# Chapter 15. Concurrency: Threads and Processes

*Processes* are instances of running programs that the operating system protects from one another. Processes that want to communicate must explicitly arrange to do so via *interprocess communication* (IPC) mechanisms, and/or via files (covered in **Chapter 11**), databases (covered in **Chapter 12**), or network interfaces (covered in **Chapter 18**). The general way in which processes communicate using data storage mechanisms such as files and databases is that one process writes data, and another process later reads that data back. This chapter covers programming with processes, including the Python standard library modules `subprocess` and `multiprocessing`; the process-related parts of the module `os`, including simple IPC by means of *pipes*; a cross-platform IPC mechanism known as *memory-mapped files*, available in the module `mmap`; 3.8+ and the `multiprocessing.shared_memory` module.

A *thread* (originally called a "lightweight process") is a flow of control that shares global state (memory) with other threads inside a single process; all threads appear to execute simultaneously, although they may in fact be "taking turns" on one or more processors/cores. Threads are far from easy to master, and multithreaded programs are often hard to test and to debug; however, as covered in **"Threading, Multiprocessing, or Async Programming?"**, when used appropriately, multithreading may improve performance in comparison to single-threaded programming. This chapter covers various facilities Python provides for dealing with threads, including the `threading`, `queue`, and `concurrent.futures` modules.

Another mechanism for sharing control among multiple activities within a single process is what has become known as *asynchronous* (or *async*) programming. When you are reading Python code, the presence of the

keywords **`async`** and **`await`** indicate it is asynchronous. Such code depends on an *event loop*, which is, broadly speaking, the equivalent of the thread switcher used within a process. When the event loop is the scheduler, each execution of an asynchronous function becomes a *task*, which roughly corresponds with a *thread* in a multithreaded program.

Both process scheduling and thread switching are *preemptive*, which is to say that the scheduler or switcher has control of the CPU and determines when any particular piece of code gets to run. Asynchronous programming, however, is *cooperative*: each task, once execution begins, can run for as long as it chooses before indicating to the event loop that it is prepared to give up control (usually because it is awaiting the completion of some other asynchronous task, most often an I/O-focused one).

Although async programming offers great flexibility to optimize certain classes of problems, it is a programming paradigm that many programmers are unfamiliar with. Because of its cooperative nature, incautious async programming can lead to *deadlocks*, and infinite loops can starve other tasks of processor time: figuring out how to avoid deadlocks creates significant extra cognitive load for the average programmer. We do not cover asynchronous programming, including the module **`asyncio`**, further in this volume, feeling that it is a complex enough topic to be well worth a book on its own.[1]

Network mechanisms are well suited for IPC, and work just as effectively between processes running on different nodes of a network as between ones that run on the same node. The `multiprocessing` module supplies some mechanisms that are suitable for IPC over a network; **Chapter 18** covers low-level network mechanisms that provide a basis for IPC. Other, higher-level mechanisms for *distributed computing* (**CORBA**, **DCOM/COM+**, **EJB**, **SOAP**, **XML-RPC**, **.NET**, **gRPC**, etc.) can make IPC a bit easier, whether locally or remotely; however, we do not cover distributed computing in this book.

When multiprocessor computers arrived, the OS had to deal with more complex scheduling problems, and programmers who wanted maximum performance had to write their applications so that code could truly be

executed in parallel, on different processors or cores (from the programming point of view, cores are simply processors implemented on the same piece of silicon). This requires both knowledge and discipline. The CPython implementation simplifies these issues by implementing a *global interpreter lock* (GIL). In the absence of any action by the Python programmer, on CPython only the thread that holds the GIL is allowed access to the processor, effectively barring CPython processes from taking full advantage of multiprocessor hardware. Libraries such as **NumPy**, which are typically required to undertake lengthy computations of compiled code that uses none of the interpreter's facilities, arrange for their code to release the GIL during such computations. This allows effective use of multiple processors, but it isn't a technique that you can use if all your code is in pure Python.

#### THREADING, MULTIPROCESSING, OR ASYNC PROGRAMMING?

In many cases, the best answer is "none of the above!" Each of these approaches is, at best, an optimization, and (as covered in **"Optimization"**) optimization is often unneeded, or at least premature. Each such approach can be prone to bugs and hard to test and debug; stick with single threading as long as you possibly can, and keep things simple.

When you *do* need optimization, and your program is *I/O-bound* (meaning that it spends much time doing I/O), async programming is fastest, as long as you can make your I/O operations *nonblocking* ones. Second best, when your I/O absolutely *has* to be blocking, the `threading` module can help an I/O-bound program's performance.

When your program is *CPU-bound* (meaning that it spends much time performing computations), in CPython `threading` usually does not help performance. This is because the GIL ensures that only one Python-coded thread at a time can execute (this also applies to **PyPy**). C-coded extensions can "release the GIL" while they're doing a time-consuming operation; NumPy (covered in **Chapter 16**) does so for array operations, for example. As a consequence, if your program is CPU-bound via calls to lengthy CPU operations in NumPy or other similarly optimized C-coded extension, the `threading` module may help your program's performance on a multiprocessor computer (as most computers are today).

When your program is CPU-bound via pure Python code and you're using CPython or PyPy on a multiprocessor computer, the `multiprocessing` module may help performance by allowing truly parallel computation. To solve problems across multiple network-connected computers (implementing distributed computing), however, you should look at the more specialized approaches and packages discussed on the **Python wiki**, which we don't cover in this book.

## Threads in Python

Python supports multithreading on platforms that support threads, such as Windows, Linux, and just about all variants of Unix (including macOS). An action is known as *atomic* when it's guaranteed that no thread switching occurs between the start and the end of the action. In practice, in CPython, operations that *look* atomic (e.g., simple assignments and ac-

cesses) mostly *are* atomic, but only when executed on built-in types (augmented and multiple assignments, however, aren't atomic). Mostly, though, it's *not* a good idea to rely on such "atomicity." You might be dealing with an instance of a user-coded class rather than of a built-in type, in which there might be implicit calls to Python code that invalidate assumptions of atomicity. Further, relying on implementation-dependent atomicity may lock your code into a specific implementation, hampering future changes. You're better-advised to use the synchronization facilities covered in the rest of this chapter, rather than relying on atomicity assumptions.

The key design issue in multithreading systems is how best to coordinate multiple threads. The `threading` module, covered in the following section, supplies several synchronization objects. The `queue` module (discussed in **"The queue Module"**) is also very useful for thread synchronization: it supplies synchronized, thread-safe queue types, handy for communication and coordination between threads. The package `concurrent` (covered in **"The concurrent.futures Module"**) supplies a unified interface for communication and coordination that can be implemented by pools of either threads or processes.

# The threading Module

The `threading` module supplies multithreading functionality. The approach of `threading` is to model locks and conditions as separate objects (in Java, for example, such functionality is part of every object), and threads cannot be directly controlled from the outside (thus, no priorities, groups, destruction, or stopping). All methods of objects supplied by `threading` are atomic.

`threading` supplies the following thread-focused classes, all of which we'll explore in this section: `Thread`, `Condition`, `Lock`, `RLock`, `Event`, `Semaphore`, `BoundedSemaphore`, `Timer`, and `Barrier`.

`threading` also supplies a number of useful functions, including those listed in **Table 15-1**.

Table 15-1. Functions of the `threading` module

| | |
|---|---|
| `active_count` | `active_count()`<br>Returns an `int`, the number of `Thread` objects currently alive (not ones that have terminated or not yet started). |
| `current_thread` | `current_thread()`<br>Returns a `Thread` object for the calling thread. If the calling thread was not created by `threading`, `current_thread` creates and returns a semi-dummy `Thread` object with limited functionality. |
| `enumerate` | `enumerate()`<br>Returns a `list` of all `Thread` objects currently alive (not ones that have terminated or not yet started). |
| `excepthook` | `excepthook(args)`<br>**3.8+** Override this function to determine how in-thread exceptions are handled; see the **online docs** for details. The `args` argument has attributes that allow you to access exception and thread details. **3.10+** `threading.__excepthook__` holds the module's original `threadhook` value. |
| `get_ident` | `get_ident()`<br>Returns a nonzero `int` as a unique identifier among all current threads. Useful to manage and track data by thread. Thread identifiers may be reused as threads exit and new threads are created. |
| `get_native_id` | `get_native_id()`<br>**3.8+** Returns the native integer ID of the current thread as assigned by the operating system kernel. Available on most common operating systems. |

| | |
|---|---|
| `stack_size` | `stack_size([`*`size`*`])`<br>Returns the current stack size, in bytes, used for new threads, and (when *`size`* is provided) establishes the value for new threads. Acceptable values for *`size`* are subject to platform-specific constraints, such as being at least `32768` (or an even higher minimum, on some platforms), and (on some platforms) being a multiple of `4096`. Passing *`size`* as `0` is always acceptable and means "use the system's default." When you pass a value for *`size`* that is not acceptable on the current platform, `stack_size` raises a `ValueError` exception. |

## Thread Objects

A `Thread` instance *`t`* models a thread. You can pass a function to be used as *`t`*'s main function as the *`target`* argument when you create *`t`*, or you can subclass `Thread` and override its `run` method (you may also override `__init__`, but you should not override other methods). *`t`* is not yet ready to run when you create it; to make *`t`* ready (active), call *`t`*`.start`. Once *`t`* is active, it terminates when its main function ends, either normally or by propagating an exception. A `Thread` *`t`* can be a *daemon*, meaning that Python can terminate even if *`t`* is still active, while a normal (nondaemon) thread keeps Python alive until the thread terminates. The `Thread` class supplies the constructor, properties, and methods detailed in **Table 15-2**.

Table 15-2. Constructor, methods, and properties of the `Thread` class

| | |
|---|---|
| `Thread` | **`class`** `Thread(name=`**`None`**`, target=`**`None`**`, args=(),`<br>`kwargs={}, *,`<br>`daemon=`**`None`**`)`<br>*Always call `Thread` with named arguments*: the number and order of parameters is not guaranteed by the specification, but the parameter names are. You have two options when constructing a `Thread`: |

- Instantiate the class `Thread` itself with a `target` function (`t.run` then calls *`target`*`(*`*`args`*`, **`*`kwargs`*`)` when the thread is started).
- Extend the `Thread` class and override its `run` method.

