GOVERNMENT ARTS AND SCIENCE COLLEGE, KOMARAPALAYAM

DEPARTMENT OF COMPUTER SCIENCE

COURSE	: M.Sc (COMPUTER SCIENCE)
SEMESTER	: III
SUBJECT NAME	: OPEN SOURCE COMPUTING
SUBJECT CODE	: 19PCS09
HANDLED BY	: Dr.V.KARTHIKEYANI

ASSISTANT PROFESSOR

DEPARTMENT OF COMPUTER SCIENCE

GOVERNMENT ARTS AND SCIENCE COLLEGE

KOMARAPALAYAM – 638 183

Scanned with CamScanner

UNIT-III

Data Types: Text Strings – Binary Data. **Storing and Retrieving Data**: File Input/Output – Structured Text Files – Structured Binary Files - Relational Databases – NoSQL Data Stores.

UNIT-IV

Web: Web Clients – Web Servers – Web Services and Automation – **Systems:** Files – Directories – Programs and Processes – Calendars and Clocks

Scanned with CamScanner

CHAPTER 9
The Web, Untangled

unit 4

Straddling the French-Swiss border is CERN—a particle physics research institute that would seem a good lair for a Bond villain. Luckily, its quest is not world domination but to understand how the universe works. This has always led CERN to generate prodigious amounts of data, challenging physicists and computer scientists just to keep up.

In 1989, the English scientist Tim Berners-Lee first circulated a proposal to help disseminate information within CERN and the research community. He called it the *World Wide Web*, and soon distilled its design into three simple ideas:

HTTP (Hypertext Transfer Protocol)
A specification 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.

He wrote the first web browser and server on a NeXT computer, invented by a short-lived company Steve Jobs founded during his hiatus from Apple 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 NCSA *httpd* server. When I downloaded these and started building sites, I had no idea that the Web and the Internet would soon become part of everyday life. At the time, the Internet was still officially noncommercial; 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 Internet frenzy that was occurring at the time, 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'll 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" on page 280 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 on 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.

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

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

If there is a web server on port 80 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 ...
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/
```

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

```
HTTP/          OK
Date: Sat,   Oct      :  :   GMT
Expires: -
Cache-Control: private, max-age=
Content-Type: text/html; charset=ISO-
Set-Cookie: PREF=ID=  a70e9eb3db9d9:FF= :TM=          :LM=         :S=y...
  expires=Mon,   -Oct-        :  :   GMT;
  path=/;
  domain=.google.com
Set-Cookie: NID=  =hTvtVC7dZJmZzGktlmbwVbNZxPQnaDijCz716B1L56GM9qvsqqeIGb...
  expires=Sun,   -Apr-        :  :   GMT
```


Scanned with CamScanner
Scanned with CamScanner

```
    path=/;
    domain=.google.com;
    HttpOnly
P3P: CP="This is not a P3P policy! See http://www.google.com/support/accounts...
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Alternate-Protocol: 80:quic
Transfer-Encoding: chunked
```

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'll 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
```

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 5 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

Let's use the standard library to get something from a website. The URL in the following example returns a random text quote, similar to a fortune cookie:

```
>>> import urllib.request as ur
>>> url = 'http://www.iheartquotes.com/api/v1/random'
>>> conn = ur.urlopen(url)
>>> print(conn)
<http.client.HTTPResponse object at 0x1006fad50>
```

In the official documentation, we find that conn is an HTTPResponse object with a number of methods, and that its read() method will give us data from the web page:

```
>>> data = conn.read()
>>> print(data)
b'You will be surprised by a loud noise.\r\n\n[codehappy]
http://iheartquotes.com/fortune/show/[illegible] \n'
```

This little chunk of Python opened a TCP/IP connection to the remote quote server, made an HTTP request, and received an HTTP response. The response contained more than just the page data (the fortune). 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 famous 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.

Web servers can send data back to you in any format they like. It's usually HTML (and usually some CSS and JavaScript), but in our fortune cookie example it's plain text. The data format is specified by the HTTP response *header* value with the name Content-Type, which we also saw in our *google.com* example:

```
>>> print(conn.getheader('Content-Type'))
text/plain
```

That text/plain string is a *MIME type*, and it means plain old text. The MIME type for HTML, which the *google.com* example sent, is text/html. I'll show you more MIME types in this chapter.

