Chapter 5. Building a Webapp: Getting Real

At this stage, you know enough Python to be dangerous.

With this book's first four chapters behind you, you're now in a position to productively use Python within any number of application areas (even though there's still lots of Python to learn). Rather than explore the long list of what these application areas are, in this and subsequent chapters, we're going to

structure our learning around the development of a web-hosted application, which is an area where Python is especially strong. Along the way, you'll learn a bit more about Python. Before we get going, however, let's have a quick recap of the Python you already know.

Python: What You Already Know

Now that you've got four chapters under your belt, let's pause for a moment and review the Python material presented so far.

BULLET POINTS

- IDLE, Python's built-in IDE, is used to experiment with and execute Python code, either as single-statement snippets or as larger multistatement programs written within IDLE's text editor. As well as using IDLE, you ran a file of Python code directly from your operating system's command line, using the `py -3` command (on Windows) or `python3` (on everything else).
- You've learned how Python supports single-value data items, such as integers and strings, as well as the booleans `True` and `False`.
- You've explored use cases for the four built-in data structures: lists, dictionaries, sets, and tuples. You know that you can create complex data structures by combining these four built-ins in any number of ways.
- You've used a collection of Python statements, including `if`, `elif`, `else`, `return`, `for`, `from`, and `import`.
- You know that Python provides a rich standard library, and you've seen the following modules in action: `datetime`, `random`, `sys`, `os`, `time`, `html`, `pprint`, `setuptools`, and `pip`.
- As well as the standard library, Python comes with a handy collection of built-in functions, known as the BIFs. Here are some of the BIFs you've worked with: `print`, `dir`, `help`, `range`, `list`, `len`, `input`, `sorted`, `dict`, `set`, `tuple`, and `type`.
- Python supports all the usual operators, and then some. Those you've already seen include: `in`, `not in`, `+`, `-`, `=` (assignment), `==` (equality), `+=`, and `*`.
- As well as supporting the square bracket notation for working with items in a sequence (i.e., `[]`), Python extends the notation to support **slices**, which allow you to specify **start**, **stop**, and **step** values.

- You've learned how to create your own custom functions in Python, using the `def` statement. Python functions can optionally accept any number of arguments as well as return a value.
- Although it's possible to enclose strings in either single or double quotes, the Python conventions (documented in **PEP 8**) suggest picking one style and sticking to it. For this book, we've decided to enclose all of our strings within single quotes, unless the string we're quoting itself contains a single quote character, in which case we'll use double quotes (as a one-off, special case).
- Triple-quoted strings are also supported, and you've seen how they are used to add docstrings to your custom functions.
- You learned that you can group related functions into modules. Modules form the basis of the code reuse mechanism in Python, and you've seen how the `pip` module (included in the standard library) lets you consistently manage your module installations.
- Speaking of things working in a consistent manner, you learned that in Python **everything is an object**, which ensures—as much as possible—that everything works just as you expect it to. This concept really pays off when you start to define your own custom objects using classes, which we'll show you how to do in a later chapter.

Let's Build Something

Let's build a webapp.

Specifically, let's take our `search4letters` function and make it accessible over the Web, enabling anyone with a web browser to access the service provided by our function.

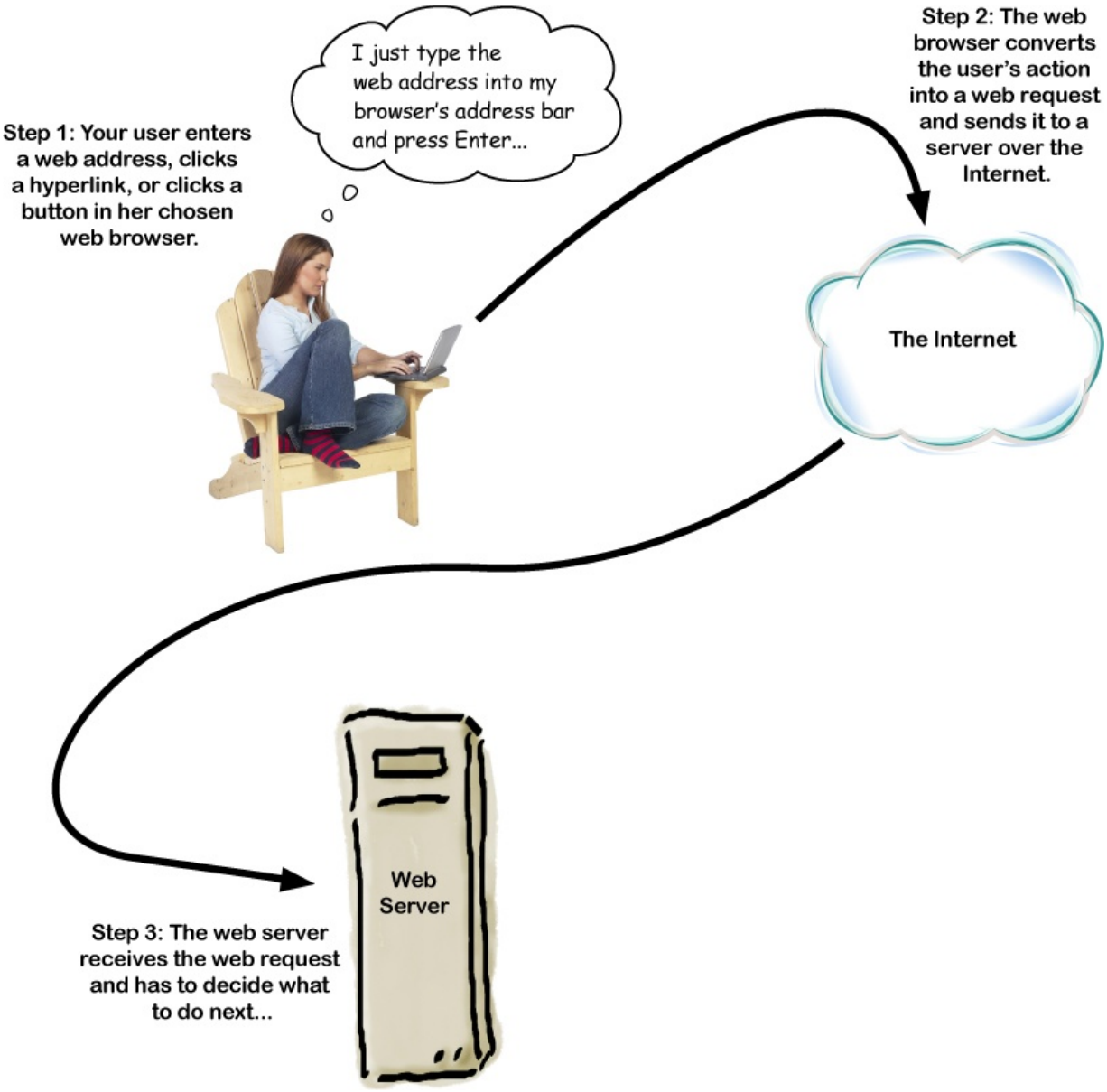
We could build any type of application, but building a working web application lets us explore a number of Python features while building something that's generally useful, as well as being a whole heap *meatier* than the code snippets you've seen so far in this book.

Python is particularly strong on the server side of the Web, which is where we're going to build and deploy our webapp in this chapter.

But, before we get going, let's make sure everyone is on the same page by reviewing how the Web works.

WEBAPPS UP CLOSE

No matter what you do on the Web, it's all about *requests* and *responses*. A **web request** is sent from a web browser to a web server as the result of some user interaction. On the web server, a **web response** (or *reply*) is formulated and returned to the web browser. The entire process can be summarized in five steps, as follows:



Deciding what to do next

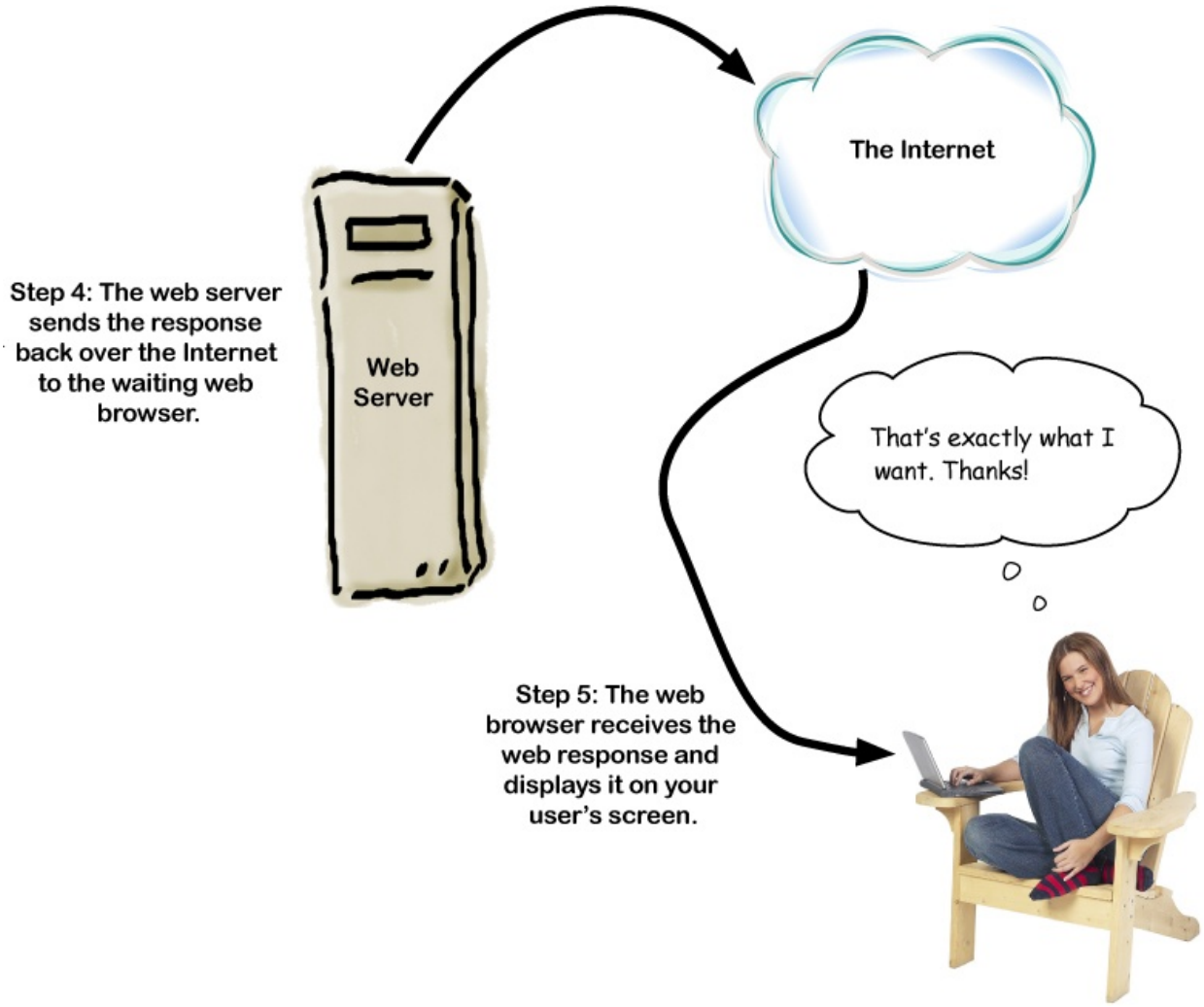
One of two things happen at this point. If the web request is for **static content**—such as an HTML file, image, or anything else stored on the web server's hard disk—the web server locates the *resource* and prepares to return it to the web browser as a web response.

