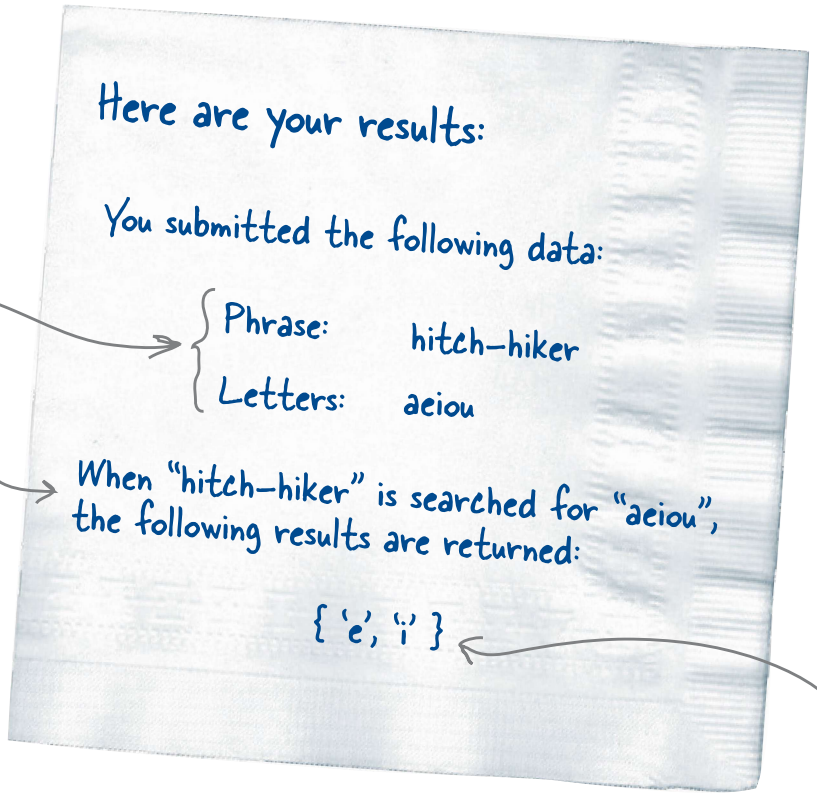
# What Happens on the Web Server?

When the user clicks on the **Do it!** button, the browser sends the data to the waiting web server, which extracts the `phrase` and `letters` values, before calling the `search4letters` function on behalf of the now-waiting user.

Any results from the function are returned to the user's browser as another web page, which we again sketch out on a paper napkin (shown below). For now, let's assume the user entered "hitch-hiker" as the `phrase` and left the `letters` value defaulted to `aeiou`. Here's what the results web page might look like:




The submitted data is echoed back to the user.

The results returned by "search4letters" are shown, too.

## What do we need to get going?

Other than the knowledge you already have about Python, the only thing you need to build a working server-side web application is a **web application framework**, which provides a set of general foundational technologies upon which you can build your webapp.

Although it's more than possible to use Python to build everything you need from scratch, it would be madness to contemplate doing so. Other programmers have already taken the time to build these web frameworks for you. Python has many choices here. However, we're not going to agonize over which framework to choose, and are instead just going to pick a popular one called *Flask* and move on.

# Let's Install Flask

We know from Chapter 1 that Python's standard library comes with lots of *batteries included*. However, there are times when we need to use an application-specific third-party module, which is *not* part of the standard library. Third-party modules are imported into your Python program as needed. However, unlike the standard library modules, third-party modules need to be installed *before* they are imported and used. Flask is one such third-party module.

As mentioned in the previous chapter, the Python community maintains a centrally managed website for third-party modules called **PyPI** (short for *the Python Package Index*), which hosts the latest version of Flask (as well as many other projects).

**Find PyPI at pypi.python.org.**

Recall how we used `pip` to install our `vsearch` module into Python earlier in this book. `pip` also works with PyPI. If you know the name of the module you want, you can use `pip` to install any PyPI-hosted module directly into your Python environment.

## Install Flask from the command-line with pip

If you are running on *Linux* or *Mac OS X*, type the following command into a terminal window:

```
$ sudo -H python3 -m pip install flask
```

Use this command on Mac OS X and Linux.

Note: case is important here. That's a lowercase "f" for "flask".

If you are running on *Windows*, open up a command prompt—being sure to *Run as Administrator* (by right-clicking on the option and choosing from the pop-up menu)—and then issue this command:

```
C:\> py -3 -m pip install flask
```

Use this command on Windows.

This command (regardless of your operating system) connects to the PyPI website, then downloads and installs the **Flask** module and four other modules Flask depends on: **Werkzeug**, **MarkupSafe**, **Jinja2**, and **itsdangerous**. Don't worry (for now) about what these extra modules do; just make sure they install correctly. If all is well, you'll see a message similar to the following at the bottom of the output generated by `pip`. Note that the output runs to over a dozen lines or so:

```
...
Successfully installed Jinja2-2.8 MarkupSafe-0.23 Werkzeug-0.11 flask-0.10.1
itsdangerous-0.24
```

At the time of writing, these are the current version numbers associated with these modules.

If you don't see the "`Successfully installed...`" message, make sure you're connected to the Internet, and that you've entered the command for your operating system *exactly* as shown above. And don't be too alarmed if the version numbers for the modules installed into your Python differ from ours (as modules are constantly being updated, and dependencies can change, too). As long as the versions you install are *at least* as current as those shown above, everything is fine.

# How Does Flask Work?

Flask provides a collection of modules that help you build server-side web applications. It's technically a *micro* web framework, in that it provides the minimum set of technologies needed for this task. This means Flask is not as feature-full as some of its competitors—such as **Django**, the mother of all Python web frameworks—but it is small, lightweight, and easy to use.

As our requirements aren't heavy (we only have two web pages), Flask is more than enough web framework for us at this time.

## Geek Bits

**Django** is a hugely popular web application framework within the Python community. It has an especially strong, prebuilt administration facility that can make working with large webapps very manageable. It's overkill for what we're doing here, so we've opted for the much simpler, but more lightweight, **Flask**.

## Check that Flask is installed and working

Here's the code for the most basic of Flask webapps, which we are going to use to test that Flask is set up and ready to go.

Use your favorite text editor to create a new file, and type the code shown below into the file, saving it is as `hello_flask.py` (you can save the file in its own folder, too, if you like—we called our folder `webapp`):

## Ready Bake Code

This is "hello_flask.py".

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

app.run()
```

Type this code in exactly as shown here... we'll get to what it means in a moment.

## Run Flask from your OS command line

Don't be tempted to run this Flask code within IDLE, as IDLE wasn't really designed to do this sort of thing well. IDLE is great for experimenting with small snippets of code, but when it comes to running applications, you are much better off running your code directly via the interpreter, at your operating system's command line. Let's do that now and see what happens.

**Don't use IDLE to run this code.**

# Running Your Flask Webapp for the First Time

If you are running on *Windows*, open a command prompt in the folder that contains your `hello_flask.py` program file. (Hint: if you have your folder open within the *File Explorer*, press the Shift key together with the right mouse button to bring up a context-sensitive menu from which you can choose *Open command window here*). With the Windows command line ready, type in this command to start your Flask app:

We saved our code in a folder called "webapp".

```
C:\webapp> py -3 hello_flask.py
```

Asks the Python interpreter to run the code in "hello_flask.py."

If you are on *Mac OS X* or *Linux*, type the following command in a terminal window. Be sure to issue this command in the same folder that contains your `hello_flask.py` program file:

```
$ python3 hello_flask.py
```

No matter which operating system you're running, Flask takes over from this point on, displaying status messages on screen whenever its built-in web server performs any operation. Immediately after starting up, the Flask web server confirms it is up and running and waiting to service web requests at Flask's test web address (`127.0.0.1`) and protocol port number (`5000`):

If you see this message, all is well.

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Flask's web server is ready and waiting. *Now what?* Let's interact with the web server using our web browser. Open whichever browser is your favorite and type in the URL from the Flask web server's opening message:

This is the address where your webapp is running. Enter it exactly as shown here.

```
http://127.0.0.1:5000/
```

After a moment, the "Hello world from Flask!" message from `hello_flask.py` should appear in your browser's window. In addition to this, take a look at the terminal window where your webapp is running...a new status message should've appeared too, as follows:

Ah ha! Something happened.

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [23/Nov/2015 20:15:46] "GET / HTTP/1.1" 200 -
```

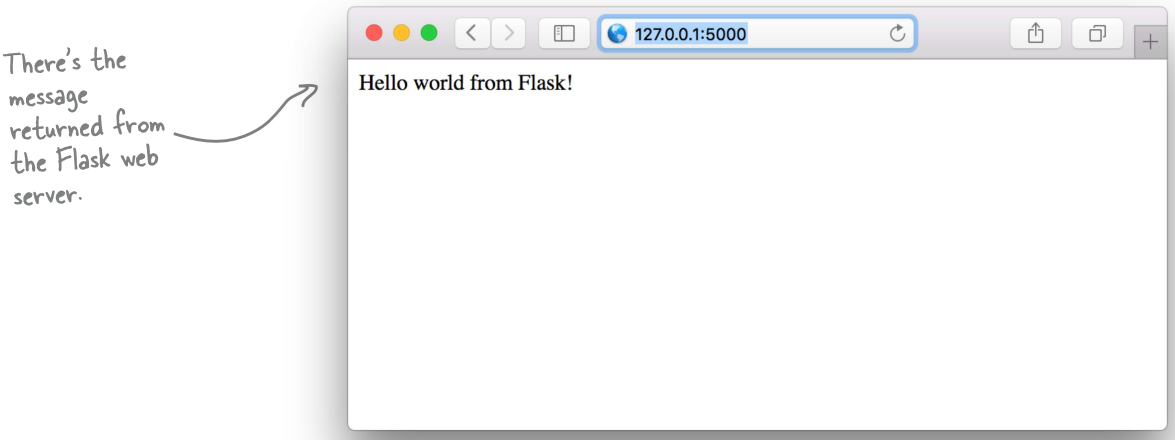
## Geek Bits

Getting into the specifics of what constitutes a **protocol port number** is beyond the scope of this book. However, if you'd like to know more, start reading here:

***https://en.wikipedia.org/wiki/Port_(computer_networking)***

# Here's What Happened (Line by Line)

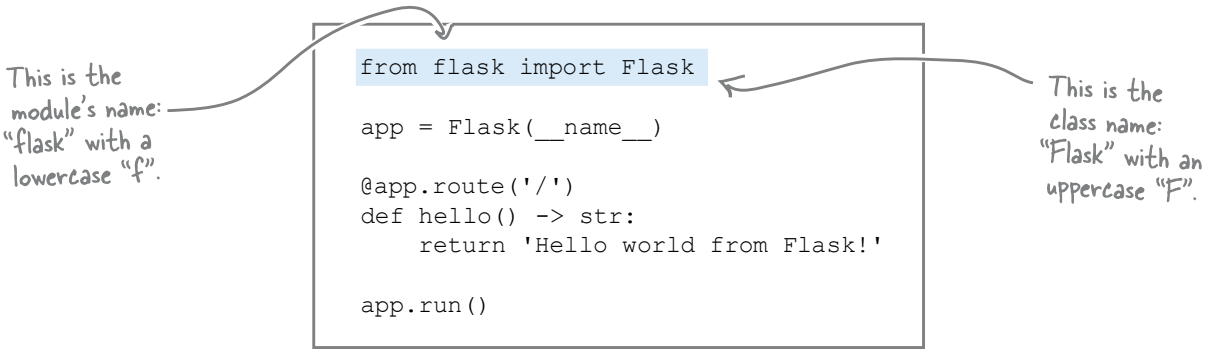
In addition to Flask updating the terminal with a status line, your web browser now displays the web server's response. Here's how our browser now looks (this is *Safari* on *Mac OS X*):




By using our browser to visit the URL listed in our webapp's opening status message, the server has responded with the "Hello world from Flask!" message.

Although our webapp has only six lines of code, there's a lot going on here, so let's review the code to see how all of this happened, taking each line in turn. Everything else we plan to do builds on these six lines of code.

The first line imports the `Flask` class from the `flask` module:

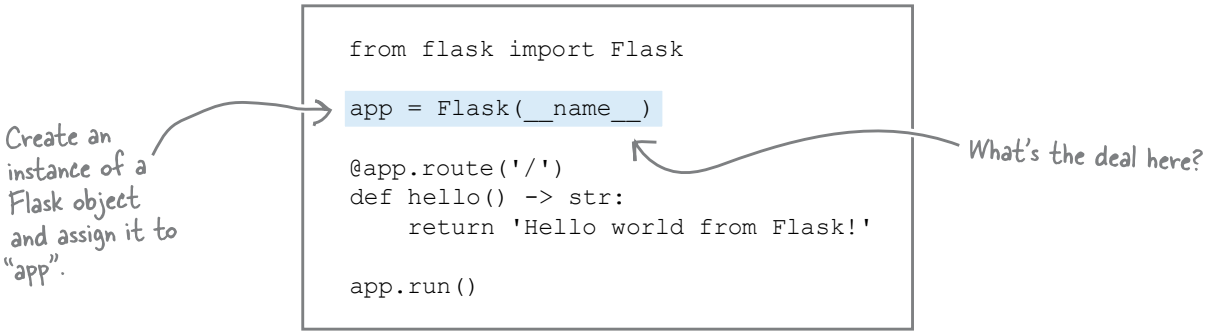



Remember when we discussed alternate ways of importing?

You could have written `import flask` here, then referred to the `Flask` class as `flask.Flask`, but using the `from` version of the `import` statement in this instance is preferred, as the `flask.Flask` usage is not as easy to read.

# Creating a Flask Webapp Object

The second line of code creates an object of type `Flask`, assigning it to the `app` variable. This looks straightforward, but for the use of the strange argument to `Flask`, namely `__name__`:




The `__name__` value is maintained by the Python interpreter and, when used anywhere within your program's code, is set to the name of the currently active module. It turns out that the `Flask` class needs to know the current value of `__name__` when creating a new `Flask` object, so it must be passed as an argument, which is why we've used it here (even though its usage does look *strange*).

This single line of code, despite being short, does an awful lot for you, as the Flask framework abstracts away many web development details, allowing you to concentrate on defining what you want to happen when a web request arrives at your waiting web server. We do just that starting on the very next line of code.

## Geek Bits

Note that `__name__` is two underscore characters followed by the word "name" followed by another two underscore characters, which are referred to as "double underscores" when used to prefix and suffix a name in Python code. You'll see this naming convention a lot in your Python travels, and rather than use the long-winded: "double underscore, name, double underscore" phrase, savvy Python programmers say: "dunder name," which is **shorthand for the same thing**. As there's a lot of double underscore usages in Python, they are collectively known as "the dunders," and you'll see lots of examples of other dunders and their usages throughout the rest of this book.

As well as the dunders, there is also a convention to use a single underscore character to prefix certain variable names. Some Python programmers refer to single-underscore-prefixed names by the groan-inducing name "wonder" (shorthand for "one underscore").

# Decorating a Function with a URL

The next line of code introduces a new piece of Python syntax: **decorators**. A function decorator, which is what we have in this code, adjusts the behavior of an existing function *without* you having to change that function's code (that is, the function being decorated).

You might want to read that last sentence a few times.

In essence, decorators allow you to take some existing code and augment it with additional behavior as needed. Although decorators can also be applied to classes as well as functions, they are mainly applied to functions, which results in most Python programmers referring to them as **function decorators**.

Let's take a look at the function decorator in our webapp's code, which is easy to spot, as it starts with the @ symbol:

## Geek Bits

Python's decorator syntax take inspiration from Java's annotation syntax, as well as the world of functional programming.

Here's the function decorator, which—like all decorators—is prefixed with the @ symbol.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

app.run()
```

This is the URL.

Although it is possible to create your own function decorators (coming up in a later chapter), for now let's concentrate on just using them. There are a bunch of decorators built in to Python, and many third-party modules (such as Flask) provide decorators for specific purposes (`route` being one of them).

Flask's `route` decorator is available to your webapp's code via the `app` variable, which was created on the previous line of code.

The `route` decorator lets you associate a URL web path with an existing Python function. In this case, the URL "/" is associated with the function defined on the very next line of code, which is called `hello`. The `route` decorator arranges for the Flask web server to call the function when a request for the "/" URL arrives at the server. The `route` decorator then waits for any output produced by the decorated function before returning the output to the server, which then returns it to the waiting web browser.

It's not important to know how Flask (and the `route` decorator) does all of the above "magic." What is important is that Flask does all of this for you, and all you have to do is write a function that produces the output you require. Flask and the `route` decorator then take care of the details.

**A function decorator adjusts the behavior of an existing function (without changing the function's code).**

# Running Your Webapp's Behavior(s)

With the `route` decorator line written, the function decorated by it starts on the next line. In our webapp, this is the `hello` function, which does only one thing: returns the message "Hello world from Flask!" when invoked:

This is just a regular Python function which, when invoked, returns a string to its caller (note the '–> str' annotation).

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

app.run()
```

The final line of code takes the Flask object assigned to the `app` variable and asks Flask to start running its web server. It does this by invoking `run`:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

app.run()
```

Asks the webapp to start running

At this point, Flask starts up its included web server and runs your webapp code within it. Any requests received by the web server for the "/" URL are responded to with the "Hello world from Flask!" message, whereas a request for any other URL results in a 404 "Resource not found" error message. To see the error handling in action, type this URL into your browser's address bar:

**http://127.0.0.1:5000/doesthiswork.html**

That URL does not exist: 404!

Your browser displays a "Not Found" message, and your webapp running within its terminal window updates its status with an appropriate message:

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [23/Nov/2015 20:15:46] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2015 21:30:26] "GET /doesthiswork.html HTTP/1.1" 404 -
```

The messages you see may differ slightly. Don't let this worry you.

# Exposing Functionality to the Web

Putting to one side the fact that you've just built a working webapp in a mere six lines of code, consider what Flask is doing for you here: it's providing a mechanism whereby you can take any existing Python function and display its output within a web browser.

To add more functionality to your webapp, all you have to do is decide on the URL you want to associate your functionality with, then write an appropriate `@app.route` decorator line above a function that does the actual work. Let's do this now, using our `search4letters` functionality from the last chapter.

## Sharpen your pencil

Let's amend `hello_flask.py` to include a second URL: `/search4`. Write the code that associates this URL with a function called `do_search`, which calls the `search4letters` function (from our `vsearch` module). Then arrange for the `do_search` function to return the results determined when searching the phrase: "life, the universe, and everything!" for this string of characters: `'eiru,!'`.

Shown below is our existing code, with space reserved for the new code you need to write. Your job is to provide the missing code.

**Hint**: the results returned from `search4letters` are a Python set. Be sure to cast the results to a string by calling the `str` BIF *before* returning anything to the waiting web browser, as it's expecting textual data, not a Python set. (Remember: "BIF" is Python-speak for *built-in function*.)

Do you need to import anything?

```
from flask import Flask
.....
.....
app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'
```

Add in a second decorator.

```
.....
.....
.....
.....
app.run()
```

Add code for the "do_search" function here.

## Sharpen your pencil Solution

You were to amend `hello_flask.py` to include a second URL, */search4*, writing the code that associates the URL with a function called `do_search`, which itself calls the `search4letters` function (from our `vsearch` module). You were to arrange for the `do_search` function to return the results determined when searching the phrase: "life, the universe, and everything!" for the string of characters: `'eiru,!'`.

Shown below is our existing code, with space reserved for the new code you need to write. Your job was to provide the missing code.

How does your code compare to ours?

You need to import the "search4letters" function from the "vsearch" module before you call it.

```
from flask import Flask

from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

@app.route('/search4')

def do_search() -> str:
    return str(search4letters('life, the universe, and everything', 'eiru,!'))

app.run()
```

A second decorator sets up the "/search4" URL.

The "do_search" function invokes "search4letters", then returns any results as a string.

To test this new functionality, you'll need to restart your Flask webapp, as it is currently running the older version of your code. To stop the webapp, return to your terminal window, then press Ctrl and C together. Your webapp will terminate, and you'll be returned to your operating system's prompt. Press the up arrow to recall the previous command (the one that previously started `hello_flask.py`) and then press the Enter key. The initial Flask status message reappears to confirm your updated webapp is waiting for requests:

Stop the webapp...

...then restart it.

```
$ python3 hello_flask.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [23/Nov/2015 20:15:46] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2015 21:30:26] "GET /doesthiswork.html HTTP/1.1" 404 -
^C
$ python3 hello_flask.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

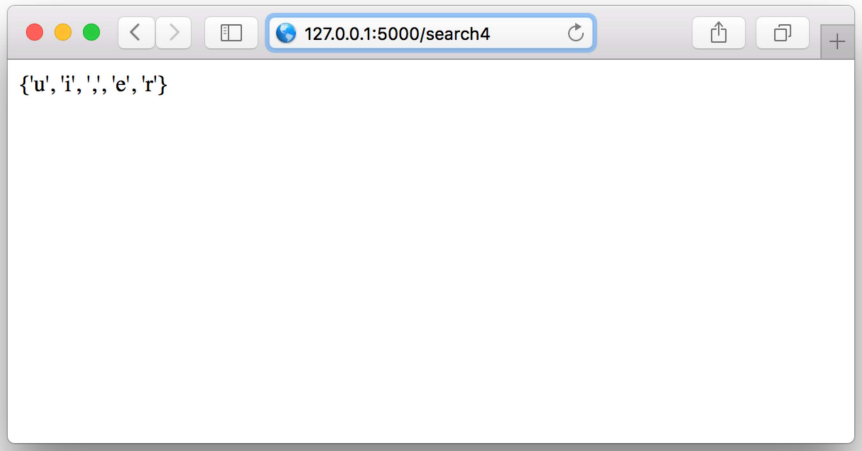
We are up and running again.

# Test Drive

As you haven't changed the code associated with the default '`/`' URL, that functionality still works, displaying the "Hello world from Flask!" message.

However, if you enter ***`http://127.0.0.1:5000/search4`*** into your browser's address bar, you'll see the results from the call to `search4letters`:

There are the results from the call to "search4letters". Granted, this output is nothing to get excited about, but it does prove that using the "/search4" URL invokes the function and returns the results.




## there are no Dumb Questions

**Q: I'm a little confused by the `127.0.0.1` and `:5000` parts of the URL used to access the webapp. What's the deal with those?**

A: At the moment, you're testing your webapp on your computer, which—because it's connected to the Internet—has its own unique IP address. Despite this fact, Flask doesn't use your IP address and instead connects its test web server to the Internet's **loopback address**: `127.0.0.1`, also commonly known as `localhost`. Both are shorthand for "my computer, no matter what its actual IP address is." For your web browser (also on your computer) to communicate with your Flask web server, you need to specify the address that is running your webapp, namely:`127.0.0.1`. This is a standard IP address reserved for this exact purpose.

The `:5000` part of the URL identifies the **protocol port number** your web server is running on.

Typically, web servers run on protocol port `80`, which is an Internet standard, and as such, doesn't need to be specified. You could type `oreilly.com:80` into your browser's address bar and it would work, but nobody does, as `oreilly.com` alone is sufficient (as the `:80` is assumed).

When you're building a webapp, it's very rare to test on protocol port `80` (as that's reserved for production servers), so most web frameworks choose another port to run on. `8080` is a popular choice for this, but Flask uses `5000` as its test protocol port.

**Q: Can I use some protocol port other than 5000 when I test and run my Flask webapp?**

A: Yes, `app.run()` allows you to specify a value for `port` that can be set to any value. But, unless you have a very good reason to change, stick with Flask's default of `5000` for now.

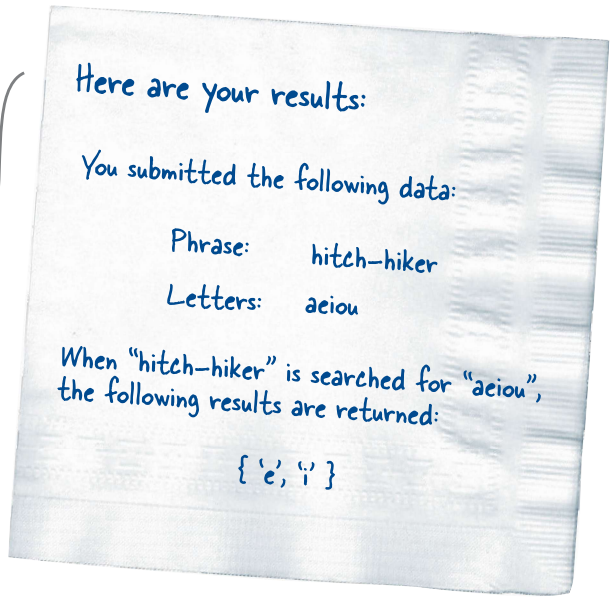
# Recall What We're Trying to Build

Our webapp needs a web page which accepts input, and another which displays the results of feeding the input to the `search4letters` function. Our current webapp code is nowhere near doing all of this, but what we have does provide a basis upon which to build what *is* required.

Shown below on the left is a copy of our current code, while on the right, we have copies of the "napkin specifications" from earlier in this chapter. We have indicated where we think the functionality for each napkin can be provided in the code:

```
from flask import Flask
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

@app.route('/search4')
def do_search() -> str:
    return str(search4letters( ... ))

app.run()
```

Note: to make everything fit, we aren't showing the entire line of code here.


## Here's the plan

Let's change the `hello` function to return the HTML form. Then we'll change the `do_search` function to accept the form's input, before calling the `search4letters` function. The results are then returned by `do_search` as another web page.

# Building the HTML Form

The required HTML form isn't all that complicated. Other than the descriptive text, the form is made up of two input boxes and a button.

## But...what if you're new to all this HTML stuff?

Don't panic if all this talk of HTML forms, input boxes, and buttons has you in a tizzy. Fear not, we have what you're looking for: the second edition of *Head First HTML and CSS* provides the best introduction to these technologies should you require a quick primer (or a speedy refresher).

Even if the thought of setting aside this book in order to bone up on HTML feels like too much work, note that we provide all the HTML you need to work with the examples in the book, and we do this without you having to be an HTML expert. A little exposure to HTML helps, but it's not a absolute requirement (after all, this is a book about Python, not HTML).


Note from Marketing: This is the book we wholeheartedly recommend for quickly getting up to speed with HTML...not that we're biased or anything. ☺

## Create the HTML, then send it to the browser

There's always more than one way to do things, and when it comes to creating HTML text from within your Flask webapp, you have choices:

I like to put my HTML inside **large strings**, which I then embed in my Python code, returning the strings as needed. That way, everything I need is right there in my code, and I have complete control... which is how I roll. What's not to like, Laura?

Well, Bob, putting all the HTML in your code works, but it doesn't scale. As your webapp gets bigger, all that embedded HTML gets kinda messy... and it's hard to hand off your HTML to a web designer to beautify. Nor is it easy to reuse chunks of HTML. Therefore, I always use a **template engine** with my webapps. It's a bit more work to begin with, but over time I find using templates really pays off...

**Laura's right—templates make HTML much easier to maintain than Bob's approach. We'll dive into templates on the next page.**

## Templates Up Close

Template engines let programmers apply the object-oriented notions of inheritance and reuse to the production of textual data, such as web pages.

A website's look and feel can be defined in a top-level HTML template, known as the **base template**, which is then inherited from by other HTML pages. If you make a change to the base template, the change is then reflected in *all* the HTML pages that inherit from it.

The template engine shipped with Flask is called *Jinja2*, and it is both easy to use and powerful. It is not this book's intention to teach you all you need to know about Jinja2, so what appears on these two pages is—by necessity—both brief and to the point. For more details on what's possible with Jinja2, see:

***http://jinja.pocoo.org/docs/dev/***

Here's the base template we'll use for our webapp. In this file, called `base.html`, we put the HTML markup that we want all of our web pages to share. We also use some Jinja2-specific markup to indicate content that will be supplied when HTML pages inheriting from this one are rendered (i.e., prepared prior to delivery to a waiting web browser). Note that markup appearing between `{{` and `}}`, as well as markup enclosed between `{%` and `%}`, is meant for the Jinja2 template engine: we've highlighted these cases to make them easy to spot:

This is standard HTML5 markup.

This is a Jinja2 directive, which indicates that a value will be provided prior to rendering (think of this as an argument to the template).

This is the base template.

```
<!doctype html>
<html>
    <head>
        <title>{{ the_title }}</title>
        <link rel="stylesheet" href="static/hf.css" />
    </head>
    <body>
        {% block body %}

        {% endblock %}
    </body>
</html>
```

This stylesheet defines the look and feel of all the web pages.

These Jinja2 directives indicate that a block of HTML will be substituted here prior to rendering, and is to be provided by any page that inherits from this one.

With the base template ready, we can inherit from it using Jinja2's `extends` directive. When we do, the HTML files that inherit need only provide the HTML for any named blocks in the base. In our case, we have only one named block: `body`.

Here's the markup for the first of our pages, which we are calling `entry.html`. This is markup for a HTML form that users can interact with in order to provide the value for `phrase` and `letters` expected by our webapp.

Note how the "boilerplate" HTML in the base template is not repeated in this file, as the `extends` directive includes this markup for us. All we need to do is provide the HTML that is specific to this file, and we do this by providing the markup within the Jinja2 block called `body`:

```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }}</h2>

<form method='POST' action='/search4'>
<table>
<p>Use this form to submit a search request:</p>
<tr><td>Phrase:</td><td><input name='phrase' type='TEXT' width='60'></td></tr>
<tr><td>Letters:</td><td><input name='letters' type='TEXT' value='aeiou'></td></tr>
</table>
<p>When you're ready, click this button:</p>
<p><input value='Do it!' type='SUBMIT'></p>
</form>

{% endblock %}
```

This template inherits from the base, and provides a replacement for the block called "body".

And, finally, here's the markup for the `results.html` file, which is used to render the results of our search. This template inherits from the base template, too:

```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }}</h2>

<p>You submitted the following data:</p>
<table>
<tr><td>Phrase:</td><td>{{ the_phrase }}</td></tr>
<tr><td>Letters:</td><td>{{ the_letters }}</td></tr>
</table>

<p>When "{{the_phrase }}" is search for "{{ the_letters }}", the following
results are returned:</p>
<h3>{{ the_results }}</h3>

{% endblock %}
```

As with "entry.html", this template also inherits from the base, and also provides a replacement for the block called "body".

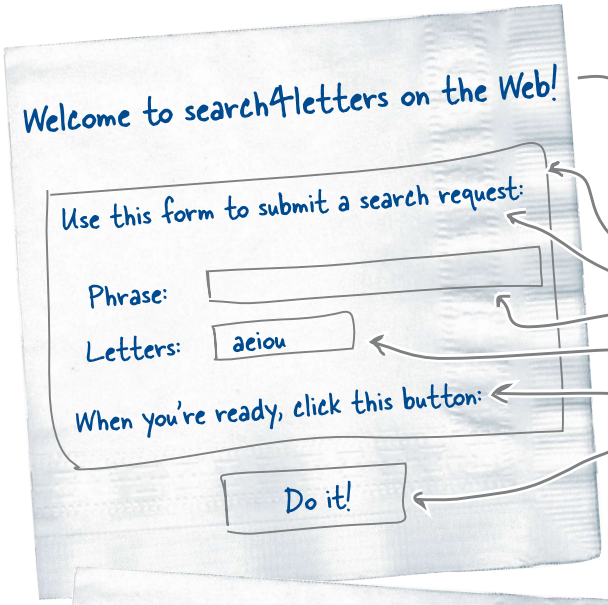
Note these additional argument values, which you need to provide values for prior to rendering.

# Templates Relate to Web Pages

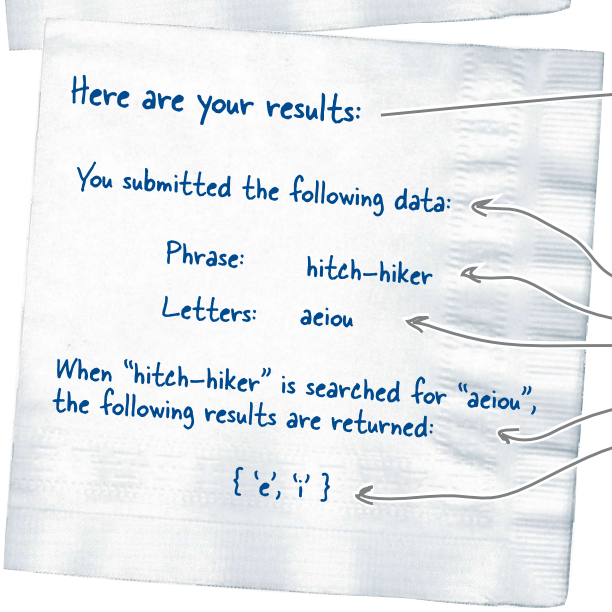
Our webapp needs to render two web pages, and now we have two templates that can help with this. Both templates inherit from the base template and thus inherit the base template's look and feel. Now all we need to do is render the pages.

**Download these templates (and the CSS) from here:** *http://python.itcarlow.ie/ed2/*.

Before we see how Flask (together with Jinja2) renders, let's take another look at our "napkin specifications" alongside our template markup. Note how the HTML enclosed within the Jinja2 `{% block %}` directive closely matches the hand-drawn specifications. The main omission is each page's title, which we'll provide in place of the `{{ the_title }}` directive during rendering. Think of each name enclosed in double curly braces as an argument to the template:




```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }}</h2>

<form method='POST' action='/search4'>
<table>
<p>Use this form to submit a search request:</p>
<tr><td>Phrase:</td><td><input name='phrase' type='TEXT'
width='60'></td></tr>
<tr><td>Letters:</td><td><input name='letters' type='TEXT'
value='aeiou'></td></tr>
</table>
<p>When you're ready, click this button:</p>
<p><input value='Do it!' type='SUBMIT'></p>
</form>

{% endblock %}
```




```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }}</h2>

<p>You submitted the following data:</p>
<table>
<tr><td>Phrase:</td><td>{{ the_phrase }}</td></tr>
<tr><td>Letters:</td><td>{{ the_letters }}</td></tr>
</table>

<p>When "{{the_phrase }}" is search for "{{ the_letters }}",
the following results are returned:</p>
<h3>{{ the_results }}</h3>

{% endblock %}
```

Don't forget those additional arguments.

# Rendering Templates from Flask

Flask comes with a function called `render_template`, which, when provided with the name of a template and any required arguments, returns a string of HTML when invoked. To use `render_template`, add its name to the list of imports from the `flask` module (at the top of your code), then invoke the function as needed.

Before doing so, however, let's rename the file containing our webapp's code (currently called `hello_flask.py`) to something more appropriate. You can use any name you wish for your webapp, but we're renaming our file `vsearch4web.py`. Here's the code currently in this file:

This code now resides in a file called "vsearch4web.py".

```
from flask import Flask
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

@app.route('/search4')
def do_search() -> str:
    return str(search4letters('life, the universe, and everything', 'eiru,!'))

app.run()
```

To render the HTML form in the `entry.html` template, we need to make a number of changes to the above code:

1. **Import the `render_template` function**
   Add `render_template` to the import list on the `from flask` line at the top of the code.

2. **Create a new URL—in this case, `/entry`**
   Every time you need a new URL in your Flask webapp, you need to add a new `@app.route` line, too. We'll do this before the `app.run()` line of code.

3. **Create a function that returns the correctly rendered HTML**
   With the `@app.route` line written, you can associate code with it by creating a function that does the actual work (and makes your webapp more useful to your users). The function calls (and returns the output from) the `render_template` function, passing in the name of the template file (`entry.html` in this case), as well as any argument values that are required by the template (in the case, we need a value for `the_title`).

**Let's make these changes to our existing code.**

# Displaying the Webapp's HTML Form

Let's add the code to enable the three changes detailed at the bottom of the last page. Follow along by making the same changes to your code:

**1** **Import the `render_template` function**

```
from flask import Flask, render_template
```

Add "render_template" to the list of technologies imported from the "flask" module.

**2** **Create a new URL—in this case, `/entry`**

```
@app.route('/entry')
```

Underneath the "do_search" function, but before the "app.run()" line, insert this line to add a new URL to the webapp.

**3** **Create a function that returns the correctly rendered HTML**

```
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')
```

Provide the name of the template to render.

Provide a value to associate with the "the_title" argument.

Add this function directly underneath the new "@app.route" line.

With these changes made, the code to our webapp—with the additions highlighted—now looks like this:

```
from flask import Flask, render_template
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

@app.route('/search4')
def do_search() -> str:
    return str(search4letters('life, the universe, and everything', 'eiru,!'))

@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

app.run()
```

We're leaving the rest of this code as is for now.

# Preparing to Run the Template Code

It's tempting to open a command prompt, then run the latest version of our code. However, for a number of reasons, this won't immediately work.

For starters, the base template refers to a stylesheet called `hf.css`, and this needs to exist in a folder called `static` (which is relative to the folder that contains your code). Here's a snippet of the base template that shows this:

**If you haven't done so already, download the templates and the CSS from here:**
***http://python.itcarlow.ie/ed2/.***

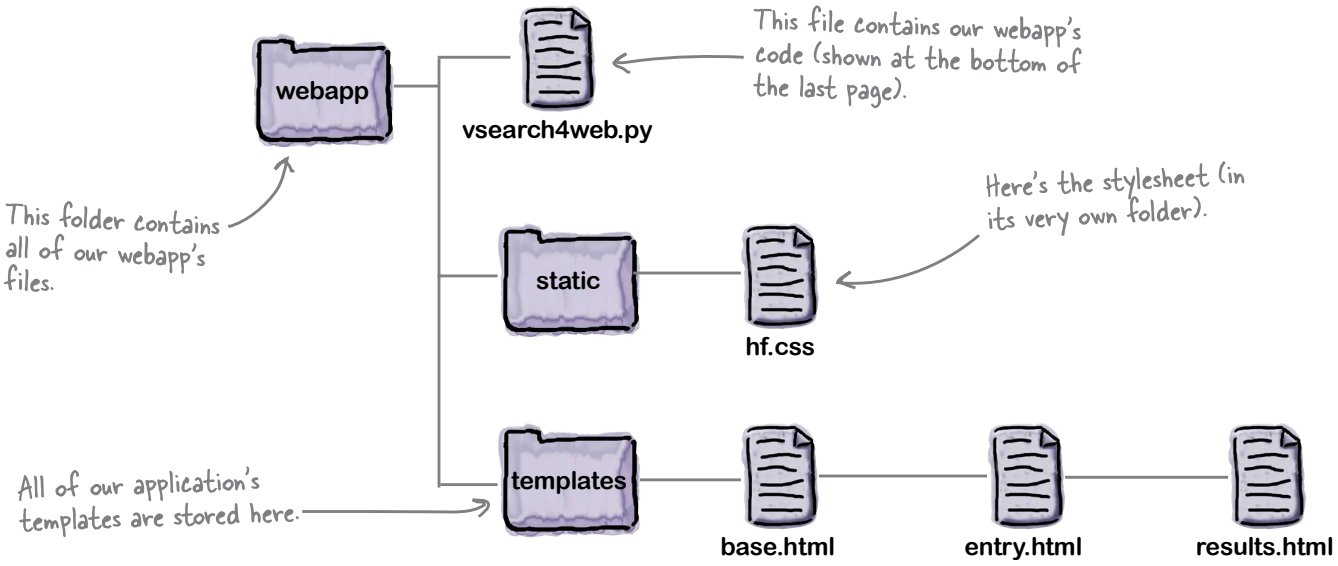
```
        ...
    <title>{{ the_title }}</title>
    <link rel="stylesheet" href="static/hf.css" />
</head>
        ...
```

The "hf.css" file needs to exist (in the "static" folder).

Feel free to grab a copy of the CSS file from this book's support website (see the URL at the side of this page). Just be sure to put the downloaded stylesheet in a folder called `static`.

In addition to this, Flask requires that your templates be stored in a folder called `templates`, which—like `static`—needs to be relative to the folder that contains your code. The download for this chapter also contains all three templates...so you can avoid typing in all that HTML!

Assuming that you've put your webapp's code file in a folder called `webapp`, here's the structure you should have in place prior to attempting to run the most recent version of `vsearch4web.py`:



# We're Ready for a Test Run

If you have everything ready—the stylesheet and templates downloaded, and the code updated—you're now ready to take your Flask webapp for another spin.

The previous version of your code is likely still running at your command prompt.

Return to that window now and press *Ctrl* and *C* together to stop the previous webapp's execution. Then press the *up arrow* key to recall the last command line, edit the name of the file to run, and then press *Enter*. Your new version of your code should now run, displaying the usual status messages:

```
        ...
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [23/Nov/2015 21:51:38] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2015 21:51:48] "GET /search4 HTTP/1.1" 200 -
^C
$ python3 vsearch4web.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Stop the webapp again...

Start up your new code (which is in the "vsearch4web.py" file).

The new code is up and running, and waiting to service requests.

Recall that this new version of our code still supports the */* and */search4* URLs, so if you use a browser to request those, the responses will be the same as shown earlier in this chapter. However, if you use this URL:

***http://127.0.0.1:5000/entry***

the response displayed in your browser should be the rendered HTML form (shown at the top of the next page). The command-prompt should display two additional status lines: one for the */entry* request and another related to your browser's request for the `hf.css` stylesheet:

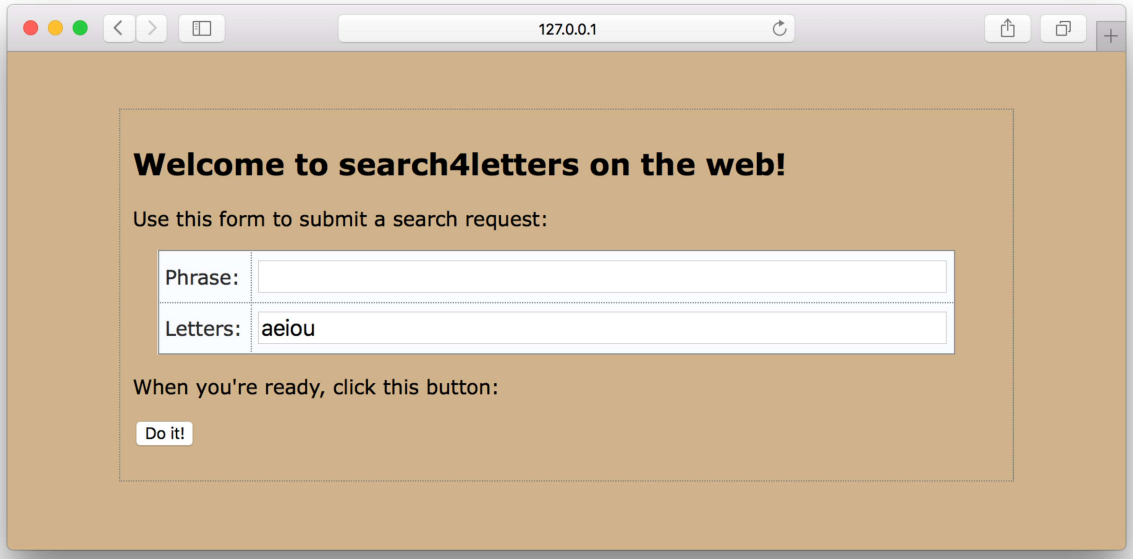
You request the HTML form....

...and your browser also requests the stylesheet.

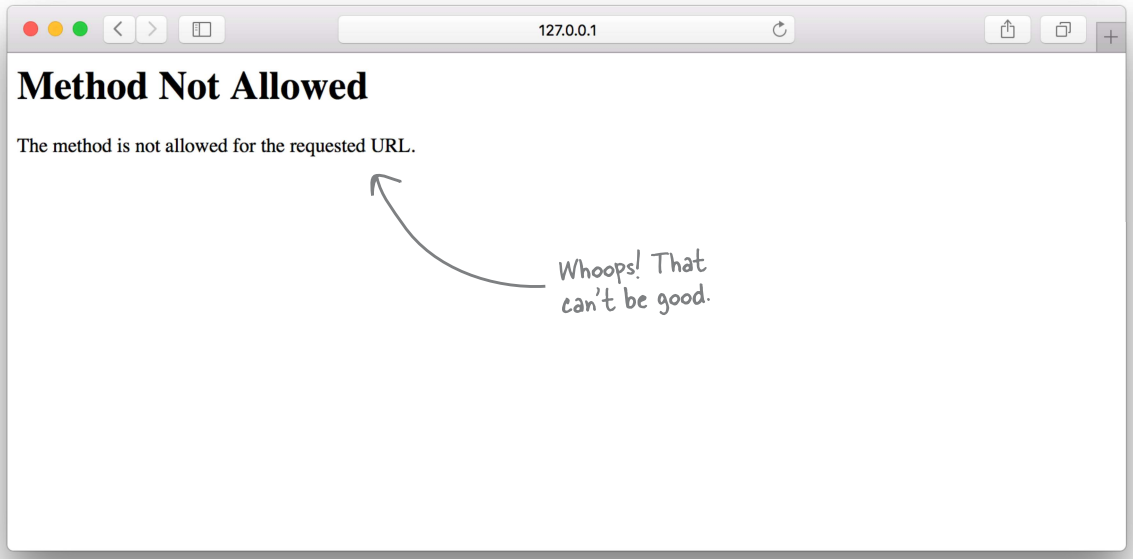
```
        ...
127.0.0.1 - - [23/Nov/2015 21:55:59] "GET /entry HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2015 21:55:59] "GET /static/hf.css HTTP/1.1" 304 -
```

# Test Drive

Here's what appears on screen when we type `http://127.0.0.1:5000/entry` into our browser:




We aren't going to win any web design awards for this page, but it looks OK, and resembles what we had on the back of our napkin. Unfortunately, when you type in a phrase and (optionally) adjust the Letters value to suit, clicking the *Do it!* button produces this error page:




This is a bit of a bummer, isn't it? Let's see what's going on.

# Understanding HTTP Status Codes

When something goes wrong with your webapp, the web server responds with a HTTP status code (which it sends to your browser). HTTP is the communications protocol that lets web browsers and servers communicate. The meaning of the status codes is well established (see the *Geek Bits* on the right). In fact, every web *request* generates an HTTP status code *response*.

To see which status code was sent to your browser from your webapp, review the status messages appearing at your command prompt. Here's what we saw:

```
    ...
127.0.0.1 - - [23/Nov/2015 21:55:59] "GET /entry HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2015 21:55:59] "GET /static/hf.css HTTP/1.1" 304 -
127.0.0.1 - - [23/Nov/2015 21:56:54] "POST /search4 HTTP/1.1" 405 -
```

Uh-oh. Something has gone wrong, and the server has generated a client-error status code.

The 405 status code indicates that the client (your browser) sent a request using a HTTP method that this server doesn't allow. There are a handful of HTTP methods, but for our purposes, you only need to be aware of two of them: *GET* and *POST*.

1. **The GET method**
   Browsers typically use this method to request a resource from the web server, and this method is by far the most used. (We say "typically" here as it is possible to—rather confusingly—use GET to *send* data from your browser to the server, but we're not focusing on that option here.) All of the URLs in our webapp currently support GET, which is Flask's default HTTP method.

2. **The POST method**
   This method allows a web browser to send data to the server over HTTP, and is closely associated with the HTML `<form>` tag. You can tell your Flask webapp to accept posted data from a browser by providing an extra argument on the `@app.route` line.

Let's adjust the `@app.route` line paired with our webapp's */search4* URL to accept posted data. To do this, return to your editor and edit the `vsearch4web.py` file once more.

## Geek Bits

Here's a quick and dirty explanation of the various HTTP status codes that can be sent from a web server (e.g., your Flask webapp) to a web client (e.g., your web browser).

There are five main categories of status code: 100s, 200s, 300s, 400s, and 500s.

Codes in the **100–199** range are **informational** messages: all is OK, and the server is providing details related to the client's request.

Codes in the **200–299** range are **success** messages: the server has received, understood, and processed the client's request. All is good.

Codes in the **300–399** range are **redirection** messages: the server is informing the client that the request can be handled elsewhere.

Codes in the **400–499** range are **client error** messages: the server received a request from the client that it does not understand and can't process. Typically, the client is at fault here.

Codes in the **500–599** range are **server error** messages: the server received a request from the client, but the server failed while trying to process it. Typically, the server is at fault here.

For more details, please see: *https://en.wikipedia.org/wiki/List_of_HTTP_status_codes*.

# Handling Posted Data

As well as accepting the URL as its first argument, the `@app.route` decorator accepts other, optional arguments.

One of these is the `methods` argument, which lists the HTTP method(s) that the URL supports. By default, Flask supports GET for all URLs. However, if the `methods` argument is assigned a list of HTTP methods to support, this default behavior is overridden. Here's what the `@app.route` line currently looks like:

```
@app.route('/search4')
```

We have not specified an HTTP method to support here, so Flask defaults to GET.