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

```
>>> for key, value in conn.getheaders():
...     print(key, value)
...
Server nginx
Date Sat, 24 Aug 2013 22:48:39 GMT
Content-Type text/plain
Transfer-Encoding chunked
Connection close
Etag "8477e32e6d053fcfdd6750f0c9c306d6"
X-Ua-Compatible IE=Edge,chrome=1
X-Runtime 0.076496
Cache-Control max-age=0, private, must-revalidate
```

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's a program that accesses a YouTube 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 into your Python environment. From a terminal window (Windows users, type `cmd` to make one), type the following command to make the Python package installer `pip` download the latest version of the `requests` package and install it:

```
$ pip install requests
```

If you have trouble, read Appendix D for details on how to install and use `pip`.

Let's redo our previous call to the quotes service with `requests`:

```
>>> import requests
>>> url = 'http://www.iheartquotes.com/api/v1/random'
>>> resp = requests.get(url)
>>> resp
<Response [200]>
>>> print(resp.text)
I know that there are people who do not love their fellow man, and I hate
people like that!

    -- Tom Lehrer, Satirist and Professor
```

```
[codehappy] http://iheartquotes.com/fortune/show/[illegible]
```

It isn't that different from using `urllib.request.urlopen`, but I think it feels a little less wordy.

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 (HTM 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 [illegible] 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 11.

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.[illegible] - - [20/Feb/2011 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 7-1, 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

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 *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.

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, we'll 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 Python web frameworks (some might say too many). A web framework handles, at a minimum, client requests and server responses. It might provide some or all of these features:

Routes
: Interpret URLs and find the corresponding server files or Python server code

Templates
: Merge server-side data into pages of HTML

Authentication and authorization
: Handle usernames, passwords, permissions

Sessions
: Maintain transient data storage during a user's visit to the website

In the coming sections, we'll write example code for two frameworks (`bottle` and `flask`). Then, we'll 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*:

```
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*:

```
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:

```
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*:

```
from [illegible] import route, run, static_file

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

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

run(host='localhost', port=[illegible])
```

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 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*:

```
import [illegible]

resp = requests.get('http://localhost:9999/echo/Mothra')
if resp.status_code == [illegible] 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 8 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*:

```
from flask import Flask

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

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

('/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.

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*:

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

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

Next, we'll 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*:

```
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 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 this as *flask3a.py*):

```
from [illegible] import Flask, render_template

app = Flask(__name__)

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

app.run(port=[illegible], 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:[illegible]/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 (save this as *flask3b.py*):

```
from [illegible] import Flask, render_template, request

app = Flask(__name__)

          ('/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=[illegible], debug=True)
```

Run the new server script:

```
$ python flask3b.py
```

This time, use this URL:

```
http://localhost:[illegible]/echo?thing=Gorgo&place=Wilnerding
```

You should get back what you see here:

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

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*):

```
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.

Non-Python Web Servers

So far, the web servers we've used have been simple: the standard library's http.server or the debugging servers in Bottle and Flask. In production, you'll want to run Python with a faster web server. The usual choices are the following:

- apache with the mod_wsgi module
- nginx with the uWSGI app server

Both work well; apache is probably the most popular, and nginx has a reputation for stability and lower memory use.

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 OS X. 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 this test file and save it as */var/www/test/home.wsgi*:

```
import bottle

application = bottle.default_app()
```

```
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 in and of 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=15
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>
```

The nginx Web Server

The nginx web server does not have an embedded Python module. Instead, it communicates by using a separate WSGI server such as uWSGI. Together they make a very fast and configurable platform for Python web development.

You can install nginx from its website. You also need to install uWSGI. uWSGI is a large system, with many levers and knobs to adjust. A short documentation page gives you instructions on how to combine Flask, nginx, and uWSGI.

Other Frameworks

Websites and databases are like peanut butter and jelly—you see them together a lot. The smaller frameworks such as bottle and flask do not include direct support for databases, although some of their contributed add-ons do.

If you need to crank out database-backed websites, and the database design doesn't change very often, it might be worth the effort to try one of the larger Python web frameworks. The current main contenders include:

`django`
: This is the most popular, especially for large sites. It's worth learning for many reasons, among them the frequent requests for django experience in Python job ads. It includes ORM code (we talked about ORMs in "The Object-Relational Mapper" on page 202) to create automatic web pages for the typical database *CRUD* functions (create, replace, update, delete) that I discussed in "SQL" on page 194. You don't have to use django's ORM if you prefer another, such as SQLAlchemy, or direct SQL queries.

`web2py`
: This covers much the same ground as django, with a different style.

