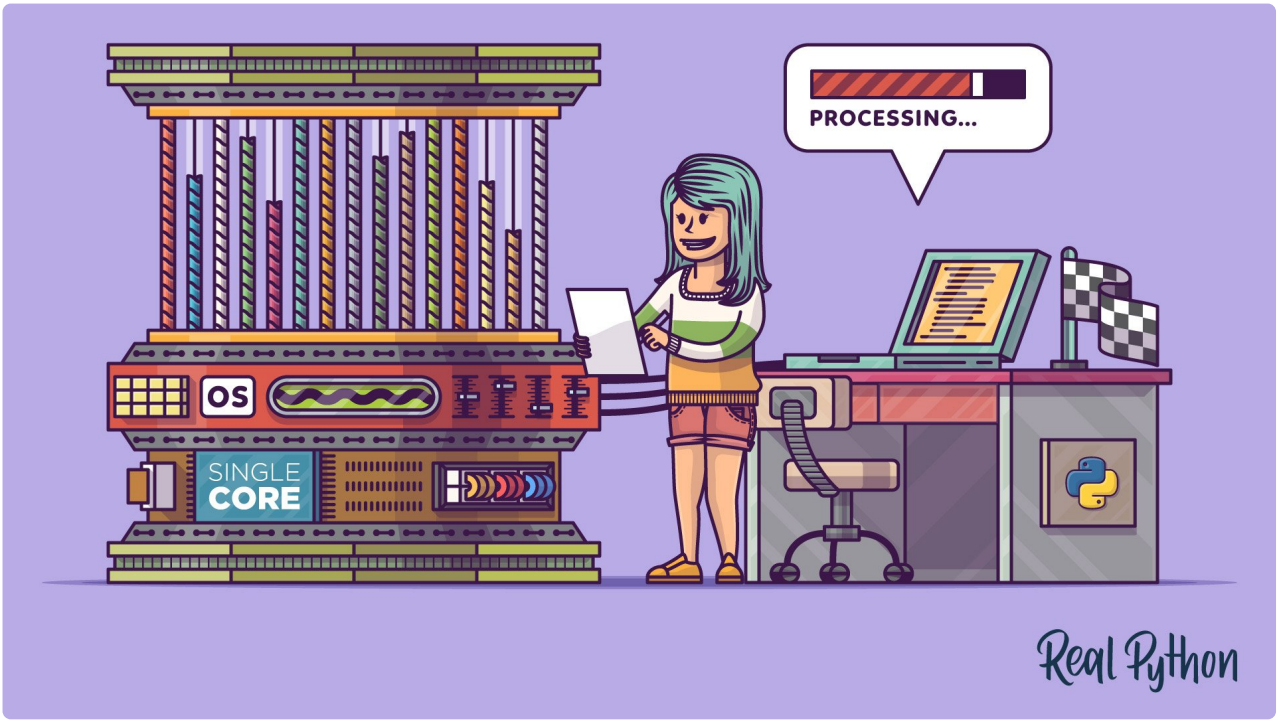
An Intro to Threading in Python

realpython.com/intro-to-python-threading/



by Jim Anderson best-practices intermediate

> Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Threading in Python**

Python threading allows you to have different parts of your program run concurrently and can simplify your design. If you've got some experience in Python and want to speed up your program using threads, then this tutorial is for you!

In this article, you'll learn:

- What threads are
- How to create threads and wait for them to finish
- How to use a `ThreadPoolExecutor`
- How to avoid race conditions
- How to use the common tools that Python `threading` provides

This article assumes you've got the Python basics down pat and that you're using at least version 3.6 to run the examples. If you need a refresher, you can start with the Python Learning Paths and get up to speed.

If you're not sure if you want to use Python `threading`, `asyncio`, or `multiprocessing`, then you can check out Speed Up Your Python Program With Concurrency.

All of the sources used in this tutorial are available to you in the *Real Python* GitHub repo.

Free Bonus: 5 Thoughts On Python Mastery, a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

Take the Quiz: Test your knowledge with our interactive "Python Threading" quiz. Upon completion you will receive a score so you can track your learning progress over time:

Take the Quiz »

What Is a Thread?

A thread is a separate flow of execution. This means that your program will have two things happening at once. But for most Python 3 implementations the different threads do not actually execute at the same time: they merely appear to.

It's tempting to think of threading as having two (or more) different processors running on your program, each one doing an independent task at the same time. That's almost right. The threads may be running on different processors, but they will only be running one at a time.

Getting multiple tasks running simultaneously requires a non-standard implementation of Python, writing some of your code in a different language, or using `multiprocessing` which comes with some extra overhead.

Because of the way CPython implementation of Python works, threading may not speed up all tasks. This is due to interactions with the GIL that essentially limit one Python thread to run at a time.

Tasks that spend much of their time waiting for external events are generally good candidates for threading. Problems that require heavy CPU computation and spend little time waiting for external events might not run faster at all.

This is true for code written in Python and running on the standard CPython implementation. If your threads are written in C they have the ability to release the GIL and run concurrently. If you are running on a different Python implementation, check with the documentation too see how it handles threads.

If you are running a standard Python implementation, writing in only Python, and have a CPU-bound problem, you should check out the `multiprocessing` module instead.

Architecting your program to use threading can also provide gains in design clarity. Most of the examples you'll learn about in this tutorial are not necessarily going to run faster because they use threads. Using threading in them helps to make the design

cleaner and easier to reason about.

So, let's stop talking about threading and start using it!

Starting a Thread

Now that you've got an idea of what a thread is, let's learn how to make one. The Python standard library provides threading, which contains most of the primitives you'll see in this article. `Thread`, in this module, nicely encapsulates threads, providing a clean interface to work with them.

To start a separate thread, you create a `Thread` instance and then tell it to `.start()`:

```
 1import logging
 2import threading
 3import time
 4
 5def thread_function(name):
 6    logging.info("Thread %s: starting", name)
 7    time.sleep(2)
 8    logging.info("Thread %s: finishing", name)
 9
10if __name__ == "__main__":
11    format = "%(asctime)s: %(message)s"
12    logging.basicConfig(format=format, level=logging.INFO,
13                        datefmt="%H:%M:%S")
14
15    logging.info("Main    : before creating thread")
16    x = threading.Thread(target=thread_function, args=(1,))
17    logging.info("Main    : before running thread")
18    x.start()
19    logging.info("Main    : wait for the thread to finish")
20    # x.join()
21    logging.info("Main    : all done")
```

If you look around the logging statements, you can see that the `main` section is creating and starting the thread:

```
x = threading.Thread(target=thread_function, args=(1,))
x.start()
```

When you create a `Thread`, you pass it a function and a list containing the arguments to that function. In this case, you're telling the `Thread` to run `thread_function()` and to pass it `1` as an argument.

For this article, you'll use sequential integers as names for your threads. There is `threading.get_ident()`, which returns a unique name for each thread, but these are usually neither short nor easily readable.

`thread_function()` itself doesn't do much. It simply logs some messages with a `time.sleep()` in between them.

When you run this program as it is (with line twenty commented out), the output will look like this:

```
$ ./single_thread.py
Main    : before creating thread
Main    : before running thread
Thread 1: starting
Main    : wait for the thread to finish
Main    : all done
Thread 1: finishing
```

You'll notice that the `Thread` finished after the `Main` section of your code did. You'll come back to why that is and talk about the mysterious line twenty in the next section.

Daemon Threads

In computer science, a daemon is a process that runs in the background.

Python `threading` has a more specific meaning for `daemon`. A `daemon` thread will shut down immediately when the program exits. One way to think about these definitions is to consider the `daemon` thread a thread that runs in the background without worrying about shutting it down.

If a program is running `Threads` that are not `daemons`, then the program will wait for those threads to complete before it terminates. `Threads` that *are* daemons, however, are just killed wherever they are when the program is exiting.

Let's look a little more closely at the output of your program above. The last two lines are the interesting bit. When you run the program, you'll notice that there is a pause (of about 2 seconds) after `__main__` has printed its `all done` message and before the thread is finished.

This pause is Python waiting for the non-daemonic thread to complete. When your Python program ends, part of the shutdown process is to clean up the threading routine.

If you look at the source for Python threading, you'll see that `threading._shutdown()` walks through all of the running threads and calls `.join()` on every one that does not have the `daemon` flag set.

So your program waits to exit because the thread itself is waiting in a sleep. As soon as it has completed and printed the message, `.join()` will return and the program can exit.

Frequently, this behavior is what you want, but there are other options available to us. Let's first repeat the program with a `daemon` thread. You do that by changing how you construct the `Thread`, adding the `daemon=True` flag:

```
x = threading.Thread(target=thread_function, args=(1,), daemon=True)
```

When you run the program now, you should see this output:

```
$ ./daemon_thread.py
Main    : before creating thread
Main    : before running thread
Thread 1: starting
Main    : wait for the thread to finish
Main    : all done
```

The difference here is that the final line of the output is missing. `thread_function()` did not get a chance to complete. It was a `daemon` thread, so when `__main__` reached the end of its code and the program wanted to finish, the daemon was killed.

