **matplotlib/seaborn**: A comprehensive set of graphing modules (which integrates with *iPython* and *Jupyter Notebook*).
- **numpy**: Among other things, allows you to efficiently store and manipulate multidimensional data. If you're a fan of matrices, you'll love numpy.
- **scipy**: A set of scientific modules optimized for numerical data analysis, which complements and expands upon what's provided by numpy.
- **pandas**: If you are coming to Python from the *R* language, then you'll feel right at home with pandas, which provides optimized analysis data structures and tools (and is built on top of numpy and matplotlib). The need to use pandas is what brings a lot of data folk to the community (and long may this continue). pandas is another *killer* Python application.
- **scikit-learn**: A set of machine learning algorithms and technologies implemented in Python.

Note: most of these libraries and modules are pip-installable.

The best place to start learning about the intersection of Python and *data science* is the *PyData* website: <http://pydata.org>. Click on *Downloads*, then marvel at what's available (all as open source). Have fun!

Note from Marketing:
This means they got
our memo. ☺



Sacrebleu! They
dare to ask me how I know
my soup recipe's the best?!?
Why...I ran a quick pandas data
analysis, then published it in a
Jupyter Notebook. Voilà—now
everyone knows.



5. Web Development Technologies

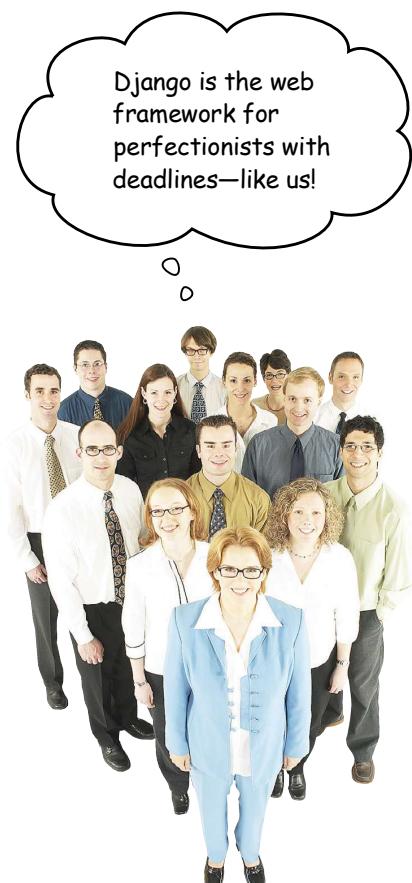
Python is very strong in the web space, but *Flask* (with *Jinja2*) isn't the only game in town when it comes to building server-side webapps (even though *Flask* is a very popular choice, especially if your needs are modest).

The best-known technology for building webapps with Python is *Django*. It wasn't used in this book due to the fact that (unlike *Flask*) you have to learn and understand quite a bit before you create your first *Django* webapp (so, for a book like this, which concentrates on teaching the basics of Python *well*, *Django* is a poor fit). That said, there's a reason *Django* is so popular among Python programmers: it's really, really good.

If you class yourself as a “web developer,” you should take the time to (at the very least) work through *Django*'s tutorial. In doing so, you'll be better informed as to whether you'll stick with *Flask* or move to *Django*.

If you do move to *Django*, you'll be in very good company: *Django* is such a large community within the wider Python community that it's able to sustain its own conference: *DjangoCon*. To date, *DjangoCon* has occurred in the US, Europe, and Australia. Here are some links to learn more:

- Djanjo's landing page (which has a link to the tutorial):
<https://www.djangoproject.com>
- DjangoCon US:
<https://djangcon.us>
- DjangoCon Europe:
<https://djangcon.eu>
- DjangoCon Australia:
<http://djangcon.com.au>



But wait, there's more

As well as *Flask* and *Django*, there are other web frameworks (and we know we'll neglect to mention somebody's favorite). Those we hear the most about include: *Pyramid*, *TurboGears*, *web2py*, *CherryPy*, and *Bottle*. Find a more complete list on the Python wiki:

<https://wiki.python.org/moin/WebFrameworks>

6. Working with Web Data

In Chapter 12, we briefly used the `requests` library to demonstrate just how cool our generator was (compared to its equivalent comprehension). Our decision to use `requests` was no accident. If you ask most Python developers working with the Web what their favorite PyPI module is, the majority responds with one word: “`requests`.”

The `requests` module lets you work with HTTP and web services via a simple, yet powerful, Python API. Even if your day job doesn’t involve working directly with the Web, you’ll learn a lot just from looking at the code for `requests` (the entire `requests` project is regarded as a master class in how to do things the Python way).

Find out more about `requests` here:

<http://docs.python-requests.org/en/master/>

PyPI: The Python Package Index lives at <https://pypi.org/>.

Scrape that web data!

As the Web is primarily a text-based platform, Python has always worked well in that space, and the *standard library* has modules for working with JSON, HTML, XML, and the other similar text-based formats, as well as all the relevant Internet protocols. See the following sections of the Python docs for a list of modules that come with the *standard library* and are of most interest to web/Internet programmers:

- Internet Data Handling:
<https://docs.python.org/3/library/netdata.html>
- Structured Markup Processing Tools:
<https://docs.python.org/3/library/markup.html>
- Internet Protocols and Support:
<https://docs.python.org/3/library/internet.html>

If you find yourself having to work with data that’s only available to you via a static web page, you’ll likely want to *scrape* that data (for a quick scraping primer, see https://en.wikipedia.org/wiki/Web_scraping). Python has two third-party modules that will save you lots of time:

- *Beautiful Soup*:
<https://www.crummy.com/software/BeautifulSoup/>
- *Scrapy*:
<http://scrapy.org>

Try both, see which one solves your problem best, and then get on with whatever else needs doing.

Soup? Soup! Did somebody mention soup?
And they said it was "beautiful"... mon dieu.



more than mysql

7. More Data Sources

To keep things as real as possible (while trying to keep it simple), we used *MySQL* as our database backend in this book. If you spend a lot of time working with SQL (regardless of the database vendor you favor), then stop whatever you're doing and take two minutes to use pip to install sqlalchemy—it may be your best two-minute installation ever.

The sqlalchemy module is to SQL geeks what requests is to web geeks: indispensable. The *SQLAlchemy* project provides a high-level, Python-inspired set of technologies for working with tabular data (as stored in the likes of *MySQL*, *PostgreSQL*, *Oracle*, *SQL Server*, and so on). If you liked what we did with the DBcm module, you're going to love *SQLAlchemy*, which bills itself as *the database toolkit for Python*.