To have the */search4* URL support `POST`, add the `methods` argument to the decorator and assign the list of HTTP methods you want the URL to support. This line of code, below, states that the */search4* URL now only supports the POST method (meaning GET requests are no longer supported):

```
@app.route('/search4', methods=['POST'])
```

The "/search4" URL now supports only the POST method.

This small change is enough to rid your webapp of the "Method Not Allowed" message, as the POST associated with the HTML form matches up with the POST on the `@app.route` line:

This HTML snippet is from "entry.html"...

```
...
<form method='POST' action='/search4'>
<table>
...
```

...and this Python code is from the "vsearch4web.py" file.

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

Note how HTML uses "method" (singular), whereas Flask uses "methods" (plural).

## there are no Dumb Questions

**Q: What if I need my URL to support both the GET method as well as POST? Is that possible?**

A: Yes, all you need to do is add the name of the HTTP method you need to support to the list assigned to the `methods` arguments. For example, if you wanted to add GET support to the */search4* URL, you need only change the `@app.route` line of code to look like this: `@app.route('/search4', methods=['GET', 'POST'])`. For more on this, see the Flask docs, which are available here *http://flask.pocoo.org*.

# Refining the Edit/Stop/Start/Test Cycle

At this point, having saved our amended code, it's a reasonable course of action to stop the webapp at the command prompt, then restart it to test our new code. This edit/stop/start/test cycle works, but becomes tedious after a while (especially if you end up making a long series of small changes to your webapp's code).

To improve the efficiency of this process, Flask allows you to run your webapp in *debugging mode*, which, among other things, automatically restarts your webapp every time Flask notices your code has changed (typically as a result of you making and saving a change). This is worth doing, so let's switch on debugging by changing the last line of code in `vsearch4web.py` to look like this:

```
app.run(debug=True)
```

Switches on debugging

Your program code should now look like this:

```
from flask import Flask, render_template
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

@app.route('/search4', methods=['POST'])
def do_search() -> str:
    return str(search4letters('life, the universe, and everything', 'eiru,!'))

@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

app.run(debug=True)
```

We are now ready to take this code for a test run. To do so, stop your currently running webapp (for the last time) by pressing *Ctrl-C*, then restart it at your command prompt by pressing the *up arrow* and *Enter*.

Rather than showing the usual "`Running on http://127...`" message, Flask spits out three new status lines, which is its way of telling you debugging mode is now active. Here's what we saw on our computer:

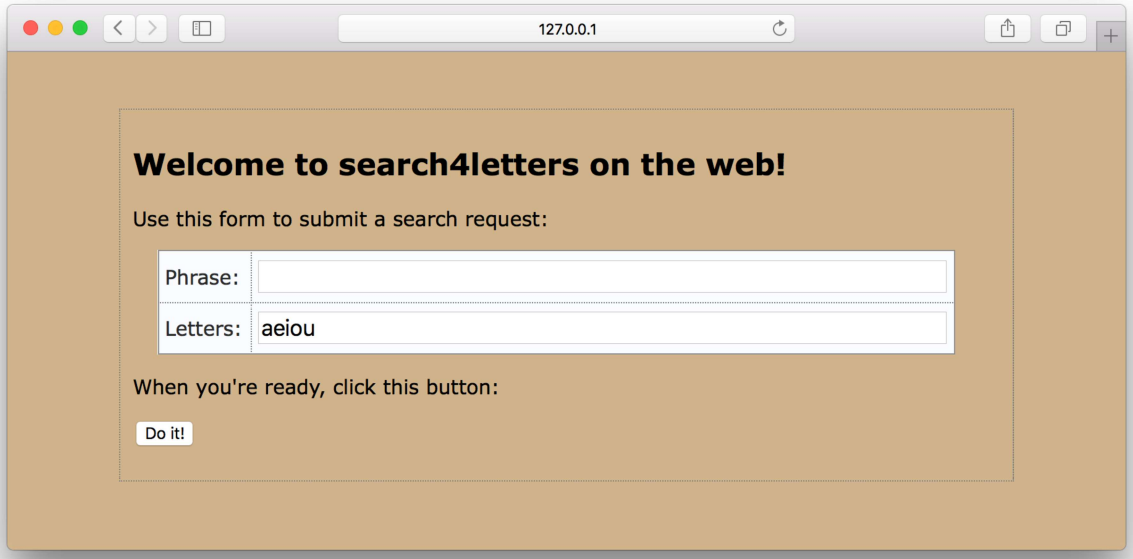
```
$ python3 vsearch4web.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 228-903-465
```

This is Flask's way of telling you that your webapp will automatically restart if your code changes. Also: don't worry if your debugger pin code is different from ours (that's OK). We won't use this pin.

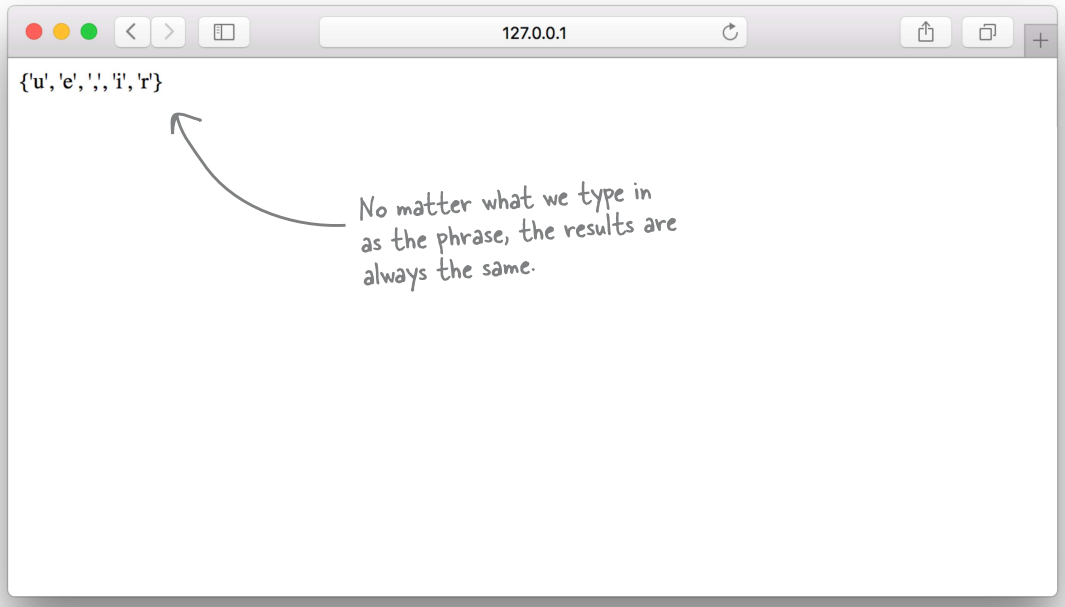
Now that we are up and running again, let's interact with our webapp once more and see what's changed.

# Test Drive

Return to the entry form by typing `http://127.0.0.1:5000/entry` into your browser:




The "Method Not Allowed" error has gone, but things still aren't working right. You can type any phrase into this form, then click the *Do it!* button without the error appearing. If you try it a few times, you'll notice that the results returned are always the same (no matter what phrase or letters you use). Let's investigate what's going on here.



# Accessing HTML Form Data with Flask

Our webapp no longer fails with a "Method Not Allowed" error. Instead, it always returns the same set of characters: *u*, *e*, *comma*, *i*, and *r*. If you take a quick look at the code that executes when the */search4* URL is posted to, you'll see why this is: the values for `phrase` and `letters` are *hardcoded* into the function:

```
              ...

@app.route('/search4', methods=['POST'])
def do_search() -> str:
    return str(search4letters('life, the universe, and everything', 'eiru,!'))

              ...
```

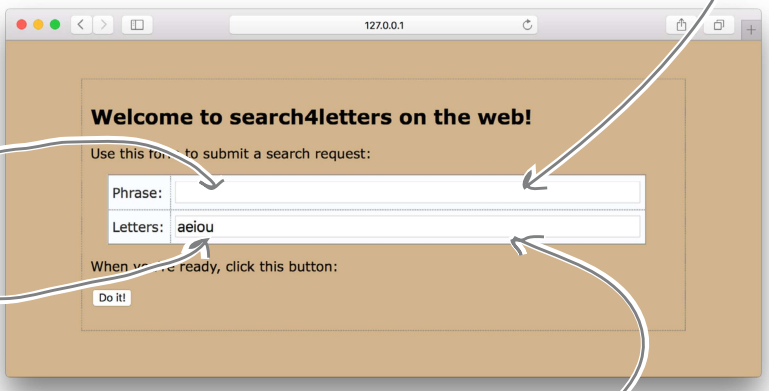
No matter what we type into the HTML form, our code is always going to use these hardcoded values.

Our HTML form posts its data to the web server, but in order to do something with the data, we need to amend our webapp's code to accept the data, then perform some operation on it.

Flask comes with a built-in object called `request` that provides easy access to posted data. The `request` object contains a dictionary attribute called `form` that provides access to a HTML form's data posted from the browser. As `form` is like any other Python dictionary, it supports the same square bracket notation you first saw in Chapter 3. To access a piece of data from the form, put the form element's name inside square brackets:

The data from this form element is available in our webapp's code as "request.form['phrase']".

```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }}</h2>

<form method='POST' action='/search4'>
<table>
<p>Use this form to submit a search request:</p>
<tr><td>Phrase:</td><td><input name='phrase' type='TEXT'
width='60'></td></tr>
<tr><td>Letters:</td><td><input name='letters' type='TEXT'
value='aeiou'></td></tr>
</table>
<p>When you're ready, click this button:</p>
<p><input value='Do it!' type='SUBMIT'></p>
</form>

{% endblock %}
```




The HTML template (in the "entry.html" file)

The rendered form in our web browser

The data from this form element is available in our webapp as "request.form['letters']".

# Using Request Data in Your Webapp

To use the `request` object, import it on the `from flask` line at the top of your program code, then access the data from `request.form` as needed. For our purposes, we want to replace the hardcoded data value in our `do_search` function with the data from the form. Doing so ensures that every time the HTML form is used with different values for `phrase` and `letters`, the results returned from our webapp adjust accordingly.

Let's make these changes to our program code. Start by adding the `request` object to the list of imports from Flask. To do that, change the first line of `vsearch4web.py` to look like this:

```
from flask import Flask, render_template, request
```

Add "request" to the list of imports.

We know from the information on the last page that we can access the `phrase` entered into the HTML form within our code as `request.form['phrase']`, whereas the entered `letters` is available to us as `request.form['letters']`. Let's adjust the `do_search` function to use these values (and remove the hardcoded strings):

Create two new variables...

```
@app.route('/search4', methods=['POST'])
def do_search() -> str:
    phrase = request.form['phrase']
    letters = request.form['letters']
    return str(search4letters(phrase, letters))
```

...and assign the HTML form's data to the newly created variables...

...then, use the variables in the call to "search4letters".

## Automatic Reloads

Now...before you do anything else (having made the changes to your program code above) save your `vsearch4web.py` file, then flip over to your command prompt and take a look at the status messages produced by your webapp. Here's what we saw (you should see something similar):

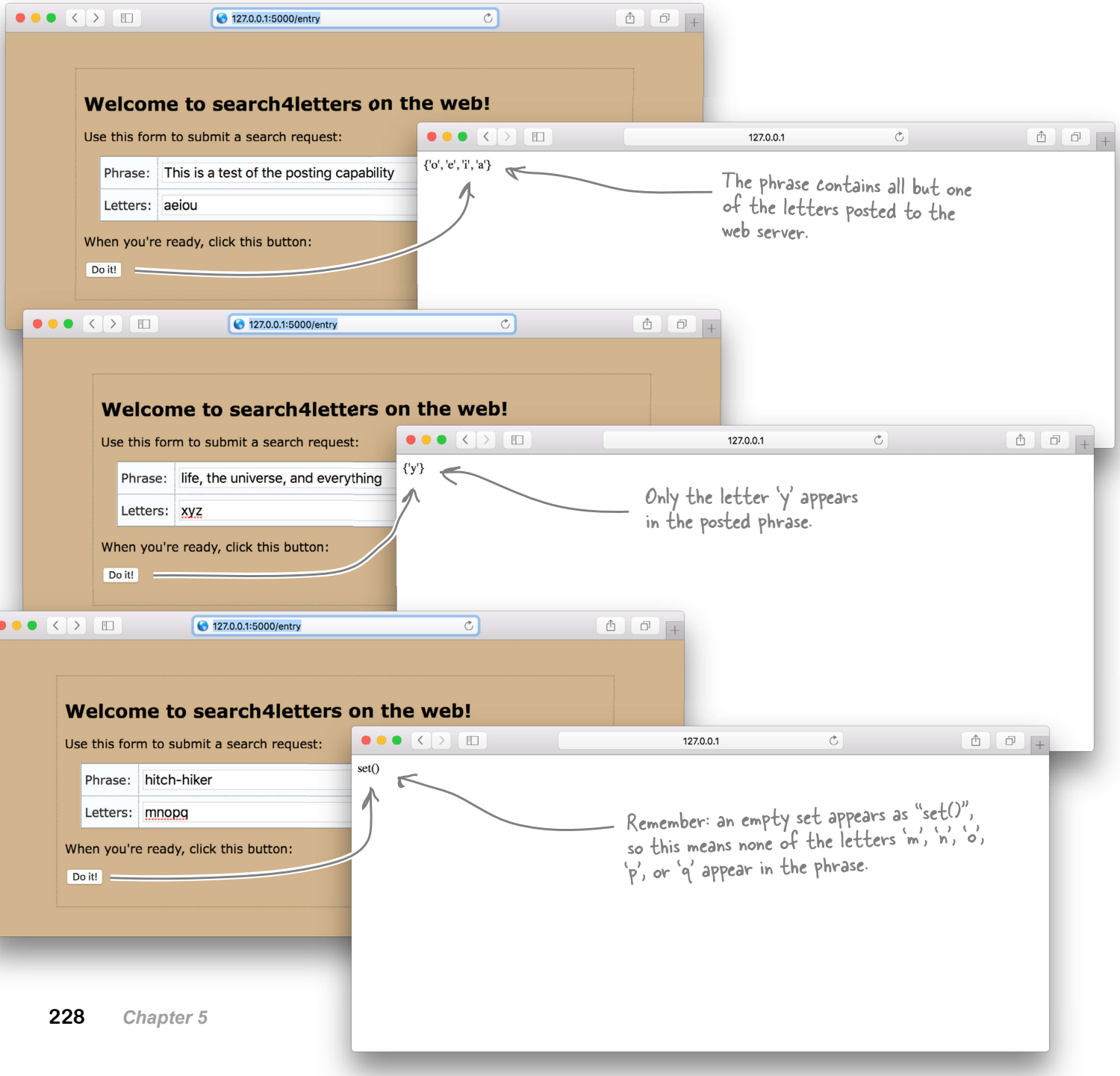
```
$ python3 vsearch4web.py
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 228-903-465
127.0.0.1 - - [23/Nov/2015 22:39:11] "GET /entry HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2015 22:39:11] "GET /static/hf.css HTTP/1.1" 200 -
127.0.0.1 - - [23/Nov/2015 22:17:58] "POST /search4 HTTP/1.1" 200 -
 * Detected change in 'vsearch4web.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 228-903-465
```

The Flask debugger has spotted the code changes, and restarted your webapp for you. Pretty handy, eh?

Don't panic if you see something other than what's shown here. Automatic reloading only works if the code changes you make are correct. If your code has errors, the webapp bombs out to your command prompt. To get going again, fix your coding errors, then restart your webapp manually (by pressing the *up arrow*, then *Enter*).

# Test Drive

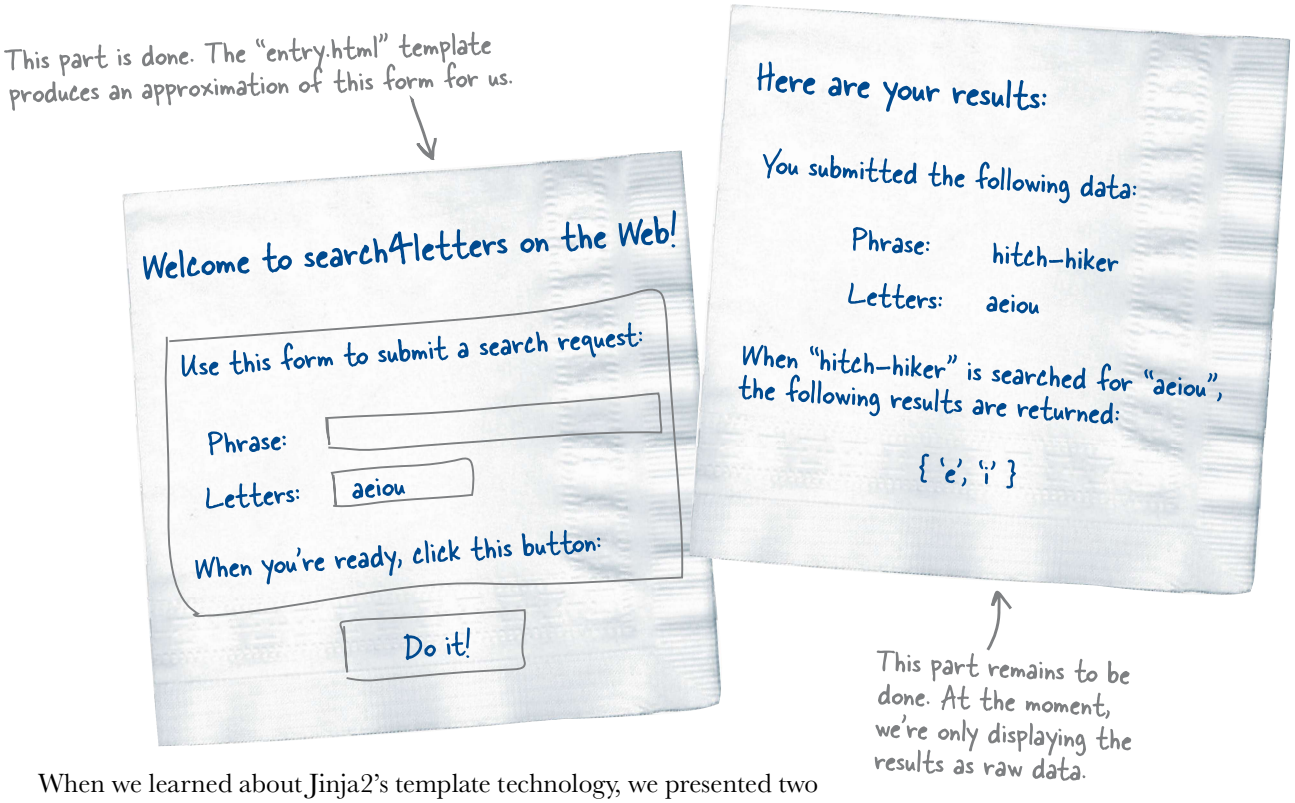
Now that we've changed our webapp to accept (and process) the data from our HTML form, we can throw different phrases and letters at it, and it should do the right thing:



# Producing the Results As HTML

At this point, the functionality associated with our webapp is working: any web browser can submit a `phrase/letters` combination, and our webapp invokes `search4letters` on our behalf, returning any results. However, the output produced isn't really a HTML webpage—it's just the raw data returned as text to the waiting browser (which displays it on screen).

Recall the back-of-the-napkin specifications from earlier in this chapter. This is what we were hoping to produce:




When we learned about Jinja2's template technology, we presented two HTML templates. The first, `entry.html`, is used to produce the form. The second, `results.html`, is used to display the results. Let's use it now to take our raw data output and turn it into HTML.

## there are no Dumb Questions

**Q: It is possible to use Jinja2 to template textual data other than HTML?**

A: Yes. Jinja2 is a text template engine that can be put to many uses. That said, its typical use case is with web development projects (as used here with Flask), but there's nothing stopping you from using it with other textual data if you really want to.

# Calculating the Data We Need

Let's remind ourselves of the contents of the `results.html` template as presented earlier in this chapter. The Jinja2-specific markup is highlighted:

This is "results.html".

```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }}</h2>

<p>You submitted the following data:</p>
<table>
<tr><td>Phrase:</td><td>{{ the_phrase }}</td></tr>
<tr><td>Letters:</td><td>{{ the_letters }}</td></tr>
</table>

<p>When "{{the_phrase }}" is search for "{{ the_letters }}", the following
results are returned:</p>
<h3>{{ the_results }}</h3>

{% endblock %}
```

The highlighted names enclosed in double curly braces are Jinja2 variables that take their value from corresponding variables in your Python code. There are four of these variables: `the_title`, `the_phrase`, `the_letters`, and `the_results`. Take another look at the `do_search` function's code (below), which we are going to adjust in just a moment to render the HTML template shown above. As you can see, this function already contains two of the four variables we need to render the template (and to keep things as simple as possible, we've used variable names in our Python code that are similar to those used in the Jinja2 template):

Here are two of the four values we need.

```
@app.route('/search4', methods=['POST'])
def do_search() -> str:
    phrase = request.form['phrase']
    letters = request.form['letters']
    return str(search4letters(phrase, letters))
```

The two remaining required template arguments (`the_title` and `the_results`) still need to be created from variables in this function and assigned values.

We can assign the `"Here are your results:"` string to `the_title`, and then assign the call to `search4letters` to `the_results`. All four variables can then be passed into the `results.html` template as arguments prior to rendering.

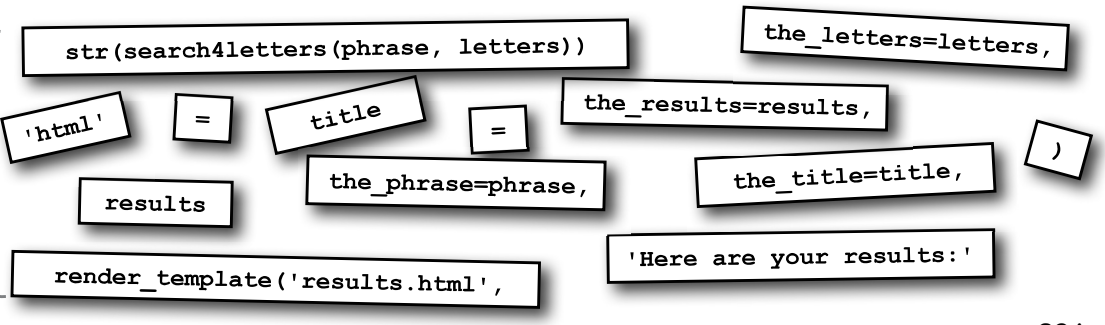
# Template Magnets

The *Head First* authors got together and, based on the requirements for the updated `do_search` function outlined at the bottom of the last page, wrote the code required. In true *Head First* style, they did so with the help of some coding magnets...and a fridge (best if you don't ask). Upon their success, the resulting celebrations got so rowdy that a certain *series editor* bumped into the fridge (while singing the *beer song*) and now the magnets are all over the floor. Your job is to stick the magnets back in their correct locations in the code.

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

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

@app.route('/search4', methods=['POST'])
def do_search() ->_____:
    phrase = request.form['phrase']
    letters = request.form['letters']
    _____
    _____
    return _____
                          _____
                          _____
                          _____
                          _____

@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

app.run(debug=True)
```

Decide which code magnet goes in each of the dashed-line locations.

Here are the magnets you have to work with.



# Template Magnets Solution

Having made a note to keep a future eye on a certain series editor's beer consumption, you set to work restoring all of the code magnets for the updated `do_search` function. Your job was to stick the magnets back in their correct locations in the code.

Here's what we came up with when we performed this task:

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

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    return render_template('results.html',
                           the_phrase=phrase,
                           the_letters=letters,
                           the_title=title,
                           the_results=results,
                           )

@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

app.run(debug=True)
```

Change the annotation to indicate that this function now returns HTML, not a plain-text string (as in the previous version of this code).

Create a Python variable called "title"...

...and assign a string to "title".

Create another Python variable called "results"...

...and assign the results of the call to "search4letters" to "results".

Render the "results.html" template. Remember: this template expects four argument values.

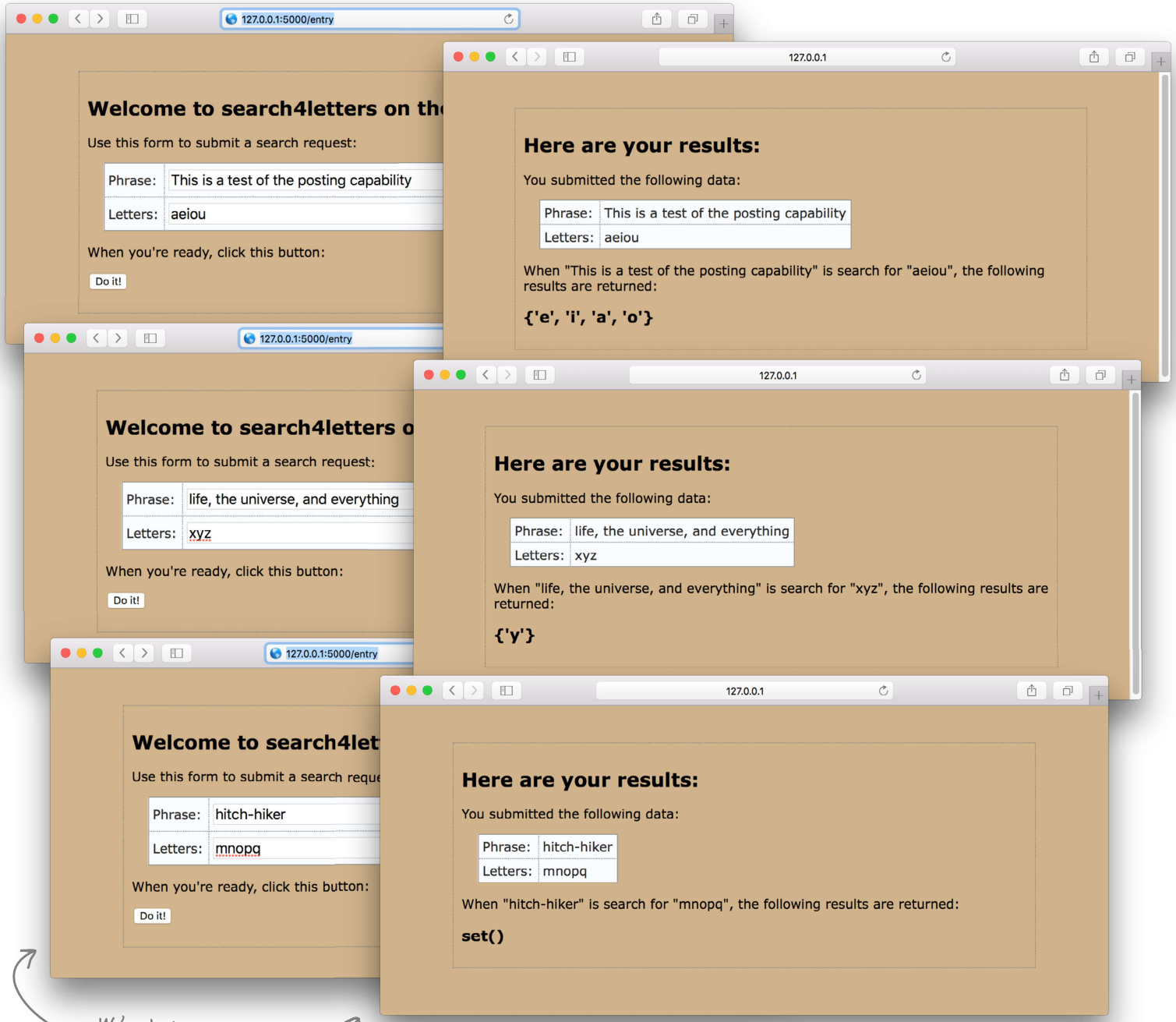
Each Python variable is assigned to its corresponding Jinja2 argument. In this way, data from our program code is passed into the template.

Don't forget the closing parenthesis to end the function call.

Now that the magnets are back in their correct locations, make these code changes to your copy of `vsearch4web.py`. Be sure to save your file to ensure that Flask automatically reloads your webapp. We're now ready for another test.

# Test Drive

Let's test the new version of our webapp using the same examples from earlier in this chapter. Note that Flask restarted your webapp the moment you saved your code.




We're looking good for input and output now.

# Adding a Finishing Touch

Let's take another look at the code that currently makes up `vsearch4web.py`. Hopefully, by now, all this code should make sense to you. One small syntactical element that often confuses programmers moving to Python is the inclusion of the final comma in the call to `render_template`, as most programmers feel this should be a syntax error and shouldn't be allowed. Although it does look somewhat strange (at first), Python allows it—but does not require it—so we can safely move on and not worry about it:

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

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

app.run(debug=True)
```

This extra comma looks a little strange, but is perfectly fine (though optional) Python syntax.

This version of our webapp supports three URLs: */*, */search4*, and */entry*, with some dating back to the very first Flask webapp we created (right at the start of this chapter). At the moment, the */* URL displays the friendly, but somewhat unhelpful, "Hello world from Flask!" message.

We could remove this URL and its associated `hello` function from our code (as we no longer need either), but doing so would result in a 404 "Not Found" error in any web browser contacting our webapp on the */* URL, which is the default URL for most webapps and websites. To avoid this annoying error message, let's ask Flask to redirect any request for the */* URL to the */entry* URL. We do this by adjusting the `hello` function to return a HTML `redirect` to any web browser that requests the */* URL, effectively substituting the */entry* URL for any request made for */*.

# Redirect to Avoid Unwanted Errors

To use Flask's redirection technology, add `redirect` to the `from flask` import line (at the top of your code), then change the `hello` function's code to look like this:

```
from flask import Flask, render_template, request, redirect
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> '302':
    return redirect('/entry')

    ...
```

Add "redirect" to the list of imports.

Adjust the annotation to more clearly indicate what's being returned by this function. Recall that HTTP status codes in the 300–399 range are redirections, and 302 is what Flask sends back to your browser when "redirect" is invoked.

The rest of the code remains unchanged.

Call Flask's "redirect" function to instruct the browser to request an alternative URL (in this case, "/entry").

This small edit ensures our webapp's users are shown the HTML form should they request the */entry* or / URL.

Make this change, save your code (which triggers an automatic reload), and then try pointing your browser to each of the URLs. The HTML form should appear each time. Take a look at the status messages being displayed by your webapp at your command prompt. You may well see something like this:

```
    ...
 * Detected change in 'vsearch4web.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 228-903-465
127.0.0.1 - - [24/Nov/2015 16:54:13] "GET /entry HTTP/1.1" 200 -
127.0.0.1 - - [24/Nov/2015 16:56:43] "GET / HTTP/1.1" 302 -
127.0.0.1 - - [24/Nov/2015 16:56:44] "GET /entry HTTP/1.1" 200 -
```

You saved your code, so Flask reloaded your webapp.

A request is made for the "/entry" URL, and it is served up immediately. Note the 200 status code (and remember from earlier in this chapter that codes in the 200–299 range are success messages: the server has received, understood, and processed the client's request).

When a request is made for the "/" URL, our webapp first responds with the 302 redirection, and then the web browser sends another request for the "/entry" URL, which is successfully served up by our webapp (again, note the 200 status code).

As a strategy, our use of redirection here works, but it is somewhat wasteful—a single request for the / URL turns into two requests every time (although client-side caching can help, this is still not optimal). If only Flask could somehow associate more than one URL with a given function, effectively removing the need for the redirection altogether. That would be nice, wouldn't it?

# Functions Can Have Multiple URLs

It's not hard to guess where we are going with this, is it?

It turns out that Flask can indeed associate more than one URL with a given function, which can reduce the need for redirections like the one demonstrated on the last page. When a function has more than one URL associated with it, Flask tries to match each of the URLs in turn, and if it finds a match, the function is executed.

It's not hard to take advantage of this Flask feature. To begin, remove `redirect` from the `from flask` import line at the top of your program code; we no longer need it, so let's not import code we don't intend to use. Next, using your editor, cut the `@app.route('/')` line of code and then paste it above the `@app.route('/entry')` line near the bottom of your file. Finally, delete the two lines of code that make up the `hello` function, as our webapp no longer needs them.

When you're done making these changes, your program code should look like this:

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

app = Flask(__name__)

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

app.run(debug=True)
```

We no longer need to import "redirect", so we've removed it from this import line.

The "hello" function has been removed.

The "entry_page" function now has two URLs associated with it.

Saving this code (which triggers a reload) allows us to test this new functionality. If you visit the / URL, the HTML form appears. A quick look at your webapp's status messages confirms that processing / now results in one request, as opposed to two (as was previously the case):

```
      ...
 * Detected change in 'vsearch4web.py', reloading
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 228-903-465
127.0.0.1 - - [24/Nov/2015 16:59:10] "GET / HTTP/1.1" 200 -
```

As always, the new version of our webapp reloads.

One request, one response. That's more like it. ☺

# Updating What We Know

We've just spent the last 40 pages creating a small webapp that exposes the functionality provided by our `search4letters` function to the World Wide Web (via a simple two-page website). At the moment, the webapp runs locally on your computer. In a bit, we'll discuss deploying your webapp to the cloud, but for now let's update what you know:

## BULLET POINTS

- You learned about the Python Package Index (**PyPI**), which is a centralized repository for third-party Python modules. When connected to the Internet, you can automatically install packages from PyPI using `pip`.
- You used `pip` to install the **Flask** micro-web framework, which you then used to build your webapp.
- The `__name__` value (maintained by the interpreter) identifies the currently active namespace (more on this later).
- The `@` symbol before a function's name identifies it as a **decorator**. Decorators let you change the behavior of an existing function without having to change the function's code. In your webapp, you used Flask's `@app.route` decorator to associate URLs with Python functions. A function can be decorated more than once (as you saw with the `do_search` function).
- You learned how to use the **Jinja2** text template engine to render HTML pages from within your webapp.

## Is that all there is to this chapter?

You'd be forgiven for thinking this chapter doesn't introduce much new Python. It doesn't. However, one of the points of this chapter was to show you just how few lines of Python code you need to produce something that's generally useful on the Web, thanks in no small part to our use of Flask. Using a template technology helps a lot, too, as it allows you to keep your Python code (your webapp's logic) separate from your HTML pages (your webapp's user interface).

It's not an awful lot of work to extend this webapp to do more. In fact, you could have an HTML whiz-kid produce more pages for you while you concentrate on writing the Python code that ties everything together. As your webapp scales, this separation of duties really starts to pay off. You get to concentrate on the Python code (as you're the programmer on the project), whereas the HTML whiz-kid concentrates on the markup (as that's their bailiwick). Of course, you both have to learn a little bit about Jinja2 templates, but that's not too difficult, is it?

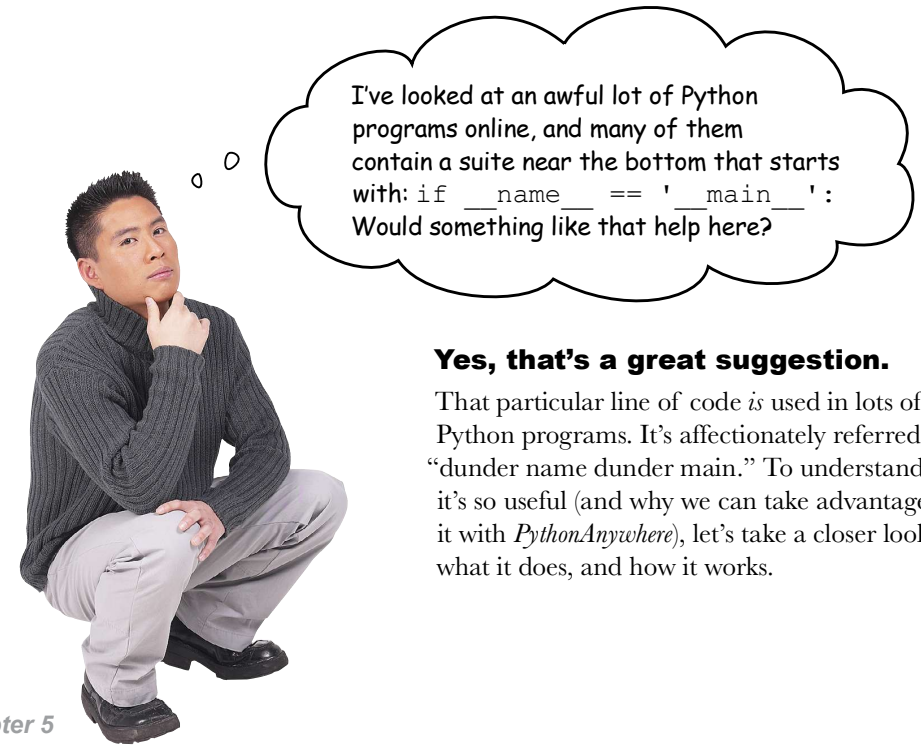
# Preparing Your Webapp for the Cloud

With your webapp working to specification locally on your computer, it's time to think about deploying it for use by a wider audience. There are lots of options here, with many different web-based hosting setups available to you as a Python programmer. One popular service is cloud-based, hosted on AWS, and is called *PythonAnywhere*. We love it over at *Head First Labs*.

Like nearly every other cloud-hosted deployment solution, *PythonAnywhere* likes to control how your webapp starts. For you, this means *PythonAnywhere* assumes responsibility for calling `app.run()` on your behalf, which means you no longer need to call `app.run()` in your code. In fact, if you try to execute that line of code, *PythonAnywhere* simply refuses to run your webapp.

A simple solution to this problem would be to remove that last line of code from your file *before* deploying to the cloud. This certainly works, but means you need to put that line of code back in again whenever you run your webapp locally. If you're writing and testing new code, you should do so locally (not on *PythonAnywhere*), as you use the cloud for deployment only, not for development. Also, removing the offending line of code effectively amounts to you having to maintain two versions of the same webapp, one with and one without that line of code. This is never a good idea (and gets harder to manage as you make more changes).

It would be nice if there were a way to selectively execute code based on whether you're running your webapp locally on your computer or remotely on *PythonAnywhere*...




**Yes, that's a great suggestion.**

That particular line of code *is* used in lots of Python programs. It's affectionately referred to as "dunder name dunder main." To understand why it's so useful (and why we can take advantage of it with *PythonAnywhere*), let's take a closer look at what it does, and how it works.

## Dunder Name Dunder Main Up Close

To understand the programming construct suggested at the bottom of the last page, let's look at a small program that uses it, called `dunder.py`. This three-line program begins by displaying a message on screen that prints the currently active namespace, stored in the `__name__` variable. An `if` statement then checks to see whether the value of `__name__` is set to `__main__`, and—if it is—another message is displayed confirming the value of `__name__` (i.e., the code associated with the `if` suite executes):

The "dunder.py" program code—all three lines of it.

```
print('We start off in:', __name__)
if __name__ == '__main__':
    print('And end up in:', __name__)
```

Displays the value of "__name__".

Displays the value of "__name__" if it is set to "__main__".

Use your editor (or IDLE) to create the `dunder.py` file, then run the program at a command prompt to see what happens. If you're on *Windows*, use this command:

```
C:\> py  -3  dunder.py
```

If you are on *Linux* or *Mac OS X*, use this command:

```
$ python3  dunder.py
```

No matter which operating system you're running, the `dunder.py` program—when executed *directly* by Python—produces this output on screen:

```
We start off in: __main__
And end up in: __main__
```

When executed directly by Python, both calls to "print" display output.

So far, so good.

Now, look what happens when we import the `dunder.py` file (which, remember, is *also* a module) into the >>> prompt. We're showing the output on *Linux/Mac OS X* here. To do the same thing on *Windows*, replace `python3` (below) with `py -3`:

```
$ python3
Python 3.5.1 ...
Type "help", "copyright", "credits" or "license" for more information.
>>> import dunder
We start off in: dunder
```

Look at this: there's only a single line displayed (as opposed to two), as "__name__" has been set to "dunder" (which is the name of the imported module).

Here's the bit you need to understand: if your program code is executed *directly* by Python, an `if` statement like the one in `dunder.py` returns `True`, as the active namespace is `__main__`. If, however, your program code is imported as a module (as in the Python Shell prompt example above), the `if` statement always returns `False`, as the value of `__name__` is not `__main__`, but the name of the imported module (`dunder` in this case).

# Exploiting Dunder Name Dunder Main

Now that you know what *dunder name dunder main* does, let's exploit it to solve the problem we have with *PythonAnywhere* wanting to execute `app.run()` on our behalf.

It turns out that when *PythonAnywhere* executes our webapp code, it does so by importing the file that contains our code, treating it like any other module. If the import is successful, *PythonAnywhere* then calls `app.run()`. This explains why leaving `app.run()` at the bottom of our code is such a problem for *PythonAnywhere*, as it assumes the `app.run()` call has not been made, and fails to start our webapp when the `app.run()` call has been made.

To get around this problem, wrap the `app.run()` call in a *dunder name dunder main* `if` statement (which ensures `app.run()` is never executed when the webapp code is imported).

Edit `vsearch4web.py` one last time (in this chapter, anyway) and change the final line of code to this:

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

The "app.run()" line of code now only runs when executed directly by Python.

This small change lets you continue to execute your webapp locally (where the `app.run()` line *will* execute) as well as deploy your webapp to *PythonAnywhere* (where the `app.run()` line *won't* execute). No matter where your webapp runs, you've now got one version of your code that does the right thing.

## Deploying to PythonAnywhere (well... almost)

All that remains is for you to perform that actual deployment to *PythonAnywhere's* cloud-hosted environment.

Note that, for the purposes of this book, deploying your webapp to the cloud is *not* an absolute requirement. Despite the fact that we intend to extend `vsearch4web.py` with additional functionality in the next chapter, you do not need to deploy to *PythonAnywhere* to follow along. You can happily continue to edit/run/test your webapp locally as we extend it in the next chapter (and beyond).

However, if you really do want to deploy to the cloud, see *Appendix B*, which provides step-by-step instructions on how to complete the deployment on *PythonAnywhere*. It's not hard, and won't take more than 10 minutes.

Whether you're deploying to the cloud or not, we'll see you in the next chapter, where we'll start to look at some of the options available for saving data from within your Python programs.

# Chapter 5's Code

```
from flask import Flask
from vsearch import search4letters

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

@app.route('/search4')
def do_search() -> str:
    return str(search4letters('life, the universe, and everything', 'eiru,!'))

app.run()
```

This is "hello_flask.py", our first webapp based on Flask (one of Python's micro-web framework technologies).

This is "vsearch4web.py". This webapp exposed the functionality provided by our "search4letters" function to the World Wide Web. In addition to Flask, this code exploited the Jinja2 template engine.

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

app = Flask(__name__)

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to... web!')

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

This is "dunder.py", which helped us understand the very handy "dunder name dunder main" mechanism.

```
print('We start off in:', __name__)
if __name__ == '__main__':
    print('And end up in:', __name__)
```

# 6 storing and manipulating data

# Where to Put Your Data


**Sooner or later, you'll need to safely store your data somewhere.**

And when it comes to **storing data**, Python has you covered. In this chapter, you'll learn about storing and retrieving data from *text files*, which—as storage mechanisms go—may feel a bit simplistic, but is nevertheless used in many problem areas. As well as storing and retrieving your data from files, you'll also learn some tricks of the trade when it comes to manipulating data. We're saving the "serious stuff" (storing data in a database) until the next chapter, but there's plenty to keep us busy for now when working with files.

# Doing Something with Your Webapp's Data

At the moment, your webapp (developed in Chapter 5) accepts input from any web browser (in the form of a `phrase` and some `letters`), performs a `search4letters` call, and then returns any results to the waiting web browser. Once done, your webapp discards any data it has.


There are a bunch of questions that we could ask of the data our webapp uses. For instance: *How many requests have been responded to? What's the most common list of letters? Which IP addresses are the requests coming from? Which browser is being used the most?* and so on, and so forth.

In order to begin answering these (and other) questions, we need to save the webapp's data as opposed to simply throwing it away. The suggestion above makes perfect sense: let's log data about each web request, then—once we have the logging mechanism in place—go about answering any questions we have.

# Python Supports Open, Process, Close

No matter the programming language, the easiest way to store data is to save it to a text file. Consequently, Python comes with built-in support for *open, process, close*. This common technique lets you **open** a file, **process** its data in some way (reading, writing, and/or appending data), and then **close** the file when you're done (which saves your changes).

Here's how to use Python's *open, process, close* technique to open a file, process it by appending some short strings to it, and then close the file. As we're only experimenting for now, let's run our code at the Python >>> shell.

We start by calling `open` on a file called `todos.txt`, using *append mode*, as our plan is to add data to this file. If the call to `open` succeeds, the interpreter returns an object (known as a *file stream*) which is an alias for the actual file. The object is assigned to a variable and given the name `todos` (although you could use whichever name you wish here):

Open a file...

...which has this filename...

```
>>> todos = open('todos.txt', 'a')
```

...and open the file in "append-mode".

If all is OK, "open" returns a file stream, which we've assigned to this variable.

The `todos` variable lets you refer to your file in your code (other programming languages refer to this as a *file handle*). Now that the file is open, let's write to it using `print`. Note how, below, `print` takes an extra argument (`file`), which identifies the file stream to write to. We have three things to remember to do (it's never-ending, really), so we call `print` three times:

We print a message...

...to the file stream.

```
>>> print('Put out the trash.', file=todos)
>>> print('Feed the cat.', file=todos)
>>> print('Prepare tax return.', file=todos)
```

As we have nothing else to add to our to-do list, let's close the file by calling the `close` method, which is made available by the interpreter to every file stream:

```
>>> todos.close()
```

We're done, so let's tidy up after ourselves by closing the file stream.

If you forget to call `close`, you could *potentially* lose data. Remembering to always call `close` is important.

## Geek Bits

To access the >>> prompt:

- run IDLE on your computer;
- run the `python3` command in a *Linux* or *Mac OS X* terminal; or
- use `py -3` at a *Windows* command line.

# Reading Data from an Existing File

Now that you've added some lines of data to the `todos.txt` file, let's look at the *open, process, close* code needed to read the saved data from the file and display it on screen.

Rather than opening the file in append mode, this time you are only interested in reading from the file. As reading is `open`'s **default mode**, you don't need to provide a mode argument; the name of the file is all you need here. We're not using `todos` as the alias for the file in this code; instead, we'll refer to the open file by the name `tasks` (as before, you can use whichever variable name you want to here):

"Reading" is the "open" function's default mode.

Open a file... ...which has this filename.

```
>>> tasks = open('todos.txt')
```

If all is OK, "open" returns a file stream, which we've assigned to this variable.

Let's now use `tasks` with a `for` loop to read each individual line from the file. When we do this, the `for` loop's iteration variable (`chore`) is assigned the current line of data as read from the file. Each iteration assigns a line of data to `chore`. When you use a file stream with Python's `for` loop, the interpreter is smart enough to read a line of data from the file each time the loop iterates. It's also smart enough to terminate the loop when there's no more data to read:

Think of "chore" as an alias for the line in the file.

The "tasks" variable is the file stream.

```
>>> for chore in tasks:
...     print(chore)
...
Put out the trash.

Feed the cat.

File tax return.
```

The output shows the data from the "todos.txt" file. Note how the loop ends when we run out of lines to read.

As you are merely reading from an already written-to file, calling `close` is less critical here than when you are writing data. But it's always a good idea to close a file when it is no longer needed, so call the `close` method when you're done:

```
>>> tasks.close()
```

We're done, so let's tidy up after ourselves by closing the file stream.

## there are no Dumb Questions

**Q: What's the deal with the extra newlines on output? The data in the file is three lines long, but the `for` loop produced six lines of output on my display. What gives?**

A: Yes, the `for` loop's output does look strange, doesn't it? To understand what's happening, consider that the `print` function appends a newline to everything it displays on screen *as its default behavior*. When you combine this with the fact that each line in the file ends in a newline character (and the newline is read in as part of the line), you end up printing two newlines: the one from the file together with the one from `print`. To instruct `print` not to include the second newline, change `print(chore)` to `print(chore, end='')`. This has the effect of suppressing `print`'s newline-appending behavior, so the extra newlines no longer appear on screen.

**Q: What other modes are available to me when I'm working with data in files?**

A: There are a few, which we've summarized in the following *Geek Bits* box. (That's a great question, BTW.)

