Chapter 15. Data in Time: Processes and Concurrency

> *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

This chapter and the next two are a bit more challenging than earlier ones. In this one we cover data in time (sequential and concurrent access on a single computer), and following that we look at data in a box (storage and retrieval with special files and databases) in Chapter 16 and then data in space (networking) in Chapter 17.

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 (macOS), Task Manager on Windows-based computers, or the `top` command in Linux.

You can also access process data from your own programs. The standard library's `os` module provides a common way of accessing some system infor-

mation. For instance, the following functions get the *process ID* and the *current working directory* of the running Python interpreter:

```
>>> import os
>>> os.getpid()
76051
>>> os.getcwd()
'/Users/williamlubanovic'
```

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

```
>>> os.getuid()
501
>>> os.getgid()
20
```

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 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". 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 returns standard output only 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 21:33:11 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:

```
>>> ret = subprocess.call('date -u', shell=True)
Tue Jan 21 04:40:04 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 04:41:59 UTC 2014
```

Create a Process with multiprocessing

You can run a Python function as a separate process, or even create multiple independent processes with the `multiprocessing` module. The sample code in Example 15-1 is short and simple; save it as *mp.py* and then run it by typing `python mp.py`:

Example 15-1. mp.py

```
import multiprocessing
import os

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=whoami,
          args=("I'm function %s" % n,))
        p.start()
```

When I run this, my output looks like this:

```
Process 6224 says: I'm the main program
Process 6225 says: I'm function 0
Process 6226 says: I'm function 1
Process 6227 says: I'm function 2
Process 6228 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" 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 Example 15-2, 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.

Example 15-2. mp2.py

```
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!
```

Get System Info with os

The standard `os` package provides a lot of details on your system, and lets you control some of it if you run your Python script as a privileged user (root or administrator). Besides file and directory functions that are covered in Chapter 14, it has informational functions like these (run on an iMac):

```
>>> import os
>>> os.uname()
posix.uname_result(sysname='Darwin',
nodename='iMac.local',
release='18.5.0',
```

```
version='Darwin Kernel Version 18.5.0: Mon Mar 11 20:40:32 PDT 2019;
  root:xnu-4903.251.3~3/RELEASE_X86_64',
machine='x86_64')
>>> os.getloadavg()
(1.794921875, 1.93115234375, 2.2587890625)
>>> os.cpu_count()
4
```

A useful function is `system()`, which executes a command string as though you typed it at a terminal:

```
>>> import os
>>> os.system('date -u')
Tue Apr 30 13:10:09 UTC 2019
0
```

It's a grab bag. See the docs for interesting tidbits.

Get Process Info with psutil

The third-party package psutil also provides system and process information for Linux, Unix, macOS, and Windows systems.

You can guess how to install it:

```
$ pip install psutil
```

Coverage includes the following:

System
: CPU, memory, disk, network, sensors

Processes
: id, parent id, CPU, memory, open files, threads

We already saw (in the previous `os` discussion) that my computer has four CPUs. How much time (in seconds) have they been using?

```
>>> import psutil
>>> psutil.cpu_times(True)
[scputimes(user=62306.49, nice=0.0, system=19872.71, idle=256097.64),
scputimes(user=19928.3, nice=0.0, system=6934.29, idle=311407.28),
scputimes(user=57311.41, nice=0.0, system=15472.99, idle=265485.56),
scputimes(user=14399.49, nice=0.0, system=4848.84, idle=319017.87)]
```

And how busy are they now?

```
>>> import psutil
>>> psutil.cpu_percent(True)
26.1
>>> psutil.cpu_percent(percpu=True)
[39.7, 16.2, 50.5, 6.0]
```

You might never need this kind of data, but it's good to know where to look if you do.

Command Automation

You often run commands from the shell (either with manually typed commands or shell scripts), but Python has more than one good third-party management tool.

A related topic, *task queues*, is discussed in "Queues".

Invoke

Version 1 of the `fabric` tool let you define local and remote (networked) tasks with Python code. The developers split this original package into `fabric2` (remote) and `invoke` (local).

Install `invoke` by running the following:

```
$ pip install invoke
```

One use of `invoke` is to make functions available as command-line arguments. Let's make a *tasks.py* file with the lines shown in Example 15-3.

Example 15-3. tasks.py

```
from invoke import task

@task
def mytime(ctx):
    import time
    now = time.time()
    time_str = time.asctime(time.localtime(now))
    print("Local time is", timestr)
```

(That `ctx` argument is the first argument for each task function, but it's used only internally by `invoke`. It doesn't matter what you call it, but an argument needs to be there.)