`join()` a Thread

Daemon threads are handy, but what about when you want to wait for a thread to stop? What about when you want to do that and not exit your program? Now let's go back to your original program and look at that commented out line twenty:

```
# x.join()
```

To tell one thread to wait for another thread to finish, you call `.join()`. If you uncomment that line, the main thread will pause and wait for the thread `x` to complete running.

Did you test this on the code with the daemon thread or the regular thread? It turns out that it doesn't matter. If you `.join()` a thread, that statement will wait until either kind of thread is finished.

Working With Many Threads

The example code so far has only been working with two threads: the main thread and one you started with the `threading.Thread` object.

Frequently, you'll want to start a number of threads and have them do interesting work. Let's start by looking at the harder way of doing that, and then you'll move on to an easier method.

The harder way of starting multiple threads is the one you already know:

```
import logging
import threading
import time

def thread_function(name):
    logging.info("Thread %s: starting", name)
    time.sleep(2)
    logging.info("Thread %s: finishing", name)

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    threads = list()
    for index in range(3):
        logging.info("Main    : create and start thread %d.", index)
        x = threading.Thread(target=thread_function, args=(index,))
        threads.append(x)
        x.start()

    for index, thread in enumerate(threads):
        logging.info("Main    : before joining thread %d.", index)
        thread.join()
        logging.info("Main    : thread %d done", index)
```

This code uses the same mechanism you saw above to start a thread, create a `Thread` object, and then call `.start()`. The program keeps a list of `Thread` objects so that it can then wait for them later using `.join()`.

Running this code multiple times will likely produce some interesting results. Here's an example output from my machine:

```
$ ./multiple_threads.py
Main    : create and start thread 0.
Thread 0: starting
Main    : create and start thread 1.
Thread 1: starting
Main    : create and start thread 2.
Thread 2: starting
Main    : before joining thread 0.
Thread 2: finishing
Thread 1: finishing
Thread 0: finishing
Main    : thread 0 done
Main    : before joining thread 1.
Main    : thread 1 done
Main    : before joining thread 2.
Main    : thread 2 done
```

If you walk through the output carefully, you'll see all three threads getting started in the order you might expect, but in this case they finish in the opposite order! Multiple runs will produce different orderings. Look for the `Thread x: finishing` message to tell

you when each thread is done.

The order in which threads are run is determined by the operating system and can be quite hard to predict. It may (and likely will) vary from run to run, so you need to be aware of that when you design algorithms that use threading.

Fortunately, Python gives you several primitives that you'll look at later to help coordinate threads and get them running together. Before that, let's look at how to make managing a group of threads a bit easier.

Using a `ThreadPoolExecutor`

There's an easier way to start up a group of threads than the one you saw above. It's called a `ThreadPoolExecutor`, and it's part of the standard library in concurrent.futures (as of Python 3.2).

The easiest way to create it is as a context manager, using the `with` statement to manage the creation and destruction of the pool.

Here's the `__main__` from the last example rewritten to use a `ThreadPoolExecutor`:

```
import concurrent.futures

# [rest of code]

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
        executor.map(thread_function, range(3))
```

The code creates a `ThreadPoolExecutor` as a context manager, telling it how many worker threads it wants in the pool. It then uses `.map()` to step through an iterable of things, in your case `range(3)`, passing each one to a thread in the pool.

The end of the `with` block causes the `ThreadPoolExecutor` to do a `.join()` on each of the threads in the pool. It is *strongly* recommended that you use `ThreadPoolExecutor` as a context manager when you can so that you never forget to `.join()` the threads.

Note: Using a `ThreadPoolExecutor` can cause some confusing errors.

For example, if you call a function that takes no parameters, but you pass it parameters in `.map()`, the thread will throw an exception.

Unfortunately, `ThreadPoolExecutor` will hide that exception, and (in the case above) the program terminates with no output. This can be quite confusing to debug at first.

Running your corrected example code will produce output that looks like this:

```
$ ./executor.py
Thread 0: starting
Thread 1: starting
Thread 2: starting
Thread 1: finishing
Thread 0: finishing
Thread 2: finishing
```

Again, notice how `Thread 1` finished before `Thread 0`. The scheduling of threads is done by the operating system and does not follow a plan that's easy to figure out.

Race Conditions

Before you move on to some of the other features tucked away in Python `threading`, let's talk a bit about one of the more difficult issues you'll run into when writing threaded programs: race conditions.

Once you've seen what a race condition is and looked at one happening, you'll move on to some of the primitives provided by the standard library to prevent race conditions from happening.

Race conditions can occur when two or more threads access a shared piece of data or resource. In this example, you're going to create a large race condition that happens every time, but be aware that most race conditions are not this obvious. Frequently, they only occur rarely, and they can produce confusing results. As you can imagine, this makes them quite difficult to debug.

Fortunately, this race condition will happen every time, and you'll walk through it in detail to explain what is happening.

For this example, you're going to write a class that updates a database. Okay, you're not really going to have a database: you're just going to fake it, because that's not the point of this article.

Your `FakeDatabase` will have `.__init__()` and `.update()` methods:

```
class FakeDatabase:
    def __init__(self):
        self.value = 0

    def update(self, name):
        logging.info("Thread %s: starting update", name)
        local_copy = self.value
        local_copy += 1
        time.sleep(0.1)
        self.value = local_copy
        logging.info("Thread %s: finishing update", name)
```

`FakeDatabase` is keeping track of a single number: `.value`. This is going to be the shared data on which you'll see the race condition.

`.__init__()` simply initializes `.value` to zero. So far, so good.

`.update()` looks a little strange. It's simulating reading a value from a database, doing some computation on it, and then writing a new value back to the database.

In this case, reading from the database just means copying `.value` to a local variable. The computation is just to add one to the value and then `.sleep()` for a little bit. Finally, it writes the value back by copying the local value back to `.value`.

Here's how you'll use this `FakeDatabase`:

```
if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    database = FakeDatabase()
    logging.info("Testing update. Starting value is %d.", database.value)
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
        for index in range(2):
            executor.submit(database.update, index)
    logging.info("Testing update. Ending value is %d.", database.value)
```

The program creates a `ThreadPoolExecutor` with two threads and then calls `.submit()` on each of them, telling them to run `database.update()`.

`.submit()` has a signature that allows both positional and named arguments to be passed to the function running in the thread:

```
.submit(function, *args, **kwargs)
```

In the usage above, `index` is passed as the first and only positional argument to `database.update()`. You'll see later in this article where you can pass multiple arguments in a similar manner.

Since each thread runs `.update()`, and `.update()` adds one to `.value`, you might expect `database.value` to be `2` when it's printed out at the end. But you wouldn't be looking at this example if that was the case. If you run the above code, the output looks like this:

```
$ ./racecond.py
Testing unlocked update. Starting value is 0.
Thread 0: starting update
Thread 1: starting update
Thread 0: finishing update
Thread 1: finishing update
Testing unlocked update. Ending value is 1.
```

You might have expected that to happen, but let's look at the details of what's really going on here, as that will make the solution to this problem easier to understand.

One Thread

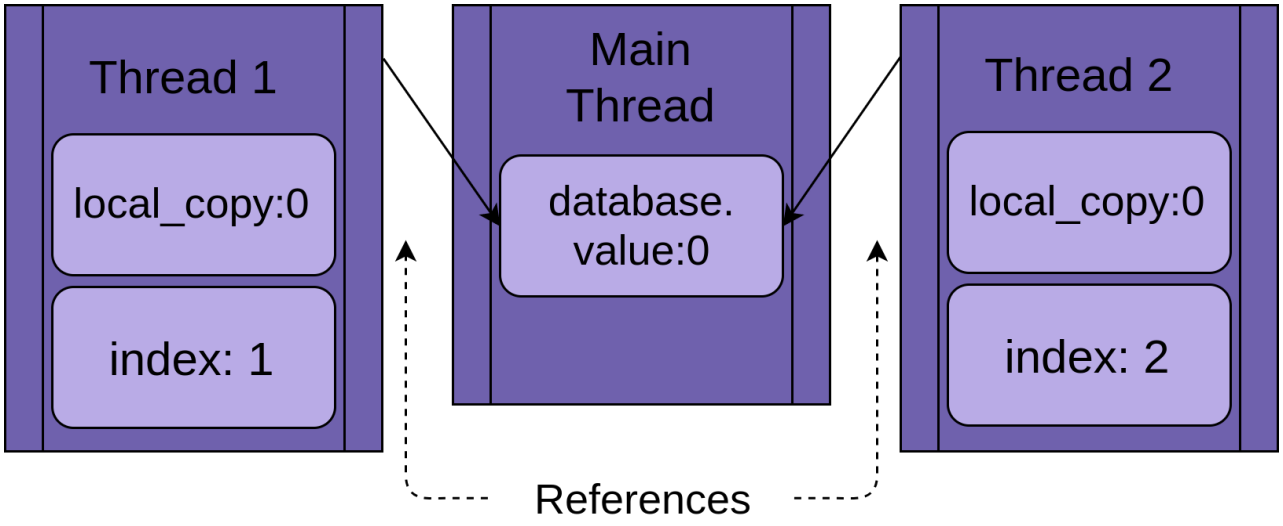
Before you dive into this issue with two threads, let's step back and talk a bit about some details of how threads work.