If the request is for **dynamic content**—that is, content that must be *generated*, such as search results or the current contents of an online shopping basket—the web server runs some code to produce the web response.

The (potentially) many substeps of Step 3

In practice, Step 3 can involve multiple substeps, depending on what the web server has to do to produce the response. Obviously, if all the server has to do is locate static content and return it to the browser, the substeps aren't too taxing, as it's just a matter of reading from the web server's disk drive.

However, when dynamic content must be generated, the substeps involve the web server running code and then capturing the output from the program as a web response, before sending the response back to the waiting web browser.

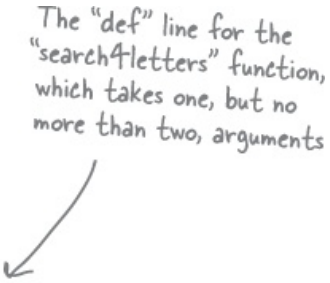


What Do We Want Our Webapp to Do?

As tempting as it always is to *just start coding*, let's first think about how our webapp is going to work.

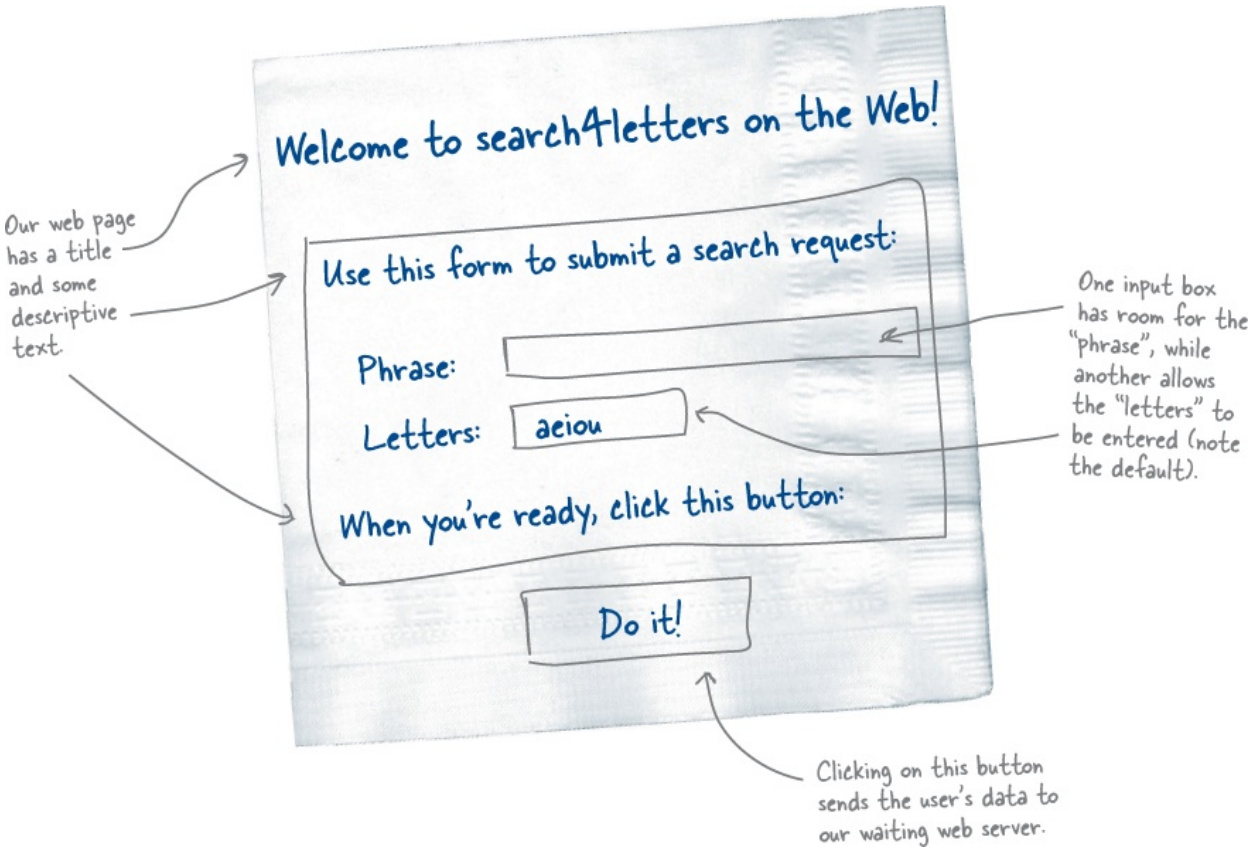
Users interact with our webapp using their favorite web browser. All they have to do is enter the URL for the webapp into their browser's address bar to access its services. A web page then appears in the browser asking the user to provide arguments to the `search4letters` function. Once these are entered, the user clicks on a button to see their results.

Recall the `def` line for our most recent version of `search4letters`, which shows the function expecting at least one—but no more than two— arguments: a `phrase` to search, together with the `letters` to search for. Remember, the `letters` argument is optional (defaulting to `aeiou`):




```
def search4letters(phrase:str, letters:str='aeiou') -> set:
```

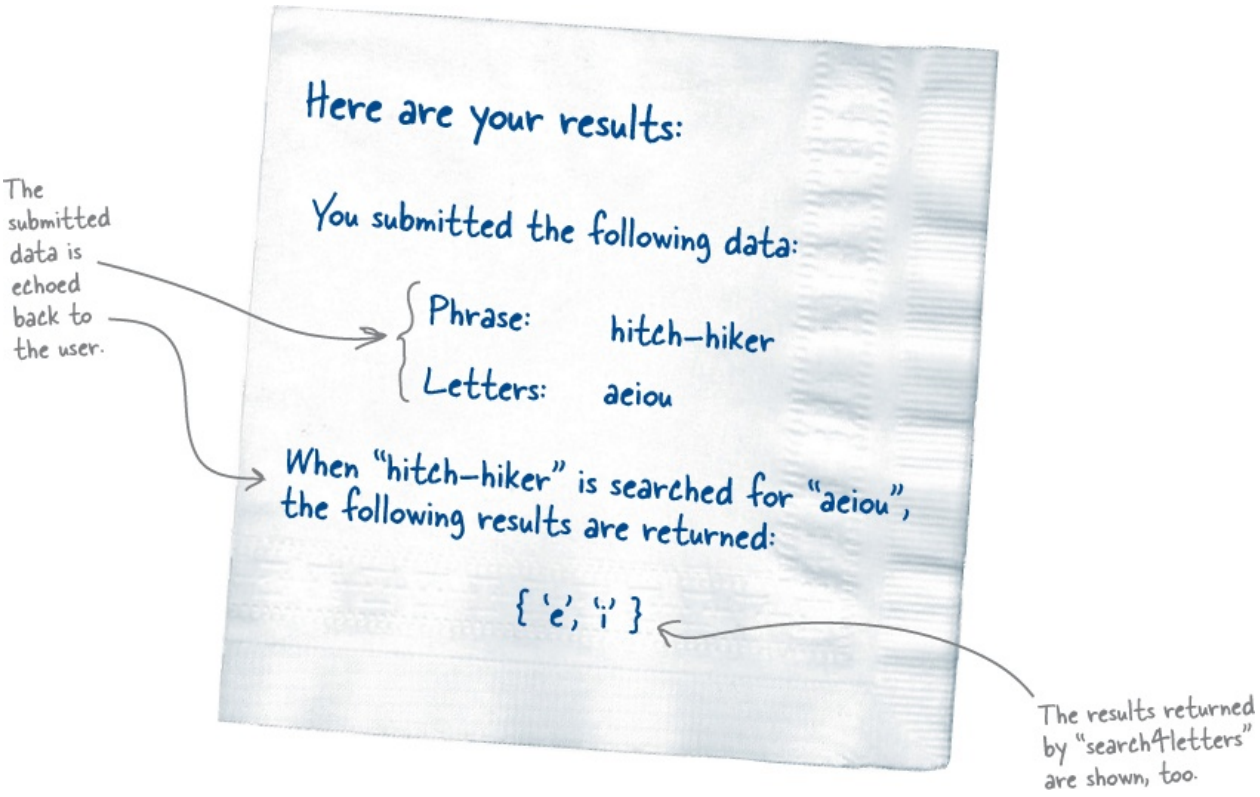
Let's grab a paper napkin and sketch out how we want our web page to appear. Here's what we came up with:



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:



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:

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
```

That's a lowercase "f" for "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

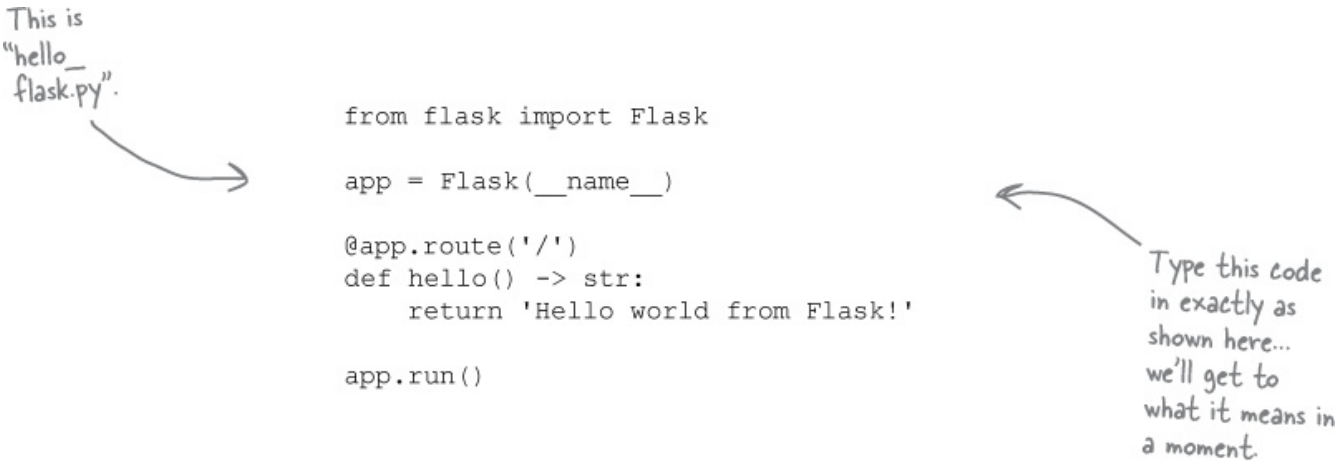
Django is a hugely popular web application framework within the Python community. It has an especially strong, prebuilt administration facility that can make working with large webapps very manageable. It's overkill for what we're doing here, so we've opted for the much simpler, but more lightweight,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



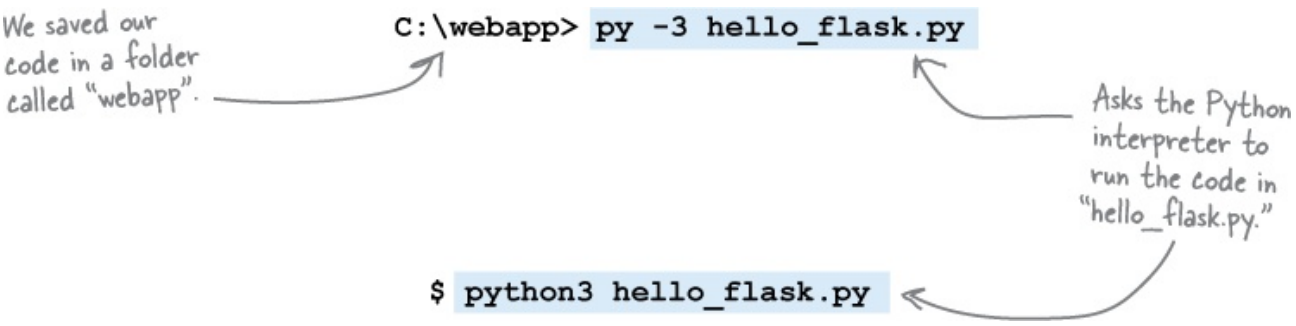
RUN FLASK FROM YOUR OS COMMAND LINE

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:



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:

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`):

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

If you see this message, all is well.

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:

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

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

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:

```
* 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 -
```

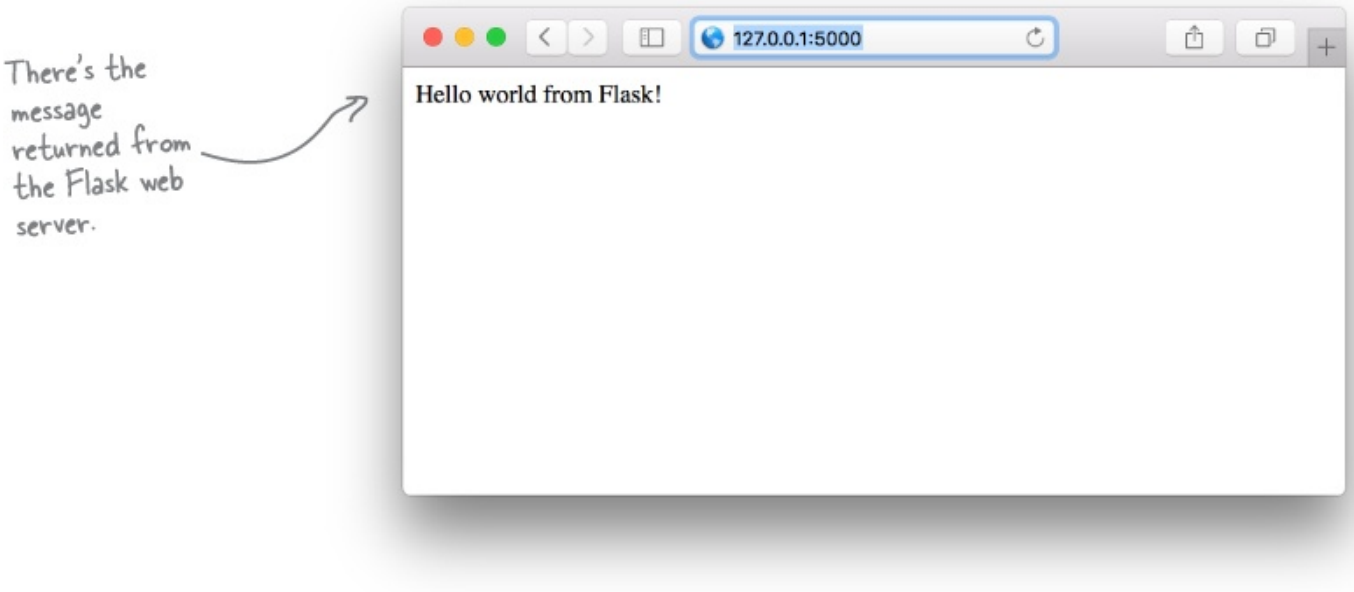
Ah ha! Something happened.

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)

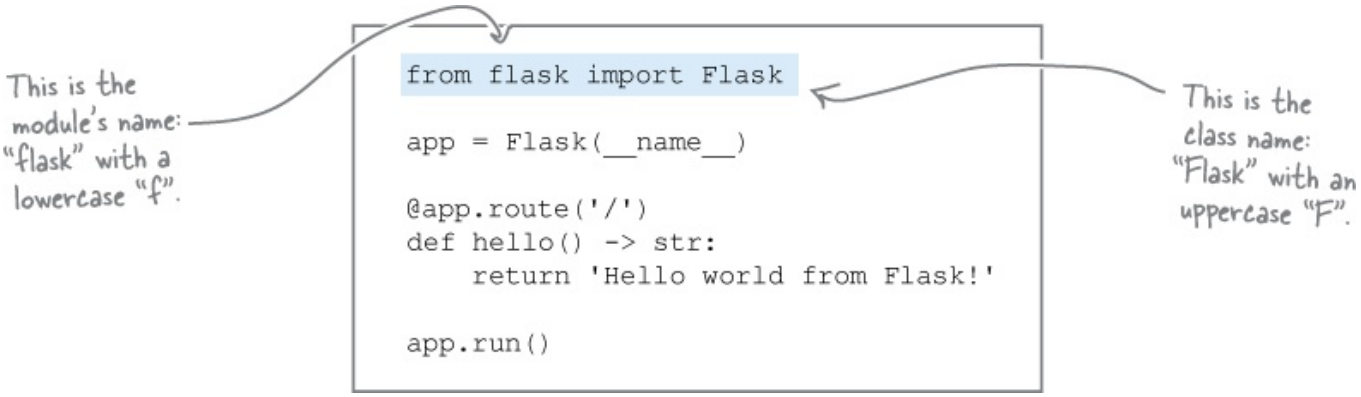
In addition to Flask updating the terminal with a status line, your web browser now displays the web server's response. Here's how our browser now looks (this is *Safari* on *Mac OS X*):



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:

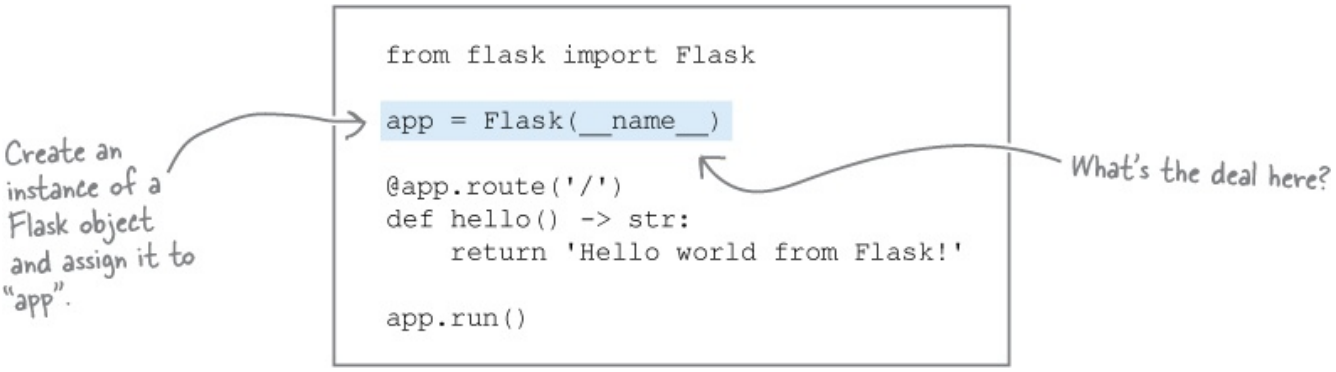


Remember when we discussed alternate ways of importing?

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).

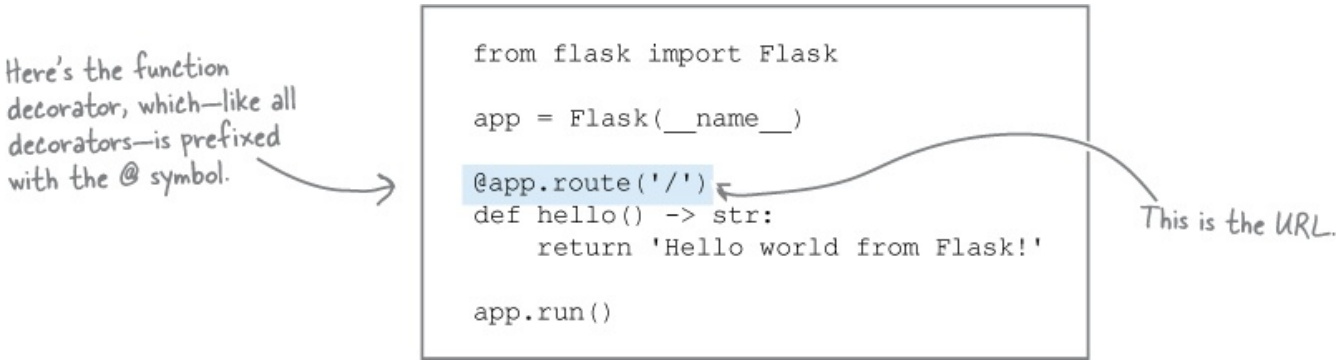
You might want to read that last sentence a few times.

GEEK BITS

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

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:



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.

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

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**

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

That URL does not exist: 404!

```
* 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 Functionalit y 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.*)

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...

```
$ 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)
```

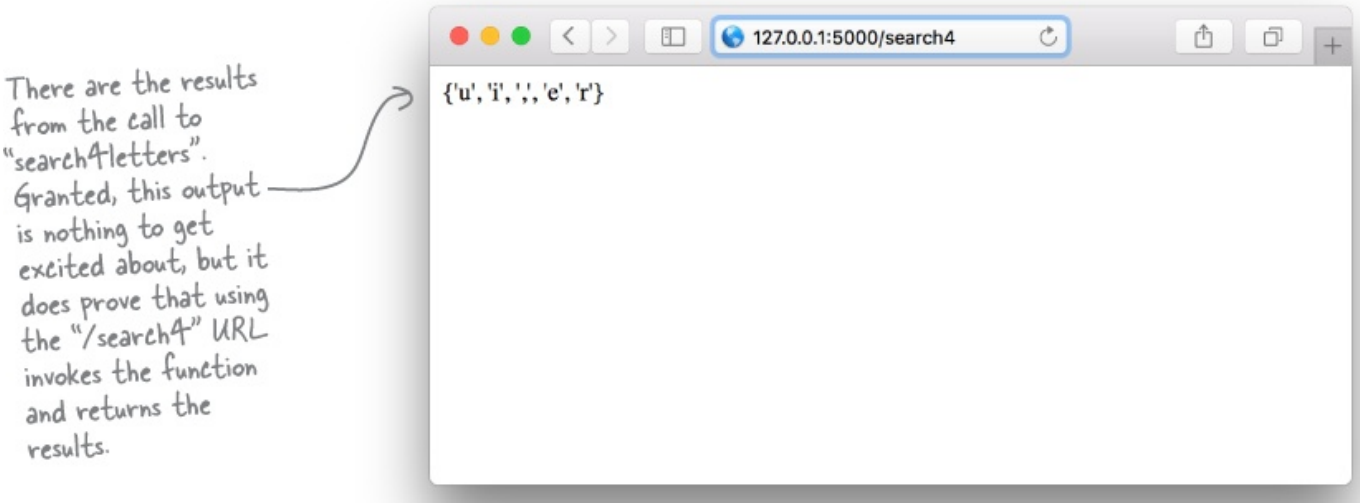
...then restart it.

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 NO DUMB QUESTIONS

Q: **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: **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.

