Chapter 18. The Web, Untangled

Oh, what a tangled web we weave…

—Walter Scott, Marmion

Straddling the French–Swiss border is CERN—a particle physics research institute that smashes atoms multiple times, just to make sure.

All that smashing generates a mountain of data. In 1989, the English scientist Tim Berners-Lee first circulated a proposal within CERN to help disseminate information there and across the research community. He called it the *World Wide Web* and distilled its design into three simple ideas:

HTTP (Hypertext Transfer Protocol)
: A protocol for web clients and servers to interchange requests and responses.

HTML (Hypertext Markup Language)
: A presentation format for results.

URL (Uniform Resource Locator)
: A way to uniquely represent a server and a *resource* on that server.

In its simplest usage, a web client (I think Berners-Lee was the first to use the term *browser*) connected to a web server with HTTP, requested a URL, and received HTML.

This was all built on the networking base from the *internet*, which at the time was noncommercial, and known only to a few universities and research organizations.

He wrote the first web browser and server on a NeXT[1] computer. Web awareness really expanded in 1993, when a group of students at the University of Illinois released the Mosaic web browser (for Windows, the Macintosh, and Unix) and the NCSA *httpd* server. When I downloaded Mosaic that summer and started building sites, I had no idea that the web and the internet would soon become part of everyday life. The internet[2] was still officially noncommercial then; there were about 500 known web servers in the world. By the end of 1994, the number of web servers had grown to 10,000. The internet was opened to commercial use, and the authors of Mosaic founded Netscape to write commercial web software. Netscape went public as part of the early internet frenzy, and the web's explosive growth has never stopped.

Almost every computer language has been used to write web clients and web servers. The dynamic languages Perl, PHP, and Ruby have been especially popular. In this chapter, I show why Python is a particularly good language for web work at every level:

- Clients, to access remote sites
- Servers, to provide data for websites and web APIs
- Web APIs and services, to interchange data in other ways than viewable web pages

And while we're at it, we'll build an actual interactive website in the exercises at the end of this chapter.

Web Clients

The low-level network plumbing of the internet is called Transmission Control Protocol/Internet Protocol, or more commonly, simply TCP/IP ("TCP/IP" goes into more detail about this). It moves bytes among computers, but doesn't care about what those bytes mean. That's the job of higher-level *protocols*—syntax definitions for specific purposes. HTTP is the standard protocol for web data interchange.

The web is a client-server system. The client makes a *request* to a server: it opens a TCP/IP connection, sends the URL and other information via HTTP, and receives a *response*.

The format of the response is also defined by HTTP. It includes the status of the request, and (if the request succeeded) the response's data and format.

The most well-known web client is a web *browser*. It can make HTTP requests in a number of ways. You might initiate a request manually by typing a URL into the location bar or clicking a link in a web page. Very often, the data returned is used to display a website —HTML documents, JavaScript files, CSS files, and images—but it can be any type of data, not just that intended for display.

An important aspect of HTTP is that it's *stateless*. Each HTTP connection that you make is independent of all the others. This simplifies basic web operations but complicates others. Here are just a few samples of the challenges:

Caching
: Remote content that doesn't change should be saved by the web client and used to avoid downloading from the server again.

Sessions
: A shopping website should remember the contents of your shopping cart.

Authentication
: Sites that require your username and password should remember them while you're logged in.

Solutions to statelessness include *cookies*, in which the server sends the client enough specific information to be able to identify it uniquely when the client sends the cookie back.

Test with telnet

HTTP is a text-based protocol, so you can actually type it yourself for web testing. The ancient `telnet` program lets you connect to any server and port, and type commands to any service that's running there. For secure (encrypted) connections to other machines, it's been replaced by `ssh`.

Let's ask everyone's favorite test site, Google, some basic information about its home page. Type this:

```
$ telnet www.google.com 80
```

If there is a web server on port 80 (this is where unencrypted `http` usually runs; encrypted `https` uses port 443) at *google.com* (I think that's a safe bet), `telnet` will print some reassuring information, and then display a final blank line that's your cue to type something else:

```
Trying 74.125.225.177...
Connected to www.google.com.
Escape character is '^]'.
```

Now, type an actual HTTP command for `telnet` to send to the Google web server. The most common HTTP command (the one your browser uses when you type a URL in its location bar) is `GET`. This retrieves the contents of the specified resource, such as an HTML file, and returns it to the client. For our first test, we'll use the HTTP command `HEAD`, which just retrieves some basic information *about* the resource:

```
HEAD / HTTP/1.1
```

Add an extra carriage return to send a blank line so the remote server knows you're all done and want a response. That `HEAD /` sends the HTTP `HEAD` *verb* (command) to get information about the home page (`/`). You'll receive a response such as this (I trimmed some of the long lines using … so they wouldn't stick out of the book):

```
HTTP/1.1 200 OK
Date: Mon, 10 Jun 2019 16:12:13 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Set-Cookie: 1P_JAR=...; expires=... GMT; path=/; domain=.google.com
Set-Cookie: NID=...; expires=... GMT; path=/; domain=.google.com; HttpOnly
Transfer-Encoding: chunked
Accept-Ranges: none
Vary: Accept-Encoding
```

These are HTTP response headers and their values. Some, like `Date` and

`Content-Type`, are required. Others, such as `Set-Cookie`, are used to track your activity across multiple visits (we talk about *state management* a little later in this chapter). When you make an HTTP `HEAD` request, you get back only headers. If you had used the HTTP `GET` or `POST` commands, you would also receive data from the home page (a mixture of HTML, CSS, JavaScript, and whatever else Google decided to throw into its home page).

I don't want to leave you stranded in `telnet`. To close `telnet`, type the following:

```
q
```

Test with curl

Using `telnet` is simple, but is a completely manual process. The `curl` program is probably the most popular command-line web client. Documentation includes the book *Everything Curl*, in HTML, PDF, and ebook formats. A table compares `curl` with similar tools. The download page includes all the major platforms, and many obscure ones.