## Geek Bits

The first argument to `open` is the name of the file to process. The second argument is **optional**. It can be set to a number of different values, and dictates the **mode** the file is opened in. Modes include "reading," "writing," and "appending." Here are the most common mode values, where each (except for `'r'`) creates a new empty file if the file named in the first argument doesn't already exist:

- `'r'` Open a file for **reading**. This is the default mode and, as such, is optional. When no second argument is provided, `'r'` is assumed. It is also assumed that the file being read from already exists.
- `'w'` Open a file for **writing**. If the file already contains data, empty the file of its data before continuing.
- `'a'` Open a file for **appending**. Preserve the file's contents, adding any new data to the end of the file (compare this behavior to `'w'`).
- `'x'` Open a **new file** for writing. Fail if the file already exists (compare this behavior to `'w'` and to `'a'`).

By default, files open in **text** mode, where the file is assumed to contain lines of textual data (e.g., ASCII or UTF-8). If you are working with nontextual data (e.g., an image file or an MP3), you can specify **binary** mode by adding "`b`" to any of the modes (e.g., `'wb'` means "write to a binary data"). If you include "+" as part of the second argument, the file is opened for reading *and* writing (e.g., `'x+b'` means "read from and write to a new binary file"). Refer to the Python docs for more details on `open` (including information on its other optional arguments).

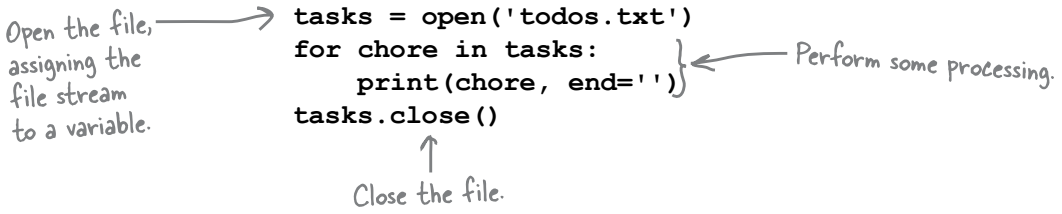
I've looked at a bunch of Python projects on GitHub, and most of them use a "with" statement when opening files. What's the deal with that?

### The *`with`* statement is more convenient.

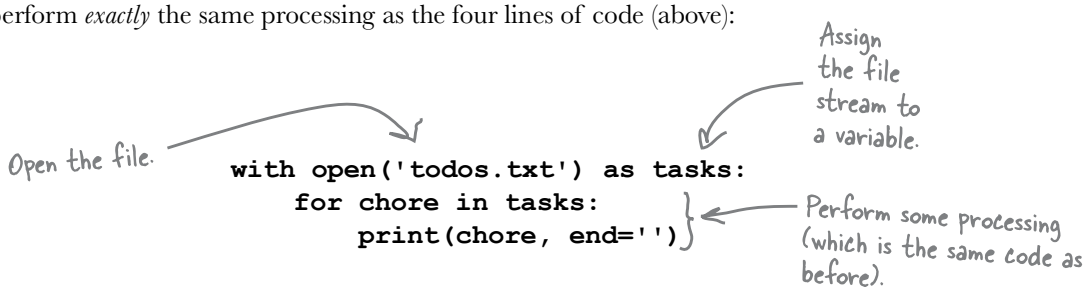
Although using the `open` function together with the `close` method (with a bit of processing in the middle) works fine, most Python programmers shun *open, process, close* in favor of the `with` statement. Let's take some time to find out why.

# A Better Open, Process, Close: "with"

Before we describe why `with` is so popular, let's take a look at some code that uses `with`. Here is the code we wrote (two pages ago) to read in and display the current contents of our `todos.txt` file. Note that we've adjusted the `print` function call to suppress the extra newline on output:




Let's rewrite this code to use a `with` statement. These next three lines of code use `with` to perform *exactly* the same processing as the four lines of code (above):




Notice anything missing? The call to `close` does not make an appearance. The `with` statement is smart enough to remember to call `close` *on your behalf* whenever its suite of code ends.

This is actually much more useful than it initially sounds, as lots of programmers often forget to call `close` when they're done processing a file. This is not such a big deal when all you're doing is reading from a file, but when you're writing to a file, forgetting to call close can potentially cause *data loss* or *data corruption*. By relieving you of the need to remember to always call `close`, the `with` statement lets you concentrate on what it is you're actually doing with the data in the open file.

**Python supports "open, process, close." But most Python programmers prefer to use the "with" statement.**

## The "with" statement manages context

The `with` statement conforms to a coding convention built into Python called the **context management protocol**. We're deferring a detailed discussion of this protocol until later in this book. For now, all you have to concern yourself with is the fact that when you use `with` when working with files, you can forget about calling `close`. The `with` statement is managing the context within which its suite runs, and when you use `with` and `open` together, the interpreter cleans up after you, calling `close` as and when required.


Let's put what you now know about working with files to use. Here is the current code for your webapp. Give it another read before we tell you what you have to do:

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

app = Flask(__name__)

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

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

This is the "vsearch4web.py" code from Chapter 5.

Your job is to write a new function, called `log_request`, which takes two arguments: `req` and `res`. When invoked, the `req` argument is assigned the current Flask request object, while the `res` argument is assigned the results from calling `search4letters`. The `log_request` function's suite should append the value of `req` and `res` (as one line) to a file called `vsearch.log`. We've got you started by providing the function's `def` line. You are to provide the missing code (hint: use `with`):

```
def log_request(req: 'flask_request', res: str) -> None:

```

Write this function's suite here.

## Exercise Solution

Your job was to write a new function, called `log_request`, which takes two arguments: `req` and `res`. When invoked, the `req` argument is assigned the current Flask request object, while the `res` argument is assigned the results from calling `search4letters`. The `log_request` function's suite should append the value of `req` and `res` (as one line) to a file called `vsearch.log`. We got you started—you were to provide the missing code:

This annotation may have thrown you a little. Recall that function annotations are meant to be read by other programmers. They are documentation, not executable code: the Python interpreter always ignores them, so you can use any annotation descriptor you like.

```
def log_request(req: 'flask_request', res: str) -> None:
    with open('vsearch.log', 'a') as log :
        print(req, res, file=log)
```

Use "with" to open "vsearch.log" in append mode.

Call the "print" BIF to write the values of "req" and "res" to the opened file.

Note the file stream is called "log" in this code.

This annotation uses Python's "None" value to indicate this function has no return value.

## Invoking the logging function

Now that the `log_request` function exists, when do we invoke it?

Well, for starters, let's add the `log_request` code into the `vsearch4web.py` file. You can put it anywhere in this file, but we inserted it directly above the `do_search` function and its associated `@app.route` decorator. We did this because we're going to invoke it from within the `do_search` function, and putting it above the calling function seems like a good idea.

We need to be sure to call `log_request` before the `do_search` function ends, but after the `results` have been returned from the call to `search4letters`. Here's a snippet of `do_search`'s code showing the inserted call:

Call the "log_request" function here.

```
    ...
phrase = request.form['phrase']
letters = request.form['letters']
title = 'Here are your results:'
results = str(search4letters(phrase, letters))
log_request(request, results)
return render_template('results.html',
    ...
```

# A Quick Review

Before taking this latest version of `vsearch4web.py` for a spin, let's check that your code is the same as ours. Here's the entire file, with the latest additions highlighted:

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

app = Flask(__name__)

def log_request(req: 'flask_request', res: str) -> None:
    with open('vsearch.log', 'a') as log:
        print(req, res, file=log)

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

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

Here are the latest additions, which arrange to log each web request to a file called "vsearch.log".

You may have noticed that none of our webapp's functions contain comments. This is a deliberate omission on our part (as there's only so much room on these pages, and something had to give). Note that any code you download from this book's support website always includes comments.

## Take your webapp for a spin...

Start up this version of your webapp (if required) at a command prompt. On *Windows*, use this command:

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

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

```
$ python3 vsearch4web.py
```

With your webapp up and running, let's log some data via the HTML form.

# Test Drive

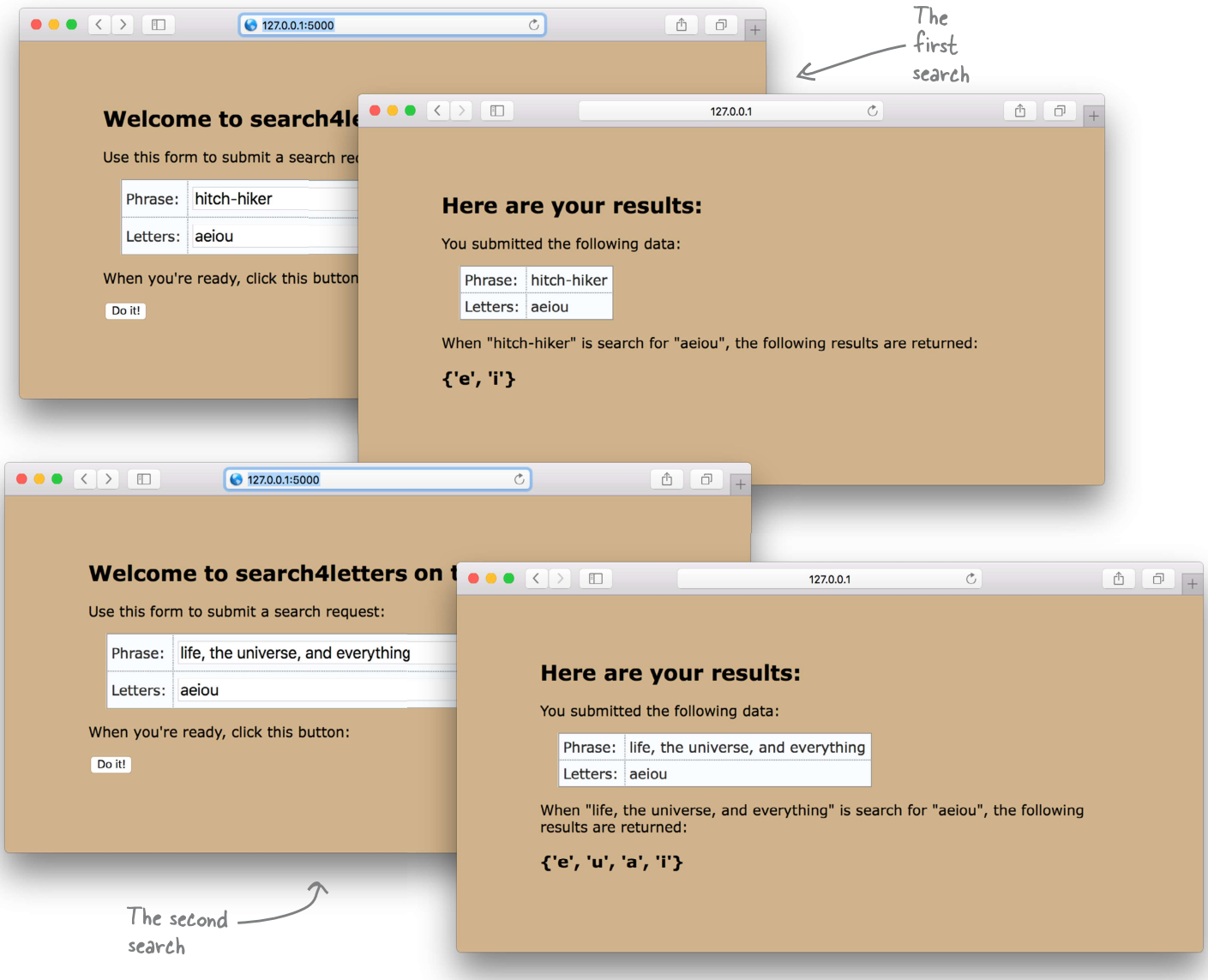
Use your web browser to submit data to it via the webapp's HTML form. If you want to follow along with what we're doing, submit three searches using the following values for `phrase` and `letters`:

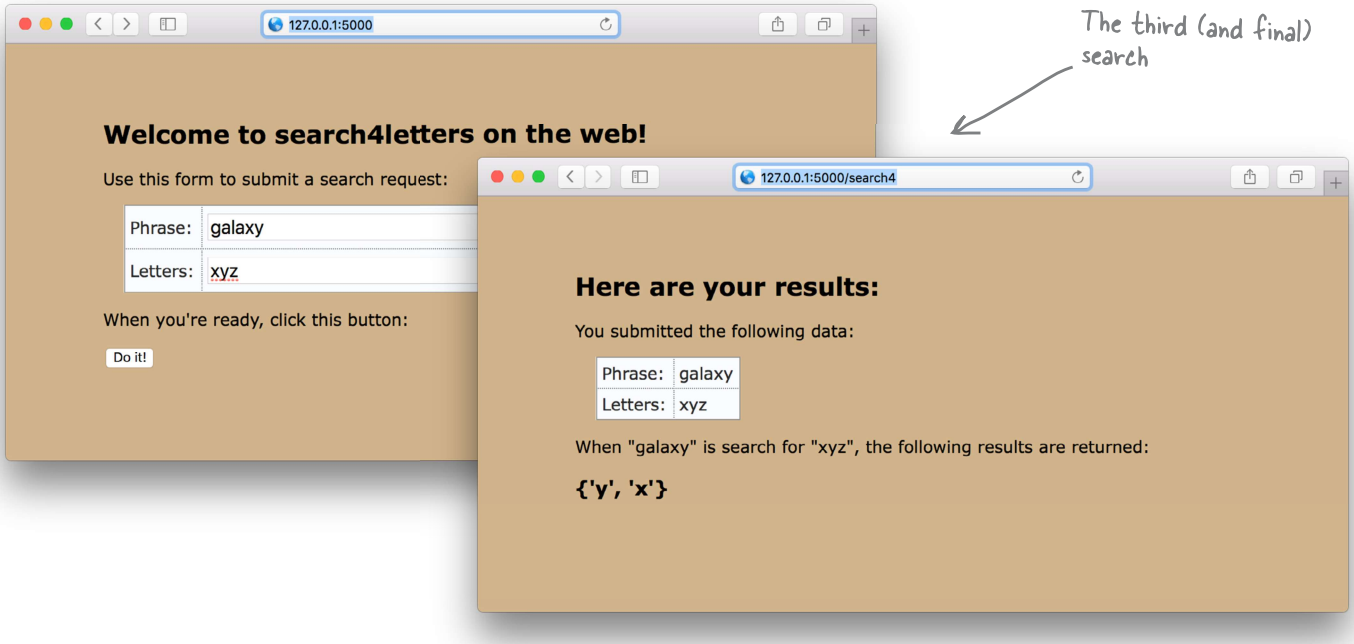
`hitch-hiker` with `aeiou`.

`life, the universe, and everything` with `aeiou`.

`galaxy` with `xyz`.

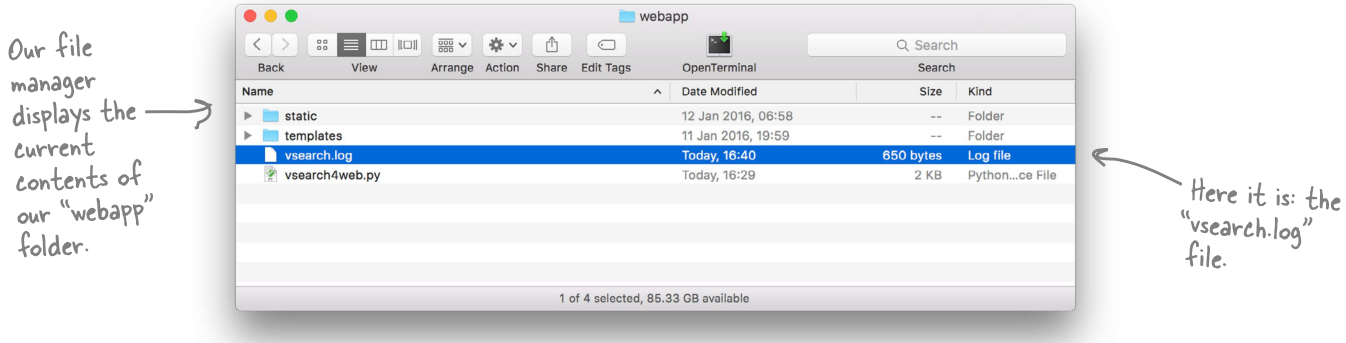
Before you begin, note that the `vsearch.log` file does not yet exist.






## Data is logged (behind the scenes)

Each time the HTML form is used to submit data to the webapp, the `log_request` function saves details of the web request and writes the results to the log file. Immediately after the first search, the `vsearch.log` file is created in the same folder as your webapp's code:




It's tempting to consider using your text editor to view the `vsearch.log` file's contents. But where's the *fun* in that? As this is a webapp, let's provide access to the logged data via the webapp itself. That way, you'll never have to move away from your web browser when interacting with your webapp's data. Let's create a new URL, called */viewlog*, which displays the log's contents on demand.

# View the Log Through Your Webapp

You're going to add support for the */viewlog* URL to your webapp. When your webapp receives a request for */viewlog*, it should open the `vsearch.log` file, read in all of its data, and then send the data to the waiting browser.

Most of what you need to do you already know. Start by creating a new `@app.route` line (we're adding this code near the bottom of `vsearch4web.py`, just above the *dunder name dunder main* line):

```
@app.route('/viewlog')
```

We have a brand new URL.

Having decided on the URL, next we'll write a function to go with it. Let's call our new function `view_the_log`. This function won't take any arguments, and will return a string to its caller; the string will be concatenation of all of the lines of data from the `vsearch.log` file. Here's the function's `def` line:

```
def view_the_log() -> str:
```

And we have a brand new function, which (according to the annotation) returns a string.

Now to write the function's suite. You have to open the file *for reading*. This is the `open` function's default mode, so you only need the name of the file as an argument to `open`. Let's manage the context within which our file processing code executes using a `with` statement:

```
with open('vsearch.log') as log:
```

Open the log file for reading.

Within the `with` statement's suite, we need to read all the lines from the file. Your first thought might be to loop through the file, reading each line as you go. However, the interpreter provides a `read` method, which, when invoked, returns the *entire* contents of the file "in one go." Here's the single line of code that does just that, creating a new string called `contents`:

```
contents = log.read()
```

Read the entire file "in one go" and assign it to a variable (which we've called "contents").

With the file read, the `with` statement's suite ends (closing the file), and you are now ready to send the data back to the waiting web browser. This is straightforward:

```
return contents
```

Take the list of lines in "contents" and return them.

With everything put together, you now have all the code you need to respond to the */viewlog* request; it looks like this:

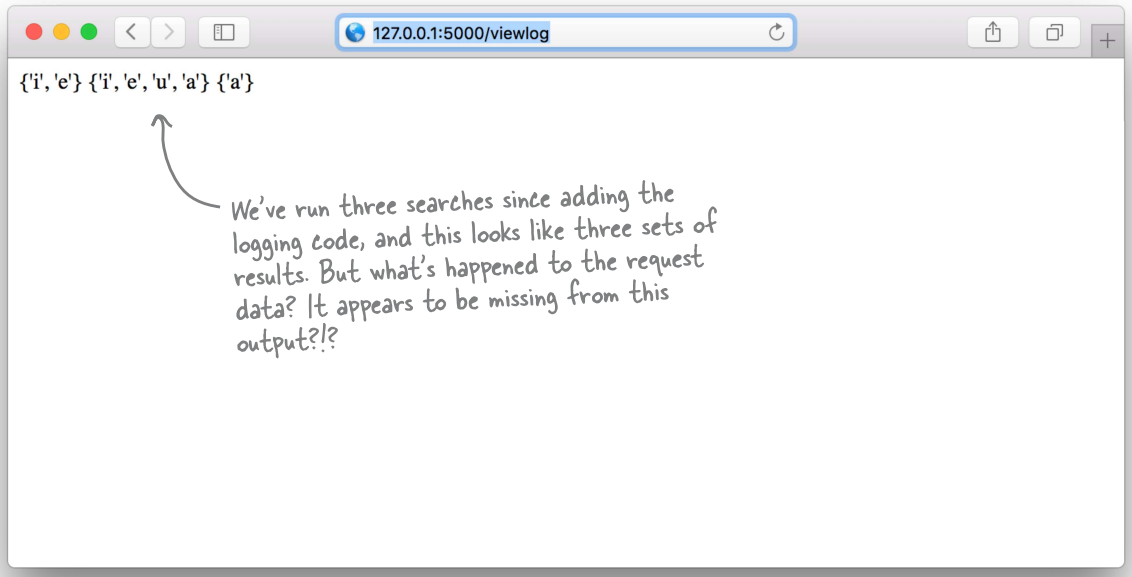
This is all of the code you need to support the "/viewlog" URL.

```
@app.route('/viewlog')
def view_the_log() -> str:
    with open('vsearch.log') as log:
        contents = log.read()
    return contents
```

# Test Drive

With the new code added and saved, your webapp should automatically reload. You can enter some new searches if you like, but the ones you ran a few pages ago are already logged. Any new searches you perform will be appended to the log file. Let's use the */viewlog* URL to take a look at what's been saved. Type **`http://127.0.0.1:5000/viewlog`** into your browser's address bar.

Here's what we saw when we used *Safari* on *Mac OS X* (we also checked *Firefox* and *Chrome*, and got the same output):




## Where to start when things go wrong with your output

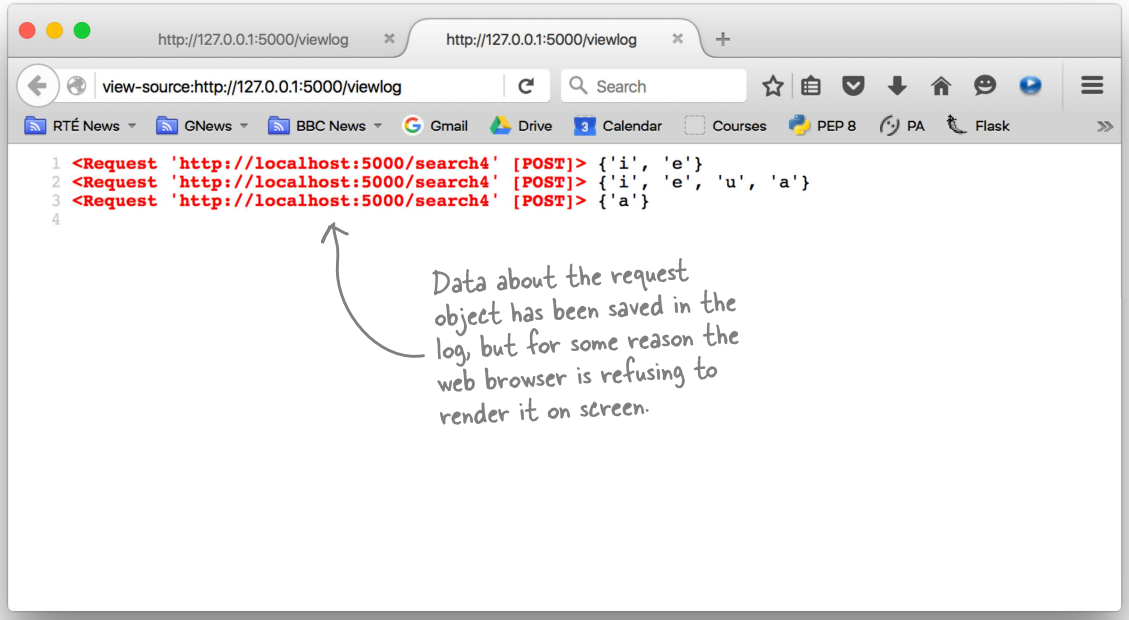
When your output doesn't quite match what you were expecting (which *is* the case above), it's best to start by checking exactly what data the webapp sent you. It's important to note that what's just appeared on screen is a *rendering* (or interpretation) of the webapp's data as performed by your web browser. All the major browsers allow you to view the raw data received with no rendering applied. This is known as the **source** of the page, and viewing it can be a useful debugging aid, as well as a great first step toward understanding what's going on here.

If you are using *Firefox* or *Chrome*, right-click on your browser window and select **View Page Source** from the pop-up menu to see the raw data as sent by your webapp. If you are running *Safari*, you'll first need to enable the developer options: open up Safari's preferences, then switch on the *Show Develop menu in the menu bar* option at the bottom of the *Advanced* tab. Once you do this, you can return to your browser window, right-click, and then select **Show Page Source** from the pop-up menu. Go ahead and view the raw data now, then compare it to what we got (on the next page).

# Examine the Raw Data with View Source

Remember, the `log_request` function saves two pieces of data for each web request it logs: the request object as well as the results of the call to `search4letters`. But when you view the log (with */viewlog*), you're only seeing the results data. Does viewing the source (i.e., the raw data returned from the webapp) offer any clue as to what happened to the request object?

Here's what we saw when we used *Firefox* to view the raw data. The fact that each request object's output is colored red is another clue that something is amiss with our log data:




The explanation as to why the request data is not rendering is subtle, and the fact that *Firefox* has highlighted the request data in red helps in understanding what's going on. It appears there's nothing wrong with the actual request data. However, it seems that the data enclosed in angle brackets (< and >) is upsetting the browser. When browsers see an opening angle bracket, they treat everything between that bracket and the matching closing angle bracket as an HTML tag. As `<Request>` is not a valid HTML tag, modern browsers simply ignore it and refuse to render any of the text between the brackets, which is what's happening here. This solves the mystery of the disappearing request data. But we still want to be able to see this data when we view the log using */viewlog*.

What we need to do is somehow tell the browser not to treat the angle brackets surrounding the request object as an HTML tag, but treat them as plain-text instead. As luck would have it, Flask comes with a function that can help.

# It's Time to Escape (Your Data)

When HTML was first created, its designers knew that some web page designers would want to display angle brackets (and the other characters that have special meaning to HTML). Consequently, they came up with the concept known as *escaping*: encoding HTML's special characters so that they could appear on a webpage but not be interpreted as HTML. A series of translations were defined, one for each special character. It's a simple idea: a special character such as < is defined as `<`, while > is defined as `>`. If you send these translations *instead of* the raw data, your web browser does the right thing: it displays < and > as opposed to ignoring them, and displays all the text between them.

Flask includes a function called `escape` (which is actually inherited from Jinja2). When provided with some raw data, `escape` translates the data into its HTML-escaped equivalent. Let's experiment with `escape` at the Python >>> prompt to get a feel for how it works.

## Geek Bits

Flask's **Markup** object is text that has been marked as being safe within an HTML/XML context. Markup inherits from Python's built-in `unicode` string, and can be used anywhere you'd use a string.

Begin by importing the `escape` function from the `flask` module, then call `escape` with a string containing none of the special characters:

```
>>> from flask import escape
>>> escape('This is a Request')
Markup('This is a Request')
```

Import the function.

Use "escape" with a normal string.

No change

The escape function returns is a *Markup object*, which—for all intents and purposes—behaves just like a string. When you pass `escape` a string containing any of HTML's special characters, the translation is done for you, as shown:

```
>>> escape('This is a <Request>')
Markup('This is a &lt;Request&gt;')
```

Use "escape" with a string containing some special characters.

The special characters have been escaped (i.e., translated).

As in the previous example (above), you can also treat this markup object as if it's a regular string.

If we can somehow arrange to call `escape` on the data in the log file, we should be able to solve the problem we currently have with the nondisplay of the request data. This should not be hard, as the log file is read "in one go" by the `view_the_log` function before being returned as a string:

```
@app.route('/viewlog')
def view_the_log() -> str:
    with open('vsearch.log') as log:
        contents = log.read()
    return contents
```

Here's our log data (as a string).

To solve our problem, all we need to do is call `escape` on `contents`.

# Viewing the Entire Log in Your Webapp

The change to your code is trivial, but makes a big difference. Add `escape` to the import list for the `flask` module (at the top of your program), then call `escape` on the string returned from calling the `join` method:

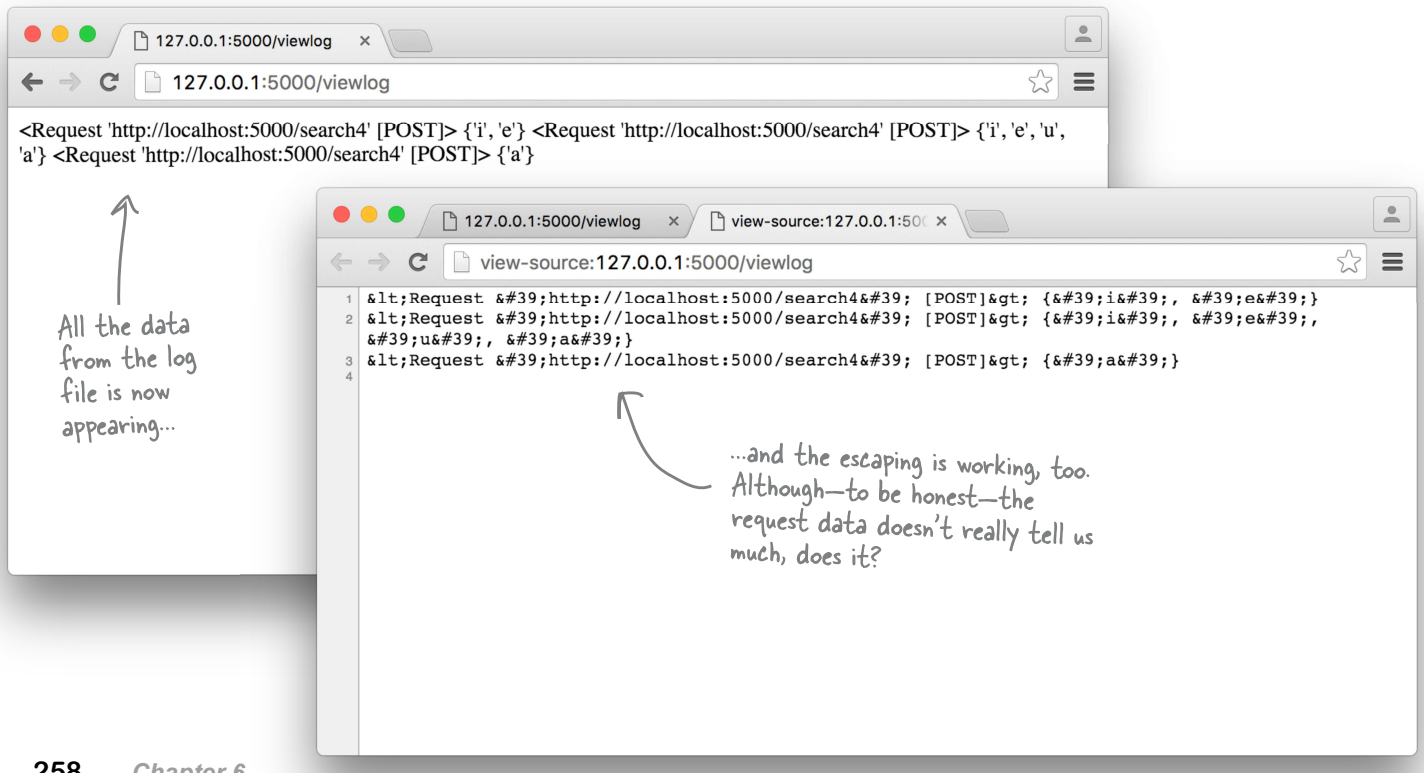
```
from flask import Flask, render_template, request, escape
      ...
@app.route('/viewlog')
def view_the_log() -> str:
    with open('vsearch.log') as log:
       contents = log.read()
    return escape(contents)
```

Add to the import list.

Call "escape" on the returned string.

## Test Drive

Amend your program to import and call `escape` as shown above, then save your code (so that your webapp reloads). Next, reload the */viewlog* URL in your browser. All of your log data should now appear on screen. Be sure to view the HTML source to confirm that the escaping is working. Here's what we saw when we tested this version of our webapp with *Chrome*:



# Learning More About the Request Object

The data in the log file relating to the web request isn't really all that useful. Here's an example of what's currently logged; although each logged result is different, each logged web request is showing up as *exactly* the same:

Each logged web request is the same.

```
<Request 'http://localhost:5000/search4' [POST]> {'i', 'e'}
<Request 'http://localhost:5000/search4' [POST]> {'i', 'e', 'u', 'a'}
<Request 'http://localhost:5000/search4' [POST]> {'a'}
```

Each logged result is different.

We're logging the web request at the object level, but really need to be looking *inside* the request and logging some of the data it contains. As you saw earlier in this book, when you need to learn what something in Python contains, you feed it to the `dir` built-in to see a list of its methods and attributes.

Let's make a small adjustment to the `log_request` function to log the output from calling `dir` on each request object. It's not a huge change...rather than passing the raw `req` as the first argument to `print`, let's pass in a stringified version of the result of calling `dir(req)`. Here's the new version of `log_request` with the change highlighted:

```
def log_request(req:'flask_request', res:str) -> None:
    with open('vsearch.log', 'a') as log:
        print(str(dir(req)), res, file=log)
```

We call "dir" on "req", which produces a list, and then we stringify the list by passing the list to "str". The resulting string is then saved to the log file along with the value of "res".

**Exercise**

Let's try out this new logging code to see what difference it makes. Perform the following steps:

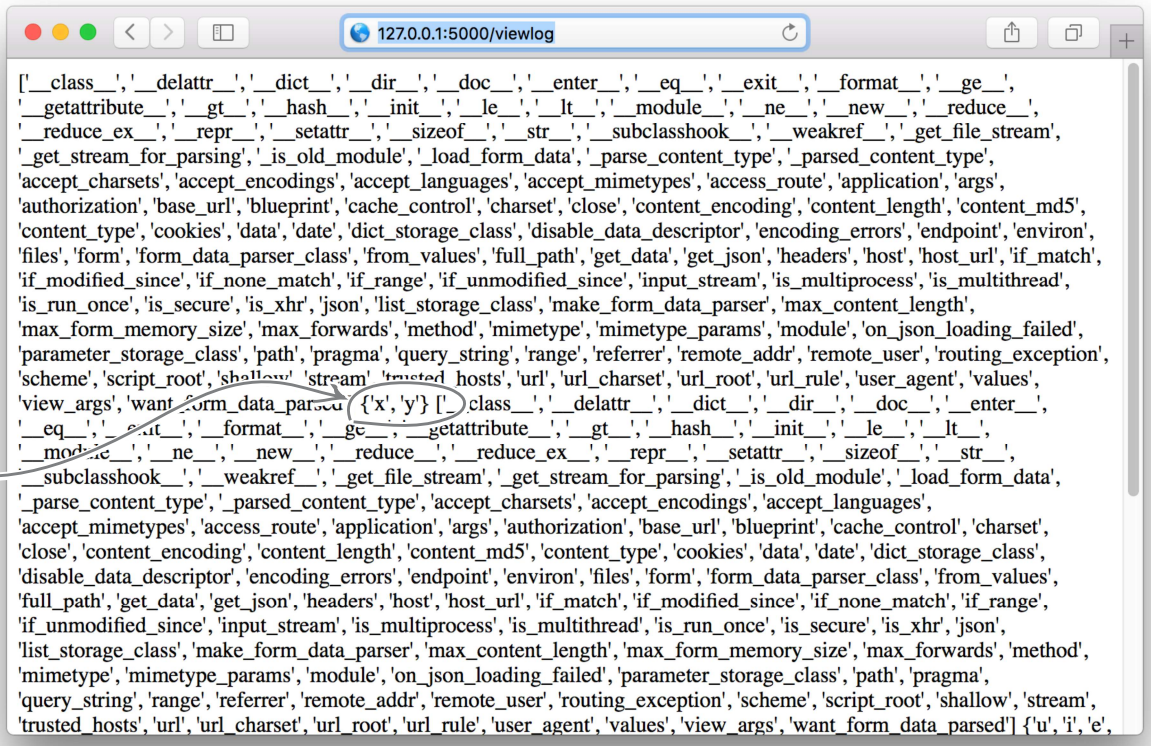
1. Amend your copy of `log_request` to match ours.
2. Save `vsearch4log.py` in order to restart your webapp.
3. Find and delete your current `vsearch.log` file.
4. Use your browser to enter three new searches.
5. View the newly created log using the */viewlog* URL.

Now: have a good look at what appears in your browser. Does what you now see help at all?

# Test Drive

Here's what we saw after we worked through the five steps from the bottom of the last page. We're using *Safari* (although every other browser shows the same thing):

This all looks kinda messy. But look closely: here's the results of one of the searches we performed.




## What's all this, then?

You can just about pick out the logged results in the above output. The rest of the output is the result of calling `dir` on the request object. As you can see, each request has a lot of methods and attributes associated with it (even when you ignore the *dunders* and *wonders*). It makes no sense to log *all* of these attributes.

We took a look at all of these attributes, and decided that there are three that we think are important enough to log:

`req.form`: The data posted from the webapp's HTML form.

`req.remote_addr`: The IP address the web browser is running on.

`req.user_agent`: The identity of the browser posting the data.

Let's adjust `log_request` to log these three specific pieces of data, in addition to the results of the call to `search4letters`.

# Logging Specific Web Request Attributes

As you now have four data items to log—the form details, the remote IP address, the browser identity, and the results of the call to `search4letters`—a first attempt at amending `log_request` might result in code that looks like this, where each data item is logged with its own `print` call:

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

Log each data item with its own "print" statement.

This code works, but it has a problem in that each `print` call appends a newline character by default, which means there are **four** lines being logged per web request. Here's what the data would look like if the log file used the above code:

The data as entered into the HTML form appears on its own line. BTW: the "ImmutableMultiDict" is a Flask-specific version of Python's dictionary (and it works in the same way).

There's a line of data for each remote IP address.

```
ImmutableMultiDict([('letters', 'aeiou'), ('phrase', 'hitch-hiker')])
127.0.0.1
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_3) ... Safari/601.4.4
{'i', 'e'}
ImmutableMultiDict([('letters', 'aeiou'), ('phrase', 'life, the universe, and everything')])
127.0.0.1
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_3) ... Safari/601.4.4
{'a', 'e', 'i', 'u'}
ImmutableMultiDict([('letters', 'xyz'), ('phrase', 'galaxy')])
127.0.0.1
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_3) ... Safari/601.4.4
{'x', 'y'}
```

The browser is identified on its own line.

The results of the call to "search4letters" are clearly shown (each on its own line).

There's nothing inherently wrong with this as a strategy (as the logged data is easy for us humans to read). However, consider what you'd have to do when reading this data into a program: each logged web request would require **four** reads from the log file—one for each line of logged data. This is in spite of the fact that the four lines of data refer to one *single* web request. As a strategy, this approach seems wasteful. It would be much better if the code only logged **one** line per web request.

# Log a Single Line of Delimited Data

A better logging strategy may be to write the four pieces of data as one line, while using an appropriately selected delimiter to separate one data item from the next.

Choosing a delimiter can be tricky, as you don't want to choose a character that might actually occur in the data you're logging. Using the space character as a delimiter is next to useless (as the logged data contains lots of spaces), and even using colon (`:`), comma (`,`), and semicolon (`;`) may be problematic given the data being logged. We checked with the programmers over at *Head First Labs*, and they suggested using a vertical bar (`|`) as a delimiter: it's easy for us humans to spot, and it's unlikely to be part of the data we log. Let's go with this suggestion and see how we get on.

**Geek Bits**

Think of a **delimiter** as a sequence of one or more characters performing the role of a boundary within a line of text. The classic example is the comma character (`,`) as used in CSV files.

As you saw earlier, we can adjust `print`'s default behavior by providing additional arguments. In addition to the `file` argument, there's the `end` argument, which allows you to specify an alternate *end-of-line* value over the default newline.

Let's amend `log_request` to use a vertical bar as the end-of-line value, as opposed to the default newline:

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

Each of these "print" statements replaces the default newline with a vertical bar.

This works as expected: each web request now results in a single line of logged data, with a vertical bar delimiting each logged data item. Here's what the data looks like in our log file when we used this amended version of `log_request`:

Each web request is written to its own line (which we've word-wrapped in order to fit on this page).

```
ImmutableMultiDict([('letters', 'aeiou'), ('phrase', 'hitch-hiker')])|127.0.0.1|Mozilla/5.0
(Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/601.3.9 (KHTML, like Gecko) Version/9.0.2
Safari/601.3.9|{'e', 'i'}
ImmutableMultiDict([('letters', 'aeiou'), ('phrase', 'life, the universe, and everything')])|12
7.0.0.1|Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/601.3.9 (KHTML, like Gecko)
Version/9.0.2 Safari/601.3.9|{'e', 'u', 'a', 'i'}
ImmutableMultiDict([('letters', 'xyz'), ('phrase', 'galaxy')])|127.0.0.1|Mozilla/5.0
(Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/601.3.9 (KHTML, like Gecko) Version/9.0.2
Safari/601.3.9|{'y', 'x'}
```

Did you spot the vertical bars used as delimiters? There are three bars, which means we have logged four pieces of data per line.

There were three web requests, so we see three lines of data in the log file.

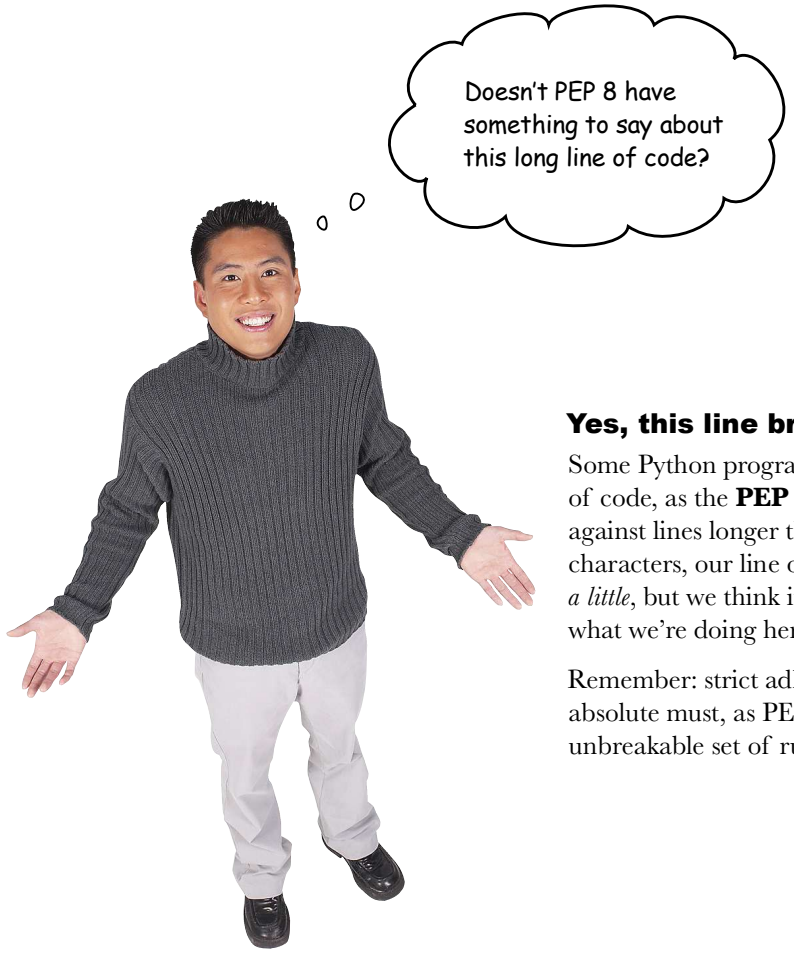
# One Final Change to Our Logging Code

Working with overly verbose code is a pet peeve of many Python programmers. Our most recent version of `log_request` works fine, but it's more verbose than it needs to be. Specifically, it feels like overkill to give each item of logged data its own `print` statement.

The `print` function has another optional argument, `sep`, which allows you to specify a separation value to be used when printing multiple values in a single call to `print`. By default, `sep` is set to a single space character, but you can use any value you wish. In the code that follows, the four calls to `print` (from the last page) have been replaced with a single `print` call, which takes advantage of the `sep` argument, setting it to the vertical bar character. In doing so, we negate the need to specify a value for `end` as the `print`'s default end-of-line value, which is why all mentions of `end` have been removed from this code:

Only one "print" call instead of four

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




### Yes, this line breaks a PEP 8 guideline.

Some Python programmers frown at this last line of code, as the **PEP 8** standard specifically warns against lines longer than 79 characters. At 80 characters, our line of code is pushing this guideline *a little*, but we think it's a reasonable trade-off given what we're doing here.

Remember: strict adherence to PEP 8 is not an absolute must, as PEP 8 is a *style guide*, not an unbreakable set of rules. We think we're good to go.


Let's see what difference this new code makes. Adjust your `log_request` function to look like this:

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

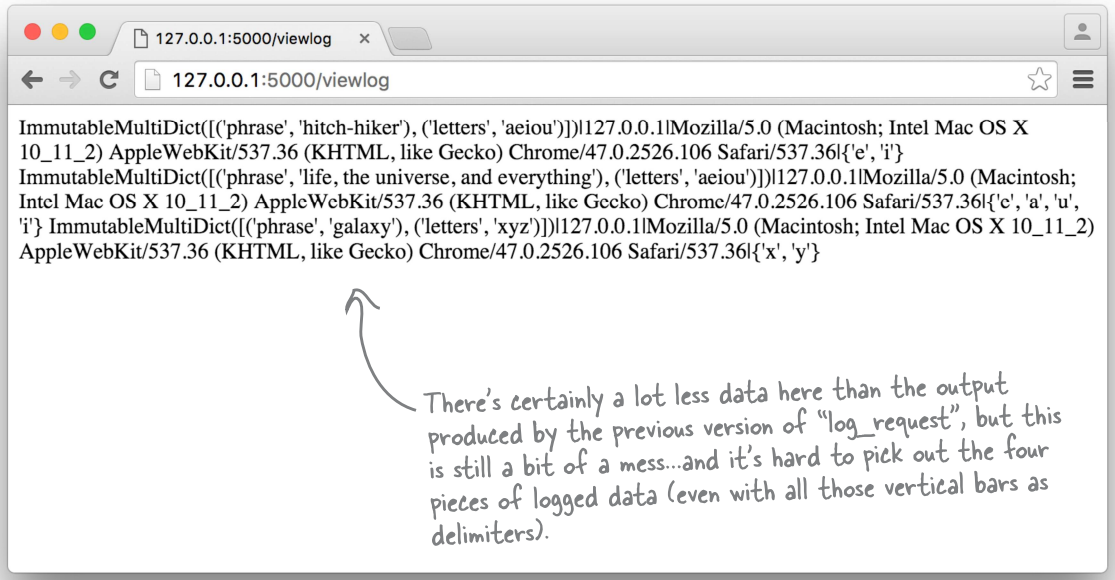
Then perform these four steps:

1. Save `vsearch4log.py` (which restarts your webapp).
2. Find and delete your current `vsearch.log` file.
3. Use your browser to enter three new searches.
4. View the newly created log using the */viewlog* URL.

Have another good look at your browser display. Is this better than before?

## Test Drive

Having completed the four steps detailed in the above exercise, we ran our latest tests using *Chrome*. Here's what we saw on screen:



# From Raw Data to Readable Output

The data displayed in the browser window is in its *raw form*. Remember, we perform HTML escaping on the data as read in from the log file but do nothing else before sending the string to the waiting web browser. Modern web browsers will receive the string, remove any unwanted whitespace characters (such as extra spaces, newlines, and so on), then dump the data to the window. This is what's happening during our *Test Drive*. The logged data—all of it—is visible, but it's anything but easy to read. We could consider performing further text manipulations on the raw data (in order to make the output easier to read), but a better approach to producing readable output might be to manipulate the raw data in such a way as to turn it into a table:

```
ImmutableMultiDict([('phrase', 'hitch-hiker'), ('letters', 'aeiou')])|127.0.0.1|Mozilla/5.0
(Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106
Safari/537.36|{'e', 'i'} ImmutableMultiDict([('phrase', 'life, the universe, and
everything'), ('letters', 'aeiou')])|127.0.0.1|Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106 Safari/537.36|{'e', 'a', 'u',
'i'} ImmutableMultiDict([('phrase', 'galaxy'), ('letters', 'xyz')])|127.0.0.1|Mozilla/5.0
(Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106
Safari/537.36|{'x', 'y'}
```

Can we take this (unreadable) raw data...

...and transform it into a table that looks like this?

| Form Data | Remote_addr | User_agent | Results |
|---|---|---|---|
| ImmutableMultiDict([('phrase', 'hitch-hiker'), ('letters', 'aeiou')]) | 127.0.0.1 | Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106 Safari/537.36 | {'e', 'i'} |
| ImmutableMultiDict([('phrase', 'life, the universe, and everything'), ('letters', 'aeiou')]) | 127.0.0.1 | Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106 Safari/537.36 | {'e', 'a', 'u', 'i'} |
| ImmutableMultiDict([('phrase', 'galaxy'), ('letters', 'xyz')]) | 127.0.0.1 | Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106 Safari/537.36 | {'x', 'y'} |

If our webapp could perform this transformation, then *anyone* could view the log data in their web browser and likely make sense of it.

