Chapter 20. Serving HTTP

When a browser (or any other web client) requests a page from a server, the server may return either static or dynamic content. Serving dynamic content involves server-side web programs generating and delivering content on the fly, often based on information stored in a database.

In the early history of the web, the standard for server-side programming was the *Common Gateway Interface* (CGI), which required the server to run a separate program each time a client requested dynamic content. Process startup time, interpreter initialization, connection to databases, and script initialization add up to measurable overhead; CGI did not scale well.

Nowadays, web servers support many server-specific ways to reduce overhead, serving dynamic content from processes that can serve for several hits rather than starting up a new process per hit. Therefore, we do not cover CGI in this book. To maintain existing CGI programs, or better yet, port them to more modern approaches, consult the online docs (especially **PEP 594** for recommendations) and check out the standard library modules `cgi` (deprecated as of 3.11) and `http.cookies`.[1]

HTTP has become even more fundamental to distributed systems design with the emergence of systems based on **microservices**, offering a convenient way to transport between processes the JSON content that is frequently used. There are thousands of publicly available HTTP data APIs on the internet. While HTTP's principles remain almost unchanged since its inception in the mid-1990s, it has been significantly enhanced over the years to extend its capabilities.[2] For a thorough grounding with excellent reference materials we recommend ***HTTP: The Definitive Guide*** by David Gourley et al. (O'Reilly).

http.server

Python's standard library includes a module containing the server and handler classes to implement a simple HTTP server.

You can run this server from the command line by just entering:

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

By default, the server listens on all interfaces and provides access to the files in the current directory. One author uses this as a simple means for file transfer: start up a Python `http.server` in the file directory on the source system, and then copy files to the destination using a utility such as `wget` or `curl`.

`http.server` has very limited security features. You can find further information on `http.server` in the **online docs**. For production use, we recommend that you use one of the frameworks mentioned in the following sections.

WSGI

Python's *Web Server Gateway Interface* (WSGI) is the standard way for all modern Python web development frameworks to interface with underlying web servers or gateways. WSGI is not meant for direct use by your application programs; rather, you code your programs using any one of many higher-abstraction frameworks, and the framework, in turn, uses WSGI to talk to the web server.

You need to care about the details of WSGI only if you're implementing the WSGI interface for a web server that doesn't already provide it (should any such server exist), or if you're building a new Python web framework.[3] In that case, study the WSGI **PEP**, the docs for the standard library package **`wsgiref`**, and the **archive** of **WSGI.org**.

A few WSGI concepts may be important to you if you use lightweight frameworks (i.e., ones that match WSGI closely). WSGI is an *interface*, and that interface has two sides: the *web server/gateway* side, and the *application/framework* side.

The framework side's job is to provide a *WSGI application* object, a callable object (often the instance of a class with a `__call__` special method, but that's an implementation detail) respecting conventions in the PEP, and to connect the application object to the server by whatever means the specific server documents (often a few lines of code, or configuration files, or just a convention such as naming the WSGI application object `application` as a top-level attribute in a module). The server calls the application object for each incoming HTTP request, and the application object responds appropriately so that the server can form the outgoing HTTP response and send it on—all according to said conventions. A framework, even a lightweight one, shields you from such details (except that you may have to instantiate and connect the application object, depending on the specific server).

WSGI Servers

An extensive list of servers and adapters you can use to run WSGI frameworks and applications (for development and testing, in production web setups, or both) is available **online**—extensive, but just partial. For example, it does not mention that Google App Engine's Python runtime is also a WSGI server, ready to dispatch WSGI apps as directed by the *app.yaml* configuration file.

If you're looking for a WSGI server to use for development, or to deploy in production behind, say, an Nginx-based load balancer, you should be happy, at least on Unix-like systems, with **Gunicorn**: pure Python goodness, supporting nothing but WSGI, very lightweight. A worthy (also pure Python and WSGI-only) alternative, currently with better Windows support, is **Waitress**. If you need richer features (such as support for Perl and Ruby as well as Python, and many other forms of extensibility), consider the bigger, more complex **uWSGI**.[4]

WSGI also has the concept of *middleware*, a subsystem that implements both the server and application sides of WSGI. A middleware object "wraps" a WSGI application; can selectively alter requests, environments, and responses; and presents itself to the server as "the application." Multiple layers of wrappers are allowed and common, forming a "stack" of middleware offering services to the actual application-level code. If you want to write a cross-framework middleware component, then you may, indeed, need to become a WSGI expert.