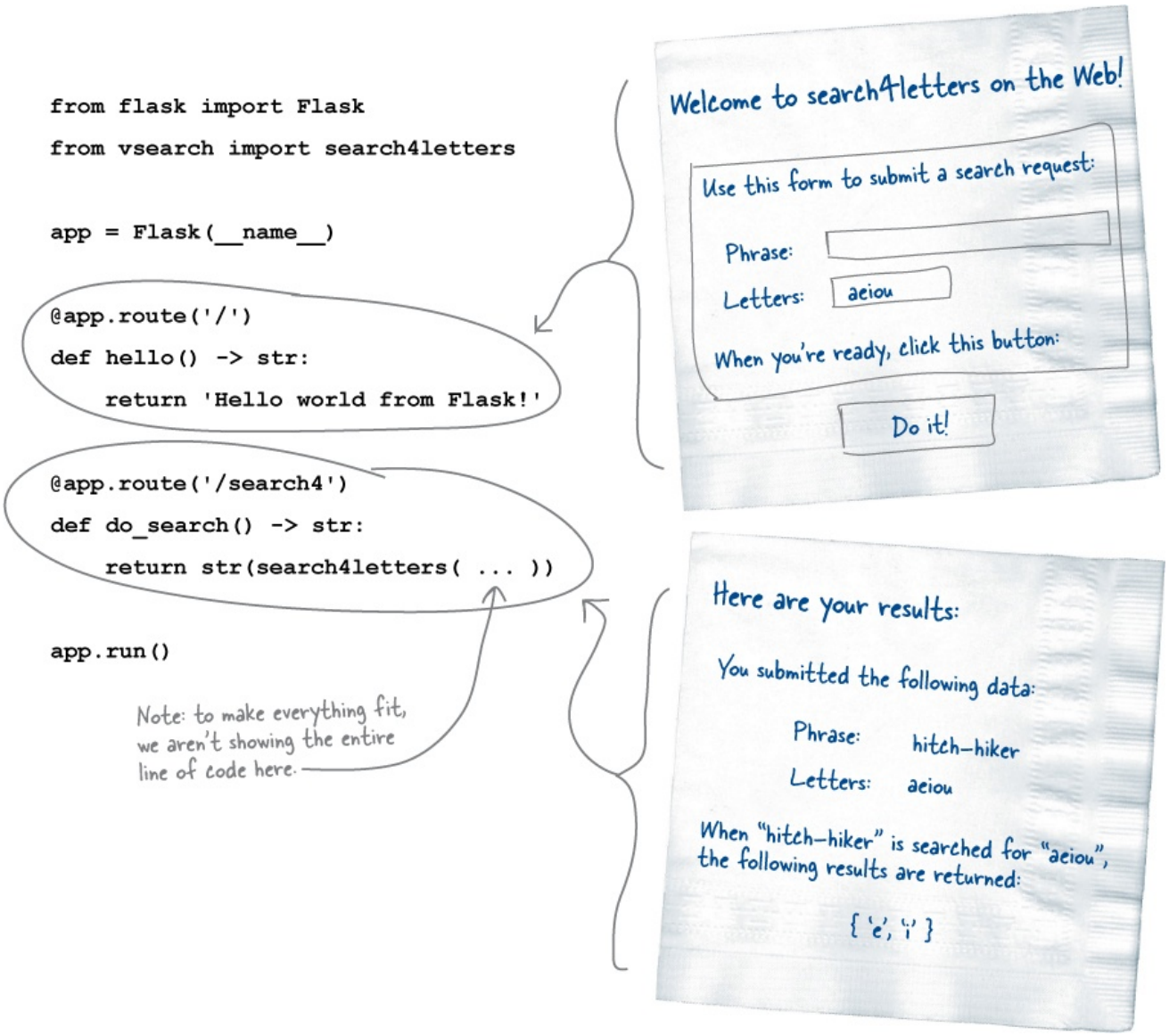
Q: **Q: Can I use some protocol port other than 5000 when I test and run my Flask webapp?**

A: **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:



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.

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. ☺

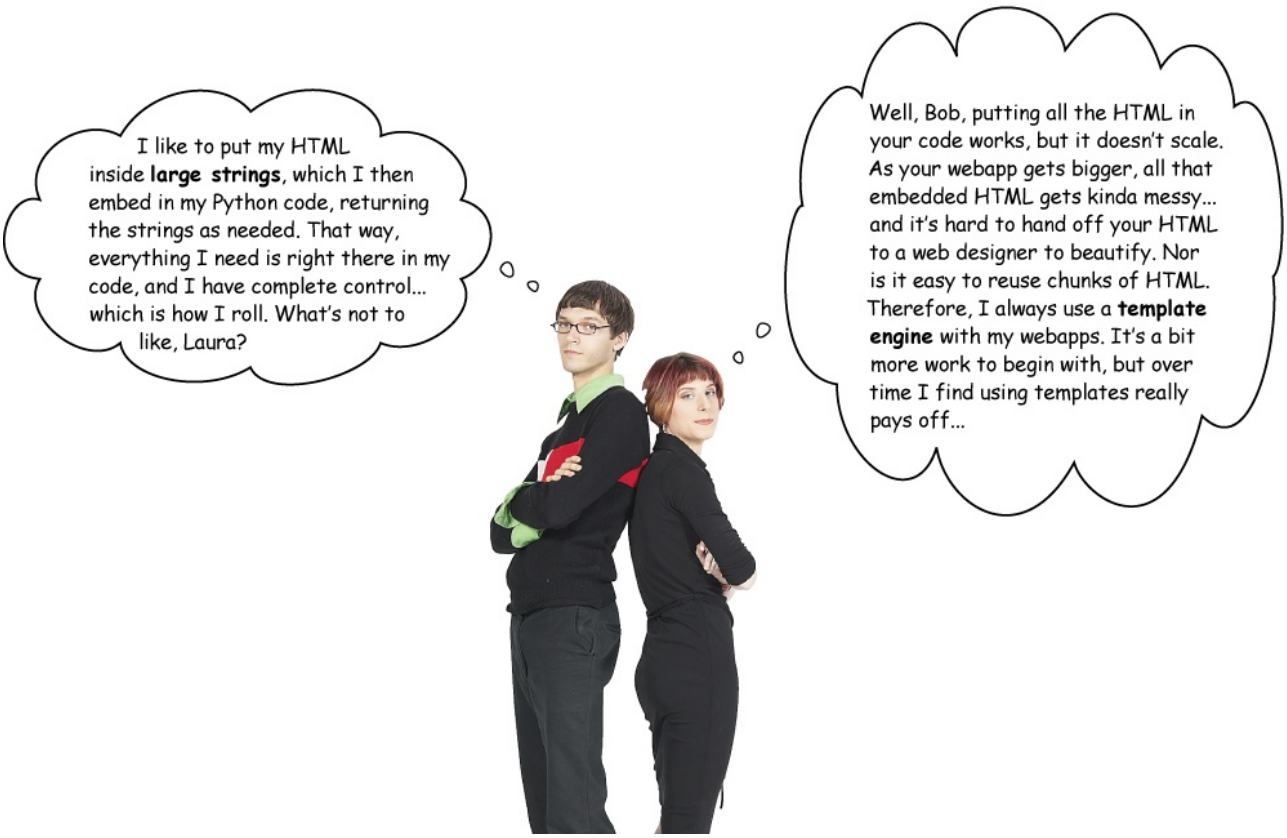
BUT...WHAT IF YOU'RE NEW TO ALL THIS HTML STUFF?

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).

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:



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

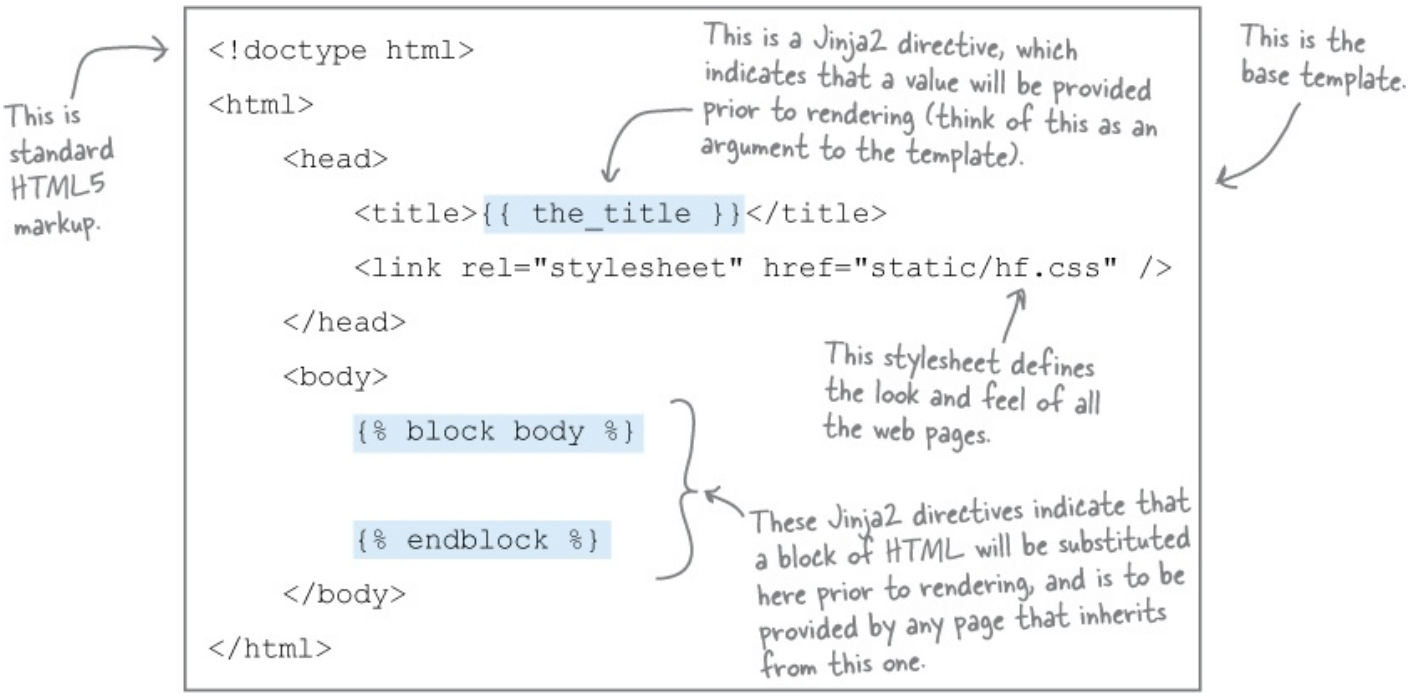
Template engines let programmers apply the object-oriented notions of inheritance and reuse to the production of textual data, such as web pages.

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:



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.