You won't be diving into all of the details here, as that's not important at this level. We'll also be simplifying a few things in a way that won't be technically accurate but will give you the right idea of what is happening.

When you tell your `ThreadPoolExecutor` to run each thread, you tell it which function to run and what parameters to pass to it: `executor.submit(database.update, index)`.

The result of this is that each of the threads in the pool will call `database.update(index)`. Note that `database` is a reference to the one `FakeDatabase` object created in `__main__`. Calling `.update()` on that object calls an instance method on that object.

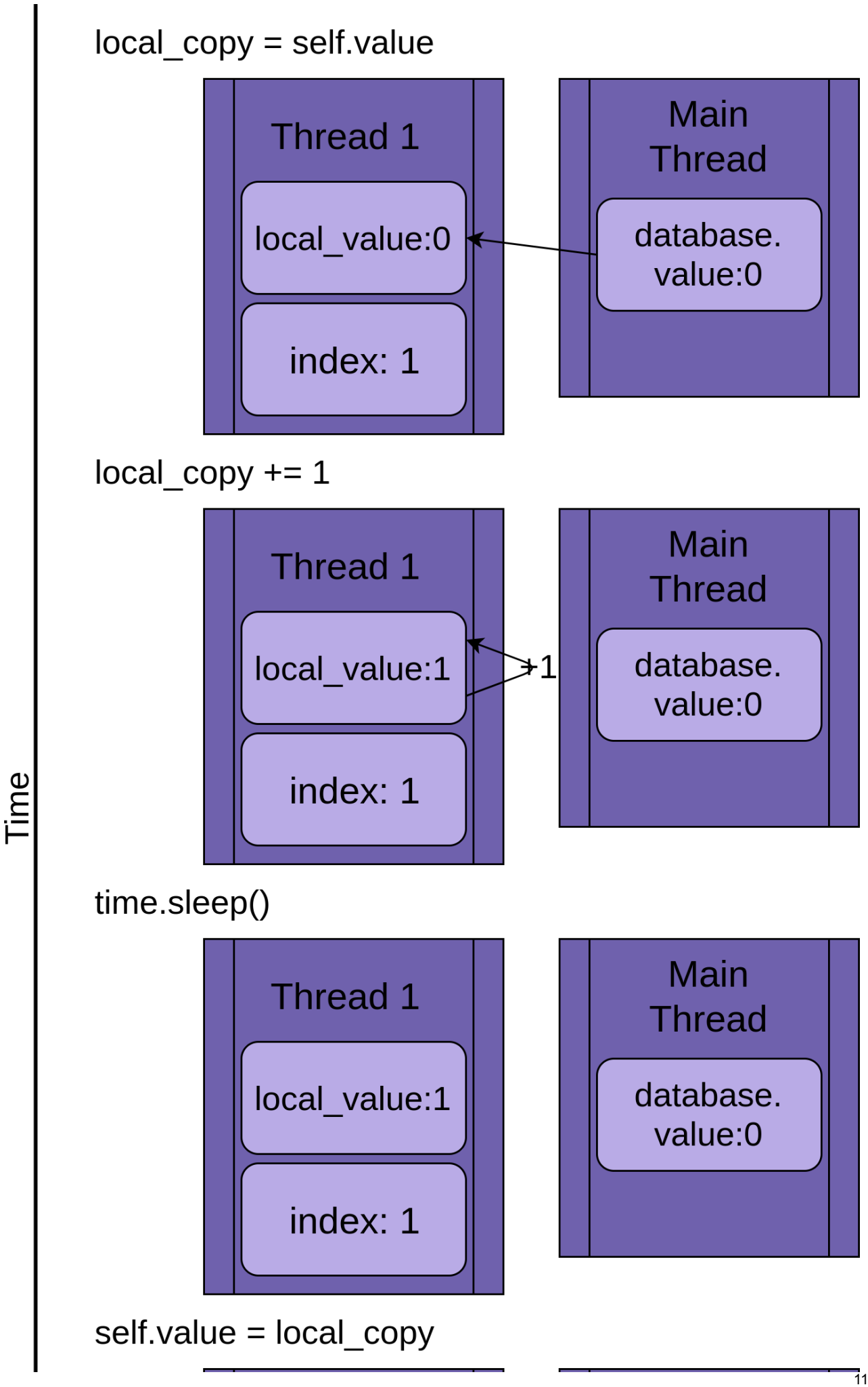
Each thread is going to have a reference to the same `FakeDatabase` object, `database`. Each thread will also have a unique value, `index`, to make the logging statements a bit easier to read:

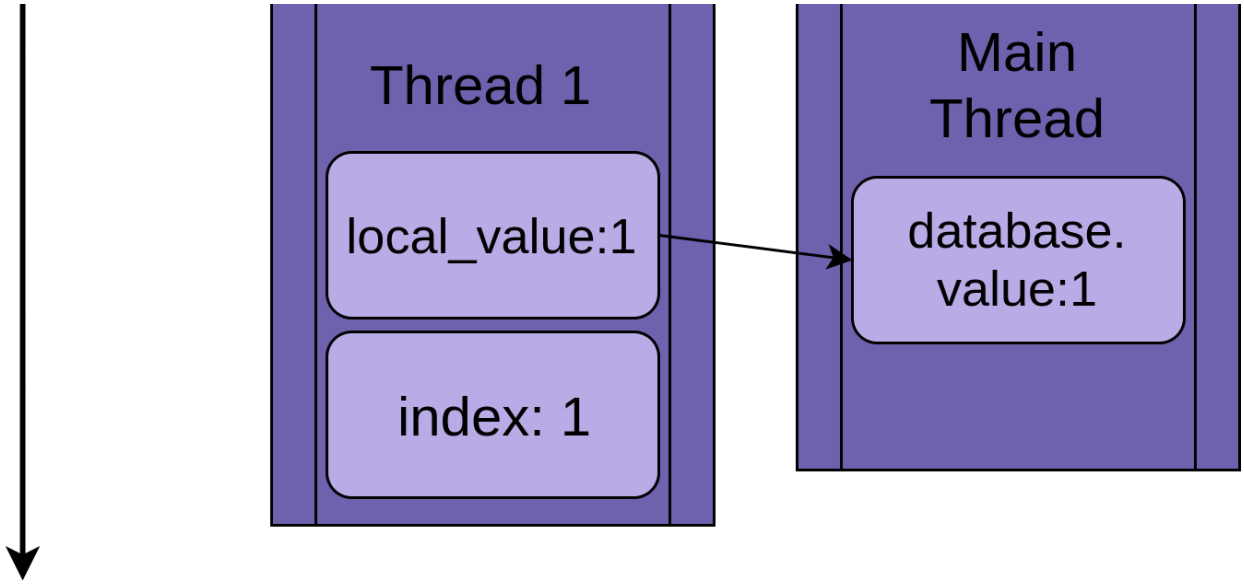


When the thread starts running `.update()`, it has its own version of all of the data **local** to the function. In the case of `.update()`, this is `local_copy`. This is definitely a good thing. Otherwise, two threads running the same function would always confuse each other. It means that all variables that are scoped (or local) to a function are **thread-safe**.

Now you can start walking through what happens if you run the program above with a single thread and a single call to `.update()`.

The image below steps through the execution of `.update()` if only a single thread is run. The statement is shown on the left followed by a diagram showing the values in the thread's `local_value` and the shared `database.value`:


Time
local_copy = self.value
Thread 1
local_value:0
index: 1
Main Thread
database. value:0
local_copy += 1
Thread 1
local_value:1
index: 1
+1
Main Thread
database. value:0
time.sleep()
Thread 1
local_value:1
index: 1
Main Thread
database. value:0
self.value = local_copy



The diagram is laid out so that time increases as you move from top to bottom. It begins when `Thread 1` is created and ends when it is terminated.

When `Thread 1` starts, `FakeDatabase.value` is zero. The first line of code in the method, `local_copy = self.value`, copies the value zero to the local variable. Next it increments the value of `local_copy` with the `local_copy += 1` statement. You can see `.value` in `Thread 1` getting set to one.

Next `time.sleep()` is called, which makes the current thread pause and allows other threads to run. Since there is only one thread in this example, this has no effect.