ASGI

If you're into asynchronous Python (which we don't cover in this book), you should definitely investigate **ASGI**, which sets out to do pretty much what WSGI does, but asynchronously. As is usually the case for asynchronous programs in a networking environment, it can offer greatly improved performance, albeit (arguably) with some increase in cognitive load for the developer.

Python Web Frameworks

For a survey of Python web frameworks, see the Python **wiki page**. It's authoritative since it's on the official **Python.org** website, and it's community curated, so it stays up-to-date as time goes by. The wiki lists and points to dozens of frameworks[5] that it identifies as "active," plus many more it identifies as "discontinued/inactive." In addition, it points to separate wiki pages about Python content management systems, web servers, and web components and libraries thereof.

"Full-Stack" Versus "Lightweight" Frameworks

Roughly speaking, Python web frameworks can be classified as being either *full-stack* (trying to supply all the functionality you may need to build a web application) or *lightweight* (supplying just a handy interface to web serving itself, and letting you pick and choose your own favorite components for tasks such as interfacing to databases and templating). Of course, like all taxonomies, this one is imprecise and incomplete, and requires value judgments; however, it’s one way to start making sense of the many Python web frameworks.

In this book, we do not thoroughly cover any full-stack frameworks—each is far too complex. Nevertheless, one of them might be the best approach for your specific applications, so we do mention a few of the most popular ones, and recommend that you check out their websites.

A Few Popular Full-Stack Frameworks

By far the most popular full-stack framework is **Django**, which is sprawling and extensible. Django’s so-called *applications* are in fact reusable subsystems, while what’s normally called “an application” Django calls a *project*. Django requires its own unique mindset, but offers enormous power and functionality in return.

An excellent alternative is **web2py**: it’s just about as powerful, easier to learn, and well known for its dedication to backward compatibility (if it keeps up its great track record, any web2py application you code today will keep working far into the future). web2py also has outstanding documentation.

A third worthy contender is **TurboGears**, which starts out as a lightweight framework but achieves “full-stack” status by fully integrating other, independent third-party projects for the various other functionalities needed in most web apps, such as database interfacing and templating, rather than designing its own. Another somewhat philosophically similar “light but rich” framework is **Pyramid**.

Considerations When Using Lightweight Frameworks

Whenever you use a lightweight framework, if you need any database, templating, or other functionality not strictly related to HTTP, you’ll be picking and choosing separate components for that purpose. However, the lighter in weight your framework, the more components you will need to understand and integrate, for purposes such as authenticating a user or maintaining state across web requests by a given user. Many WSGI middleware packages can help you with such tasks. Some excellent ones are quite focused—for example, **Oso** for access control, **Beaker** for maintaining state in the form of lightweight sessions of any one of several kinds, and so forth.

However, when we (the authors of this book) require good WSGI middleware for just about any purpose, we almost invariably first check **Werkzeug**, a collection of such components that's amazing in breadth and quality. We don't cover Werkzeug in this book (just as we don't cover other middleware), but we recommend it highly (Werkzeug is also the foundation on which Flask—our favorite lightweight framework, which we do cover later in this chapter—is built).

You may notice that properly using lightweight frameworks requires you to understand HTTP (in other words, to know what you're doing), while a full-stack framework tries to lead you by the hand and have you do the right thing without really needing to understand how or why it is right—at the cost of time and resources, and of accepting the full-stack framework's conceptual map and mindset. The authors of this book are enthusiasts of the knowledge-heavy, resources-light approach of lightweight frameworks, but we acknowledge that there are many situations where the rich, heavy, all-embracing full-stack frameworks are more appropriate. To each their own!

A Few Popular Lightweight Frameworks

As mentioned, Python has multiple frameworks, including many lightweight ones. We cover two of the latter here: the popular, general-purpose Flask, and API-centric FastAPI.

Flask

The most popular Python lightweight framework is **Flask**, a third-party `pip`-installable package. Although lightweight, it includes a development server and debugger, and it explicitly relies on other well-chosen packages such as Werkzeug for middleware and **Jinja** for templating (both packages were originally authored by Armin Ronacher, the author of Flask).

In addition to the project website (which includes rich, detailed docs), look at the **sources on GitHub** and the **PyPI entry**. If you want to run Flask on Google App Engine (locally on your computer, or on Google's servers at **appspot.com**), Dough Mahugh's **Medium article** can be quite handy.