# Does This Remind You of Anything?

Take another look at what you are trying to produce. To save on space, we're only showing the top portion of the table shown on the previous page. Does what you're trying to produce here remind you of anything from earlier in this book?

| Form Data | Remote_addr | User_agent | Results |
|---|---|---|---|
| ImmutableMultiDict([('phrase', 'hitch-hiker'), ('letters', 'aeiou')]) | 127.0.0.1 | Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.106 Safari/537.36 | {'e', 'i'} |


**Yes. That does look like something we've seen before.**

At the end of Chapter 3, recall that we took the table of data below and transformed it into a complex data structure—a dictionary of dictionaries:

| Name | Gender | Occupation | Home Planet |
|---|---|---|---|
| Ford Prefect | Male | Researcher | Betelgeuse Seven |
| Arthur Dent | Male | Sandwich-Maker | Earth |
| Tricia McMillan | Female | Mathematician | Earth |
| Marvin | Unknown | Paranoid Android | Unknown |

The shape of this table is similar to what we're hoping to produce above, but is a dictionary of dictionaries the right data structure to use here?

# Use a Dict of Dicts...or Something Else?

The table of data from Chapter 3 fit the dictionary of dictionaries model because it allowed you to quickly dip into the data structure and extract specific data. For instance, if you wanted to know Ford Prefect's home planet, all you had to do was this:

```
people['Ford']['Home Planet']
```

Access Ford's data...

...then extract the value associated with the "Home Planet" key.

When it comes to randomly accessing a data structure, nothing beats a dictionary of dictionaries. However, is this what we want for our logged data?

Let's consider what we currently have.

## Take a closer look at the logged data

Remember, every logged line contains four pieces of data, each separated by vertical bars: the HTML form's data, the remote IP address, the identity of the web browser, and the results of the call to `search4letters`.

Here's a sample line of data from our `vsearch.log` file with each of the vertical bars highlighted:

The form data

The IP address of the remote machine

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

The web browser's identity string

The results of the call to "search4letters"

When the logged data is read from the `vsearch.log` file, it arrives in your code as a *list of strings* thanks to our use of the `readlines` method. Because you probably won't need to randomly access individual data items from the logged data, converting the data to a dictionary of dictionaries seems like a bad move. However, you need to process each line *in order*, as well as process each individual data item within each line *in order*. You already have a list of strings, so you're half-way there, as it's easy to process a list with a `for` loop. However, the line of data is currently one string, and this is the issue. It would be easier to process each line if it were a list of data items, as opposed to one large string. The question is: *is it possible to convert a string to a list?*

# What's Joined Together Can Be Split Apart

You already know that you can take a list of strings and convert them to a single string using the "join trick." Let's show this once more at the >>> prompt:

A list of individual strings

```
>>> names = ['Terry', 'John', 'Michael', 'Graham', 'Eric']
>>> pythons = '|'.join(names)
>>> pythons
'Terry|John|Michael|Graham|Eric'
```

The "join trick" in action.

A single string with each string from the "names" list concatenated with the next and delimited by a vertical bar

Thanks to the "join trick," what was a list of strings is now a single string, with each list item separated from the next by a vertical bar (in this case). You can reverse this process using the `split` method, which comes built in to every Python string:

Take the string and split it into a list using the given delimiter.

```
>>> individuals = pythons.split('|')
>>> individuals
['Terry', 'John', 'Michael', 'Graham', 'Eric']
```

And now we are back to our list of strings.

## Getting to a list of lists from a list of strings

Now that you have the `split` method in your coding arsenal, let's return to the data stored in the log file and consider what needs to happen to it. At the moment, each individual line in the `vsearch.log` file is a string:

The raw data

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

Your code currently reads all the lines from `vsearch.log` into a list of strings called `contents`. Shown here are the last three lines of code from the `view_the_log` function, which read the data from the file and produce the large string:

Open the log file...

```
    ...
    with open('vsearch.log') as log:
        contents = log.readlines()
    return escape(''.join(contents))
```

...and read all the lines of log data into a list called "contents".

The last line of the `view_the_log` function takes the list of strings in `contents` and concatenates them into one large string (thanks to `join`). This single string is then returned to the waiting web browser.

If `contents` were a list of *lists* instead of a list of *strings*, it would open up the possibility of processing `contents` *in order* using a `for` loop. It should then be possible to produce more readable output than what we're currently seeing on screen.

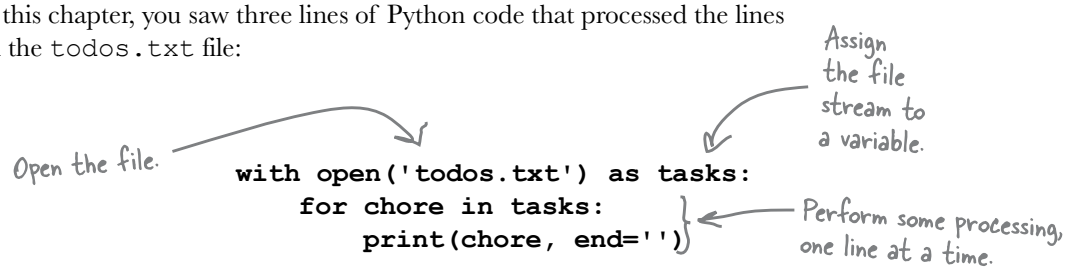
# When Should the Conversion Occur?

At the moment, the `view_the_log` function reads all the data from the log file into a list of strings (called `contents`). But we'd rather have the data as a list of lists. The thing is, when's the "best time" to do this conversion? Should we read in all the data into a list of strings, then convert it to a list of lists "as we go," or should we build the list of lists while reading in each line of data?

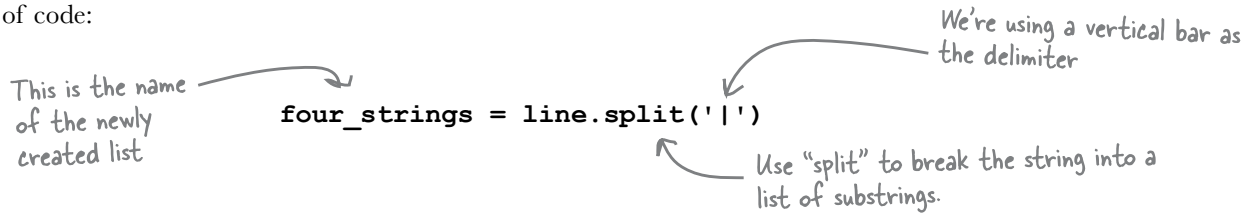

The fact that the data is already in `contents` (thanks to our use of the `readlines` method) shouldn't blind us to the fact that we've already looped through the data *once* at this point. Invoking `readlines` may only be a single call for us, but the interpreter (while executing `readlines`) *is* looping through the data in the file. If we then loop through the data again (to convert the strings to lists), we're **doubling** the amount of looping that's occurring. This isn't a big deal when there's only a handful of log entries...but it might be an issue when the log grows in size. The bottom line is this: *if we can make do by only looping once, then let's do so!*

# Processing Data: What We Already Know

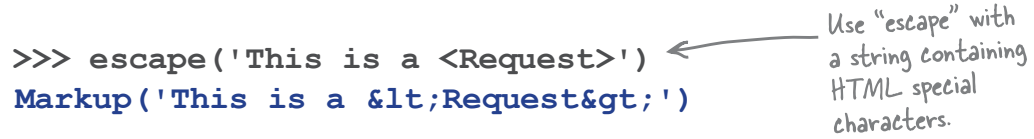
Earlier in this chapter, you saw three lines of Python code that processed the lines of data in the `todos.txt` file:



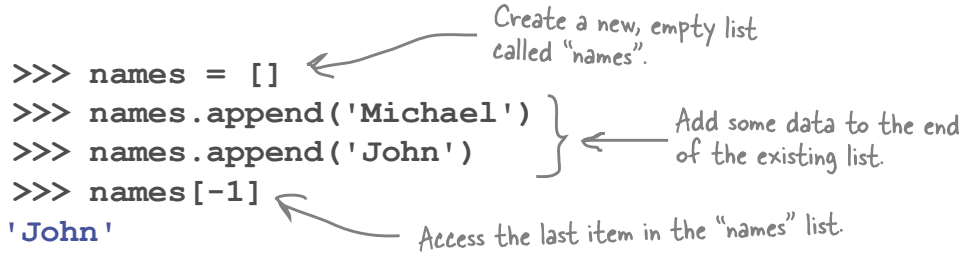

You've also seen the `split` method, which takes a string and converts it to a list of strings based on some delimiter (defaulting to a space, if none is provided). In our data, the delimiter is a vertical bar. Let's assume that a line of logged data is stored in a variable called `line`. You can turn the single string in `line` into a list of four individual strings—using the vertical bar as the delimiter—with this line of code:




As you can never be sure whether the data you're reading from the log file is free of any characters that have special meaning to HTML, you've also learned about the `escape` function. This function is provided by Flask, and converts any string's HTML special characters into their equivalent escaped values:




And, starting way back in Chapter 2, you learned that you can create a new list by assigning an empty list to it (`[]`). You also know that you can assign values to the end of an existing list by calling the `append` method, and that you can access the last item in any list using the `[-1]` notation:




Armed with this knowledge, see if you can complete the exercise on the next page.

## Sharpen your pencil

Here is the `view_the_log` function's current code:

```
@app.route('/viewlog')
def view_the_log() -> str:
    with open('vsearch.log') as log:
        contents = log.readlines()
    return escape(''.join(contents))
```

This code reads the data from the log file into a list of strings. Your job is to convert this code to read the data into a list of lists.

Make sure that the data written to the list of lists is properly escaped, as you do not want any HTML special characters sneaking through.

Also, ensure that your new code still returns a string to the waiting web browser.

We've got you started—fill in the missing code:

The first two lines remain unchanged.

```
@app.route('/viewlog')
def view_the_log() -> 'str':
```

.....

.....

.....

.....

.....

Add your new code here.

.....

.....

.....

.....

The function still returns a string.

```
return str(contents)
```

Take your time here. Feel free to experiment at the >>> shell as needed, and don't worry if you get stuck—it's OK to flip the page and look at the solution.

## Sharpen your pencil Solution

Here is the `view_the_log` function's code:

```
@app.route('/viewlog')
def view_the_log() -> str:
    with open('vsearch.log') as log:
        contents = log.readlines()
    return escape(''.join(contents))
```

Your job was to convert this code to read the data into a list of lists.

You were to ensure that the data written to the list of lists is properly escaped, as you do not want any HTML special characters sneaking through.

You were also to ensure that your new code still returns a string to the waiting web browser.

We'd started for you, and you were to fill in the missing code:

```
@app.route('/viewlog')
def view_the_log() -> 'str':
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    return str(contents)
```

Create a new, empty list called "contents".

Open the log file and assign it to a file stream called "log".

Loop through each line in the "log" file stream.

Append a new, empty list to "contents".

Split the line (based on the vertical bar), then process each item in the resulting "split list".

Did you remember to call "escape"?

Append the escaped data to the end of the list at the end of "contents".

Don't worry if this line of code from the above rewrite of the `view_the_log` function has your head spinning:

```
contents[-1].append(escape(item))
```

Read this code from the inside out, and from right to left.

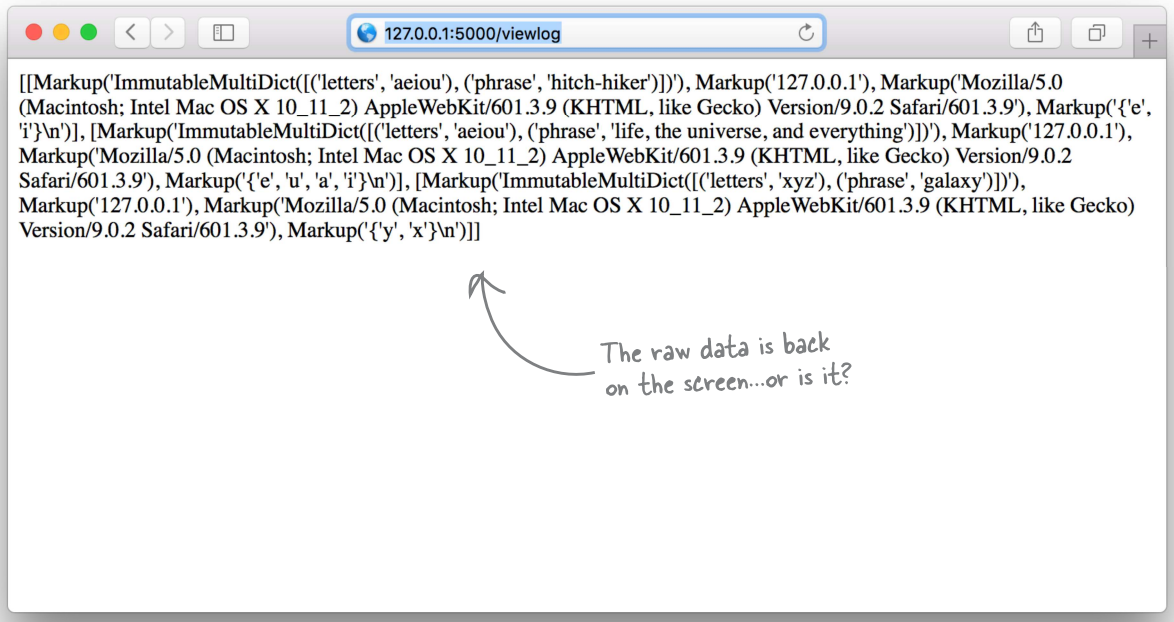
The trick to understanding this (initially daunting) line is to read it from the inside out, and from right to left. You start with the `item` from the enclosing `for` loop, which gets passed to `escape`. The resulting string is then `append`ed to the list at the end (`[-1]`) of `contents`. Remember: `contents` is itself a *list of lists*.

# Test Drive

Go ahead and change your `view_the_log` function to look like this:

```
@app.route('/viewlog')
def view_the_log() -> 'str':
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    return str(contents)
```

Save your code (which causes your webapp to reload), then reload the */viewlog* URL in your browser. Here's what we saw in ours:




## Take a closer look at the output

At first glance, the output produced by this new version of `view_the_log` looks very similar to what you had before. But it isn't: this new output is a list of lists, not a list of strings. This a crucial change. If you can now arrange to process `contents` using an appropriately designed Jinja2 template, you should be able to get pretty close to the readable output required here.

# Generate Readable Output With HTML

Recall that our goal is to produce output that looks better on screen than the raw data from the last page. To that end, HTML comes with a set of tags for defining the content of tables, including: `<table>`, `<th>`, `<tr>`, and `<td>`. With this in mind, let's take another look at the top portion of the table we're hoping to produce once more. It has one row of data for each line in the log, arranged as four columns (each with a descriptive title).

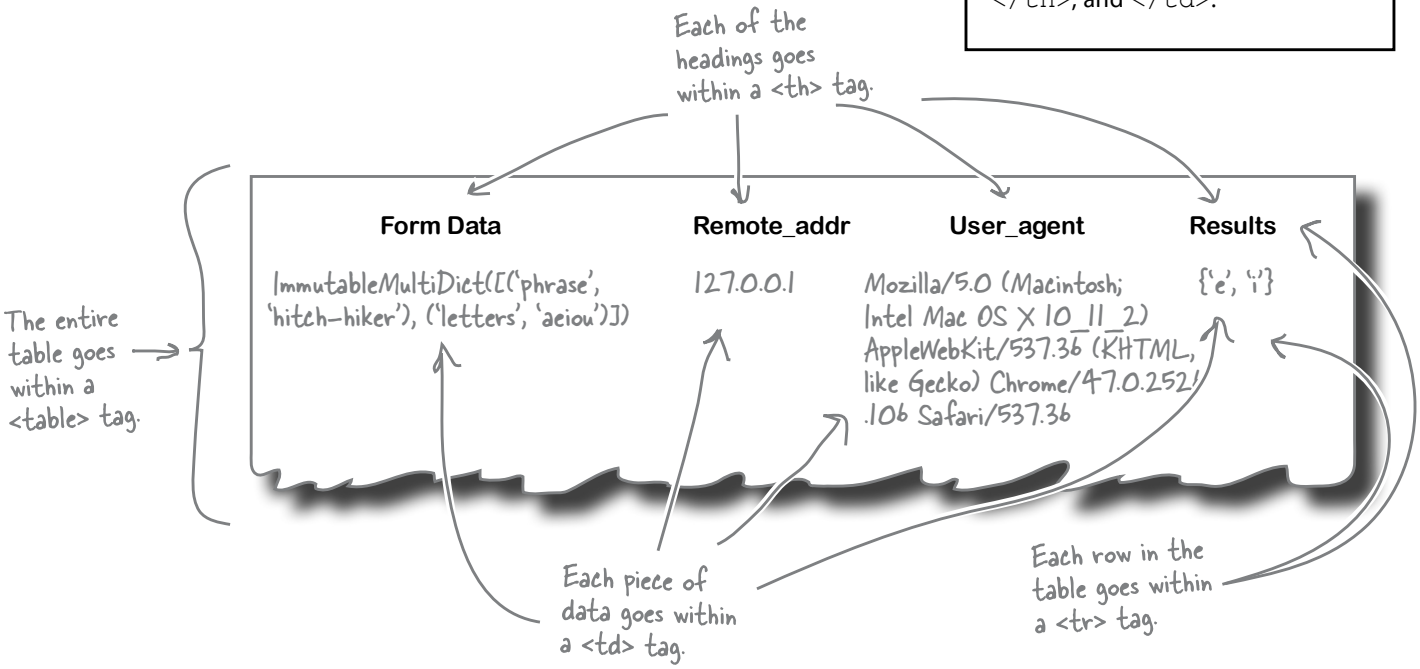
You could put the entire table within an HTML `<table>` tag, with each row of data having its own `<tr>` tag. The descriptive titles each get `<th>` tags, while each piece of raw data gets its own `<td>` tag:

## Geek Bits

Here's a quick review of the HTML table tags:

`<table>`: A table.
`<tr>`: A row of table data.
`<th>`: A table column heading.
`<td>`: A table data item (cell).

Each tag has a corresponding end tag: `</table>`, `</tr>`, `</th>`, and `</td>`.




Whenever you find yourself needing to generate any HTML (especially a `<table>`), remember Jinja2. The Jinja2 template engine is primarily designed to generate HTML, and the engine contains some basic programming constructs (loosely based on Python syntax) that you can use to "automate" any required display logic you might need.

In the last chapter, you saw how the Jinja2 `{{` and `}}` tags, as well as the `{% block %}` tag, allow you to use variables and blocks of HTML as arguments to templates. It turns out the `{%` and `%}` tags are much more general, and can contain any Jinja2 *statement*, with one of the supported statements being a `for` loop construct. On the next page you'll find a new template that takes advantage of Jinja2's `for` loop to build the readable output from the list of lists contained in `contents`.

# Embed Display Logic in Your Template

Below is a new template, called `viewlog.html`, which can be used to transform the raw data from the log file into an HTML table. The template expects the `contents` list of lists to be one of its arguments. We've highlighted the bits of this template we want you to concentrate on. Note that Jinja2's `for` loop construct is very similar to Python's. There are two major differences:

- There's no need for a colon (`:`) at the end of the `for` line (as the `%}` tag acts as a delimiter).
- The loop's suite is terminated with `{% endfor %}`, as Jinja2 doesn't support indentation (so some other mechanism is required).

**You don't have to create this template yourself. Download it from *http://python.itcarlow.ie/ed2/*.**

As you can see, the first `for` loop expects to find its data in a variable called `the_row_titles`, while the second `for` loop expects its data in something called `the_data`. A third `for` loop (embedded in the second) expects its data to be a list of items:

## Ready Bake Code

To ensure a consistent look and feel, this template inherits from the same base template used throughout our webapp.

```
{% extends 'base.html' %}

{% block body %}

<h2>{{ the_title }}</h2>

<table>
    <tr>
        {% for row_title in the_row_titles %}
            <th>{{row_title}}</th>
        {% endfor %}
    </tr>
    {% for log_row in the_data %}
        <tr>
            {% for item in log_row %}
                <td>{{item}}</td>
            {% endfor %}
        </tr>
    {% endfor %}
</table>

{% endblock %}
```

The descriptive titles (each within a <th> tag) get their own row (the <tr> tag).

The entire table goes within a <table> tag.

Each individual item of logged data is enclosed within a <td> tag, and each line from the log file has its own <tr> tag.

Be sure to place this new template in your webapp's `templates` folder prior to use.

# Producing Readable Output with Jinja2

As the `viewlog.html` template inherits from `base.html`, you need to remember to provide a value for the `the_title` argument and provide a list of column headings (the descriptive titles) in `the_row_titles`. And don't forget to assign `contents` to the `the_data` argument.

The `view_the_log` function currently looks like this:

We currently return a string to the waiting web browser.

```
@app.route('/viewlog')
def view_the_log() -> 'str':
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    return str(contents)
```

You need to call `render_template` on `viewlog.html`, and pass it values for each of the three arguments it expects. Let's create a tuple of descriptive titles and assign it to `the_row_titles`, then assign the value of `contents` to `the_data`. We'll also provide an appropriate value for `the_title` before rendering the template.

**Remember: a tuple is a read-only list.**

With all of that in mind, let's amend `view_the_log` (we've highlighted the changes):

Change the annotation to indicate that HTML is being returned (instead of a string).

Create a tuple of descriptive titles.

Call "render_template", providing values for each of the template's arguments.

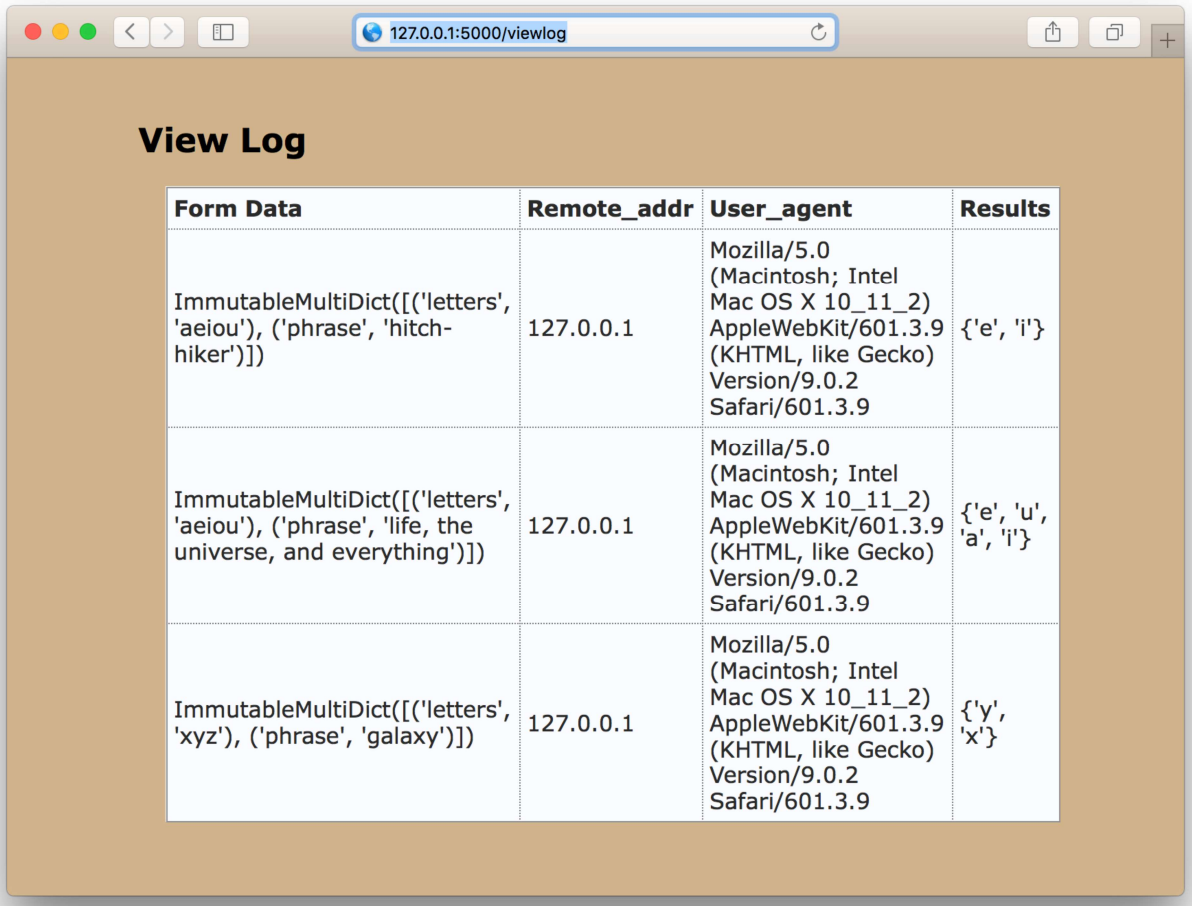
```
@app.route('/viewlog')
def view_the_log() -> 'html':
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    titles = ('Form Data', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)
```

Go ahead and make these changes to your `view_the_log` function and then save them so that Flask restarts your webapp. When you're ready, view the log within your browser using the *http://127.0.0.1:5000/viewlog* URL.

# Test Drive

Here's what we saw when we viewed the log using our updated webapp. The page has the same look and feel as all our other pages, so we are confident that our webapp is using the correct template.

We're pretty pleased with the result (and we hope you are too), as this looks very similar to what we were hoping to achieve: readable output.




Not only is this output readable, but it looks good, too. ☺

If you view the source of the above page—right-click on the page, then choose the appropriate option from the pop-up menu—you'll see that every single data item from the log is being given its own `<td>` tag, each line of data has its own `<tr>` tag, and the entire table is within a HTML `<table>`.

# The Current State of Our Webapp Code

Let's pause for a moment and review our webapp's code. The addition of the logging code (`log_request` and `view_the_log`) has added to our webapp's codebase, but everything still fits on a single page. Here's the code for `vsearch4web.py` displayed in an IDLE edit window (which lets you review the code in all its syntax-highlighted glory):

vsearch4web.py - /Users/paul/Desktop/_NewBook/ch06/webapp/vsearch4web.py (3.5.1)

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

app = Flask(__name__)

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

@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')

@app.route('/viewlog')
def view_the_log() -> 'html':
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    titles = ('Form Data', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)

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

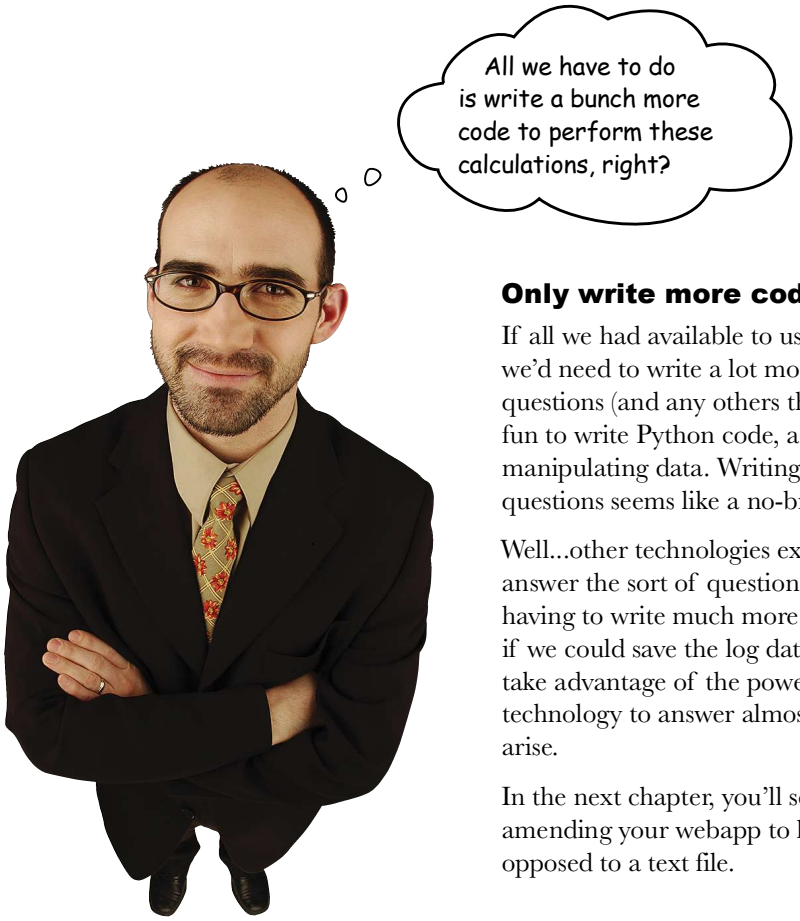
Ln: 2 Col: 0

# Asking Questions of Your Data

Our webapp's functionality is shaping up nicely, but are we any closer to answering the questions posed at the start of this chapter: *How many requests have been responded to? What's the most common list of letters? Which IP addresses are the requests coming from? Which browser is being used the most?*

The last two questions can be somewhat answered by the output displayed by the */viewlog* URL. You can tell where the requests are coming from (the **Remote_addr** column), as well as see which web browser is being used (the **User_agent** column). But, if you want to calculate which of the major browsers is used most by users of your site, that's not so easy. Simply looking at the displayed log data isn't enough; you'll have to perform additional calculations.

The first two questions cannot be easily answered either. It should be clear that further calculations must be performed here, too.




### Only write more code when you have to.

If all we had available to us was Python, then, yes, we'd need to write a lot more code to answer these questions (and any others that might arise). After all, it's fun to write Python code, and Python is also great at manipulating data. Writing more code to answer our questions seems like a no-brainer, doesn't it?

Well...other technologies exist that make it easy to answer the sort of questions we're posing without us having to write much more Python code. Specifically, if we could save the log data to a database, we could take advantage of the power of the database's querying technology to answer almost any question that might arise.

In the next chapter, you'll see what's involved in amending your webapp to log its data to a database as opposed to a text file.

# Chapter 6's Code

Remember: they both do the same thing, but Python programmers prefer this code over this.

```
with open('todos.txt') as tasks:
    for chore in tasks:
        print(chore, end='')
```

```
tasks = open('todos.txt')
for chore in tasks:
    print(chore, end='')
tasks.close()
```

Here's the code we added to the webapp to support logging our web requests to a text file.

```
       ...

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

       ...


@app.route('/viewlog')
def view_the_log() -> 'html':
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    titles = ('Form Data', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)

       ...
```

We aren't showing all the "vsearch4web.py" code here, just the new stuff. (You'll find the entire program two pages back.)

# 7 using a database

# Putting Python's DB-API to Use


## Storing data in a relational database system is handy.

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

# Database-Enabling Your Webapp

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

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

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

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

With these four tasks complete, we'll be in a position to amend the `vsearch4web.py` code to log to MySQL as opposed to a text file. We'll then use SQL to ask and—with luck—answer our questions.

## there are no Dumb Questions

**Q: Do we have to use MySQL here?**

A: If you want to follow along with the examples in the remainder of this chapter, the answer is yes.

**Q: Can I use MariaDB instead of MySQL?**

A: Yes. As MariaDB is a clone of MySQL, we have no issue with you using MariaDB as your database system instead of the "official" MySQL. (In fact, over at *Head First Labs*, MariaDB is a favorite among the DevOps team.)

**Q: What about PostgreSQL? Can I use that?**

A: Emm, eh...yes, subject to the following caveat: if you are already using PostgreSQL (or any other SQL-based database management system), you can try using it in place of MySQL. However, note that this chapter doesn't provide any specific instructions related to PostgreSQL (or anything else), so you may have to experiment on your own when something we show you working with MySQL doesn't work in quite the same way with your chosen database. There's also the standalone, single-user **SQLite,** which comes with Python and lets you work with SQL *without* the need for a separate server. That said, which database technology you use very much depends on what you're trying to do.

# Task 1: Install the MySQL Server

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

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

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

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

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

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

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

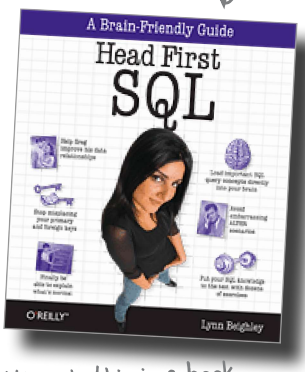
***http://dev.mysql.com/downloads/mysql/***

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

***https://mariadb.org/download/***

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

Note from Marketing: Of all the MySQL books...in all the world...this is the one we brought to the ~~bar~~ ...eh...office when we first learned MySQL.




Although this is a book about the SQL query language, it uses the MySQL database management system for all its examples. Despite its age, it's a still great learning resource.

This is going to be painful, as I've never used MySQL before...

**Don't worry if this is new to you.**

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

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

# Introducing Python's DB-API

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

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

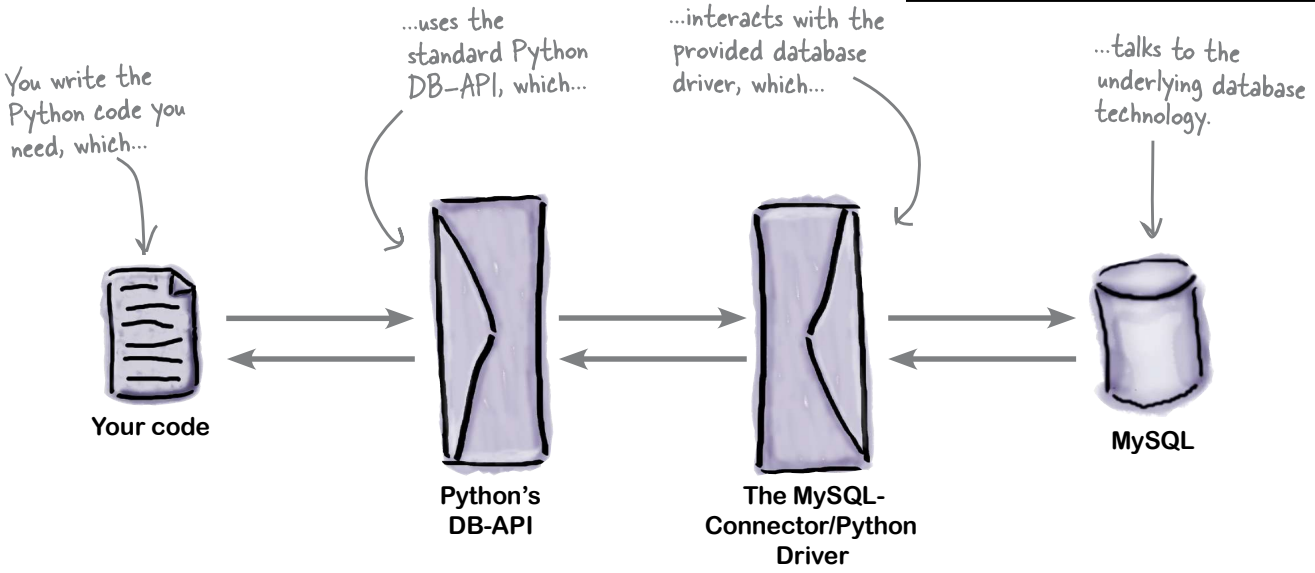
Out of the box, the Python interpreter comes with some support for working with databases, but nothing specific to MySQL. What's provided is a standard database API (application programmer interface) for working with SQL-based databases, known as *DB-API*. What's missing is the **driver** to connect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

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

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

## Geek Bits

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




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

# Task 2: Install a MySQL Database Driver for Python

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

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

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

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

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


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

# Install MySQL-Connector/Python

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

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

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

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

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

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

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

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

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

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

## there are no Dumb Questions

**Q: Should I worry about that "Not Installing C Extension" message?**

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

# Task 3: Create Our Webapp's Database and Tables

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
mysql> quit
```

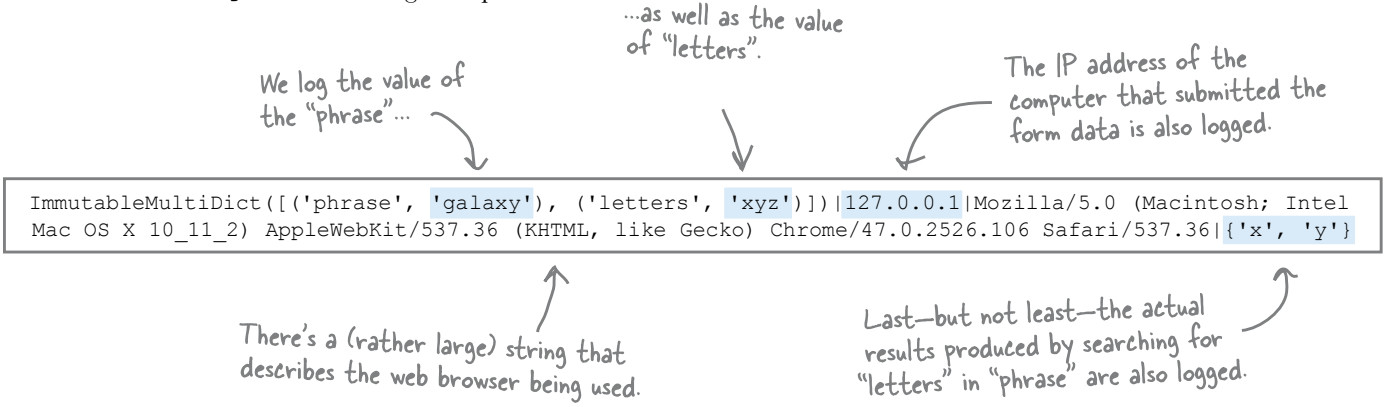
You'll see a friendly `Bye` message from the console before being returned to your operating system.

# Decide on a Structure for Your Log Data

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

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

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




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

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

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

Remember: we set this user's password to "vsearchpasswd".

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

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

This is the console's continuation symbol.

MySQL will automatically provide data for these fields.

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

# Confirm Your Table Is Ready for Data

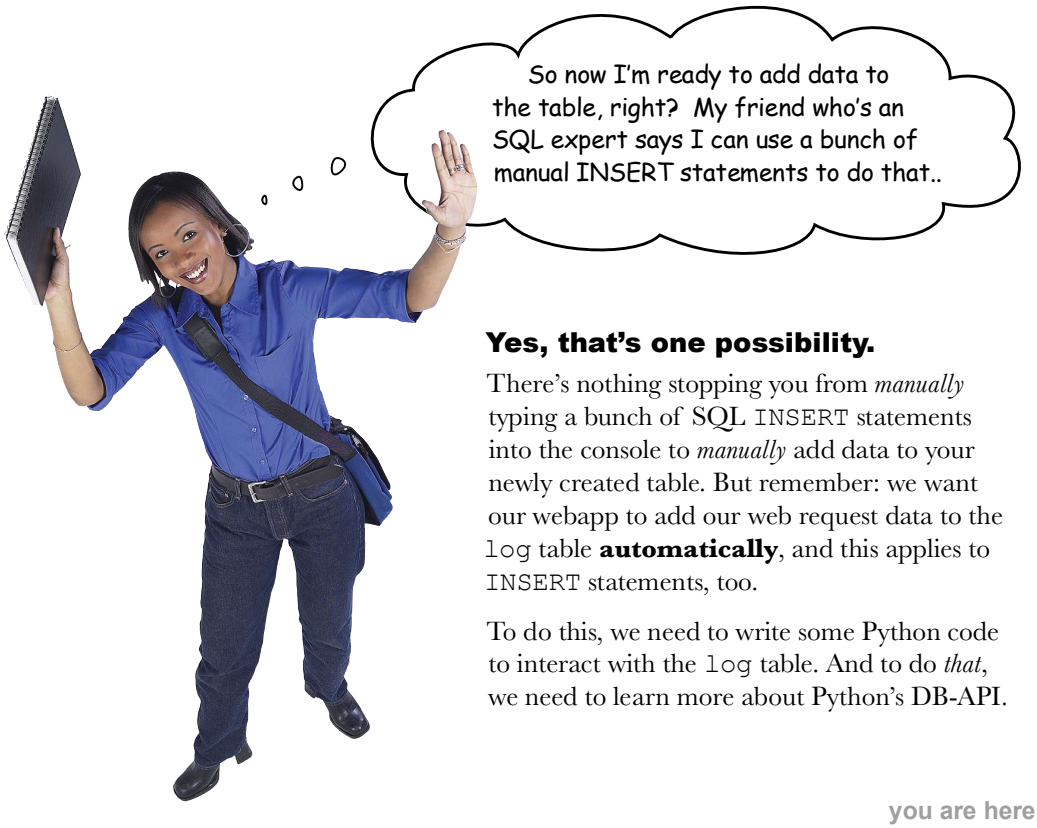
- [x] Install MySQL on your computer.
- [x] Install a MySQL Python driver.
- [x] Create the database and tables.
- [ ] Create code to read/write data.

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

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

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

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



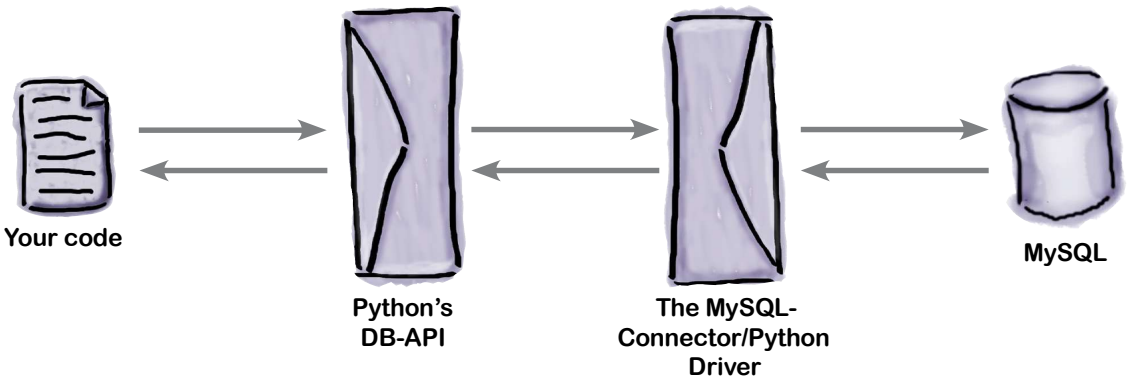

**Yes, that's one possibility.**

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

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

## DB-API Up Close, 1 of 3

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




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

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

## DB-API Step 1: Define your connection characteristics

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

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

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

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

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

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

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

## DB-API Step 2: Import your database driver

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

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

Import the driver for the database you are using.

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

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

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

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

This call establishes the connection.

Pass in the dictionary of connection characteristics.

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

## DB-API Step 4: Open a cursor

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

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

```
>>> cursor = conn.cursor()
```

Create a cursor to send commands to the server, and to receive results.

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

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

## DB-API Up Close, 2 of 3

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

### DB-API Step 5: Do the SQL thing!

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

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

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

Assign the SQL query to a variable.

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

Send the query in the "_SQL" variable to MySQL for execution.

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

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

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

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

Get all the data returned from MySQL.

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

Display the results.

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

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

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

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

Take the SQL query...

...then send it to the server...

...and then access the results.

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

```
>>> for row in res:
        print(row)
('id', 'int(11)', 'NO', 'PRI', None, 'auto_increment')
('ts', 'timestamp', 'NO', '', 'CURRENT_TIMESTAMP', '')
('phrase', 'varchar(128)', 'NO', '', None, '')
('letters', 'varchar(32)', 'NO', '', None, '')
('ip', 'varchar(16)', 'NO', '', None, '')
('browser_string', 'varchar(256)', 'NO', '', None, '')
('results', 'varchar(64)', 'NO', '', None, '')
```

Take each row in the results...

...and display it on its own line.

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

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

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

Look closely. It's the same data.

## DB-API Up Close, 3 of 3

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

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

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

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

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

Compare these commands below with those shown above:

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

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

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

How can this be?

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

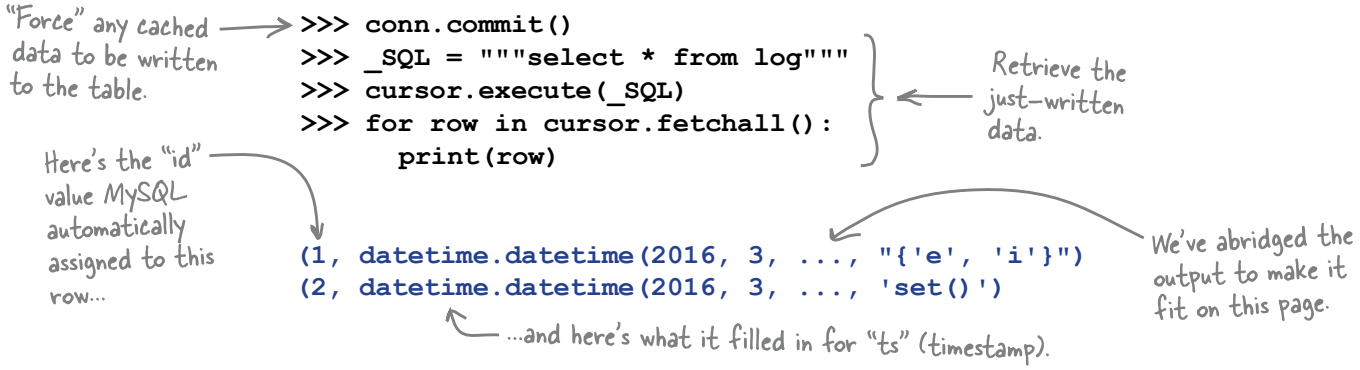
Take another, closer look at that line of code. See the pair of parentheses around the data values? The use of parentheses turns the five data values into a single tuple (containing the individual data values). In effect, the above line of code supplies two arguments to `cursor.execute`: the placeholder-containing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

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

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

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

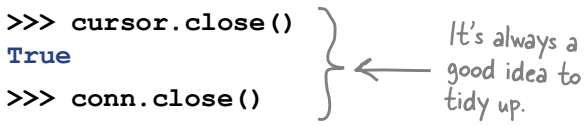
If you are happy to take the performance hit associated with a database write, you can force your database system to commit all potentially cached data to your table using the `conn.commit` method. Let's do that now to ensure the two `insert` statements from the previous page are applied to the `log`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




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

## DB-API Step 6: Close your cursor and connection

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




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

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

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

Our task list is done!

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

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

Define your connection characteristics.

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

import mysql.connector
```

Import the database driver.

Establish a connection and create a cursor.

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

cursor = conn.cursor()
```

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

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

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

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

Force the database to write your data.

```
conn.commit()

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

cursor.execute(_SQL)

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

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

```
cursor.close()

conn.close()
```

Tidy up when you're done.

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

# Database Magnets

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

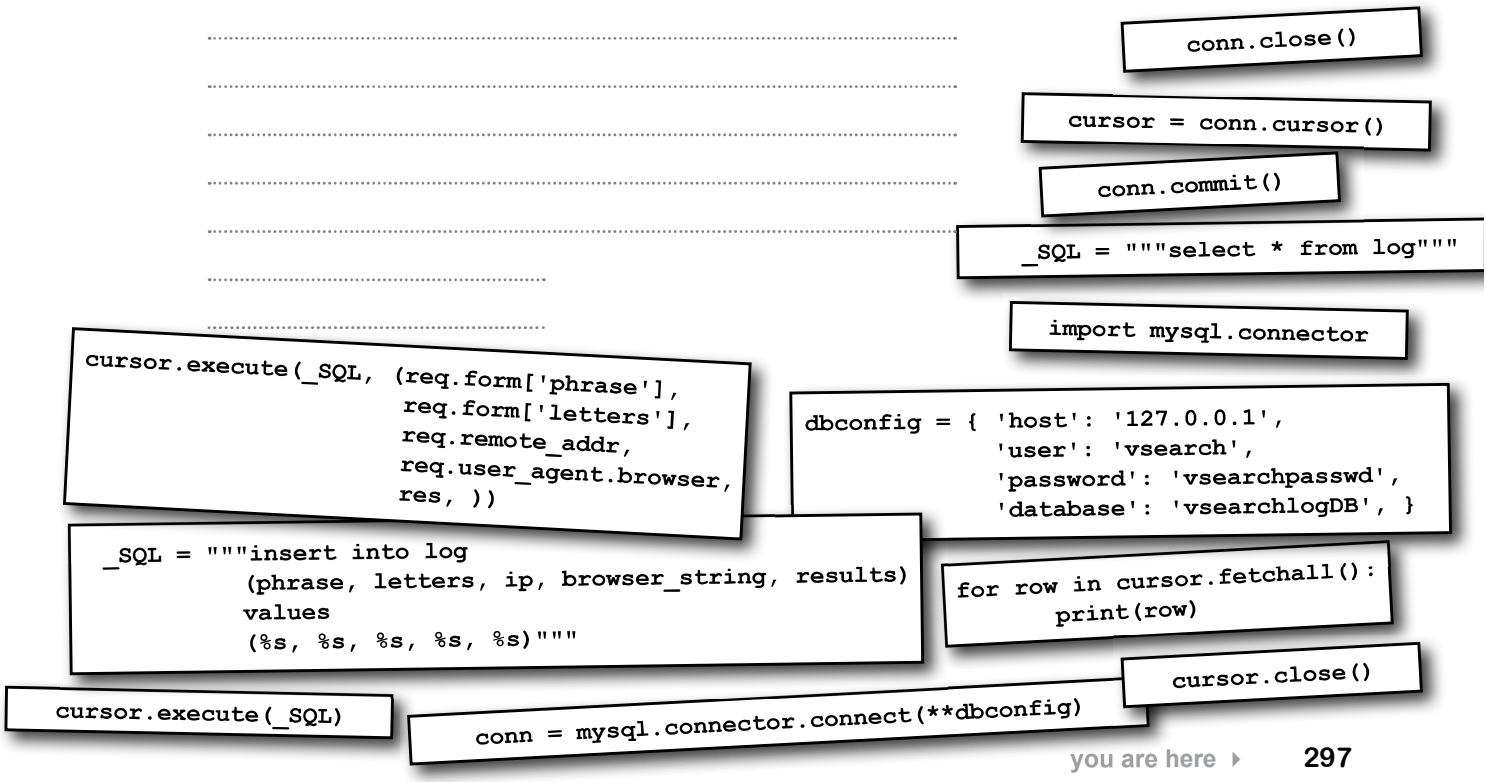
Recall that this small function accepts two arguments: a web request object, and the results of the vsearch:

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

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

```
def log_request(req: 'flask_request', res: str) -> None:
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
```

Crikey! What a mess of magnets. Can you help?