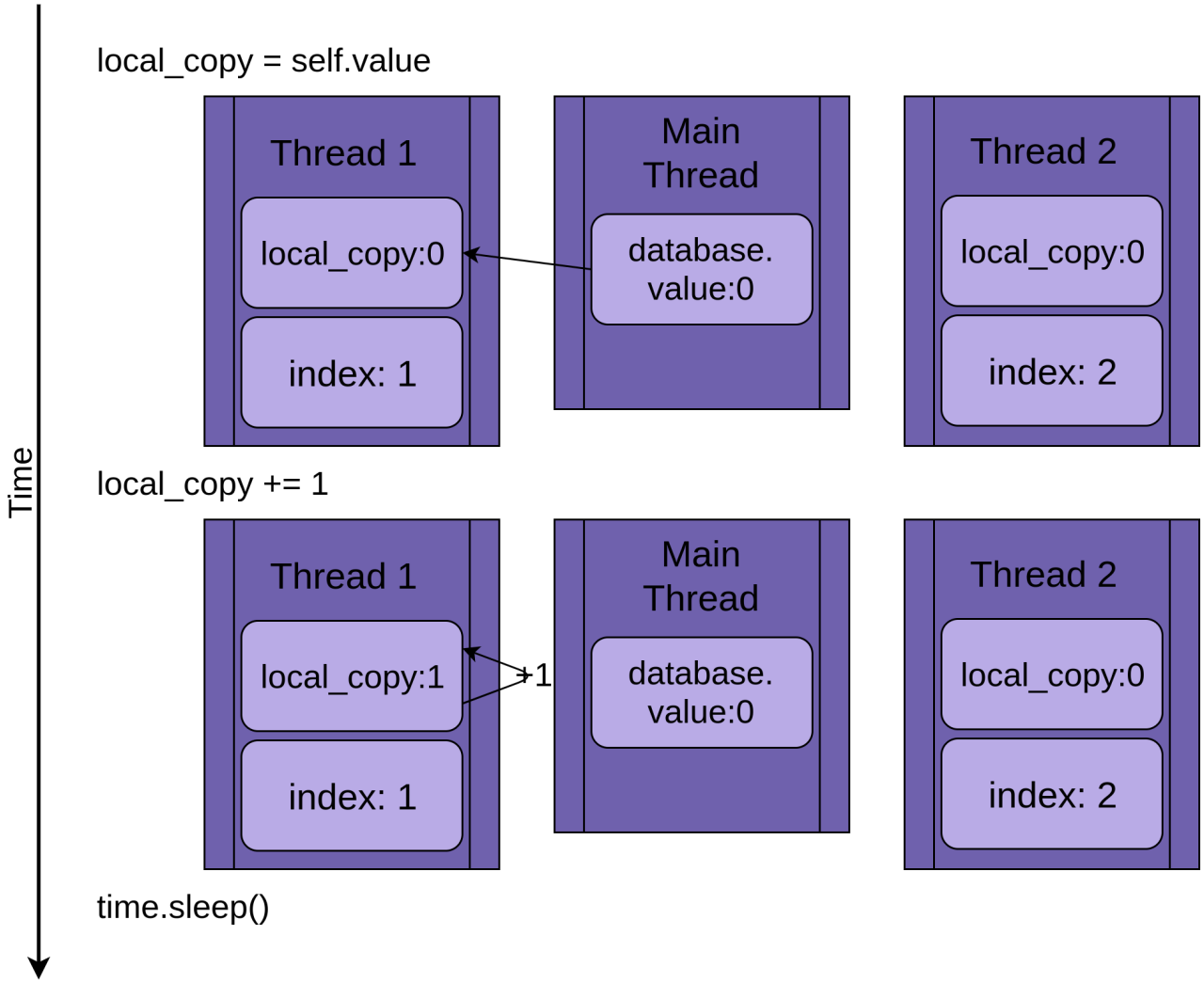
When `Thread 1` wakes up and continues, it copies the new value from `local_copy` to `FakeDatabase.value`, and then the thread is complete. You can see that `database.value` is set to one.

So far, so good. You ran `.update()` once and `FakeDatabase.value` was incremented to one.

Two Threads

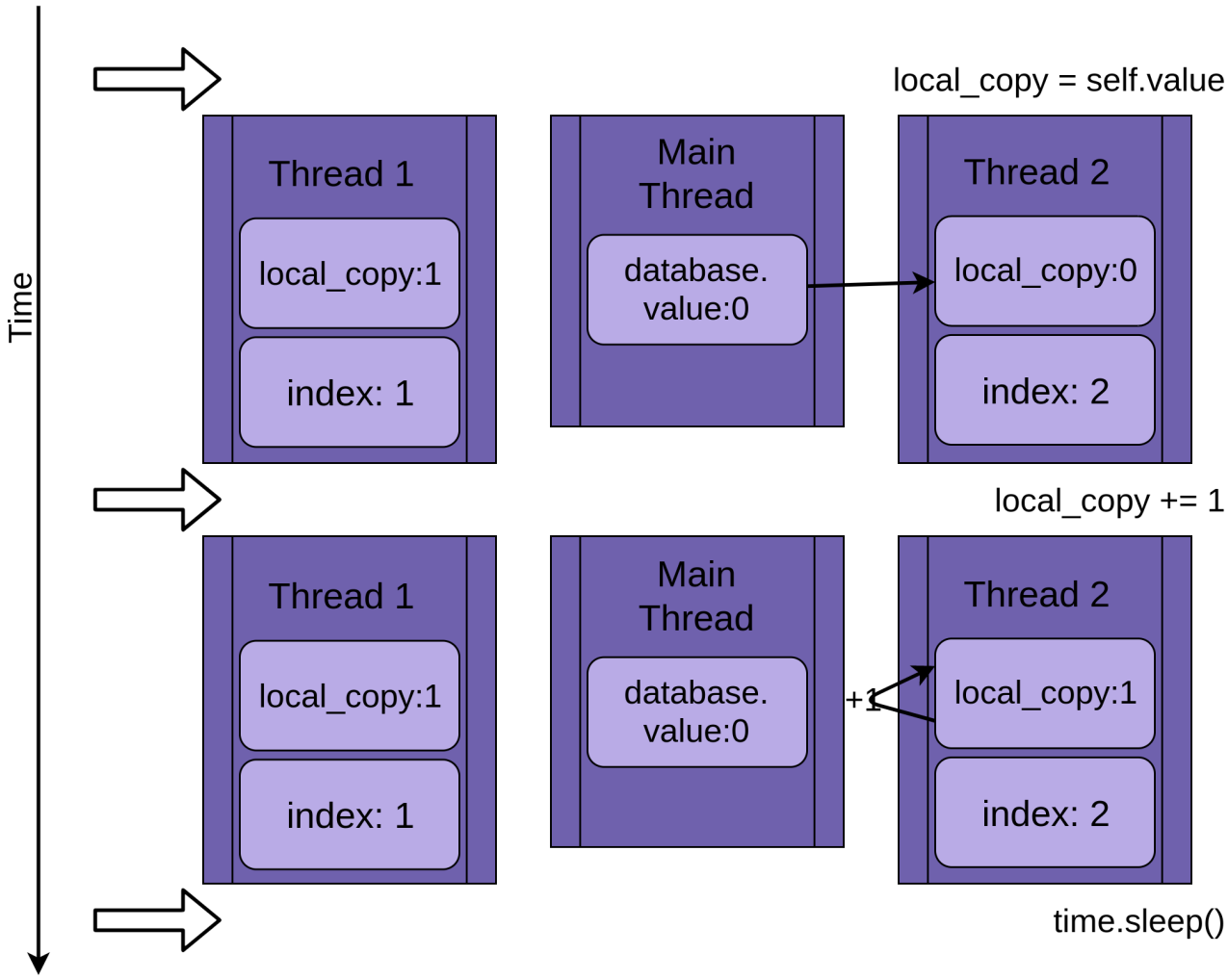
Getting back to the race condition, the two threads will be running concurrently but not at the same time. They will each have their own version of `local_copy` and will each point to the same `database`. It is this shared `database` object that is going to cause the problems.

The program starts with `Thread 1` running `.update()`:



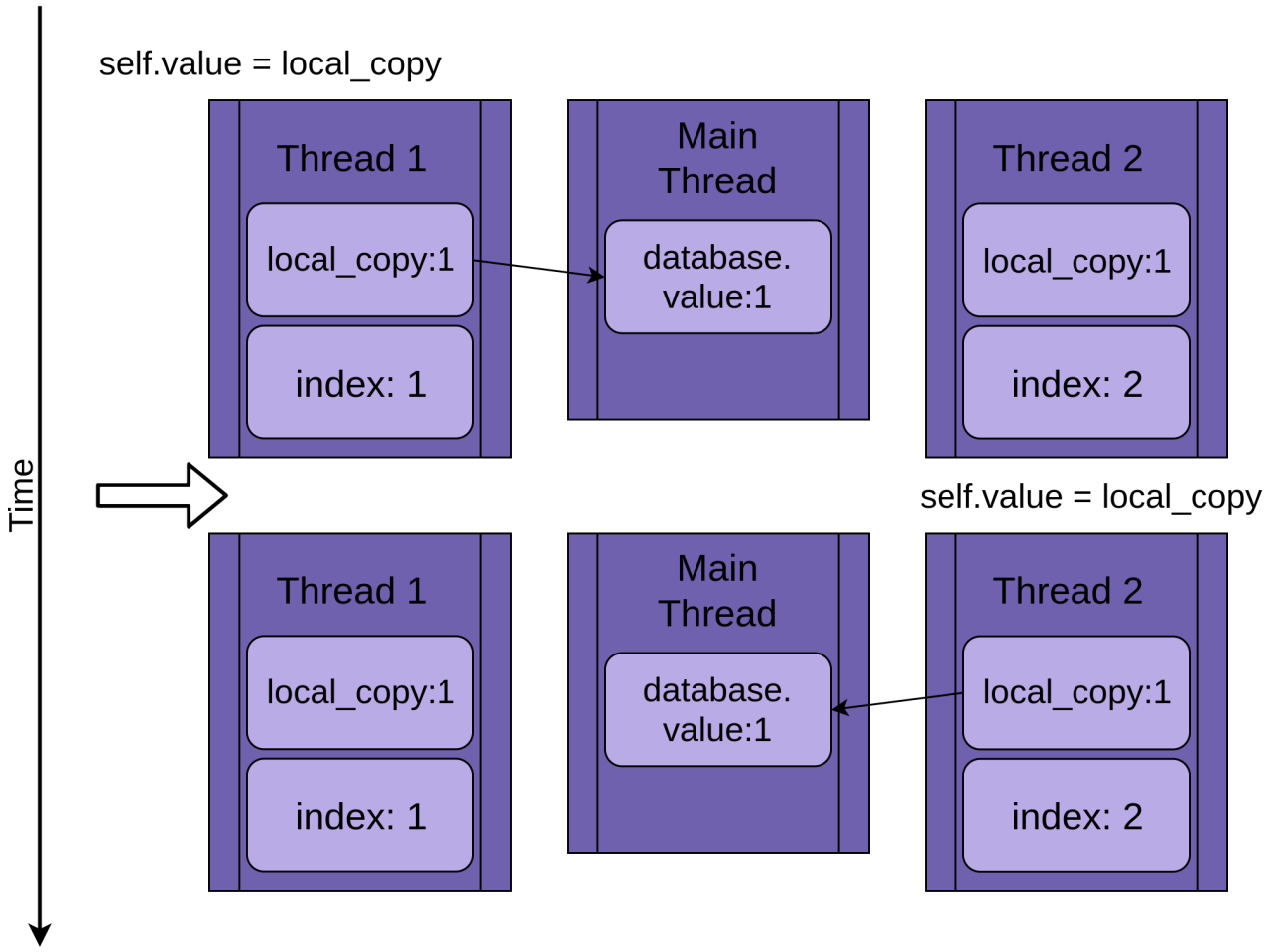
When `Thread 1` calls `time.sleep()`, it allows the other thread to start running. This is where things get interesting.

`Thread 2` starts up and does the same operations. It's also copying `database.value` into its private `local_copy`, and this shared `database.value` has not yet been updated:



When `Thread 2` finally goes to sleep, the shared `database.value` is still unmodified at zero, and both private versions of `local_copy` have the value one.

`Thread 1` now wakes up and saves its version of `local_copy` and then terminates, giving `Thread 2` a final chance to run. `Thread 2` has no idea that `Thread 1` ran and updated `database.value` while it was sleeping. It stores *its* version of `local_copy` into `database.value`, also setting it to one:



The two threads have interleaving access to a single shared object, overwriting each other's results. Similar race conditions can arise when one thread frees memory or closes a file handle before the other thread is finished accessing it.

Why This Isn't a Silly Example

The example above is contrived to make sure that the race condition happens every time you run your program. Because the operating system can swap out a thread at any time, it is possible to interrupt a statement like `x = x + 1` after it has read the value of `x` but before it has written back the incremented value.

The details of how this happens are quite interesting, but not needed for the rest of this article, so feel free to skip over this hidden section.

The code above isn't quite as out there as you might originally have thought. It was designed to force a race condition every time you run it, but that makes it much easier to solve than most race conditions.

There are two things to keep in mind when thinking about race conditions:

1. Even an operation like `x += 1` takes the processor many steps. Each of these steps is a separate instruction to the processor.

2. The operating system can swap which thread is running *at any time*. A thread can be swapped out after any of these small instructions. This means that a thread can be put to sleep to let another thread run in the *middle* of a Python statement.

Let's look at this in detail. The REPL below shows a function that takes a parameter and increments it:

```
>>>

>>> def inc(x):
...     x += 1
...
>>> import dis
>>> dis.dis(inc)
  2           0 LOAD_FAST                0 (x)
              2 LOAD_CONST               1 (1)
              4 INPLACE_ADD
              6 STORE_FAST               0 (x)
              8 LOAD_CONST               0 (None)
             10 RETURN_VALUE
```

The REPL example uses dis from the Python standard library to show the smaller steps that the processor does to implement your function. It does a `LOAD_FAST` of the data value `x`, it does a `LOAD_CONST 1`, and then it uses the `INPLACE_ADD` to add those values together.

We're stopping here for a specific reason. This is the point in `.update()` above where `time.sleep()` forced the threads to switch. It is entirely possible that, every once in while, the operating system would switch threads at that exact point even without `sleep()`, but the call to `sleep()` makes it happen every time.

As you learned above, the operating system can swap threads at any time. You've walked down this listing to the statement marked `4`. If the operating system swaps out this thread and runs a different thread that also modifies `x`, then when this thread resumes, it will overwrite `x` with an incorrect value.

Technically, this example won't have a race condition because `x` is local to `inc()`. It does illustrate how a thread can be interrupted during a single Python operation, however. The same LOAD, MODIFY, STORE set of operations also happens on global and shared values. You can explore with the `dis` module and prove that yourself.

It's rare to get a race condition like this to occur, but remember that an infrequent event taken over millions of iterations becomes likely to happen. The rarity of these race conditions makes them much, much harder to debug than regular bugs.

Now back to your regularly scheduled tutorial!

Now that you've seen a race condition in action, let's find out how to solve them!

Basic Synchronization Using `Lock`

There are a number of ways to avoid or solve race conditions. You won't look at all of them here, but there are a couple that are used frequently. Let's start with `Lock`.

To solve your race condition above, you need to find a way to allow only one thread at a time into the read-modify-write section of your code. The most common way to do this is called `Lock` in Python. In some other languages this same idea is called a `mutex`. Mutex comes from MUTual EXclusion, which is exactly what a `Lock` does.

A `Lock` is an object that acts like a hall pass. Only one thread at a time can have the `Lock`. Any other thread that wants the `Lock` must wait until the owner of the `Lock` gives it up.

The basic functions to do this are `.acquire()` and `.release()`. A thread will call `my_lock.acquire()` to get the lock. If the lock is already held, the calling thread will wait until it is released. There's an important point here. If one thread gets the lock but never gives it back, your program will be stuck. You'll read more about this later.

Fortunately, Python's `Lock` will also operate as a context manager, so you can use it in a `with` statement, and it gets released automatically when the `with` block exits for any reason.

Let's look at the `FakeDatabase` with a `Lock` added to it. The calling function stays the same:

```
class FakeDatabase:
    def __init__(self):
        self.value = 0
        self._lock = threading.Lock()

    def locked_update(self, name):
        logging.info("Thread %s: starting update", name)
        logging.debug("Thread %s about to lock", name)
        with self._lock:
            logging.debug("Thread %s has lock", name)
            local_copy = self.value
            local_copy += 1
            time.sleep(0.1)
            self.value = local_copy
            logging.debug("Thread %s about to release lock", name)
        logging.debug("Thread %s after release", name)
        logging.info("Thread %s: finishing update", name)
```

Other than adding a bunch of debug logging so you can see the locking more clearly, the big change here is to add a member called `._lock`, which is a `threading.Lock()` object. This `._lock` is initialized in the unlocked state and locked and released by the `with` statement.

It's worth noting here that the thread running this function will hold on to that `Lock`

until it is completely finished updating the database. In this case, that means it will hold the `Lock` while it copies, updates, sleeps, and then writes the value back to the database.

If you run this version with logging set to warning level, you'll see this:

```
$ ./fixrace.py
Testing locked update. Starting value is 0.
Thread 0: starting update
Thread 1: starting update
Thread 0: finishing update
Thread 1: finishing update
Testing locked update. Ending value is 2.
```

Look at that. Your program finally works!

You can turn on full logging by setting the level to `DEBUG` by adding this statement after you configure the logging output in `__main__` :

```
logging.getLogger().setLevel(logging.DEBUG)
```

Running this program with `DEBUG` logging turned on looks like this:

```
$ ./fixrace.py
Testing locked update. Starting value is 0.
Thread 0: starting update
Thread 0 about to lock
Thread 0 has lock
Thread 1: starting update
Thread 1 about to lock
Thread 0 about to release lock
Thread 0 after release
Thread 0: finishing update
Thread 1 has lock
Thread 1 about to release lock
Thread 1 after release
Thread 1: finishing update
Testing locked update. Ending value is 2.
```

In this output you can see `Thread 0` acquires the lock and is still holding it when it goes to sleep. `Thread 1` then starts and attempts to acquire the same lock. Because `Thread 0` is still holding it, `Thread 1` has to wait. This is the mutual exclusion that a `Lock` provides.