The simplest use of `curl` does an implicit `GET` (output truncated here):

```
$ curl http://www.example.com
<!doctype html>
<html>
<head>
    <title>Example Domain</title>
    ...
```

This uses `HEAD`:

```
$ curl --head http://www.example.com
HTTP/1.1 200 OK
Content-Encoding: gzip
Accept-Ranges: bytes
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Sun, 05 May 2019 16:14:30 GMT
Etag: "1541025663"
```

```
Expires: Sun, 12 May 2019 16:14:30 GMT
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
Server: ECS (agb/52B1)
X-Cache: HIT
Content-Length: 606
```

If you're passing arguments, you can include them in the command line or a data file. In these examples, I use the following:

- *url* for any website
- `data.txt` as a text data file with these contents: `a=1&b=2`
- `data.json` as a JSON data file with these contents: `{"a":1, "b": 2}`
- `a=1&b=2` as two data arguments

Using default (*form-encoded*) arguments:

```
$ curl -X POST -d "a=1&b=2" url
$ curl -X POST -d "@data.txt" url
```

For JSON-encoded arguments:

```
$ curl -X POST -d "{'a':1,'b':2}" -H "Content-Type: application/json" url
$ curl -X POST -d "@data.json" url
```

Test with httpie

A more Pythonic alternative to curl is `httpie`.

```
$ pip install httpie
```

To make a form-encoded POST, similar to the methods for `curl` above (`-f` is a synonym for `--form`):

```
$ http -f POST url a=1 b=2
$ http POST -f url < data.txt
```

The default encoding is JSON:

```
$ http POST url a=1 b=2
$ http POST url < data.json
```

`httpie` also handles HTTP headers, cookies, file uploads, authentication, redirects, SSL, and so on. As usual, see the docs

Test with httpbin

You can test your web queries against the site `httpbin`, or download and run the site in a local Docker image:

```
$ docker run -p 80:80 kennethreitz/httpbin
```

Python's Standard Web Libraries

In Python 2, web client and server modules were a bit scattered. One of the Python 3 goals was to bundle these modules into two *packages* (remember from Chapter 11 that a package is just a directory containing module files):

- `http` manages all the client-server HTTP details:
  - `client` does the client-side stuff
  - `server` helps you write Python web servers
  - `cookies` and `cookiejar` manage cookies, which save data between site visits
- `urllib` runs on top of `http`:
  - `request` handles the client request
  - `response` handles the server response
  - `parse` cracks the parts of a URL

NOTE

If you're trying to write code that's compatible with both Python 2 and Python 3, keep in mind that `urllib` changed a lot between the two versions. For a better alternative, refer to "Beyond the Standard Library: requests".

Let's use the standard library to get something from a website. The URL in the following example returns information from a test website:

```
>>> import urllib.request as ur
>>>
>>> url = 'http://www.example.com/'
>>> conn = ur.urlopen(url)
```

This little chunk of Python opened a TCP/IP connection to the remote web server `www.example.com`, made an HTTP request, and received an HTTP response. The response contained more than just the page data. In the official documentation, we find that `conn` is an `HTTPResponse` object with a number of methods and attributes. One of the most important parts of the response is the HTTP *status code*:

```
>>> print(conn.status)
200
```

A `200` means that everything was peachy. There are dozens of HTTP status codes, grouped into five ranges by their first (hundreds) digit:

1xx (information)
: The server received the request but has some extra information for the client.

2xx (success)
: It worked; every success code other than 200 conveys extra details.

3xx (redirection)
: The resource moved, so the response returns the new URL to the client.

4xx (client error)
: Some problem from the client side, such as the well-known 404 (not found). 418 (*I'm a teapot*) was an April Fool's joke.

5xx (server error)
: 500 is the generic whoops; you might see a 502 (bad gateway) if there's some disconnect between a web server and a backend application server.

To get the actual data contents from the web page, use the `read()` method of the `conn` variable. This returns a `bytes` value. Let's get the data and print the first 50 bytes:

```
>>> data = conn.read()
>>> print(data[:50])
```

```
b'<!doctype html>\n<html>\n<head>\n    <title>Example D'
```

We can convert these bytes to a string and print its first 50 characters:

```
>>> str_data = data.decode('utf8')
>>> print(str_data[:50])
<!doctype html>
<html>
<head>
    <title>Example D
>>>
```

The rest is more HTML and CSS.

Out of sheer curiosity, what HTTP headers were sent back to us?

```
>>> for key, value in conn.getheaders():
...     print(key, value)
...
```

```
Cache-Control max-age=604800
Content-Type text/html; charset=UTF-8
Date Sun, 05 May 2019 03:09:26 GMT
Etag "1541025663+ident"
Expires Sun, 12 May 2019 03:09:26 GMT
Last-Modified Fri, 09 Aug 2013 23:54:35 GMT
Server ECS (agb/5296)
Vary Accept-Encoding
X-Cache HIT
Content-Length 1270
Connection close
```

Remember that `telnet` example a little earlier? Now, our Python library is parsing all those HTTP response headers and providing them in a dictionary. `Date` and `Server` seem straightforward; some of the others, less so. It's helpful to know that HTTP has a set of standard headers such as `Content-Type`, and many optional ones.

Beyond the Standard Library: requests

At the beginning of Chapter 1, there was a program that accessed a Wayback Machine API by using the standard libraries `urllib.request` and `json`. Following that example is a version that uses the third-party module `requests`. The `requests` version is shorter and easier to understand.

For most purposes, I think web client development with `requests` is easier. You can browse the documentation (which is pretty good) for full details. I'll show the basics of `requests` in this section and use it throughout this book for web client tasks.

First, install the `requests` library:

```
$ pip install requests
```

Now, let's redo our example.com query with `requests`:

```
>>> import requests
>>> resp = requests.get('http://example.com')
>>> resp
<Response [200]>
>>> resp.status_code
200
>>> resp.text[:50]
'<!doctype html>\n<html>\n<head>\n    <title>Example D'
```

To show a JSON query, here's a minimal version of a program that appears at the end of this chapter. You provide a string, and it uses an Internet Archive search API to look through the titles of billions of multimedia items saved there. Notice that in the `requests.get()` call shown in Example 18-1, you only need to pass a `params` dictionary, and `requests` handles all the query construction and character escaping.