Find out more about the project at:

<http://www.sqlalchemy.org>

There's more to querying data than SQL

Not all the data you'll ever need is in an SQL database, so there will be times when *SQLAlchemy* won't do. NoSQL database backends are now accepted as a valid addition to any data center, with *MongoDB* serving as the classic example as well as the most popular choice (even though there are many).

If you end up working with data that's being presented to you as JSON, or in a nontabular (yet structured) format, *MongoDB* (or something similar) may be just what you're looking for. Find out more about *MongoDB* here:

<https://www.mongodb.com>

And check out the Python support for programming *MongoDB* using the pymongo database driver from the PyMongo documentation page:

<https://api.mongodb.com/python/current/>

No matter where
our data is—in an SQL
or NoSQL data store—
Python and its third-party
modules do the trick.



8. Programming Tools

No matter how good you think your code is, bugs happen.

When they do, Python has lots to help you: the >>> prompt, the pdb debugger, *IDLE*, print statements, unittest, and doctest. When these options aren't enough, there are some third-party modules that might help.

Sometimes, you'll make a classic mistake that everyone else has made before you. Or perhaps you've forgotten to import some required module, and the problem doesn't crop up until you're showing off how great your code is to a room full of strangers (whoops).

To help avoid this type of thing, get *PyLint*, Python's code analysis tool:

<https://www.pylint.org>

PyLint takes your code and tells you what might be wrong with it *before* you run it for the first time.

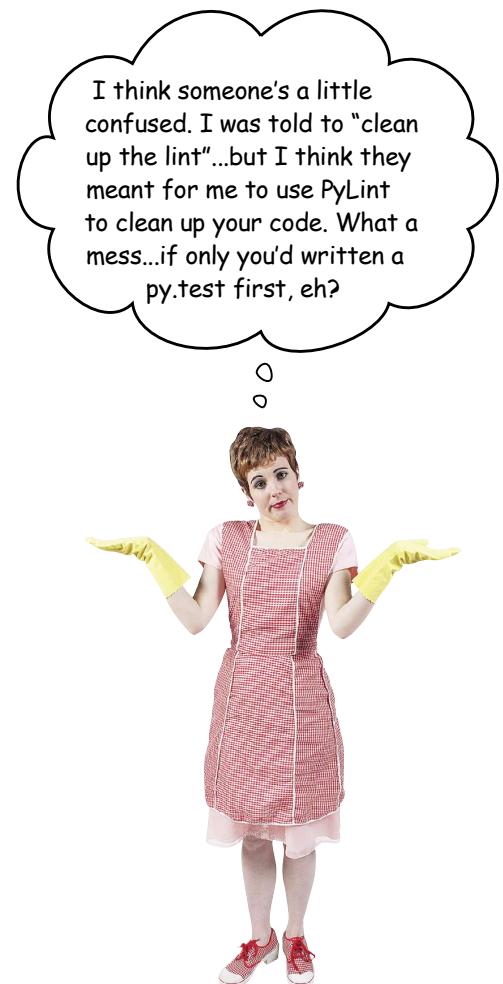
If you use *PyLint* on your code before you run it in front of a room full of strangers, it may very well prevent blushing. *PyLint* might also hurt your feelings, as no one likes to be told their code is not up to scratch. But the pain is worth the gain (or maybe that should be: *the pain is better than the public embarrassment*).

More help with testing, too

In *Appendix C*, #9, we discussed the built-in support Python provides for automated testing. There are other such tools, too, and you already know that `py.test` is one of them (as we used it earlier in this book to check our code for *PEP 8* compliance).

Testing frameworks are like web frameworks: everyone has their favorite. That said, more Python programmers than not favor `py.test`, so we'd encourage you to take a closer look:

<http://doc.pytest.org/en/latest/>



kivy kivy kivy

9. Kivy: Our Pick for “Coolest Project Ever”

One area where Python is not as strong as it could be is in the world of mobile touch devices. There are a lot of reasons why this is (which we aren’t going to get into here). Suffice it to say, at the time of publication, it is still a challenge to create an *Android* or *iOS* app with Python alone.

One project is attempting to make progress in this area: *Kivy*.

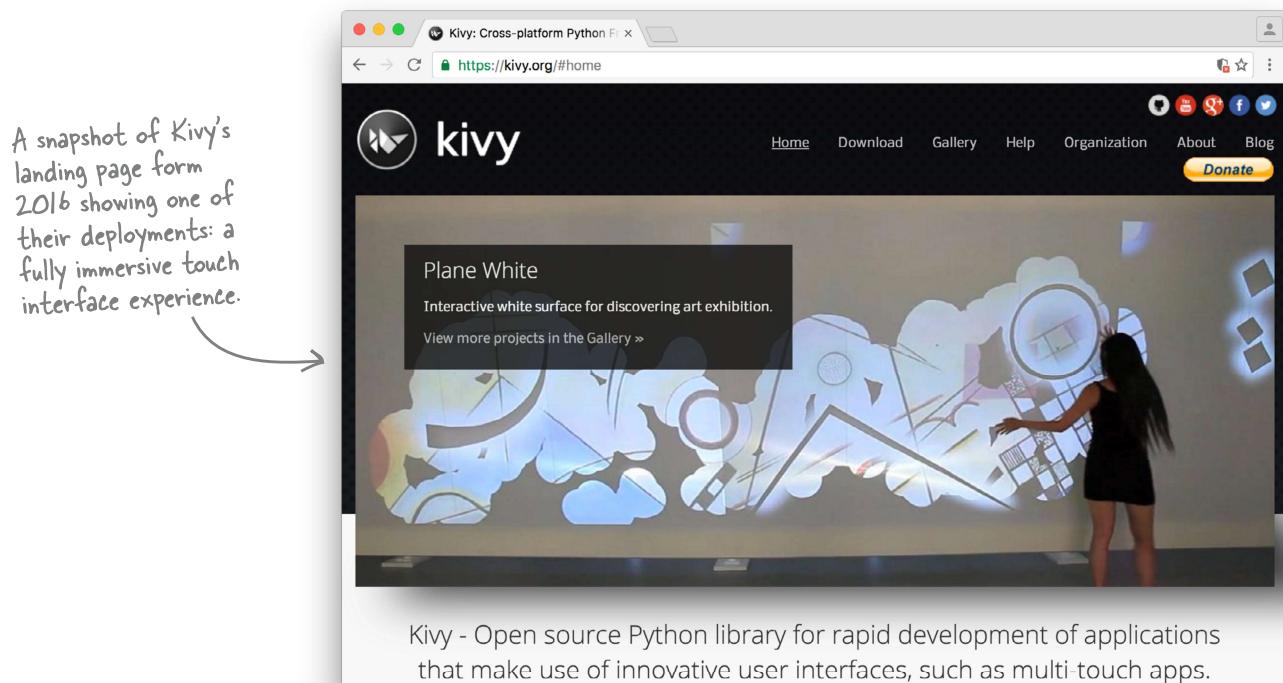
Kivy is a Python library that allows for the development of applications that use multitouch interfaces. Pop on over to the *Kivy* landing page to see what’s on offer:

<https://kivy.org>

Once there, click on the *Gallery* link and sit back for a moment while the page loads. If a project grabs your eye, click on the graphic for more information and a demo. While you view the demo, keep the following in mind: *everything you are looking at was coded with Python*. The *Blog* link has some excellent material, too.

What’s really cool is that your *Kivy* user interface code is written once, then deployed on any supported platform *unchanged*.

If you are looking around for a Python project to contribute to, consider donating your time to *Kivy*: it’s a great project, has a great team working on it, and is technically challenging. If nothing else, you won’t be bored.



10. Alternative Implementations

You already know from item #1 in *Appendix C* that there's more than one Python language release (Python 2 *and* Python 3). This means that there's *at least* two Python interpreters: one that runs Python 2 code, and one that runs Python 3 code (which is the one we've used throughout this book). When you download and install one of the Python interpreters from the Python website (like you did in *Appendix A*), the interpreter is referred to as the *C**Python reference implementation*. *C**Python* is the version of Python distributed by the *Python core developers*, and takes its name from the fact that it's written in portable C code: it's designed to be easily ported to other computing platforms. As you saw in *Appendix A*, you can download installers for *Windows* and *Mac OS X*, as well as find the interpreter preinstalled within your favorite Linux distribution. All of these interpreters are based on *C**Python*.

Python is open source, so anyone is free to take *C**Python* and change it in any way they wish. Developers can also take the Python language and implement their own interpreter for it in whichever programming language they wish, using whichever compiler techniques they like, running on whatever platform they're using. Although doing all of this is not for the faint of heart, plenty of developers do this (some of them describe it as "fun"). Here are short descriptions and links to some of the more active projects:

- *PyPy* (pronounced "pie-pie") is a experimental compiler testbed for Python 2 (with Python 3 support on the way). *PyPy* takes your Python code and runs it through a just-in-time compilation process, producing a final product that runs faster than *C**Python* in many instances. Find out more here:

<http://pypy.org>

I'm open source and
written in C. You'll find
me everywhere!

- *IronPython* is a version of Python 2 for the .NET platform:

<http://ironpython.net>

- *Jython* is a version of Python 2 that runs on Java's *JVM*:

<http://www.jython.org>

- *MicroPython* is a port of Python 3 for use on the *pyboard* microcontroller, which is no bigger than your two thumbs side by side, and may well be the coolest little thing you've ever seen. Take a look:

<http://micropython.org>



Despite all these alternative Python interpreters, the majority of Python programmers remain happy with *C**Python*. Increasingly, more developers are choosing Python 3.

appendix e: getting involved



The Python Community

No, no...there's no one else here. They've all gone to PyCon.



Python is much more than a great programming language.

It's a great community, too. The Python Community is welcoming, diverse, open, friendly, sharing, and giving. We're just amazed that no one, to date, has thought to put that on a greeting card! Seriously, though, there's more to programming in Python than the language. An entire ecosystem has grown up around Python, in the form of excellent books, blogs, websites, conferences, meetups, user groups, and personalities. In this appendix, we take a survey of the Python community and see what it has to offer. Don't just sit around programming on your own: **get involved!**

gotta love guido

BDFL: Benevolent Dictator for Life

Guido van Rossum is a Dutch programmer whose gift to the world is the Python programming language (which he started as a “hobby” in the last 1980s). The ongoing development and direction of the language is set by the *Python core developers*, of which Guido is but one (albeit a very important one). Guido’s title of *Benevolent Dictator for Life* is in recognition of the central role he continues to play in the day-to-day life of Python. If you see the letters BDFL in relation to Python, that’s a reference to Guido.

Guido is on the record as stating that the name “Python” is a nod (and a wink) toward the British television comedy troupe *Monty Python’s Flying Circus*, which helps explain the use of the name *spam* for many of the variables referred to in the Python docs.

Despite Guido’s leading role, he does **not** own Python: nobody does. However, the interests of the language are protected by the PSF.

PSF: The Python Software Foundation

The PSF is a nonprofit organization that looks after the interests of Python, and is run by a nominated/elected board of directors. The PSF promotes and sponsors the continued development of the language. This is from the PSF’s mission statement:

The mission of the Python Software Foundation is to promote, protect, and advance the Python programming language, and to support and facilitate the growth of a diverse and international community of Python programmers.

Anyone can join the PSF and get involved. See the PSF website for details:

<https://www.python.org/psf/>

One of the PSF’s major activities is involvement in (and the underwriting of) the annual Python conference: *PyCon*.

**Have your say:
join the PSF.**

PyCon: The Python Conference

Anyone can attend (and speak at) PyCon. In 2016, Portland, Oregon, hosted the conference, with thousands of Python developers in attendance (the previous two PyCons were held in Montreal, Canada). PyCon is the largest Python conference, but not the only one. You’ll find Python conferences across the globe, ranging in size from small, regional conferences (tens of attendees), through national conferences (hundreds of attendees), up to the likes of *EuroPython* (thousands of attendees).

To see if there’s a PyCon near you, search for the word “PyCon” together with the name of your nearest city (or the country you live in). Chances are, you’ll be pleasantly surprised by what you find. Attending a local PyCon is a great way to meet and interact with like-minded developers. Many of the talks and sessions at the various PyCons are recorded: pop over to *YouTube* and type “PyCon” for an idea of what’s available to view.

**Get involved:
attend PyCon.**

A Tolerant Community: Respect for Diversity

Of all the programming conferences that exist today, PyCon was one of the first to introduce and insist on a *Code of Conduct*. You can read the 2016 Code of Conduct here:

<https://us.pycon.org/2016/about/code-of-conduct/>

Such a development is a *very good thing*. More and more, the smaller regional PyCons are adopting the Code of Conduct, too, which is also very welcome. A community grows to be strong and inclusive when there are clear guidelines about what's acceptable and what isn't, and the Code of Conduct helps to make sure all the world's PyCons are as welcoming as they can be.