In either case, execution will begin only when you call *`t`*`.start`. `name` becomes *`t`*'s name. If `name` is **`None`**, `Thread` generates a unique name for *`t`*. If a subclass *`T`* of `Thread` overrides `__init__`, *`T`*`.__init__` *must* call `Thread.__init__` on `self` (usually via the `super` built-in function) before any other `Thread` method. `daemon` can be assigned a Boolean value or, if **`None`**, will take this value from the `daemon` attribute of the creating thread.

| | |
|---|---|
| `daemon` | `daemon` is a writable Boolean property that indicates whether *`t`* is a daemon (i.e., the process can terminate even when *`t`* is still active; such a termination also ends *`t`*). You can assign to *`t`*`.daemon` only before calling *`t`*`.start`; assigning a true value sets *`t`* to be a daemon. Threads created by a daemon thread have *`t`*`.daemon` set to **`True`** by default. |
| `is_alive` | *`t`*`.is_alive()`<br>`is_alive` returns **`True`** when *`t`* is active (i.e., when *`t`*`.start` has executed and *`t`*`.run` has not yet terminated); otherwise, returns **`False`**. |
| `join` | *`t`*`.join(timeout=`**`None`**`)`<br>`join` suspends the calling thread (which must not be *`t`*) until *`t`* terminates (when *`t`* is already terminated, the calling thread does not suspend). `timeout` is covered in **"Timeout parameters"**. You can call *`t`*`.join` only after *`t`*`.start`. It's OK to call `join` more than once. |

| | |
|---|---|
| `name` | *`t`*`.name`<br>`name` is a property returning *`t`*'s name; assigning `name` rebinds *`t`*'s name (`name` exists only to help you debug; `name` need not be unique among threads). If omitted, the thread will receive a generated name `Thread-`*`n`*, where *`n`* is an incrementing integer (3.10+ and if `target` is specified, `(target.__name__)` will be appended). |
| `run` | *`t`*`.run()`<br>`run` is the method called by *`t`*`.start` that executes *`t`*'s main function. Subclasses of `Thread` can override `run`. Unless overridden, `run` calls the *`target`* callable passed on *`t`*'s creation. Do *not* call *`t`*`.run` directly; calling *`t`*`.run` is the job of *`t`*`.start`! |
| `start` | *`t`*`.start()`<br>`start` makes *`t`* active and arranges for *`t`*`.run` to execute in a separate thread. You must call *`t`*`.start` only once for any given `Thread` object *`t`*; calling it again raises an exception. |

## Thread Synchronization Objects

The `threading` module supplies several synchronization primitives (types that let threads communicate and coordinate). Each primitive type has specialized uses, discussed in the following sections.

---

YOU MAY NOT NEED THREAD SYNCHRONIZATION PRIMITIVES

As long as you avoid having (nonqueue) global variables that change and which several threads access, `queue` (covered in **"The queue Module"**) can often provide all the coordination you need, as can `concurrent` (covered in **"The concurrent.futures Module"**). **"Threaded Program Architecture"** shows how to use `Queue` objects to give your multithreaded programs simple and effective architectures, often without needing any explicit use of synchronization primitives.

---

### Timeout parameters

The synchronization primitives `Condition` and `Event` supply `wait` methods that accept an optional `timeout` argument. A `Thread` object's `join` method also accepts an optional `timeout` argument (see **Table 15-2**). Using the default `timeout` value of **`None`** results in normal blocking behavior (the calling thread suspends and waits until the desired condition is met). When it is not **`None`**, a `timeout` argument is a floating-point value that indicates an interval of time, in seconds (`timeout` can have a fractional part, so it can indicate any time interval, even a very short one). When `timeout` seconds elapse, the calling thread becomes ready again, even if the desired condition has not been met; in this case, the waiting method returns **`False`** (otherwise, the method returns **`True`**). `timeout` lets you design systems that are able to overcome occasional anomalies in a few threads, and thus are more robust. However, using `timeout` may slow your program down: when that matters, be sure to measure your code's speed accurately.

### Lock and RLock objects

`Lock` and `RLock` objects supply the same three methods, described in **Table 15-3**.

Table 15-3. Methods of an instance `L` of `Lock`

| | |
|---|---|
| `acquire` | *`L`*`.acquire(blocking=`**`True`**`, timeout=-1)`<br>When *`L`* is unlocked, or if *`L`* is an `RLock` acquired by the same thread that's calling `acquire`, this thread immediately locks it (incrementing the internal counter if *`L`* is an `RLock`, as described shortly) and returns **`True`**.<br>When *`L`* is already locked and `blocking` is **`False`**, `acquire` immediately returns **`False`**. When `blocking` is **`True`**, the calling thread is suspended until either:<br>• Another thread releases the lock, in which case this thread locks it and returns **`True`**. |

- The operation times out before the lock can be acquired, in which case `acquire` returns **`False`**. The default `-1` value never times out.

| | |
|---|---|
| `locked` | *`L`*`.locked()`<br>Returns **`True`** when *`L`* is locked; otherwise, returns **`False`**. |
| `release` | *`L`*`.release()`<br>Unlocks *`L`*, which must be locked (for an `RLock`, this means to decrement the lock count, which cannot go below zero—the lock can only be acquired by a new thread when the lock count is zero). When *`L`* is locked, any thread may call *`L`*`.release`, not just the thread that locked *`L`*. When more than one thread is blocked on *`L`* (i.e., has called *`L`*`.acquire`, found *`L`* locked, and is waiting for *`L`* to be unlocked), `release` wakes up an arbitrary one of the waiting threads. The thread calling `release` does not suspend: it remains ready and continues to execute. |

The following console session illustrates the automatic `acquire`/`release` done on locks when they are used as a context manager (as well as other data Python maintains for the lock, such as the owner thread ID and the number of times the lock's `acquire` method has been called):

```
>>> lock = threading.RLock()
>>> print(lock)
```

```
<unlocked _thread.RLock object owner=0 count=0 at 0x102878e00>
```

```
>>> with lock:
```

```
...     print(lock)
...
```

```
<locked _thread.RLock object owner=4335175040 count=1 at 0x102878e00>
```

```
>>> print(lock)
```

```
<unlocked _thread.RLock object owner=0 count=0 at 0x102878e00>
```

The semantics of an `RLock` object *r* are often more convenient (except in peculiar architectures where you need threads to be able to release locks that a different thread has acquired). `RLock` is a *reentrant* lock, meaning that when *r* is locked, it keeps track of the *owning* thread (i.e., the thread that locked it, which for an `RLock` is also the only thread that can release it—when any other thread tries to release an `RLock`, this raises a `RuntimeError` exception). The owning thread can call *r*`.acquire` again without blocking; *r* then just increments an internal count. In a similar situation involving a `Lock` object, the thread would block until some other thread releases the lock. For example, consider the following code snippet:

```
lock = threading.RLock()
global_state = []
def recursive_function(some, args):
    with lock:  # acquires lock, guarantees release at end
        # ...modify global_state...
        if more_changes_needed(global_state):
            recursive_function(other, args)
```

If `lock` was an instance of `threading.Lock`, `recursive_function` would block its calling thread when it calls itself recursively: the **`with`** statement, finding that the lock has already been acquired (even though that was

done by the same thread), would block and wait…and wait. With a `threading.RLock`, no such problem occurs: in this case, since the lock has already been acquired *by the same thread*, on getting acquired again it just increments its internal count and proceeds.

An `RLock` object *r* is unlocked only when it has been released as many times as it has been acquired. An `RLock` is useful to ensure exclusive access to an object when the object's methods call each other; each method can acquire at the start, and release at the end, the same `RLock` instance.

---

**USE WITH STATEMENTS TO AUTOMATICALLY ACQUIRE AND RELEASE SYNCHRONIZATION OBJECTS**

Using a **`try/finally`** statement (covered in **"try/finally"**) is one way to ensure that an acquired lock is indeed released. Using a **`with`** statement, covered in **"The with Statement and Context Managers"**, is usually better: all locks, conditions, and semaphores are context managers, so an instance of any of these types can be used directly in a **`with`** clause to acquire it (implicitly with blocking) and ensure it is released at the end of the **`with`** block.

---

## Condition objects

A `Condition` object *c* wraps a `Lock` or `RLock` object *L*. The class `Condition` exposes the constructor and methods described in **Table 15-4**.

Table 15-4. Constructor and methods of the `Condition` class

| | |
|---|---|
| `Condition` | **`class`** `Condition(lock=`**`None`**`)`<br>Creates and returns a new `Condition` object *c* with the lock *L* set to `lock`. If `lock` is **`None`**, *L* is set to a newly created `RLock` object. |
| `acquire`, `release` | *c*`.acquire(blocking=`**`True`**`)`, *c*`.release()`<br>These methods just call *L*'s corresponding methods. A thread must never call any other method on *c* unless the thread holds (i.e., has `acquired`) lock *L*. |

| | |
|---|---|
| `notify`, `notify_all` | `c.notify()`, `c.notify_all()`<br>`notify` wakes up an arbitrary one of the threads waiting on *c*. The calling thread must hold *L* before it calls `c.notify`, and `notify` does not release *L*. The awakened thread does not become ready until it can acquire *L* again. Therefore, the calling thread normally calls `release` after calling `notify`. `notify_all` is like `notify`, but wakes up *all* waiting threads, not just one. |
| `wait` | `c.wait(timeout=None)`<br>`wait` releases *L*, then suspends the calling thread until some other thread calls `notify` or `notify_all` on *c*. The calling thread must hold *L* before it calls `c.wait`. `timeout` is covered in **"Timeout parameters"**. After a thread wakes up, either by notification or timeout, the thread becomes ready when it acquires *L* again. When `wait` returns **`True`** (meaning it has exited normally, not by timeout), the calling thread is always holding *L* again. |

Usually, a `Condition` object *c* regulates access to some global state *s* shared among threads. When a thread must wait for *s* to change, the thread loops:

```
with c:
    while not is_ok_state(s):
        c.wait()
    do_some_work_using_state(s)
```

Meanwhile, each thread that modifies *s* calls `notify` (or `notify_all` if it needs to wake up all waiting threads, not just one) each time *s* changes:

```
with c:
    do_something_that_modifies_state(s)
```

```
        c.notify()    # or, c.notify_all()
    # no need to call c.release(), exiting 'with' intrinsically does that
```

You must always acquire and release *c* around each use of *c*'s methods: doing so via a **`with`** statement makes using `Condition` instances less error prone.