```
$ invoke mytime
Local time is Thu May  2 13:16:23 2019
```

Use the argument `-l` or `--list` to see what tasks are available:

```
$ invoke -l
Available tasks:

  mytime
```

Tasks can have arguments, and you can invoke multiple tasks at one time from the command line (similar to `&&` use in shell scripts).

Other uses include:

- Running local shell commands with the `run()` function
- Responding to string output patterns of programs

This was a brief glimpse. See the docs for all the details.

Other Command Helpers

These Python packages have some similarity to `invoke`, but one or another may be a better fit when you need it:

- `click`
- `doit`
- `sh`
- `delegator`
- `pypeln`

Concurrency

The official Python site discusses concurrency in general and in the standard library. Those pages have many links to various packages and techniques; in this chapter, we show the most useful ones.

In computers, if you're waiting for something, it's usually for one of two reasons:

I/O bound
: This is by far the most common. Computer CPUs are ridiculously fast—hundreds of times faster than computer memory and many thousands of times faster than disks or networks.

CPU bound
: The CPU keeps busy. This happens with *number crunching* tasks such as scientific or graphic calculations.

Two more terms are related to concurrency:

Synchronous
: One thing follows the other, like a line of goslings behind their parents.

Asynchronous
: Tasks are independent, like random geese splashing down in a pond.

As you progress from simple systems and tasks to real-life problems, you'll need at some point to deal with concurrency. Consider a website, for example. You can usually provide static and dynamic pages to web clients fairly quickly. A fraction of a second is considered interactive, but if the display or interaction takes longer people become impatient. Tests by companies such as Google and Amazon showed that traffic drops off quickly if the page loads even a little slower.

But what if you can't help it when something takes a long time, such as uploading a file, resizing an image, or querying a database? You can't do it within your synchronous web server code anymore, because someone's waiting.

On a single machine, if you want to perform multiple tasks as fast as possible, you want to make them independent. Slow tasks shouldn't block all the others.

This chapter showed earlier how multiprocessing can be used to overlap work on a single machine. If you needed to resize an image, your web server code could call a separate, dedicated-image resizing process to run asynchronously and concurrently. It could scale your application horizontally by invoking multiple resizing processes.

The trick is getting them all to work with one another. Any shared control or state means that there will be bottlenecks. An even bigger trick is dealing with failures, because concurrent computing is harder than regular computing. Many more things can go wrong, and your odds of end-to-end success are lower.

All right. What methods can help you to deal with these complexities? Let's begin with a good way to manage multiple tasks: *queues*.

Queues

A queue is like a list: things are added at one end and taken away from the other. The most common is referred to as *FIFO* (first in, first out).

Suppose that you're washing dishes. If you're stuck with the entire job, you need to wash each dish, dry it, and put it away. You can do this in a number of ways. You might wash the first dish, dry it, and then put it away. You then repeat with the second dish, and so on. Or, you might *batch* operations and wash all the dishes, dry them all, and then put them away; this assumes you have space in your sink and drainer for all the dishes that accumulate at each step. These are all synchronous approaches—one worker, one thing at a time.

As an alternative, you could get a helper or two. If you're the washer, you can hand each cleaned dish to the dryer, who hands each dried dish to the put-away-er. As long as each of you works at the same pace, you should finish much faster than by yourself.

However, what if you wash faster than the dryer dries? Wet dishes either fall on the floor, or you pile them up between you and the dryer, or you just whistle off-key until the dryer is ready. And if the last person is slower than the dryer, dry dishes can end up falling on the floor, or piling up, or the dryer does the whistling. You have multiple workers, but the overall task is still synchronous and can proceed only as fast as the slowest worker.

Many hands make light work, goes the old saying (I always thought it was Amish, because it makes me think of barn building). Adding workers can build a barn or do the dishes, faster. This involves *queues*.

In general, queues transport *messages*, which can be any kind of information. In this case, we're interested in queues for distributed task management, also known as *work queues*, *job queues*, or *task queues*. Each dish in the sink is given to an available washer, who washes and hands it off to the first available dryer, who dries and hands it to a put-away-er. This can be synchronous (workers wait for a dish to handle and another worker to whom to give it), or asynchronous (dishes are stacked between workers with different paces). As long as you have enough workers, and they keep up with the dishes, things move a lot faster.

Processes

You can implement queues in many ways. For a single machine, the standard library's `multiprocessing` module (which you saw earlier) contains a `Queue` function. Let's simulate just a single washer and multiple dryer processes (someone can put the dishes away later) and an intermediate `dish_queue`. Call this program *dishes.py* (Example 15-4).

Example 15-4. dishes.py