`pyramid`
: This grew from the earlier pylons project, and is similar to django in scope.

`turbogears`
: This framework supports an ORM, many databases, and multiple template languages.

`wheezy.web`
: This is a newer framework optimized for performance. It was faster than the others in a recent test.

You can compare the frameworks by viewing this online table.

If you want to build a website backed by a relational database, you don't necessarily need one of these larger frameworks. You can use `bottle`, `flask`, and others directly with relational database modules, or use SQLAlchemy to help gloss over the differences. Then, you're writing generic SQL instead of specific ORM code, and more developers know SQL than any particular ORM's syntax.

Also, there's nothing written in stone demanding that your database must 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 "NoSQL Data Stores" on page 204. I once worked on a website 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.

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" on page 262) 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 11.

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.

The webbrowser Module

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.[1]

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.

Web APIs and Representational State Transfer (REST)

Often, data is only available 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

1. If you don't see it for some reason, visit xkcd.

in formats that are easier for programs to consume, such as JSON or XML (refer to Chapter 8 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, as is described here:

HEAD
: Gets information about the resource, but not its data.

GET
: As its name implies, GET retrieves the resource's data from the server. This is the standard method used by your browser. Any time you see a URL with a question mark (?) followed by a bunch of arguments, that's a GET request. GET should not be used to create, change, or delete data.

POST
: This verb updates data on the server. It's often used by HTML forms and web APIs.

PUT
: This verb creates a new resource.

DELETE
: This one speaks for itself: 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.

JSON

Chapter 1 shows two Python code samples to get information on popular YouTube videos, and Chapter 8 introduces JSON. JSON is especially well suited to web client-server data interchange. It's especially popular in web-based APIs, such as OpenStack.

Crawl and Scrape

Sometimes, you might want a little bit of information—a movie rating, stock price, or product availability—but the information 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* (unappealing terms to arachnophobes). After the contents have been retrieved from the remote web servers, a *scraper* parses it to find the needle in the haystack.

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

```
$ pip install scrapy
```

Scrapy is a framework, not a module such as `BeautifulSoup`. It does more, but it's more complex to set up. To learn more about Scrapy, read the documentation or the online introduction.

Scrape HTML with 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 Chapter 7) 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 the following example, 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:

```
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
```

Things to Do

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

9.2. 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.

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

9.4. 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>
```

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

CHAPTER 10
Systems

> One thing a computer can do that most humans can't is be sealed up in a cardboard box and sit in a warehouse.
>
> — Jack Handey

In your everyday use of a computer, you do such things as list the contents of a folder or directory, create and remove files, and other housekeeping that's necessary if not particularly exciting. You can also carry out these tasks, and more, within your own Python programs. Will this power drive you mad or cure your insomnia? We'll see.

Python provides many system functions through a module named `os` (for "operating system"), which we'll import for all the programs in this chapter.

Files

Python, like many other languages, patterned its file operations after Unix. Some functions, such as `chown()` and `chmod()`, have the same names, but there are a few new ones.

Create with open()

"File Input/Output" on page 173 introduced you to the `open()` function and explains how you can use it to open a file or create one if it doesn't already exist. Let's create a text file called *oops.txt*:

```
>>> fout = open('oops.txt', 'wt')
>>> print('Oops, I created a file.', file=fout)
>>> fout.close()
```

With that done, let's perform some tests with it.

Check Existence with exists()

To verify whether the file or directory is really there or you just imagined it, you can provide `exists()`, with a relative or absolute pathname, as demonstrated here:

```
>>> import os
>>> os.path.exists('oops.txt')
True
>>> os.path.exists('./oops.txt')
True
>>> os.path.exists('waffles')
False
>>> os.path.exists('.')
True
>>> os.path.exists('..')
True
```

Check Type with isfile()

The functions in this section check whether a name refers to a file, directory, or symbolic link (see the examples that follow for a discussion of links).

The first function we'll look at, `isfile`, asks a simple question: is it a plain old law-abiding file?