# Database Magnets Solution

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

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

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

```
def log_request(req: 'flask_request', res: str) -> None:
    dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }
```

Define the connection characteristics.

```
    import mysql.connector
    conn = mysql.connector.connect(**dbconfig)
    cursor = conn.cursor()
```

Import the driver, then establish a connection, and then create a cursor.

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

Create a string containing the query you want to use.

```
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))
```

Execute the query.

This is new: rather than store the entire browser string (stored in "req.user_agent"), we're only extracting the name of the browser.

```
    conn.commit()
    cursor.close()
    conn.close()
```

After ensuring the data is saved, we're tidying up by closing the cursor and the connection.

These magnets weren't needed.

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

# Test Drive

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

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

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

```
$ python3 vsearch4web.py
```

Your webapp should start running at this web address:

***http://127.0.0.1:5000/***

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

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

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

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

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

Log in to the MySQL console.

This query asks to see all the data in the "log" table (your actual data will likely differ).

```
File  Edit  Window  Help  Checking our log DB
$ mysql -u vsearch -p vsearchlogDB
Enter password:

Welcome to MySQL monitor...


mysql> select * from log;
+----+---------------------+--------------------------+---------+-----------+----------------+----------------------+
| id | ts                  | phrase                   | letters | ip        | browser_string | results              |
+----+---------------------+--------------------------+---------+-----------+----------------+----------------------+
|  1 | 2016-03-09 13:40:46 | life, the uni ... ything | aeiou   | 127.0.0.1 | firefox        | {'u', 'e', 'i', 'a'} |
|  2 | 2016-03-09 13:42:07 | hitch-hiker              | aeiou   | 127.0.0.1 | safari         | {'i', 'e'}           |
|  3 | 2016-03-09 13:42:15 | galaxy                   | xyz     | 127.0.0.1 | chrome         | {'y', 'x'}           |
|  4 | 2016-03-09 13:43:07 | hitch-hiker              | xyz     | 127.0.0.1 | firefox        | set()                |
+----+---------------------+--------------------------+---------+-----------+----------------+----------------------+
4 rows in set (0.0 sec)


mysql> quit
Bye
```

Don't forget to quit the console when you're done.

Remember: we're only storing the browser name.

# Storing Data Is Only Half the Battle

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

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

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

    import mysql.connector

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

## This new function is a big change

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

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

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

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

# How Best to Reuse Your Database Code?

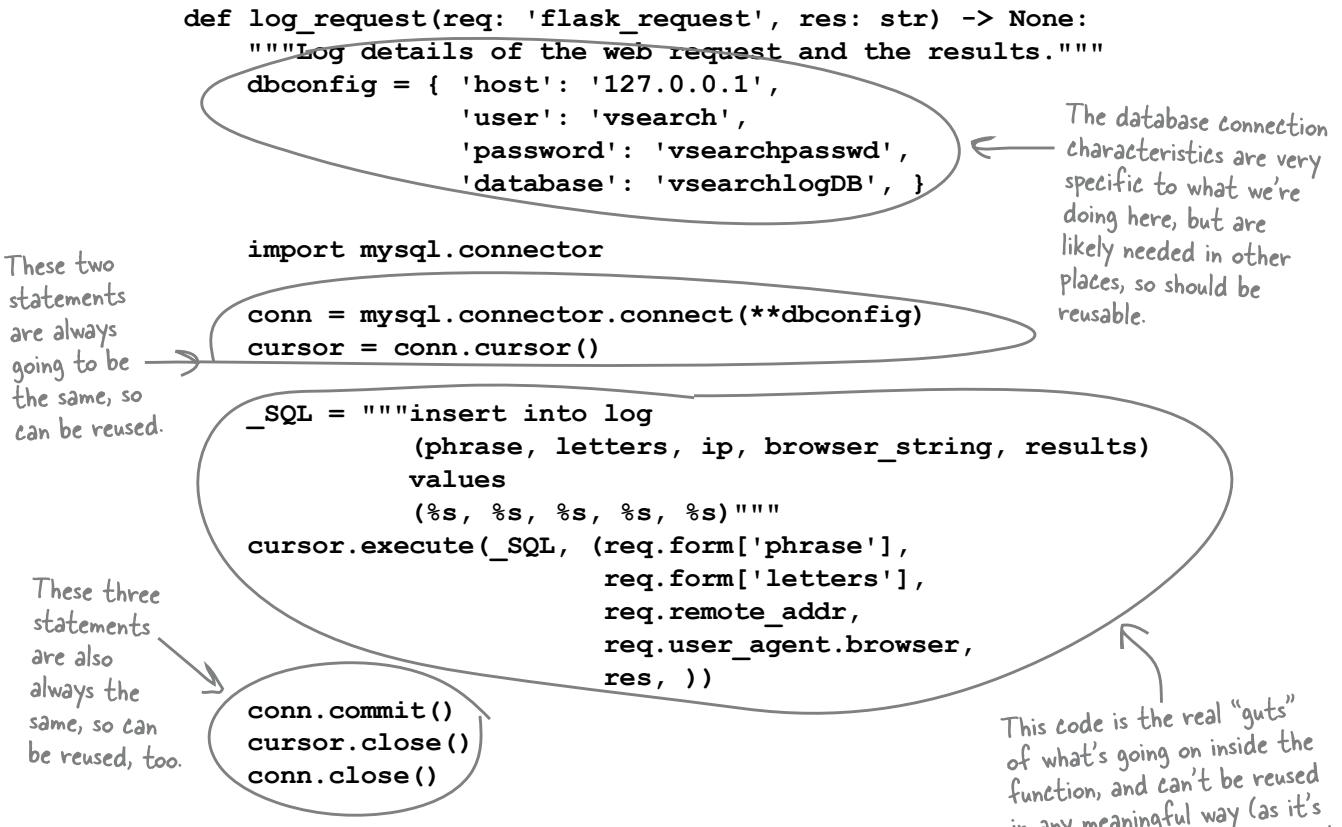
You now have code that logs the details of each of your webapp's requests to MySQL. It shouldn't be too much work to do something similar in order to retrieve the data from the `log` table for use in the `view_the_log` function. The question is: what's the best way to do this? We asked three programmers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


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

# Consider What You're Trying to Reuse

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

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




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

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

# What About That Import?


**Nope, we didn't forget.**

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

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

## Be careful when positioning your import statements

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

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

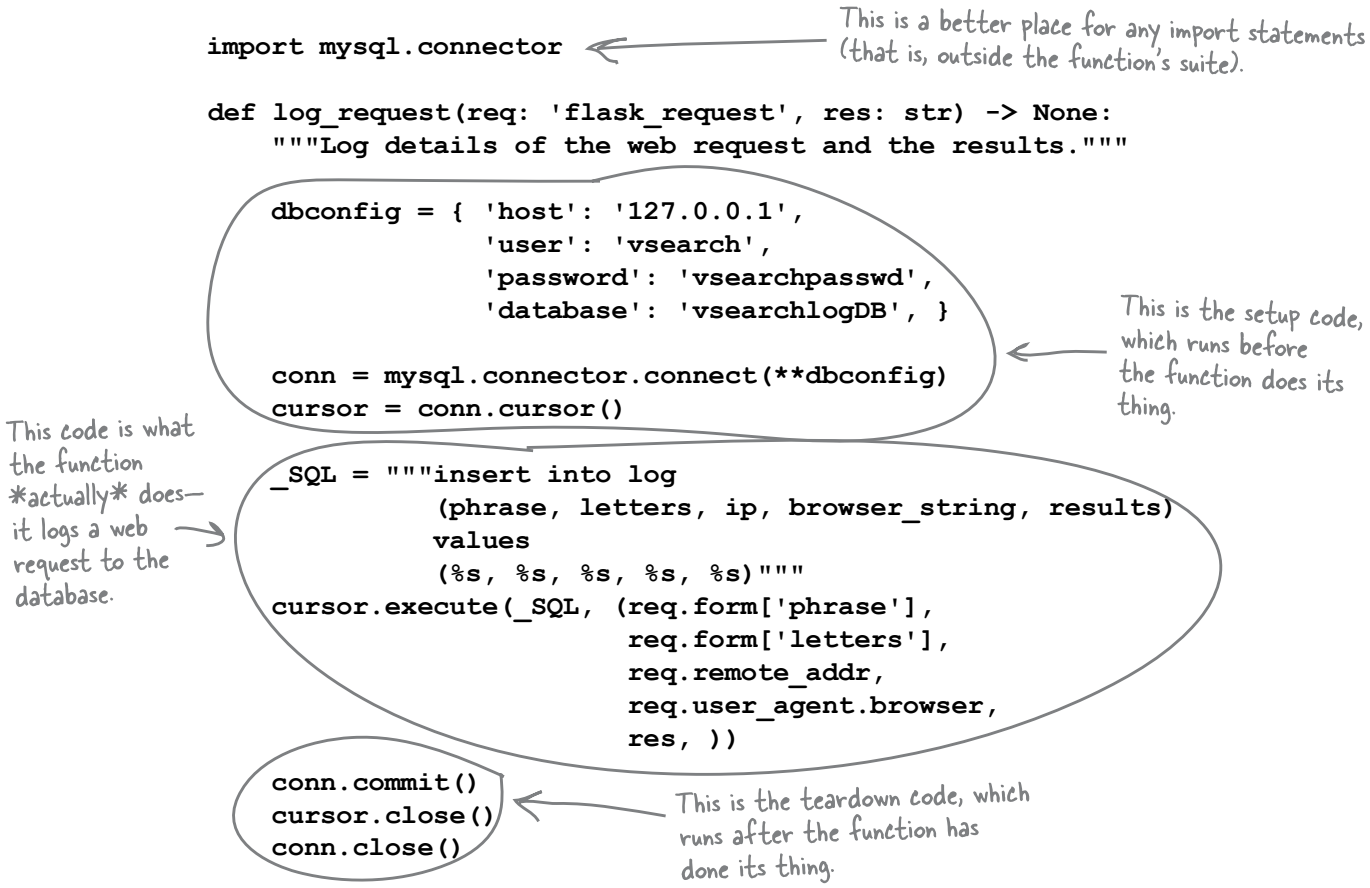
However, when an `import` statement appears *within* a function, it is read *and* executed **every time the function is called**. This is regarded as an extremely wasteful practice (even though, as we've seen, the interpreter won't stop you from putting an `import` statement in a function). Our advice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

# Consider What You're Trying to Do

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

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

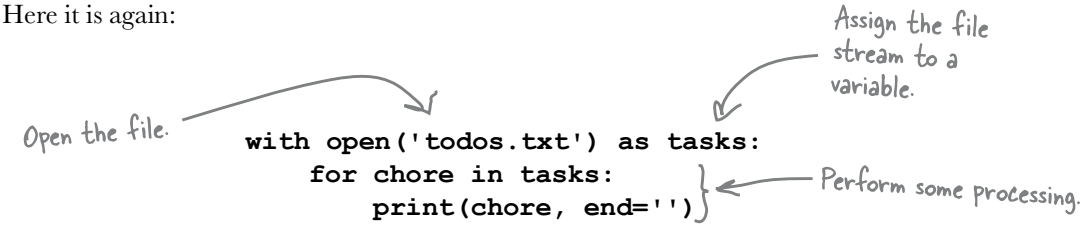
With this *setup, do, teardown* pattern in mind, let's look at the function once more. Note that we've repositioned the `import` statement to execute outside of the `log_request` function's suite (so as to avoid any further disapproving gasps):




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

# You've Seen This Pattern Before

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




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

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

We still need to define the connection characteristics.

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

This "with" statement works with databases as opposed to disk files, and returns a cursor for us to work with.

```
with UseDatabase(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, ))
```

The "do code" from the last page remains unchanged.

Don't try to run this code, as you've yet to write the "UseDatabase" context manager.

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

# The Bad News Isn't Really All That Bad

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

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

```
    ...
with UseDatabase(dbconfig) as cursor:
    ...
```

This "with" statement is similar to the one used with files and the "open" BIF, except that this one works with a database instead.

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

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

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

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

# Chapter 7's Code

This is the database code that currently runs within your webapp (i.e., the "log_request" function).

```
import mysql.connector

def log_request(req: 'flask_request', res: str) -> None:
    """Log details of the web request and the results."""

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

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

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

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

with UseDatabase(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, ))
```

This is the code that we'd like to be able to write in order to do the same thing as our current code (replacing the suite in the "log_request" function). But don't try to run this code yet, as it won't work without the "UseDatabase" context manager.

# 8 a little bit of class

# Abstracting Behavior and State


## Classes let you bundle code behavior and state together.

In this chapter, you're setting your webapp aside while you learn about creating Python **classes.** You're doing this in order to get to the point where you can create a context manager with the help of a Python class. As creating and using classes is such a useful thing to know about anyway, we're dedicating this chapter to them. We won't cover everything about classes, but we'll touch on all the bits you'll need to understand in order to confidently create the context manager your webapp is waiting for. Let's dive in and see what's involved.

# Hooking into the "with" Statement

At stated at the end of the last chapter, understanding how to hook your setup and teardown code into Python's `with` statement is straightforward...assuming you know how to create a Python **class**.

Despite being well over halfway through this book, you've managed to get by without having to define a class. You've written useful and reusable code using nothing more than Python's function machinery. There are other ways to write and organize your code, and object orientation is very popular.

You're never forced to program exclusively in the object-oriented paradigm when using Python, and the language is flexible when it comes to how you go about writing your code. But, when it comes to hooking into the `with` statement, doing so through a class is the **recommended approach**, even though the standard library comes with support for doing something similar *without* a class (although the standard library's approach is less widely applicable, so we aren't going to use it here).

So, to hook into the `with` statement, you'll have to create a class. Once you know how to write classes, you can then create one that implements and adheres to the **context management protocol**. This protocol is the mechanism (built into Python) that hooks into the `with` statement.

Let's learn how to create and use classes in Python, before returning to our context management protocol discussion in the next chapter.

**The context management protocol lets your write a class that hooks into the "with" statement.**

## there are no Dumb Questions

**Q: Exactly what type of programming language is Python: object-oriented, functional, or procedural?**

A: That's a great question, which many programmers moving to Python eventually ask. The answer is that Python supports programming paradigms borrowed from all three of these popular approaches, and Python encourages programmers to mix and match as needed. This concept can be hard to get your head around, especially if you come from the perspective where all the code you write has to be in a class that you instantiate objects from (as in other programming languages like, for instance, Java).

Our advice is not to let this worry you: create code in whatever paradigm you're comfortable with, but don't discount the others simply because—as approaches—they appear alien to you.

**Q: So...is it wrong to always start by creating a class?**

A: No, it isn't, if that's what your application needs. You don't have to put all your code in classes, but if you want to, Python won't get in your way.

So far in this book, we've gotten by without having to create a class, but we're now at the point where it makes sense to use one to solve a specific application issue we're grappling with: how best to share our database processing code within our webapp. We're mixing and matching programming paradigms to solve our current problem, and that's OK.

# An Object-Oriented Primer

Before we get going with classes, it's important to note that we don't intend to cover everything there is to know about classes in Python in this chapter. Our intention is merely to show you enough to enable you to confidently create a class that implements the context management protocol.

Therefore, we won't discuss some topics that seasoned practitioners of object-oriented programming (OOP) might expect to see here, such as *inheritance* and *polymorphism* (even though Python provides support for both). That's because we're primarily interested in **encapsulation** when creating a context manager.

If the jargon in that last paragraph has put you in a *blind panic*, don't worry: you can safely read on without knowing what any of that OOP-speak actually means.

On the last page, you learned that you need to create a class in order to hook into the `with` statement. Before getting to the specifics of how to do that, let's look at what constitutes a class in Python, writing an example class as we go. Once you understand how to write a class, we'll return to the problem of hooking into the `with` statement (in the next chapter).


**Don't be freaked out by all the buzzwords on this page!**

If we were to run a competition to determine the page in this book with this most buzzwords on it, this one would win hands-down. Don't be put off by all the jargon used here, though. If you already know OOP, this should all make sense. **If not, the really important bits are shown below**. Don't worry: all this will become clearer as you work through the example on the next few pages.

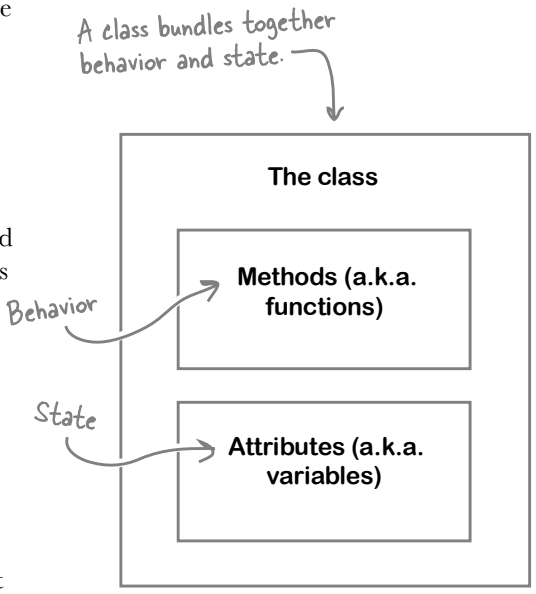
## A class bundles behavior and state

Using a class lets you bundle **behavior** and **state** together in an object.

When you hear the word *behavior*, think *function*—that is, a chunk of code that does something (or *implements a behavior*, if you prefer).

When you hear the word *state*, think *variables*—that is, a place to store values within a class. When we assert that a class bundles behavior and state *together*, we're simply stating that a class packages functions and variables.

The upshot of all of the above is this: if you know what a function is and what variables are, you're most of the way to understanding what a class is (as well as how to create one).




## Classes have methods and attributes

In Python, you define a class behavior by creating a method.

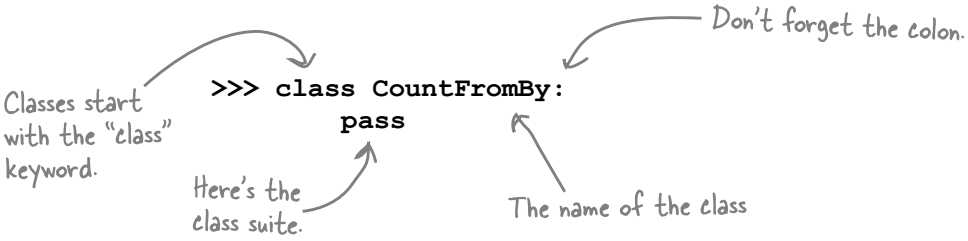
The word *method* is the OOP name given to a function that's defined within a class. Just why methods aren't simply known as *class functions* has been lost in the mists of time, as has the fact that *class variables* aren't referred to as such—they are known by the name *attribute*.

# Creating Objects from Classes

To use a class, you create an object from it (you'll see an example of this below). This is known as **object instantiation**. When you hear the word *instantiate*, think *invoke*; that is, you invoke a class to create an object.

Perhaps surprisingly, you can create a class that has no state or behavior, yet is still a class as far as Python is concerned. In effect, such a class is *empty*. Let's start our class examples with an empty one and take things from there. We'll work at the interpreter's >>> prompt, and you're encouraged to follow along.

We begin by creating an empty class called `CountFromBy`. We do this by prefixing the class name with the `class` keyword, then providing the suite of code that implements the class (after the obligatory colon):




Note how this class's suite contains the Python keyword `pass`, which is Python's empty statement (in that it does nothing). You can use `pass` in any place the interpreter expects to find actual code. In this case, we aren't quite ready to fill in the details of the `CountFromBy` class, so we use `pass` to avoid any syntax errors that would normally result when we try to create a class without any code in its suite.

"pass" is a valid statement (i.e., it is syntactically correct), but it does nothing. Think of it as an empty statement.

Now that the class exists, let's create two objects from it, one called `a` and another called `b`. Note how creating an object from a class looks very much like calling a function:

```
>>> a = CountFromBy()
>>> b = CountFromBy()
```

These look like function calls, don't they?

Create an object by appending parentheses to the class name, then assign the newly created object to a variable.

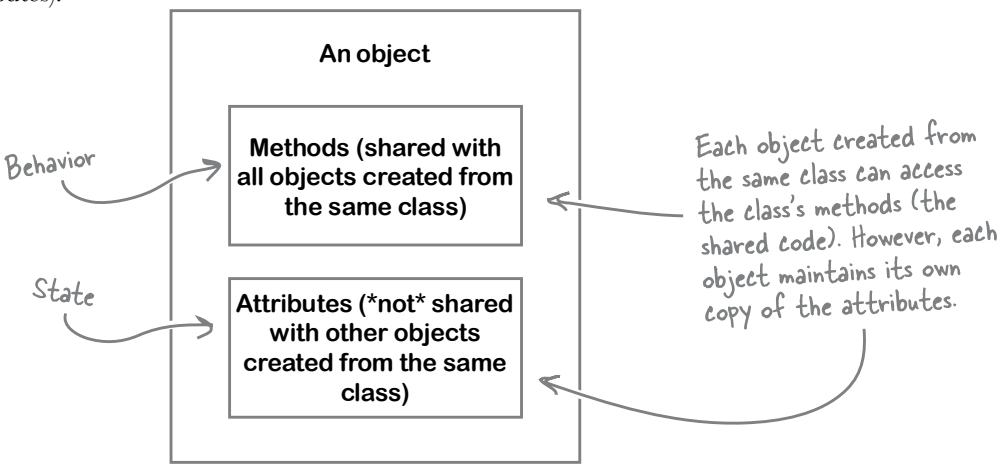
## there are no Dumb Questions

**Q: When I'm looking at someone else's code, how do I know if something like `CountFromBy()` is code that creates an object or code that calls a function? That looks like a function call to me...**

A: That's a great question. On the face of things, you don't know. However, there's a well-established convention in the Python programming community to name functions using lowercase letters (with underscores for emphasis), while *CamelCase* (concatenated words, capitalized) is used to name classes. Following this convention, it should be clear that `count_from_by()` is a function call, whereas `CountFromBy()` creates an object. All is fine just so long as everyone follows this convention, and you're **strongly encouraged** to do so, too. However, if you ignore this suggestion, all bets are off, and most Python programmers will likely avoid you and your code.

# Objects Share Behavior but Not State

When you create objects from a class, each object shares the class's coded behaviors (the methods defined in the class), but maintains its own copy of any state (the attributes):




This distinction will make more sense as we flesh out the `CountFromBy` example.

## Defining what we want CountFromBy to do

Let's now define what we want the `CountFromBy` class to actually do (as an empty class is rarely useful).

Let's make `CountFromBy` an incrementing counter. By default, the counter will start at 0 and be incremented (on request) by 1. We'll also make it possible to provide an alternative starting value and/or amount to increment by. This means you'll be able to create, for example, a `CountFromBy` object that starts at 100 and increments by 10.

Let's preview what the `CountFromBy` class will be able to do (once we have written its code). By understanding how the class will be used, you'll be better equipped to understand the `CountFromBy` code as we write it. Our first example uses the class defaults: start at 0, and increment by 1 on request by calling the `increase` method. The newly created object is assigned to a new variable, which we've called `c`:

**Note: this new "CountFromBy" class doesn't exist just yet. You'll create it in a little bit.**

```
>>> c = CountFromBy()
>>> c
0
>>> c.increase()
>>> c.increase()
>>> c.increase()
>>> c
3
```

Create another new object, and assign it to an object called "c".

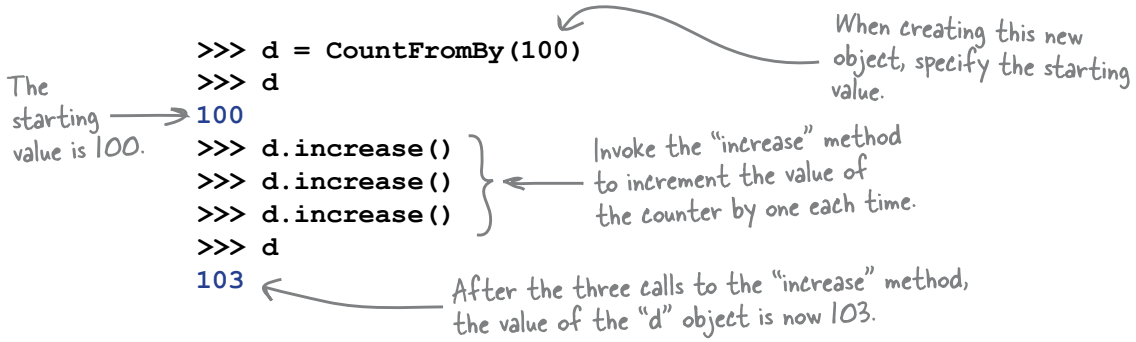
The starting value is 0.

Invoke the "increase" method to increment the value of the counter by one each time.

After the three calls to the "increase" method, the value of the object is now three.

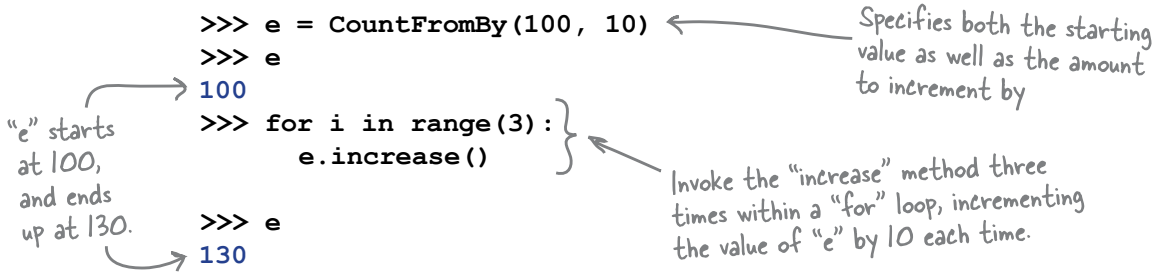
# Doing More with CountFromBy

The example usage of `CountFromBy` at the bottom of the last page demonstrated the default behavior: unless specified, the counter maintained by a `CountFromBy` object starts at 0 and is incremented by 1. It's also possible to specify an alternative starting value, as demonstrated in this next example, where the count starts from 100:



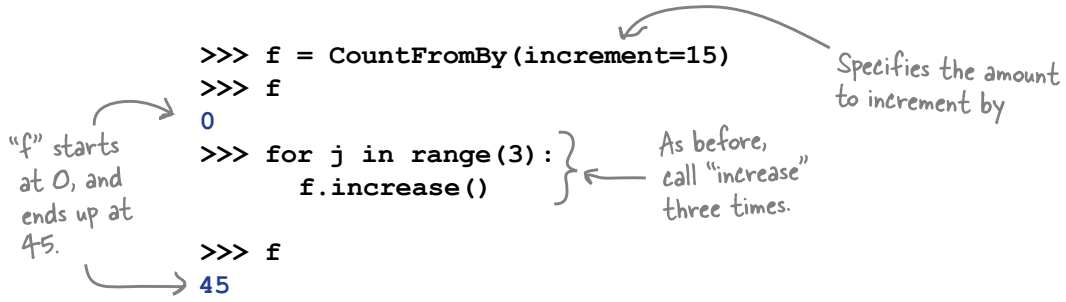

```
>>> d = CountFromBy(100)
>>> d
100
>>> d.increase()
>>> d.increase()
>>> d.increase()
>>> d
103
```

As well as specifying the starting value, it's also possible to specify the amount to increase by, as shown here, where we start at 100 and increment by 10:




```
>>> e = CountFromBy(100, 10)
>>> e
100
>>> for i in range(3):
        e.increase()

>>> e
130
```

In this final example, the counter starts at 0 (the default), but increments by 15. Rather than having to specify `(0, 15)` as the arguments to the class, this example uses a keyword argument that allows us to specify the amount to increment by, while leaving the starting value at the default (0):




```
>>> f = CountFromBy(increment=15)
>>> f
0
>>> for j in range(3):
        f.increase()

>>> f
45
```

# It's Worth Repeating Ourselves: Objects Share Behavior but Not State

The previous examples created four new `CountFromBy` objects: `c`, `d`, `e`, and `f`, each of which has access to the `increase` method, which is a behavior that's shared by all objects created from the `CountFromBy` class. There's only ever one copy of the `increase` method's code, which all these objects use. However, each object maintains its own attribute values. In these examples, that's the current value of the counter, which is different for each of the objects, as shown here:

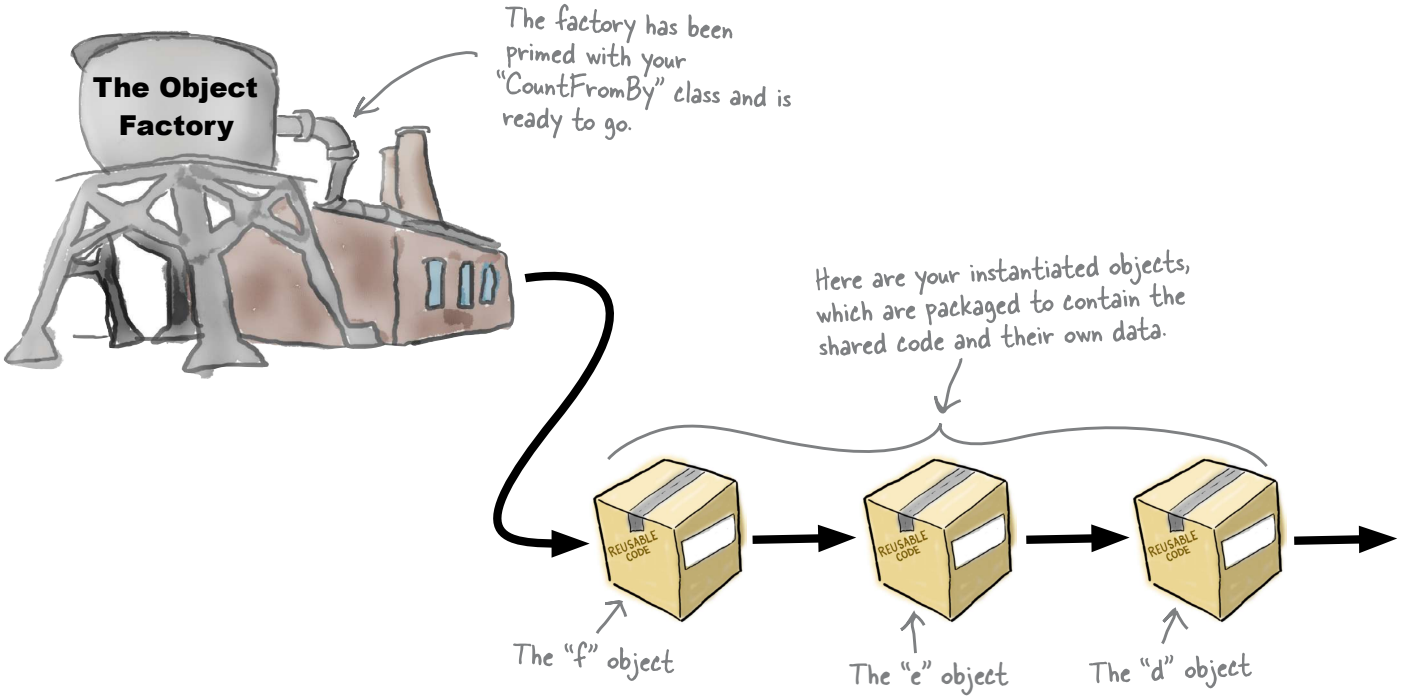
```
>>> c
3
>>> d
103
>>> e
130
>>> f
45
```

These four "CountFromBy" objects maintain their own attribute values.

**Class behavior is shared by each of its objects, whereas state is not. Each object maintains its own state.**

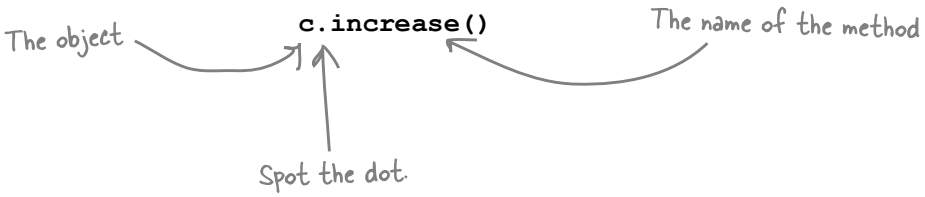
*Here's the key point again*: the method code is shared, but the attribute data isn't.

It can be useful to think of a class as a "cookie-cutter template" that is used by a factory to churn out objects that all behave the same, but have their own data.

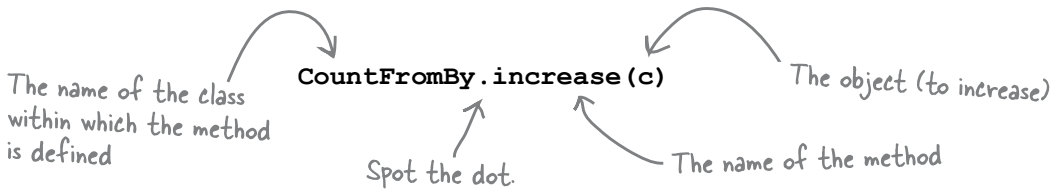


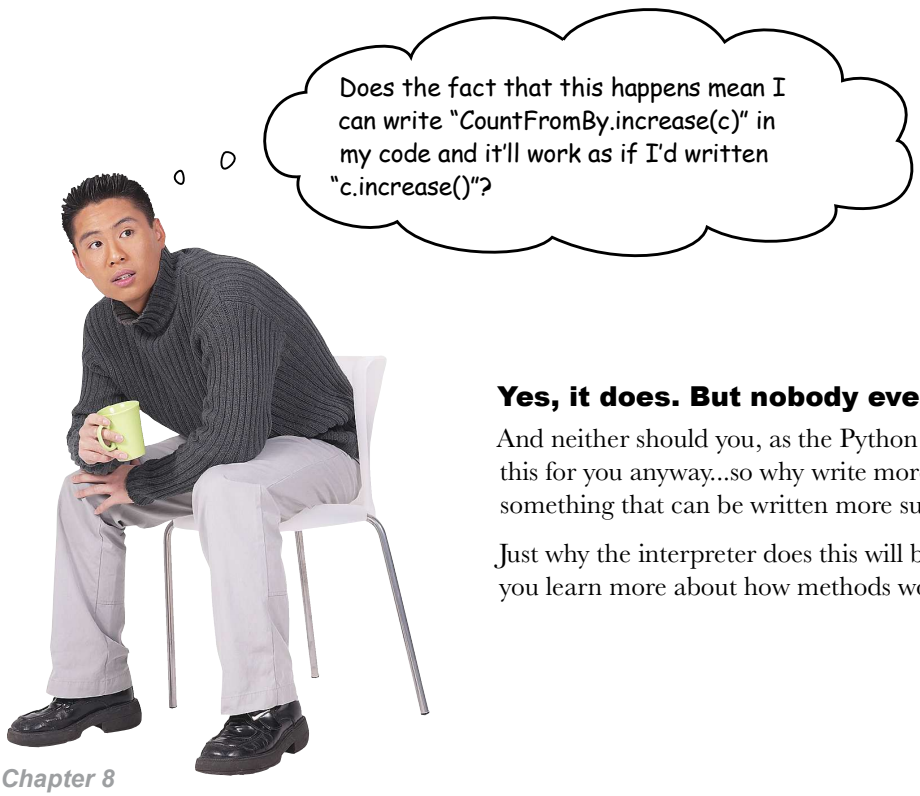
# Invoking a Method: Understand the Details

We stated earlier that a method is *a function defined within a class*. We also saw examples of a method from `CountFromBy` being invoked. The `increase` method is invoked using the familiar dot notation:




It is instructive to consider the code the interpreter *actually* executes (behind the scenes) when it encounters the above line. Here is the call the interpreter *always* turns the above line of code into. Note what happens to `c`:




**Yes, it does. But nobody ever does that.**

And neither should you, as the Python interpreter does this for you anyway...so why write more code to do something that can be written more succinctly?

Just why the interpreter does this will become clearer as you learn more about how methods work.

# Method Invocation: What Actually Happens

At first sight, the interpreter turning `c.increase()` into `CountFromBy.increase(c)` may look a little strange, but understanding that this happens helps explain why every method you write takes *at least* one argument.

It's OK for methods to take more than one argument, but the first argument *always* has to exist in order to take the object as an argument (which, in the example from the last page, is `c`). In fact, it is a well-established practice in the Python programming community to give each method's first argument a special name: `self`.

When `increase` is invoked as `c.increase()`, you'd imagine the method's `def` line should look like this:

```
def increase():
```

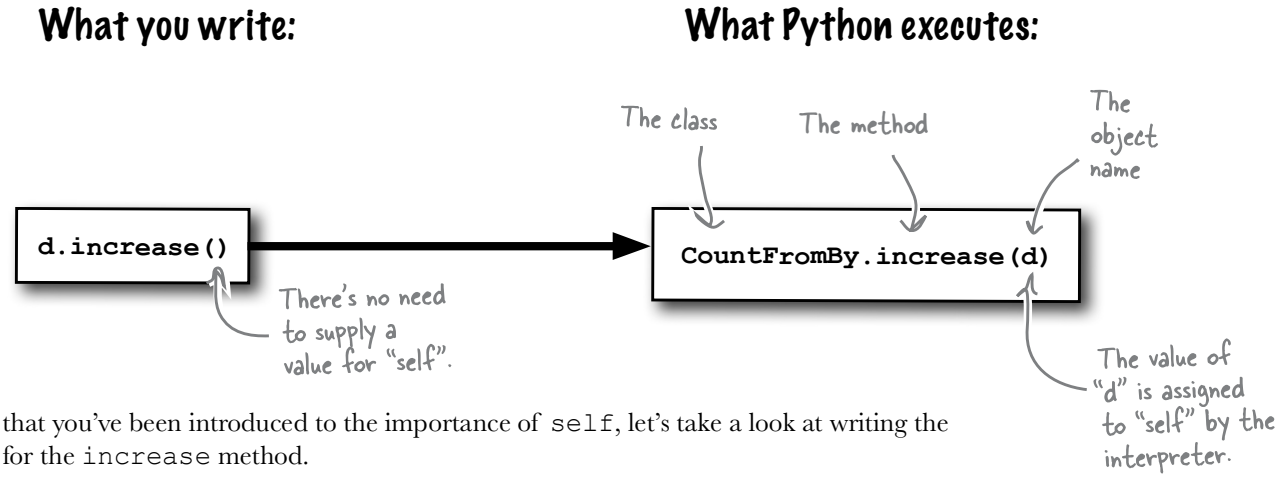
However, defining a method without the mandatory first argument will cause the interpreter to raise an error when your code runs. Consequently, the `increase` method's `def` line actually needs to be written as follows:

```
def increase(self):
```

When writing code in a class, think of "self" as an alias to the current object.

It is regarded as **very bad form** to use something other than the name `self` in your class code, even though the use of `self` does take a bit of getting used to. (Many other programming languages have a similar notion, although they favor the name `this`. Python's `self` is basically the same idea as `this`.)

When you invoke a method on an object, Python arranges for the first argument to be the invoking object instance, which is *always* assigned to each method's `self` argument. This fact alone explains why `self` is so important and also why `self` needs to be the *first argument* to every object method you write. When you invoke a method, you don't need to supply a value for `self`, as the interpreter does this for you:




Now that you've been introduced to the importance of `self`, let's take a look at writing the code for the `increase` method.

# Adding a Method to a Class

Let's create a new file to save our class code into. Create `countfromby.py`, then add in the class code from earlier in this chapter:

```
class CountFromBy:
    pass
```

We're going to add the `increase` method to this class, and to do so we'll remove the `pass` statement and replace it with `increase`'s method definition. Before doing this, recall how `increase` is invoked:

```
c.increase()
```

Based on this call, you'd be forgiven for assuming the `increase` method takes no arguments, as there's nothing between the parentheses, right? However, this is only half true. As you just learned, the interpreter transforms the above line of code into the following call:

```
CountFromBy.increase(c)
```

The method code we write needs to take this transformation into consideration. With all of the above in mind, here's the `def` line for the `increase` method that we'd use in this class:

```
class CountFromBy:
    def increase(self) -> None:
```

As with the other functions in this book, we provide an annotation for the return value.

Methods are just like functions, so are defined with "def".

The first argument to every method is always "self", and its value is supplied by the interpreter.

There are no other arguments to the `increase` method, so we do not need to provide anything other than `self` on the `def` line. However, it *is* vitally important that we include `self` here, as forgetting to results in syntax errors.

With the `def` line written, all we need to do now is add some code to `increase`. Let's assume that the class maintains two attributes: `val`, which contains the current value of the current object, and `incr`, which contains the amount to increment `val` by every time `increase` is invoked. Knowing this, you might be tempted to add this **incorrect** line of code to `increase` in an attempt to perform the increment:

```
val += incr
```

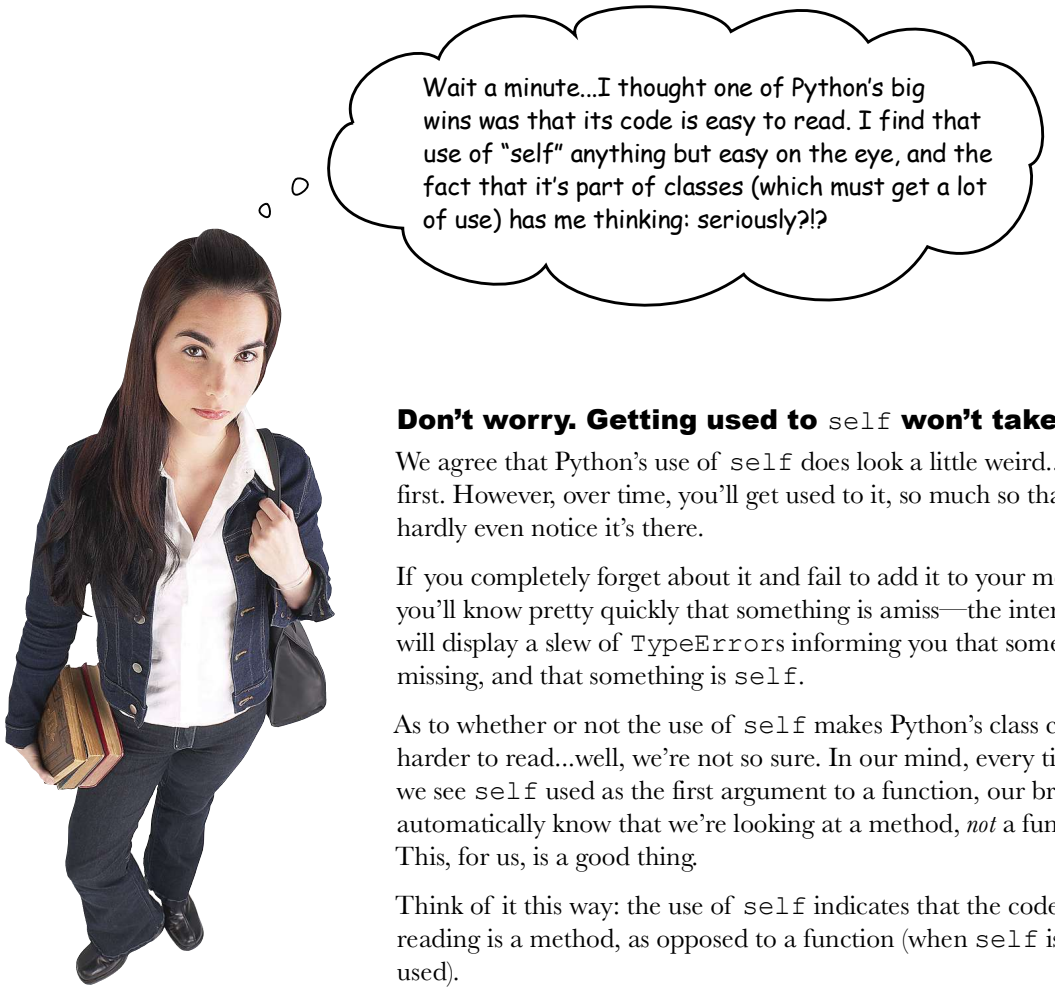
But here's the **correct** line of code to add to the `increase` method:

```
class CountFromBy:
    def increase(self) -> None:
        self.val += self.incr
```

Take the object's current value of "val" and increase it by the value of "incr".

Why do you think this line of code is correct, whereas the previous was incorrect?

# Are You Serious About "self"?




**Don't worry. Getting used to `self` won't take long.**

We agree that Python's use of `self` does look a little weird...at first. However, over time, you'll get used to it, so much so that you'll hardly even notice it's there.

If you completely forget about it and fail to add it to your methods, you'll know pretty quickly that something is amiss—the interpreter will display a slew of `TypeErrors` informing you that something is missing, and that something is `self`.

As to whether or not the use of `self` makes Python's class code harder to read...well, we're not so sure. In our mind, every time we see `self` used as the first argument to a function, our brains automatically know that we're looking at a method, *not* a function. This, for us, is a good thing.

Think of it this way: the use of `self` indicates that the code you're reading is a method, as opposed to a function (when `self` is *not* used).

# The Importance of "self"

The `increase` method, shown below, prefixes each of the class's attributes with `self` within its suite. You were asked to consider why this might be:

```
class CountFromBy:
    def increase(self) -> None:
        self.val += self.incr
```

What's the deal with using "self" within the method's suite?

You already know that `self` is assigned the current object by the interpreter when a method is invoked, and that the interpreter expects each method's first argument to take this into account (so that the assignment can occur).

Now, consider what we already know about each object created from a class: it shares the class's method code (a.k.a. behavior) with every other object created from the same class, but maintains its *own copy* of any attribute data (a.k.a. state). It does this by associating the attribute values with the object—that is, with `self`.

Knowing this, consider this version of the `increase` method, which, as we said a couple of pages ago, is **incorrect**:

```
class CountFromBy:
    def increase(self) -> None:
        val += incr
```

Don't do this—it won't do what you think it should.

On the face of things, that last line of code seems innocent enough, as all it does is increment the current value of `val` by the current value of `incr`. But consider what happens when this `increase` method terminates: `val` and `incr`, which exist *within* `increase`, both go out of scope and consequently are destroyed the moment the method ends.

Ummm...let me just make a note of "going out of scope" and "destroyed." I'll need to look both of those up later...or did I miss something?