Many of the examples in the rest of this article will have `WARNING` and `DEBUG` level logging. We'll generally only show the `WARNING` level output, as the `DEBUG` logs can be quite lengthy. Try out the programs with the logging turned up and see what they do.

Deadlock

Before you move on, you should look at a common problem when using `Locks` . As you

saw, if the `Lock` has already been acquired, a second call to `.acquire()` will wait until the thread that is holding the `Lock` calls `.release()`. What do you think happens when you run this code:

```
import threading

l = threading.Lock()
print("before first acquire")
l.acquire()
print("before second acquire")
l.acquire()
print("acquired lock twice")
```

When the program calls `l.acquire()` the second time, it hangs waiting for the `Lock` to be released. In this example, you can fix the deadlock by removing the second call, but deadlocks usually happen from one of two subtle things:

1. An implementation bug where a `Lock` is not released properly
2. A design issue where a utility function needs to be called by functions that might or might not already have the `Lock`

The first situation happens sometimes, but using a `Lock` as a context manager greatly reduces how often. It is recommended to write code whenever possible to make use of context managers, as they help to avoid situations where an exception skips you over the `.release()` call.

The design issue can be a bit trickier in some languages. Thankfully, Python threading has a second object, called `RLock`, that is designed for just this situation. It allows a thread to `.acquire()` an `RLock` multiple times before it calls `.release()`. That thread is still required to call `.release()` the same number of times it called `.acquire()`, but it should be doing that anyway.

`Lock` and `RLock` are two of the basic tools used in threaded programming to prevent race conditions. There are a few other that work in different ways. Before you look at them, let's shift to a slightly different problem domain.

Producer-Consumer Threading

The Producer-Consumer Problem is a standard computer science problem used to look at threading or process synchronization issues. You're going to look at a variant of it to get some ideas of what primitives the Python `threading` module provides.

For this example, you're going to imagine a program that needs to read messages from a network and write them to disk. The program does not request a message when it wants. It must be listening and accept messages as they come in. The messages will not come in at a regular pace, but will be coming in bursts. This part of the program is called the producer.

On the other side, once you have a message, you need to write it to a database. The database access is slow, but fast enough to keep up to the average pace of messages. It is *not* fast enough to keep up when a burst of messages comes in. This part is the consumer.

In between the producer and the consumer, you will create a `Pipeline` that will be the part that changes as you learn about different synchronization objects.

That's the basic layout. Let's look at a solution using `Lock`. It doesn't work perfectly, but it uses tools you already know, so it's a good place to start.

Producer-Consumer Using `Lock`

Since this is an article about Python `threading`, and since you just read about the `Lock` primitive, let's try to solve this problem with two threads using a `Lock` or two.

The general design is that there is a `producer` thread that reads from the fake network and puts the message into a `Pipeline`:

```
import random

SENTINEL = object()

def producer(pipeline):
    """Pretend we're getting a message from the network."""
    for index in range(10):
        message = random.randint(1, 101)
        logging.info("Producer got message: %s", message)
        pipeline.set_message(message, "Producer")

    # Send a sentinel message to tell consumer we're done
    pipeline.set_message(SENTINEL, "Producer")
```

To generate a fake message, the `producer` gets a random number between one and one hundred. It calls `.set_message()` on the `pipeline` to send it to the `consumer`.

The `producer` also uses a `SENTINEL` value to signal the consumer to stop after it has sent ten values. This is a little awkward, but don't worry, you'll see ways to get rid of this `SENTINEL` value after you work through this example.

On the other side of the `pipeline` is the consumer:

```
def consumer(pipeline):
    """Pretend we're saving a number in the database."""
    message = 0
    while message is not SENTINEL:
        message = pipeline.get_message("Consumer")
        if message is not SENTINEL:
            logging.info("Consumer storing message: %s", message)
```

The `consumer` reads a message from the `pipeline` and writes it to a fake database, which in this case is just printing it to the display. If it gets the `SENTINEL` value, it returns from the function, which will terminate the thread.

Before you look at the really interesting part, the `Pipeline`, here's the `__main__` section, which spawns these threads:

```
if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")
    # logging.getLogger().setLevel(logging.DEBUG)

    pipeline = Pipeline()
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
        executor.submit(producer, pipeline)
        executor.submit(consumer, pipeline)
```

This should look fairly familiar as it's close to the `__main__` code in the previous examples.

Remember that you can turn on `DEBUG` logging to see all of the logging messages by uncommenting this line:

```
# logging.getLogger().setLevel(logging.DEBUG)
```

It can be worthwhile to walk through the `DEBUG` logging messages to see exactly where each thread acquires and releases the locks.

Now let's take a look at the `Pipeline` that passes messages from the `producer` to the `consumer`:

```
class Pipeline:
    """
    Class to allow a single element pipeline between producer and consumer.
    """
    def __init__(self):
        self.message = 0
        self.producer_lock = threading.Lock()
        self.consumer_lock = threading.Lock()
        self.consumer_lock.acquire()

    def get_message(self, name):
        logging.debug("%s:about to acquire getlock", name)
        self.consumer_lock.acquire()
        logging.debug("%s:have getlock", name)
        message = self.message.copy()
        logging.debug("%s:about to release setlock", name)
        self.producer_lock.release()
        logging.debug("%s:setlock released", name)
        return message

    def set_message(self, message, name):
        logging.debug("%s:about to acquire setlock", name)
        self.producer_lock.acquire()
        logging.debug("%s:have setlock", name)
        self.message = message.copy()
        logging.debug("%s:about to release getlock", name)
        self.consumer_lock.release()
        logging.debug("%s:getlock released", name)
```

Woah! That's a lot of code. A pretty high percentage of that is just logging statements to make it easier to see what's happening when you run it. Here's the same code with all of the logging statements removed:

```
class Pipeline:
    """
    Class to allow a single element pipeline between producer and consumer.
    """
    def __init__(self):
        self.message = 0
        self.producer_lock = threading.Lock()
        self.consumer_lock = threading.Lock()
        self.consumer_lock.acquire()

    def get_message(self, name):
        self.consumer_lock.acquire()
        message = self.message.copy()
        self.producer_lock.release()
        return message

    def set_message(self, message, name):
        self.producer_lock.acquire()
        self.message = message.copy()
        self.consumer_lock.release()
```

That seems a bit more manageable. The `Pipeline` in this version of your code has three members:

1. **`.message`** stores the message to pass.
2. **`.producer_lock`** is a `threading.Lock` object that restricts access to the message by the `producer` thread.
3. **`.consumer_lock`** is also a `threading.Lock` that restricts access to the message by the `consumer` thread.

`__init__()` initializes these three members and then calls `.acquire()` on the `.consumer_lock`. This is the state you want to start in. The `producer` is allowed to add a new message, but the `consumer` needs to wait until a message is present.

`.get_message()` and `.set_messages()` are nearly opposites. `.get_message()` calls `.acquire()` on the `consumer_lock`. This is the call that will make the `consumer` wait until a message is ready.

Once the `consumer` has acquired the `.consumer_lock`, it copies out the value in `.message` and then calls `.release()` on the `.producer_lock`. Releasing this lock is what allows the `producer` to insert the next message into the `pipeline`.

Before you go on to `.set_message()`, there's something subtle going on in `.get_message()` that's pretty easy to miss. It might seem tempting to get rid of `message` and just have the function end with `return self.message`. See if you can figure out why you don't want to do that before moving on.

Here's the answer. As soon as the `consumer` calls `.producer_lock.release()`, it can be swapped out, and the `producer` can start running. That could happen before `.release()` returns! This means that there is a slight possibility that when the function returns `self.message`, that could actually be the *next* message generated, so you would lose the first message. This is another example of a race condition.

Moving on to `.set_message()`, you can see the opposite side of the transaction. The `producer` will call this with a message. It will acquire the `.producer_lock`, set the `.message`, and the call `.release()` on then `consumer_lock`, which will allow the `consumer` to read that value.

Let's run the code that has logging set to `WARNING` and see what it looks like:

```
$ ./prodcom_lock.py
Producer got data 43
Producer got data 45
Consumer storing data: 43
Producer got data 86
Consumer storing data: 45
Producer got data 40
Consumer storing data: 86
Producer got data 62
Consumer storing data: 40
Producer got data 15
Consumer storing data: 62
Producer got data 16
Consumer storing data: 15
Producer got data 61
Consumer storing data: 16
Producer got data 73
Consumer storing data: 61
Producer got data 22
Consumer storing data: 73
Consumer storing data: 22
```

At first, you might find it odd that the producer gets two messages before the consumer even runs. If you look back at the `producer` and `.set_message()`, you will notice that the only place it will wait for a `Lock` is when it attempts to put the message into the pipeline. This is done after the `producer` gets the message and logs that it has it.

When the `producer` attempts to send this second message, it will call `.set_message()` the second time and it will block.

The operating system can swap threads at any time, but it generally lets each thread have a reasonable amount of time to run before swapping it out. That's why the `producer` usually runs until it blocks in the second call to `.set_message()`.