We also highly recommend Miguel Grinberg's book ***Flask Web Development*** (O'Reilly): although the second edition is rather dated (almost four years old at the time of this writing), it still provides an excellent foundation, on top of which you'll have a far easier time learning the latest new additions.

The main class supplied by the `flask` package is named `Flask`. An instance of `flask.Flask`, besides being a WSGI application itself, also wraps a WSGI application as its `wsgi_app` property. When you need to further wrap the WSGI app in some WSGI middleware, use the idiom:

```
import flask

app = flask.Flask(__name__)
app.wsgi_app = some_middleware(app.wsgi_app)
```

When you instantiate `flask.Flask`, always pass it as the first argument the application name (often just the `__name__` special variable of the module where you instantiate it; if you instantiate it from within a package, usually in *__init__.py*, `__name__.partition('.')[0]` works). Optionally, you can also pass named parameters such as `static_folder` and `template_folder` to customize where static files and Jinja templates are found; however, that's rarely needed—the default values (subfolders named *static* and *templates*, respectively, located in the same folder as the Python script that instantiates `flask.Flask`) make perfect sense.

An instance *app* of `flask.Flask` supplies more than 100 methods and properties, many of them decorators to bind functions to *app* in various roles, such as *view functions* (serving HTTP verbs on a URL) or *hooks* (letting you alter a request before it's processed or a response after it's built, handling errors, and so forth).

`flask.Flask` takes just a few parameters at instantiation (and the ones it takes are not ones that you usually need to compute in your code), and it supplies decorators you'll want to use as you define, for example, view functions. Thus, the normal pattern in `flask` is to instantiate *app* early in your main script, just as your application is starting up, so that the app's decorators, and other methods and properties, are available as you **`def`** view functions and so on.

Since there is a single global *app* object, you may wonder how thread-safe it can be to access, mutate, and rebind *app*'s properties and attributes. Not to worry: the names you see are actually just proxies to actual objects living in the *context* of a specific request, in a specific thread or **greenlet**. Never type-check those properties (their types are in fact obscure proxy types), and you'll be fine.

Flask also supplies many other utility functions and classes; often, the latter subclass or wrap classes from other packages to add seamless, convenient Flask integration. For example, Flask's `Request` and `Response` classes add just a little handy functionality by subclassing the corresponding Werkzeug classes.

Flask request objects

The class `flask.Request` supplies a large number of **thoroughly documented properties**. **Table 20-1** lists the ones you'll be using most often.

Table 20-1. Useful properties of `flask.Request`

`args`	A `MultiDict` of the request's query arguments
`cookies`	A `dict` with the cookies from the request

`data`	A `bytes` string, the request's body (typically for POST and PUT requests)
`files`	A `MultiDict` of uploaded files in the request, mapping the files' names to file-like objects containing each file's data
`form`	A `MultiDict` with the request's form fields, provided in the request's body
`headers`	A `MultiDict` with the request's headers
`values`	A `MultiDict` combining the `args` and `form` properties

A `MultiDict` is like a `dict`, except that it can have multiple values for a key. Indexing and `get` on a `MultiDict` instance *`m`* return an arbitrary one of the values; to get the list of values for a key (an empty list, if the key is not in *`m`*), call *`m`*`.getlist(`*`key`*`)`.

Flask response objects

Often, a Flask view function can just return a string (which becomes the response's body): Flask transparently wraps an instance *`r`* of `flask.Response` around the string, so you don't have to worry about the response class. However, sometimes you want to alter the response's headers; in this case, in the view function, call *`r`* `= flask.make_response(`*`astring`*`)`, alter *`r`*`.headers` as you want, then `return` *`r`*. (To set a cookie, don't use *`r`*`.headers`; rather, call ***`r`*****`.set_cookie`**.)

Some of Flask's built-in integrations with other systems don't require subclassing: for example, the templating integration implicitly injects into the Jinja context the Flask globals `config`, `request`, `session`, and `g` (the latter being the handy "globals catch-all" object `flask.g`, a proxy in application context, in which your code can store whatever you want to "stash" for the duration of the request being served) and the functions `url_for` (to translate an endpoint to the corresponding URL, same as `flask.url_for`) and get_flashed_messages (to support *flashed messages*, which we do not cover in this book; same as flask.get_flashed_messages). Flask also provides convenient ways for your code to inject more filters, functions, and values into the Jinja context, without any subclassing.