```
import multiprocessing as mp

def washer(dishes, output):
    for dish in dishes:
        print('Washing', dish, 'dish')
        output.put(dish)

def dryer(input):
    while True:
        dish = input.get()
        print('Drying', dish, 'dish')
        input.task_done()

dish_queue = mp.JoinableQueue()
dryer_proc = mp.Process(target=dryer, args=(dish_queue,))
dryer_proc.daemon = True
dryer_proc.start()

dishes = ['salad', 'bread', 'entree', 'dessert']
washer(dishes, dish_queue)
dish_queue.join()
```

Run your new program, thusly:

```
$ python dishes.py
Washing salad dish
Washing bread dish
Washing entree dish
Washing dessert dish
Drying salad dish
```

```
Drying bread dish
Drying entree dish
Drying dessert dish
```

This queue looked a lot like a simple Python iterator, producing a series of dishes. It actually started up separate processes along with the communication between the washer and dryer. I used a `JoinableQueue` and the final `join()` method to let the washer know that all the dishes have been dried. There are other queue types in the `multiprocessing` module, and you can read the documentation for more examples.

Threads

A *thread* runs within a process with access to everything in the process, similar to a multiple personality. The `multiprocessing` module has a cousin called `threading` that uses threads instead of processes (actually, `multiprocessing` was designed later as its process-based counterpart). Let's redo our process example with threads, as shown in Example 15-5.

Example 15-5. thread1.py

```
import threading

def do_this(what):
    whoami(what)

def whoami(what):
    print("Thread %s says: %s" % (threading.current_thread(), what))

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

Here's what prints for me:

```
Thread <_MainThread(MainThread, started 140735207346960)> says: I
program
Thread <Thread(Thread-1, started 4326629376)> says: I'm function
Thread <Thread(Thread-2, started 4342157312)> says: I'm function
Thread <Thread(Thread-3, started 4347412480)> says: I'm function
Thread <Thread(Thread-4, started 4342157312)> says: I'm function
```

We can reproduce our process-based dish example by using threads, as shown in Example 15-6.

Example 15-6. thread_dishes.py

```
import threading, queue
import time

def washer(dishes, dish_queue):
    for dish in dishes:
        print ("Washing", dish)
        time.sleep(5)
        dish_queue.put(dish)

def dryer(dish_queue):
    while True:
        dish = dish_queue.get()
        print ("Drying", dish)
        time.sleep(10)
        dish_queue.task_done()


dish_queue = queue.Queue()
for n in range(2):
    dryer_thread = threading.Thread(target=dryer, args=(dish_queue,))
    dryer_thread.start()

dishes = ['salad', 'bread', 'entree', 'dessert']
washer(dishes, dish_queue)
dish_queue.join()
```

One difference between `multiprocessing` and `threading` is that `threading` does not have a `terminate()` function. There's no easy way to

terminate a running thread, because it can cause all sorts of problems in your code, and possibly in the space-time continuum itself.