```
>>> name = 'oops.txt'
>>> os.path.isfile(name)
True
```

Here's how you determine a directory:

```
>>> os.path.isdir(name)
False
```

A single dot (`.`) is shorthand for the current directory, and two dots (`..`) stands for the parent directory. These always exist, so a statement such as the following will always report `True`:

```
>>> os.path.isdir('.')
True
```

The `os` module contains many functions dealing with *pathnames* (fully qualified filenames, starting with `/` and including all parents). One such function, `isabs()`, determines whether its argument is an absolute pathname. The argument doesn't need to be the name of a real file:

```
>>> os.path.isabs(name)
False
>>> os.path.isabs('/big/fake/name')
True
>>> os.path.isabs('big/fake/name/without/a/leading/slash')
False
```

Copy with copy()

The copy() function comes from another module, shutil. This example copies the file *oops.txt* to the file *ohno.txt*:

```
>>> import shutil
>>> shutil.copy('oops.txt', 'ohno.txt')
```

The shutil.move() function copies a file and then removes the original.

Change Name with rename()

This function does exactly what it says. In the example here, it renames *ohno.txt* to *ohwell.txt*:

```
>>> import os
>>> os.rename('ohno.txt', 'ohwell.txt')
```

Link with link() or symlink()

In Unix, a file exists in one place, but it can have multiple names, called *links*. In low-level *hard links*, it's not easy to find all the names for a given file. A *symbolic link* is an alternative method that stores the new name as its own file, making it possible for you to get both the original and new names at once. The link() call creates a hard link, and symlink() makes a symbolic link. The islink() function checks whether the file is a symbolic link.

Here's how to make a hard link to the existing file *oops.txt* from the new file *yikes.txt*:

```
>>> os.link('oops.txt', 'yikes.txt')
>>> os.path.isfile('yikes.txt')
True
```

To create a symbolic link to the existing file *oops.txt* from the new file *jeepers.txt*, use the following:

```
>>> os.path.islink('yikes.txt')
False
>>> os.symlink('oops.txt', 'jeepers.txt')
>>> os.path.islink('jeepers.txt')
True
```

Change Permissions with chmod()

On a Unix system, chmod() changes file permissions. There are read, write, and execute permissions for the user (that's usually you, if you created the file), the main group that the user is in, and the rest of the world. The command takes an intensely compressed octal (base 8) value that combines user, group, and other permissions. For instance, to make *oops.txt* only readable by its owner, type the following:

```
>>> os.chmod('oops.txt', 0o400)
```

If you don't want to deal with cryptic octal values and would rather deal with (slightly) obscure cryptic symbols, you can import some constants from the `stat` module and use a statement such as the following:

```
>>> import stat
>>> os.chmod('oops.txt', stat.S_IRUSR)
```

Change Ownership with chown()

This function is also Unix/Linux/Mac–specific. You can change the owner and/or group ownership of a file by specifying the numeric user ID (*uid*) and group ID (*gid*):

```
>>> uid = 5
>>> gid = 22
>>> os.chown('oops', uid, gid)
```

Get a Pathname with abspath()

This function expands a relative name to an absolute one. If your current directory is */usr/gaberlunzie* and the file *oops.txt* is there, also, you can type the following:

```
>>> os.path.abspath('oops.txt')
 /usr/gaberlunzie/oops.txt'
```

Get a symlink Pathname with realpath()

In one of the earlier sections, we made a symbolic link to *oops.txt* from the new file *jeepers.txt*. In circumstances such as this, you can get the name of *oops.txt* from *jeepers.txt* by using the `realpath()` function, as shown here:

```
>>> os.path.realpath('jeepers.txt')
'/usr/gaberlunzie/oops.txt'
```

Delete a File with remove()

In this snippet, we use the `remove()` function and say farewell to *oops.txt*:

```
>>> os.remove('oops.txt')
>>> os.path.exists('oops.txt')
False
```

Directories

In most operating systems, files exist in a hierarchy of *directories* (more often called *folders* these days). The container of all of these files and directories is a *file system* (sometimes called a *volume*). The standard os module deals with operating specifics such as these and provides the following functions with which you can manipulate them.

Create with mkdir()

This example shows how to create a directory called poems to store that precious verse:

```
>>> os.mkdir('poems')
>>> os.path.exists('poems')
True
```

Delete with rmdir()

Upon second thought, you decide you don't need that directory after all. Here's how to delete it:

```
>>> os.rmdir('poems')
>>> os.path.exists('poems')
False
```

List Contents with listdir()

Okay, take two; let's make poems again, with some contents:

```
>>> os.mkdir('poems')
```

Now, get a list of its contents (none so far):

```
>>> os.listdir('poems')
[]
```

Next, make a subdirectory:

```
>>> os.mkdir('poems/mcintyre')
>>> os.listdir('poems')
['mcintyre']
```

Create a file in this subdirectory (don't type all these lines unless you really feel poetic; just make sure you begin and end with matching quotes, either single or tripled):