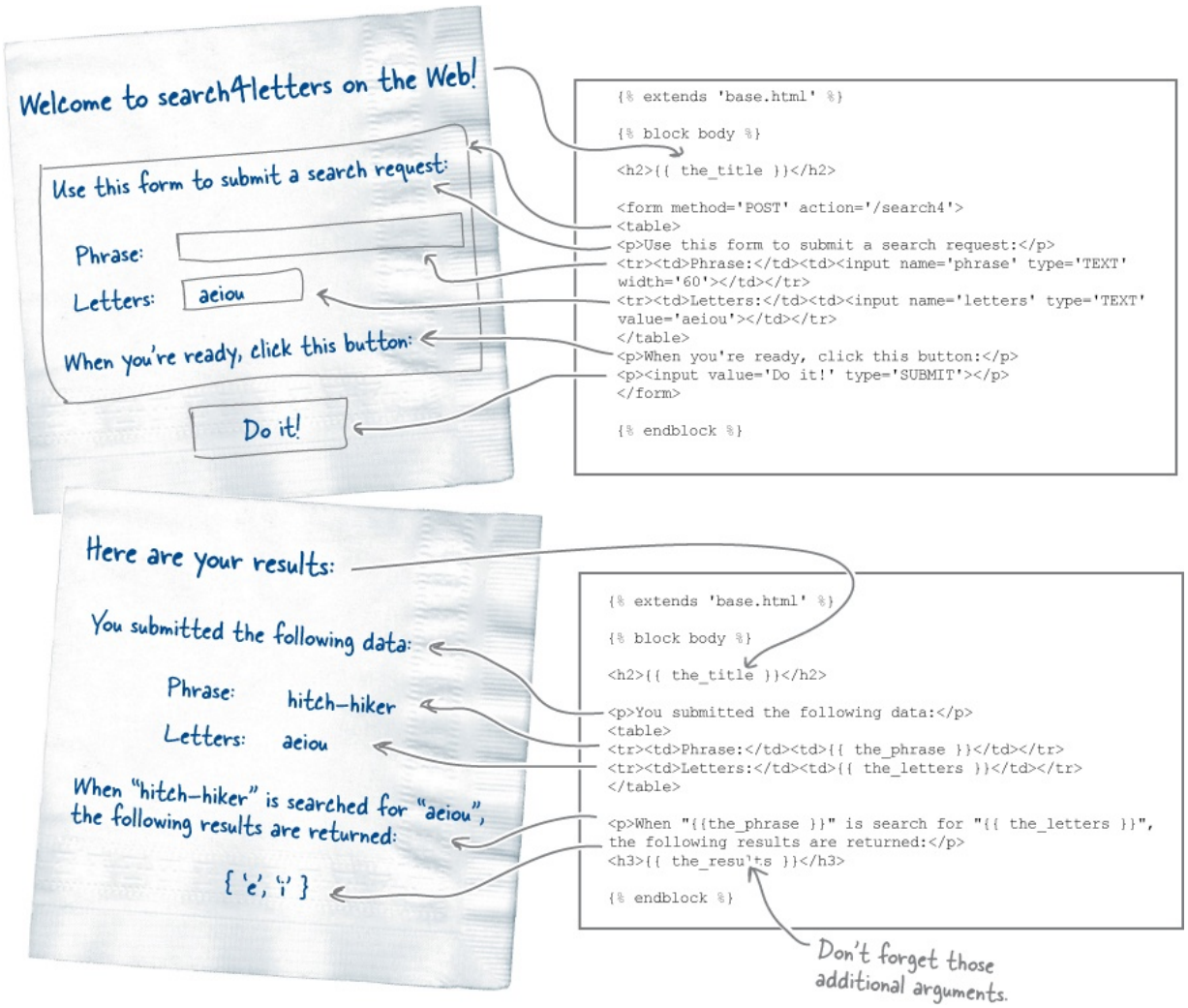
Templates Relate to Web Pages

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.

NOTE

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:



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:


```
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.

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

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

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.

NOTE

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

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:

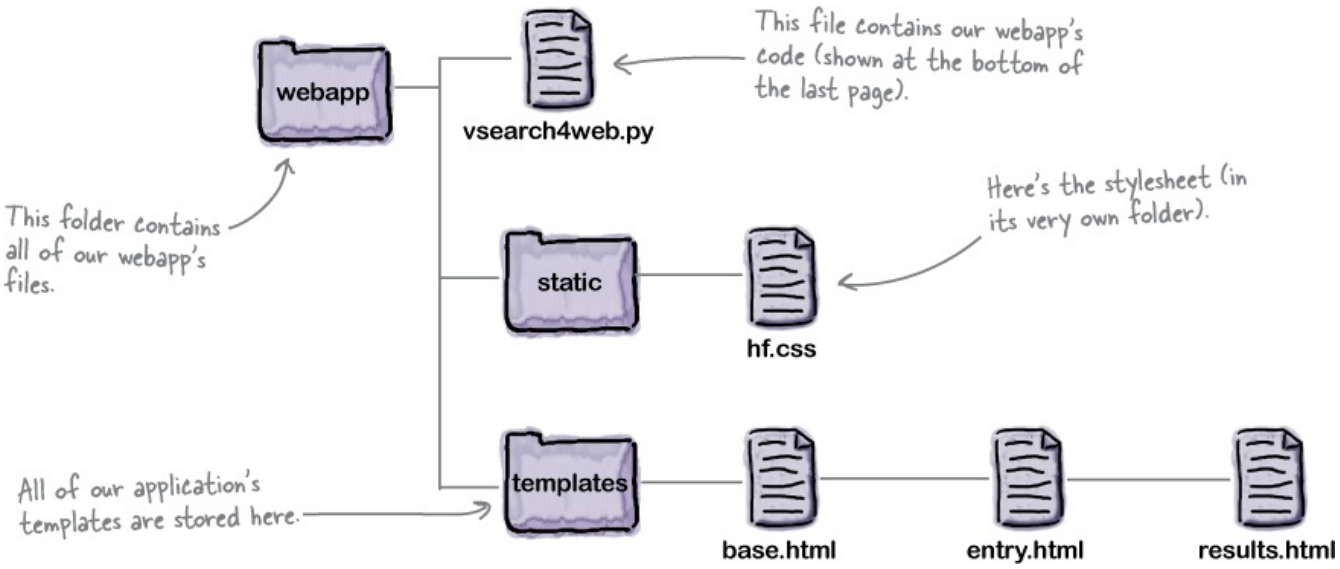
```
            ...
        <title>{{ the_title }}</title>
        <link rel="stylesheet" href="static/hf.css" />
    </head>
            ...
```

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...

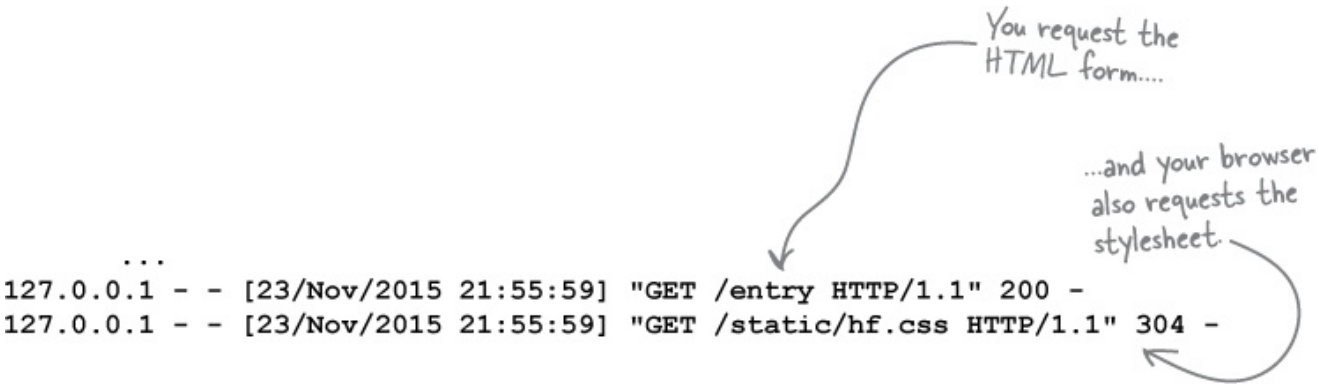
Start up your new code (which is in the "vsearch4web.py" file).

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:

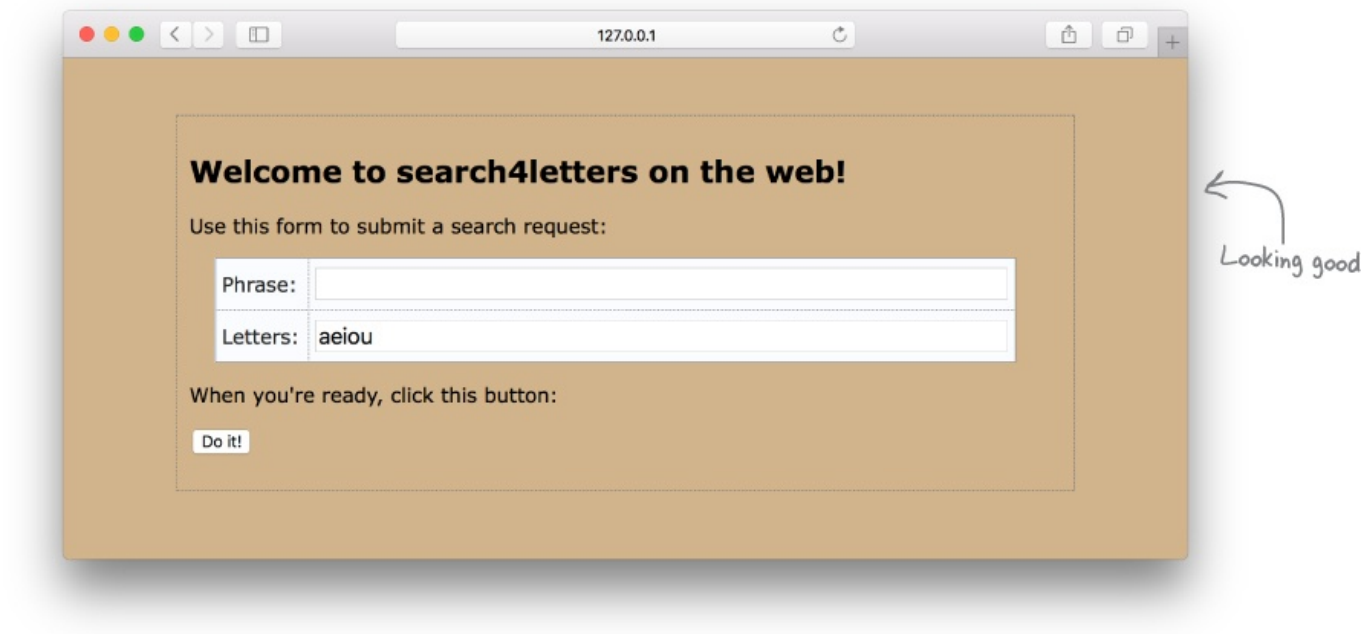
http://127.0.0.1:5000/entry

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:

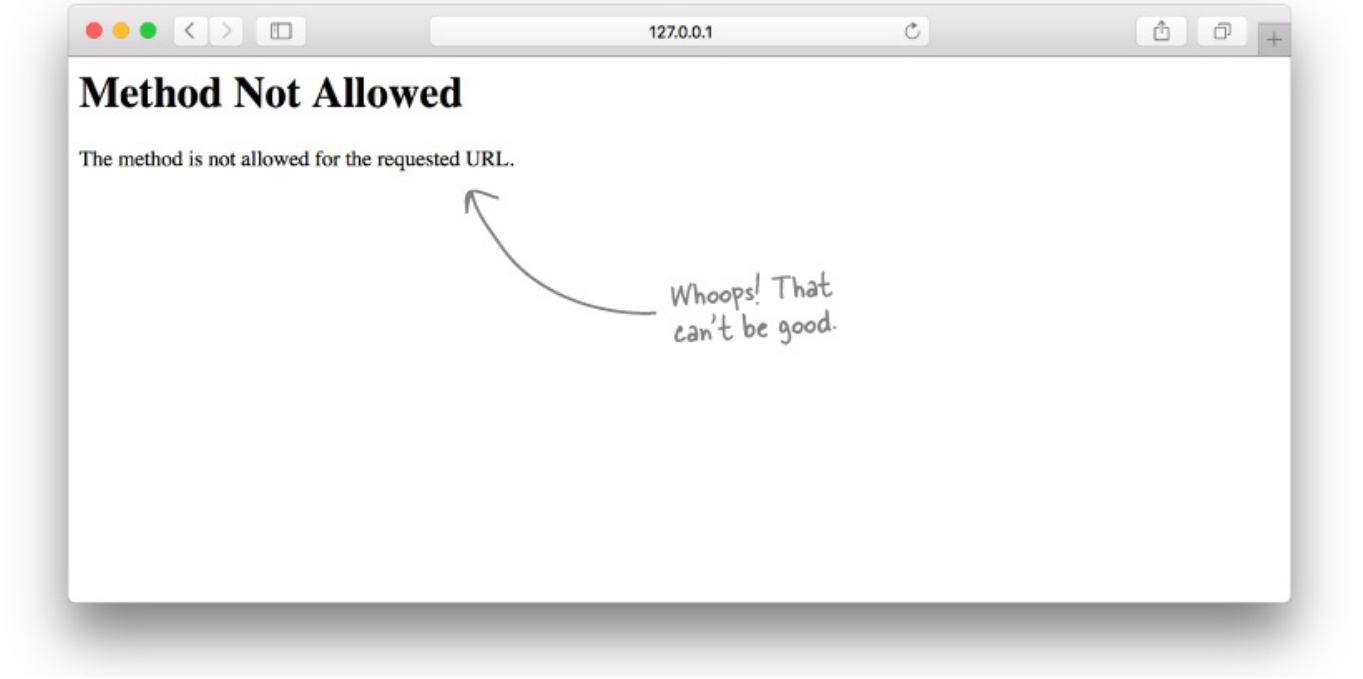


TEST DRIVE

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:



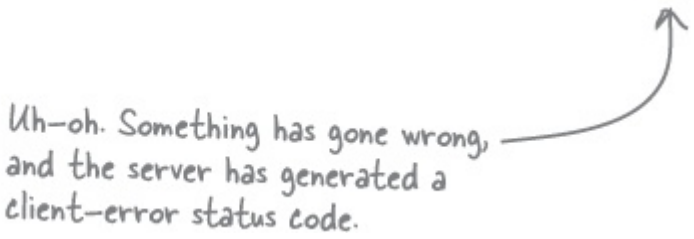
This is a bit of a bummer, isn't it? Let's see what's going on.

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 -
```