Threads can be dangerous. Like manual memory management in languages such as C and C++, they can cause bugs that are extremely hard to find, let alone fix. To use threads, all the code in the program (and in external libraries that it uses) must be *thread safe*. In the preceding example code, the threads didn't share any global variables, so they could run independently without breaking anything.

Imagine that you're a paranormal investigator in a haunted house. Ghosts roam the halls, but none are aware of the others, and at any time, any of them can view, add, remove, or move any of the house's contents.

You're walking apprehensively through the house, taking readings with your impressive instruments. Suddenly you notice that the candlestick you passed seconds ago is now missing.

The contents of the house are like the variables in a program. The ghosts are threads in a process (the house). If the ghosts only cast spectral glances at the house's contents, there would be no problem. It's like a thread reading the value of a constant or variable without trying to change it.

Yet, some unseen entity could grab your flashlight, blow cold air down your neck, put marbles on the stairs, or make the fireplace come ablaze. The *really* subtle ghosts would change things in other rooms that you might never notice.

Despite your fancy instruments, you'd have a very hard time figuring out who did what, how, when, and where.

If you used multiple processes instead of threads, it would be like having a number of houses but with only one (living) person in each. If you put your brandy in front of the fireplace, it would still be there an hour later—some lost to evaporation, but in the same place.

Threads can be useful and safe when global data is not involved. In particular, threads are useful for saving time while waiting for some I/O operation to

complete. In these cases, they don't have to fight over data, because each has completely separate variables.

But threads do sometimes have good reasons to change global data. In fact, one common reason to launch multiple threads is to let them divide up the work on some data, so a certain degree of change to the data is expected.

The usual way to share data safely is to apply a software *lock* before modifying a variable in a thread. This keeps the other threads out while the change is made. It's like having a Ghostbuster guard the room you want to remain unhaunted. The trick, though, is that you need to remember to unlock it. Plus, locks can be nested: what if another Ghostbuster is also watching the same room, or the house itself? The use of locks is traditional but hard to get right.

NOTE

In Python, threads do not speed up CPU-bound tasks because of an implementation detail in the standard Python system called the *Global Interpreter Lock* (GIL). This exists to avoid threading problems in the Python interpreter, and can actually make a multithreaded program slower than its single-threaded counterpart, or even a multiprocess version.

So for Python, the recommendations are as follows:

- Use threads for I/O-bound problems
- Use processes, networking, or events (discussed in the next section) for CPU-bound problems

concurrent.futures

As you've just seen, using threads or multiple processes involves a number of details. The `concurrent.futures` module was added to the Python 3.2 standard library to simplify these. It lets you schedule an asynchronous pool of workers, using threads (when I/O-bound) or processes (when CPU-bound). You get back a *future* to track their state and collect the results.

Example 15-7 contains a test program that you can save as *cf.py*. The task function `calc()` sleeps for one second (our way of faking being busy with something), calculates the square root of its argument, and returns it. The program takes an optional command-line argument of the number of workers to use, which defaults to 3. It starts this number of workers in a thread pool, then a process pool, and then prints the elapsed times. The `values` list contains five numbers, sent to `calc()` one at time in a worker thread or process.

Example 15-7. cf.py

```
from concurrent import futures
import math
import time
import sys

def calc(val):
    time.sleep(1)
    result = math.sqrt(float(val))
    return result

def use_threads(num, values):
    t1 = time.time()
    with futures.ThreadPoolExecutor(num) as tex:
        results = tex.map(calc, values)
    t2 = time.time()
    return t2 - t1

def use_processes(num, values):
    t1 = time.time()
    with futures.ProcessPoolExecutor(num) as pex:
        results = pex.map(calc, values)
    t2 = time.time()
    return t2 - t1

def main(workers, values):
    print(f"Using {workers} workers for {len(values)} values")
    t_sec = use_threads(workers, values)
    print(f"Threads took {t_sec:.4f} seconds")
    p_sec = use_processes(workers, values)
    print(f"Processes took {p_sec:.4f} seconds")
```

```
if __name__ == '__main__':
    workers = int(sys.argv[1])
    values = list(range(1, 6)) # 1 .. 5
    main(workers, values)
```