## Event objects

`Event` objects let any number of threads suspend and wait. All threads waiting on `Event` object *e* become ready when any other thread calls *e*`.set`. *e* has a flag that records whether the event happened; it is initially **`False`** when *e* is created. `Event` is thus a bit like a simplified `Condition`. `Event` objects are useful to signal one-shot changes, but brittle for more general use; in particular, relying on calls to *e*`.clear` is error prone. The `Event` class exposes the constructor and methods in **Table 15-5**.

Table 15-5. Constructor and methods of the `Event` class

| | |
|---|---|
| `Event` | **`class`** `Event()`<br>Creates and returns a new `Event` object *e*, with *e*'s flag set to **`False`**. |
| `clear` | *e*`.clear()`<br>Sets *e*'s flag to **`False`**. |
| `is_set` | *e*`.is_set()`<br>Returns the value of *e*'s flag: **`True`** or **`False`**. |
| `set` | *e*`.set()`<br>Sets *e*'s flag to **`True`**. All threads waiting on *e*, if any, become ready to run. |

`wait` `e.wait(timeout=None)`

Returns immediately if *e*'s flag is `True`; otherwise, suspends the calling thread until some other thread calls `set`. `timeout` is covered in **"Timeout parameters"**.

The following code shows how `Event` objects explicitly synchronize processing across multiple threads:

```
import datetime, random, threading, time

def runner():
    print('starting')
    time.sleep(random.randint(1, 3))
    print('waiting')
    event.wait()
    print(f'running at {datetime.datetime.now()}')

num_threads = 10
event = threading.Event()

threads = [threading.Thread(target=runner) for _ in range(num_threads)]
for t in threads:
    t.start()

event.set()

for t in threads:
    t.join()
```

### Semaphore and BoundedSemaphore objects

*Semaphores* (also known as *counting semaphores*) are a generalization of locks. The state of a `Lock` can be seen as `True` or `False`; the state of a `Semaphore` *s* is a number between `0` and some *n* set when *s* is created (both bounds included). `Semaphores` can be useful to manage a fixed pool of resources—e.g., 4 printers or 20 sockets—although it's often more robust to use `Queues` (described later in this chapter) for such purposes. The

class `BoundedSemaphore` is very similar, but raises `ValueError` if the state ever becomes higher than the initial value: in many cases, such behavior can be a useful indicator of a bug. **Table 15-6** shows the constructors of the `Semaphore` and `BoundedSemaphore` classes and the methods exposed by an object *s* of either class.

Table 15-6. Constructors and methods of the `Semaphore` and `BoundedSemaphore` classes

| | |
|---|---|
| `Semaphore`, `Bounded Semaphore` | **`class`** `Semaphore(n=1)`, **`class`** `BoundedSemaphore(n=1)`<br>`Semaphore` creates and returns a `Semaphore` object *s* with the state set to `n`; `BoundedSemaphore` is very similar, except that *s*`.release` raises `ValueError` if the state becomes higher than `n`. |
| `acquire` | *s*`.acquire(blocking=`**`True`**`)`<br>When *s*'s state is `>0`, `acquire` decrements the state by `1` and returns **`True`**. When *s*'s state is `0` and `blocking` is **`True`**, `acquire` suspends the calling thread and waits until some other thread calls *s*`.release`. When *s*'s state is `0` and `blocking` is **`False`**, `acquire` immediately returns **`False`**. |
| `release` | *s*`.release()`<br>When *s*'s state is `>0`, or when the state is `0` but no thread is waiting on *s*, `release` increments the state by `1`. When *s*'s state is `0` and some threads are waiting on *s*, `release` leaves *s*'s state at `0` and wakes up an arbitrary one of the waiting threads. The thread that calls `release` does not suspend; it remains ready and continues to execute normally. |

## Timer objects

A `Timer` object calls a specified callable, in a newly made thread, after a given delay. The class `Timer` exposes the constructor and methods in

Table 15-7.

Table 15-7. Constructor and methods of the `Timer` class

| | |
|---|---|
| `Timer` | **`class`** `Timer(`*`interval`*`,` *`callback`*`,` `args=`**`None`**`,` `kwargs=`**`None`**`)`<br>Creates an object *`t`* that calls *`callback`*, *`interval`* seconds after starting (*`interval`* is a floating-point number of seconds). |
| `cancel` | *`t`*`.cancel()`<br>Stops the timer and cancels the execution of its action, as long as *`t`* is still waiting (hasn't called its callback yet) when you call `cancel`. |
| `start` | *`t`*`.start()`<br>Starts *`t`*. |

`Timer` extends `Thread` and adds the attributes `function`, `interval`, `args`, and `kwargs`.

A `Timer` is "one-shot": *`t`* calls its callback only once. To call *`callback`* periodically, every *`interval`* seconds, here's a simple recipe—the `Periodic` timer runs *`callback`* every *`interval`* seconds, stopping only when *`callback`* raises an exception:

```
class Periodic(threading.Timer):
    def __init__(self, interval, callback, args=None, kwargs=None):
        super().__init__(interval, self._f, args, kwargs)
        self.callback = callback

    def _f(self, *args, **kwargs):
        p = type(self)(self.interval, self.callback, args, kwargs)
        p.start()
        try:
            self.callback(*args, **kwargs)
```

```
        except Exception:
            p.cancel()
```

### Barrier objects

A `Barrier` is a synchronization primitive allowing a certain number of threads to wait until they've all reached a certain point in their execution, at which point they all resume. Specifically, when a thread calls *`b`*`.wait`, it blocks until the specified number of threads have made the same call on *`b`*; at that time, all the threads blocked on *`b`* are allowed to resume.

The `Barrier` class exposes the constructor, methods, and properties listed in **Table 15-8**.

Table 15-8. Constructor, methods, and properties of the `Barrier` class

| | |
|---|---|
| `Barrier` | **`class`** `Barrier(`*`num_threads`*`, action=`**`None`**`, timeout=`**`None`**`)`<br>Creates a `Barrier` object *`b`* for *`num_threads`* threads. `action` is a callable without arguments: if you pass this argument, it executes on any single one of the blocked threads when they are all unblocked. `timeout` is covered in **"Timeout parameters"**. |
| `abort` | *`b`*`.abort()`<br>Puts `Barrier` *`b`* in the *broken* state, meaning that any thread currently waiting resumes with a `threading.BrokenBarrierException` (the same exception also gets raised on any subsequent call to *`b`*`.wait`). This is an emergency action typically used when a waiting thread is suffering some abnormal termination, to avoid deadlocking the whole program. |
| `broken` | *`b`*`.broken`<br>**`True`** when *`b`* is in the broken state; otherwise, **`False`**. |

| | |
|---|---|
| `n_waiting` | *b*`.n_waiting`<br>The number of threads currently waiting on *b*. |
| `parties` | `parties`<br>The value passed as *num_threads* in the constructor of *b*. |
| `reset` | *b*`.reset()`<br>Returns *b* to the initial empty, nonbroken state; any thread currently waiting on *b*, however, resumes with a `threading.BrokenBarrierException`. |
| `wait` | *b*`.wait()`<br>The first *b*`.parties-1` threads calling *b*`.wait` block; when the number of threads blocked on *b* is *b*`.parties-1` and one more thread calls *b*`.wait`, all the threads blocked on *b* resume. *b*`.wait` returns an `int` to each resuming thread, all distinct and in `range(`*b*`.parties)`, in unspecified order; threads can use this return value to determine which one should do what next (though passing `action` in the `Barrier`'s constructor is simpler and often sufficient). |

The following code shows how `Barrier` objects synchronize processing across multiple threads (contrast this with the example code shown earlier for `Event` objects):

```
import datetime, random, threading, time

def runner():
    print('starting')
    time.sleep(random.randint(1, 3))
    print('waiting')
    try:
        my_number = barrier.wait()
    except threading.BrokenBarrierError:
```

```
            print('Barrier abort() or reset() called, thread exiting...')
            return
        print(f'running ({my_number}) at {datetime.datetime.now()}')

def announce_release():
    print('releasing')

num_threads = 10
barrier = threading.Barrier(num_threads, action=announce_release)

threads = [threading.Thread(target=runner) for _ in range(num_threads)]
for t in threads:
    t.start()

for t in threads:
    t.join()
```

## Thread Local Storage

The `threading` module supplies the class `local`, which a thread can use to obtain *thread-local storage*, also known as *per-thread data*. An instance `L` of `local` has arbitrary named attributes that you can set and get, stored in a dictionary `L.__dict__` that you can also access. `L` is fully thread-safe, meaning there is no problem if multiple threads simultaneously set and get attributes on `L`. Each thread that accesses `L` sees a disjoint set of attributes: any changes made in one thread have no effect in other threads. For example:

```
import threading

L = threading.local()
print('in main thread, setting zop to 42')
L.zop = 42

def targ():
  print('in subthread, setting zop to 23')
  L.zop = 23
  print('in subthread, zop is now', L.zop)
```

```
t = threading.Thread(target=targ)
t.start()
t.join()
print('in main thread, zop is now', L.zop)
# prints:
# in main thread, setting zop to 42
# in subthread, setting zop to 23
# in subthread, zop is now 23
# in main thread, zop is now 42
```

Thread-local storage makes it easier to write code meant to run in multiple threads, since you can use the same namespace (an instance of `threading.local`) in multiple threads without the separate threads interfering with each other.

# The queue Module

The `queue` module supplies queue types supporting multithreaded access, with one main class `Queue`, one simplified class `SimpleQueue`, two subclasses of the main class (`LifoQueue` and `PriorityQueue`), and two exception classes (`Empty` and `Full`), described in **Table 15-9**. The methods exposed by instances of the main class and its subclasses are detailed in **Table 15-10**.