Most of the officially recognized or approved Flask **extensions** (hundreds are available from PyPI at the time of this writing) adopt similar approaches, supplying classes and utility functions to seamlessly integrate other popular subsystems with your Flask applications.

In addition, Flask introduces other features, such as ***signals*** to provide looser dynamic coupling in a "pub/sub" pattern and ***blueprints***, offering a substantial subset of a Flask application's functionality to ease refactoring large applications in highly modular, flexible ways. We do not cover these advanced concepts in this book.

Example 20-1 shows a simple Flask example. (After using pip to install Flask, run the example using the command **flask --app flask_example run**.)

Example 20-1. A Flask example

```
import datetime, flask
app = flask.Flask(__name__)

# secret key for cryptographic components such as encoding session cookies;
# for production use, use secrets.token_bytes()
app.secret_key = b'\xc5\x8f\xbc\xa2\x1d\xeb\xb3\x94;:d\x03'

@app.route('/')
def greet():
    lastvisit = flask.session.get('lastvisit')
    now = datetime.datetime.now()
    newvisit = now.ctime()
    template = '''
      <html><head><title>Hello, visitor!</title>
      </head><body>
      {% if lastvisit %}
        <p>Welcome back to this site!</p>
```

```
          <p>You last visited on {{lastvisit}} UTC</p>
          <p>This visit on {{newvisit}} UTC</p>
        {% else %}
          <p>Welcome to this site on your first visit!</p>
          <p>This visit on {{newvisit}} UTC</p>
          <p>Please Refresh the web page to proceed</p>
        {% endif %}
        </body></html>'''
      flask.session['lastvisit'] = newvisit
      return flask.render_template_string(
        template, newvisit=newvisit, lastvisit=lastvisit)
```

This example shows how to use just a few of the many building blocks that Flask offers—the `Flask` class, a view function, and rendering the response (in this case, using `render_template_string` on a Jinja template; in real life, templates are usually kept in separate files rendered with `render_template`). The example also shows how to maintain continuity of state among multiple interactions with the server from the same browser, with the handy `flask.session` variable. (It could alternatively have put together the HTML response in Python code instead of using Jinja, and used a cookie directly instead of the session; however, real-world Flask apps do tend to use Jinja and sessions by preference.)

If this app had multiple view functions, it might want to set `lastvisit` in the session to whatever URL had triggered the request. Here's how to code and decorate a hook function to execute after each request:

```
@app.after_request
def set_lastvisit(response):
    now = datetime.datetime.now()
    flask.session['lastvisit'] = now.ctime()
    return response
```

You can now remove the `flask.session['lastvisit'] = newvisit` statement from the view function `greet`, and the app will keep working fine.

FastAPI

FastAPI is of a more recent design than Flask or Django. While both of the latter have very usable extensions to provide API services, FastAPI aims squarely at producing HTTP-based APIs, as its name suggests. It's also perfectly capable of producing dynamic web pages intended for browser consumption, making it a versatile server. FastAPI's **home page** provides simple, short examples showing how it works and highlighting the advantages, backed up by very thorough and detailed reference documentation.

As type annotations (covered in **Chapter 5**) entered the Python language, they found wider use than originally intended in tools like **`pydantic`**, which uses them to perform runtime parsing and validation. The FastAPI server exploits this support for clean data structures, demonstrating great potential to improve web coding productivity through built-in and tailored conversion and validation of inputs.

FastAPI also relies on **Starlette**, a high-performance asynchronous web framework, which in turn uses an ASGI server such as **Uvicorn** or **Hypercorn**. You don't need to use async techniques directly to take advantage of FastAPI. You can write your application in more traditional Python style, though it might perform even faster if you do switch to the async style.

FastAPI's ability to provide type-accurate APIs (and automatically generated documentation for them) aligned with the types indicated by your annotations means it can provide automatic parsing of incoming data and conversion on both input and output.

Consider the sample code shown in **Example 20-2**, which defines a simple model for both `pydantic` and `mongoengine`. Each has four fields: `name` and `description` are strings, `price` and `tax` are decimal. Values are required for the `name` and `price` fields, but `description` and `tax` are optional. `pydantic` establishes a default value of **`None`** for the latter two fields; `mongoengine` does not store a value for fields whose value is **`None`**.