Once a thread is blocked, however, the operating system will always swap it out and find a different thread to run. In this case, the only other thread with anything to do is the `consumer`.

The `consumer` calls `.get_message()`, which reads the message and calls `.release()` on the `.producer_lock`, thus allowing the `producer` to run again the next time threads are swapped.

Notice that the first message was `43`, and that is exactly what the `consumer` read, even though the `producer` had already generated the `45` message.

While it works for this limited test, it is not a great solution to the producer-consumer problem in general because it only allows a single value in the pipeline at a time. When the `producer` gets a burst of messages, it will have nowhere to put them.

Let's move on to a better way to solve this problem, using a `Queue`.

Producer-Consumer Using `Queue`

If you want to be able to handle more than one value in the pipeline at a time, you'll need a data structure for the pipeline that allows the number to grow and shrink as data backs up from the `producer`.

Python's standard library has a `queue` module which, in turn, has a `Queue` class. Let's change the `Pipeline` to use a `Queue` instead of just a variable protected by a `Lock`. You'll also use a different way to stop the worker threads by using a different primitive from Python `threading`, an `Event`.

Let's start with the `Event`. The `threading.Event` object allows one thread to signal an `event` while many other threads can be waiting for that `event` to happen. The key usage in this code is that the threads that are waiting for the event do not necessarily need to stop what they are doing, they can just check the status of the `Event` every once in a while.

The triggering of the event can be many things. In this example, the main thread will simply sleep for a while and then `.set()` it:

```
1if __name__ == "__main__":
2    format = "%(asctime)s: %(message)s"
3    logging.basicConfig(format=format, level=logging.INFO,
4                        datefmt="%H:%M:%S")
5    # logging.getLogger().setLevel(logging.DEBUG)
6
7    pipeline = Pipeline()
8    event = threading.Event()
9    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
10       executor.submit(producer, pipeline, event)
11       executor.submit(consumer, pipeline, event)
12
13       time.sleep(0.1)
14       logging.info("Main: about to set event")
15       event.set()
```

The only changes here are the creation of the `event` object on line 6, passing the `event` as a parameter on lines 8 and 9, and the final section on lines 11 to 13, which sleep for a second, log a message, and then call `.set()` on the event.

The `producer` also did not have to change too much:

```
1def producer(pipeline, event):
2    """Pretend we're getting a number from the network."""
3    while not event.is_set():
4        message = random.randint(1, 101)
5        logging.info("Producer got message: %s", message)
6        pipeline.set_message(message, "Producer")
7
8    logging.info("Producer received EXIT event. Exiting")
```

It now will loop until it sees that the event was set on line 3. It also no longer puts the `SENTINEL` value into the `pipeline`.

`consumer` had to change a little more:

```
1def consumer(pipeline, event):
2    """Pretend we're saving a number in the database."""
3    while not event.is_set() or not pipeline.empty():
4        message = pipeline.get_message("Consumer")
5        logging.info(
6            "Consumer storing message: %s  (queue size=%s)",
7            message,
8            pipeline.qsize(),
9        )
10
11    logging.info("Consumer received EXIT event. Exiting")
```

While you got to take out the code related to the `SENTINEL` value, you did have to do a slightly more complicated `while` condition. Not only does it loop until the `event` is set, but it also needs to keep looping until the `pipeline` has been emptied.

Making sure the queue is empty before the consumer finishes prevents another fun issue. If the `consumer` does exit while the `pipeline` has messages in it, there are two bad things that can happen. The first is that you lose those final messages, but the more serious one is that the `producer` can get caught attempting to add a message to a full queue and never return.

This happens if the `event` gets triggered after the `producer` has checked the `.is_set()` condition but before it calls `pipeline.set_message()`.

If that happens, it's possible for the producer to wake up and exit with the queue still completely full. The `producer` will then call `.set_message()` which will wait until there is space on the queue for the new message. The `consumer` has already exited, so this will not happen and the `producer` will not exit.

The rest of the `consumer` should look familiar.

The `Pipeline` has changed dramatically, however:

```
class Pipeline(queue.Queue):
    def __init__(self):
        super().__init__(maxsize=10)

    def get_message(self, name):
        logging.debug("%s:about to get from queue", name)
        value = self.get()
        logging.debug("%s:got %d from queue", name, value)
        return value

    def set_message(self, value, name):
        logging.debug("%s:about to add %d to queue", name, value)
        self.put(value)
        logging.debug("%s:added %d to queue", name, value)
```

You can see that `Pipeline` is a subclass of `queue.Queue`. `Queue` has an optional parameter when initializing to specify a maximum size of the queue.

If you give a positive number for `maxsize`, it will limit the queue to that number of elements, causing `.put()` to block until there are fewer than `maxsize` elements. If you don't specify `maxsize`, then the queue will grow to the limits of your computer's memory.

`.get_message()` and `.set_message()` got much smaller. They basically wrap `.get()` and `.put()` on the `Queue`. You might be wondering where all of the locking code that prevents the threads from causing race conditions went.

The core devs who wrote the standard library knew that a `Queue` is frequently used in multi-threading environments and incorporated all of that locking code inside the `Queue` itself. `Queue` is thread-safe.

Running this program looks like the following:

```
$ ./prodcom_queue.py
Producer got message: 32
Producer got message: 51
Producer got message: 25
Producer got message: 94
Producer got message: 29
Consumer storing message: 32 (queue size=3)
Producer got message: 96
Consumer storing message: 51 (queue size=3)
Producer got message: 6
Consumer storing message: 25 (queue size=3)
Producer got message: 31

[many lines deleted]

Producer got message: 80
Consumer storing message: 94 (queue size=6)
Producer got message: 33
Consumer storing message: 20 (queue size=6)
Producer got message: 48
Consumer storing message: 31 (queue size=6)
Producer got message: 52
Consumer storing message: 98 (queue size=6)
Main: about to set event
Producer got message: 13
Consumer storing message: 59 (queue size=6)
Producer received EXIT event. Exiting
Consumer storing message: 75 (queue size=6)
Consumer storing message: 97 (queue size=5)
Consumer storing message: 80 (queue size=4)
Consumer storing message: 33 (queue size=3)
Consumer storing message: 48 (queue size=2)
Consumer storing message: 52 (queue size=1)
Consumer storing message: 13 (queue size=0)
Consumer received EXIT event. Exiting
```

If you read through the output in my example, you can see some interesting things happening. Right at the top, you can see the `producer` got to create five messages and place four of them on the queue. It got swapped out by the operating system before it could place the fifth one.

The `consumer` then ran and pulled off the first message. It printed out that message as well as how deep the queue was at that point:

```
Consumer storing message: 32 (queue size=3)
```

This is how you know that the fifth message hasn't made it into the `pipeline` yet. The queue is down to size three after a single message was removed. You also know that the `queue` can hold ten messages, so the `producer` thread didn't get blocked by the `queue`. It was swapped out by the OS.

Note: Your output will be different. Your output will change from run to run. That's the fun part of working with threads!

As the program starts to wrap up, can you see the main thread generating the `event` which causes the `producer` to exit immediately. The `consumer` still has a bunch of work do to, so it keeps running until it has cleaned out the `pipeline`.

Try playing with different queue sizes and calls to `time.sleep()` in the `producer` or the `consumer` to simulate longer network or disk access times respectively. Even slight changes to these elements of the program will make large differences in your results.

This is a much better solution to the producer-consumer problem, but you can simplify it even more. The `Pipeline` really isn't needed for this problem. Once you take away the logging, it just becomes a `queue.Queue`.

Here's what the final code looks like using `queue.Queue` directly:

```
import concurrent.futures
import logging
import queue
import random
import threading
import time

def producer(queue, event):
    """Pretend we're getting a number from the network."""
    while not event.is_set():
        message = random.randint(1, 101)
        logging.info("Producer got message: %s", message)
        queue.put(message)

    logging.info("Producer received event. Exiting")

def consumer(queue, event):
    """Pretend we're saving a number in the database."""
    while not event.is_set() or not queue.empty():
        message = queue.get()
        logging.info(
            "Consumer storing message: %s (size=%d)", message, queue.qsize()
        )

    logging.info("Consumer received event. Exiting")

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    pipeline = queue.Queue(maxsize=10)
    event = threading.Event()
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
        executor.submit(producer, pipeline, event)
        executor.submit(consumer, pipeline, event)

        time.sleep(0.1)
        logging.info("Main: about to set event")
        event.set()
```

That's easier to read and shows how using Python's built-in primitives can simplify a complex problem.

`Lock` and `Queue` are handy classes to solve concurrency issues, but there are others provided by the standard library. Before you wrap up this tutorial, let's do a quick survey of some of them.

Threading Objects

There are a few more primitives offered by the Python `threading` module. While you didn't need these for the examples above, they can come in handy in different use cases, so it's good to be familiar with them.

Semaphore

The first Python `threading` object to look at is `threading.Semaphore`. A `Semaphore` is a counter with a few special properties. The first one is that the counting is atomic. This means that there is a guarantee that the operating system will not swap out the thread in the middle of incrementing or decrementing the counter.