Table 15-9. Classes of the `queue` module

| | |
|---|---|
| `Queue` | **`class`** `Queue(maxsize=0)`<br>`Queue`, the main class in the module `queue`, implements a first-in, first-out (FIFO) queue: the item retrieved each time is the one that was added earliest.<br>When `maxsize > 0`, the new `Queue` instance *q* is considered full when *q* has `maxsize` items. When *q* is full, a thread inserting an item with `block=True` suspends until another thread extracts an item. When `maxsize <= 0`, *q* is never considered full and |

| | |
|---|---|
| | is limited in size only by available memory, like most Python containers. |
| `SimpleQueue` | **`class`** `SimpleQueue`<br>`SimpleQueue` is a simplified `Queue`: an unbounded FIFO queue lacking the methods `full`, `task_done`, and `join` (see **Table 15-10**) and with the method `put` ignoring its optional arguments but guaranteeing reentrancy (which makes it usable in `__del__` methods and `weakref` callbacks, where `Queue.put` would not be). |
| `LifoQueue` | **`class`** `LifoQueue(maxsize=0)`<br>`LifoQueue` is a subclass of `Queue`; the only difference is that `LifoQueue` implements a last-in, first-out (LIFO) queue, meaning the item retrieved each time is the most recently added one (often called a *stack*). |
| `PriorityQueue` | **`class`** `PriorityQueue(maxsize=0)`<br>`PriorityQueue` is a subclass of `Queue`; the only difference is that `PriorityQueue` implements a *priority* queue, meaning the item retrieved each time is the smallest one currently in the queue. Since there is no way to specify ordering, you'll typically use (*`priority`*`,` *`payload`*) pairs as items, with low values of *`priority`* meaning earlier retrieval. |
| `Empty` | `Empty` is the exception that *`q`*`.get(block=`**`False`**`)` raises when *`q`* is empty. |
| `Full` | `Full` is the exception that *`q`*`.put(`*`x`*`, block=`**`False`**`)` raises when *`q`* is full. |

An instance *q* of the class `Queue` (or either of its subclasses) supplies the methods listed in **Table 15-10**, all thread-safe and guaranteed to be atomic. For details on the methods exposed by an instance of `SimpleQueue`, see **Table 15-9**.

Table 15-10. Methods of an instance `q` of class `Queue`, `LifoQueue`, or `PriorityQueue`

| | |
|---|---|
| `empty` | *q*`.empty()`<br>Returns **`True`** when *q* is empty; otherwise, returns **`False`**. |
| `full` | *q*`.full()`<br>Returns **`True`** when *q* is full; otherwise, returns **`False`**. |
| `get`, `get_nowait` | *q*`.get(block=`**`True`**`, timeout=`**`None`**`)`, *q*`.get_nowait()`<br>When `block` is **`False`**, `get` removes and returns an item from *q* if one is available; otherwise, `get` raises `Empty`. When `block` is **`True`** and `timeout` is **`None`**, `get` removes and returns an item from *q*, suspending the calling thread, if need be, until an item is available. When `block` is **`True`** and `timeout` is not **`None`**, `timeout` must be a number `>=0` (which may include a fractional part to specify a fraction of a second), and `get` waits for no longer than `timeout` seconds (if no item is yet available by then, `get` raises `Empty`). *q*`.get_nowait()` is like *q*`.get(`**`False`**`)`, which is also like *q*`.get(timeout=0.0)`. `get` removes and returns items: in the same order as `put` inserted them (FIFO) if *q* is a direct instance of `Queue` itself, in LIFO order if *q* is an instance of `LifoQueue`, or in smallest-first order if *q* is an instance of `PriorityQueue`. |
| `put`, `put_nowait` | *q*`.put(`*`item`*`, block=`**`True`**`, timeout=`**`None`**`)`<br>*q*`.put_nowait(`*`item`*`)`<br>When `block` is **`False`**, `put` adds *`item`* to *q* if *q* is not full; otherwise, `put` raises `Full`. When `block` is **`True`** and |

| | |
|---|---|
| | `timeout` is **`None`**, `put` adds *`item`* to *`q`*, suspending the calling thread, if need be, until *`q`* is not full. When `block` is **`True`** and `timeout` is not **`None`**, `timeout` must be a number `>=0` (which may include a fractional part to specify a fraction of a second), and `put` waits for no longer than `timeout` seconds (if *`q`* is still full by then, `put` raises `Full`). *`q`*`.put_nowait(`*`item`*`)` is like *`q`*`.put(`*`item`*`, False)`, which is also like *`q`*`.put(`*`item`*`, timeout=0.0)`. |
| `qsize` | *`q`*`.qsize()`<br>Returns the number of items that are currently in *`q`*. |

*`q`* maintains an internal, hidden count of *unfinished tasks*, which starts at zero. Each call to `put` increments the count by one. To decrement the count by one, when a worker thread has finished processing a task, it calls *`q`*`.task_done`. To synchronize on "all tasks done," call *`q`*`.join`: when the count of unfinished tasks is nonzero, *`q`*`.join` blocks the calling thread, unblocking later when the count goes to zero; when the count of unfinished tasks is zero, *`q`*`.join` continues the calling thread.

You don't have to use `join` and `task_done` if you prefer to coordinate threads in other ways, but they provide a simple, useful approach when you need to coordinate systems of threads using a `Queue`.

`Queue` offers a good example of the idiom "It's easier to ask forgiveness than permission" (EAFP), covered in **"Error-Checking Strategies"**. Due to multithreading, each nonmutating method of *`q`* (`empty`, `full`, `qsize`) can only be advisory. When some other thread mutates *`q`*, things can change between the instant a thread gets information from a nonmutating method and the very next moment, when the thread acts on the information. Relying on the "look before you leap" (LBYL) idiom is therefore futile, and fiddling with locks to try to fix things is a substantial waste of effort. Avoid fragile LBYL code, such as:

```
if q.empty():
    print('no work to perform')
else:  # Some other thread may now have emptied the queue!
    x = q.get_nowait()
    work_on(x)
```

and instead use the simpler and more robust EAFP approach:

```
try:
    x = q.get_nowait()
except queue.Empty:  # Guarantees the queue was empty when accessed
    print('no work to perform')
else:
    work_on(x)
```

# The multiprocessing Module

The `multiprocessing` module supplies functions and classes to code pretty much as you would for multithreading, but distributing work across processes, rather than across threads: these include the class `Process` (analogous to `threading.Thread`) and classes for synchronization primitives (`Lock`, `RLock`, `Condition`, `Event`, `Semaphore`, `BoundedSemaphore`, and `Barrier`—each similar to the class with the same name in the `threading` module—as well as `Queue` and `JoinableQueue`, both similar to `queue.Queue`). These classes make it easy to take code written to use `threading` and port it to a version using `multiprocessing` instead; just pay attention to the differences we cover in the following subsection.

It's usually best to avoid sharing state among processes: use queues, instead, to explicitly pass messages among them. However, for those rare occasions in which you do need to share some state, `multiprocessing` supplies classes to access shared memory (`Value` and `Array`), and—more flexibly (including coordination among different computers on a network) though with more overhead—a `Process` subclass, `Manager`, de-

signed to hold arbitrary data and let other processes manipulate that data via *proxy* objects. We cover state sharing in **"Sharing State: Classes Value, Array, and Manager"**.

When you're writing new code, rather than porting code originally written to use `threading`, you can often use different approaches supplied by `multiprocessing`. The `Pool` class, in particular (covered in **"Process Pools"**), can often simplify your code. The simplest and highest-level way to do multiprocessing is to use the `concurrent.futures` module (covered in **"The concurrent.futures Module"**) along with the `ProcessPoolExecutor`.

Other highly advanced approaches, based on `Connection` objects built by the `Pipe` factory function or wrapped in `Client` and `Listener` objects, are even more flexible, but quite a bit more complex; we do not cover them further in this book. For more in-depth coverage of `multiprocessing`, refer to the **online docs**[2] and third-party online tutorials like in **PyMOTW**.

## Differences Between multiprocessing and threading

You can pretty easily port code written to use `threading` into a variant using `multiprocessing` instead—however, there are several differences you must consider.

### Structural differences

All objects that you exchange between processes (for example, via a queue, or an argument to a `Process`'s `target` function) are serialized via `pickle`, covered in **"The pickle Module"**. Therefore, you can only exchange objects that can be thus serialized. Moreover, the serialized bytestring cannot exceed about 32 MB (depending on the platform), or else an exception is raised; therefore, there are limits to the size of objects you can exchange.

Especially in Windows, child processes *must* be able to import as a module the main script that's spawning them. Therefore, be sure to guard all

top-level code in the main script (meaning code that must not be executed again by child processes) with the usual `if __name__ == '__main__'` idiom, covered in **"The Main Program"**.

If a process is abruptly killed (for example, via a signal) while using a queue or holding a synchronization primitive, it won't be able to perform proper cleanup on that queue or primitive. As a result, the queue or primitive may get corrupted, causing errors in all other processes trying to use it.

### The Process class

The class `multiprocessing.Process` is very similar to `threading.Thread`; it supplies all the same attributes and methods (see **Table 15-2**), plus a few more, listed in **Table 15-11**. Its constructor has the following signature:

| | |
|---|---|
| `Process` | **`class`** `Process(name=`**`None`**`, target=`**`None`**`, args=(), kwargs={})`<br>*Always call `Process` with named arguments*: the number and order of parameters is not guaranteed by the specification, but the parameter names are. Either instantiate the class `Process` itself, passing a `target` function (*`p`*`.run` then calls *`target(*args, **kwargs)`* when the thread is started); or, instead of passing `target`, extend the `Process` class and override its `run` method. In either case, execution will begin only when you call *`p`*`.start`. `name` becomes *`p`*'s name. If `name` is **`None`**, `Process` generates a unique name for *`p`*. If a subclass *`P`* of `Process` overrides `__init__`, *`P`*`.__init__` *must* call `Process.__init__` on `self` (usually via the `super` built-in function) before any other `Process` method. |

Table 15-11. Additional attributes and methods of the `Process` class

| | |
|---|---|
| `authkey` | The process's authorization key, a bytestring. This is initialized to random bytes supplied by `os.urandom`, but you can reassign it later if you wish. Used in the authorization handshake for advanced uses we do not cover in this book. |
| `close` | `close()`<br>Closes a `Process` instance and releases all resources associated with it. If the underlying process is still running, raises `ValueError`. |
| `exitcode` | **`None`** when the process has not exited yet; otherwise, the process's exit code. This is an `int`: `0` for success, `>0` for failure, `<0` when the process was killed. |
| `kill` | `kill()`<br>Same as `terminate`, but on Unix sends a `SIGKILL` signal. |
| `pid` | **`None`** when the process has not started yet; otherwise, the process's identifier as set by the operating system. |
| `terminate` | `terminate()`<br>Kills the process (without giving it a chance to execute termination code, such as cleanup of queues and synchronization primitives; beware of the likelihood of causing errors when the process is using a queue or holding a synchronization primitive!). |

### Differences in queues

The class `multiprocessing.Queue` is very similar to `queue.Queue`, except that an instance *q* of `multiprocessing.Queue` does *not* supply the methods `join` and `task_done` (described in **"The queue Module"**). When methods of *q* raise exceptions due to timeouts, they raise instances of

`queue.Empty` or `queue.Full`. `multiprocessing` has no equivalents to `queue`'s `LifoQueue` and `PriorityQueue` classes.