In addition to striving to ensure everyone is welcome, a number of initiatives attempt to increase the representation of specific groups within the Python community, especially where—traditionally—such groups have been underrepresented. The best-known of these is *PyLadies*, which was established per their mission to help “more women become active participants and leaders in the Python open source community.” If you're lucky, there's a *PyLadies* “chapter” near you: find out by starting your search from the *PyLadies* website:

<http://www.pyladies.com>

Just like the Python community, *PyLadies* started out small, but has very quickly grown to have global reach (which is truly inspirational).

Come for the language, stay for the community

Many programmers new to Python comment on how inclusive the Python community is. A lot of this attitude stems from Guido's guiding hand and example: firm, yet benevolent. There are other leading lights, too, and plenty of inspirational stories.

It doesn't get much more inspirational than *Naomi Ceder*'s talk at *EuroPython* (which was repeated at other regional conferences, including *PyCon Ireland*). Here's a link to Naomi's talk, which we encourage you to watch:

<https://www.youtube.com/watch?v=cCCiA-IIVco>

Naomi's talk surveys a life in Python, and discusses how the community supports diversity, and how there's always more work for everyone to do.

One way to learn more about a community is to listen to some of the podcasts generated by its participants. We discuss two Python podcasts next.

**Encourage
and support
diversity
within the
Python
community.**

Python Podcasts

There are podcasts on *everything* these days. Within the Python community, there are two we feel are well worth subscribing and listening to. Whether it's something to listen to while driving, cycling, running, or chilling out, these podcasts are both deserving of your attention:

- *Talk Python to Me*: <https://talkpython.fm>
- *Podcast.__init__*: <http://pythonpodcast.com>

Follow both of these podcasts on *Twitter*, tell your friends about them, and give the producers of these podcasts your full support. Both *Talk Python To Me* and *Podcast.__init__* are produced by regular members of the Python community for the benefit of all of us (and *not* for profit).

There's nothing quite like working out to the Python-related podcasts.



Python Newsletters

If podcasts aren't your thing, but you still want to keep up with what's happening in the Python world, there are three weekly newsletters that can help:

- Pycoder's Weekly: <http://pycoders.com>
- Python Weekly: <http://www.pythonweekly.com>
- Import Python: <http://importpython.com/newsletter>

These curated newsletters provide links to all types of material: blogs, vlogs, articles, books, videos, talks, new modules, and projects. And their weekly announcements arrive right to your email inbox. So, go ahead and sign up.

As well as a foundation, multiple conferences, subgroups like *PyLadies*, codes of conduct, recognition of diversity, podcasts, and newsletters, Python also has its very own notion of *Zen*.

Reciting the Zen of Python helps me get in the zone...



The Zen of Python

Many moons ago, Tim Peters (one of Python's early leading lights) sat down and wondered: *what is it that makes Python Python?*

The answer came to Tim as *The Zen of Python*, which you can read by starting any version of the interpreter and typing the following incantation into the >>> prompt:

```
import this
```

We've done this for you, and shown the output of the above line of code in the screenshot at the bottom of this page. Be sure to read *The Zen of Python* at least once a month.

Many have tried to compress *The Zen of Python* into something a little easier to digest. None other than xkcd has given it a go. If you're connected to the Internet, type this line of code into your >>> prompt to see (quite literally) how xkcd got on:

```
import antigravity
```