The internal counter is incremented when you call `.release()` and decremented when you call `.acquire()`.

The next special property is that if a thread calls `.acquire()` when the counter is zero, that thread will block until a different thread calls `.release()` and increments the counter to one.

Semaphores are frequently used to protect a resource that has a limited capacity. An example would be if you have a pool of connections and want to limit the size of that pool to a specific number.

Timer

A `threading.Timer` is a way to schedule a function to be called after a certain amount of time has passed. You create a `Timer` by passing in a number of seconds to wait and a function to call:

```
t = threading.Timer(30.0, my_function)
```

You start the `Timer` by calling `.start()`. The function will be called on a new thread at some point after the specified time, but be aware that there is no promise that it will be called exactly at the time you want.

If you want to stop a `Timer` that you've already started, you can cancel it by calling `.cancel()`. Calling `.cancel()` after the `Timer` has triggered does nothing and does not produce an exception.

A `Timer` can be used to prompt a user for action after a specific amount of time. If the user does the action before the `Timer` expires, `.cancel()` can be called.

Barrier

A `threading.Barrier` can be used to keep a fixed number of threads in sync. When creating a `Barrier`, the caller must specify how many threads will be synchronizing on it. Each thread calls `.wait()` on the `Barrier`. They all will remain blocked until the specified number of threads are waiting, and then the are all released at the same time.

Remember that threads are scheduled by the operating system so, even though all of the threads are released simultaneously, they will be scheduled to run one at a time.

One use for a `Barrier` is to allow a pool of threads to initialize themselves. Having the threads wait on a `Barrier` after they are initialized will ensure that none of the threads start running before all of the threads are finished with their initialization.

Conclusion: Threading in Python

You've now seen much of what Python `threading` has to offer and some examples of how to build threaded programs and the problems they solve. You've also seen a few instances of the problems that arise when writing and debugging threaded programs.

If you'd like to explore other options for concurrency in Python, check out Speed Up Your Python Program With Concurrency.

If you're interested in doing a deep dive on the `asyncio` module, go read Async IO in Python: A Complete Walkthrough.

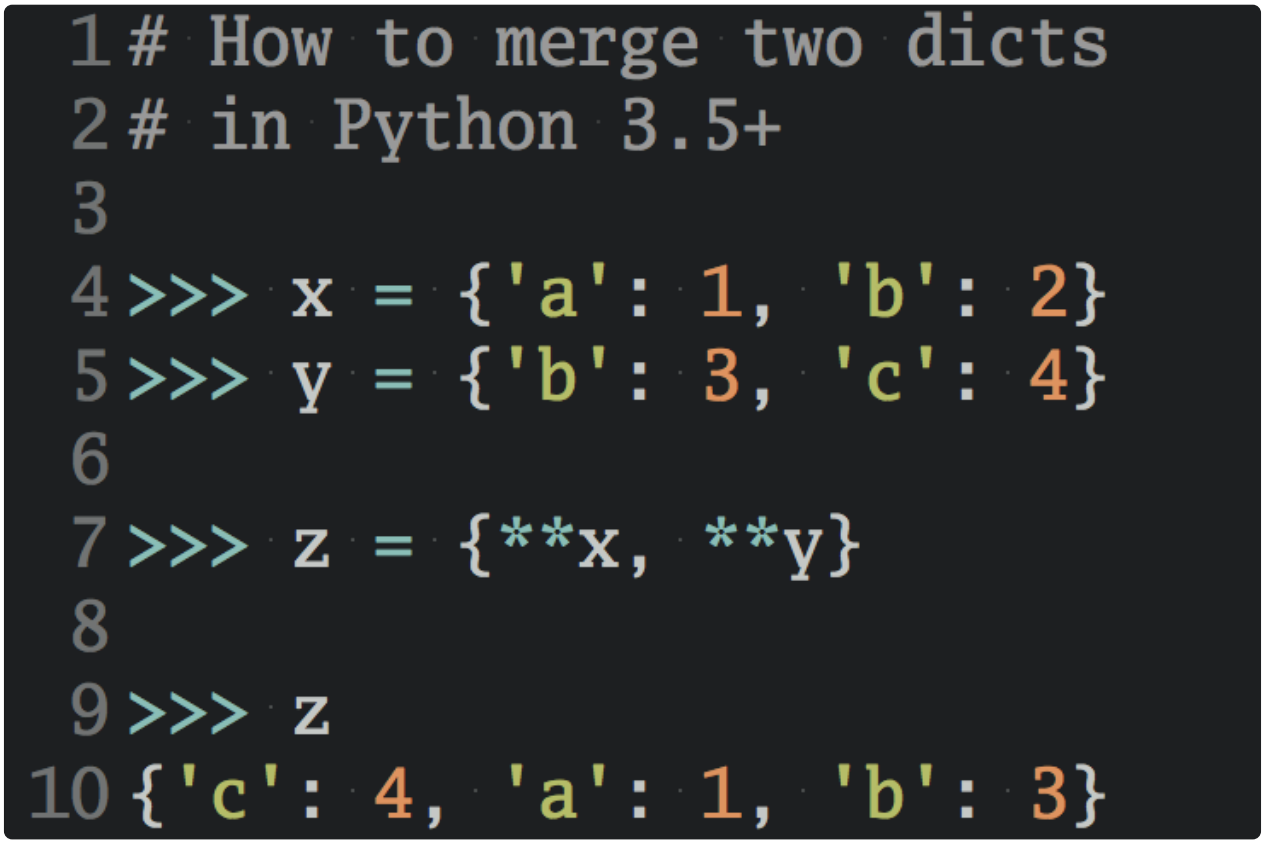
Whatever you do, you now have the information and confidence you need to write programs using Python threading!

Special thanks to reader JL Diaz for helping to clean up the introduction.

Take the Quiz: Test your knowledge with our interactive "Python Threading" quiz. Upon completion you will receive a score so you can track your learning progress over time:

Take the Quiz »

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Threading in Python**



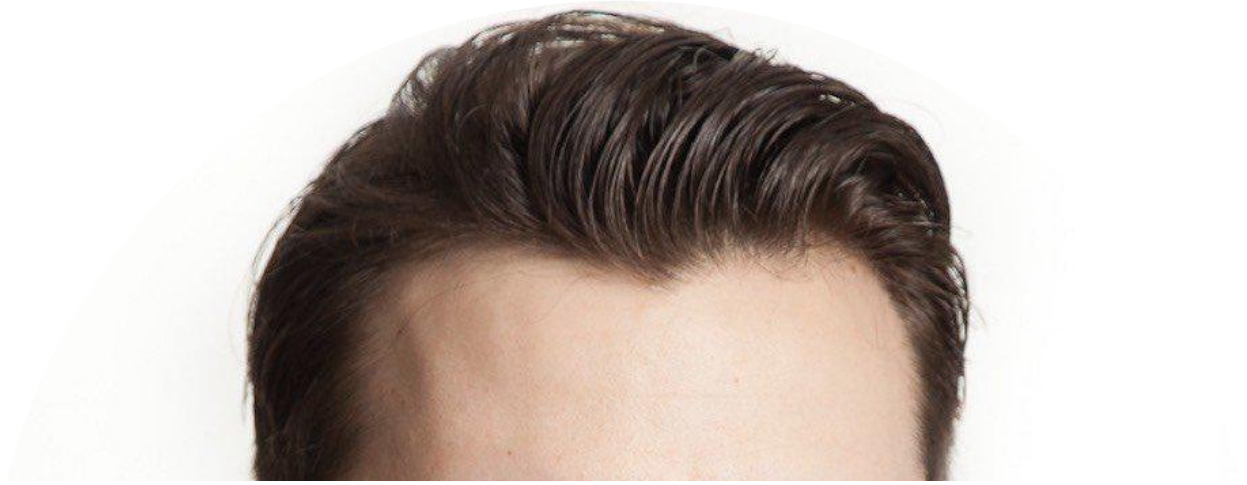
About **Jim Anderson**

Jim has been programming for a long time in a variety of languages. He has worked on embedded systems, built distributed build systems, done off-shore vendor management, and sat in many, many meetings.

» More about Jim

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

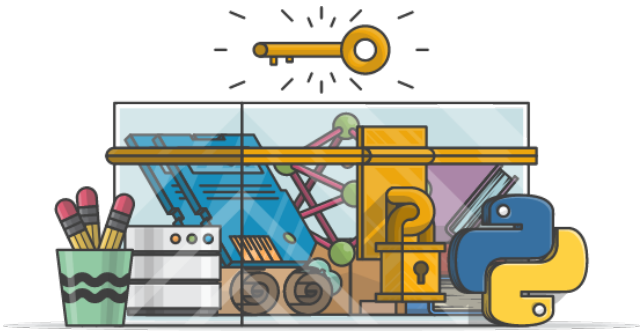
Aldren



Brad

Joanna

Master Real-World Python Skills With Unlimited Access to Real Python



Join us and get access to hundreds of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here. What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Related Tutorial Categories: best-practices intermediate

Recommended Video Course: Threading in Python