```
>>> fout = open('poems/mcintyre/the_good_man', 'wt')
>>> fout.write('''Cheerful and happy was his mood,
... He to the poor was kind and good,
... And he oft' times did find them food,
... Also supplies of coal and wood,
... He never spake a word was rude,
... And cheer'd those did o'er sorrows brood,
... He passed away not understood,
... Because no poet in his lays
... Had penned a sonnet in his praise,
... 'Tis sad, but such is world's ways.
... ''')
...
>>> fout.close()
```

Finally, let's see what we have. It had better be there:

```
>>> os.listdir('poems/mcintyre')
['the_good_man']
```

Change Current Directory with chdir()

With this function, you can go from one directory to another. Let's leave the current directory and spend a little time in poems:

```
>>> import os
>>> os.chdir('poems')
>>> os.listdir('.')
['mcintyre']
```

List Matching Files with glob()

The `glob()` function matches file or directory names by using Unix shell rules rather than the more complete regular expression syntax. Here are those rules:

- * matches everything (re would expect .*)
- ? matches a single character
- [abc] matches character a, b, or c
- [!abc] matches any character *except* a, b, or c

Try getting all files or directories that begin with m:

```
>>> import glob
>>> glob.glob('m*')
['mcintyre']
```

How about any two-letter files or directories?

```
>>> glob.glob('??')
[]
```

I'm thinking of an eight-letter word that begins with m and ends with e:

```
>>> glob.glob('m??????e')
['mcintyre']
```

What about anything that begins with a k, l, or m, and ends with e?

```
>>> glob.glob('[klm]*e')
['mcintyre']
```

Programs and Processes

When you run an individual program, your operating system creates a single *process*. It uses system resources (CPU, memory, disk space) and data structures in the operating system's *kernel* (file and network connections, usage statistics, and so on). A process is isolated from other processes—it can't see what other processes are doing or interfere with them.

The operating system keeps track of all the running processes, giving each a little time to run and then switching to another, with the twin goals of spreading the work around fairly and being responsive to the user. You can see the state of your processes with graphical interfaces such as the Mac's Activity Monitor (OS X), or Task Manager on Windows-based computers.

You can also access process data from your own programs. The standard library's os module provides a common way of accessing some system information. For instance, the following functions get the *process ID* and the *current working directory* of the running Python interpreter:

```
>>> import os
>>> os.getpid()
...
>>> os.getcwd()
'/Users/williamlubanovic'
```

And these get my *user ID* and *group ID*:

```
>>> os.getuid()
...
>>> os.getgid()
...
```

Create a Process with subprocess

All of the programs that you've seen here so far have been individual processes. You can start and stop other existing programs from Python by using the standard library's subprocess module. If you just want to run another program in a shell and grab whatever output it created (both standard output and standard error output), use the getoutput() function. Here, we'll get the output of the Unix date program:

```
>>> import subprocess
>>> ret = subprocess.getoutput('date')
>>> ret
'Sun Mar 30 22:54:37 CDT 2014'
```

You won't get anything back until the process ends. If you need to call something that might take a lot of time, see the discussion on *concurrency* in "Concurrency" on page 262. Because the argument to getoutput() is a string representing a complete shell command, you can include arguments, pipes, < and > I/O redirection, and so on:

```
>>> ret = subprocess.getoutput('date -u')
>>> ret
'Mon Mar 31 03:55:01 UTC 2014'
```

Piping that output string to the `wc` command counts one line, six "words," and 29 characters:

```
>>> ret = subprocess.getoutput('date -u | wc')
>>> ret
'       1       6      29'
```

A variant method called `check_output()` takes a list of the command and arguments. By default it only returns standard output as type bytes rather than a string and does not use the shell:

```
>>> ret = subprocess.check_output(['date', '-u'])
>>> ret
b'Mon Mar 31 04:01:50 UTC 2014\n'
```

To show the exit status of the other program, `getstatusoutput()` returns a tuple with the status code and output:

```
>>> ret = subprocess.getstatusoutput('date')
>>> ret
(0, 'Sat Jan 18 21:36:23 CST 2014')
```

If you don't want to capture the output but might want to know its exit status, use `call()`:

```
>>> ret = subprocess.call('date')
Sat Jan 18 21:36:11 CST 2014
>>> ret
0
```

(In Unix-like systems, `0` is usually the exit status for success.)

That date and time was printed to output but not captured within our program. So, we saved the return code as `ret`.

You can run programs with arguments in two ways. The first is to specify them in a single string. Our sample command is `date -u`, which prints the current date and time in UTC (you'll read more about UTC in a few pages):