Example 18-1. ia.py

```python
import json
import sys

import requests

def search(title):
    url = "http://archive.org/advancedsearch.php"
    params = {"q": f"title:({title})",
              "output": "json",
              "fields": "identifier,title",
              "rows": 50,
              "page": 1,}
    resp = requests.get(url, params=params)
    return resp.json()

if __name__ == "__main__":
    title = sys.argv[1]
    data = search(title)
    docs = data["response"]["docs"]
    print(f"Found {len(docs)} items, showing first 10")
    print("identifier\ttitle")
    for row in docs[:10]:
        print(row["identifier"], row["title"], sep="\t")
```

How’s their stock of wendigo items?

```
$ python ia.py wendigo
Found 24 items, showing first 10
identifier  title
cd_wendigo_penny-sparrow  Wendigo
Wendigo1  Wendigo 1
wendigo_ag_librivox The Wendigo
thewendigo10897gut  The Wendigo
isbn_9780843944792  Wendigo mountain ; Death camp
jamendo-060508  Wendigo - Audio Leash
fav-lady_wendigo  lady_wendigo Favorites
011bFearTheWendigo  011b Fear The Wendigo
CharmedChats112 Episode 112 - The Wendigo
jamendo-076964  Wendigo - Tomame o Dejame>
```

The first column (the *identifier*) can be used to actually view the item at the *archive.org* site. You'll see how to do this at the end of this chapter.

Web Servers

Web developers have found Python to be an excellent language for writing web servers and server-side programs. This has led to such a variety of Python-based web *frameworks* that it can be hard to navigate among them and make choices—not to mention deciding what deserves to go into a book.

A web framework provides features with which you can build websites, so it does more than a simple web (HTTP) server. You'll see features such as routing (URL to server function), templates (HTML with dynamic inclusions), debugging, and more.

I'm not going to cover all of the frameworks here—just those that I've found to be relatively simple to use and suitable for real websites. I'll also show how to run the dynamic parts of a website with Python and other parts with a traditional web server.

The Simplest Python Web Server

You can run a simple web server by typing just one line of Python:

```
$ python -m http.server
```

This implements a bare-bones Python HTTP server. If there are no problems, this will print an initial status message:

```
Serving HTTP on 0.0.0.0 port 8000 ...
```

That `0.0.0.0` means *any TCP address*, so web clients can access it no matter what address the server has. There's more low-level details on TCP and other network plumbing for you to read about in Chapter 17.

You can now request files, with paths relative to your current directory, and they will be returned. If you type `http://localhost:8000` in your web browser, you should see a directory listing there, and the server will print access log lines such as this:

```
127.0.0.1 - - [20/Feb/2013 22:02:37] "GET / HTTP/1.1" 200 -
```

`localhost` and `127.0.0.1` are TCP synonyms for *your local computer*, so this works regardless of whether you're connected to the internet. You can interpret this line as follows:

- `127.0.0.1` is the client's IP address
- The first `-` is the remote username, if found
- The second `-` is the login username, if required
- `[20/Feb/2013 22:02:37]` is the access date and time
- `"GET / HTTP/1.1"` is the command sent to the web server:
  - The HTTP method (`GET`)
  - The resource requested (`/` , the top)
  - The HTTP version (`HTTP/1.1`)
- The final `200` is the HTTP status code returned by the web server

Click any file. If your browser can recognize the format (HTML, PNG, GIF, JPEG, and so on) it should display it, and the server will log the request. For instance, if you have the file *oreilly.png* in your current directory, a request for *http://localhost:8000/oreilly.png* should return the image of the unsettling fellow in Figure 20-2, and the log should show something such as this:

```
127.0.0.1 - - [20/Feb/2013 22:03:48] "GET /oreilly.png HTTP/1.1" 200 -
```

If you have other files in the same directory on your computer, they should show up in a listing on your display, and you can click any one to download it. If your browser is configured to display that file's format, you'll see the results on your screen; otherwise, your browser will ask you if you want to download and save the file.

The default port number used is 8000, but you can specify another:

```
$ python -m http.server 9999
```

You should see this:

```
Serving HTTP on 0.0.0.0 port 9999 ...
```

This Python-only server is best suited for quick tests. You can stop it by killing its process; in most terminals, press Ctrl+C.

You should not use this basic server for a busy production website. Traditional web servers such as Apache and NGINX are much faster for serving static files. In addition, this simple server has no way to handle dynamic content, which more extensive servers can do by accepting parameters.

Web Server Gateway Interface (WSGI)

All too soon, the allure of serving simple files wears off, and we want a web server that can also run programs dynamically. In the early days of the web, the *Common Gateway Interface* (CGI) was designed for clients to make web servers run external programs and return the results. CGI also handled getting input arguments from the client through the server to the external programs. However, the programs were started anew for *each* client access. This could not scale well, because even small programs have appreciable startup time.

To avoid this startup delay, people began merging the language interpreter into the web server. Apache ran PHP within its `mod_php` module, Perl in `mod_perl`, and Python in `mod_python`. Then, code in these dynamic languages could be executed within the long-running Apache process itself rather than in external programs.

An alternative method was to run the dynamic language within a separate long-running program and have it communicate with the web server. FastCGI and SCGI are examples.

Python web development made a leap with the definition of the *Web Server Gateway Interface* (WSGI), a universal API between Python web applications and web servers. All of the Python web frameworks and web servers in the rest of this chapter use WSGI. You don’t normally need to know how WSGI works (there really isn’t much to it), but it helps to know what some of the parts under the hood are called. This is a *synchronous* connection—one step follows another.

ASGI

In a few places so far, I've mentioned that Python has been introducing *asynchronous* language features like `async`, `await`, and `asyncio`. ASGI (Asynchronous Server Gateway Interface) is a counterpart of WSGI that uses these new features. In Appendix C, you'll see more discussion, and examples of new web frameworks that use ASGI.

Apache

The `apache` web server's best WSGI module is `mod_wsgi`. This can run Python code within the Apache process or in separate processes that communicate with Apache.

You should already have `apache` if your system is Linux or macOS. For Windows, you'll need to install apache.

Finally, install your preferred WSGI-based Python web framework. Let's try `bottle` here. Almost all of the work involves configuring Apache, which can be a dark art.

Create the test file shown in Example 18-2 and save it as */var/www/test/home.wsgi*.

Example 18-2. home.wsgi

```
import bottle

application = bottle.default_app()

@bottle.route('/')
def home():
    return "apache and wsgi, sitting in a tree"
```

Do not call `run()` this time, because that starts the built-in Python web server. We need to assign to the variable `application` because that's what `mod_wsgi` looks for to marry the web server and the Python code.

If `apache` and its `mod_wsgi` module are working correctly, we just need to connect them to our Python script. We want to add one line to the file that defines the default website for this `apache` server, but finding that file is a task itself. It could be */etc/apache2/httpd.conf*, or */etc/apache2/sites-available/default*, or the Latin name of someone's pet salamander.