Example 20-2. models.py: `pydantic` and `mongoengine` data models

```
from decimal import Decimal
from pydantic import BaseModel, Field
from mongoengine import Document, StringField, DecimalField
from typing import Optional

class PItem(BaseModel):
    "pydantic typed data class."
    name: str
    price: Decimal
    description: Optional[str] = None
    tax: Optional[Decimal] = None

class MItem(Document):
    "mongoengine document."
    name = StringField(primary_key=True)
    price = DecimalField()
    description = StringField(required=False)
    tax = DecimalField(required=False)
```

Suppose you wanted to accept such data through a web form or as JSON, and be able to retrieve the data as JSON or display it in HTML. The skeletal Example 20-3 (offering no facilities to maintain existing data) shows you how you might do this with FastAPI. This example uses the Uvicorn HTTP server, but makes no attempt to explicitly use Python's async features. As with Flask, the program begins by creating an application object `app`. This object has decorator methods for each HTTP method, but the `app.route` decorator (while available) is eschewed in favor of `app.get` for HTTP GET, `app.post` for HTTP POST, and the like, and those determine which view function handles requests to the paths for different HTTP methods.

Example 20-3. server.py: FastAPI sample code to accept and display item data

```
from decimal import Decimal
from fastapi import FastAPI, Form
```

```python
from fastapi.responses import HTMLResponse, FileResponse
from mongoengine import connect
from mongoengine.errors import NotUniqueError
from typing import Optional
import json
import uvicorn
from models import PItem, MItem

DATABASE_URI = "mongodb://localhost:27017"
db=DATABASE_URI+"/mydatabase"
connect(host=db)
app = FastAPI()

def save(item):
    try:
        return item.save(force_insert=True)
    except NotUniqueError:
        return None

@app.get('/')
def home_page():
    "View function to display a simple form."
    return FileResponse("index.html")

@app.post("/items/new/form/", response_class=HTMLResponse)
def create_item_from_form(name: str=Form(...),
                          price: Decimal=Form(...),
                          description: Optional[str]=Form(""),
                          tax: Optional[Decimal]=Form(Decimal("0.0"))):
    "View function to accept form data and create an item."
    mongoitem = MItem(name=name, price=price, description=description,
                      tax=tax)
    value = save(mongoitem)
    if value:
        body = f"Item({name!r}, {price!r}, {description!r}, {tax!r})"
    else:
        body = f"Item {name!r} already present."
    return f"""<html><body><h2>{body}</h2></body></html>"""

@app.post("/items/new/")
def create_item_from_json(item: PItem):
    "View function to accept JSON data and create an item."
    mongoitem = MItem(**item.dict())
    value = save(mongoitem)
    if not value:
        return f"Primary key {item.name!r} already present"
    return item.dict()

@app.get("/items/{name}/")
def retrieve_item(name: str):
    "View function to return the JSON contents of an item."
    m_item = MItem.objects(name=name).get()
    return json.loads(m_item.to_json())

if __name__ == "__main__":
    # host as "localhost" or "127.0.0.1" allows only local apps to access the
    # web page. Using "0.0.0.0" will accept access from apps on other hosts,
    # but this can raise security concerns, and is generally not recommended.
    uvicorn.run("__main__:app", host="127.0.0.1", port=8000, reload=True)
```

The home_page function, which takes no arguments, simply renders a minimal HTML home page containing a form from the *index.html* file, shown in **Example 20-4**. The form posts to the */items/new/form/* endpoint, which triggers a call to the `create_item_from_form` function, which is declared in the routing decorator as producing an HTML response rather than the default JSON.

Example 20-4. The index.html file

```
<!DOCTYPE html>
<html lang="en">
  <body>
  <h2>FastAPI Demonstrator</h2>
  <form method="POST" action="/items/new/form/">
    <table>
    <tr><td>Name</td><td><input name="name"></td></tr>
    <tr><td>Price</td><td><input name="price"></td></tr>
    <tr><td>Description</td><td><input name="description"></td></tr>
    <tr><td>Tax</td><td><input name="tax"></td></tr>
    <tr><td></td><td><input type="submit"></td></tr>
    </table>
  </form>
  </body>
</html>
```

The form, shown in **Figure 20-1**, is handled by the `create_item_from_form` function, whose signature takes an argument for each form field, with annotations defining each as a form field. Note that the signature defines its own default values for `description` and `tax`. The function creates an `MItem` object from the form data and tries to save it in the database. The `save` function forces insertions, inhibiting the update of an existing record, and reports failure by returning **`None`**; the return value is used to formulate a simple HTML reply. In a production application, a templating engine such as Jinja would typically be used to render the response.