The class `multiprocessing.JoinableQueue` does supply the methods `join` and `task_done`, but with a semantic difference compared to `queue.Queue`: with an instance *q* of `multiprocessing.JoinableQueue`, the process that calls `q.get` *must* call `q.task_done` when it's done processing that unit of work (it's not optional, as it is when using `queue.Queue`).

All objects you `put` in multiprocessing queues must be serializable by `pickle`. There may be a delay between the time you execute `q.put` and the time the object is available from `q.get`. Lastly, remember that an abrupt exit (crash or signal) of a process using *q* may leave *q* unusable for any other process.

## Sharing State: Classes Value, Array, and Manager

To use shared memory to hold a single primitive value in common among two or more processes, `multiprocessing` supplies the class `Value`, and for a fixed-length array of primitive values, it provides the class `Array`. For more flexibility (including sharing nonprimitive values and "sharing" among different systems joined by a network but sharing no memory), at the cost of higher overhead, `multiprocessing` supplies the class `Manager`, which is a subclass of `Process`. We'll look at each of these in the following subsections.

### The Value class

The constructor for the class `Value` has the signature:

`Value` **`class`** `Value(`*`typecode`*`, *`*`args`*`, *, lock=`**`True`**`)`
*`typecode`* is a string defining the primitive type of the value, just like for the `array` module, covered in **"The array Module"**. (Alternatively, *`typecode`* can be a type from the module `ctypes`, discussed in **"ctypes" in Chapter 25**, but this is rarely necessary.) *`args`* is passed

on to the type's constructor: therefore, *args* is either absent (in which case the primitive is initialized as per its default, typically `0`) or a single value, which is used to initialize the primitive.
When `lock` is **`True`** (the default), `Value` makes and uses a new lock to guard the instance. Alternatively, you can pass as `lock` an existing `Lock` or `RLock` instance. You can even pass `lock=`**`False`**, but that is rarely advisable: when you do, the instance is not guarded (thus, it is not synchronized among processes) and is missing the method `get_lock`. If you do pass `lock`, you *must* pass it as a named argument, using `lock=`*`something`*.

An instance *v* of the class `Value` supplies the method `get_lock`, which returns (but neither acquires nor releases) the lock guarding *v*, and the read/write attribute `value`, used to set and get *v*'s underlying primitive value.

To ensure atomicity of operations on *v*'s underlying primitive value, guard the operation in a **`with`** *v*`.get_lock():` statement. A typical example of such usage might be for augmented assignment, as in:

```
with v.get_lock():
    v.value += 1
```

If any other process does an unguarded operation on that same primitive value, however—even an atomic one such as a simple assignment like *v*`.value =` *x*—all bets are off: the guarded operation and the unguarded one can get your system into a *race condition*.[3] Play it safe: if *any* operation at all on *v*`.value` is not atomic (and thus needs to be guarded by being within a **`with`** *v*`.get_lock():` block), guard *all* operations on *v*`.value` by placing them within such blocks.

### The Array class

A `multiprocessing.Array` is a fixed-length array of primitive values, with all items of the same primitive type. The constructor for the class `Array` has the signature:

`Array` **class** `Array(`*typecode*`, `*size_or_initializer*`, *, lock=`**True**`)`

*typecode* is a string defining the primitive type of the value, just like for the module `array`, as covered in **"The array Module"**. (Alternatively, *typecode* can be a type from the module `ctypes`, discussed in **"ctypes" in Chapter 25**, but this is rarely necessary.)

*size_or_initializer* can be an iterable, used to initialize the array, or an integer used as the length of the array, in which case each item of the array is initialized to `0`.

When `lock` is **`True`** (the default), `Array` makes and uses a new lock to guard the instance. Alternatively, you can pass as `lock` an existing `Lock` or `RLock` instance. You can even pass `lock=`**`False`**, but that is rarely advisable: when you do, the instance is not guarded (thus it is not synchronized among processes) and is missing the method `get_lock`. If you do pass `lock`, you *must* pass it as a named argument, using `lock=`*something*.

An instance *a* of the class `Array` supplies the method `get_lock`, which returns (but neither acquires nor releases) the lock guarding *a*.

*a* is accessed by indexing and slicing, and modified by assigning to an indexing or to a slice. *a* is fixed length: therefore, when you assign to a slice, you must assign an iterable of exactly the same length as the slice you're assigning to. *a* is also iterable.

In the special case where *a* was built with a *typecode* of `'c'`, you can also access *a*`.value` to get *a*'s contents as a bytestring, and you can assign to

`a.value` any bytestring no longer than `len(a)`. When `s` is a bytestring with `len(s) < len(a)`, `a.value = s` means `a[:len(s)+1] = s + b'\0'`; this mirrors the representation of `char` strings in the C language, terminated with a `0` byte. For example:

```
a = multiprocessing.Array('c', b'four score and seven')
a.value = b'five'
print(a.value)   # prints b'five'
print(a[:])      # prints b'five\x00score and seven'
```

### The Manager class

`multiprocessing.Manager` is a subclass of `multiprocessing.Process`, with the same methods and attributes. In addition, it supplies methods to build an instance of any of the multiprocessing synchronization primitives, plus `Queue`, `dict`, `list`, and `Namespace`, the latter being a class that just lets you set and get arbitrary named attributes. Each of the methods has the name of the class whose instances it builds, and returns a *proxy* to such an instance, which any process can use to call methods (including special methods, such as indexing of instances of `dict` or `list`) on the instance held in the manager process.

Proxy objects pass most operators, and accesses to methods and attributes, on to the instance they proxy for; however, they don't pass on *comparison* operators—if you need a comparison, you need to take a local copy of the proxied object. For example:

```
manager = multiprocessing.Manager()
p = manager.list()

p[:] = [1, 2, 3]
print(p == [1, 2, 3])       # prints False,  it compares with p itself
print(list(p) == [1, 2, 3]) # prints True,  it compares with copy
```

The constructor of `Manager` takes no arguments. There are advanced ways to customize `Manager` subclasses to allow connections from unrelated processes (including ones on different computers connected via a network) and to supply a different set of building methods, but we do not cover them in this book. Rather, one simple, often-sufficient approach to using `Manager` is to explicitly transfer to other processes the proxies it produces, typically via queues, or as arguments to a `Process`'s *`target`* function.

For example, suppose there is a long-running, CPU-bound function *`f`* that, given a string as an argument, eventually returns a corresponding result; given a `set` of strings, we want to produce a `dict` with the strings as keys and the corresponding results as values. To be able to follow on which processes *`f`* runs, we also `print` the process ID just before calling *`f`*. **Example 15-1** shows one way to do this.

**Example 15-1. Distributing work to multiple worker processes**

```
import multiprocessing as mp
def f(s):
    """Run a long time, and eventually return a result."""
    import time, random
    time.sleep(random.random()*2)  # simulate slowness
    return s+s                     # some computation or other

def runner(s, d):
    print(os.getpid(), s)
    d[s] = f(s)

def make_dict(strings):
    mgr = mp.Manager()
    d = mgr.dict()
    workers = []
    for s in strings:
        p = mp.Process(target=runner, args=(s, d))
        p.start()
        workers.append(p)
    for p in workers:
        p.join()
    return {**d}
```

## Process Pools

In real life, you should always avoid creating an unbounded number of worker processes, as we did in **Example 15-1**. Performance benefits accrue only up to the number of cores in your machine (available by calling `multiprocessing.cpu_count`), or a number just below or just above this, depending on such minutiae as your platform, how CPU-bound or I/O-bound your code is, other tasks running on your computer, etc. Making many more worker processes than such an optimal number incurs substantial extra overhead without any compensating benefit.

As a consequence, it's a common design pattern to start a *pool* with a limited number of worker processes, and farm out work to them. The class `multiprocessing.Pool` lets you orchestrate this pattern.

### The Pool class

The constructor for the class `Pool` has the signature:

`Pool` **`class`** `Pool(processes=None, initializer=`**`None`**`, initargs=(), maxtasksperchild=`**`None`**`)`

`processes` is the number of processes in the pool; it defaults to the value returned by `cpu_count`. When `initializer` is not **`None`**, it's a function, called at the start of each process in the pool, with `initargs` as arguments, like `initializer(*`*`initargs`*`)`.

When `maxtasksperchild` is not **`None`**, it's the maximum number of tasks that can be executed in each process in the pool. When a process in the pool has executed that many tasks, it terminates, then a new process starts and joins the pool. When `maxtasksperchild` is **`None`** (the default), each process lives as long as the pool.

An instance *p* of the class `Pool` supplies the methods listed in **Table 15-12** (each of them must be called only in the process that built instance *p*).