Let's assume for now that you understand `apache` and found that file. Add this line inside the `<VirtualHost>` section that governs the default website:

```
    WSGIScriptAlias / /var/www/test/home.wsgi
```

That section might then look like this:

```
<VirtualHost *:80>
    DocumentRoot /var/www

    WSGIScriptAlias / /var/www/test/home.wsgi

    <Directory /var/www/test>
    Order allow,deny
    Allow from all
    </Directory>
</VirtualHost>
```

Start `apache`, or restart it if it was running to make it use this new configuration. If you then browse to *http://localhost/*, you should see:

```
apache and wsgi, sitting in a tree
```

This runs `mod_wsgi` in *embedded mode*, as part of `apache` itself.

You can also run it in *daemon mode*, as one or more processes, separate from `apache`. To do this, add two new directive lines to your `apache` config file:

```
WSGIDaemonProcess domain-name user=user-name group=group-name threads=25
WSGIProcessGroup domain-name
```

In the preceding example, `user-name` and `group-name` are the operating system user and group names, and the `domain-name` is the name of your internet domain. A minimal `apache` config might look like this:

```
<VirtualHost *:80>
    DocumentRoot /var/www

    WSGIScriptAlias / /var/www/test/home.wsgi

    WSGIDaemonProcess mydomain.com user=myuser group=mygroup threads=25
    WSGIProcessGroup mydomain.com

    <Directory /var/www/test>
    Order allow,deny
    Allow from all
    </Directory>
</VirtualHost>
```

NGINX

The `NGINX` web server does not have an embedded Python module. Instead, it's a frontend to a separate WSGI server such as uWSGI or gUnicorn. Together they make a very fast and configurable platform for Python web development.

You can install `nginx` from its website. For examples of setting up Flask with NGINX and a WSGI server, see this.

Other Python Web Servers

Following are some of the independent Python-based WSGI servers that work like `apache` or `nginx`, using multiple processes and/or threads (see "Concurrency") to handle simultaneous requests:

- `uwsgi`
- `cherrypy`
- `pylons`

Here are some *event-based* servers, which use a single process but avoid blocking on any single request:

- `tornado`
- `gevent`
- `gunicorn`

I have more to say about events in the discussion about *concurrency* in Chapter 15.

Web Server Frameworks

Web servers handle the HTTP and WSGI details, but you use web *frameworks* to actually write the Python code that powers the site. So, let's talk about frameworks for a while and then get back to alternative ways of actually serving sites that use them.

If you want to write a website in Python, there are many (some say too many) Python web frameworks. A web framework handles, at a minimum, client requests and server responses. Most major web frameworks include these tasks:

- HTTP protocol handling
- Authentication (*authn*, or who are you?)
- Authorization (*authz*, or what can you do?)
- Establish a session
- Get parameters
- Validate parameters (required/optional, type, range)
- Handle HTTP verbs
- Route (functions/classes)
- Serve static files (HTML, JS, CSS, images)
- Serve dynamic data (databases, services)
- Return values and HTTP status

Optional features include:

- Backend templates
- Database connectivity, ORMs
- Rate limiting
- Asynchronous tasks

In the coming sections, we write example code for two frameworks

(`bottle` and `flask`). These are *synchronous*. Later, I talk about alternatives, especially for database-backed websites. You can find a Python framework to power any site that you can think of.

Bottle

Bottle consists of a single Python file, so it's very easy to try out, and it's easy to deploy later. Bottle isn't part of standard Python, so to install it, type the following command:

```
$ pip install bottle
```

Here's code that will run a test web server and return a line of text when your browser accesses the URL *http://localhost:9999/*. Save it as *bottle1.py* (Example 18-3).

Example 18-3. bottle1.py

```
from bottle import route, run

@route('/')
def home():
  return "It isn't fancy, but it's my home page"

run(host='localhost', port=9999)
```

Bottle uses the `route` decorator to associate a URL with the following function; in this case, `/` (the home page) is handled by the `home()` function. Make Python run this server script by typing this:

```
$ python bottle1.py
```

You should see this on your browser when you access *http://localhost:9999/*:

```
It isn't fancy, but it's my home page
```

The `run()` function executes `bottle`'s built-in Python test web server.

You don't need to use this for `bottle` programs, but it's useful for initial development and testing.

Now, instead of creating text for the home page in code, let's make a separate HTML file called *index.html* that contains this line of text:

```
My <b>new</b> and <i>improved</i> home page!!!
```

Make `bottle` return the contents of this file when the home page is requested. Save this script as *bottle2.py* (Example 18-4).

Example 18-4. bottle2.py

```
from bottle import route, run, static_file

@route('/')
def main():
    return static_file('index.html', root='.')

run(host='localhost', port=9999)
```

In the call to `static_file()`, we want the file `index.html` in the directory indicated by `root` (in this case, `'.'`, the current directory). If your previous server example code was still running, stop it. Now, run the new server:

```
$ python bottle2.py
```

When you ask your browser to get *http:/localhost:9999/*, you should see this:

```
My new and improved home page!!!
```

Let's add one last example that shows how to pass arguments to a URL and use them. Of course, this will be *bottle3.py*, which you can see in Example 18-5.

Example 18-5. bottle3.py

```
from bottle import route, run, static_file

@route('/')
def home():
    return static_file('index.html', root='.')

@route('/echo/<thing>')
def echo(thing):
    return "Say hello to my little friend: %s!" % thing

run(host='localhost', port=9999)
```

We have a new function called `echo()` and want to pass it a string argument in a URL. That's what the line `@route('/echo/<thing>')` in the preceding example does. That `<thing>` in the route means that whatever was in the URL after `/echo/` is assigned to the string argument `thing`, which is then passed to the `echo` function. To see what happens, stop the old server if it's still running and then start it with the new code:

```
$ python bottle3.py
```

Then, access *http://localhost:9999/echo/Mothra* in your web browser. You should see the following:

```
Say hello to my little friend: Mothra!
```

Now, leave *bottle3.py* running for a minute so that we can try something else. You've been verifying that these examples work by typing URLs into your browser and looking at the displayed pages. You can also use client libraries such as `requests` to do your work for you. Save this as *bottle_test.py* (Example 18-6).

Example 18-6. bottle_test.py

```
import requests

resp = requests.get('http://localhost:9999/echo/Mothra')
if resp.status_code == 200 and \
  resp.text == 'Say hello to my little friend: Mothra!':
    print('It worked! That almost never happens!')
```

```
else:
    print('Argh, got this:', resp.text)
```

Great! Now, run it:

```
$ python bottle_test.py
```

You should see this in your terminal:

```
It worked! That almost never happens!
```

This is a little example of a *unit test*. Chapter 19 provides more details on why tests are good and how to write them in Python.