Python 3.5.2 Shell

```
>>>
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

Ln: 28 Col: 4



Remember: read this **at least** once a month.

to read next

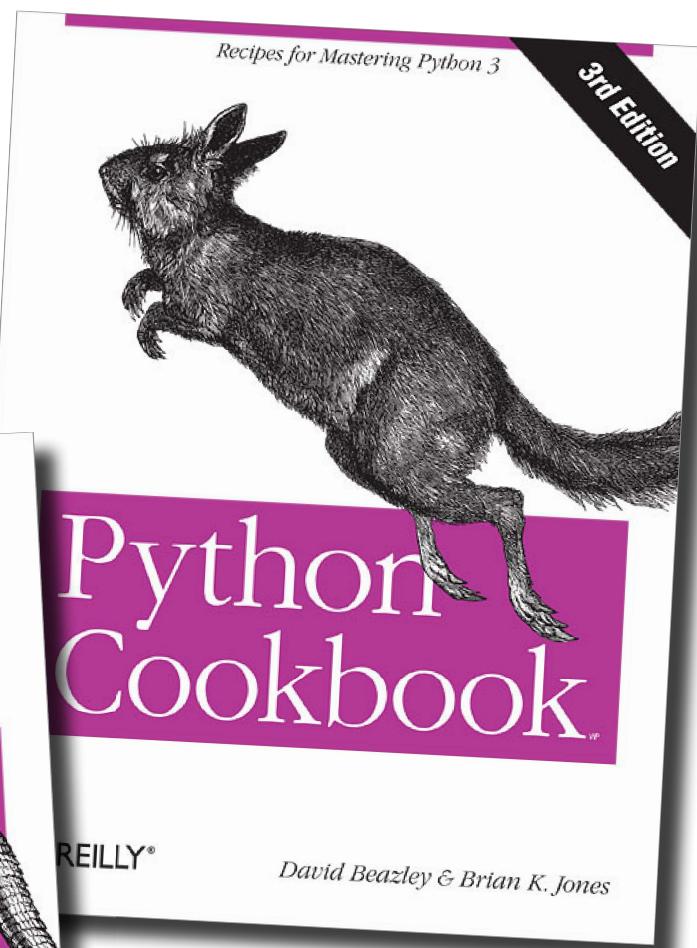
Which Book Should I Read Next?



Our Favorite Python Books

As Python has grown in popularity, the number of books devoted to the language has blossomed. Of all the books out there, there are two we regard as indispensable.

We mentioned David Beazley's work in an earlier appendix. In this book, David teams up with Brian K. Jones to document a wonderful collection of Python coding recipes. If you find yourself wondering how you do something in Python, wonder no more: look up the answer in *Python Cookbook*.



If deep-dives are more your thing, read this excellent book. There's a lot in here, but it's all good (and you'll be a better Python programmer for the experience).



Index

Symbols

>>>. *See* Python Shell
<> (angle brackets) 256–257
= (assignment operator) 13, 55, 72–74
\ (backslash) 77
^ (caret) 192
: (colon). *See* colon (:)
, (comma) 54, 123, 134
+ (concatenation operator) 543
{ } (curly braces). *See* curly braces {}
-= (decrement operator) 106
/ (forward slash) 207
+= (increment operator) 106, 318
* (multiplication operator) 87
* notation 390–391

A

Alt-P key combination (Linux/Windows) 31, 118
angle brackets 256–257
annotations (function) 162–163
append method 58–59, 72, 270

app.run() function 207, 211, 217
apt-get utility 527
*args keyword 390, 401
arguments
 about 147, 154–155
 adding multiple 165
 any number and type of 394
 by-address argument passing 184, 186–187
 by-reference argument passing 184, 186–187
 by-value argument passing 184–185, 187
 dictionary of 392–393
 function decorators 223, 390–395, 401
 interpreter processing 148
 list of 390
 methods and 317, 319–320, 322
 positional versus keyword assignment 171
 specifying default values for 170
arrays. *See* lists
arrow symbol 162–163
assignment operator 13, 55, 72–74
assignment statements 13–14
associative arrays. *See* dictionaries
asterisks 390–393
asyncio module 546
async keyword 546
AttributeError exception 483
attributes (state)
 about 49
 classes and 311–312, 322
 dictionary lookup retrieves 369
 displaying 30
 Flask’s session technology and 368
 initializing values 323–325

the index

methods and 322
objects and 313, 315, 322
authentication 364
automated testing 548
automatic reloading webapps 227
await keyword 546

B

backslash 77
BDFL (Benevolent Dictator for Life) 564
Beazley, David 546, 569
behavior. *See* methods (behavior)
BIF (built-in functions) 161
binary mode 247
blocks of code. *See* suites of code
bokeh library 555
bool built-in function 156–158
boolean values 116
by-address argument passing 184, 186–187
by-reference argument passing 184, 186–187
by-value argument passing 184–185

C

call-by-reference semantics 184, 186
call-by-value semantics 184–185
CamelCase 312
caret 192
case sensitivity and conventions 116, 312
cd command 175
Ceder, Naomi 565
ChainMap class 545
classes
 about 311–312
 attributes and 311–312, 322
 creating 310
 defining functionality of 313–314

empty 312, 444
methods and 311–312, 318
naming 312
objects and 312–313
 with statement and 305, 310, 337–338
class keyword 312
@classmethod decorator 542
client error messages 222
close method 245–246
Code of Conduct 565
collections module 545
colon (:)
 blocks of code and 16–17
 comprehensions and 506
 dictionaries and 98, 123, 506
 functions and 149, 162
 lists and 76
combinations function (itertools module) 545
comma 54, 123, 134
command-line, running Python from 175–177, 190
comments 147
comparison operators 13, 15
compilation 7
comprehensions. *See also* specific types of loops
 about 493, 504, 517
 Bahama Buzzers example 478–517
 converting patterns into 491
 dictionary 493–496, 499–502, 506
 examining 492
 list 493–496, 504, 506, 508–511
 reading CSV data as dictionaries 480–484
 reading CSV data as lists 479
 set 504–505
 spotting patterns 489–490
 transforming data 484–486
 tuples and 504, 507
concatenation operator 543
concurrency options 465
concurrent.futures module 546

- connect function 291
 constant lists (tuples) 51
 constructor methods 323
 contextlib module 337
 context management protocol
 about 305–306, 310, 338–339
 creating a context manager 337, 339
 creating a context manager class 340
 exception handling and 440–441
 function decorators and 407
 initializing context manager class 338–342
 performing set-up 338–340, 343–344
 performing tear-down 338–340, 345
 readdressing webapp code 348–358
 testing context manager 346–347
 control statements 16
 copy method 73
 Counter class 545
 CPython 561
 Cross-site Scripting (XSS) 419, 422
 CSV data
 Bahamas Buzzers example 478–482
 reading as dictionaries 480–484
 reading as lists 479
 csv module 479
 Ctrl-C key combination 210, 220
 Ctrl-P key combination (Mac) 31, 118
 curly braces { }
 blocks of code and 16
 comprehensions and 506
 dictionaries and 104, 137–139
 sets and 123
 template engines 214
 current working directory 9–10, 174
 cursor method 291–295, 463–464
- ## D
- database-enabling webapps
 creating code to work with database and tables 296
 creating database and tables 287–295
 exception handling and 418, 420, 422, 440
 installing MySQL-Connector/Python 286
 installing MySQL database driver 285
 installing MySQL server 283
 introducing Python’s DB API 284
 reusing database code 301–306
 sharing code. *See* context management protocol
 storing data 300
 Data Science 555
 data structures
 built-in 13, 50, 161
 complex 135–142, 266–267
 copying 73
 dictionaries. *See* dictionaries
 lists. *See* lists
 sets. *See* sets
 tuples. *See* tuples
 datetime module 8, 11, 486
 day attribute (date.today) 11
 DB-API , 281, 284, xvi
 debugging 224, 549
 decorators, function. *See* function decorators
 decrement operator 106
 default values for arguments 170–171
 def statement
 about 147, 149
 async keyword and 546
 default values for arguments 170
 positional versus keyword assignment 171
 delimiters 77, 262, 506
 describe log command 289, 293
 dictionaries
 about 52, 103
 of arguments 392–393
 checking for membership in 117–119
 dictionaries within 136–140
 dynamic 114
 easy to read 97
 empty 104, 136, 161

the index

- frequency counts in 102–106, 131
growing at run-time 101
iterating over 107
iterating over keys and values 108
iterating over rows of data 110
key/value pairs and 52, 96, 115–120
of lists 485–486