```
>>> ret = subprocess.call('date -u', shell=True)
Tue Jan 21 20:30:04 UTC 2014
```

You need that `shell=True` to recognize the command line `date -u`, splitting it into separate strings and possibly expanding any wildcard characters such as `*` (we didn't use any in this example).

The second method makes a list of the arguments, so it doesn't need to call the shell:

```
>>> ret = subprocess.call(['date', '-u'])
Tue Jan 21 20:31:59 UTC 2014
```

Create a Process with multiprocessing

You can run a Python function as a separate process or even run multiple independent processes in a single program with the `multiprocessing` module. Here's a short example that does nothing useful; save it as *mp.py* and then run it by typing `python mp.py`:

```
import multiprocessing
import os

def do_this(what):
    whoami(what)

def whoami(what):
    print("Process %s says: %s" % (os.getpid(), what))

if __name__ == "__main__":
    whoami("I'm the main program")
    for n in range(4):
        p = multiprocessing.Process(target=do_this,
          args=("I'm function %s" % n,))
        p.start()
```

When I run this, my output looks like this:

```
Process 6234 says: I'm the main program
Process 6235 says: I'm function 0
Process 6236 says: I'm function 1
Process 6237 says: I'm function 2
Process 6238 says: I'm function 3
```

The `Process()` function spawned a new process and ran the `do_this()` function in it. Because we did this in a loop that had four passes, we generated four new processes that executed `do_this()` and then exited.

The `multiprocessing` module has more bells and whistles than a clown on a calliope. It's really intended for those times when you need to farm out some task to multiple processes to save overall time; for example, downloading web pages for scraping, resizing images, and so on. It includes ways to queue tasks, enable intercommunication among processes, and wait for all the processes to finish. "Concurrency" on page 262 delves into some of these details.

Kill a Process with terminate()

If you created one or more processes and want to terminate one for some reason (perhaps it's stuck in a loop, or maybe you're bored, or you want to be an evil overlord), use `terminate()`. In the example that follows, our process would count to a million, sleeping at each step for a second, and printing an irritating message. However, our main program runs out of patience in five seconds and nukes it from orbit:

```
import multiprocessing
import time
import os

def whoami(name):
    print("I'm %s, in process %s" % (name, os.getpid()))

def loopy(name):
    whoami(name)
    start = 1
    stop = 1000000
    for num in range(start, stop):
        print("\tNumber %s of %s. Honk!" % (num, stop))
        time.sleep(1)

if __name__ == "__main__":
    whoami("main")
    p = multiprocessing.Process(target=loopy, args=("loopy",))
    p.start()
    time.sleep(5)
    p.terminate()
```

When I run this program, I get the following:

```
I'm main, in process 97080
I'm loopy, in process 97081
    Number 1 of 1000000. Honk!
    Number 2 of 1000000. Honk!
    Number 3 of 1000000. Honk!
    Number 4 of 1000000. Honk!
    Number 5 of 1000000. Honk!
```

Calendars and Clocks

Programmers devote a surprising amount of effort to dates and times. Let's talk about some of the problems they encounter, and then get to some best practices and tricks to make the situation a little less messy.

Dates can be represented in many ways—too many ways, actually. Even in English with the Roman calendar, you'll see many variants of a simple date:

- July 29 1984
- 29 Jul 1984
- 29/7/1984
- 7/29/1984

Among other problems, date representations can be ambiguous. In the previous examples, it's easy to determine that 7 stands for the month and 29 is the day of the month,

largely because months don't go to 29. But how about 1/6/2012? Is that referring to January 6 or June 1?

The month name varies by language within the Roman calendar. Even the year and month can have a different definition in other cultures.

Leap years are another wrinkle. You probably know that every four years is a leap year (and the summer Olympics and the American presidential election). Did you also know that every 100 years is not a leap year, but that every 400 years is? Here's code to test various years for leapiness:

```
>>> import
>>> calendar.isleap(
False
>>> calendar.isleap(
True
>>> calendar.isleap(
False
>>> calendar.isleap(
True
>>> calendar.isleap(
False
>>> calendar.isleap(
True
```

Times have their own sources of grief, especially because of time zones and daylight savings time. If you look at a time zone map, the zones follow political and historic boundaries rather than every 15 degrees (360 degrees / 24) of longitude. And countries start and end daylight saving times on different days of the year. In fact, countries in the southern hemisphere advance their clocks when the northern hemisphere is winding them back, and vice versa. (If you think about it a bit, you will see why.)

Python's standard library has many date and time modules: `datetime`, `time`, `calendar`, `dateutil`, and others. There's some overlap, and it's a bit confusing.

The datetime Module

Let's begin by investigating the standard `datetime` module. It defines four main objects, each with many methods:

- `date` for years, months, and days
- `time` for hours, minutes, seconds, and fractions
- `datetime` for dates and times together
- `timedelta` for date and/or time intervals

You can make a `date` object by specifying a year, month, and day. Those values are then available as attributes:

```
>>> from datetime import date
>>> halloween = date(2014, 10, 31)
>>> halloween
datetime.date(2014, 10, 31)
>>> halloween.day
31
>>> halloween.month
10
>>> halloween.year
2014
```

You can print a date with its `isoformat()` method:

```
>>> halloween.isoformat()
'2014-10-31'
```

The `iso` refers to ISO 8601, an international standard for representing dates and times. It goes from most general (year) to most specific (day). It also sorts correctly: by year, then month, then day. I usually pick this format for date representation in programs, and for filenames that save data by date. The next section describes the more complex `strptime()` and `strftime()` methods for parsing and formatting dates.

This example uses the `today()` method to generate today's date:

```
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2014, 2, 2)
```

This one makes use of a `timedelta` object to add some time interval to a date:

```
>>> from datetime import timedelta
>>> one_day = timedelta(days=1)
>>> tomorrow = now + one_day
>>> tomorrow
datetime.date(2014, 2, 3)
>>> now + 17*one_day
datetime.date(2014, 2, 19)
>>> yesterday = now - one_day
>>> yesterday,
datetime.date(2014, 2, 1)
```

The range of `date` is from `date.min` (year=1, month=1, day=1) to `date.max` (year=9999, month=12, day=31). As a result, you can't use it for historic or astronomical calculations.

The `datetime` module's `time` object is used to represent a time of day:

```
>>> from datetime import time
>>> noon = time(12, 0, 0)
>>> noon
datetime.time(12, 0)
>>> noon.hour
12
```

```
>>> noon.minute
0
>>> noon.second
0
>>> noon.microsecond
0
```

The arguments go from the largest time unit (hours) to the smallest (microseconds). If you don't provide all the arguments, time assumes all the rest are zero. By the way, just because you can store and retrieve microseconds doesn't mean you can retrieve time from your computer to the exact microsecond. The accuracy of subsecond measurements depends on many factors in the hardware and operating system.

The datetime object includes both the date and time of day. You can create one directly, such as the one that follows, which is for January 2, 2014, at 3:04 A.M., plus 5 seconds and 6 microseconds:

```
>>> from datetime import datetime
>>> some_day = datetime(2014, 1, 2, 3, 4, 5, 6)
>>> some_day
datetime.datetime(2014, 1, 2, 3, 4, 5, 6)
```

The datetime object also has an isoformat() method:

```
>>> some_day.isoformat()
'2014-01-02T03:04:05.000006'
```

That middle T separates the date and time parts.

datetime has a now() method with which you can get the current date and time:

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> now
datetime.datetime(2014, 2, 2, 23, 15, 34, 694988)
14
>>> now.month
2
>>> now.day
2
>>> now.hour
23
>>> now.minute
15
>>> now.second
34
>>> now.microsecond
694988
```

You can merge a date object and a time object into a datetime object by using combine():

```
>>> from [illegible] import datetime, time, date
>>> noon = time([illegible])
>>> this_day = date.today()
>>> noon_today = datetime.combine(this_day, noon)
>>> noon_today
datetime.datetime([illegible])
```

You can yank the date and time from a datetime by using the `date()` and `time()` methods:

```
>>> noon_today.date()
datetime.date([illegible])
>>> noon_today.time()
datetime.time([illegible])
```

Using the time Module

It is confusing that Python has a `datetime` module with a `time` object, and a separate `time` module. Furthermore, the `time` module has a function called—wait for it—`time()`.

One way to represent an absolute time is to count the number of seconds since some starting point. *Unix time* uses the number of seconds since midnight on January 1, 1970.[1] This value is often called the *epoch*, and it is often the simplest way to exchange dates and times among systems.