Here are some results that I got:

```
$ python cf.py 1
Using 1 workers for 5 values
Threads took 5.0736 seconds
Processes took 5.5395 seconds
$ python cf.py 3
Using 3 workers for 5 values
Threads took 2.0040 seconds
Processes took 2.0351 seconds
$ python cf.py 5
Using 5 workers for 5 values
Threads took 1.0052 seconds
Processes took 1.0444 seconds
```

That one-second `sleep()` forced each worker to take a second for each calculation:

- With only one worker at a time, everything was serial, and the total time was more than five seconds.
- Five workers matched the size of the values being tested, so we had an elapsed time just more than a second.
- With three workers, we needed two runs to handle all five values, so two seconds elapsed.

In the program, I ignored the actual `results` (the square roots that we calculated) to emphasize the elapsed times. Also, using `map()` to define the pool causes us to wait for all workers to finish before returning `results`. If you wanted to get each result as it completed, let's try another test (call it *cf2.py*) in which each worker returns the value and its square root as soon as it calculates it (Example 15-8).

Example 15-8. cf2.py

```
from concurrent import futures
import math
import sys

def calc(val):
    result = math.sqrt(float(val))
    return val, result

def use_threads(num, values):
    with futures.ThreadPoolExecutor(num) as tex:
        tasks = [tex.submit(calc, value) for value in values]
        for f in futures.as_completed(tasks):
            yield f.result()

def use_processes(num, values):
    with futures.ProcessPoolExecutor(num) as pex:
        tasks = [pex.submit(calc, value) for value in values]
        for f in futures.as_completed(tasks):
            yield f.result()

def main(workers, values):
    print(f"Using {workers} workers for {len(values)} values")
    print("Using threads:")
    for val, result in use_threads(workers, values):
        print(f'{val} {result:.4f}')
    print("Using processes:")
    for val, result in use_processes(workers, values):
        print(f'{val} {result:.4f}')

if __name__ == '__main__':
    workers = 3
    if len(sys.argv) > 1:
        workers = int(sys.argv[1])
    values = list(range(1, 6)) # 1 .. 5
    main(workers, values)
```

Our `use_threads()` and `use_processes()` functions are now generator functions that call `yield` to return on each iteration. From one run on my

machine, you can see how the workers don't always finish `1` through `5` in order:

```
$ python cf2.py 5
Using 5 workers for 5 values
Using threads:
3 1.7321
1 1.0000
2 1.4142
4 2.0000
5 2.2361
Using processes:
1 1.0000
2 1.4142
3 1.7321
4 2.0000
5 2.2361
```

You can use `concurrent.futures` any time you want to launch a bunch of concurrent tasks, such as the following:

- Crawling URLs on the web
- Processing files, such as resizing images
- Calling service APIs

As usual, the docs provide additional details, but are much more technical.

Green Threads and gevent

As you've seen, developers traditionally avoid slow spots in programs by running them in separate threads or processes. The Apache web server is an example of this design.

One alternative is *event-based* programming. An event-based program runs a central *event loop*, doles out any tasks, and repeats the loop. The NGINX web server follows this design, and is generally faster than Apache.

The `gevent` library is event-based and accomplishes a neat trick: you write normal imperative code, and it magically converts pieces to *coroutines*. These

are like generators that can communicate with one another and keep track of where they are. `gevent` modifies many of Python's standard objects such as `socket` to use its mechanism instead of blocking. This does not work with Python add-in code that was written in C, as some database drivers are.

You install `gevent` by using `pip`:

```
$ pip install gevent
```

Here's a variation of sample code at the `gevent` website. You'll see the `socket` module's `gethostbyname()` function in the upcoming DNS section. This function is synchronous, so you wait (possibly many seconds) while it chases name servers around the world to look up that address. But you could use the `gevent` version to look up multiple sites independently. Save this as *gevent_test.py* (Example 15-9).