reading CSV data as 480–484
specifying ordering on output 109–110
spotting in code 98
spotting pattern with 489
square brackets and 99–101
dictionary comprehensions 493–496, 499–502, 506
difference method 125, 127
dir built-in function 30, 259–260, 324
distribution packages 178–182
Django framework 203, 556
docstring
 about 147
 adding 151, 168
 adding information to 162
 updating 165
doctest module 548
documenting functions 162–163
dot-notation syntax 29, 58, 316
dunder name 206, 238–239, 324–325, 338–345
duplicate objects, sets and 53, 59
dynamic assignment of variables 13, 48–49
dynamic dictionaries 114
dynamic lists 50–51, 62
- ## E
- Eclipse IDE 553
edit window 3–8, 57, 150–151
elif statement 17
else statement 16–17, 117
embedded dictionaries 136–140
embedded suites of code 18
- empty classes 312, 444
empty dictionaries 104, 136, 161
empty lists 55, 58, 161
empty sets 160–161
empty statements 312
empty strings 157
empty tuples 161
Enter key 22–24
__enter__ method 338–340, 443
environ attribute (os module) 10
environment variables 10
escape characters 77, 257
escape function (flask module) 257–258, 270
escape function (html module) 11
Exception class 427
exception handling. *See also* specific exceptions
 built-in exceptions 427
 catch-all exception handler 428, 431
 context manager and 440–441
 creating custom exceptions 444–447
 databases and 418, 420, 422, 440, 448–455
 functions and 421–422
 import mechanisms 29
 indentation errors 45
 misspelled variables 45
 output display and 255
 PEP 8 failure messages 191–192
 run-time and 115–121, 423–424, 474
 syntax errors 5, 57
 webapps and 255, 418–420, 422, 433, 437–439
 with statement and 443, 451–452
- executing code. *See also* run-time
 Alt-P key combination for 31, 118
 Ctrl-P key combination for 31, 118
 F5 key for 4, 6, 151
 interpreter processing in 8
 invoking functions 150
 pausing execution 20, 28
 running concurrently 546

running immediately 7, 22
 running multiple times 20
`__exit__` method 338–340, 443, 452–453
`extend` method 64
`extends` directive (Jinja2) 214

F

`F5` key 4, 6, 151
`False` value 156–157
`FileNotFoundException` exception 423–424, 431
`Flask` class 205, 217
`Flask` framework
 about 203, 556
 accessing HTML form data 226
 associating function with multiple URLs 236
 creating webapp objects 206
 debugging mode 224
 installing 202
 Jinja2 template engine 214–215, 229–230, 274, 276
 Markup object 257
 rendering templates from 217
 request object 226–227, 259
 running webapps 204–205
 session mechanism 367–368
 testing webapps 220–221
 threading and 471
`flask` module
 escape function 257–258, 270
 `Flask` class 205, 217
 session dictionary 368–379
`for` loop. *See also* comprehensions
 about 20, 24–27, 504
 lists and 86–88, 479
 slices and 87
 spotting patterns in 489–490
`format` method 543
`formatting`
 data 484–486
 strings 543

form dictionary (Flask) 226
`<form>` tag 222
 forward slash 207
 frequency counts
 about 102–103, 131
 incrementing 105
 initializing 105
 selecting data structure 104
 updating 105–106
 function decorators
 about 209, 385
 adding 217–218
 adjusting behaviors 207
 arguments and 223, 390–395, 401
 components in writing 385–394
 context managers and 407
 creating 395–410
 URLs and 207, 209, 211, 218, 223, 396, 408
 function objects 386–389, 395, 397–398
 functions. *See also* arguments (functions); *See also* specific functions
 about 9, 147–148
 best practice for 153
 built-in 161
 creating 149, 166–169
 documenting 162–163
 editing 150–151
 embedding generators within 511–516
 exception handling and 421–422
 importing 9, 28–29
 invoking 150
 invoking passed functions 387
 methods and 316, 322
 modules and 9, 173
 multiple URLs 236
 naming 149, 165, 312
 nested 388, 400
 passing to functions 386
 returning from functions 389
 returning results 156–159
 reusing code with 146, 173

the index

sharing 173
string quote characters 152
troubleshooting 184, 187
variables and 321–322
functools module 402, 545

G

generators 508, 510
getcwd function (os module) 9–10
GET method (HTTP) 222–223
global variables 366

H

hashes. *See* dictionaries
Hellman, Doug 545–546
help command 31, 41, 66
Homebrew package manager 283, 525
HTML forms
 access with Flask 226
 building 213–215
 displaying 218
 producing results 229–230
 redirecting to avoid unwanted errors 234–235
 rendering templates from Flask 216–217
 testing template code 219–221
html module 11
HTTP (HyperText Transfer Protocol)
 status codes 222
 web servers and 366

I

id built-in function 328
IDLE (Python IDE) 3–7, 203, 553
if statement 16–17, 117–119
ImmutableMultiDict dictionary 261
ImportError exception 176–177

import statement
 about 9, 28–29
 Flask framework and 205
 interpreter search considerations 174–177
 positioning 303
 sharing modules with 173
 threading module and 465
 Zen of Python 567
increment operator 106, 318
indentation levels for suites 18, 45
indenting suites of code
 about 15–18, 40
 for functions 147
 for loops 24, 27
index values, lists and 63, 75
informational messages 222
__init__ method 323–327, 330, 338–340, 443
inner functions 388, 400
in operator
 about 15
 dictionaries and 115–119
 lists and 56, 59
 sets and 125
input built-in function 60
insert method 65
INSERT statement (SQL) 289, 463–464
InterfaceError exception 423, 441, 443
interpreter (Python)
 about 7–8
 alternative implementations 561
 asking for help 31, 41
 case sensitivity 116
 dictionary keys and 108
 functions and 148
 identifying operating system 10
 identifying site-package locations 174
 internal ordering used by 52, 108
 running from command-line 175–177
 syntax errors 5, 57

whitespace and 40
intersection method 125, 128, 159, 167
ipython shell 552
IronPython project 561
isofomat function (datetime module) 11
items method 110
itertools module 545

J

Java VM 7
Jinja2 template engine
about 214–215, 229
calculating data needed 230
readable output with 274, 276
join method 67, 258, 268
Jones, Brian K. 569
Jupyter Notebook IDE 554
Jython project 561

K

keyboard shortcuts 27
KeyError exception 115–121
key/value pairs (dictionaries)
about 52, 96
adding 101
creating on the fly 115–120
interpreter processing and 108
keyword assignment of arguments 171
Kivy library 560
**kwargs keyword 392–393, 401

L

len built-in function 58
level of indentation for suites 18
list built-in function 42, 126
list comprehensions 493, 504, 506, 508–511

lists

about 13, 50–51, 54, 89
assignment operators 13, 55, 72–74