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

A: **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)
```

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:

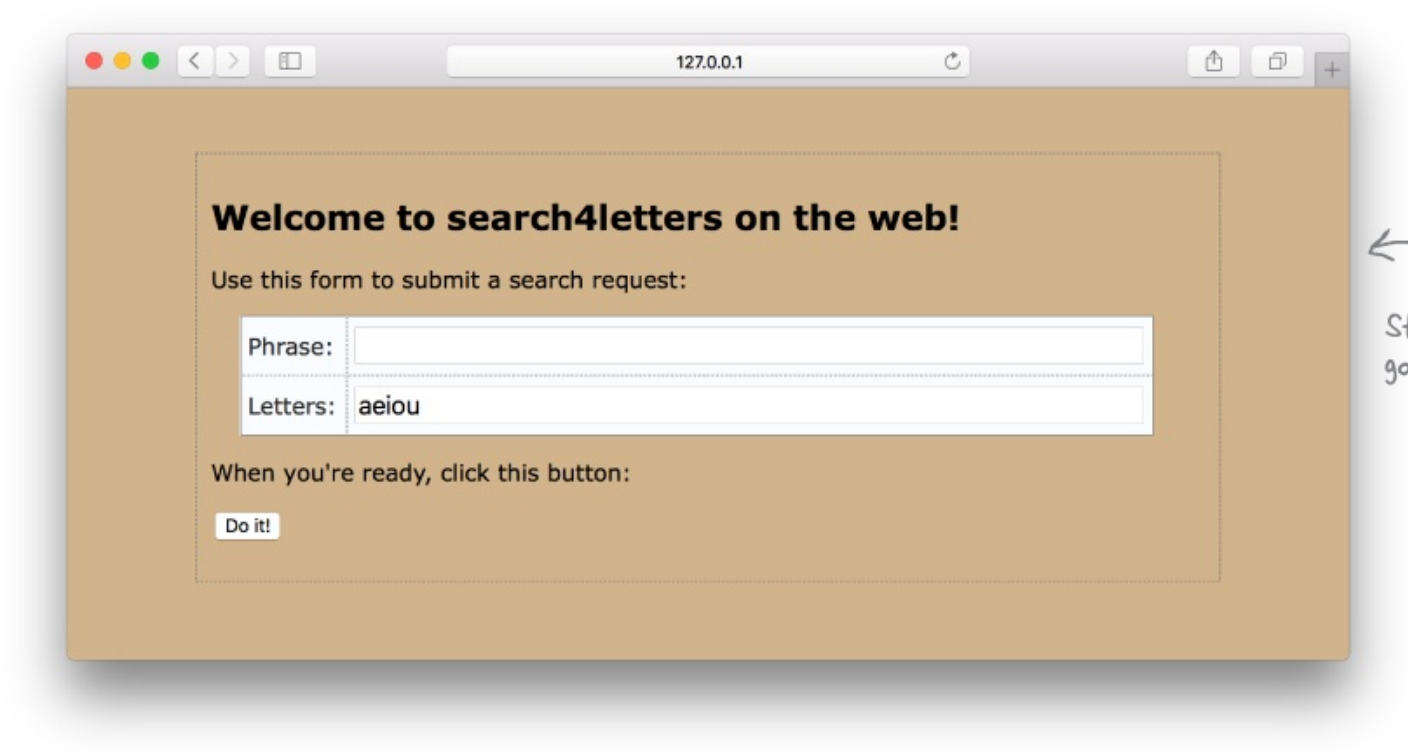
```
$ python3 vsearch4web.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 228-903-465
```

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.

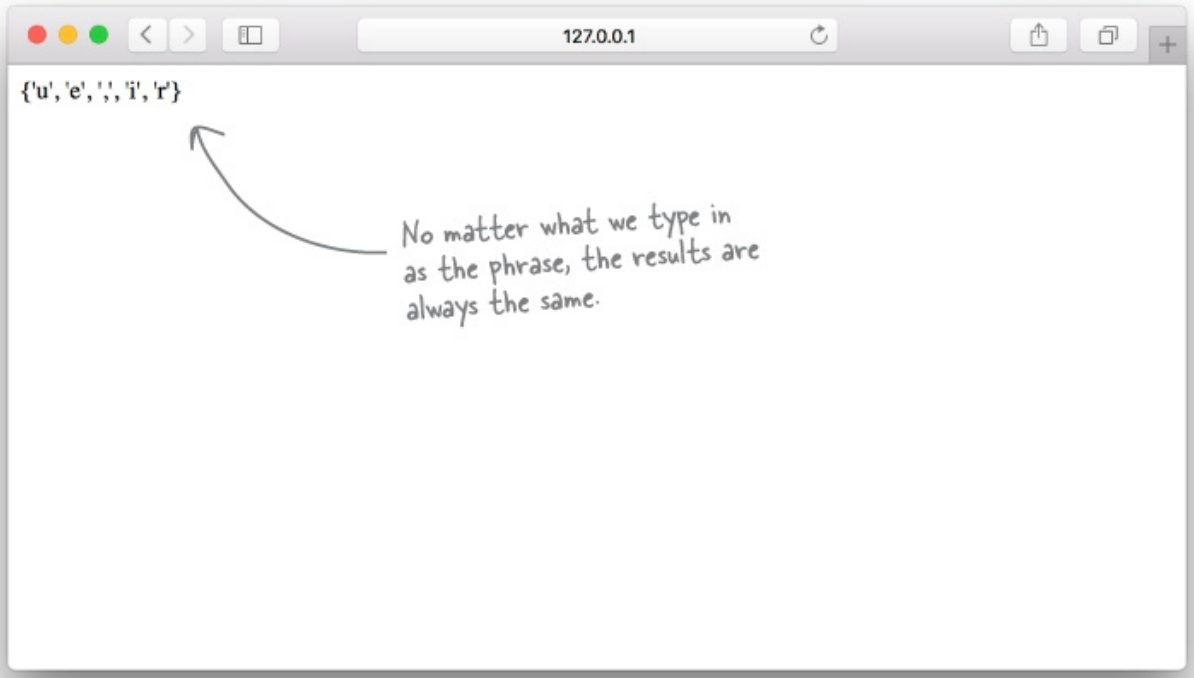
Now that we are up and running again, let's interact with our webapp once more and see what's changed.

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

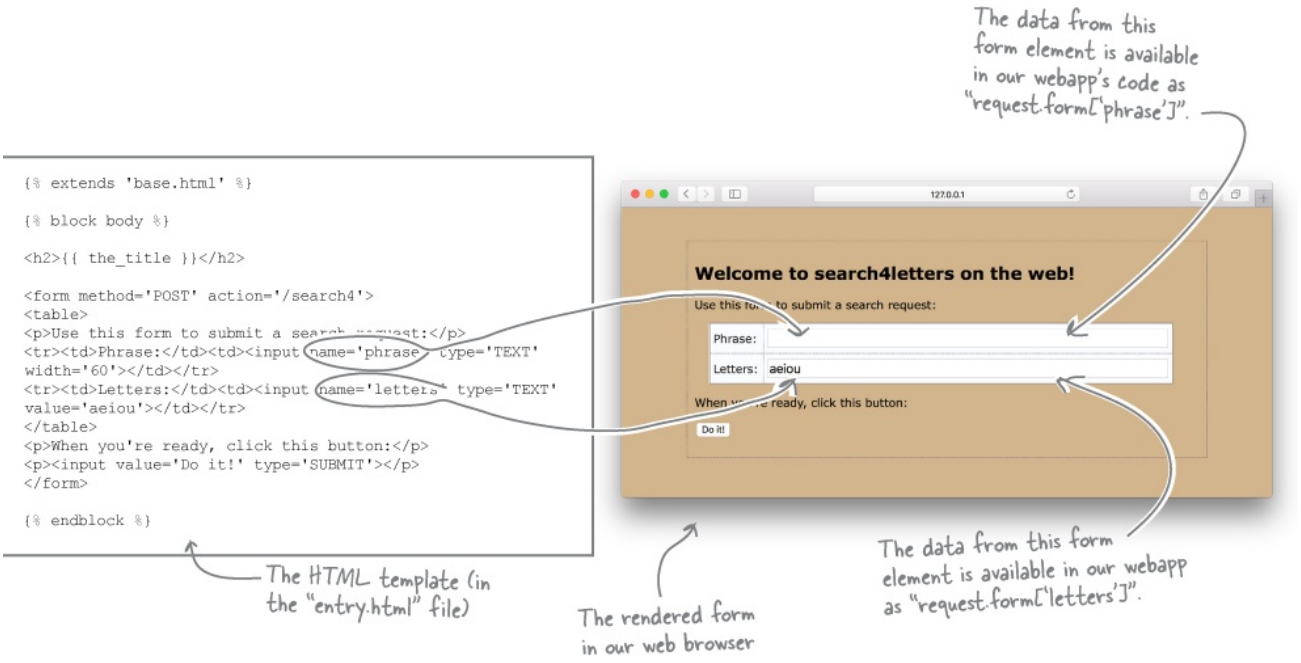
Our webapp no longer fails with a "Method Not Allowed" error. Instead, it always returns the same set of characters: *u*, *e*, *comma*, *i*, and *r*. If you take a quick look at the code that executes when the */search4* URL is posted to, you'll see why this is: the values for `phrase` and `letters`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



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):

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