Example 15-9. gevent_test.py

```
import gevent
from gevent import socket
hosts = ['www.crappytaxidermy.com', 'www.walterpottertaxidermy.com',
    'www.antique-taxidermy.com']
jobs = [gevent.spawn(gevent.socket.gethostbyname, host) for host in hosts]
gevent.joinall(jobs, timeout=5)
for job in jobs:
    print(job.value)
```

There's a one-line for-loop in the preceding example. Each hostname is submitted in turn to a `gethostbyname()` call, but they can run asynchronously because it's the `gevent` version of `gethostbyname()`.

Run *gevent_test.py*:

```
$ python gevent_test.py
66.6.44.4
74.125.142.121
78.136.12.50
```

`gevent.spawn()` creates a *greenlet* (also known sometimes as a *green thread* or a *microthread*) to execute each `gevent.socket.gethostbyname(url)`.

The difference from a normal thread is that it doesn't block. If something occurred that would have blocked a normal thread, `gevent` switches control to one of the other greenlets.

The `gevent.joinall()` method waits for all the spawned jobs to finish. Finally, we dump the IP addresses that we got for these hostnames.

Instead of the `gevent` version of `socket`, you can use its evocatively named *monkey-patching* functions. These modify standard modules such as `socket` to use greenlets rather than calling the `gevent` version of the module. This is useful when you want `gevent` to be applied all the way down, even into code that you might not be able to access.

At the top of your program, add the following call:

```
from gevent import monkey
monkey.patch_socket()
```

This inserts the `gevent` socket everywhere the normal `socket` is called, anywhere in your program, even in the standard library. Again, this works only for Python code, not libraries written in C.

Another function monkey-patches even more standard library modules:

```
from gevent import monkey
monkey.patch_all()
```

Use this at the top of your program to get as many `gevent` speedups as possible.

Save this program as *gevent_monkey.py* (Example 15-9).

Example 15-10. gevent_monkey.py

```
import gevent
from gevent import monkey; monkey.patch_all()
import socket
hosts = ['www.crappytaxidermy.com', 'www.walterpottertaxidermy.com',
    'www.antique-taxidermy.com']
jobs = [gevent.spawn(socket.gethostbyname, host) for host in hosts]
gevent.joinall(jobs, timeout=5)
for job in jobs:
    print(job.value)
```

Again, run the program:

```
$ python gevent_monkey.py
66.6.44.4
74.125.192.121
78.136.12.50
```

There are potential dangers when using `gevent`. As with any event-based system, each chunk of code that you execute should be relatively quick. Although it's nonblocking, code that does a lot of work is still slow.

The very idea of monkey-patching makes some people nervous. Yet, many large sites such as Pinterest use `gevent` to speed up their sites significantly. Like the fine print on a bottle of pills, use `gevent` as directed.

For more examples, see this thorough gevent tutorial.

NOTE

You might also want to consider `tornado` or `gunicorn`, two other popular event-driven frameworks. They provide both the low-level event handling and a fast web server. They're worth a look if you'd like to build a fast website without messing with a traditional web server such as Apache.

---

twisted

`twisted` is an asynchronous, event-driven networking framework. You connect functions to events such as data received or connection closed, and those functions are called when those events occur. This is a *callback* design, and if you've written anything in JavaScript, it might seem familiar. If it's new to you, it can seem backwards. For some developers, callback-based code becomes harder to manage as the application grows.

You install it by running the following:

```
$ pip install twisted
```

`twisted` is a large package, with support for many internet protocols on top of TCP and UDP. To be short and simple, we show a little knock-knock server and client, adapted from twisted examples. First, let's look at the server, *knock_server.py*: (Example 15-11).

Example 15-11. knock_server.py

```
from twisted.internet import protocol, reactor

class Knock(protocol.Protocol):
    def dataReceived(self, data):
        print('Client:', data)
        if data.startswith("Knock knock"):
            response = "Who's there?"
        else:
            response = data + " who?"
        print('Server:', response)
        self.transport.write(response)

class KnockFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return Knock()

reactor.listenTCP(8000, KnockFactory())
reactor.run()
```

Now let's take a glance at its trusty companion, *knock_client.py* (Example 15-12).