There's more to Bottle than I've shown here. In particular, you can try adding these arguments when you call `run()`:

- `debug=True` creates a debugging page if you get an HTTP error;
- `reloader=True` reloads the page in the browser if you change any of the Python code.

It's well documented at the developer site.

Flask

Bottle is a good initial web framework. If you need a few more cowbells and whistles, try Flask. It started in 2010 as an April Fools' joke, but enthusiastic response encouraged the author, Armin Ronacher, to make it a real framework. He named the result Flask as a wordplay on Bottle.

Flask is about as simple to use as Bottle, but it supports many extensions that are useful in professional web development, such as Facebook authentication and database integration. It's my personal favorite among Python web frameworks because it balances ease of use with a rich feature set.

The `flask` package includes the `werkzeug` WSGI library and the `jinja2` template library. You can install it from a terminal:

```
$ pip install flask
```

Let's replicate the final Bottle example code in Flask. First, though, we need to make a few changes:

- Flask's default directory home for static files is `static`, and URLs for files there also begin with `/static`. We change the folder to `'.'` (current directory) and the URL prefix to `''` (empty) to allow the URL `/` to map to the file *index.html*.
- In the `run()` function, setting `debug=True` also activates the automatic reloader; `bottle` used separate arguments for debugging and reloading.

Save this file to *flask1.py* (Example 18-7).

Example 18-7. flask1.py

```
from flask import Flask

app = Flask(__name__, static_folder='.', static_url_path='')

@app.route('/')
def home():
    return app.send_static_file('index.html')

@app.route('/echo/<thing>')
def echo(thing):
    return "Say hello to my little friend: %s" % thing

app.run(port=9999, debug=True)
```

Then, run the server from a terminal or window:

```
$ python flask1.py
```

Test the home page by typing this URL into your browser:

```
http://localhost:9999/
```

You should see the following (as you did for `bottle`):

```
My new and improved home page!!!
```

Try the `/echo` endpoint:

```
http://localhost:9999/echo/Godzilla
```

You should see this:

```
Say hello to my little friend: Godzilla
```

There's another benefit to setting `debug` to `True` when calling `run`. If an exception occurs in the server code, Flask returns a specially formatted page with useful details about what went wrong, and where. Even better, you can type some commands to see the values of variables in the server program.

WARNING

Do not set `debug = True` in production web servers. It exposes too much information about your server to potential intruders.

So far, the Flask example just replicates what we did with Bottle. What can Flask do that Bottle can't? Flask includes `jinja2`, a more extensive templating system. Here's a tiny example of how to use `jinja2` and Flask together.

Create a directory called `templates` and a file within it called *flask2.html* (Example 18-8).

Example 18-8. flask2.html

```
<html>
<head>
<title>Flask2 Example</title>
</head>
<body>
```

```
Say hello to my little friend: {{ thing }}
</body>
</html>
```

Next, we write the server code to grab this template, fill in the value of *thing* that we passed it, and render it as HTML (I'm dropping the `home()` function here to save space). Save this as *flask2.py* (Example 18-9).

Example 18-9. flask2.py

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/echo/<thing>')
def echo(thing):
    return render_template('flask2.html', thing=thing)

app.run(port=9999, debug=True)
```

That `thing = thing` argument means to pass a variable named `thing` to the template, with the value of the string `thing` .

Ensure that *flask1.py* isn't still running and then start *flask2.py*:

```
$ python flask2.py
```

Now, type this URL:

```
http://localhost:9999/echo/Gamera
```

You should see the following:

```
Say hello to my little friend: Gamera
```

Let's modify our template and save it in the *templates* directory as *flask3.html*:

```
<html>
```

```
<head>
<title>Flask3 Example</title>
</head>
<body>
Say hello to my little friend: {{ thing }}.
Alas, it just destroyed {{ place }}!
</body>
</html>
```

You can pass this second argument to the `echo` URL in many ways.

Pass an argument as part of the URL path

Using this method, you simply extend the URL itself. Save the code shown in Example 18-10 as *flask3a.py*.

Example 18-10. flask3a.py

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/echo/<thing>/<place>')
def echo(thing, place):
    return render_template('flask3.html', thing=thing, place=place)

app.run(port=9999, debug=True)
```

As usual, stop the previous test server script if it's still running and then try this new one:

```
$ python flask3a.py
```

The URL would look like this:

```
http://localhost:9999/echo/Rodan/McKeesport
```

And you should see the following:

```
Say hello to my little friend: Rodan. Alas, it just destroyed McKeesport!
```

Or, you can provide the arguments as `GET` parameters, as shown in Example 18-11; save this as *flask3b.py*.

Example 18-11. flask3b.py

```
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route('/echo/')
def echo():
    thing = request.args.get('thing')
    place = request.args.get('place')
    return render_template('flask3.html', thing=thing, place=place)

app.run(port=9999, debug=True)
```

Run the new server script:

```
$ python flask3b.py
```

This time, use this URL:

```
http://localhost:9999/echo?thing=Gorgo&place=Wilmerding
```

You should get back what you see here:

```
Say hello to my little friend: Gorgo. Alas, it just destroyed Wilmerding!
```

When a `GET` command is used for a URL, any arguments are passed in the form `&` *`key1`* `=` *`val1`* `&` *`key2`* `=` *`val2`* `&...`

You can also use the dictionary `**` operator to pass multiple arguments to a template from a single dictionary (call this *flask3c.py*), as shown in Example 18-12.

Example 18-12. flask3c.py

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

```
app = Flask(__name__)

@app.route('/echo/')
def echo():
    kwargs = {}
    kwargs['thing'] = request.args.get('thing')
    kwargs['place'] = request.args.get('place')
    return render_template('flask3.html', **kwargs)

app.run(port=9999, debug=True)
```

That `**kwargs` acts like `thing=thing, place=place`. It saves some typing if there are a lot of input arguments.

The `jinja2` templating language does a lot more than this. If you've programmed in PHP, you'll see many similarities.

Django

`Django` is a very popular Python web framework, especially for large sites. It's worth learning for many reasons, including frequent requests for `django` experience in Python job ads. It includes ORM code (we talked about ORMs in "The Object-Relational Mapper (ORM)") to create automatic web pages for the typical database *CRUD* functions (create, replace, update, delete) that we looked at in Chapter 16. It also includes some automatic admin pages for these, but they're designed for internal use by programmers rather than public web page use. You don't have to use `Django`'s ORM if you prefer another, such as SQLAlchemy, or direct SQL queries.