Table 15-12. Methods of an instance p of class `Pool`

| | |
|---|---|
| `apply` | `apply(`*`func`*`, args=(), kwds={})`<br>In an arbitrary one of the worker processes, runs *`func`*`(*`*`args`*`, **`*`kwds`*`)`, waits for it to finish, and returns *`func`*'s result. |
| `apply_async` | `apply_async(`*`func`*`, args=(), kwds={}, callback=`**`None`**`)`<br>In an arbitrary one of the worker processes, starts running *`func`*`(*`*`args`*`, **`*`kwds`*`)` and, without waiting for it to finish, immediately returns an `AsyncResult` instance, which eventually gives *`func`*'s result, when that result is ready. (The `AsyncResult` class is discussed in the following section.) When `callback` is not **`None`**, it's a function to call (in a new, separate thread in the process that calls `apply_async`), with *`func`*'s result as the only argument, when that result is ready; `callback` should execute rapidly, because otherwise it blocks the calling process. `callback` may mutate its argument if that argument is mutable; `callback`'s return value is irrelevant (so, the best, clearest style is to have it return **`None`**). |
| `close` | `close()`<br>Sets a flag prohibiting further submissions to the pool. Worker processes terminate when they're done with all outstanding tasks. |
| `imap` | `imap(`*`func`*`, `*`iterable`*`, chunksize=1)`<br>Returns an iterator calling *`func`* on each item of *`iterable`*, in order. `chunksize` determines how many consecutive items are sent to each process; on a very long *`iterable`*, a large `chunksize` can improve performance. When `chunksize` is `1` (the default), the returned iterator has a method `next` (even though the canonical name of the iterator's method is `__next__`), |

| | |
|---|---|
| | accepting an optional *timeout* argument (a floating-point value, in seconds) and raising multiprocessing.TimeoutError should the result not yet be ready after *timeout* seconds. |
| imap_unordered | imap_unordered(*func*, *iterable*, chunksize=1)<br>Same as imap, but the ordering of the results is arbitrary (this can sometimes improve performance when the order of iteration is unimportant). It is usually helpful if the function's return value includes enough information to allow the results to be associated with the values from the iterable used to generate them. |
| join | join()<br>Waits for all worker processes to exit. You must call close or terminate before you call join. |
| map | map(*func*, *iterable*, chunksize=1)<br>Calls *func* on each item of *iterable*, in order, in worker processes in the pool; waits for them all to finish, and returns the list of results. chunksize determines how many consecutive items are sent to each process; on a very long *iterable*, a large chunksize can improve performance. |
| map_async | map_async(*func*, *iterable*, chunksize=1, callback=**None**)<br>Arranges for *func* to be called on each item of *iterable* in worker processes in the pool; without waiting for any of this to finish, immediately returns an AsyncResult instance (described in the following section), which eventually gives the list of *func*'s results, when that list is ready.<br>When callback is not **None**, it's a function to call (in a separate thread in the process that calls map_async) |

| | |
|---|---|
| | with the list of *func*'s results, in order, as the only argument, when that list is ready; `callback` should execute rapidly, since otherwise it blocks the process. `callback` may mutate its list argument; `callback`'s return value is irrelevant (so, best, clearest style is to have it return **None**). |
| `terminate` | `terminate()`<br>Terminates all worker processes at once, without waiting for them to complete work. |

For example, here's a `Pool`-based approach to perform the same task as the code in **Example 15-1**:

```
import os, multiprocessing as mp
def f(s):
    """Run a long time, and eventually return a result."""
    import time, random
    time.sleep(random.random()*2)  # simulate slowness
    return s+s                     # some computation or other

def runner(s):
    print(os.getpid(), s)
    return s, f(s)

def make_dict(strings):
    with mp.Pool() as pool:
        d = dict(pool.imap_unordered(runner, strings))
        return d
```

### The AsyncResult class

The methods `apply_async` and `map_async` of the class `Pool` return an instance of the class `AsyncResult`. An instance *r* of the class `AsyncResult` supplies the methods listed in **Table 15-13**.

Table 15-13. Methods of an instance `r` of class `AsyncResult`

| | |
|---|---|
| `get` | `get(timeout=None)`<br>Blocks and returns the result when ready, or re-raises the exception raised while computing the result. When `timeout` is not `None`, it's a floating-point value in seconds; `get` raises `multiprocessing.TimeoutError` should the result not yet be ready after `timeout` seconds. |
| `ready` | `ready()`<br>Does not block; returns `True` if the call has completed with a result or has raised an exception; otherwise, returns `False`. |
| `successful` | `successful()`<br>Does not block; returns `True` if the result is ready and the computation did not raise an exception, or returns `False` if the computation raised an exception. If the result is not yet ready, `successful` raises `AssertionError`. |
| `wait` | `wait(timeout=None)`<br>Blocks and waits until the result is ready. When `timeout` is not `None`, it's a floating-point value in seconds: `wait` raises `multiprocessing.TimeoutError` should the result not yet be ready after `timeout` seconds. |

### The ThreadPool class

The `multiprocessing.pool` module also offers a class called `ThreadPool`, with exactly the same interface as `Pool`, implemented with multiple threads within a single process (not with multiple processes, despite the module's name). The equivalent `make_dict` code to **Example 15-1** using a `ThreadPool` would be:

```
def make_dict(strings):
    num_workers=3
    with mp.pool.ThreadPool(num_workers) as pool:
        d = dict(pool.imap_unordered(runner, strings))
        return d
```

Since a `ThreadPool` uses multiple threads but is limited to running in a single process, it is most suitable for applications where the separate threads are performing overlapping I/O. As stated previously, Python threading offers little advantage when the work is primarily CPU-bound.

In modern Python, you should generally prefer the `Executor` abstract class from the module `concurrent.futures`, covered in next section, and its two implementations, `ThreadPoolExecutor` and `ProcessPoolExecutor`. In particular, the `Future` objects returned by `submit` methods of the executor classes implemented by `concurrent.futures` are compatible with the `asyncio` module (which, as previously mentioned, we do not cover in this book, but which is nevertheless a crucial part of much concurrent processing in recent versions of Python). The `AsyncResult` objects returned by the methods `apply_async` and `map_async` of the pool classes implemented by `multiprocessing` are not `asyncio` compatible.

# The concurrent.futures Module

The `concurrent` package supplies a single module, `futures`. `concurrent.futures` provides two classes, `ThreadPoolExecutor` (using threads as workers) and `ProcessPoolExecutor` (using processes as workers), which implement the same abstract interface, `Executor`. Instantiate either kind of pool by calling the class with one argument, `max_workers`, specifying how many threads or processes the pool should contain. You can omit `max_workers` to let the system pick the number of workers.

An instance *e* of the `Executor` class supports the methods in **Table 15-14**.

Table 15-14. Methods of an instance *e* of class `Executor`

| | |
|---|---|
| `map` | `map(`*`func`*`, *`*`iterables`*`, timeout=`**`None`**`, chunksize=1)`<br>Returns an iterator *`it`* whose items are the results of *`func`* called with one argument from each of the *`iterables`*, in order (using multiple worker threads or processes to execute *`func`* in parallel). When *`timeout`* is not **`None`**, it's a `float` number of seconds: should `next(`*`it`*`)` not produce any result in `timeout` seconds, raises `concurrent.futures.TimeoutError`.<br>You may also optionally specify (by name, only) argument `chunksize`: ignored for a `ThreadPoolExecutor`; for a `ProcessPoolExecutor` it sets how many items of each iterable in *`iterables`* are passed to each worker process. |
| `shutdown` | `shutdown(wait=`**`True`**`)`<br>No more calls to `map` or `submit` allowed. When `wait` is **`True`**, `shutdown` blocks until all pending futures are done; when **`False`**, `shutdown` returns immediately. In either case, the process does not terminate until all pending futures are done. |
| `submit` | `submit(`*`func`*`, *`*`a`*`, **`*`k`*`)`<br>Ensures *`func`*`(*`*`a`*`, **`*`k`*`)` executes on an arbitrary one of the pool's processes or threads. Does not block, but rather immediately returns a `Future` instance. |

Any instance of an `Executor` is also a context manager, and therefore suitable for use on a **`with`** statement (`__exit__` being like `shutdown(wait=`**`True`**`)`).

For example, here's a `concurrent`-based approach to perform the same task as in **Example 15-1**:

```
import concurrent.futures as cf
def f(s):
    """run a long time and eventually return a result"""
    # ... like before!

def runner(s):
    return s, f(s)

def make_dict(strings):
    with cf.ProcessPoolExecutor() as e:
        d = dict(e.map(runner, strings))
    return d
```

The `submit` method of an `Executor` returns a `Future` instance. A `Future` instance *f* supplies the methods described in **Table 15-15**.

Table 15-15. Methods of an instance `f` of class `Future`

| | |
|---|---|
| `add_done_callback` | `add_done_callback(`*func*`)`<br>Adds callable *func* to *f*; *func* gets called, with *f* as the only argument, when *f* completes (i.e., is canceled, or finishes). |
| `cancel` | `cancel()`<br>Tries canceling the call. Returns **`False`** when the call is being executed and cannot be canceled; otherwise, returns **`True`**. |
| `cancelled` | `cancelled()`<br>Returns **`True`** if the call was successfully canceled; otherwise, returns **`False`**. |
| `done` | `done()`<br>Returns **`True`** when the call is completed (i.e., finished, or successfully canceled). |

| | |
|---|---|
| `exception` | `exception(timeout=None)`<br>Returns the exception raised by the call, or `None` if the call raised no exception. When `timeout` is not `None`, it's a `float` number of seconds to wait. If the call hasn't completed after `timeout` seconds, `exception` raises `concurrent.futures.TimeoutError`; if the call is canceled, `exception` raises `concurrent.futures.CancelledError`. |
| `result` | `result(timeout=None)`<br>Returns the call's result. When `timeout` is not `None`, it's a `float` number of seconds. If the call hasn't completed within `timeout` seconds, `result` raises `concurrent.futures.TimeoutError`; if the call is canceled, `result` raises `concurrent.futures.CancelledError`. |
| `running` | `running()`<br>Returns `True` when the call is executing and cannot be canceled; otherwise, returns `False`. |

The `concurrent.futures` module also supplies two functions, detailed in Table 15-16.