Example 15-12. knock_client.py

```
from twisted.internet import reactor, protocol

class KnockClient(protocol.Protocol):
    def connectionMade(self):
        self.transport.write("Knock knock")

    def dataReceived(self, data):
        if data.startswith("Who's there?"):
            response = "Disappearing client"
            self.transport.write(response)
        else:
            self.transport.loseConnection()
            reactor.stop()

class KnockFactory(protocol.ClientFactory):
    protocol = KnockClient

def main():
    f = KnockFactory()
    reactor.connectTCP("localhost", 8000, f)
    reactor.run()

if __name__ == '__main__':
    main()
```

Start the server first:

```
$ python knock_server.py
```

Then, start the client:

```
$ python knock_client.py
```

The server and client exchange messages, and the server prints the conversation:

```
Client: Knock knock
Server: Who's there?
Client: Disappearing client
Server: Disappearing client who?
```

Our trickster client then ends, keeping the server waiting for the punch line.

If you'd like to enter the `twisted` passages, try some of the other examples from its documentation.

asyncio

Python added the `asyncio` library in version 3.4. It's a way of defining concurrent code using the new `async` and `await` capabilities. It's a big topic with many details. To avoid overstuffing this chapter, I've moved the discussion of `asyncio` and related topics to Appendix C.

Redis

Our earlier dishwashing code examples, using processes or threads, were run on a single machine. Let's take another approach to queues that can run on a single machine or across a network. Even with multiple singing processes and dancing threads, sometimes one machine isn't enough, You can treat this section as a bridge between single-box (one machine) and multiple-box concurrency.

To try the examples in this section, you'll need a Redis server and its Python module. You can see where to get them in "Redis". In that chapter, Redis's role is that of a database. Here, we're featuring its concurrency personality.

A quick way to make a queue is with a Redis list. A Redis server runs on one machine; this can be the same one as its clients, or another that the clients can access through a network. In either case, clients talk to the server via TCP, so they're networking. One or more provider clients pushes messages onto

one end of the list. One or more client workers watches this list with a *blocking pop* operation. If the list is empty, they all just sit around playing cards. As soon as a message arrives, the first eager worker gets it.

Like our earlier process- and thread-based examples, *redis_washer.py* generates a sequence of dishes (Example 15-13).

Example 15-13. redis_washer.py

```
import redis
conn = redis.Redis()
print('Washer is starting')
dishes = ['salad', 'bread', 'entree', 'dessert']
for dish in dishes:
    msg = dish.encode('utf-8')
    conn.rpush('dishes', msg)
    print('Washed', dish)
conn.rpush('dishes', 'quit')
print('Washer is done')
```

The loop generates four messages containing a dish name, followed by a final message that says "quit." It appends each message to a list called `dishes` in the Redis server, similar to appending to a Python list.

And as soon as the first dish is ready, *redis_dryer.py* does its work (Example 15-14).

Example 15-14. redis_dryer.py

```
import redis
conn = redis.Redis()
print('Dryer is starting')
while True:
    msg = conn.blpop('dishes')
    if not msg:
        break
    val = msg[1].decode('utf-8')
    if val == 'quit':
        break
```

```
        print('Dried', val)
    print('Dishes are dried')
```

This code waits for messages whose first token is “dishes” and prints that each one is dried. It obeys the *quit* message by ending the loop.