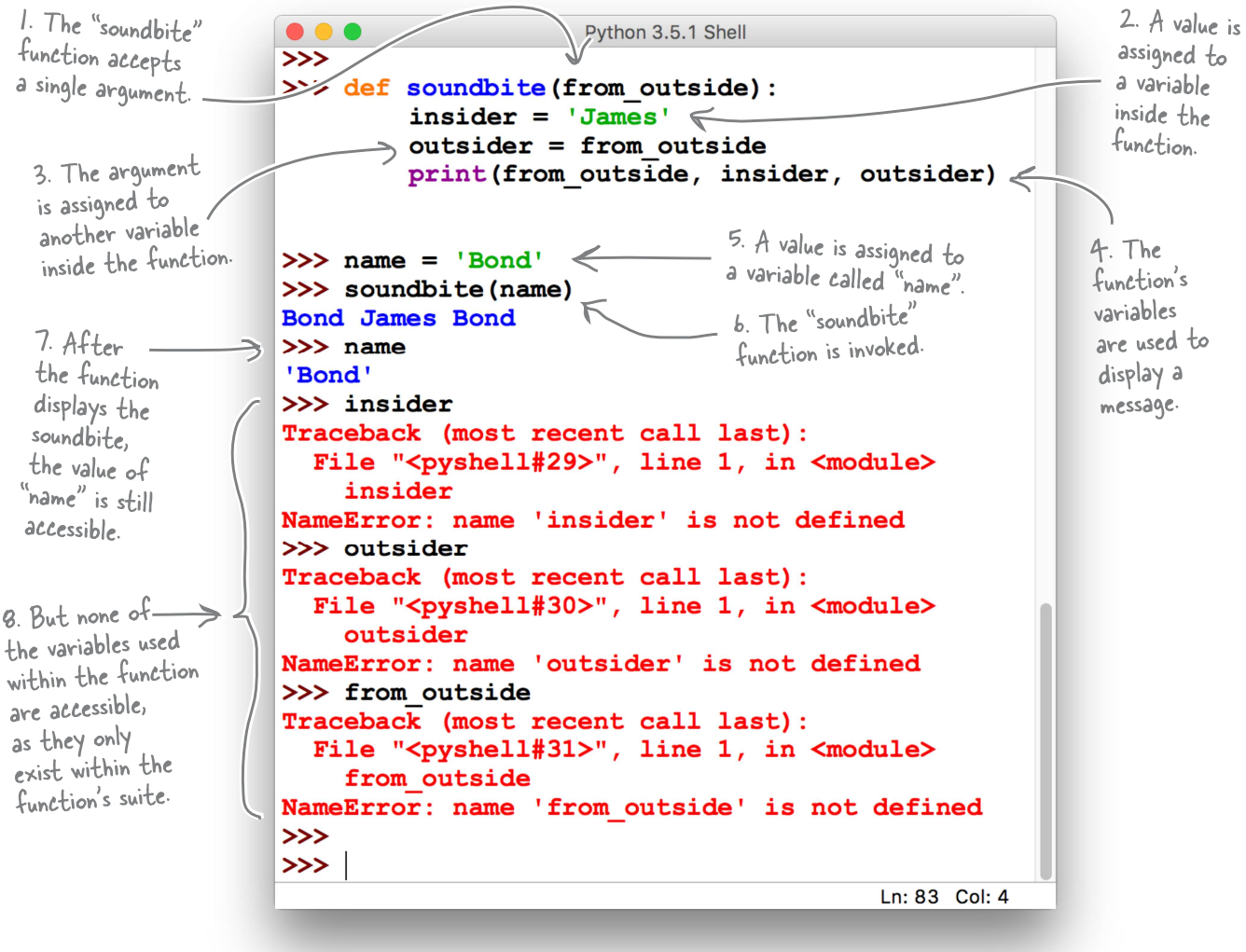
### Whoops. That's our bad...

We slipped in that statement about scope without much explanation, didn't we?

In order to understand what has to happen when you refer to attributes in a method, let's first spend some time understanding what happens to variables used in a function.

# Coping with Scoping

To demonstrate what happens to variables used within a function, let's experiment at the >>> prompt. Try out the code below as you read it. We've numbered the annotations 1 through 8 to guide you as you follow along:




When variables are defined within a function's suite, they exist while the function runs. That is, the variables are "in scope," both visible and usable within the function's suite. However, once the function ends, any variables defined within the function are destroyed—they are "out of scope," and any resources they used are reclaimed by the interpreter.

This is what happens to the three variables used within the `soundbite` function, as shown above. The moment the function terminates, `insider`, `outsider`, and `from_outside` cease to exist. Any attempt to refer to them outside the suite of function (a.k.a. outside the function's scope) results in a `NameError`.

# Prefix Your Attribute Names with "self"

This function behavior described on the last page is fine when you're dealing with a function that gets invoked, does some work, and then returns a value. You typically don't care what happens to any variables used within a function, as you're usually only interested in the function's return value.

Now that you know what happens to variables when a function ends, it should be clear that this (incorrect) code is likely to cause problems when you attempt to use variables to store and remember attribute values with a class. As methods are functions by another name, neither `val` nor `incr` will survive an invocation of the `increase` method if this is how you code `increase`:

```
class CountFromBy:
    def increase(self) -> None:
        val += incr
```

Don't do this, as these variables won't survive once the method ends.

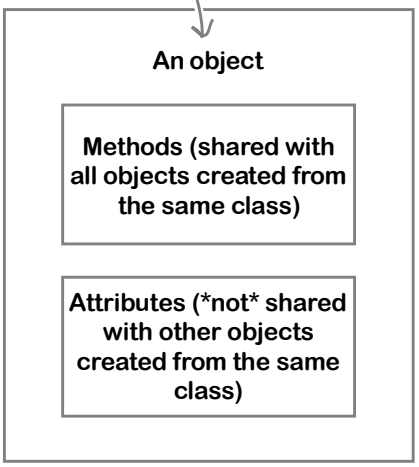
However, with methods, things are *different*. The method uses attribute values that belong to an object, and the object's attributes continue to exist *after* the method terminates. That is, an object's attribute values are **not** destroyed when the method terminates.

In order for an attribute assignment to survive method termination, the attribute value has to be assigned to something that doesn't get destroyed as soon as the method ends. That *something* is the current object invoking the method, which is stored in `self`, which explains why each attribute value needs to be prefixed with `self` in your method code, as shown here:

```
class CountFromBy:
    def increase(self) -> None:
        self.val += self.incr
```

This is much better, as "val" and "incr" are now associated with the object thanks to the use of "self".

"self" is an alias to the object.



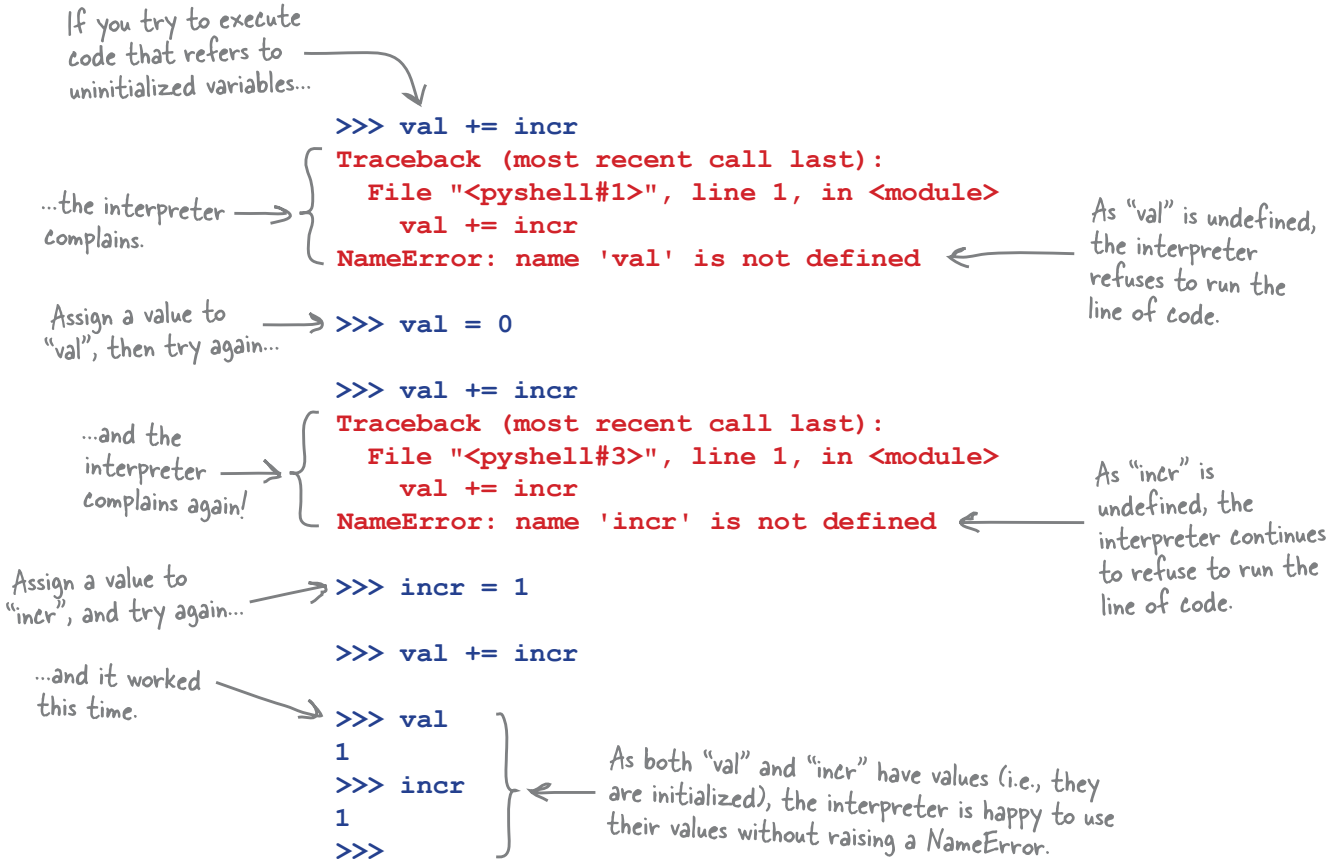

The rule is straightforward: if you need to refer to an attribute in your class, you *must* prefix the attribute name with `self`. The value in `self` as an *alias* that points back to the object invoking the method.

In this context, when you see `self`, think "this object's." So, `self.val` can be read as "this object's `val`."

# Initialize (Attribute) Values Before Use

All of the discussion of the importance of `self` sidestepped an important issue: how are attributes assigned a starting value? As it stands, the code in the `increase` method—the correct code, which uses `self`—fails if you execute it. This failure occurs because in Python you can't use a variable before it has been assigned a value, no matter where the variable is used.

To demonstrate the seriousness of this issue, consider this short session at the >>> prompt. Note how the first statement fails to execute when *either* of the variables is undefined:

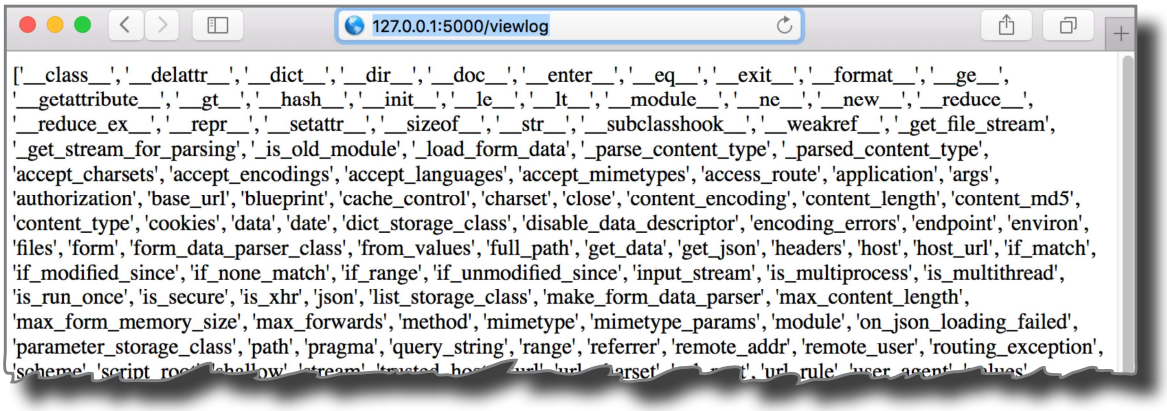



No matter where you use variables in Python, you have to initialize them with a starting value. The question is: *how do we do this for a new object created from a Python class?*

If you know OOP, the word "constructor" may be popping into your brain right about now. In other languages, a constructor is a special method that lets you define what happens when an object is first created, and it usually involves both object instantiation and attribute initialization. In Python, object instantiation is handled automatically by the interpreter, so you don't need to define a constructor to do this. A magic method called `__init__` lets you initialize attributes as needed. Let's take a look at what dunder `init` can do.

# Dunder "init" Initializes Attributes

Cast your mind back to the last chapter, when you used the `dir` built-in function to display all the details of Flask's `req` object. Remember this output?




At the time, we suggested you ignore all those dunders. However, it's now time to reveal their purpose: the dunders provide hooks into every class's standard behavior.

Unless you override it, this standard behavior is implemented in a class called `object`. The `object` class is built into the interpreter, and every other Python class *automatically* inherits from it (including yours). This is OOP-speak for stating that the dunder methods provided by `object` are available to your class to use as is, or to override as needed (by providing your own implementation of them).

You don't have to override any `object` methods if you don't want to. But if, for example, you want to specify what happens when objects created from your class are used with the equality operator (==), then you can write your own code for the `__eq__` method. If you want to specify what happens when objects are used with the greater-than operator (>), you can override the `__ge__` method. And when you want to *initialize* the attributes associated with your object, you can use the `__init__` method.

As the dunders provided by `object` are so useful, they're held in near-mystical reverence by Python programmers. So much so, in fact, that many Python programmers refer to these dunders as *the magic methods* (as they give the appearance of doing what they do "as if by magic").

**The standard dunder methods, available to all classes, are known as "the magic methods."**

All of this means that if you provide a method in your class with a `def` line like the one below, the interpreter will call your `__init__` method every time you create a new object from your class. Note the inclusion of `self` as this dunder `init`'s first argument (as per the rule for all methods in all classes):

```
def __init__(self):
```

Despite the strange-looking name, dunder "init" is a method like any other. Remember: you must pass "self" as its first argument.

# Initializing Attributes with Dunder "init"

Let's add `__init__` to our `CountFromBy` class in order to initialize the objects we create from our class.

For now, let's add an *empty* `__init__` method that does nothing but `pass` (we'll add behavior in just a moment):

```
class CountFromBy:
    def __init__(self) -> None:
        pass
    def increase(self) -> None:
        self.val += self.incr
```

At the moment, this dunder "init" doesn't do anything. However, the use of "self" as its first argument is a BIG CLUE that dunder "init" is a method.

We know from the code already in `increase` that we can access attributes in our class by prefixing their names with `self`. This means we can use `self.val` and `self.incr` to refer to our attributes within `__init__`, too. However, we want to use `__init__` to *initialize* our class's attributes (`val` and `incr`). The question is: where do these initialization values come from and how do their values get into `__init__`?

## Pass any amount of argument data to dunder "init"

As `__init__` is a method, and methods are functions in disguise, you can pass as many argument values as you like to `__init__` (or any method, for that matter). All you have to do is give your arguments names. Let's give the argument that we'll use to initialize `self.val` the name `v`, and use the name `i` for `self.incr`.

Let's add `v` and `i` to the `def` line of our `__init__` method, then use the values in dunder `init`'s suite to initialize our class attributes, as follows:

```
class CountFromBy:
    def __init__(self, v: int, i: int) -> None:
        self.val = v
        self.incr = i
    def increase(self) -> None:
        self.val += self.incr
```

Add "v" and "i" as arguments to dunder "init".

Use the values of "v" and "i" to initialize the class's attributes (which are "self.val" and "self.incr", respectively).

If we can now somehow arrange for `v` and `i` to acquire values, the latest version of `__init__` will initialize our class's attributes. Which raises yet another question: how do we get values into `v` and `i`? To help answer this question, we need to try out this version of our class and see what happens. Let's do that now.

# Test Drive

Using the edit window in IDLE, take a moment to update the code in your `countfromby.py` file to look like that shown below. When you're done, press F5 to start creating objects at IDLE's >>> prompt:

Press F5 to try out the "CountFromBy" class in IDLE's shell.

countfromby.py - /Users/paul/Desktop/_NewBook/ch08/countfromby.py (3.5.1)

```
class CountFromBy:

    def __init__(self, v: int, i: int) -> None:
        self.val = v
        self.incr = i

    def increase(self) -> None:
        self.val += self.incr
```

Ln: 2 Col: 0

The latest version of our "CountFromBy" class.

Pressing F5 executes the code in the edit window, which imports the `CountFromBy` class into the interpreter. Look at what happens when we try to create a new object from our `CountFromBy` class:

Create a new object (called "g") from the class...but when you do this, you get an error!

Python 3.5.1 Shell

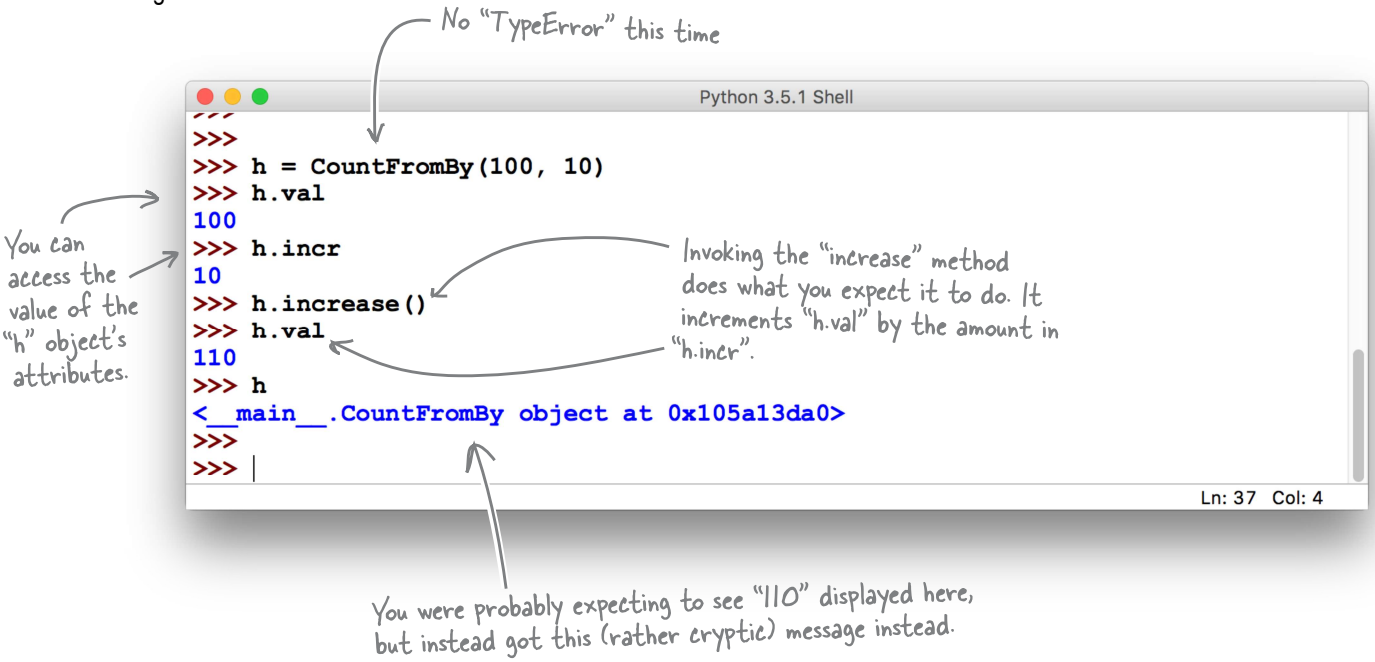
```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>>
========= RESTART: /Users/paul/Desktop/_NewBook/ch07/countfromby.py =========
>>>
>>> g = CountFromBy()
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    g = CountFromBy()
TypeError: __init__() missing 2 required positional arguments: 'v' and 'i'
>>>
>>>
```

Ln: 13 Col: 4

This may not have been what you were expecting to see. But take a look at the error message (which is classed as a `TypeError`), paying particular attention to the message on the `TypeError` line. The interpreter is telling us that the `__init__` method expected to receive two argument values, `v` and `i`, but received something else (in this case, nothing). We provided no arguments to the class, but this error message tells us that any arguments provided to the class (when creating a new object) are passed to the `__init__` method.

Bearing this in mind, let's have another go at creating a `CountFromBy` object.

Let's return to the >>> prompt, and create another object (called `h`) that takes two integer values as arguments for `v` and `i`:




As you can see above, things work better this time, as the `TypeError` exception is gone, which means the `h` object was created successfully. You can access the values of `h`'s attributes using `h.val` and `h.incr`, as well as call the object's `increase` method. Only when you try to access the value of `h` do things get strange again.

## What have we learned from this Test Drive?

Here are the main takeaways from this *Test Drive*:

- When you're creating objects, any argument values provided to the class are passed to the `__init__` method, as was the case with `100` and `10` above. (Note that `v` and `i` cease to exist as soon as dunder `init` ends, but we aren't worried, as their values are safely stored in the object's `self.val` and `self.incr` attributes, respectively.)
- We can access the attribute values by combining the object's name with the attribute name. Note how we used `h.val` and `h.incr` to do this. (For those readers coming to Python from a "stricter" OOP language, note that we did this without having to create *getters* or *setters*.)
- When we use the object name on its own (as in the last interaction with the shell above), the interpreter spits back a cryptic message. Just what this is (and why this happens) will be discussed next.

# Understanding CountFromBy's Representation

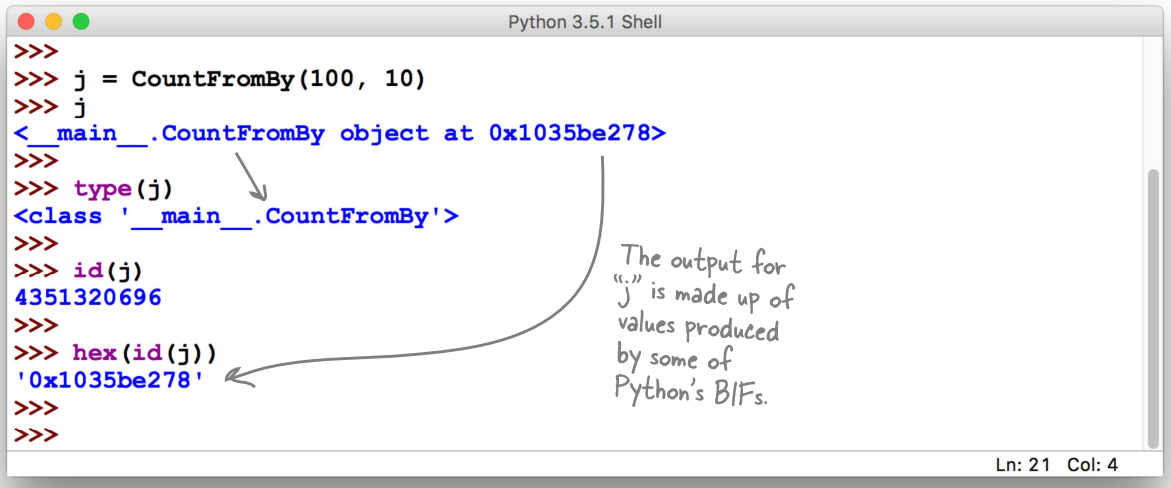
When we typed the name of the object into the shell in an attempt to display its current value, the interpreter produced this output:

```
<__main__.CountFromBy object at 0x105a13da0>
```

Don't worry if you have a different value here. All will become clear before the end of this page.

We described the above output as "strange," and on first glance, it would certainly appear to be. To understand what this output means, let's return to IDLE's shell and create yet another object from `CountFromBy`, which due to our deeply ingrained unwillingness to rock the boat, we're calling `j`.

In the session below, note how the strange message displayed for `j` is made up of values that are produced when we call certain built-in functions (BIFs). Follow along with the session first, then read on for an explanation of what these BIFs do:




The `type` BIF displays information on the class the object was created from, reporting (above) that `j` is a `CountFromBy` object.

The `id` BIF displays information on an object's memory address (which is a unique identifier used by the interpreter to keep track of your objects). What you see on your screen is likely different from what is reported above.

The memory address displayed as part of `j`'s output is the value of `id` converted to a hexadecimal number (which is what the `hex` BIF does). So, the entire message displayed for `j` is a combination of `type`'s output, as well as `id`'s (converted to hexadecimal).

A reasonable question is: *why does this happen?*

In the absence of you telling the interpreter how you want to represent your objects, the interpreter has to do *something*, so it does what's shown above. Thankfully, you can override this default behavior by coding your own `__repr__` magic method.

**Override dunder "repr" to specify how your objects are represented by the interpreter.**

# Defining CountFromBy's Representation

As well as being a magic method, the `__repr__` functionality is also available as a built-in function called `repr`. Here's part of what the `help` BIF displays when you ask it to tell you what `repr` does: "Return the canonical string representation of the object." In other words, the `help` BIF is telling you that `repr` (and by extension, `__repr__`) needs to return a stringified version of an object.

What this "stringified version of an object" looks like depends on what each individual object does. You can control what happens for *your* objects by writing a `__repr__` method for your class. Let's do this now for the `CountFromBy` class.

Begin by adding a new `def` line to the `CountFromBy` class for dunder `repr`, which takes no arguments other than the required `self` (remember: it's a method). As is our practice, let's also add an annotation that lets readers of our code know this method returns a string:

Like every other method you'll write, this one has to take into account that the interpreter always provides a value for the first argument.

```
def __repr__(self) -> str:
```

This lets users of this method know that this function intends to return a string. Remember: using annotations in your code is optional, but helpful.

With the `def` line written, all that remains is to write the code that returns a string representation of a `CountFromBy` object. For our purposes, all we want to do here is take the value in `self.val`, which is an integer, and convert it to a string.

Thanks to the `str` BIF, doing so is straightforward:

```
def __repr__(self) -> str:
    return str(self.val)
```

Take the value in "self.val", turn it into a string, and then return it to this method's caller.

When you add this short function to your class, the interpreter uses it whenever it needs to display a `CountFromBy` object at the >>> prompt. The `print` BIF also uses dunder `repr` to display objects.

Before making this change and taking the updated code for a spin, let's return briefly to another issue that surfaced during the last *Test Drive*.

# Providing Sensible Defaults for CountFromBy

Let's remind ourselves of the current version of the `CountFromBy` class's `__init__` method:

```
        ...
def __init__(self, v: int, i: int) -> None:
    self.val = v
    self.incr = i
        ...
```

This version of the dunder "init" method expects two argument values to be provided every time it is invoked.

Recall that when we tried to create a new object from this class without passing values for `v` and `i`, we got a `TypeError`:

```
>>>
>>> g = CountFromBy()
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    g = CountFromBy()
TypeError: __init__() missing 2 required positional arguments: 'v' and 'i'
>>>
>>>
```

Yikes! Not good.

Earlier in this chapter, we specified that we wanted the `CountFromBy` class to support the following default behavior: the counter will start at 0 and be incremented (on request) by 1. You already know how to provide default values to function arguments, and the same goes for methods, too—assign the default values on the `def` line:

```
        ...
def __init__(self, v: int=0, i: int=1) -> None:
    self.val = v
    self.incr = i
        ...
```

As methods are functions, they support the use of default values for arguments (although we're scoring a B– here for our use of single-character variable names: "v" is the value, whereas "i" is the incrementing value).

If you make this small (but important) change to your `CountFromBy` code, then save the file (before pressing F5 once more), you'll see that objects can now be created with this default behavior:

```
Python 3.5.1 Shell
>>>
>>>
>>> i = CountFromBy()
>>> i.val
0
>>> i.incr
1
>>> i.increase()
>>> i.val
1
>>>
>>>
Ln: 50  Col: 4
```

We haven't specified values to use when initializing the object, so the class provides the default values as specified in dunder "init".

This all works as expected, with the "increase" method incremented "i.val" by one each time it's invoked. This is the default behavior.

# Test Drive

Make sure your class code (in `countfromby.py`) is the same as ours below. With your class code loaded into IDLE's edit window, press F5 to take your latest version of the `CountFromBy` class for a spin:

This is the "CountFromBy" class with the code for dunder "repr" added.

countfromby.py - /Users/paul/Desktop/_NewBook/ch08/countfromby.py (3.5.1)

```
class CountFromBy:

    def __init__(self, v: int=0, i: int=1) -> None:
        self.val = v
        self.incr = i

    def increase(self) -> None:
        self.val += self.incr

    def __repr__(self) -> str:
        return str(self.val)
```

Ln: 13 Col: 0

Python 3.5.1 Shell

```
>>> k = CountFromBy()
>>> k
0
>>> k.increase()
>>> k
1
>>> print(k)
1
>>> l = CountFromBy(100)
>>> l
100
>>> l.increase()
>>> print(l)
101
>>> m = CountFromBy(100, 10)
>>> m
100
>>> m.increase()
>>> m
110
>>> n = CountFromBy(i=15)
>>> n
0
>>> n.increase()
>>> n
15
>>>
```

Ln: 33 Col: 4

The "k" object uses the class's default values, which start at 0 and are increased by 1.

When you refer to the object at the >>> prompt, or in a call to "print", the dunder "repr" code runs.

The "l" object provides an alternative starting value, then increments by 1 each time "increase" is called.

The "m" object provides alternative values for both defaults.

The "n" object uses a keyword argument to provide an alternative value to increment by (but starts at 0).

# Classes: What We Know

With the `CountFromBy` class behaving as specified earlier in this chapter, let's review what we now know about classes in Python:

## BULLET POINTS

- Python classes let you share **behavior** (a.k.a. methods) and **state** (a.k.a. attributes).
- If you remember that methods are **functions**, and attributes are **variables**, you won't go far wrong.
- The `class` keyword introduces a new class in your code.
- Creating a new object from a class looks very like a function call. Remember: to create an object called `mycount` from a class called `CountFromBy`, you'd use this line of code:

```
mycount = CountFromBy()
```

- When an object is created from a class, the object **shares** the class's code with every other object created from the class. However, each object maintains its **own copy** of the attributes.
- You add behaviors to a class by creating **methods**. A method is a function defined within a class.
- To add an **attribute** to a class, create a variable.
- Every method is passed an **alias** to the current object as its first argument. Python convention insists that this first argument is called `self`.
- Within a method's suite, referrals to attributes are prefixed with `self`, ensuring the attribute's value **survives** after the method code ends.
- The `__init__` method is one of the many **magic methods** provided with all Python classes.
- Attribute values are initialized by the `__init__` method (a.k.a. dunder `init`). This method lets you assign starting values to your attributes when a new object is created. Dunder `init` receives a **copy** of any values passed to the class when an object is created. For example, the values `100` and `10` are passed into `__init__` when this object is created:

```
mycount2 = CountFromBy(100, 10)
```

- Another magic method is `__repr__`, which allows you to control how an object appears when displayed at the `>>>` prompt, as well as when used with the `print` BIF.


### We wanted to create a context manager.

We know it's been a while, but the reason we started down this path was to learn enough about classes to enable us to create code that hooks into Python's **context management protocol**. If we can hook into the protocol, we can use our webapp's database code with Python's `with` statement, as doing so should make it easier to share the database code, as well as reuse it. Now that you know a bit about classes, you're ready to get hooked into the context management protocol (in the next chapter).

# Chapter 8's Code

This is the code in the "countfromby.py" file.

```
class CountFromBy:

    def __init__(self, v: int=0, i: int=1) -> None:
        self.val = v
        self.incr = i

    def increase(self) -> None:
        self.val += self.incr

    def __repr__(self) -> str:
        return str(self.val)
```

# 9 the context management protocol

# Hooking into Python's with Statement


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

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

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

During Chapter 7 you created database code in your `log_request` function that worked, but you had to pause to consider how best to share it. Recall the suggestions from the end of Chapter 7:


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

# Consider What You're Trying to Do, Revisited

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

```
import mysql.connector

def log_request(req: 'flask_request', res: str) -> None:
    """Log details of the web request and the results."""

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

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

    _SQL = """insert into log
              (phrase, letters, ip, browser_string, results)
              values
              (%s, %s, %s, %s, %s)"""
    cursor.execute(_SQL, (req.form['phrase'],
                          req.form['letters'],
                          req.remote_addr,
                          req.user_agent.browser,
                          res, ))

    conn.commit()
    cursor.close()
    conn.close()
```

This dictionary details the database connection characteristics.

This bit uses the credentials to connect to the database, then creates a cursor.

This is the code that does the actual work: it adds the request data to the "log" database table.

Finally, this code tears down the database connection.

## How best to create a context manager?

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

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

# Managing Context with Methods

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

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

## Dunder "enter" performs setup

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

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

## Dunder "exit" does teardown

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

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

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

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

## (As you know) dunder "init" initializes

In addition to dunder `enter` and dunder `exit`, you can add other methods to your class as needed, including defining your own `__init__` method. As you know from the last chapter, defining dunder `init` lets you perform additional object initialization. Dunder `init` runs *before* __enter__ (that is, *before your context manager's setup code executes*).

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

# You've Already Seen a Context Manager in Action

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

Your first-ever "with" statement (borrowed from Chapter 6).

```
with open('todos.txt') as tasks:
    for chore in tasks:
        print(chore, end='')
```

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

1. When the interpreter encounters this "with" statement, it begins by calling any dunder "init" associated with the call to "open".

2. As soon as dunder "init" executes, the interpreter calls dunder "enter" to ensure that the result of calling "open" will be assigned to the "tasks" variable.

```
with open('todos.txt') as tasks:
    for chore in tasks:
        print(chore, end='')
```

3. When the "with" statement ends, the interpreter calls the context manager's dunder "exit" to tidy up. In this example, the interpreter ensures that the opened file is closed properly before continuing.

## What's required from you

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

1. an `__init__` method to perform initialization (if needed);
2. an `__enter__` method to do any setup; and
3. an `__exit__` method to do any teardown (a.k.a. tidying-up).

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

# Create a New Context Manager Class

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

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

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

1. provide an `__init__` method that performs initialization;
2. provide an `__enter__` method that includes any setup code; and
3. provide an `__exit__` method that includes any teardown code.

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

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

This is what our "DBcm.py" file looks like in IDLE. At the moment, it's made up from a single "import" statement, together with a class called "UseDatabase" that contains three "empty" methods.

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

```
import mysql.connector


class UseDatabase:

    def __init__(self):
        pass

    def __enter__(self):
        pass

    def __exit__(self):
        pass
```

Ln: 14 Col: 0

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

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

# Initialize the Class with the Database Config

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

Import the context manager from the "DBcm.py" file.

Here's the database connection characteristics.

The context manager returns a "cursor".

The "UseDatabase" context manager expects to receive a dictionary of database connection characteristics.

This code stays the same as before.

```
from DBcm import UseDatabase

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

with UseDatabase(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, ))
```

## Sharpen your pencil

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

Complete the "def" line.

Is there anything missing from here?

Save the configuration dictionary to an attribute.

```
import mysql.connector

class UseDatabase:

    def __init__(self, ..... )

        .....
```

## Sharpen your pencil Solution

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

```
import mysql.connector


class UseDatabase:

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

Dunder "init" accepts a single dictionary, which we're calling "config".

The value of the "config" argument is assigned to an attribute called "configuration". Did you remember to prefix the attribute with "self"?

The (optional) "None" annotation confirms that this method has no return value (which is nice to know), and the colon terminates the "def" line.

## Your context manager begins to take shape

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

Make sure your dunder "init" matches ours.

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

```
import mysql.connector


class UseDatabase:

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

    def __enter__(self):
        pass

    def __exit__(self):
        pass
```

Ln: 14 Col: 0

# Perform Setup with Dunder "enter"

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

Here's the setup code from the "log_request" function.

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

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

_SQL = """insert into log
          (phrase, letters, ip, browser_string, results)
        ...
```

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

## Sharpen your pencil

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

Add the setup code here.

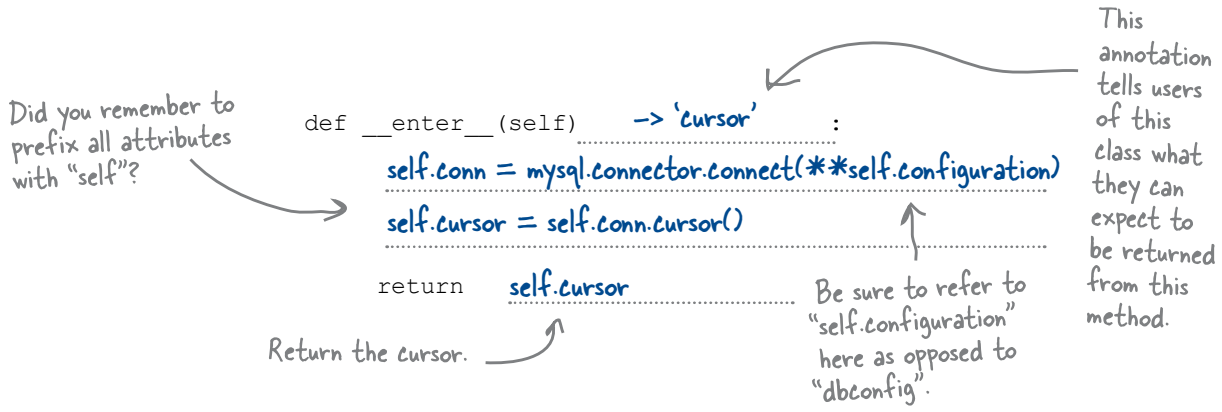
Can you think of an appropriate annotation?

```
def __enter__(self)..........:
    ..........
    ..........
    return ..........
```

Don't forget to return the cursor.

## Sharpen your pencil Solution

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




## Don't forget to prefix all attributes with "self"

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

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

You're nearly done. Only one more method to write.

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

```
import mysql.connector


class UseDatabase:

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

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

    def __exit__(self):
        pass
```

Ln: 16 Col: 0

# Perform Teardown with Dunder "exit"

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

```
            ...
cursor.execute(_SQL, (req.form['phrase'],
                      req.form['letters'],
                      req.remote_addr,
                      req.user_agent.browser,
                      res, ))

conn.commit()
cursor.close()
conn.close()
```

This is the teardown code.

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

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

## Sharpen your pencil

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

Don't worry about these arguments for now.

```
def __exit__(self, exc_type, exc_value, exc_trace) ..... :
```

Add the teardown code here.

.....

.....

.....

## Sharpen your pencil Solution

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

Don't worry about these arguments for now.

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

The previously saved attributes are used to commit unsaved data, as well as close the cursor and connection. As always, remember to prefix your attribute names with "self".

This annotation confirms that this method has no return value; such annotations are optional but are good practice..

## Your context manager is ready for testing

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

The completed "UseDatabase" context manager class.

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

```
import mysql.connector


class UseDatabase:

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

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

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

Ln: 18 Col: 0

A "real" class would include documentation, but we've removed it from this code to save on space (on this page). This book's downloads always include comments.

# Test Drive

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

Import the context manager class from the "DBcm.py" module file.

Use the context manager to send some SQL to the server and get some data back.

Put the connection characteristics in a dictionary.

```
Python 3.5.1 Shell
>>>
>>> from DBcm import UseDatabase
>>>
>>> dbconfig = { 'host': '127.0.0.1',
                 'user': 'vsearch',
                 'password': 'vsearchpasswd',
                 'database': 'vsearchlogDB', }
>>>
>>> with UseDatabase(dbconfig) as cursor:
        _SQL = """show tables"""
        cursor.execute(_SQL)
        data = cursor.fetchall()


>>> data
[('log',)]
>>>
>>>
                                          Ln: 45  Col: 4
```

The returned data may look a little strange...until you remember that the "cursor.fetchall" call returns a list of tuples, with each tuple corresponding to a row of results (as returned from the database).

## There's not much code here, is there?

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

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

# Reconsidering Your Webapp Code, 1 of 2

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

The last time you worked on it (in Chapter 7), you updated the `log_request` function to save the webapp's web request to the MySQL database. The reason we started down the path to learning about classes (in Chapter 8) was to determine the best way to share the database code you added to `log_request`. We now know that the best way (for this situation) is to use the just-written `UseDatabase` context manager class.

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

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

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

import mysql.connector

app = Flask(__name__)


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

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

We need to import "DBcm" here instead.

This code has to be amended to use the "UseDatabase" context manager.

# Reconsidering Your Webapp Code, 2 of 2

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    """Extract the posted data; perform the search; return results."""
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)


@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    """Display this webapp's HTML form."""
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')


@app.route('/viewlog')
def view_the_log() -> 'html':
    """Display the contents of the log file as a HTML table."""
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    titles = ('Form Data', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)


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

This code needs to be amended to use the data in the database via the "UseDatabase" context manager.

# Recalling the "log_request" Function

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

Take a look at `log_request` once more. At the moment, the database connection characteristics dictionary (`dbconfig` in the code) is defined within `log_request`. As you'll want to use this dictionary in the other function you have to amend (`view_the_log`), let's move it out of the `log_request`'s function so that you can share it with other functions as needed:

Let's move this dictionary out of the function so it can be shared with other functions as required.

```
def log_request(req: 'flask_request', res: str) -> None:

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

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

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

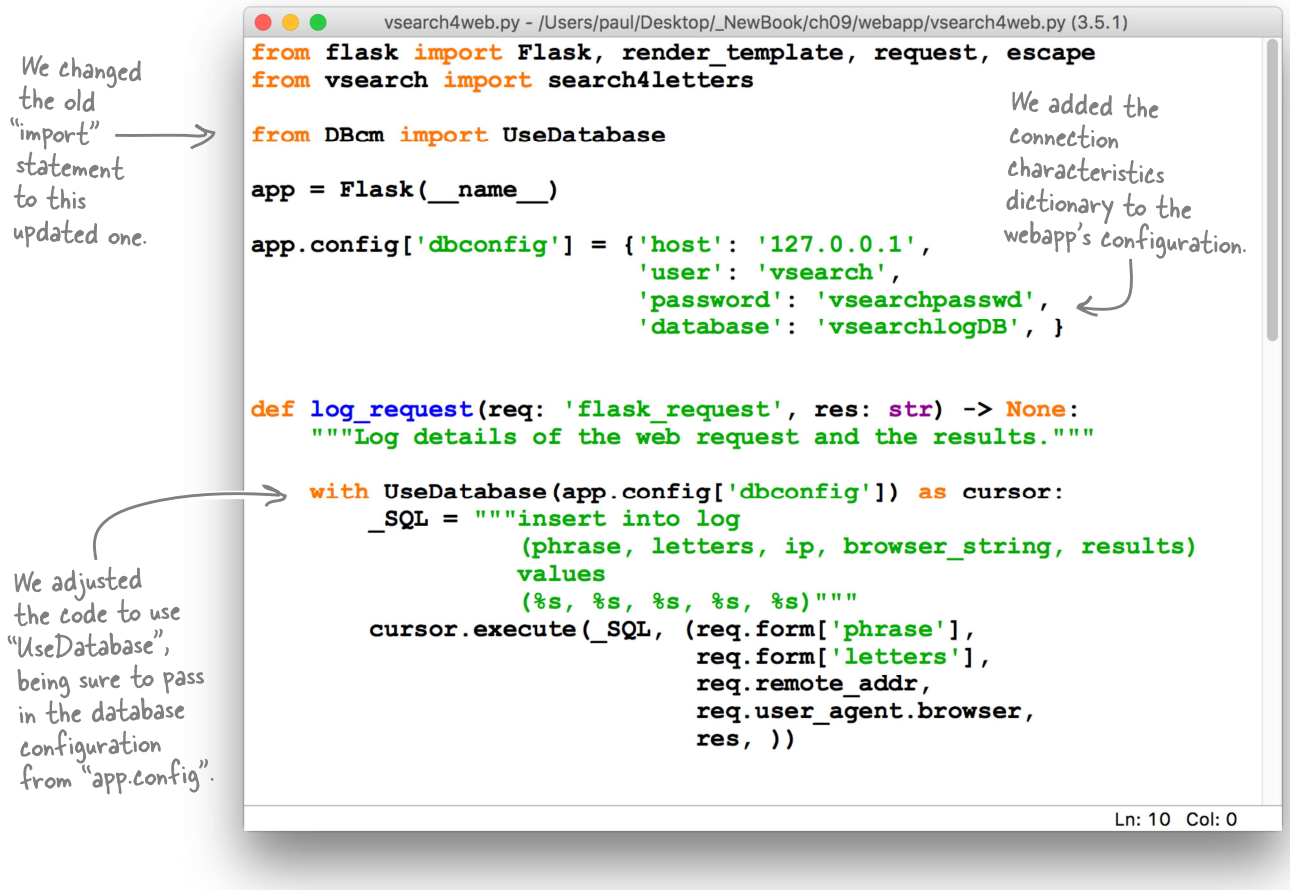
As luck would have it, Flask (like many other web frameworks) comes with a built-in configuration mechanism: a dictionary (which Flask calls `app.config`) allows you to adjust some of your webapp's internal settings. As `app.config` is a regular Python dictionary, you can add your own keys and values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

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

**Let's make these changes now.**

# Amending the "log_request" Function

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




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

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

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

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

# Recalling the "view_the_log" Function

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

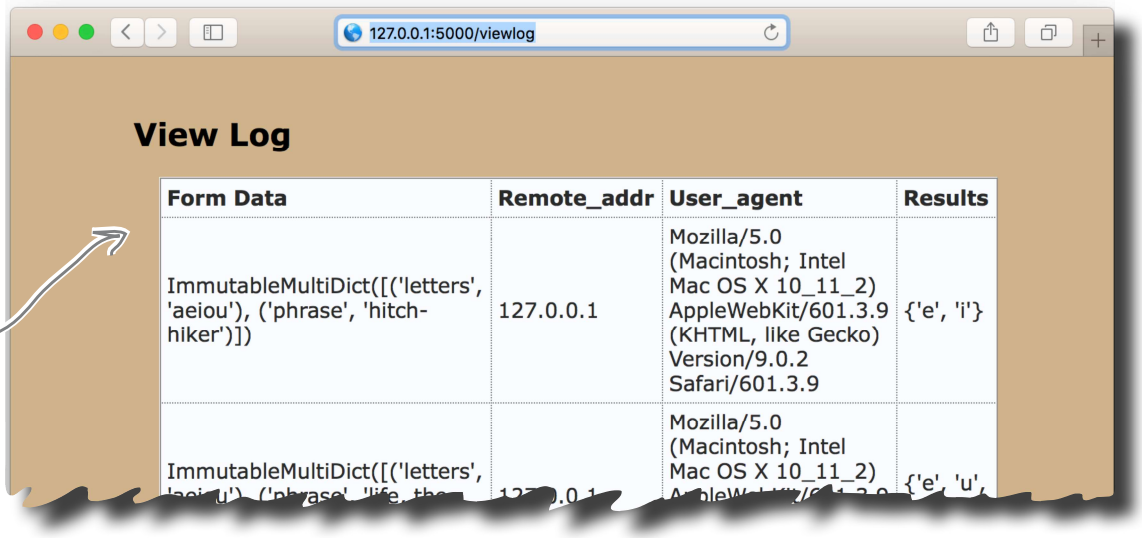
Grab each line of data from the file, and then transform it into a list of escaped items, which are appended to the "contents" list.

```
@app.route('/viewlog')
def view_the_log() -> 'html':

    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))

    titles = ('Form Data', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)
```

The processed log data is sent to the template for display.

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

The data from "contents" is displayed in the form. Note how the form data ("phrase" and "letters") is presented in a single column.



# It's Not Just the Code That Changes

Before diving in and changing the code in `view_the_log` to use your context manager, let's pause to consider the data as stored in the `log` table in your database. When you tested your initial `log_request` code in Chapter 7, you were able to log into the MySQL console, then check that the data was saved. Recall this MySQL console session from earlier:

File Edit Window Help Checking our log DB

```
$ mysql -u vsearch -p vsearchlogDB
Enter password:

Welcome to MySQL monitor...


mysql> select * from log;
+----+---------------------+--------------------------+---------+-----------+----------------+----------------------+
| id | ts                  | phrase                   | letters | ip        | browser_string | results              |
+----+---------------------+--------------------------+---------+-----------+----------------+----------------------+
|  1 | 2016-03-09 13:40:46 | life, the uni ... ything | aeiou   | 127.0.0.1 | firefox        | {'u', 'e', 'i', 'a'} |
|  2 | 2016-03-09 13:42:07 | hitch-hiker              | aeiou   | 127.0.0.1 | safari         | {'i', 'e'}           |
|  3 | 2016-03-09 13:42:15 | galaxy                   | xyz     | 127.0.0.1 | chrome         | {'y', 'x'}           |
|  4 | 2016-03-09 13:43:07 | hitch-hiker              | xyz     | 127.0.0.1 | firefox        | set()                |
+----+---------------------+--------------------------+---------+-----------+----------------+----------------------+
4 rows in set (0.0 sec)


mysql> quit
Bye
```

The log data saved in a database table

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

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

The log data saved as one long string in the "vsearch.log" file.