Table 15-16. Functions of the `concurrent.futures` module

| | |
|---|---|
| `as_completed` | `as_completed(fs, timeout=None)`<br>Returns an iterator *it* over the `Future` instances that are the items of iterable *fs*. If there are duplicates in *fs*, each gets yielded just once. *it* yields one completed future at a time, in order, as they complete. If `timeout` is not `None`, it's a `float` number of seconds; should it ever happen that no new future can yet be yielded within `timeout` seconds from the previous one, `as_completed` raises `concurrent.futures.Timeout`. |

`wait` `wait(fs, timeout=None, return_when=ALL_COMPLETED)`

Waits for the `Future` instances that are the items of iterable *`fs`*. Returns a named 2-tuple of `sets`: the first `set`, named `done`, contains the futures that completed (meaning that they either finished or were canceled) before `wait` returned; the second `set`, named `not_done`, contains as-yet-uncompleted futures.

`timeout`, if not **`None`**, is a `float` number of seconds, the maximum time `wait` lets elapse before returning (when `timeout` is **`None`**, `wait` returns only when `return_when` is satisfied, no matter how much time elapses before that happens).

`return_when` controls when, exactly, `wait` returns; it must be one of three constants supplied by the module `concurrent.futures`:

`ALL_COMPLETED`
: Return when all futures finish or are canceled.

`FIRST_COMPLETED`
: Return when any future finishes or is canceled.

`FIRST_EXCEPTION`
: Return when any future raises an exception; should no future raise an exception, becomes equivalent to `ALL_COMPLETED`.

This version of `make_dict` illustrates how to use `concurrent.futures.as_completed` to process each task as it finishes (in contrast with the previous example using `Executor.map`, which always returns the tasks in the order in which they were submitted):

```
import concurrent.futures as cf

def make_dict(strings):
    with cf.ProcessPoolExecutor() as e:
```

```
        futures = [e.submit(runner, s) for s in strings]
        d = dict(f.result() for f in cf.as_completed(futures))
    return d
```

# Threaded Program Architecture

A threaded program should always try to arrange for a *single* thread to "own" any object or subsystem that is external to the program (such as a file, a database, a GUI, or a network connection). Having multiple threads that deal with the same external object is possible, but can often create intractable problems.

When your threaded program must deal with some external object, devote a dedicated thread to just such dealings, and use a `Queue` object from which the external-interfacing thread gets work requests that other threads post. The external-interfacing thread returns results by putting them on one or more other `Queue` objects. The following example shows how to package this architecture into a general, reusable class, assuming that each unit of work on the external subsystem can be represented by a callable object:

```
import threading, queue

class ExternalInterfacing(threading.Thread):
    def __init__(self, external_callable, **kwds):
        super().__init__(**kwds)
        self.daemon = True
        self.external_callable = external_callable
        self.request_queue = queue.Queue()
        self.result_queue = queue.Queue()
        self.start()

    def request(self, *args, **kwds):
        """called by other threads as external_callable would be"""
        self.request_queue.put((args, kwds))
        return self.result_queue.get()
```

```
    def run(self):
        while True:
            a, k = self.request_queue.get()
            self.result_queue.put(self.external_callable(*a, **k))
```

Once some `ExternalInterfacing` object *`ei`* is instantiated, any other thread may call *`ei`*`.request` just as it would call `external_callable` absent such a mechanism (with or without arguments, as appropriate). The advantage of `ExternalInterfacing` is that calls to `external_callable` are *serialized*. This means that just one thread (the `Thread` object bound to *`ei`*) performs them, in some defined sequential order, without overlap, race conditions (hard-to-debug errors that depend on which thread just happens to "get there" first), or other anomalies that might otherwise result.

If you need to serialize several callables together, you can pass the callable as part of the work request, rather than passing it at the initialization of the class `ExternalInterfacing`, for greater generality. The following example shows this more general approach:

```
import threading, queue

class Serializer(threading.Thread):
    def __init__(self, **kwds):
        super().__init__(**kwds)
        self.daemon = True
        self.work_request_queue = queue.Queue()
        self.result_queue = queue.Queue()
        self.start()

    def apply(self, callable, *args, **kwds):
        """called by other threads as `callable` would be"""
        self.work_request_queue.put((callable, args, kwds))
        return self.result_queue.get()

    def run(self):
        while True:
            callable, args, kwds = self.work_request_queue.get()
            self.result_queue.put(callable(*args, **kwds))
```

Once a `Serializer` object *ser* has been instantiated, any other thread may call *ser*`.apply(external_callable)` just as it would call `external_callable` without such a mechanism (with or without further arguments, as appropriate). The `Serializer` mechanism has the same advantages as `ExternalInterfacing`, except that all calls to the same or different callables wrapped by a single *ser* instance are now serialized.

The user interface of the whole program is an external subsystem, and thus should be dealt with by a single thread—specifically, the main thread of the program (this is mandatory for some user interface toolkits, and advisable even when using other toolkits that don't mandate it). A `Serializer` thread is therefore inappropriate. Rather, the program's main thread should deal only with user-interface issues, and farm out all actual work to worker threads that accept work requests on a `Queue` object and return results on another. A set of worker threads is generally known as a *thread pool*. As shown in the following example, all worker threads should share a single queue of requests and a single queue of results, since the main thread is the only one to post work requests and harvest results:

```
import threading

class Worker(threading.Thread):
    IDlock = threading.Lock()
    request_ID = 0

    def __init__(self, requests_queue, results_queue, **kwds):
        super().__init__(**kwds)
        self.daemon = True
        self.request_queue = requests_queue
        self.result_queue = results_queue
        self.start()

    def perform_work(self, callable, *args, **kwds):
        """called by main thread as `callable` would be,
                           but w/o return"""
        with self.IDlock:
            Worker.request_ID += 1
```

```
            self.request_queue.put(
                (Worker.request_ID, callable, args, kwds))
            return Worker.request_ID

    def run(self):
        while True:
            request_ID, callable, a, k = self.request_queue.get()
            self.result_queue.put((request_ID, callable(*a, **k)))
```

The main thread creates the two queues, then instantiates worker threads, as follows:

```
import queue
requests_queue = queue.Queue()
results_queue = queue.Queue()
number_of_workers = 5
for i in range(number_of_workers):
    worker = Worker(requests_queue, results_queue)
```

Whenever the main thread needs to farm out work (execute some callable object that may take substantial time to produce results), the main thread calls *`worker`*`.perform_work(`*`callable`*`)`, much as it would call *`callable`* without such a mechanism (with or without further arguments, as appropriate). However, `perform_work` does not return the result of the call. Instead of the results, the main thread gets an ID that identifies the work request. When the main thread needs the results, it can keep track of that ID, since the request's results are tagged with the ID when they appear. The advantage of this mechanism is that the main thread never blocks waiting for the callable's execution to complete, but rather becomes ready again at once and can immediately return to its main business of dealing with the user interface.

The main thread must arrange to check the `results_queue`, since the result of each work request eventually appears there, tagged with the request's ID, when the worker thread that took that request from the queue finishes computing the result. How the main thread arranges to check for both user interface events and the results coming back from

worker threads onto the results queue depends on what user interface toolkit is used, or—if the user interface is text-based—on the platform on which the program runs.

A widely applicable, though not always optimal, general strategy is for the main thread to *poll* (check the state of the results queue periodically). On most Unix-like platforms, the function `alarm` of the module `signal` allows polling. The `tkinter` GUI toolkit supplies an `after` method that is usable for polling. Some toolkits and platforms afford more effective strategies (such as letting a worker thread alert the main thread when it places some result on the results queue), but there is no generally available, cross-platform, cross-toolkit way to arrange for this. Therefore, the following artificial example ignores user interface events and just simulates work by evaluating random expressions, with random delays, on several worker threads, thus completing the previous example:

```
import random, time, queue, operator
# copy here class Worker as defined earlier

requests_queue = queue.Queue()
results_queue = queue.Queue()

number_of_workers = 3
workers = [Worker(requests_queue, results_queue)
           for i in range(number_of_workers)]
work_requests = {}

operations = {
    '+': operator.add,
    '-': operator.sub,
    '*': operator.mul,
    '/': operator.truediv,
    '%': operator.mod,
}

def pick_a_worker():
    return random.choice(workers)

def make_work():
```

```
    o1 = random.randrange(2, 10)
    o2 = random.randrange(2, 10)
    op = random.choice(list(operations))
    return f'{o1} {op} {o2}'

def slow_evaluate(expression_string):
    time.sleep(random.randrange(1, 5))
    op1, oper, op2 = expression_string.split()
    arith_function = operations[oper]
    return arith_function(int(op1), int(op2))

def show_results():
    while True:
        try:
            completed_id, results = results_queue.get_nowait()
        except queue.Empty:
            return
        work_expression = work_requests.pop(completed_id)
        print(f'Result {completed_id}: {work_expression} -> {results}')

for i in range(10):
    expression_string = make_work()
    worker = pick_a_worker()
    request_id = worker.perform_work(slow_evaluate, expression_string)
    work_requests[request_id] = expression_string
    print(f'Submitted request {request_id}: {expression_string}')
    time.sleep(1.0)
    show_results()

while work_requests:
    time.sleep(1.0)
    show_results()
```

# Process Environment

The operating system supplies each process *P* with an *environment*, a set of variables whose names are strings (most often, by convention, upper-case identifiers) and whose values are also strings. In **“Environment Variables”**, we cover environment variables that affect Python’s opera-

tions. Operating system shells offer ways to examine and modify the environment via shell commands and other means mentioned in that section.

---

**PROCESS ENVIRONMENTS ARE SELF-CONTAINED**

The environment of any process *P* is determined when *P* starts. After startup, only *P* itself can change *P*'s environment. Changes to *P*'s environment affect only *P*: the environment is *not* a means of interprocess communication. Nothing that *P* does affects the environment of *P*'s parent process (the process that started *P*), nor that of any child process *previously* started from *P* and now running, or of any process unrelated to *P*. Child processes of *P* normally get a copy of *P*'s environment as it stands at the time *P* creates that process as a starting environment. In this narrow sense, changes to *P*'s environment do affect child processes that *P* starts *after* such changes.