```
$ 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
```

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*).

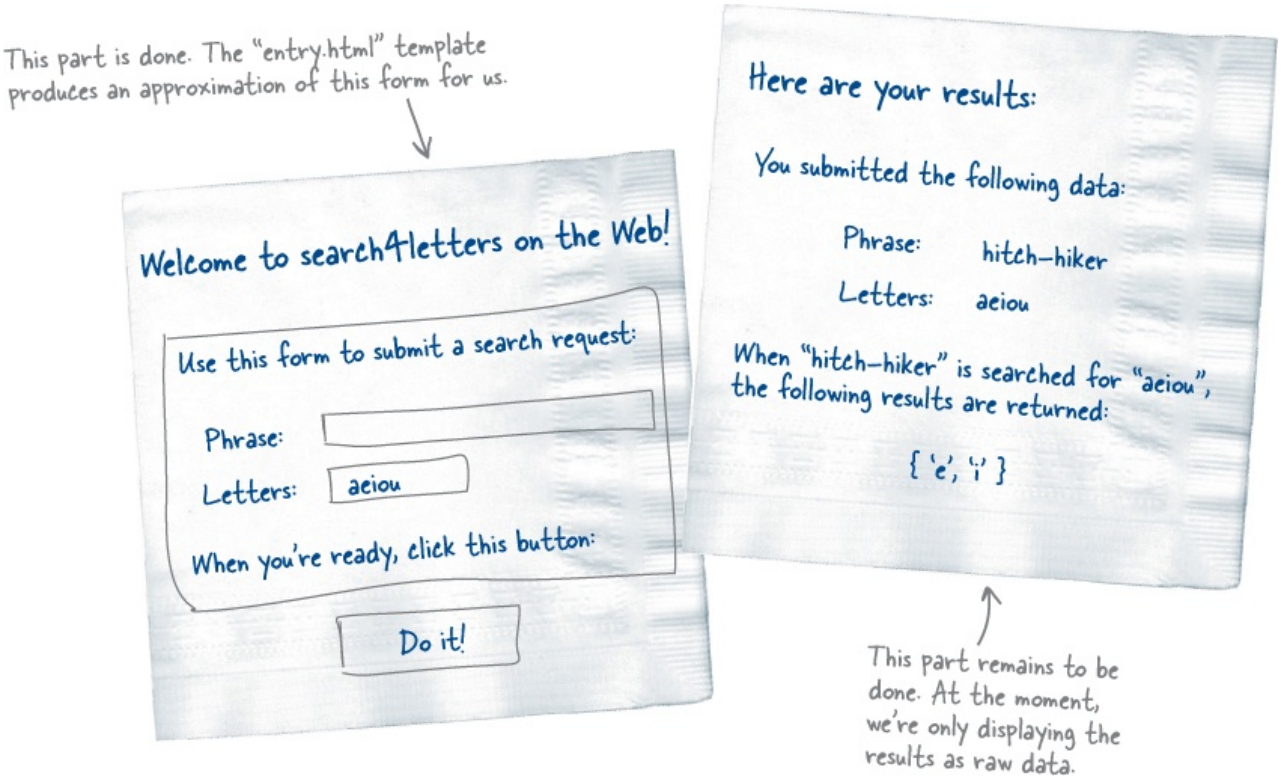
TEST DRIVE

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.

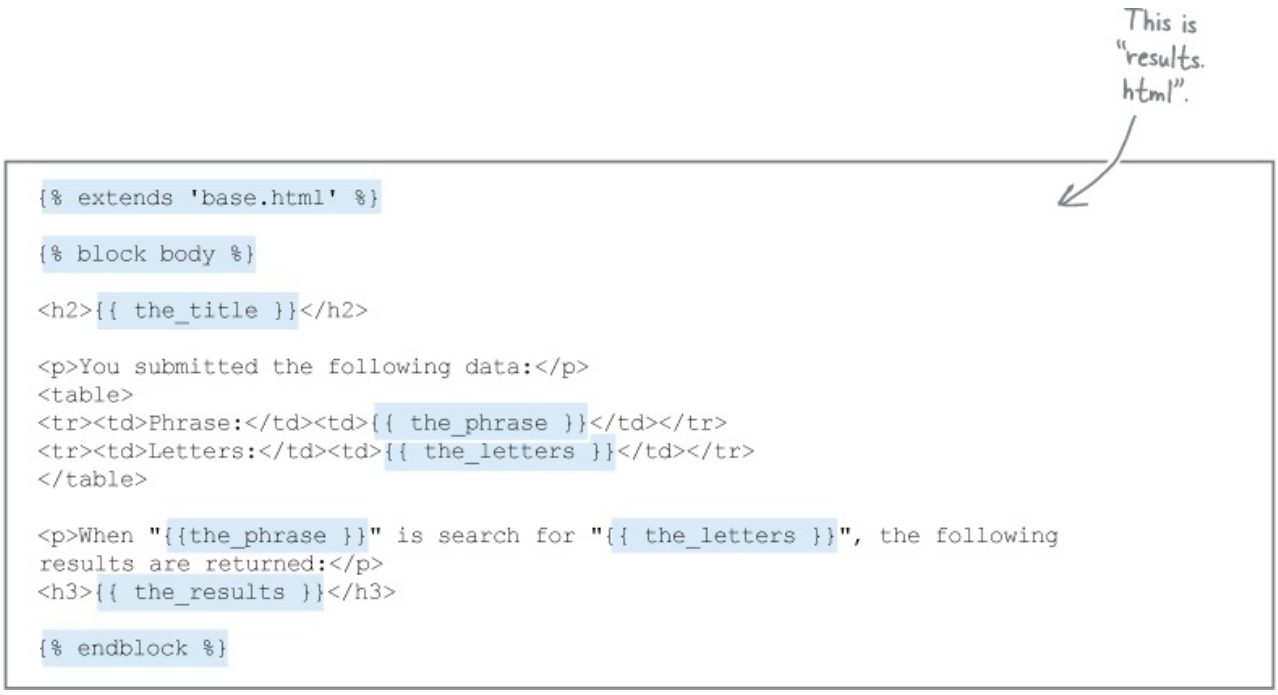
THERE ARE NO DUMB QUESTIONS

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

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




```
{% 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.

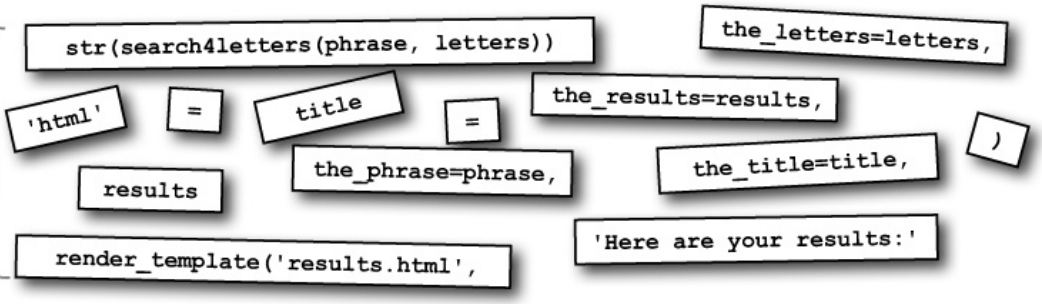
TEMPLATE MAGNETS

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.

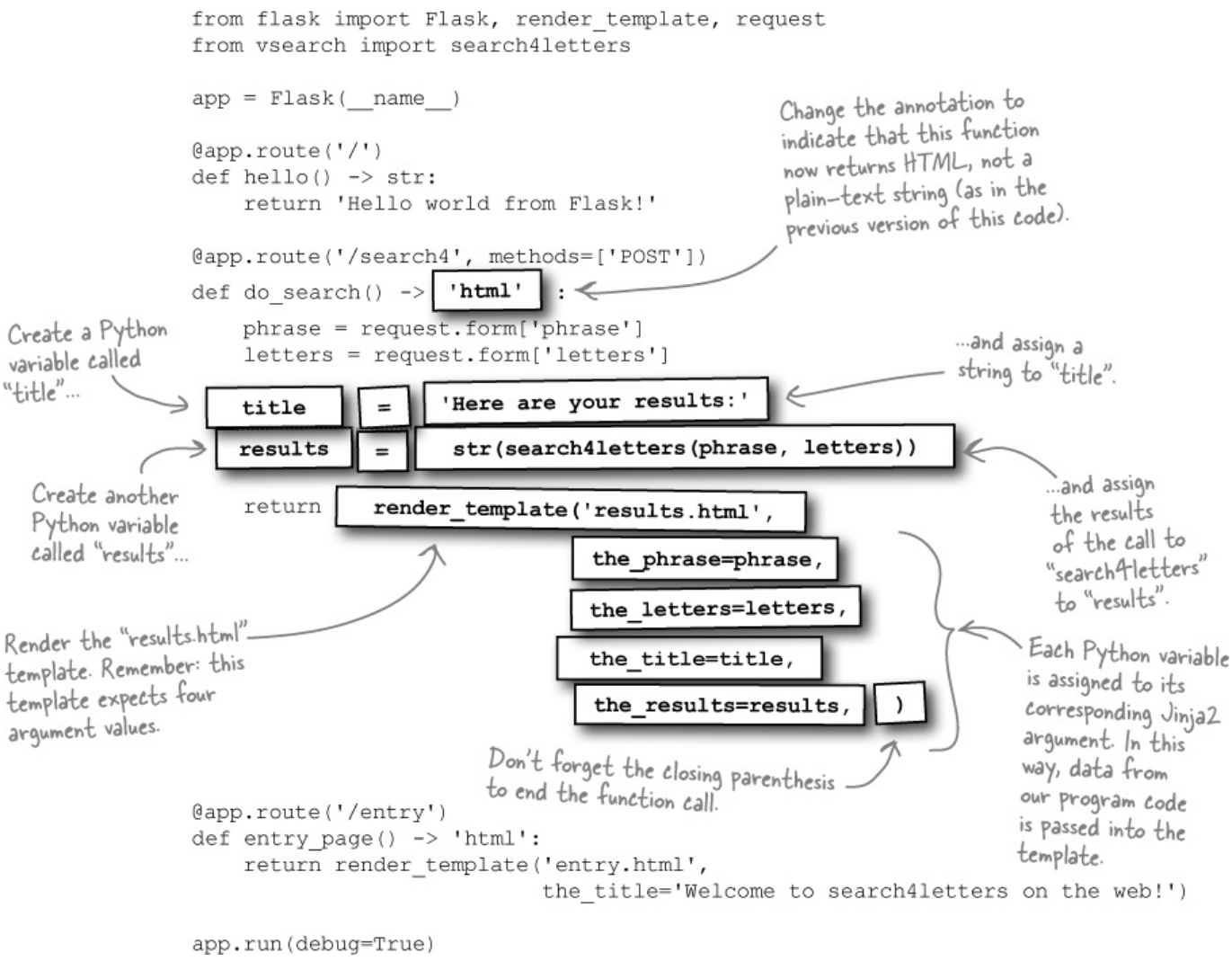
Here are the magnets you have to work with.



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:



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.