Other Frameworks

You can compare the frameworks by viewing this online table:

- `fastapi` handles both synchronous (WSGI) and asynchronous (ASGI) calls, uses type hints, generates test pages, and is well documented. Recommended.
- `web2py` covers much the same ground as `django`, with a different style.

- `pyramid` grew from the earlier `pylons` project, and is similar to `django` in scope.
- `turbogears` supports an ORM, many databases, and multiple template languages.
- `wheezy.web` is a newer framework optimized for performance. It was faster than the others in a recent test.
- `molten` also uses type hints, but only supports WSGI.
- `apistar` is similar to fastapi, but is more of an API validation tool than a web framework.
- `masonite` is a Python version of Ruby on Rails, or PHP's Laravel.

Database Frameworks

The web and databases are the peanut butter and jelly of computing: where you find one, you'll eventually find the other. In real-life Python applications, at some point you'll probably need to provide a web interface (site and/or API) to a relational database.

You could build your own with:

- A web framework like Bottle or Flask
- A database package, like db-api or SQLAlchemy
- A database driver, like pymysql

Instead, you could use a web/database package like one of these:

- connexion
- datasette
- sandman2
- flask-restless

Or, you could use a framework with built-in database support, like Django.

Your database may not be a relational one. If your data schema varies significantly—columns that differ markedly across rows—it might be worthwhile to consider a *schemaless* database, such as one of the *NoSQL* databases discussed in Chapter 16. I once worked on a site that initially stored its data in a NoSQL database, switched to a relational one, on to another relational one, to a different NoSQL one, and then finally back to one of the relational ones.

Web Services and Automation

We've just looked at traditional web client and server applications, consuming and generating HTML pages. Yet the web has turned out to be a powerful way to glue applications and data in many more formats than HTML.

webbrowser

Let's start begin a little surprise. Start a Python session in a terminal window and type the following:

```
>>> import antigravity
```

This secretly calls the standard library's `webbrowser` module and directs your browser to an enlightening Python link.[3]

You can use this module directly. This program loads the main Python site's page in your browser:

```
>>> import webbrowser
>>> url = 'http://www.python.org/'
>>> webbrowser.open(url)
True
```

This opens it in a new window:

```
>>> webbrowser.open_new(url)
True
```

And this opens it in a new tab, if your browser supports tabs:

```
>>> webbrowser.open_new_tab('http://www.python.org/')
True
```

The `webbrowser` makes your browser do all the work.

webview

Rather than calling your browser as `webbrowser` does, `webview` displays the page in its own window, using your machine's native GUI. To install on Linux or macOS:

```
$ pip install pywebview[qt]
```

For Windows:

```
$ pip install pywebview[cef]
```

See the installation notes if you have problems.

Here's an example in which I gave it the official US government current time site:

```
>>> import webview
>>> url = input("URL? ")
URL? http://time.gov
>>> webview.create_window(f"webview display of {url}", url)
```

Figure 18-1 shows the result I got back.

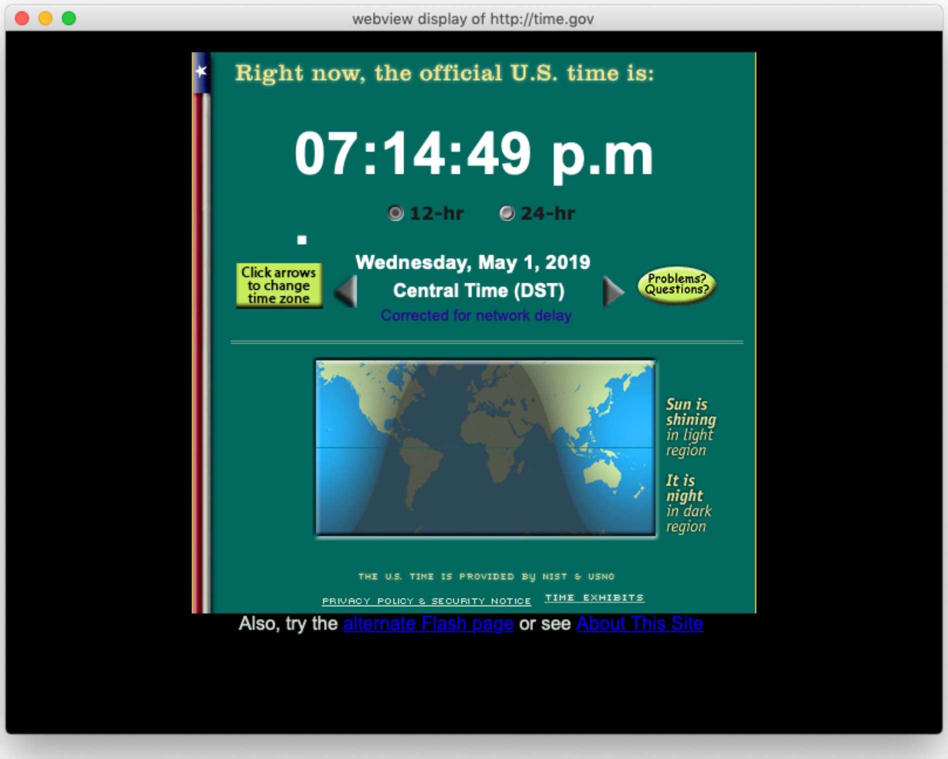

webview display of http://time.gov
Right now, the official U.S. time is:
07:14:49 p.m
12-hr
24-hr
Click arrows to change time zone
Wednesday, May 1, 2019
Central Time (DST)
Corrected for network delay
Problems? Questions?
Sun is shining in light region
It is night in dark region
THE U.S. TIME IS PROVIDED BY NIST & USNO
PRIVACY POLICY & SECURITY NOTICE
TIME EXHIBITS
Also, try the alternate Flash page or see About This Site

Figure 18-1. `webview` display window

To stop the program, kill the display window.

Web APIs and REST

Often, data is available only within web pages. If you want to access it, you need to access the pages through a web browser and read it. If the authors of the website made any changes since the last time you visited, the location and style of the data might have changed.

Instead of publishing web pages, you can provide data through a web *application programming interface* (API). Clients access your service by making requests to URLs and getting back responses containing status and data. Instead of HTML pages, the data is in formats that are easier for programs to consume, such as JSON or XML (refer to Chapter 16 for more about these formats).