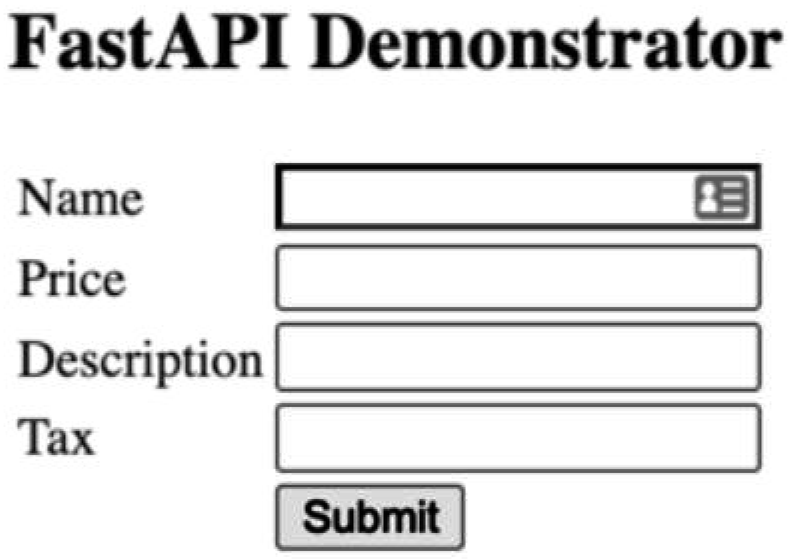

Figure 20-1. Input form for FastAPI Demonstrator

The `create_item_from_json` function, routed from the */items/new/* end-point, takes JSON input from a POST request. Its signature accepts a `pydantic` record, so in this case, FastAPI will use `pydantic`'s validation to determine whether the input is acceptable. The function returns a Python dictionary, which FastAPI automatically converts to a JSON response. This can easily be tested with a simple client, shown in **Example 20-5**.

Example 20-5. FastAPI test client

```
import requests, json
```

```
result = requests.post('http://localhost:8000/items/new/',
                       json={"name": "Item1",
                             "price": 12.34,
                             "description": "Rusty old bucket"})
print(result.status_code, result.json())
result = requests.get('http://localhost:8000/items/Item1/')
print(result.status_code, result.json())
result = requests.post('http://localhost:8000/items/new/',
                       json={"name": "Item2",
                             "price": "Not a number"})
print(result.status_code, result.json())
```

The results of running this program are as follows:

```
200 {'name': 'Item1', 'price': 12.34, 'description': 'Rusty old
bucket'> 'tax': None}
200 {'_id': 'Item1', 'price': 12.34, 'description': 'Rusty old bucket'}
422 {'detail': [{'loc': ['body', 'price'], 'msg': 'value is not a valid
decimal', 'type': 'type_error.decimal'}]}
```

The first POST request to */items/new/* sees the server returning the same data it was presented with, confirming that it has been saved in the database. Note that the `tax` field was not supplied, so the `pydantic` default value is used here. The second line shows the output from retrieving the newly stored item (`mongoengine` identifies the primary key using the name `_id`). The third line shows an error message, generated by the attempt to store a nonnumeric value in the `price` field.

Finally, the `retrieve_item` view function, routed from URLs such as */items/Item1/*, extracts the key as the second path element and returns the JSON representation of the given item. It looks up the given key in `mongoengine` and converts the returned record to a dictionary that is rendered as JSON by FastAPI.

1 One historical legacy is that, in CGI, a server provided the CGI script with information about the HTTP request to be served mostly via the operating system's envi-

ronment (in Python, that's `os.environ`); to this day, interfaces between web servers and application frameworks rely on "an environment" that's essentially a dictionary and generalizes and speeds up the same fundamental idea.

2 More **advanced versions of HTTP exist**, but we do not cover them in this book.

3 Please don't. As Titus Brown once pointed out, Python is (in)famous for having more web frameworks than keywords. One of this book's authors once showed Guido how to easily fix that problem when he was first designing Python 3—just add a few hundred new keywords—but, for some reason, Guido was not very receptive to this suggestion.

4 Installing uWSGI on Windows currently requires compiling it with Cygwin.

5 Since Python has fewer than 40 keywords, you can see why Titus Brown once pointed out that Python has more web frameworks than keywords.