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)
```

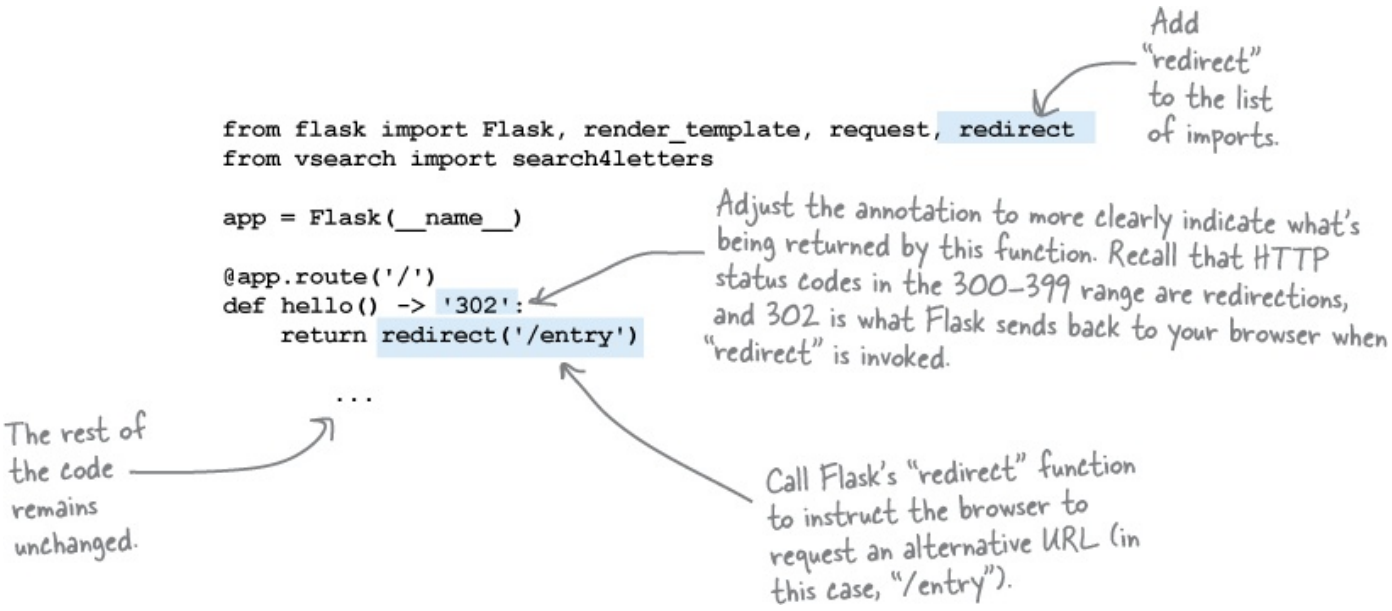
This extra comma looks a little strange, but is perfectly fine (though optional) Python syntax.

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 /.

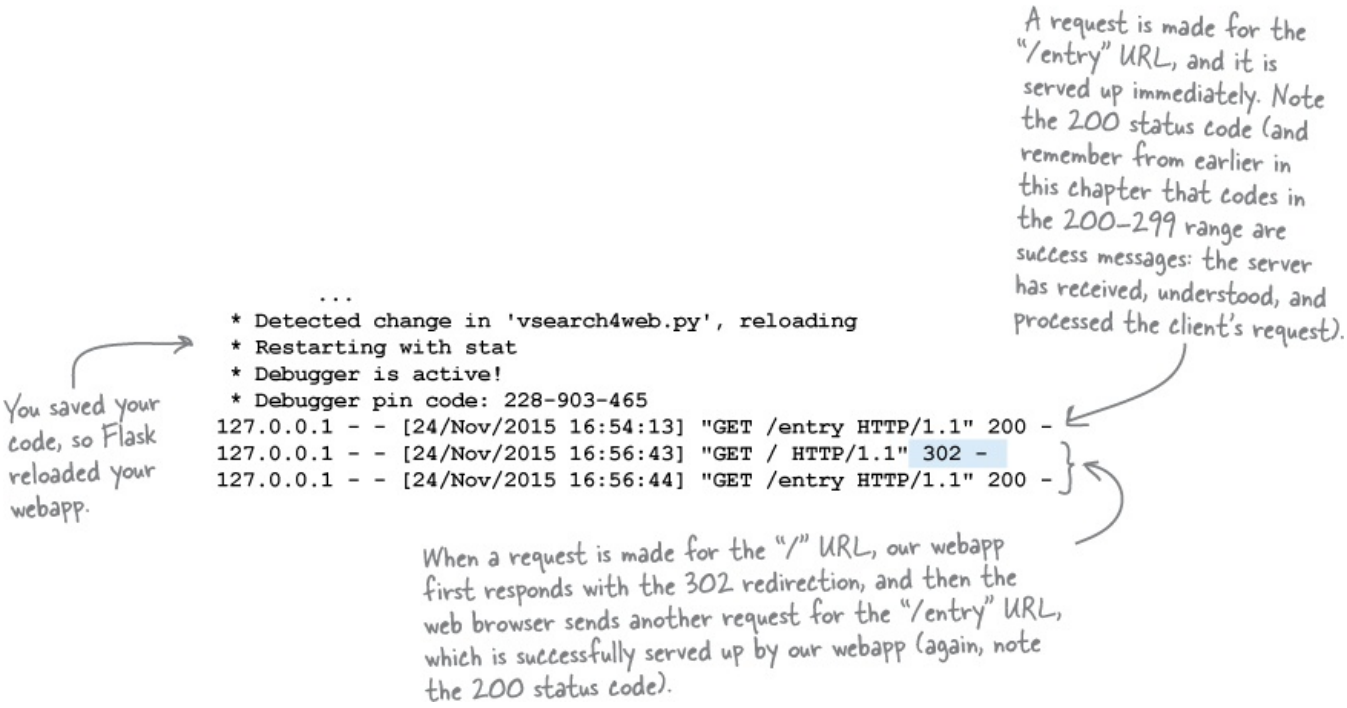
Redirect to Avoid Unwanted Errors

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:



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:



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?

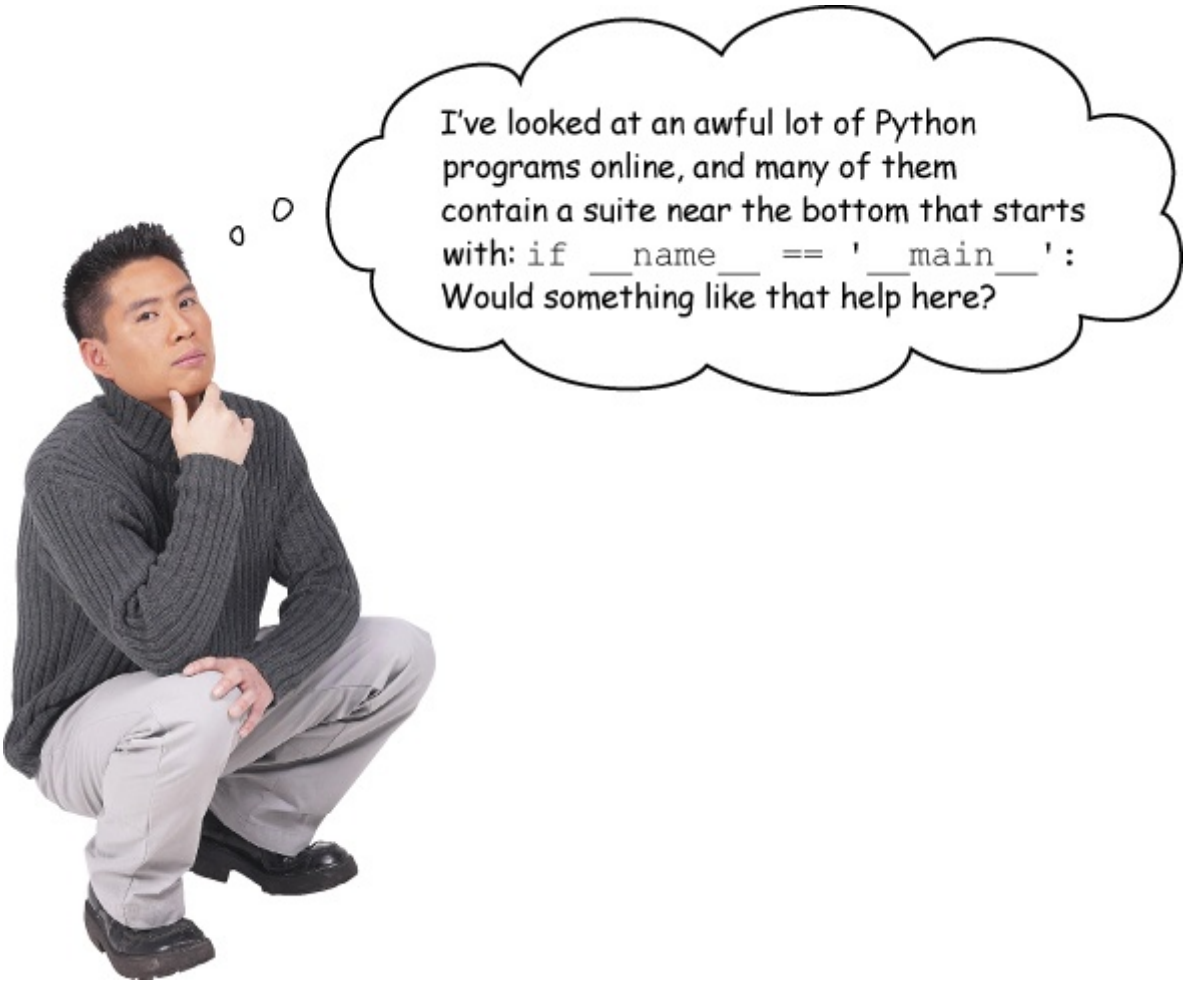
Preparing Your Webapp for the Cloud

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*...



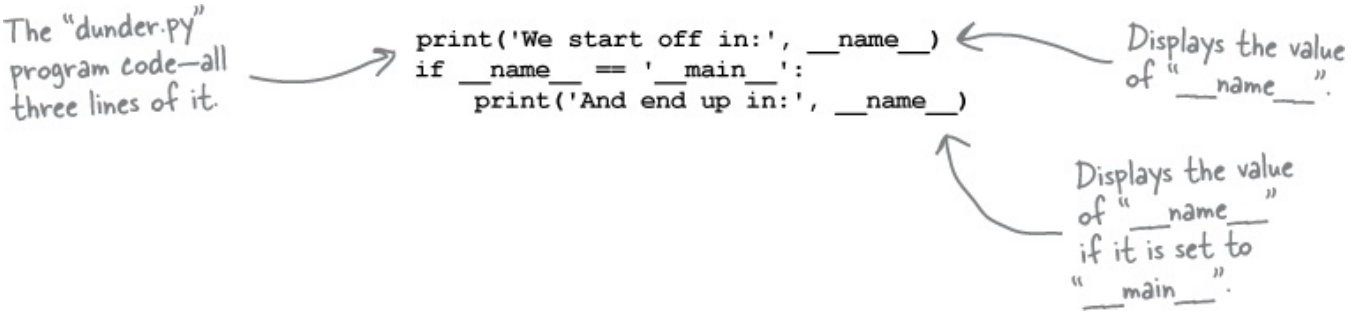
Yes, that's a great suggestion.

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):



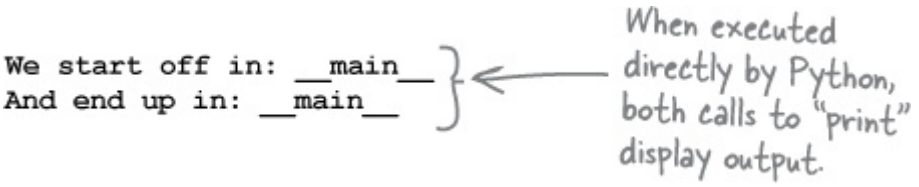
Use your editor (or IDLE) to create the `dunder.py` file, then run the program at a command prompt to see what happens. If you're on *Windows*, use this command:

```
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:



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__)
```