Representational State Transfer (REST) was defined by Roy Fielding in his doctoral thesis. Many products claim to have a *REST interface* or a *RESTful interface*. In practice, this often only means that they have a *web* interface—definitions of URLs to access a web service.

A *RESTful* service uses the HTTP *verbs* in specific ways:

- `HEAD` gets information about the resource, but not its data.
- `GET` retrieves the resource's data from the server. This is the standard method used by your browser. `GET` should not be used to create, change, or delete data.
- `POST` creates a new resource.
- `PUT` replaces an existing resource, creating it if it doesn't exist.
- `PATCH` partially updates a resource.
- `DELETE` deletes. Truth in advertising!

A RESTful client can also request one or more content types from the server by using HTTP request headers. For example, a complex service with a REST interface might prefer its input and output to be JSON strings.

Crawl and Scrape

Sometimes, you might want a little bit of information—a movie rating, stock price, or product availability—but what you need is available only in HTML pages, surrounded by ads and extraneous content.

You could extract what you're looking for manually by doing the following:

1. Type the URL into your browser.
2. Wait for the remote page to load.
3. Look through the displayed page for the information you want.
4. Write it down somewhere.
5. Possibly repeat the process for related URLs.

However, it's much more satisfying to automate some or all of these steps. An automated web fetcher is called a *crawler* or *spider*.[4] After the contents have been retrieved from the remote web servers, a *scraper* parses it to find the needle in the haystack.

Scrapy

If you need an industrial-strength combined crawler *and* scraper, Scrapy is worth downloading:

```
$ pip install scrapy
```

This installs the module and a standalone command-line `scrapy` program.

Scrapy is a framework, not just a module such as `BeautifulSoup`. It does more, but it's more complex to set up. To learn more about Scrapy, read "Scrapy at a Glance" and the tutorial.

BeautifulSoup

If you already have the HTML data from a website and just want to extract data from it, `BeautifulSoup` is a good choice. HTML parsing is harder than it sounds. This is because much of the HTML on public web pages is technically invalid: unclosed tags, incorrect nesting, and other complications. If you try to write your own HTML parser by using regular expressions (discussed in "Text Strings: Regular Expressions") you'll soon encounter these messes.

To install `BeautifulSoup`, type the following command (don't forget the final `4`, or `pip` will try to install an older version and probably fail):

```
$ pip install beautifulsoup4
```

Now, let's use it to get all the links from a web page. The HTML `a` element represents a link, and `href` is its attribute representing the link destination. In Example 18-13, we'll define the function `get_links()` to do the grunt work, and a main program to get one or more URLs as command-line arguments.

Example 18-13. links.py

```
def get_links(url):
    import requests
    from bs4 import BeautifulSoup as soup
    result = requests.get(url)
    page = result.text
    doc = soup(page)
    links = [element.get('href') for element in doc.find_all('a')]
```

```
        return links

if __name__ == '__main__':
    import sys
    for url in sys.argv[1:]:
        print('Links in', url)
        for num, link in enumerate(get_links(url), start=1):
            print(num, link)
        print()
```

I saved this program as *links.py* and then ran this command:

```
$ python links.py http://boingboing.net
```

Here are the first few lines that it printed:

```
Links in http://boingboing.net/
1 http://boingboing.net/suggest.html
2 http://boingboing.net/category/feature/
3 http://boingboing.net/category/review/
4 http://boingboing.net/category/podcasts
5 http://boingboing.net/category/video/
6 http://bbs.boingboing.net/
7 javascript:void(0)
8 http://shop.boingboing.net/
9 http://boingboing.net/about
10 http://boingboing.net/contact
```

Requests-HTML

Kenneth Reitz, the author of the popular web client package `requests`, has written a new scraping library called requests-html (for Python 3.6 and newer versions).

It gets a page and processes its elements, so you can find, for example, all of its links, or all the contents or attributes of any HTML element.

It has a clean design, similar to `requests` and other packages by the same author. Overall, it may be easier to use than `beautifulsoup` or Scrapy.

Let’s Watch a Movie

Let’s build a full program.

It searches for videos using an API at the Internet Archive.[5] This is one of the few APIs that allows anonymous access *and* should still be around after this book is printed.

NOTE

Most web APIs require you to first get an *API key*, and provide it every time you access that API. Why? It’s the tragedy of the commons: free resources with anonymous access are often overused or abused. That’s why we can’t have nice things.

The following program shown in Example 18-14 does the following:

- Prompts you for part of a movie or video title
- Searches for it at the Internet Archive
- Returns a list of identifiers, names, and descriptions
- Lists them and asks you to select one
- Displays that video in your web browser

Save this as *iamovies.py*.

The `search()` function uses `requests` to access the URL, get the results, and convert them to JSON. The other functions handle everything else. You’ll see usage of list comprehensions, string slices, and other things that you’ve seen in previous chapters. (The line numbers are not part of the source; they’ll be used in the exercises to locate code pieces.)