checking for membership in 15, 56, 59
copying existing 72
creating literally 55
dictionaries of 485–486
dynamic 50–51, 62
empty 55, 58, 161
extending with objects 64
growing at run-time 58
iterating over a sequence of objects 24–25
of arguments 390–391
popping objects off 63
reading CSV data as 479
removing objects from 62
slice notation 77–81, 85
sorted 126
spotting in code 54
spotting pattern with 490
square bracket notation 13, 54, 66, 74–80, 85
starting and stopping with 78
stepping with 79
tuples and 51, 132
when not to use 90–91
working with 56, 71
working within edit window 5
literal lists 54–55
localhost 211
logins/logouts 374–381, 384
logs and logging. *See also* database-enabling webapps
determining structure for 288
dir built-in function and 259–260
examining raw data 256
open, process, close technique 250, 253
single line of delimited data 262
updating webapps 350–356
viewing through webapps 254, 258
loopback address 211
loop comprehensions. *See* comprehensions
loops. *See* specific types of loops

M

MacPorts package manager 525
maps. *See* dictionaries
MariaDB 282–283
Markup object (Flask) 257
matplotlib/seaborn modules 555
memory management 62
messages, HTTP status codes 222, 235
methods (behavior)
 about 49
 arguments and 317, 319–320, 322
 attributes and 322
 chaining method calls 483
 classes and 311–312, 318
 decorators adjusting 207
 functions and 316, 322
 invoking 316–317
 objects and 313, 315, 322
 running webapp 208
MicroPython project 561
modules
 about 8
 adding to site-packages 178
 creating 173
 functions and 9, 173
 ImportError exception 176–177
 importing 29, 173–174
 sharing code 183
 third party 12
MongoDB 558
month attribute (`date.today()`) 11
multiplication operator 87
multiprocessing module 546
MySQL
 benefits of 358
 DB-API and 284
 exception handling and 418, 420, 422, 440, 448–455
 installing MySQL-Connector/Python driver 285–286

installing MySQL server 283
querying considerations 462–463
MySQL console 287

N

NameError exception 321
namespaces 29
 `__name__` value 206
naming conventions 5
nested functions 388, 400
newsletters (Python) 566
Not Found message 208
not in operator 59, 118–119
numbers
 assigning to variables 48
 generating randomly 20, 30–31
numpy package 555

O

object class 324
object instantiation 312, 323
object-oriented programming (OOP) 311, 324, 542
objects
 about 48–53
 attributes and 313, 315, 322
 classes and 312–313
 creating 312, 323
 defining representation of 328–329
 duplicate 53, 59
 extending lists with 64
 function 386–389, 395, 397–398
 key/value pairs and 96
 methods and 313, 315, 322
 popping off lists 63
 removing from lists 62
 sequence of 24–25, 124
 webapp 206
open function 245–246

- opening editing window 3
- open, process, close technique
 - about 245
 - invoking logging function 250, 253
 - reading data from existing files 246
 - with statement and 247–248
- operating system, identifying for interpreter 10
- operators
 - assignment 13, 55, 72–74
 - checking for membership with 15, 56, 59, 117–119
 - comparison 13, 15
 - concatenation 543
 - decrement 106
 - increment 106, 318
 - multiplication 87
 - super 15
 - ternary 117
- ordered data structures 50–51
- OrderedDict dictionary 545
- os module
 - about 9
 - environ attribute 10
 - getcwd function 9–10
 - platform attribute 10
 - usage example 10–11
- output display
 - exception handling and 255
 - Python Shell and 22
 - raw data to readable 265–266, 274
 - readable via Jinja2 276
 - specifying dictionary ordering for 109–110
- P**
- pandas tools 555
- parentheses ()
 - comprehensions and 506
 - function arguments in 149
 - object instantiation and 312
 - return statement and 158
 - tuples in 132, 134
- partial function 545
- pass keyword 312
- pausing execution 20, 28
- pdb debugger 549
- pep8 plug-in 189–190
- PEP (Python Enhancement Protocol)
 - about 153
 - DB-API specification 284
 - line length standard 263
 - testing for compliance 188–193, 548
- PermissionError exception 426, 431
- permutations function (itertools module) 545
- Peters, Tim 567
- pip (Package Installer for Python)
 - downloading requests library 509
 - installing Flask 202
 - installing packages with 182
 - installing pep8 plug-in 189–190
 - installing pyreadline module 524
 - installing pytest testing framework 189–190
- platform attribute (os module) 10
- podcasts (Python) 566
- pop method 63
- positional assignment of arguments 171
- PostgreSQL 282
- POST method (HTTP) 222–223
- pprint function (pprint module) 139
- pprint module 139
- print built-in function
 - about 15
 - accessing dictionary data values 108
 - default behavior 247
 - displaying objects 329
 - identifying Python version 10
 - optional arguments 263
- product function (itertools module) 545
- programming tools 559
- prompt (Python Shell). *See* Python Shell

the index

- protocol port number 204, 211
PSF (Python Software Foundation) 564
ptpython REPL 553
PyCharm IDE 553
py command 175, 190
PyCon (Python Conference) 564
PyLint tool 559
pymongo database driver 558
PyPI (Python Package Index) 183, 202, 557
PyPy project 561
pyreadline module 524
pytest testing framework 189–190
py.test tool 548, 559
Python 2 540
Python 3
 about 310
 installing on Linux 527
 installing on Mac OS X 525–526
 installing on Windows 522–524
 usage recommendations 540
PythonAnywhere
 about 529
 configuring webapps 536
 creating starter webapp 534–535
 extracting and installing code 533
 preparing webapps 530
 signing up for 531
 testing deployed webapp 537
 uploading files to the cloud 238, 240, 532
Python community , xxvi–xxx
Python Core Developers 561
Python Packaging Authority 183
Python Shell
 about 4
 accessing prompt within 10, 21–22
 alternatives to 552
 asking for help 31, 41
 copying code to editor 57
experimenting at 21, 23–32
recalling last commands typed 31
running interpreter from 175–177
terminating statements with Enter key 24
- ## Q
- quit command 175
quotation marks
 comments and 147
 strings and 77, 152
- ## R
- Ramalho, Luciano 569
randint function (random module) 20, 30–31, 174
random module 20, 30–31, 174
random number generation 20, 30–31
range built-in function 25, 40–42
reading
 CSV data as dictionaries 80
 CSV data as lists 479
 data from existing files 246
README.txt file 179–181
redirect function (Flask) 234–235
redirection messages 222
remove method 62
render_template function (Flask) 217–218, 234
REPL tool 4, 553
request object (Flask) 226–227, 259–260, 324