Start the dryer and then the washer. Using the `&` at the end puts the first program in the *background*; it keeps running, but doesn’t listen to the keyboard anymore. This works on Linux, macOS, and Windows, although you might see different output on the next line. In this case (macOS), it’s some information about the background dryer process. Then, we start the washer process normally (in the *foreground*). You’ll see the mingled output of the two processes:

```
$ python redis_dryer.py &
[2] 81691
Dryer is starting
$ python redis_washer.py
Washer is starting
Washed salad
Dried salad
Washed bread
Dried bread
Washed entree
Dried entree
Washed dessert
Washer is done
Dried dessert
Dishes are dried
[2]+  Done                    python redis_dryer.py
```

As soon as dish IDs started arriving at Redis from the washer process, our hard-working dryer process started pulling them back out. Each dish ID was a number, except the final *sentinel* value, the string `'quit'`. When the dryer process read that `quit` dish ID, it quit, and some more background process information printed to the terminal (also system dependent). You can use a sentinel (an otherwise invalid value) to indicate something special from the data stream itself—in this case, that we’re done. Otherwise, we’d need to add a lot more program logic, such as the following:

- Agreeing ahead of time on some maximum dish number, which would kind of be a sentinel anyway.
- Doing some special *out-of-band* (not in the data stream) interprocess communication.
- Timing out after some interval with no new data.

Let's make a few last changes:

- Create multiple `dryer` processes.
- Add a timeout to each dryer rather than looking for a sentinel.

The new *redis_dryer2.py* is shown in Example 15-15.

Example 15-15. redis_dryer2.py

```
def dryer():
    import redis
    import os
    import time
    conn = redis.Redis()
    pid = os.getpid()
    timeout = 20
    print('Dryer process %s is starting' % pid)
    while True:
        msg = conn.blpop('dishes', timeout)
        if not msg:
            break
        val = msg[1].decode('utf-8')
        if val == 'quit':
            break
        print('%s: dried %s' % (pid, val))
        time.sleep(0.1)
    print('Dryer process %s is done' % pid)

import multiprocessing
DRYERS=3
for num in range(DRYERS):
    p = multiprocessing.Process(target=dryer)
    p.start()
```

Start the dryer processes in the background and then the washer process in the foreground:

```
$ python redis_dryer2.py &
Dryer process 44447 is starting
Dryer process 44448 is starting
Dryer process 44446 is starting
$ python redis_washer.py
Washer is starting
Washed salad
44447: dried salad
Washed bread
44448: dried bread
Washed entree
44446: dried entree
Washed dessert
Washer is done
44447: dried dessert
```

One dryer process reads the `quit` ID and quits:

```
Dryer process 44448 is done
```

After 20 seconds, the other dryer processes get a return value of `None` from their `blpop` calls, indicating that they've timed out. They say their last words and exit:

```
Dryer process 44447 is done
Dryer process 44446 is done
```

After the last dryer subprocess quits, the main dryer program ends:

```
[1]+  Done                    python redis_dryer2.py
```

Beyond Queues

With more moving parts, there are more possibilities for our lovely assembly lines to be disrupted. If we need to wash the dishes from a banquet, do we have enough workers? What if the dryers get drunk? What if the sink clogs? Worries, worries!

How will you cope with it all? Common techniques include these:

Fire and forget
: Just pass things on and don't worry about the consequences, even if no one is there. That's the dishes-on-the-floor approach.

Request-reply
: The washer receives an acknowledgment from the dryer, and the dryer from the put-away-er, for each dish in the pipeline.

Back pressure or throttling
: This technique directs a fast worker to take it easy if someone downstream can't keep up.

In real systems, you need to be careful that workers are keeping up with the demand; otherwise, you hear the dishes hitting the floor. You might add new tasks to a *pending* list, while some worker process pops the latest message and adds it to a *working* list. When the message is done, it's removed from the working list and added to a *completed* list. This lets you know what tasks have failed or are taking too long. You can do this with Redis yourself, or use a system that someone else has already written and tested. Some Python-based queue packages that add this extra level of management include:

- `celery` can execute distributed tasks synchronously or asynchronously, using the methods we've discussed: `multiprocessing`, `gevent`, and others.
- rq is a Python library for job queues, also based on Redis.

Queues offers a discussion of queuing software, Python-based and otherwise.

Coming Up

In this chapter, we flowed data through processes. In the next chapter, you'll see how to store and retrieve data in various file formats and databases.

Things to Do

15.1 Use `multiprocessing` to create three separate processes. Make each one wait a random number of seconds between zero and one, print the current time, and then exit.