Example 18-14. iamovies.py

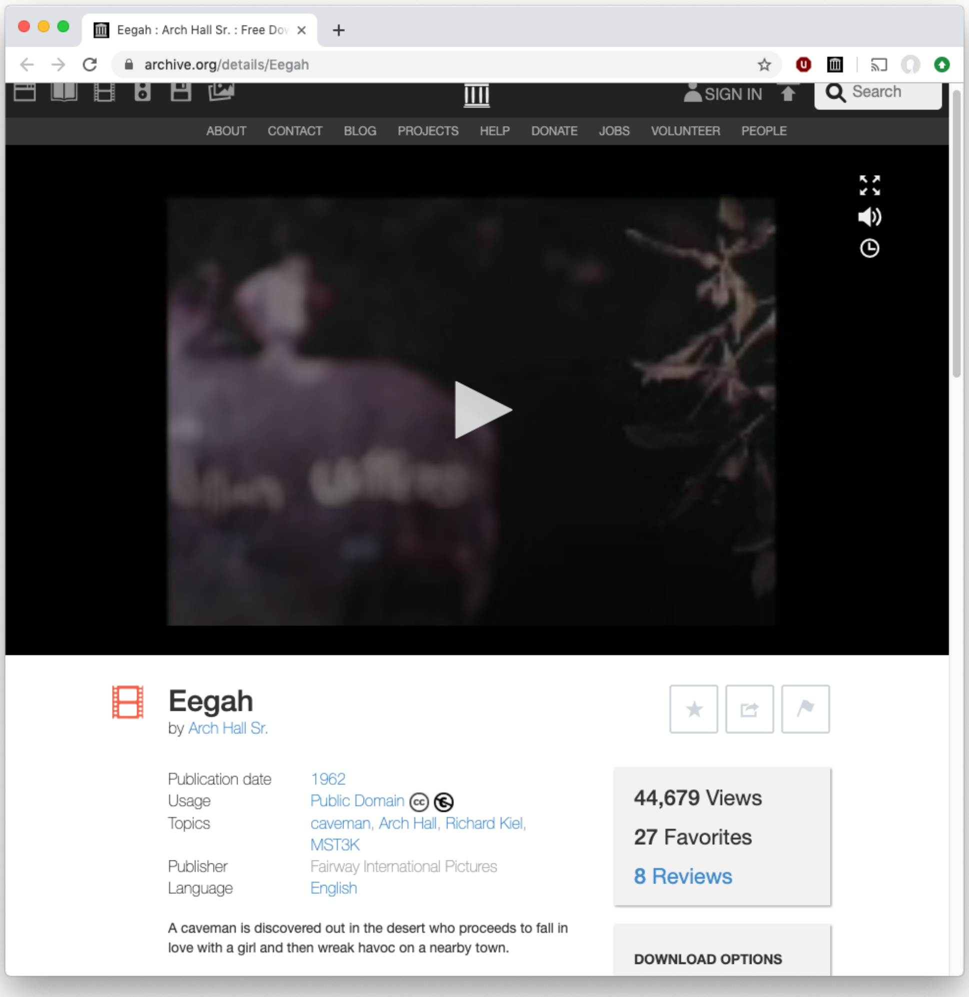
```
1 """Find a video at the Internet Archive
2 by a partial title match and display it."""
3
4 import sys
5 import webbrowser
6 import requests
7
```

```python
def search(title):
    """Return a list of 3-item tuples (identifier,
       title, description) about videos
       whose titles partially match :title."""
    search_url = "https://archive.org/advancedsearch.php"
    params = {
        "q": "title:({}) AND mediatype:(movies)".format(title),
        "fl": "identifier,title,description",
        "output": "json",
        "rows": 10,
        "page": 1,
        }
    resp = requests.get(search_url, params=params)
    data = resp.json()
    docs = [(doc["identifier"], doc["title"], doc["description"])
            for doc in data["response"]["docs"]]
    return docs

def choose(docs):
    """Print line number, title and truncated description for
       each tuple in :docs. Get the user to pick a line
       number. If it's valid, return the first item in the
       chosen tuple (the "identifier"). Otherwise, return None."""
    last = len(docs) - 1
    for num, doc in enumerate(docs):
        print(f"{num}: ({doc[1]}) {doc[2][:30]}...")
    index = input(f"Which would you like to see (0 to {last})? ")
    try:
        return docs[int(index)][0]
    except:
        return None

def display(identifier):
    """Display the Archive video with :identifier in the browser"""
    details_url = "https://archive.org/details/{}".format(identifier)
    print("Loading", details_url)
    webbrowser.open(details_url)

def main(title):
    """Find any movies that match :title.
       Get the user's choice and display it in the browser."""
    identifiers = search(title)
    if identifiers:
        identifier = choose(identifiers)
        if identifier:
            display(identifier)
        else:
            print("Nothing selected")
    else:
        print("Nothing found for", title)

if __name__ == "__main__":
    main(sys.argv[1])
```

Here’s what I got when I ran this program and searched for `eegah`:[6]

```
$ python iamovies.py eegah
0: (Eegah) From IMDb : While driving thro...
1: (Eegah) This film has fallen into the ...
2: (Eegah) A caveman is discovered out in...
3: (Eegah (1962)) While driving through the dese...
4: (It's "Eegah" - Part 2) Wait till you see how this end...
5: (EEGAH trailer) The infamous modern-day cavema...
6: (It's "Eegah" - Part 1) Count Gore De Vol shares some ...
7: (Midnight Movie show: eegah) Arch Hall Jr...
Which would you like to see (0 to 7)? 2
Loading https://archive.org/details/Eegah
```

It displayed the page in my browser, ready to run (Figure 18-2).



Eegah
by Arch Hall Sr.

Publication date	1962
Usage	Public Domain
Topics	caveman, Arch Hall, Richard Kiel, MST3K
Publisher	Fairway International Pictures
Language	English

A caveman is discovered out in the desert who proceeds to fall in love with a girl and then wreak havoc on a nearby town.

44,679 Views
27 Favorites
8 Reviews

DOWNLOAD OPTIONS

Figure 18-2. Movie search result

Coming Up

The next chapter is an extremely practical one, covering the nuts and bolts of modern Python development. Learn how to become a steely-eyed, card-carrying Pythonista.

Things to Do

18.1 If you haven't installed `flask` yet, do so now. This will also install `werkzeug`, `jinja2`, and possibly other packages.

18.2 Build a skeleton website, using Flask's debug/reload development web server. Ensure that the server starts up for hostname `localhost` on default port `5000`. If your computer is already using port 5000 for something else, use another port number.

18.3 Add a `home()` function to handle requests for the home page. Set it up to return the string `It's alive!`.

18.4 Create a Jinja2 template file called *home.html* with the following contents:

```
<html>
<head>
<title>It's alive!</title>
<body>
I'm of course referring to {{thing}}, which is {{height}} feet tall and {{color}}.
</body>
</html>
```

18.5 Modify your server's `home()` function to use the *home.html* template. Provide it with three `GET` parameters: `thing`, `height`, and `color`.

A company founded

by Steve Jobs dur-ing his ex-ile from Apple.

Let’s kill a *zom-bie lie* here. Senator (and later, Vice President) Al Gore cham-pi-oned bi-par-ti-san leg-is-la-tion and co-op-er-a-tion that greatly advanced the early internet, including funding for the group that wrote Mosaic. He never claimed that he “invented the internet”; that phrase was falsely attributed to him by polit-

i-
cal
ri-
vals
as
he
be-
gan
run-
ning
for
pres-
i-
dent
in
2000.

If
you
don’t
see
it
for
some
rea-
son,
visit
xkcd.

Unappealing
terms
to
arachno-
phobes.

If
you
re-
mem-
ber,

I used an-other Archive API in the main ex-am-ple pro-gram we looked at in Chapter 1.

With Richard Kiel as the cave-man, years be-fore he was Jaws in a Bond movie.