requests library 509, 557
requests module 557–558
results, functions returning 156–159
return keyword 147
return statement
 about 156
 parentheses and 158
 returning multiple values 159
 returning one value 158

return values (functions)
 about 156
 interpreter processing 148
 variable scope and 322

route decorator
 about 209
 adding 217–218
 adjusting behaviors 207
 optional arguments 223

run-time
 exception handling 115–121, 423–424, 474
 growing dictionaries at 101
 growing lists at 58

`RuntimeError` exception 423

S

scikit-learn tools 555
 scipy modules 555
 scope of variables 321–322
`SELECT` statement (SQL) 464
 self argument 317, 319–320, 322
 sequence of objects 24–25, 124
 server error messages 222
 session dictionary (flask module) 368–379
 sessions
 about 367
 managing logins/logouts with 374–381
 state and 368–373

set built-in function 124–125, 160–161, 167
 set comprehensions 504–505
`setdefault` method 119–121
 sets
 about 53, 123
 combining 125–126
 commonality between 125, 128
 creating efficiently 124
 creating from sequences 124
 difference between 125, 127
 duplicate objects and 53, 59, 123

empty 160–161
 spotting in code 123
 setup function (setuptools module) 179
 setuptools module 178–179
 single-object tuples 134
 site-packages 174, 177–179
 sleep function (time module) 20, 28
 slice notation
 for loop and 87–88
 lists and 77–81, 85
`__slots__` directive 542
 sorted built-in function
 about 544
 dictionaries and 109–110
 sets and 123, 126
 spaces versus tabs 40
 split method 268, 270, 479, 482–483
 sqlalchemy module 558
`SQLError` exception 453–454
`SQL injection (SQLi)` 419, 422
`SQLite` 282
 square brackets []
 comprehensions and 506
 dictionaries and 99–101, 141
 lists and 13, 54, 66, 74–80, 85
 tuples and 133
 standard library
 about 9, 10, 146
 additional information 12, 545, 547
 cautions adding/removing modules 178
 concurrency options 465
 identifying locations 174
 usage examples 8, 10–11
 start value 41, 76, 78
 state. *See* attributes (state)
 statements
 assignment 13–14
 control 16
 displaying output 22

the index

- empty 312
 - reusability of 302
 - terminating with Enter key 22–24
 - try..except 424–431, 434, 441–442
 - @staticmethod decorator 542
 - status codes (HTTP) 222
 - step value 41, 76, 79
 - stop value 41, 76, 78
 - storing data
 - in databases and tables, xvi–xxxviii
 - in data structures, xii–xxxviii
 - in text files 245
 - strftime function (time module) 11
 - strings
 - assigning to variables 48
 - empty 157
 - formatting 543
 - iterating a specific number of times 24–25
 - joining together 67, 258, 268
 - key/value pairs and 96
 - quotation marks and 77, 152
 - splitting apart 268
 - turning into list of letters 78
 - whitespace and 482
 - strip method 482–483
 - submodules 8
 - success messages 222, 235
 - sudo command 190, 202, 527
 - suites of code
 - comments in 147
 - embedded suites within 18
 - functions and 147
 - indentation levels and 18
 - indenting 15–18, 24, 27, 40
 - looping 20, 24–27
 - running multiple times 20
 - unindenting 27
 - super operators 15
 - syntax errors 5, 57, 312
 - sys module 10, 429–430
- ## T
- tables 288–289, 296. *See also* dictionaries
 - <table> tag 274
 - tabs versus spaces 40
 - <td> tag 274
 - template engines
 - about 213–215
 - embedding display logic in 275
 - preparing to run code 219–221
 - relating to web pages 216
 - rendering from Flask 217–218
 - ternary operator 117
 - testing developer tools 189–190, 548
 - text files, saving data to 245
 - text mode 247
 - Thread class 465–466
 - threading library 465
 - threading module 465, 469–470, 546
 - <th> tag 274
 - time module 11, 20, 28
 - tkinter library 547
 - today function (datetime module) 11
 - trailing whitespace 40
 - <tr> tag 274
 - True value 156–157
 - try..except statements 424–431, 434, 441–442
 - tuples
 - about 51, 132–133
 - comprehensions and 504, 507
 - empty 161
 - lists and 51, 132
 - single-object 134
 - spotting in code 132
 - turtle module 547

type built-in function 132, 328
 TypeError exception 319, 326–327, 330
 type hints (annotations) 162–163

U

unindenting suites of code 27
 union method 125–126
 unittest module 548
 unordered data structures 52–53
 URLs
 function decorators and 207, 209, 211, 218, 223, 396, 408
 functions with multiple 236
 processing with generators 510–511
 processing with list comprehensions 509
 restricting access to 382–383, 396, 408

V

van Rossum, Guido 39, 564
 variables
 assigning objects to 48–49
 assigning values to 13, 48–49
 displaying values of 22
 dynamic assignment of 13, 48–49
 environment 10
 functions and 321–322
 global 366
 initializing values 323–325
 misspelling 45
 scope of 321–322
 usage example 13
 venv technology 541
 version, identifying 10
 vertical bar 262
 vim text editor 553
 virtualenv module 541
 virtual programming environments 541

W

web applications. *See also* database-enabling webapps
 adding finishing touch 234
 adding robustness to 455
 automatic reloading 227
 calculating data needed 230
 creating Flask webapp object 206
 deploying with PythonAnywhere 529–537
 edit/stop/start/test cycle 224–225
 exception handling and 255, 418–420, 422, 433, 437–439
 exposing functionality to the Web 209–210
 function decorators 207, 209, 211, 218, 223
 functions with multiple URLs 236
 global variables and 366
 handling posted data 223
 how they work 198–199
 HTML forms 213–221, 226
 HTTP status codes 222
 installing and using Flask 202–203
 preparing for the cloud 238–240
 producing results as HTML 229–230
 redirecting to avoid unwanted errors 235
 request data in 227
 restarting 210, 220
 running behaviors 208
 running for first time 204–205
 stopping 210, 220
 testing 210, 220
 updating 348–349
 viewing logs 254
 web development technologies 556
 what do we want them to do 200, 212
 what happens on the web server 201
 web development technologies 556
 web servers
 about 365–366
 HTTP status codes and 222
 webapp process and 198, 201, 366
 while loop 24
 whitespace 40, 192, 482, 518

the index

WingWare IDE 553

with statement

 classes and 305, 310, 337–338

 context management protocol and 339

 exception handling and 443, 451–452

 open, process, close technique and 247–248

 split method and 482

 viewing logs through webapps 254

wonder name 206

wraps function (functools module) 401

X

xkcd webcomic 567

XSS (Cross-site Scripting) 419, 422

Y

year attribute (`date.today`) 11

Z

Zen of Python 567

ZIP files 180