The `time` module's `time()` function returns the current time as an epoch value:

```
>>> import [illegible]
>>> now = time.time()
>>> now
[illegible]
```

If you do the math, you'll see that it has been over one billion seconds since New Year's, 1970. Where did the time go?

You can convert an epoch value to a string by using `ctime()`:

```
>>> time.ctime(now)
'Mon Feb  3 22:31:03 2014'
```

In the next section, you'll see how to produce more attractive formats for dates and times.

Epoch values are a useful least-common denominator for date and time exchange with different systems, such as JavaScript. Sometimes, though, you need actual days, hours, and so forth, which `time` provides as `struct_time` objects. `localtime()` provides the time in your system's time zone, and `gmtime()` provides it in UTC:

1. This starting point is roughly when Unix was born.

```
>>> time.localtime(now)
time.struct_time(tm_year=2014, tm_mon=2, tm_mday=3, tm_hour=22, tm_min=31,
tm_sec=3, tm_wday=0, tm_yday=34, tm_isdst=0)
>>> time.gmtime(now)
time.struct_time(tm_year=2014, tm_mon=2, tm_mday=4, tm_hour=4, tm_min=31,
tm_sec=3, tm_wday=1, tm_yday=35, tm_isdst=0)
```

In my (Central) time zone, 22:31 was 04:31 of the next day in UTC (formerly called *Greenwich time* or *Zulu time*). If you omit the argument to `localtime()` or `gmtime()`, they assume the current time.

The opposite of these is `mktime()`, which converts a `struct_time` object to epoch seconds:

```
>>> tm = time.localtime(now)
>>> time.mktime(tm)
1391488263.0
```

This doesn't exactly match our earlier epoch value of `now()` because the `struct_time` object preserves time only to the second.

Some advice: wherever possible, *use UTC* instead of time zones. UTC is an absolute time, independent of time zones. If you have a server, set its time to UTC; do not use local time.

Here's some more advice (free of charge, no less): *never use daylight savings time* if you can avoid it. If you use daylight savings time, an hour disappears at one time of year ("spring ahead") and occurs twice at another time ("fall back"). For some reason, many organizations use daylight savings in their computer systems, but are mystified every year by data duplicates and dropouts. It all ends in tears.

Remember, your friends are UTC for times, and UTF-8 for strings (for more about UTF-8, see Chapter 7).

Read and Write Dates and Times

`isoformat()` is not the only way to write dates and times. You already saw the `ctime()` function in the `time` module, which you can use to convert epochs to strings:

```
>>> import time
>>> now = time.time()
>>> time.ctime(now)
'Mon Feb  3 21:14:36 2014'
```

You can also convert dates and times to strings by using `strftime()`. This is provided as a method in the `datetime`, `date`, and `time` objects, and as a function in the `time`

module. strftime() uses format strings to specify the output, which you can see in Table 10-1.

Table 10-1. Outut specifiers for strftime()

Format string	Date/time unit	Range
%Y	year	1900-...
%m	month	01-12
%B	month name	January, ...
%b	month abbrev	Jan, ...
%d	day of month	01-31
%A	weekday name	Sunday, ...
a	weekday abbrev	Sun, ...
%H	hour (24 hr)	00-23
%I	hour (12 hr)	01-12
%p	AM/PM	AM, PM
%M	minute	00-59
%S	second	00-59

Numbers are zero-padded on the left.

Here's the strftime() function provided by the time module. It converts a struct_time object to a string. We'll first define the format string fmt and use it again later:

```
>>> import
>>> fmt = "It's %A, %B %d, %Y, local time %I:%M:%S%p"
>>> t = time.localtime()
>>> t
time.struct_time(tm_year=2014, tm_mon=2, tm_mday=4, tm_hour=7,
tm_min=28, tm_sec=38, tm_wday=1, tm_yday=35, tm_isdst=0)
>>> time.strftime(fmt, t)
"It's Tuesday, February 04, 2014, local time 07:28:38PM"
```

If we try this with a date object, only the date parts will work, and the time defaults to midnight:

```
>>> from datetime import date
>>> some_day = date(2014, 7, 4)
>>> fmt = "It's %B %d, %Y, local time %I:%M:%S%p"
>>> some_day.strftime(fmt)
"It's Friday, July 04, 2014, local time 12:00:00AM"
```

For a time object, only the time parts are converted:

```
>>> from          import time
>>> some_time = time(10, 35)
>>> some_time.strftime(fmt)
"It's Monday, January 01, 1900, local time 10:35:00AM"
```

END UNIT 4