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

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

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

# Amending the "view_the_log" Function

Based on everything discussed on the last few pages, you've two things to do to amend your current `view_the_log` code:

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

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

Here's the `view_the_log` function's current code, which you are about to amend:

```
@app.route('/viewlog')
def view_the_log() -> 'html':
    contents = []
    with open('vsearch.log') as log:
        for line in log:
            contents.append([])
            for item in line.split('|'):
                contents[-1].append(escape(item))
    titles = ('Form Data', 'Remote_addr', 'User_agent', 'Results')
    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)
```

As a result of task #1 above, this code needs to be replaced.

As a result of task #2 above, this line needs to be amended.

## Here's the SQL query you'll need

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

```
select phrase, letters, ip, browser_string, results
from log
```

## Sharpen your pencil

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

```
@app.route('/viewlog')
def view_the_log() -> 'html':

    with ..... :

        _SQL = """select phrase, letters, ip, browser_string, results
                  from log"""

        .....
        .....

    titles = ( ....., ....., 'Remote_addr', 'User_agent', 'Results')

    return render_template('viewlog.html',
                           the_title='View Log',
                           the_row_titles=titles,
                           the_data=contents,)
```

Use your context manager here, and don't forget the cursor.

Send the query to the server, then fetch the results.

Which column titles are missing from here?

I'm just going to make a note of what's going on here. Not only is my new code shorter than what I had before, it's easier for me to understand and read, too.

**Yep—that was our goal all along.**

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

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

## Sharpen your pencil Solution

Here's the `view_the_log` function, which has to be amended to use the data in the `log` table. Your job was to provide the missing code.

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

This is the same line of code from the "log_request" function.

Send the query to the server, then fetch the results. Note the assignment of the fetched data to "contents".

Add in the correct column names.

## It's nearly time for one last Test Drive

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

vsearch4web.py - /Users/paul/Desktop/_NewBook/ch09/webapp/vsearch4web.py (3.5.1)

```
@app.route('/viewlog')
def view_the_log() -> 'html':
    """Display the contents of the log file as a HTML table."""
    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,)
```

Ln: 1 Col: 0

# Test Drive

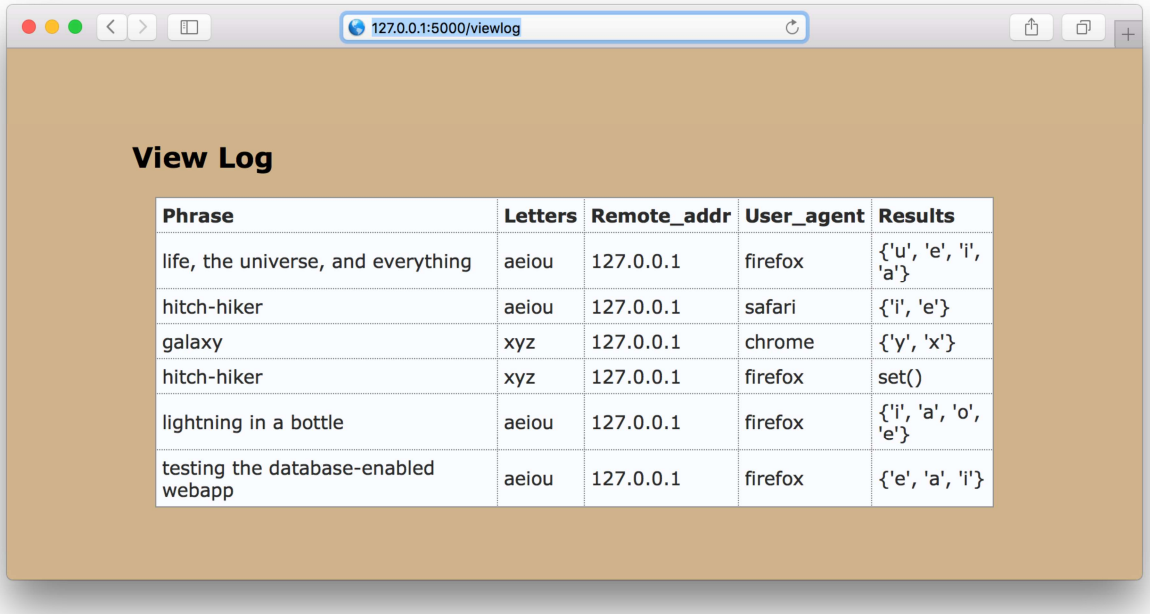
It's time to take your database-ready webapp for a spin.

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

- Use `python3 vsearch4web.py` on *Linux*/*Mac OS X*
- Use `py -3 vsearch4web.py` on *Windows*.

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

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




| Phrase | Letters | Remote_addr | User_agent | Results |
|---|---|---|---|---|
| life, the universe, and everything | aeiou | 127.0.0.1 | firefox | {'u', 'e', 'i', 'a'} |
| hitch-hiker | aeiou | 127.0.0.1 | safari | {'i', 'e'} |
| galaxy | xyz | 127.0.0.1 | chrome | {'y', 'x'} |
| hitch-hiker | xyz | 127.0.0.1 | firefox | set() |
| lightning in a bottle | aeiou | 127.0.0.1 | firefox | {'i', 'a', 'o', 'e'} |
| testing the database-enabled webapp | aeiou | 127.0.0.1 | firefox | {'e', 'a', 'i'} |

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

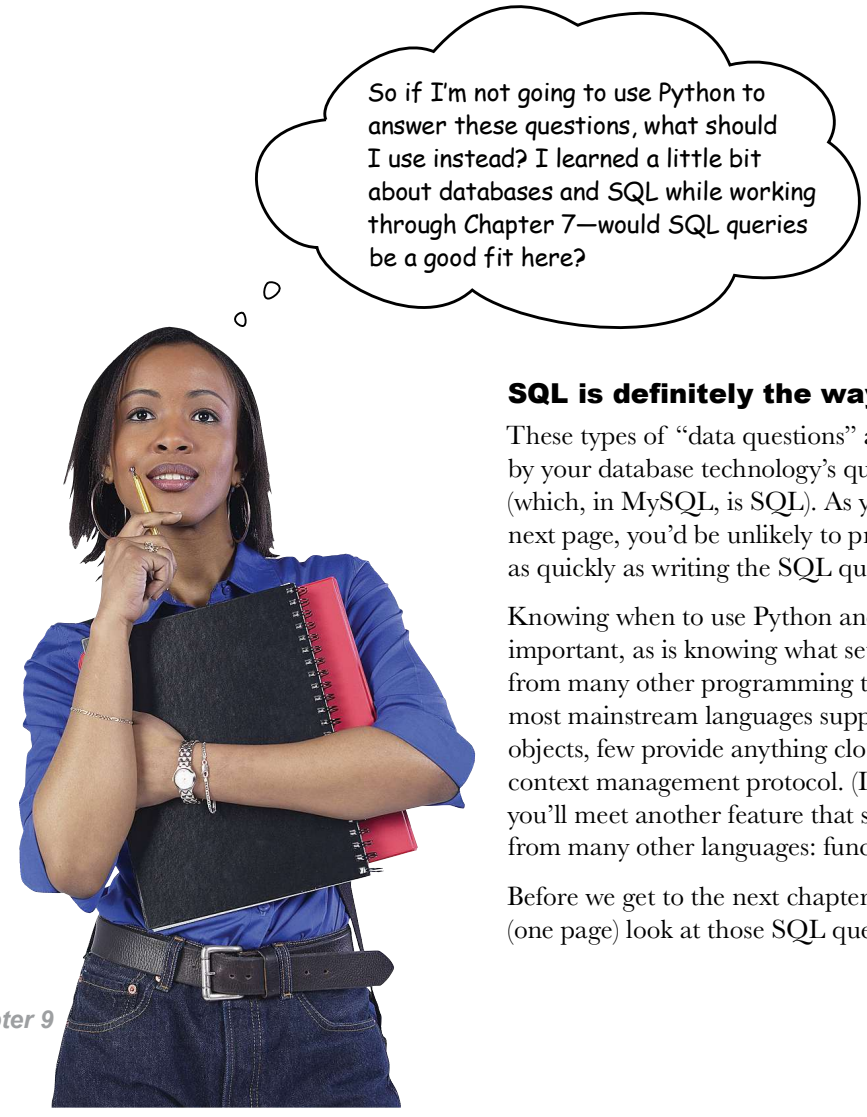
Only if you feel the need, take a few moments to log into your MySQL database using the MySQL console to confirm that the data is safely stored in your database server. (Or just trust us: based on what our webapp is displaying above, it is.)

# All That Remains...

It's now time to return to the questions first posed in Chapter 7:

- *How many requests have been responded to?*
- *What's the most common list of letters?*
- *Which IP addresses are the requests coming from?*
- *Which browser is being used the most?*

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




**SQL is definitely the way to go.**

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

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

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

# Answering the Data Questions

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

## How many requests have been responded to?

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

```
select * from log;
```

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

```
select count(*) from log;
```

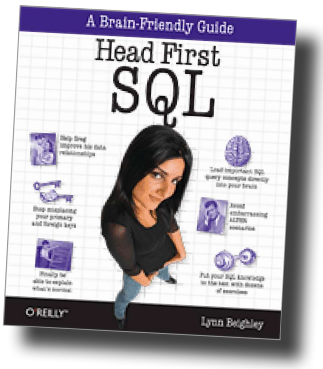
We're *not* showing you the answers here. If you want to see them, you'll have to run these queries yourself in the MySQL console (see Chapter 7 for a refresh).

## What's the most common list of letters?

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

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

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




## Which IP addresses are the requests coming from?

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

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

## Which browser is being used the most?

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

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

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

# Chapter 9's Code, 1 of 2

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

```
import mysql.connector

class UseDatabase:

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

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

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

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

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

from DBcm import UseDatabase

app = Flask(__name__)

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

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

# Chapter 9's Code, 2 of 2

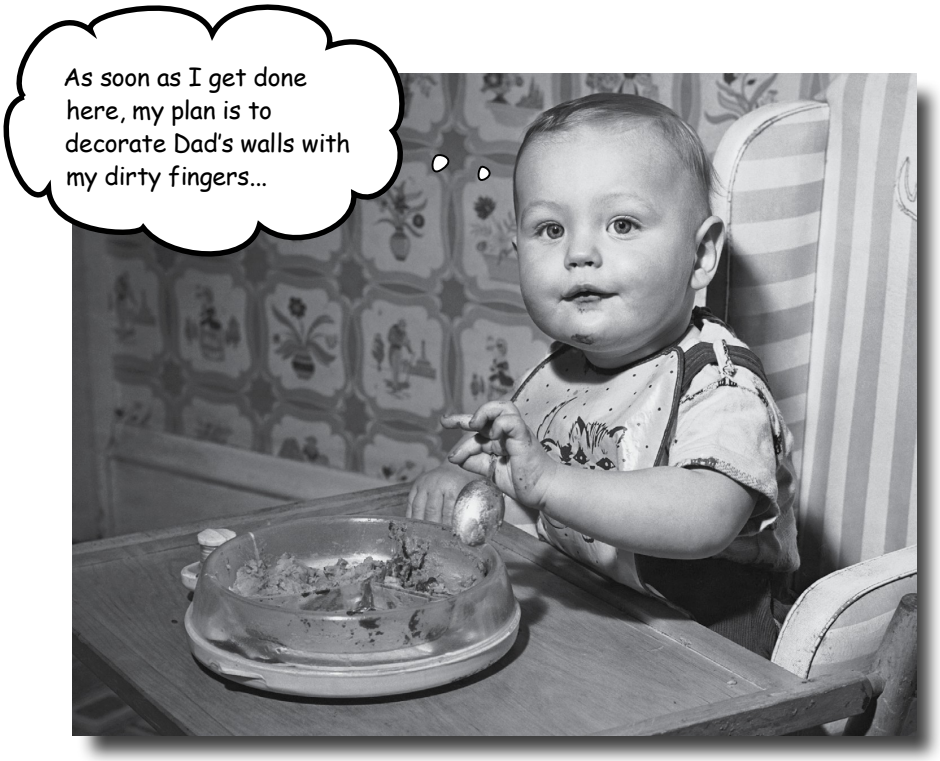
This is the second half of the webapp code in "vsearch4web.py".

```
@app.route('/search4', methods=['POST'])
def do_search() -> 'html':
    phrase = request.form['phrase']
    letters = request.form['letters']
    title = 'Here are your results:'
    results = str(search4letters(phrase, letters))
    log_request(request, results)
    return render_template('results.html',
                           the_title=title,
                           the_phrase=phrase,
                           the_letters=letters,
                           the_results=results,)

@app.route('/')
@app.route('/entry')
def entry_page() -> 'html':
    return render_template('entry.html',
                           the_title='Welcome to search4letters on the web!')


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


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

# 10 function decorators



# Wrapping Functions




**When it comes to augmenting your code, Chapter 9's context management protocol is not the only game in town.**

Python also lets you use function **decorators**, a technique whereby you can add code to an existing function *without* having to change any of the existing function's code. If you think this sounds like some sort of black art, don't despair: it's nothing of the sort. However, as coding techniques go, creating a function decorator is often considered to be on the harder side by many Python programmers, and thus is not used as often as it should be. In this chapter, our plan is to show you that, despite being an advanced technique, creating and using your own decorators is not that hard.

*pause for thought*

# Your Webapp Is Working Well, But...

You've shown the latest version of your webapp to a colleague, and they're impressed by what you've done. However, they pose an interesting question: *is it wise to let any web user view the log page?*

The point they're making is that anybody who is aware of the */viewlog* URL can use it to view the logged data whether they have your permission or not. In fact, at the moment, every one of your webapp's URLs are public, so any web user can access any of them.

Depending on what you're trying to do with your webapp, this may or may not be an issue. However, it is common for websites to require users to authenticate before certain content is made available to them. It's probably a good idea to be prudent when it comes to providing access to the */viewlog* URL. The question is: *how do you restrict access to certain pages in your webapp?*

## Only authenticated users gain access

You typically need to provide an **ID** and **password** when you access a website that serves restricted content. If your ID/password combination match, access is granted, as you've been authenticated. Once you're authenticated, the system knows to let you access the restricted content. Maintaining this state (whether authenticated or not) seems like it might be as simple as setting a switch to `True` (access allowed; you are logged in) or `False` (access forbidden; you are *not* logged in).

That sounds straightforward to me. A simple HTML form can ask for the user's credentials, and then a boolean on the server can be set to "True" or "False" as needed, right?

### It's a bit more complicated than that.

There's a twist here (due to the way the Web works) which makes this idea a tad more complicated than it at first appears. Let's explore what this complication is first (and see how to deal with it) before solving our restricted access issue.

# The Web Is Stateless

In its most basic form, a web server appears incredibly silly: each and every request that a web server processes is treated as an independent request, having nothing whatsoever to do with what came before, nor what comes after.

This means that sending three quick requests to a web server from your computer appears as three independent *individual* requests. This is in spite of the fact that the three requests originated from the same web browser running on the same computer, which is using the same unchanging IP address (which the web server sees as part of the request).

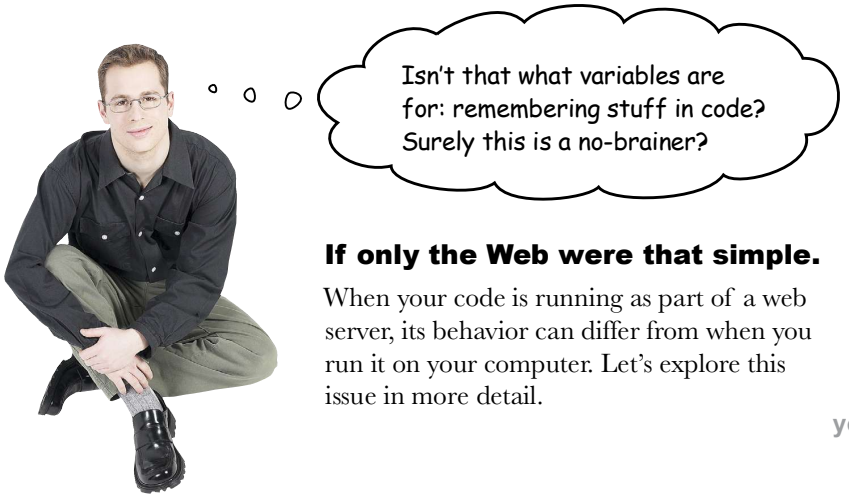
As stated at the top of the page: it's as if the web server is being silly. Even though we assume the three requests sent from our computer are related, the web server doesn't see things this way: *every web request is independent of what came before it, as well as what comes after.*


## HTTP is to blame...

The reason web servers behave in this way is due to the protocol that underpins the Web, and which is used by both the web server and your web browser: HTTP (the HyperText Transfer Protocol).

HTTP dictates that web servers must work as described above, and the reason for this has to do with performance: if the amount of work a web server needs to do is minimized, it's possible to scale web servers to handle many, many requests. Higher performance is achieved at the expense of requiring the web server to maintain information on how a series of requests may be related. This information—known as **state** in HTTP (and not related to OOP in any way)—is of no interest to the web server, as every request is treated as an independent entity. In a way, the web server is optimized to respond quickly, but forget fast, and is said to operate in a **stateless** manner.

Which is all well and good until such time as your webapp needs to remember something.




**If only the Web were that simple.**

When your code is running as part of a web server, its behavior can differ from when you run it on your computer. Let's explore this issue in more detail.

# Your Web Server (Not Your Computer) Runs Your Code

When Flask runs your webapp on your computer, it keeps your code in memory at all times. With this in mind, recall these two lines from the bottom of your webapp's code, which we initially discussed at the end of Chapter 5:

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

This line of code does NOT execute when this code is imported.

This `if` statement checks to see whether the interpreter is executing the code directly or whether the code is being imported (by the interpreter or by something like *PythonAnywhere*). When Flask executes on your computer, your webapp's code runs directly, resulting in this `app.run` line executing. However, when a web server is configured to execute your code your webapp's code is *imported*, and the `app.run` line does **not** run.

Why? Because the web server runs your webapp code *as it sees fit*. This can involve the web server importing your webapp's code, then calling its functions as needed, keeping your webapp's code in memory at all times. Or the web server may decide to load/unload your webapp code as needed, the assumption being that, during periods of inactivity, the web server will only load and run the code it needs. It's this second mode of operation—where the web server loads your code as and when it needs it—that can lead to problems with storing your webapp's state in variables. For instance, consider what would happen if you were to add this line of code to your webapp:

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

The "logged_in" variable could be used to indicate whether a user of your webapp is logged in or not.

The idea here is that other parts of your webapp can refer to the variable `logged_in` in order to determine whether a user is authenticated. Additionally, your code can change this variable's value as needed (based on, say, a successful login). As the `logged_in` variable is *global* in nature, all of your webapp's code can access and set its value. This seems like a reasonable approach, but has *two* problems.

Firstly, your web server can unload your webapp's running code at any time (and without warning), so any values associated with global variables are likely **lost**, and are going to be reset to their starting value when your code is next imported. If a previously loaded function sets `logged_in` to `True`, your reimported code helpfully resets `logged_in` to `False`, and confusion reigns...

Secondly, as it stands, there's only a *single copy* of the global `logged_in` variable in your running code, which is fine if all you ever plan to have is a single user of your webapp (good luck with that). If you have two or more users each accessing and/or changing the value of `logged_in`, not only will confusion reign, but frustration will make a guest appearance, too. As a general rule of thumb, storing your webapp's state in a global variable is a bad idea.

**Don't store your webapp's state in global variables.**

# It's Time for a Bit of a Session

As a result of what we learned on the last page, we need two things:

- A way to store variables without resorting to using globals
- A way to keep one webapp user's data from interferring with another's

Most webapp development frameworks (including Flask) provide for both of these requirements using a single technology: the **session**.

Think of a session as a layer of state spread on top of the stateless Web.

By adding a small piece of identification data to your browser (a *cookie*), and linking this to a small piece of identification data on the web server (the *session ID*), Flask uses its session technology to keep everything straight. Not only can you store state in your webapp that persists over time, but each user of your webapp gets their own copy of the state. Confusion and frustration are no more.

To demonstrate how Flask's session mechanism works, let's take a look at a very small webapp that is saved to a file called `quick_session.py`. Take a moment to read the code first, paying particular attention to the highlighted parts. We'll discuss what's going on after you've had a chance to read this code:

## Ready Bake Code

This is the "quick_session.py" code.

```
from flask import Flask, session

app = Flask(__name__)

app.secret_key = 'YouWillNeverGuess'

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


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


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

Be sure to add "session" to your list of imports.

Your secret key should be hard to guess.

Manipulate the data in "session" as required.

# Flask's Session Technology Adds State

In order to use Flask's session technology, you first have to import `session` from the `flask` module, which the `quick_session.py` webapp you just saw does on its very first line. Think of `session` as a global Python dictionary within which you store your webapp's state (albeit a dictionary with some added superpowers):

```
from flask import Flask, session
...
```

Start by importing "session".

Even though your webapp is still running on the stateless Web, this single import gives your webapp the ability to remember state.

Flask ensures that any data stored in `session` exists for the entire time your webapp runs (no matter how many times your web server loads and reloads your webapp code). Additionally, any data stored in `session` is keyed by a unique browser cookie, which ensures your session data is kept away from that of every other user of your webapp.

Just how Flask does all of this is not important: the fact that it does *is*. To enable all this extra goodness, you need to seed Flask's cookie generation technology with a "secret key," which is used by Flask to encrypt your cookie, protecting it from any prying eyes. Here's how `quick_session.py` does this:

**Find out more about Flask sessions here: http://flask.pocoo.org/docs/0.11/api/#sessions**

```
...
app = Flask(__name__)

app.secret_key = 'YouWillNeverGuess'
...
```

Create a new Flask webapp in the usual way.

Seed Flask's cookie-generation technology with a secret key. (Note: any string will do here. Although, like any other password you use, it should be hard to guess.)

Flask's documentation suggests picking a secret key that is hard to guess, but any stringed value works here. Flask uses the string to encrypt your cookie prior to transmitting it to your browser.

Once `session` is imported and the secret key set, you can use `session` in your code as you would any other Python dictionary. Within `quick_session.py`, the */setuser* URL (and its associated `setuser` function) assigns a user-supplied value to the `user` key in `session`, then returns the value to your browser:

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

The value of the "user" variable is assigned to the "user" key in the "session" dictionary.

The URL expects to be provided with a value to assign to the "user" variable (you'll see how this works in a little bit).

Now that we've set some session data, let's look at the code that accesses it.

# Dictionary Lookup Retrieves State

Now that a value is associated with the `user` key in `session`, it's not hard to access the data associated with `user` when you need it.

The second URL in the `quick_session.py` webapp, */getuser*, is associated with the `getuser` function. When invoked, this function accesses the value associated with the `user` key and returns it to the waiting web browser as part of the stringed message. The `getuser` function is shown below, together with this webapp's *dunder name equals dunder main* test (first discussed near the end of Chapter 5):

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

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

As is the custom with all Flask apps, we control when "app.run" executes using this well-established Python idiom.

Accessing the data in "session" is not hard. It's a dictionary lookup.

## Time for a Test Drive?

It's nearly time to take the `quick_session.py` webapp for a spin. However, before we do, let's think a bit about what it is we want to test.

For starters, we want to check that the webapp is storing and retrieving the session data provided to it. On top of that, we want to ensure that more than one user can interact with the webapp without stepping on any other user's toes: the session data from one user shouldn't impact the data of any other.

To perform these tests, we're going to simulate multiple users by running multiple browsers. Although the browsers are all running on one computer, as far as the web server is concerned, they are all independent, individual connections: the Web is stateless, after all. If we were to repeat these tests on three physically different computers on three different networks, the results would be the same, as all web servers see each request in isolation, no matter where the request originates. Recall that the `session` technology in Flask layers a stateful technology on top of the stateless Web.

To start this webapp, use this command within a terminal on *Linux* or *Mac OS X*:

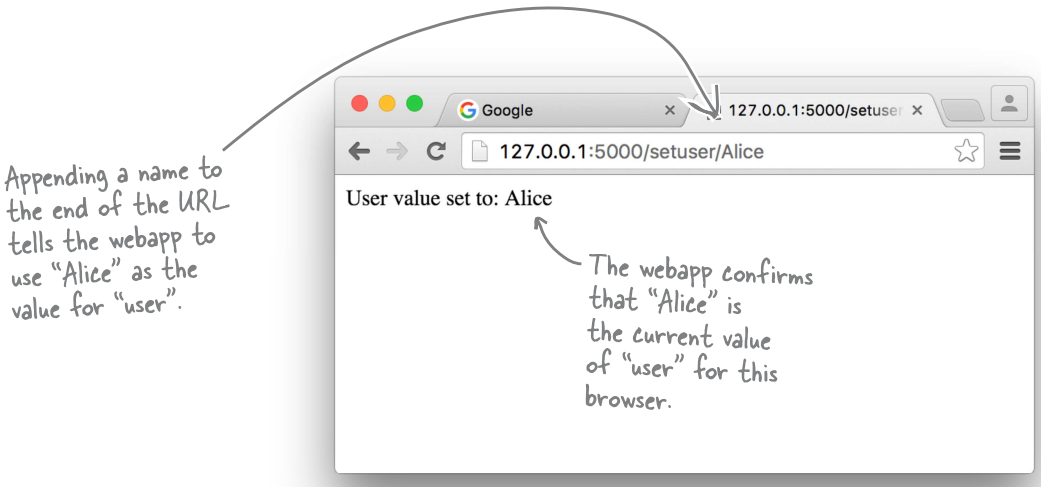
```
$ python3 quick_session.py
```

or use this command at a command prompt on *Windows*:

```
C:\> py -3 quick_session.py
```

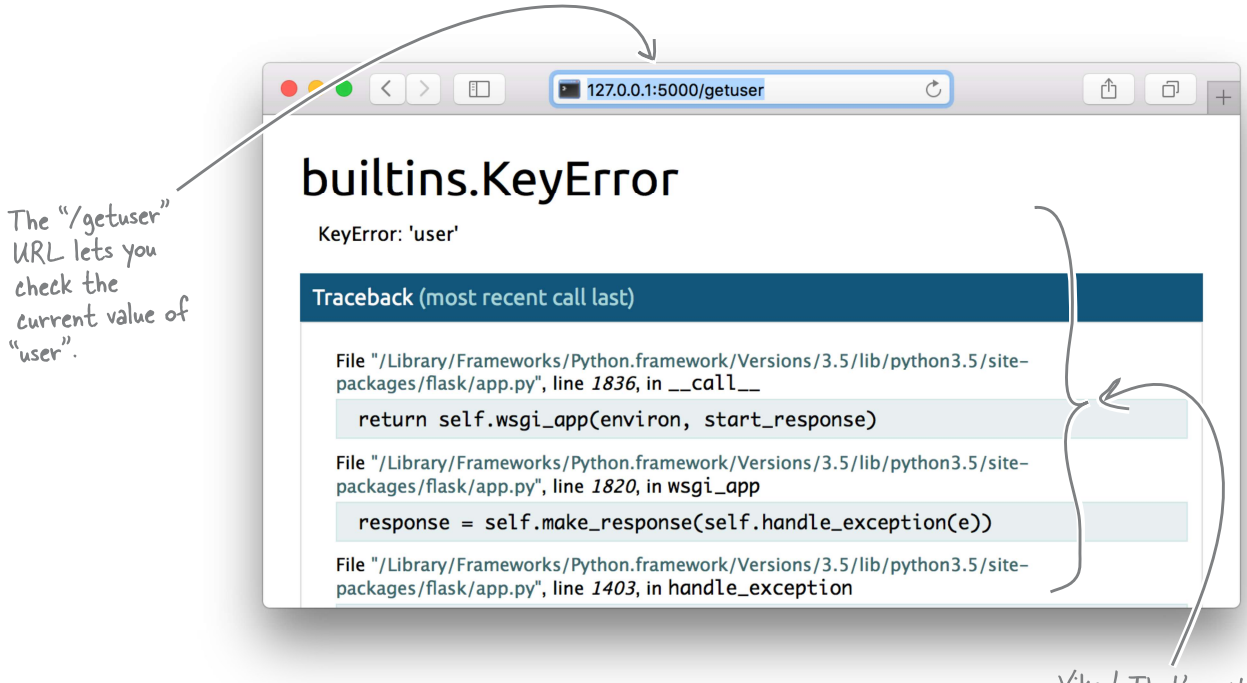
# Test Drive, 1 of 2

With the `quick_session.py` webapp up and running, let's open a Chrome browser and use it to set a value for the `user` key in `session`. We do this by typing ***`/setuser/Alice`*** into the location bar, which instructs the webapp to use the value `Alice` for `user`:




Next, let's open up the Opera browser and use it to set the value of `user` to `Bob` (if you don't have access to Opera, use whichever browser is handy, as long as it's not Chrome):

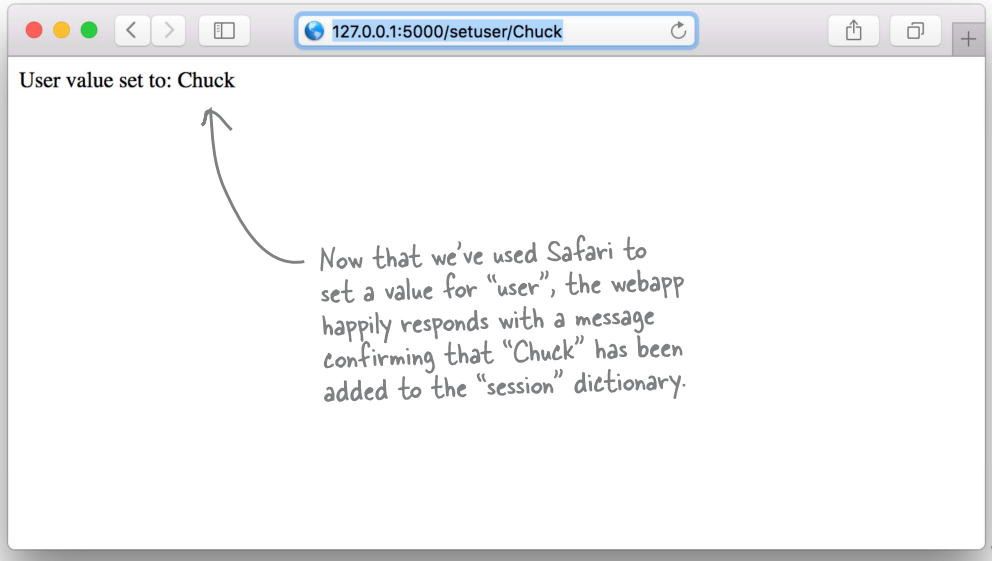
When we opened up Safari (or you can use Edge if you are on Windows), we used the webapp's other URL, */getuser*, to retrieve the current value of `user` from the webapp. However, when we did this, we're greeted with a rather intimidating error message:




The "/getuser" URL lets you check the current value of "user".

Yikes! That's quite the error message, isn't it? The important bit is at the top: we have a "KeyError", as we haven't used Safari to set a value for "user" yet. (Remember: we set a "user" value using Chrome and Opera, not Safari.)

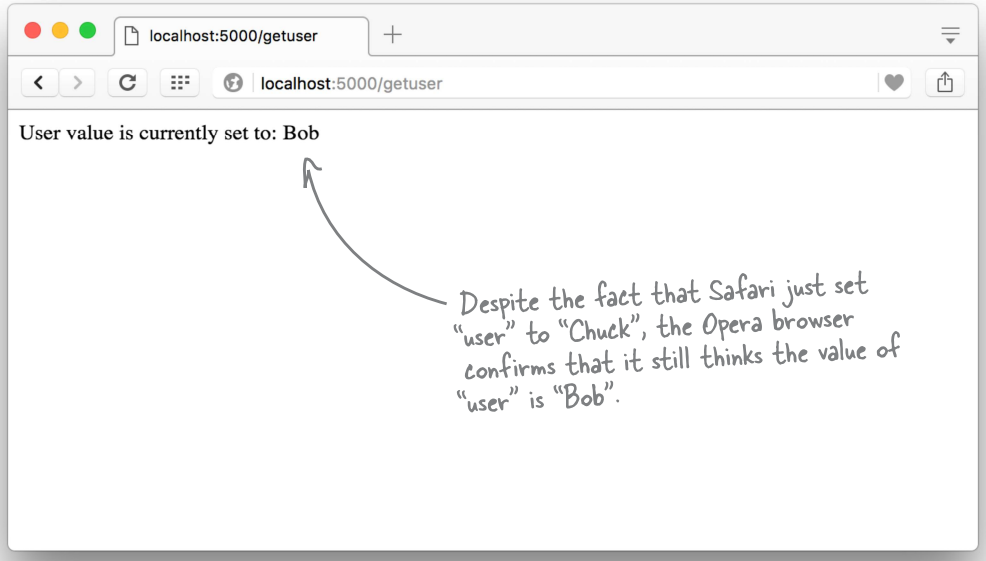
Let's use Safari to set the value of `user` to `Chuck`:



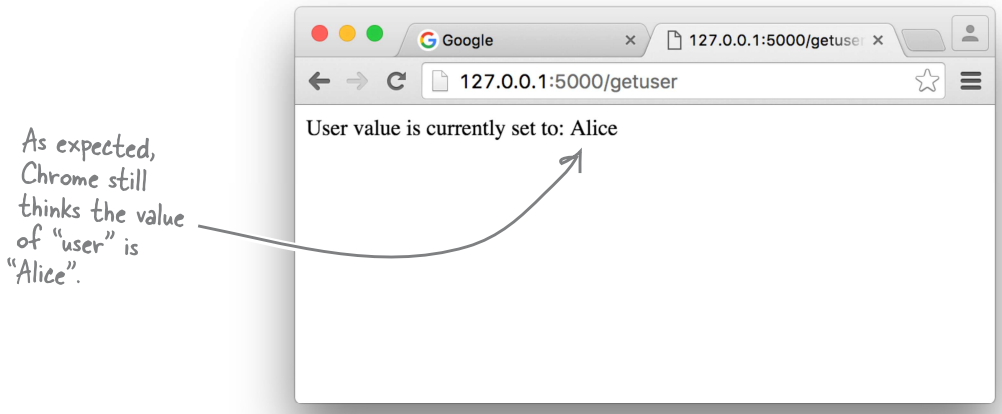

Now that we've used Safari to set a value for "user", the webapp happily responds with a message confirming that "Chuck" has been added to the "session" dictionary.

# Test Drive, 2 of 2

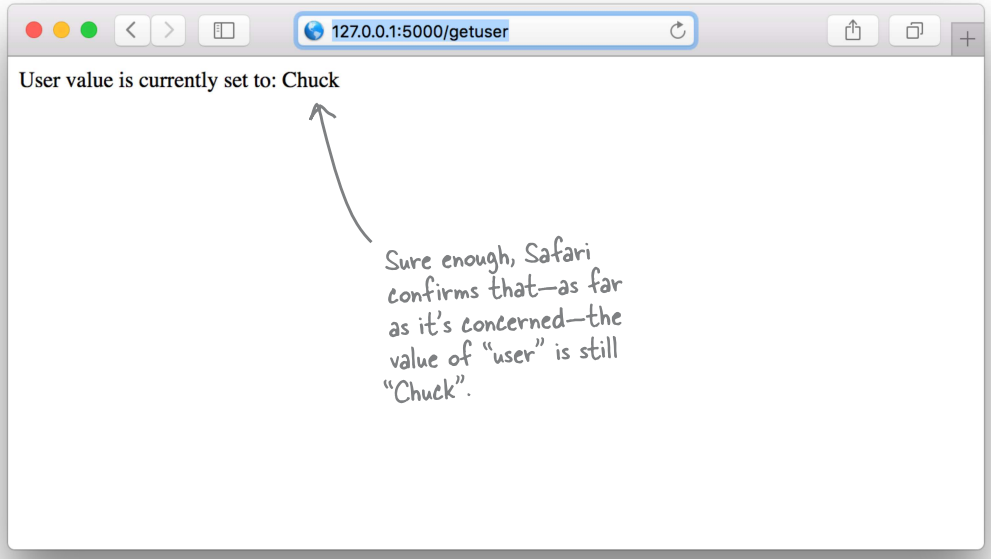
Now that we've used the three browsers to set values for `user`, let's confirm that the webapp (thanks to our use of `session`) is stopping each browser's value of `user` from interfering with any other browser's data. Even though we've just used Safari to set the value of `user` to `Chuck`, let's see what its value is in Opera by using the */getuser* URL:

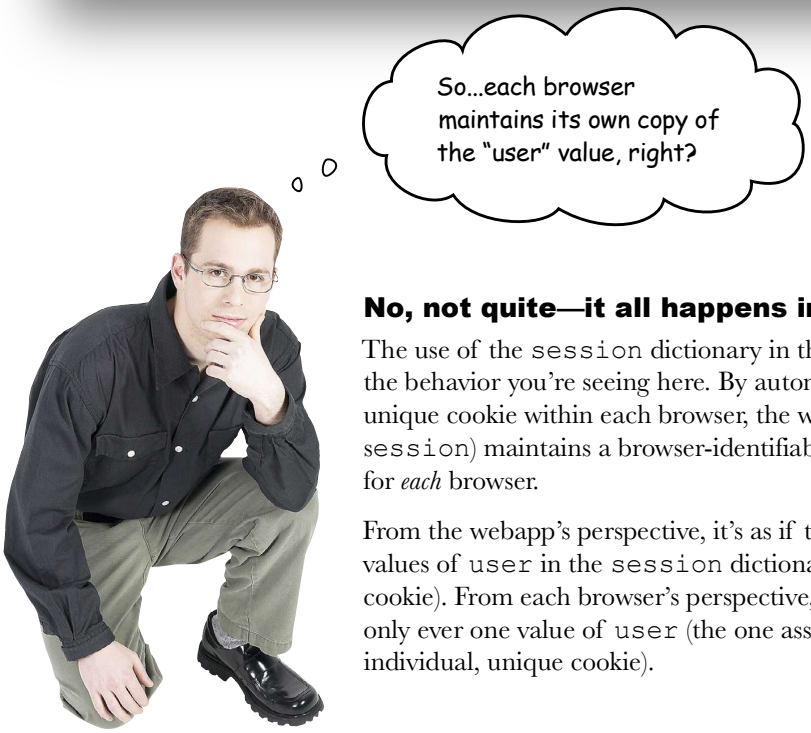



Having confirmed that Opera is showing `user`'s value as `Bob`, let's return to the Chrome browser window and issue the */getuser* URL there. As expected, Chrome confirms that, as far as it's concerned, the value of `user` is `Alice`:



We've just used Opera and Chrome to access the value of `user` using the */getuser* URL, which just leaves Safari. Here's what we see when we issue */getuser* in Safari, which doesn't produce an error message this time, as `user` has a value associated with it now (so, no more `KeyError`):




**No, not quite—it all happens in the webapp.**

The use of the `session` dictionary in the webapp enables the behavior you're seeing here. By automatically setting a unique cookie within each browser, the webapp (thanks to `session`) maintains a browser-identifiable value of `user` for *each* browser.

From the webapp's perspective, it's as if there are multiple values of `user` in the `session` dictionary (keyed by cookie). From each browser's perspective, it's as if there is only ever one value of `user` (the one associated with their individual, unique cookie).

# Managing Logins with Sessions

Based on our work with `quick_session.py`, we know we can store browser-specific state in `session`. No matter how many browsers interact with our webapp, each browser's server-side data (a.k.a. *state*) is managed for us by Flask whenever `session` is used.

Let's use this new know-how to return to the problem of controlling web access to specific pages within the `vsearch4web.py` webapp. Recall that we want to get to the point where we can restrict who has access to the */viewlog* URL.

Rather than experimenting on our working `vsearch4web.py` code, let's put that code to one side for now and work with some other code, which we'll experiment with in order to work out what we need to do. We'll return to the `vsearch4web.py` code once we've worked out the best way to approach this. We can then confidently amend the `vsearch4web.py` code to restrict access to */viewlog*.

Here's the code to yet another Flask-based webapp. As before, take some time to read this code prior to our discussion of it. This is `simple_webapp.py`:

## Ready Bake Code

```
from flask import Flask

app = Flask(__name__)

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

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

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

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

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

This is "simple_webapp.py". At this stage in this book, you should have no difficulty reading this code and understanding what this webapp does.

# Let's Do Login

The `simple_webapp.py` code is straightforward: all of the URLs are public in that they can be accessed by anyone using a browser.

In addition to the default / URL (which results in the `hello` function executing), there are three other URLs, */page1*, */page2*, and */page3* (which invoke similarly named functions when accessed). All of the webapp's URLs return a specific message to the browser.

As webapps go, this one is really just a shell, but will do for our purposes. We'd like to get to the point where */page1*, */page2*, and */page3* are only visible to logged-in users, but restricted to everyone else. We're going to use Flask's `session` technology to enable this functionality.

Let's begin by providing a really simple */login* URL. For now, we're not going to worry about providing an HTML form that asks for a login ID and password. All we're going to do here is create some code that adjusts `session` to indicate that a successful login has occurred.

## Sharpen your pencil

Let's write the code for the */login* URL below. In the space shown, provide code that adjusts `session` by setting a value for the `logged_in` key to `True`. Additionally, have the URL's function return the "You are now logged in" message to the waiting browser:

Add the new code here.

```
@app.route('/login')
def do_login() -> str:
    .....
    return .....
```

In addition to creating the code for the */login* URL, you'll need to make two other changes to the code to enable sessions. Detail what you think these changes are here:

1. .....
2. .....

## Sharpen your pencil Solution

You were to write the code for the */login* URL below. You were to provide code that adjusts `session` by setting a value for the `logged_in` key to `True`. Additionally, you were to have the URL's function return the "You are now logged in" message to the waiting browser:

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

Set the "logged_in" key in the "session" dictionary to "True".

Return this message to the waiting browser.

In addition to creating the code for the */login* URL, you needed to make two other changes to the code to enable sessions. You were to detail what you think these changes were:

1. We need to add 'session' to the import line at the top of the code.
2. We need to set a value for this webapp's secret key.

Let's not forget to do these.

## Amend the webapp's code to handle logins

We're going to hold off on testing this new code until we've added another two URLs: */logout* and */status*. Before you move on, make sure your copy of `simple_webapp.py` has been amended to include the changes shown below. Note: we're not showing all of the webapp's code here, just the new bits (which are highlighted):

```
from flask import Flask, session

app = Flask(__name__)

...

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

app.secret_key = 'YouWillNeverGuessMySecretKey'

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

Remember to import "session".

Add the code for the "/login" URL.

Set a value for this webapp's secret key (which enables the use of sessions).

# Let's Do Logout and Status Checking

Adding the code for the */logout* and */status* URLs is our next task.

When it comes to logging out, one strategy is to set the `session` dictionary's `logged_in` key to `False`. Another strategy is to *remove* the `logged_in` key from `session` altogether. We're going to go with the second option; the reason why will become clear after we code the */status* URL.

## Sharpen your pencil

Let's write the code for the */logout* URL, which needs to remove the `logged_in` key from the `session` dictionary, then return the "You are now logged out" message to the waiting browser. Add your code into the spaces below:

Add the logout code here.

```
@app.route('/logout')
def do_logout() -> str:
    ..........
    return ..........
```

Hint: if you've forgotten how to remove a key from a dictionary, type "dir(dict)" at the >>> prompt for a list of available dictionary methods.

With */logout* written, we now turn our attention to */status*, which returns one of two messages to the waiting web browser.

The message "You are currently logged in" is returned when `logged_in` exists as a value in the `session` dictionary (and, by definition, is set to `True`).

The message "You are NOT logged in" is returned when the `session` dictionary doesn't have a `logged_in` key. Note that we can't check `logged_in` for `False`, as the */logout* URL removes the key from the `session` dictionary as opposed to changing its value. (We haven't forgotten that we still need to explain why we're doing things this way, and we'll get to the explanation in a while. For now, trust that this is the way you have to code this functionality.)

Let's write the code for the */status* URL in the space below:

Check if the "logged_in" key exists in the "session" dictionary, then return the appropriate message.

Put your status-checking code here.

```
@app.route('/status')
def check_status() -> str:
    if ..........
        return ..........
    return ..........
```

## Sharpen your pencil Solution

You were to write the code for the */logout* URL, which needed to remove the `logged_in` key from the `session` dictionary, then return the "You are now logged out" message to the waiting browser:

Use the "pop" method to remove the "logged_in" key from the "session" dictionary.

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

With */logout* written, you were to turn your attention to the */status* URL, which returns one of two messages to the waiting web browser.

The message "You are currently logged in" is returned when `logged_in` exists as a value in the `session` dictionary (and, by definition, is set to `True`).

The message "You are NOT logged in" is returned when the `session` dictionary doesn't have a `logged_in` key.

You were to write the code for */status* in the space below:

Does the "logged_in" key exist in the "session" dictionary?

If yes, return this message.

If no, return this message.

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

## Amend the webapp's code once more

We're still holding off on testing this new version of the webapp, but here (on the right) is a highlighted version of the code you need to add to your copy of `simple_webapp.py`.

Make sure you've amended your code to match ours before getting to the next *Test Drive*, which is coming up right after we make good on an earlier promise.

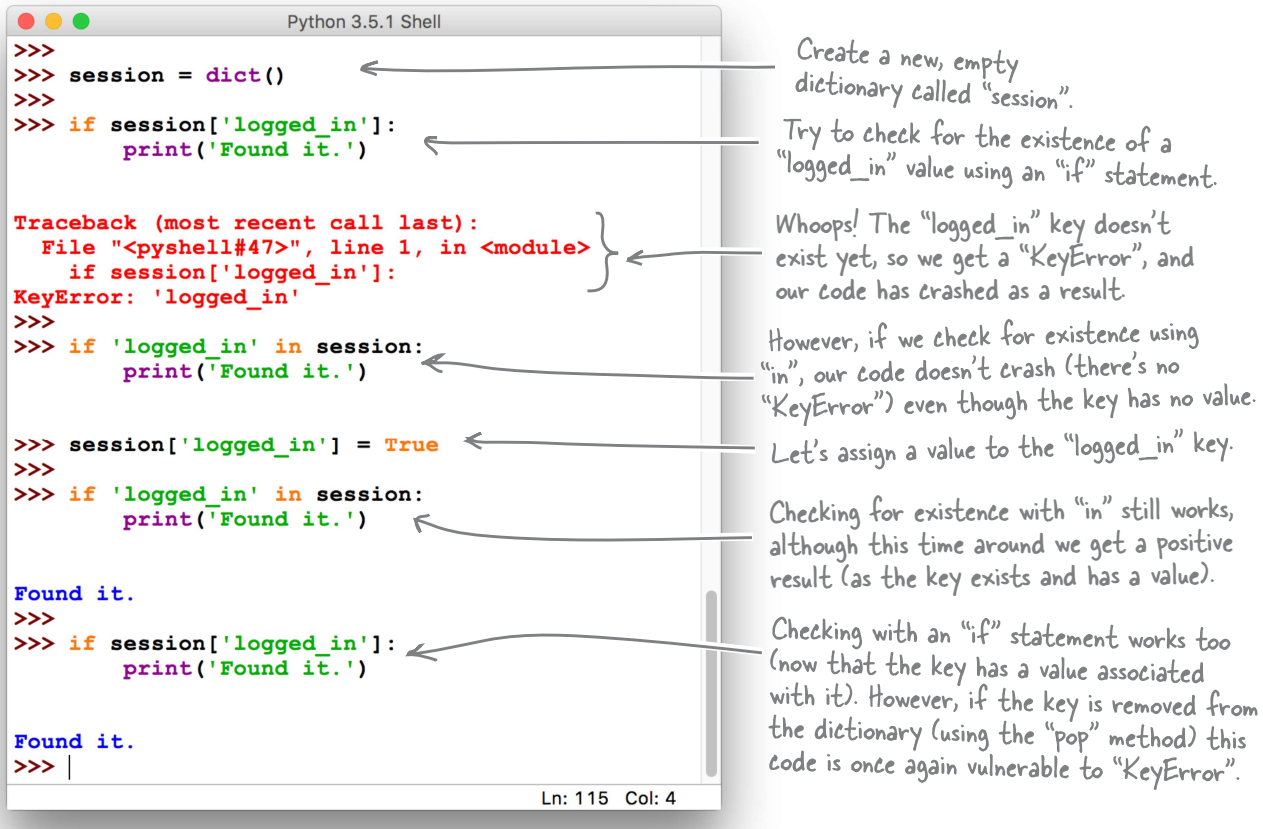
Two new URL routes

```
...

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

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

app.secret_key = 'YouWillNeverGuessMySecretKey'

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

# Why Not Check for False?

When you coded the */login* URL, you set the `logged_in` key to `True` in the `session` dictionary (which indicated that the browser was logged into the webapp). However, when you coded the */logout* URL, the code didn't set the value associated with the `logged_in` key to `False`, as we preferred instead to remove all trace of the `logged_in` key from the `session` dictionary. In the code that handled the */status* URL, we checked the "login status" by determining whether or not the `logged_in` key existed in the `session` dictionary; we didn't check whether `logged_in` is `False` (or `True`, for that matter). Which begs the question: *why does the webapp not use* `False` *to indicate "not logged in"?*

The answer is subtle, but important, and it has to do with the way dictionaries work in Python. To illustrate the issue, let's experiment at the >>> prompt and simulate what can happen to the `session` dictionary when used by the webapp. Be sure to follow along with this session, and carefully read each of the annotations:




The above experimentation shows that it is **not** possible to check a dictionary for a key's value until a key/value pairing exists. Trying to do so results in an `KeyError`. As it's a good idea to avoid errors like this, the `simple_webapp.py` code checks for the existence of the `logged_in` key as proof that the browser's logged in, as opposed to checking the key's actual value, thus avoiding the possibility of a `KeyError`.

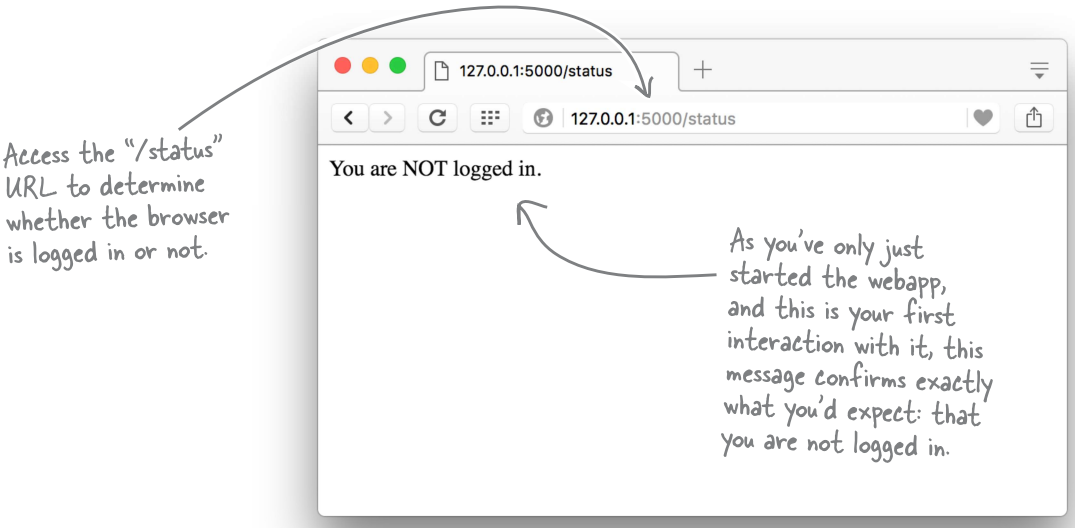
# Test Drive

Let's take the `simple_webapp.py` webapp for a spin to see how well the */login*, */logout*, and */status* URLs perform. As with the last *Test Drive*, we're going to test this webapp using more than one browser in order to confirm that each browser maintains its own "login state" on the server. Let's start the webapp from our operating system's terminal:

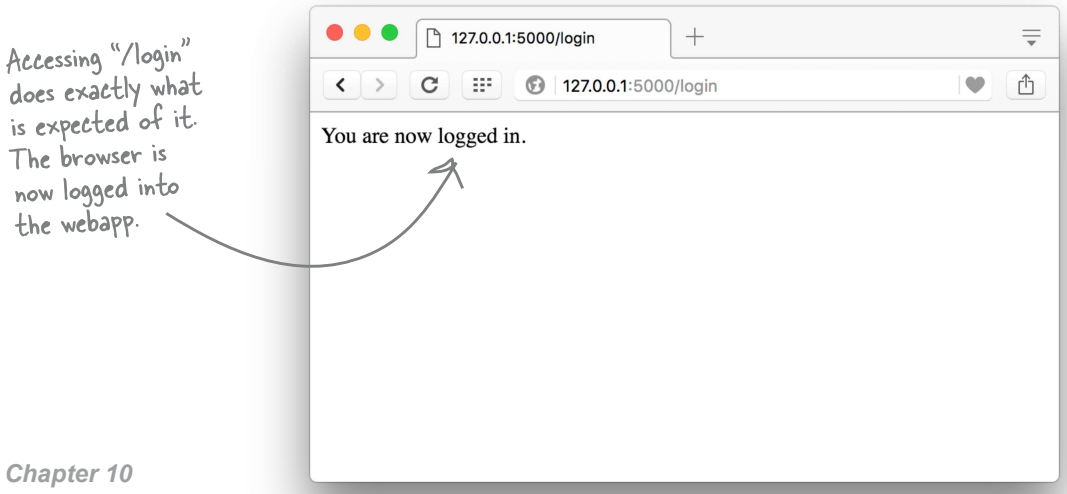
On Linux and Mac OS X: **`python3 simple_webapp.py`**

On Windows: **`py -3 simple_webapp.py`**

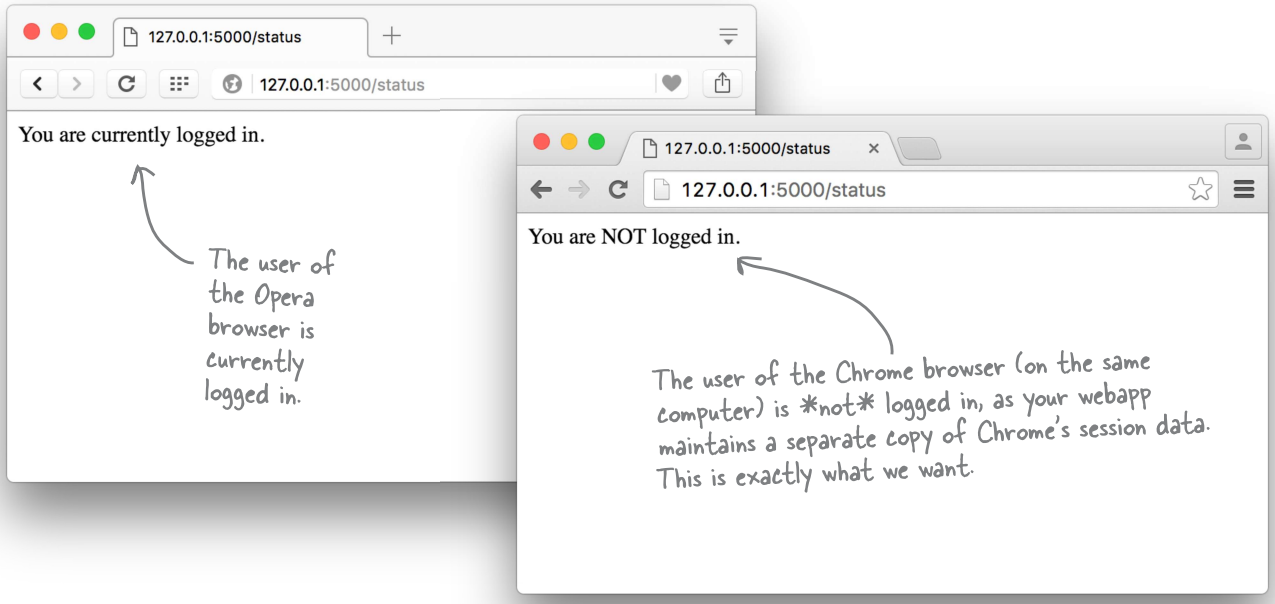
Let's fire up Opera and check its initial login status by accessing the */status* URL. As expected, the browser is not logged in:




Let's simulate logging in, by accessing the */login* URL. The message changes to confirm that the login was successful:



Now that you are logged in, let's confirm the status change by accessing the */status* URL within Opera. Doing so confirms that the user of the Opera browser is logged in. If you use Chrome to check the status, too, you'll see that the user of Chrome isn't logged in, which is exactly what we want (as each user of the webapp—each browser—has its own state maintained by the webapp):




To conclude, let's access the */logout* URL within Opera to tell the webapp that we are logging out of the session:


Although we haven't asked any of our browser's users for a login ID or password, the */login*, */logout*, and */status* URLs allow us to simulate what would happen to the webapp's `session` dictionary if we were to create the required HTML form, then hook up the form's data to a backend "credentials" database. The details of how this might happen are very much application-specific, but the basic mechanism (i.e., manipulating `session`) is the same no matter what a specific webapp might want to do.

Are we now ready to restrict access to the */page1*, */page2*, and */page3* URLs?

# Can We Now Restrict Access to URLs?


**Jim**: Hey, Frank...what are you stuck on?

**Frank**: I need to come up with a way to restrict access to the */page1*, */page2*, and */page3* URLs...

**Joe**: It can't be that hard, can it? You've already got the code you need in the function that handles */status*...

**Frank**: ...and it knows if a user's browser is logged in or not, right?

**Joe**: Yeah, it does. So, all you have to do is copy and paste that checking code from the function that handles */status* into each of the URLs you want to restrict, and then you're home and dry!

**Jim**: Oh, man! Copy and paste...the web developer's *Achilles' heel*. You really don't want to copy and paste code like that... it can only lead to problems down the road.

**Frank**: Of course! CS 101... I'll create a function with the code from */status*, then call *that* function as needed within the functions that handle the */page1*, */page2*, and */page3* URLs. Problem solved.

**Joe**: I like that idea...and I think it'll work. (I knew there was a reason we sat through all those *boring* CS lectures.)

**Jim**: Hang on...not so fast. What you're suggesting with a function is much better than your copy-and-paste idea, but I'm still not convinced it's the best way to go here.

**Frank** and **Joe** (together, and incredulously): *What's not to like?!?!?*

**Jim**: It bugs me that you're planning to add code to the functions that handle the */page1*, */page2*, and */page3* URLs that has nothing to do with what those functions actually *do*. Granted, you need to check whether a user is logged in before granting access, but adding a function call to do this to every URL doesn't sit quite right with me...

**Frank**: So what's your big idea, then?

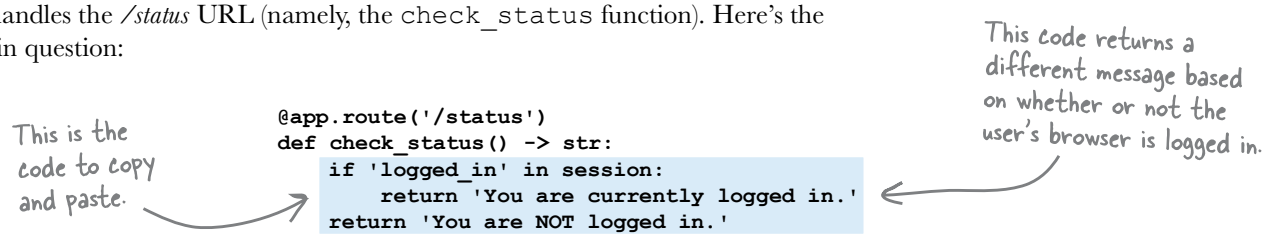
**Jim**: If it were me, I'd create, then use, a decorator.

**Joe**: Of course! That's an even better idea. Let's do that.

# Copy-and-Paste Is Rarely a Good Idea

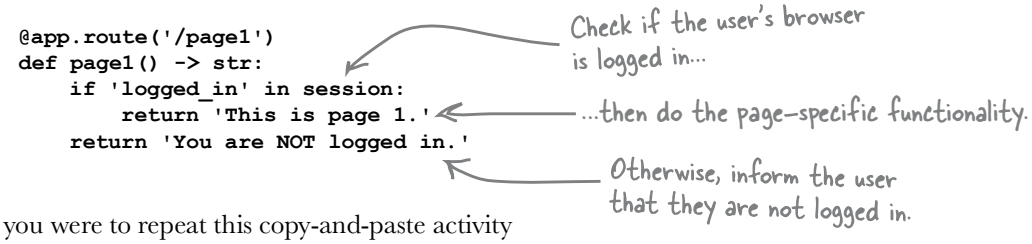
Let's convince ourselves that the ideas suggested on the last page are *not* the best way to approach the problem at hand—namely, how best to restrict access to specific web pages.

The first suggestion was to copy and paste some of the code from the function that handles the */status* URL (namely, the `check_status` function). Here's the code in question:




Here's what the `page1` function currently looks like:


If we copy and paste the highlighted code from `check_status` into `page1`, the latter's code would end up looking like this:




The above code works, but if you were to repeat this copy-and-paste activity for the */page2* and */page3* URLs (as well as any other URLs you were to add to your webapp), you'd quickly create a *maintenance nightmare*, especially when you consider all the edits you'd have to make should you decide to change how your login-checking code works (by, maybe, checking a submitted user ID and password against data stored in a database).

## Put shared code into its own function

When you have code that you need to use in many different places, the classic solution to the maintenance problem inherent in any copy-and-paste "quick fix" is to put the shared code into a function, which is then invoked as needed.

As such a strategy solves the maintenance problem (as the shared code exists in only one place as opposed to being copied and pasted willy-nilly), let's see what creating a login-checking function does for our webapp.

# Creating a Function Helps, But...

Let's create a new function called `check_logged_in`, which, when invoked, returns `True` if the user's browser is currently logged in, and `False` otherwise.

It's not a big job (most of the code is already in `check_status`); here's how we'd write this new function:

```
def check_logged_in() -> bool:
    if 'logged_in' in session:
        return True
    return False
```

Rather than returning a message, this code returns a boolean based on whether or not the user's browser is logged in.

With this function written, let's use it in the `page1` function instead of that copied and pasted code:

```
@app.route('/page1')
def page1() -> str:
    if not check_logged_in():
        return 'You are NOT logged in.'
    return 'This is page 1.'
```

We're checking if we are *not* logged in.

Call the "check_logged_in" function to determine the login status, then act accordingly.

This code only ever runs if the user's browser is logged in.

This strategy is a bit better than copy-and-paste, as you can now change how the login process works by making changes to the `check_logged_in` function. However, to use the `check_logged_in` function you still have to make similar changes to the `page2` and `page3` functions (as well as to any new URLs you create), and you do that by copying and pasting this new code from `page1` into the other functions... In fact, if you compare what you did to the `page1` function on this page with what you did to `page1` on the last page, it's roughly the same amount of work, and it's *still* copy-and-paste! Additionally, with *both* of these "solutions," the added code is **obscuring** what `page1` actually does.

It would be nice if you could somehow check if the user's browser is logged in *without* having to amend *any* of your existing function's code (so as not to obscure anything). That way, the code in each of your webapp's functions can remain *directly* related to what each function does, and the login status-checking code won't get in the way. If only there was a way to do this?

As we learned from our three friendly developers—Frank, Joe, and Jim—a few pages back, Python includes a language feature that can help here, and it goes by the name **decorator**. A decorator allows you to augment an existing function with extra code, and it does this by letting you change the behavior of the existing function *without* having to change its code.

If you're reading that last sentence and saying: *"What?!?!?"*, don't worry: it does sound strange the first time you hear it. After all, how can you possibly change how a function works without changing the function's code? Does it even make sense to try?

Let's find out by learning about decorators.

# You've Been Using Decorators All Along

You've been *using* decorators for as long as you've written webapps with Flask, which you started back in Chapter 5.

Here's the earliest version of the `hello_flask.py` webapp from that chapter, which highlights the use of a decorator called `@app.route`, which comes with Flask. The `@app.route` decorator is applied to an existing function (`hello` in this code), and the decorator augments the function it precedes by arranging to call `hello` whenever the webapp processes the / URL. Decorators are easy to spot; they're prefixed with the `@` symbol:

Here's the decorator, which—like all decorators—is prefixed with the @ symbol.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello() -> str:
    return 'Hello world from Flask!'

app.run()
```

Note how, as a user of the `@app.route` decorator, you have no idea how the decorator works its magic. All you're concerned with is that the decorator does what it promises: links a given URL with a function. All of the nitty-gritty, behind-the-scenes details of how the decorator works are hidden from you.

When you decide to create a decorator, you need to peek under the covers and (much like when you created a context manager in the last chapter) hook into Python's decorator machinery. There are four things that you need to know and understand to write a decorator:

1. **How to create a function**
2. **How to pass a function as an argument to a function**
3. **How to return a function from a function**
4. **How to process any number and type of function arguments**

You've been successfully creating and using your own functions since Chapter 4, which means this list of "four things to know" is really only three. Let's take some time to work through items 2 through 4 from this list as we progress toward writing a decorator of our own.

# Pass a Function to a Function

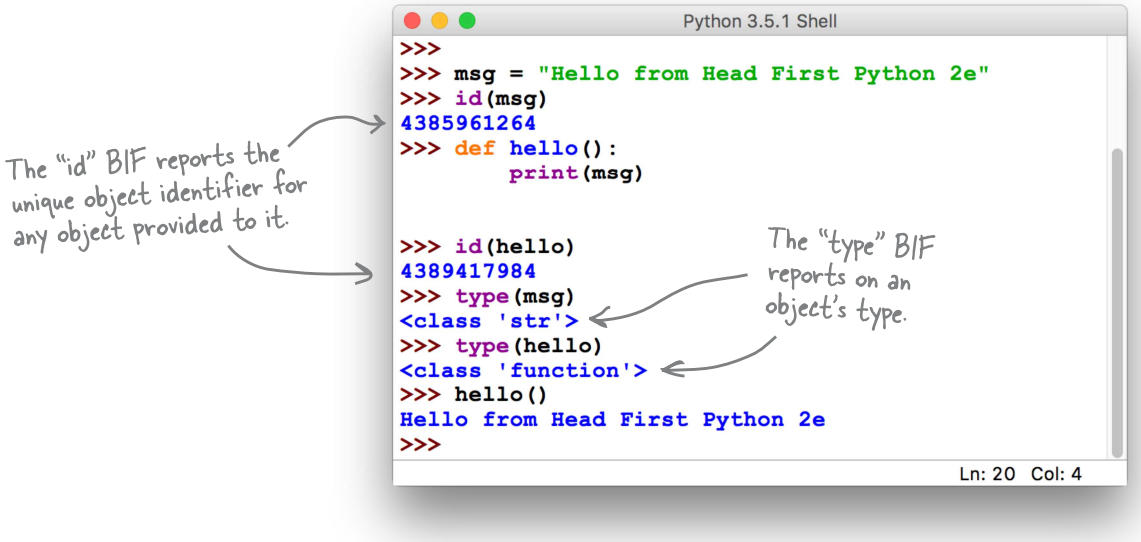
- [ ] Pass a function to a function.
- [ ] Return a function from a function.
- [ ] Process any number/type of arguments.

We'll check off each completed topic as we work through this material.

It's been a while, but way back in Chapter 2 we introduced the notion that *everything is an object* in Python. Although it may sound counterintuitive, the "everything" includes functions, which means functions are objects, too.

Clearly, when you invoke a function, it runs. However, like everything else in Python, functions are objects, and have an object ID: think of functions as "function objects."

Take a quick look at the short IDLE session below. A string is assigned to a variable called `msg`, and then its object ID is reported through a call to the `id` built-in function (BIF). A small function, called `hello`, is then defined. The `hello` function is then passed to the `id` BIF that reports the function's object ID. The `type` BIF then confirms that `msg` is a string and `hello` is a function, and finally `hello` is invoked and prints the current value of `msg` on screen:




We were a little devious in not drawing your attention to this before we had you look at the above IDLE session, but...did you notice *how* we passed `hello` to the `id` and `type` BIFs? We didn't invoke `hello`; we passed its *name* to each of the functions as an argument. In doing so, we passed a function to a function.

## Functions can take a function as an argument

The calls to `id` and `type` above demonstrate that some of Python's built-in functions accept a function as an argument (or to be more precise: *a function object*). What a function does with the argument is up to the function. Neither `id` nor `type` invokes the function, although it could have. Let's see how that works.

# Invoking a Passed Function

- ☐ Pass a function to a function.
- ☐ Return a function from a function.
- ☐ Process any number/type of arguments.

When a function object is passed as an argument to a function, the receiving function can *invoke* the passed-in function object.

Here's a small function (called `apply`) that takes two arguments: a function object and a value. The `apply` function invokes the function object and passes the value to the invoked function as an argument, returning the results of invoking the function on the value to the calling code:

The "apply" function accepts a function object as an argument. The "object" annotation helps to confirm our intention here (and the use of the argument name "func" is a common convention).

```
func.py - /Users/paul/Documents/func.py (3.5.1)

def apply(func: object, value: object) -> object:
    return func(value)

Ln: 6  Col: 0
```

Any value (of any type) can be passed as the second argument. Again, the annotations hint at what's allowed as an argument type here: any object.

The function (passed as an argument) is invoked, with the "value" passed to it as its only argument. The result of this function call is returned from the "apply" function.

Note how `apply`'s annotations hint that it accepts any function object together with any value, then returns anything (which is all very *generic*). A quick test of `apply` at the >>> prompt confirms that `apply` works as expected:

The "apply" function runs a bunch of BIFs against some values (and works as expected).

```
Python 3.5.1 Shell
>>>
>>> apply(print, 42)
42
>>> apply(id, 42)
4297539264
>>> apply(type, 42)
<class 'int'>
>>> apply(len, 'Marvin')
6
>>> apply(type, apply)
<class 'function'>
>>>

Ln: 110  Col: 4
```

The "apply" function takes any object for "value". In this example, it takes itself as "value" and confirms that it's a function.

In each of these examples, the first argument to "apply" is assigned to the "func" argument (above).

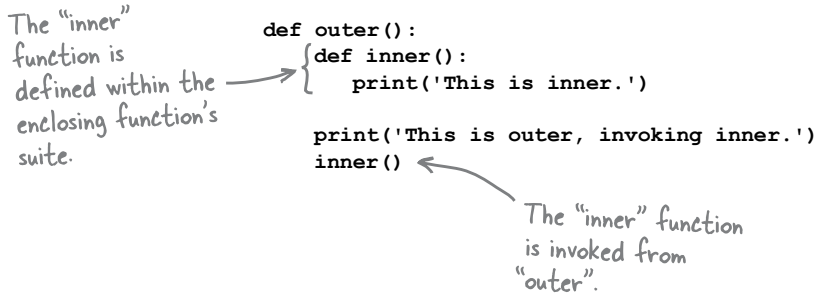
If you're reading this page and wondering when you'd ever need to do something like this, don't fret: we'll get to that when we write our decorator. For now, concentrate on understanding that it's possible to pass a function object to a function, which the latter can then invoke.

# Functions Can Be Nested Inside Functions

- [x] Pass a function to a function.
- [ ] Return a function from a function.
- [ ] Process any number/type of arguments.

Usually, when you create a function, you take some existing code and make it reusable by giving it a name, and using the existing code as the function's suite. This is the most common function use case. However, what sometimes comes as a surprise is that, in Python, the code in a function's suite can be *any* code, including code that defines another function (often referred to as a *nested* or *inner* function). Even more surprising is that the nested function can be *returned* from the outer, enclosing function; in effect, what gets returned is a *function object*. Let's look at a few examples that demonstrate these other, less common function use cases.

First up is an example that shows a function (called `inner`) nested inside another function (called `outer`). It is not possible to invoke `inner` from anywhere other than within `outer`'s suite, as `inner` is local in scope to `outer`:




When `outer` is invoked, it runs all the code in its suite: `inner` is defined, the call to the `print` BIF in `outer` is executed, and then the `inner` function is invoked (which calls the `print` BIF within `inner`). Here's what appears on screen:

```
This is outer, invoking inner.
This is inner.
```

The printed messages appear in the order: "outer" first, then "inner".

## When would you ever use this?

Looking at this simple example, you might find it hard to think of a situation where creating a function inside another function would be useful. However, when a function is complex and contains many lines of code, abstracting some of the function's code into a nested function often makes sense (and can make the enclosing function's code easier to read).

A more common usage of this technique arranges for the enclosing function to return the nested function as its value, using the `return` statement. This is what allows you to create a decorator.

So, let's see what happens when we return a function from a function.

# Return a Function from a Function

- [x] Pass a function to a function.
- [ ] Return a function from a function.
- [ ] Process any number/type of arguments.

Our second example is very similar to the first, but for the fact that the `outer` function no longer invokes `inner`, but instead returns it. Take a look at the code:

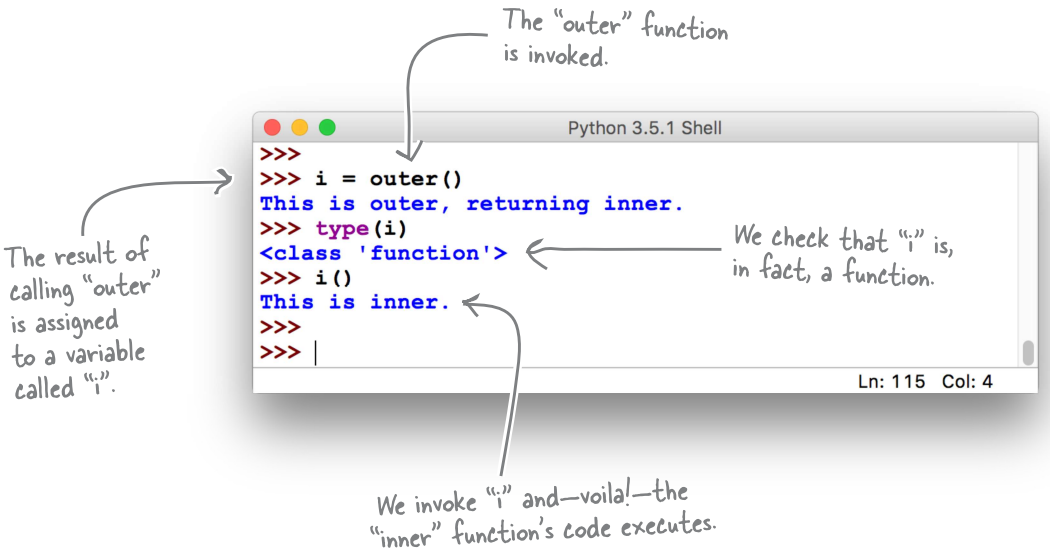
```
def outer():
    def inner():
        print('This is inner.')

    print('This is outer, returning inner.')
    return inner
```

The "inner" function is still defined within "outer".

The "return" statement does not invoke "inner"; instead, it returns the "inner" function object to the calling code.

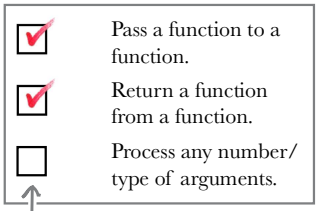
Let's see what this new version of the `outer` function does, by returning to the IDLE shell and taking `outer` for a spin.

Note how we assign the result of invoking `outer` to a variable, called `i` in this example. We then use `i` as if it were a function object—first checking its type by invoking the `type` BIF, then invoking `i` as we would any other function (by appending parentheses). When we invoke `i`, the `inner` function executes. In effect, `i` is now an *alias* for the `inner` function as created inside `outer`:




So far, so good. You can now *return* a function from a function, as well as *send* a function to a function. You're nearly ready to put all this together in your quest to create a decorator. There's just one more thing you need to understand: creating a function that can handle any number and type of arguments. Let's look at how to do this now.

# Accepting a List of Arguments




You're nearly there. One more topic to cover, and then you'll be ready to create a decorator.

Imagine you have a requirement to create a function (which we'll call `myfunc` in this example) that can be called with any number of arguments. For example, you might call `myfunc` like this:

```
myfunc(10)
```

One argument

or you might call `myfunc` like this:

```
myfunc()
```

No arguments

or you might call `myfunc` like this:

```
myfunc(10, 20, 30, 40, 50, 60, 70)
```

Many arguments (which, in this example, are all numbers, but could be anything: numbers, strings, booleans, list.

In fact, you might call `myfunc` with *any* number of arguments, with the proviso that you don't know ahead of time how many arguments are going to be provided.

As it isn't possible to define three distinct versions of `myfunc` to handle each of the three above invocations, the question becomes: *is it possible to accept any number of arguments in a function?*

## Use * to accept an arbitrary list of arguments

Python provides a special notation that allows you to specify that a function can take any number of arguments (where "any number" means "zero or more"). This notation uses the `*` character to represent *any number*, and is combined with an argument name (by convention, `args` is used) to specify that a function can accept an arbitrary list of arguments (even though `*args` is technically a tuple).

**Think of * as meaning "expand to a list of values."**

Here's a version of `myfunc` that uses this notation to accept any number of arguments when invoked. If any arguments are provided, `myfunc` prints their values to the screen:

The "*args" notation means "zero or more arguments."

myfunc.py - /Users/paul/Desktop/_NewBook/ch10/myfunc.py (3.5.1)

```
def myfunc(*args):
    for a in args:
        print(a, end=' ')
    if args:
        print()
```

Ln: 7 Col: 0

Think of "args" as a list of arguments, which can be processed like any other list (even though it's a tuple).

Arranges to display the list of argument values on a single line

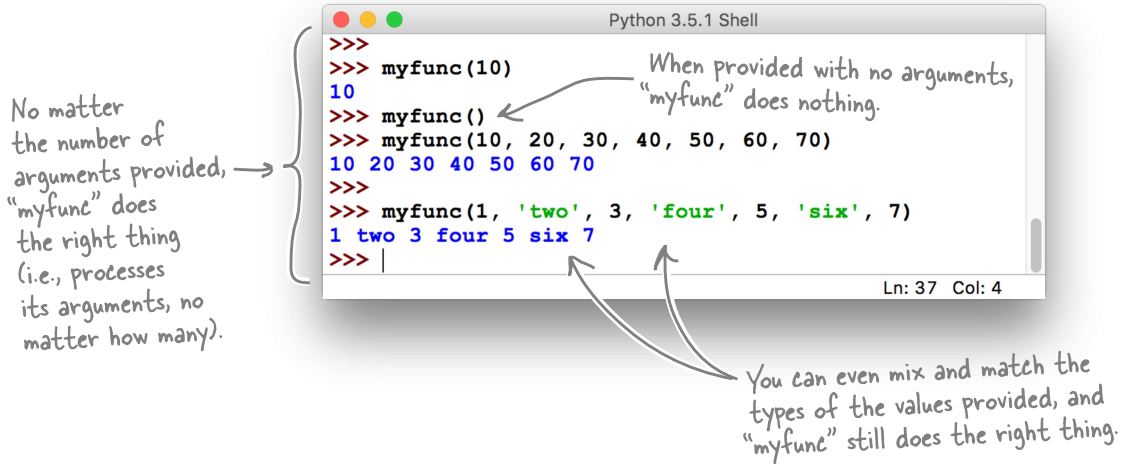
# Processing a List of Arguments

- [x] Pass a function to a function.
- [x] Return a function from a function.
- [ ] Process any number/type of arguments.

Now that `myfunc` exists, let's see if it can handle the example invocations from the last page, namely:

```
myfunc(10)
myfunc()
myfunc(10, 20, 30, 40, 50, 60, 70)
```

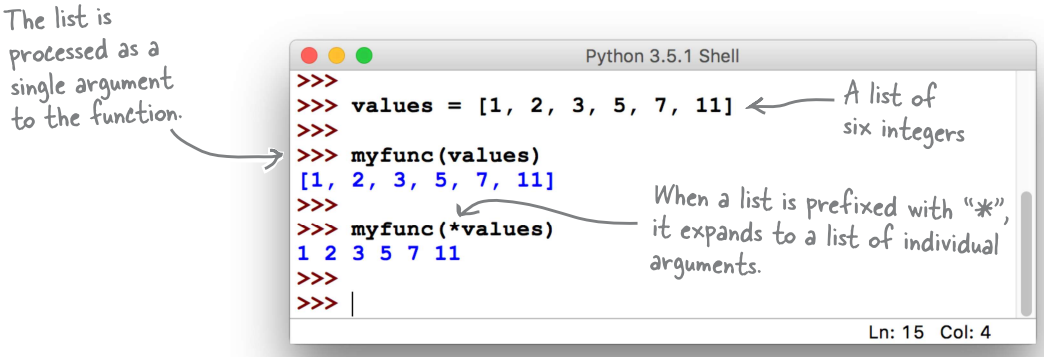
Here's another IDLE session that confirms that `myfunc` is up to the task. No matter how many arguments we supply (including *none*), `myfunc` processes them accordingly:




## * works on the way in, too

If you provide a list to `myfunc` as an argument, the list (despite potentially containing many values) is treated as one item (i.e., it's *one* list). To instruct the interpreter to **expand** the list to behave as if each of the list's items were an *individual* argument, prefix the list's name with the `*` character when invoking the function.

Another short IDLE session demonstrates the difference using `*` can have:

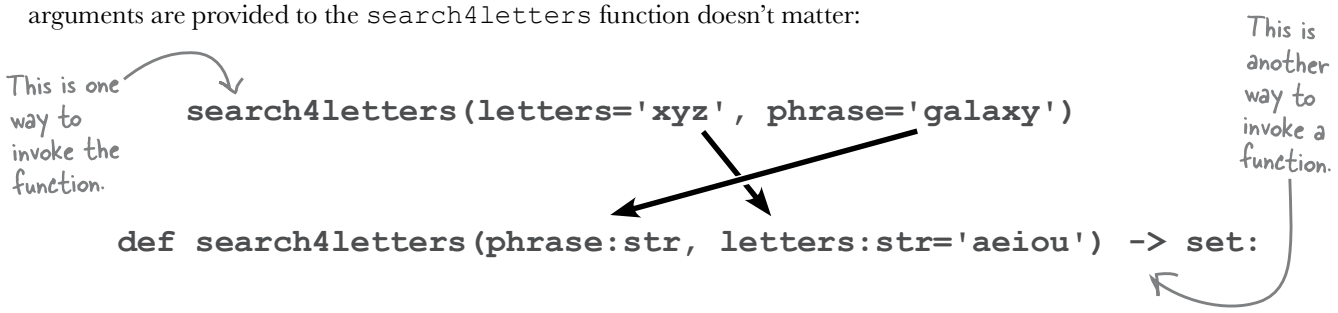


# Accepting a Dictionary of Arguments

- [x] Pass a function to a function.
- [x] Return a function from a function.
- [ ] Process any number/ type of arguments.

When it comes to sending values into functions, it's also possible to provide the names of the arguments together with their associated values, then rely on the interpreter to match things up accordingly.

You first saw this technique in Chapter 4 with the `search4letters` function, which—you may recall—expects two argument values, one for `phrase` and another for `letters`. When keyword arguments are used, the order in which the arguments are provided to the `search4letters` function doesn't matter:



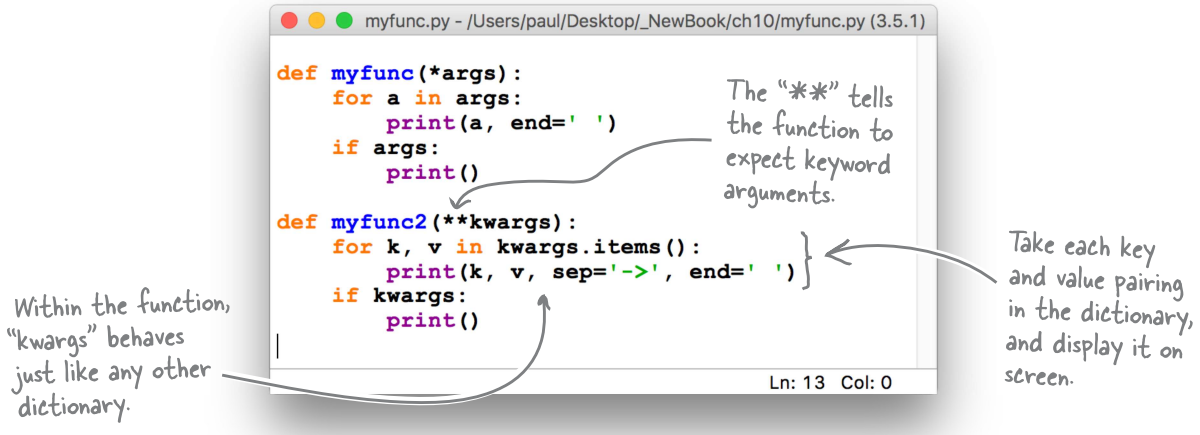

Like with lists, it's also possible arrange for a function to accept an arbitrary number of keyword arguments—that is, keys with values assigned to them (as with `phrase` and `letters` in the above example).

## Use ** to accept arbitrary keyword arguments

In addition to the `*` notation, Python also provides `**`, which expands to a collection of keyword arguments. Where `*` uses `args` as its variable name (by convention), `**` uses `kwargs`, which is short for "keyword arguments." (Note: you can use names other than `args` and `kwargs` within this context, but very few Python programmers do.)

**Think of ** as meaning "expand to a dictionary of keys and values."**

Let's look at another function, called `myfunc2`, which accepts any number of keyword arguments:

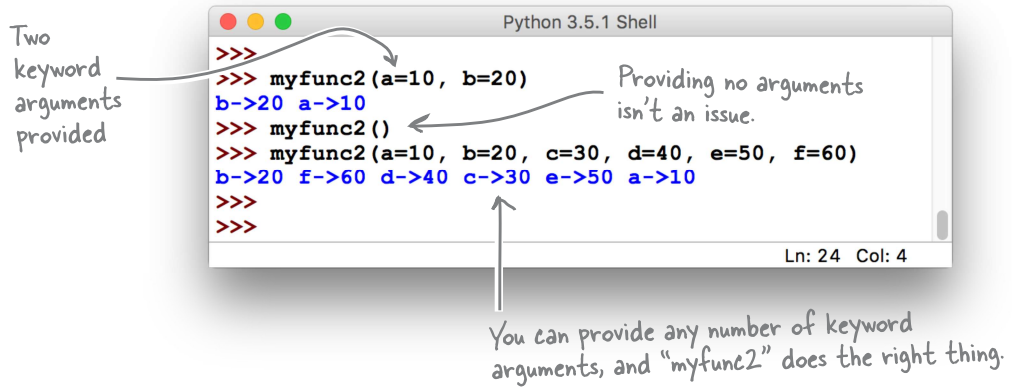


# Processing a Dictionary of Arguments

- [x] Pass a function to a function.
- [x] Return a function from a function.
- [ ] Process any number/type of arguments.

The code within `myfunc2`'s suite takes the dictionary of arguments and processes them, displaying all the key/value pairings on a single line.

Here's another IDLE session that demonstrates `myfunc2` in action. No matter how many key/value pairings are provided (including none), `myfunc2` does the right thing:




## ** works on the way in, too

You probably guessed this was coming, didn't you? As with `*args`, when you use `**kwargs` it's also possible to use `**` when invoking the `myfunc2` function. Rather than demonstrate how this works with `myfunc2`, we're going to remind you of a prior usage of this technique from earlier in this book. Back in Chapter 7, when you learned how to use Python's DB-API, you defined a dictionary of connection characteristics as follows:

A dictionary of key/value pairings

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

When it came time to establish a connection to your waiting MySQL (or MariaDB) database server, you used the `dbconfig` dictionary as follows. Notice anything about the way the `dbconfig` argument is specified?

Does this look familiar?

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

By prefixing the `dbconfig` argument with `**`, we tell the interpreter to treat the single dictionary as a collection of keys and their associated values. In effect, it's as if you invoked `connect` with four individual keyword arguments, like this:

```
conn = mysql.connector.connect('host'='127.0.0.1', 'user'='vsearch',
                               'password'='vsearchpasswd', 'database'='vsearchlogDB')
```

# Accepting Any Number and Type of Function Arguments

- [x] Pass a function to a function.
- [x] Return a function from a function.
- [ ] Process any number/type of arguments.

When creating your own functions, it's neat that Python lets you accept a list of arguments (using `*`), in addition to any number of keyword arguments (using `**`). What's even neater is that you can combine the two techniques, which lets you create a function that can accept any number and type of arguments.

Here's a third version of `myfunc` (which goes by the shockingly imaginative name of `myfunc3`). This function accepts any list of arguments, any number of keyword arguments, or a combination of both:

The original "myfunc" works with any list of arguments.

The "myfunc2" function works with any amount of key/value pairs.

The "myfunc3" function works with any input, whether a list of arguments, a bunch of key/value pairs, or both.

Both "*args" and "**kwargs" appear on the "def" line.

myfunc.py - /Users/paul/Desktop/_NewBook/ch10/myfunc.py (3.5.1)

```
def myfunc(*args):
    for a in args:
        print(a, end=' ')
    if args:
        print()

def myfunc2(**kwargs):
    for k, v in kwargs.items():
        print(k, v, sep='->', end=' ')
    if kwargs:
        print()

def myfunc3(*args, **kwargs):
    if args:
        for a in args:
            print(a, end=' ')
        print()
    if kwargs:
        for k, v in kwargs.items():
            print(k, v, sep='->', end=' ')
        print()
```

Ln: 7 Col: 0

This short IDLE session showcases `myfunc3`:

Works with no arguments

Works with a list

Works with keyword arguments

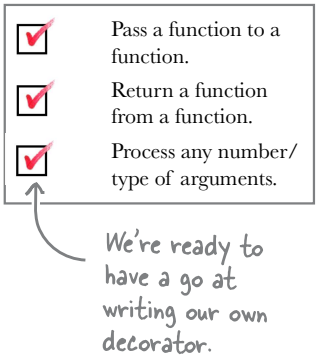
Works with a combination of a list and keyword arguments

Python 3.5.1 Shell

```
>>>
>>> myfunc3()
>>> myfunc3(1, 2, 3)
1 2 3
>>> myfunc3(a=10, b=20, c=30)
a->10 b->20 c->30
>>> myfunc3(1, 2, 3, a=10, b=20, c=30)
1 2 3
a->10 b->20 c->30
>>>
```

Ln: 68 Col: 4

# A Recipe for Creating a Function Decorator




With three items marked in the checklist on the right, you now have an understanding of the Python language features that allow you to create a decorator. All you need to know now is how you take these features and combine them to create the decorator you need.

Just like when you created your own context manager (in the last chapter), creating a decorator conforms to a set of rules or *recipe*. Recall that a decorator allows you to augment an existing function with extra code, without requiring you to change the existing function's code (which, we'll admit, still sounds freaky).

To create a function decorator, you need to know that:

1. **A decorator is a function**
   In fact, as far as the interpreter is concerned, your decorator is *just another function*, albeit one that manipulates an existing function. Let's refer to this existing function as *the decorated function* from here on in. Having made it this far in this book, you know that creating a function is easy: use Python's `def` keyword.

2. **A decorator takes the decorated function as an argument**
   A decorator needs to accept the decorated function as an argument. To do this, you simply pass the decorated function as a *function object* to your decorator. Now that you've worked through the last 10 pages, you know that this too is easy: you arrive at a function object by referring to the function *without* parentheses (i.e., using just the function's name).

3. **A decorator returns a new function**
   A decorator returns a new function as its return value. Much like when `outer` returned `inner` (a few pages back), your decorator is going to do something similar, except that the function it returns needs to *invoke* the decorated function. Doing this is—*dare we say it?*—easy but for one small complication, which is what Step 4 is all about.

4. **A decorator maintains the decorated function's signature**
   A decorator needs to ensure that the function it returns takes the same number and type of arguments as expected by the decorated function. The number and type of any function's arguments is known as its **signature** (as each function's `def` line is unique).

It's time to grab a pencil and put this information to work creating your first decorator.

# Recap: We Need to Restrict Access to Certain URLs


We've been working with the `simple_webapp.py` code, and we need our decorator to check to see whether the user's browser is logged in or not. If it is logged in, restricted web pages are visible. If the browser isn't logged in, the webapp should advise the user to log in prior to viewing any restricted pages. We'll create a decorator to handle this logic. Recall the `check_status` function, which demonstrates the logic we want our decorator to mimic:

We want to avoid copying and pasting this code.

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

Remember: this code returns a different message based on whether or not the user's browser is logged in.

# Creating a Function Decorator

To comply with item 1 in our list, you had to create a new function. Remember:

1. **A decorator is a function**
   In fact, as far as the interpreter is concerned, your decorator is *just another function*, albeit one that manipulates an existing function. Let's refer to this existing function as *the decorated function* from here on in. You know that creating a function is easy: use Python's `def` keyword.

Complying with item 2 involves ensuring your decorator accepts a function object as an argument. Again, remember:

2. **A decorator takes the decorated function as an argument**
   Your decorator needs to accept the decorated function as an argument. To do this, you simply pass the decorated function as a *function object* to your decorator. You arrive at a function object by referring to the function *without* parentheses (i.e., using the function's name).

## Sharpen your pencil

Let's put your decorator in its own module (so that you can more easily reuse it). Begin by creating a new file called `checker.py` in your text editor.

You're going to create a new decorator in `checker.py` called `check_logged_in`. In the space below, provide your decorator's `def` line. Hint: use `func` as the name of your function object argument:

Put the decorator's "def" line here.

.....

## there are no Dumb Questions

**Q: Does it matter where on my system I create `checker.py`?**

A: Yes. Our plan is to import `checker.py` into webapps that need it, so you need to ensure that the interpreter can find it when your code includes the `import checker` line. For now, put `checker.py` in the same folder as `simple_webapp.py`.

## Sharpen your pencil Solution

We decided to put your decorator in its own module (so that you can more easily reuse it).

You began by creating a new file called `checker.py` in your text editor.

Your new decorator (in `checker.py`) is called `check_logged_in` and, in the space below, you were to provide your decorator's `def` line:

```
def check_logged_in(func):
```

The "check_logged_in" decorator takes a single argument: the function object of the decorated function.

## That's almost too easy, isn't it?

Remember: a decorator is *just another function,* which takes a function object as an argument (`func` in the above `def` line).

Let's move on to the next item in our "create a decorator" recipe, which is a little more involved (but not by much). Recall what you need your decorator to do:

3 **A decorator returns a new function**
Your decorator returns a new function as its return value. Just like when `outer` returned `inner` (a few pages back), your decorator is going to do something similar, except that the function it returns needs to *invoke* the decorated function.

Earlier in this chapter, you met the `outer` function, which, when invoked, returned the `inner` function. Here's `outer`'s code once more:

```
def outer():
    def inner():
        print('This is inner.')

    print('This is outer, returning inner.')
    return inner
```

All of this code is in the "outer" function's suite.

The "inner" function is nested inside "outer".

The "inner" function object is returned as the result of invoking "outer". Note the lack of parentheses after "inner", as we're returning a function object. We are *not* invoking "inner".

## Sharpen your pencil

Now that you've written your decorator's `def` line, let's add some code to its suite. You need to do four things here.

1. Define a nested function called `wrapper` that is returned by `check_logged_in`. (You could use any function name here, but, as you'll see in a bit, `wrapper` is a pretty good choice.)

2. Within `wrapper`, add some of the code from your existing `check_status` function that implements one of two behaviors based on whether the user's browser is logged in or not. To save you the page-flip, here's the `check_status` code once more (with the important bits highlighted):

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

3. As per item 3 of our decorator-creating recipe, you need to adjust the nested function's code so that it invokes the decorated function (as opposed to returning the "You are currently logged in" message).

4. With the nested function written, you need to return its function object from `check_logged_in`.

Add the required code to `check_logged_in`'s suite in the spaces provided below:

```
def check_logged_in(func):
```

1. Define your nested function.

.....

.....

.....

.....

.....

2 and 3. Add the code you want the nested function to execute.

4. Don't forget to return the nested function.

## Sharpen your pencil Solution

With your decorator's `def` line written, you were to add some code to its suite. You needed to do four things:

1. Define a nested function called `wrapper` that is returned by `check_logged_in`.

2. Within `wrapper`, add some of the code from your existing `check_status` function that implements one of two behaviors based on whether the user's browser is logged in or not.

3. As per item 3 of our decorator-creating recipe, adjust the nested function's code so that it invokes the decorated function (as opposed to returning the "You are currently logged in" message).

4. With the nested function written, return its function object from `check_logged_in`.

You were to add the required code to `check_logged_in`'s suite in the spaces provided:

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

A nested "def" line starts the "wrapper" function.

If the user's browser is logged in...

...invoke the decorated function.

If the user's browser isn't logged in, return an appropriate message.

Did you remember to return the nested function?

## Can you see why the nested function is called "wrapper"?

If you take a moment to study the decorator's code (so far), you'll see that the nested function not only invokes the decorated function (stored in `func`), but also augments it by *wrapping* extra code around the call. In this case, the extra code is checking to see if the `logged_in` key exists within your webapp's `session`. Critically, if the user's browser is *not* logged in, the decorated function is *never* invoked by `wrapper`.