---

The module `os` supplies the attribute `environ`, a mapping that represents the current process's environment. When Python starts, it initializes `os.environ` from the process environment. Changes to `os.environ` update the current process's environment if the platform supports such updates. Keys and values in `os.environ` must be strings. On Windows (but not on Unix-like platforms), keys into `os.environ` are implicitly uppercased. For example, here's how to try to determine which shell or command processor you're running under:

```
import os
shell = os.environ.get('COMSPEC')
if shell is None:
    shell = os.environ.get('SHELL')
if shell is None:
    shell = 'an unknown command processor'
print('Running under ', shell)
```

When a Python program changes its environment (e.g., via `os.environ['X'] = 'Y'`), this does not affect the environment of the shell or command processor that started the program. As already explained—and for **all** programming languages, including Python—changes to a

process's environment affect only the process itself, not other processes that are currently running.

# Running Other Programs

You can run other programs via low-level functions in the `os` module, or (at a higher and usually preferable level of abstraction) with the `subprocess` module.

## Using the Subprocess Module

The `subprocess` module supplies one very broad class: `Popen`, which supports many diverse ways for your program to run another program. The constructor for `Popen` has the signature:

`Popen`

**`class`** `Popen(`*`args`*`, bufsize=0, executable=`**`None`**`, capture_output=`**`False`**`, stdin=`**`None`**`, stdout=`**`None`**`, stderr=`**`None`**`, preexec_fn=`**`None`**`, close_fds=`**`False`**`, shell=`**`False`**`, cwd=`**`None`**`, env=`**`None`**`, text=`**`None`**`, universal_newlines=`**`False`**`, startupinfo=`**`None`**`, creationflags=0)`

`Popen` starts a subprocess to run a distinct program, and creates and returns an object *`p`*, representing that subprocess. The *`args`* mandatory argument and the many optional named arguments control all details of how the subprocess is to run.

When any exception occurs during the subprocess creation (before the distinct program starts), `Popen` re-raises that exception in the calling process with the addition of an attribute named `child_traceback`, which is the Python traceback object for the subprocess. Such an exception would normally be an instance of `OSError` (or possibly `TypeError` or `ValueError` to indicate that you've passed to `Popen` an argument that's invalid in type or value).

SUBPROCESS.RUN() IS A CONVENIENCE WRAPPER FUNCTION FOR POPEN

The subprocess module includes the `run` function that encapsulates a `Popen` instance and executes the most common processing flow on it. `run` accepts the same arguments as `Popen`'s constructor, runs the given command, waits for completion or timeout, and returns a `CompletedProcess` instance with attributes for the return code and stdout and `stderr` contents.

If the output of the command needs to be captured, the most common argument values would be to set the `capture_output` and `text` arguments to **`True`**.

---

## What to run, and how

*`args`* is a sequence of strings: the first item is the path to the program to execute, and the following items, if any, are arguments to pass to the program (*`args`* can also be just a string, when you don't need to pass arguments). `executable`, when not **`None`**, overrides *`args`* in determining which program to execute. When `shell` is **`True`**, `executable` specifies which shell to use to run the subprocess; when `shell` is **`True`** and `executable` is **`None`**, the shell used is */bin/sh* on Unix-like systems (on Windows, it's `os.environ['COMSPEC']`).

## Subprocess files

`stdin`, `stdout`, and `stderr` specify the subprocess's standard input, output, and error files, respectively. Each may be `PIPE`, which creates a new pipe to/from the subprocess; **`None`**, meaning that the subprocess is to use the same file as this ("parent") process; or a file object (or file descriptor) that's already suitably open (for reading, for standard input; for writing, for standard output and standard error). `stderr` may also be `subprocess.STDOUT`, meaning that the subprocess's standard error must use the same file as its standard output.[4] When `capture_output` is true, you can not specify `stdout`, nor `stderr`: rather, behavior is just as if each was specified as `PIPE`. `bufsize` controls the buffering of these files (unless they're already open), with the same semantics as the same argument to the `open` function covered in **"Creating a File Object with open"** (the default, `0`, means "unbuffered"). When `text` (or its synonym

`universal_newlines`, provided for backward compatibility) is true, `stdout` and `stderr` (unless they are already open) are opened as text files; otherwise, they're opened as binary files. When `close_fds` is true, all other files (apart from standard input, output, and error) are closed in the subprocess before the subprocess's program or shell executes.

### Other, advanced arguments

When `preexec_fn` is not **`None`**, it must be a function or other callable object, and it gets called in the subprocess before the subprocess's program or shell is executed (only on Unix-like systems, where the call happens after `fork` and before `exec`).

When `cwd` is not **`None`**, it must be a string that gives the full path to an existing directory; the current directory gets changed to `cwd` in the subprocess before the subprocess's program or shell executes.

When `env` is not **`None`**, it must be a mapping with strings as both keys and values, and fully defines the environment for the new process; otherwise, the new process's environment is a copy of the environment currently active in the parent process.

`startupinfo` and `creationflags` are Windows-only arguments passed to the `CreateProcess` Win32 API call used to create the subprocess, for Windows-specific purposes (we do not cover them further in this book, which focuses almost exclusively on cross-platform uses of Python).

### Attributes of subprocess.Popen instances

An instance *p* of the class `Popen` supplies the attributes listed in **Table 15-17**.

Table 15-17. Attributes of an instance *p* of class `Popen`

| | |
|---|---|
| `args` | `Popen`'s *`args`* argument (string or sequence of strings). |
| `pid` | The process ID of the subprocess. |

| | |
|---|---|
| `returncode` | `None` to indicate that the subprocess has not yet exited; otherwise, an integer: `0` for successful termination, `>0` for termination with an error code, or `<0` if the subprocess was killed by a signal. |
| `stderr`, `stdin`, `stdout` | When the corresponding argument to `Popen` was `subprocess.PIPE`, each of these attributes is a file object wrapping the corresponding pipe; otherwise, each of these attributes is **`None`**. Use the `communicate` method of *`p`*, rather than reading and writing to/from these file objects, to avoid possible deadlocks. |

## Methods of subprocess.Popen instances

An instance *`p`* of the class `Popen` supplies the methods listed in **Table 15-18**.

Table 15-18. Methods of an instance *`p`* of class `Popen`

| | |
|---|---|
| `communicate` | *`p`*`.communicate(input=`**`None`**`, timeout=`**`None`**`)`<br>Sends the string `input` as the subprocess's standard input (when `input` is not **`None`**), then reads the subprocess's standard output and error files into in-memory strings *`so`* and *`se`* until both files are finished, and finally waits for the subprocess to terminate and returns the pair (two-item tuple) `(`*`so`*`, `*`se`*`)`. |
| `poll` | *`p`*`.poll()`<br>Checks if the subprocess has terminated; returns *`p`*`.returncode` if it has; otherwise, returns **`None`**. |

| | |
|---|---|
| `wait` | `p.wait(timeout=None)`<br>Waits for the subprocess to terminate, then returns `p.returncode`. Should the subprocess not terminate within `timeout` seconds, raises `TimeoutExpired`. |

## Running Other Programs with the os Module

The best way for your program to run other processes is usually with the `subprocess` module, covered in the previous section. However, the `os` module (introduced in **Chapter 11**) also offers several lower-level ways to do this, which, in some cases, may be simpler to use.

The simplest way to run another program is through the function `os.system`, although this offers no way to *control* the external program. The `os` module also provides a number of functions whose names start with `exec`. These functions offer fine-grained control. A program run by one of the `exec` functions replaces the current program (i.e., the Python interpreter) in the same process. In practice, therefore, you use the `exec` functions mostly on platforms that let a process duplicate itself using `fork` (i.e., Unix-like platforms). `os` functions whose names start with `spawn` and `popen` offer intermediate simplicity and power: they are cross-platform and not quite as simple as `system`, but simple enough for many purposes.

The `exec` and `spawn` functions run a given executable file, given the executable file's path, arguments to pass to it, and optionally an environment mapping. The `system` and `popen` functions execute a command, which is a string passed to a new instance of the platform's default shell (typically */bin/sh* on Unix, *cmd.exe* on Windows). A *command* is a more general concept than an *executable file*, as it can include shell functionality (pipes, redirection, and built-in shell commands) using the shell syntax specific to the current platform.

`os` provides the functions listed in **Table 15-19**.

Table 15-19. Functions of the os module related to processes

| | |
|---|---|
| `execl`, | `execl(`*`path`*`, *`*`args`*`)`, |
| `execle`, | `execle(`*`path`*`, *`*`args`*`)`, |
| `execlp`, | `execlp(`*`path`*`,*`*`args`*`)`, |
| `execv`, | `execv(`*`path`*`, `*`args`*`)`, |
| `execve`, | `execve(`*`path`*`, `*`args`*`, `*`env`*`)`, |
| `execvp`, | `execvp(`*`path`*`, `*`args`*`)`, |
| `execvpe` | `execvpe(`*`path`*`, `*`args`*`, `*`env`*`)` |

Run the executable file (program) indicated by string *`path`*, replacing the current program (i.e., the Python interpreter) the current process. The distinctions encoded in the functio names (after the prefix `exec`) control three aspects of how tl new program is found and run:

- Does *`path`* have to be a complete path to the program's executable file, or can the function accept a name as the *`path`* argument and search for the executable in several directories, as operating system shells do? `execlp`, `execvp`, and `execvpe` can accept a *`path`* argument that is just a filename rather than a complete path. In this case, the functions search for an executable file of that name in the directories listed in `os.environ['PATH']`. The other functions require *`pat`* to be a complete path to the executable file.
- Does the function accept arguments for the new program as a single sequence argument *`args`*, or as separate arguments to the function? Functions whose names start with `execv` take a single argument *`args`* that is the sequence of arguments to use for the new program. Functions whose names start with `execl` tak the new program's arguments as separate arguments (`execle`, in particular, uses its last argument as the environment for the new program).
- Does the function accept the new program's environment as an explicit mapping argument *`env`*, or implicitly use `os.environ`? `execle`, `execve`, and `execvpe` take

| | |
|---|---|
| | an argument *env* that is a mapping to use as the new program's environment (keys and values must be strings), while the other functions use `os.environ` for this purpose.<br>Each `exec` function uses the first item in *args* as the name under which the new program is told it's runnir (for example, `argv[0]` in a C program's `main`); only `args[1:]` are arguments proper to the new program. |
| popen | `popen(`*cmd*`, mode='r', buffering=-1)`<br>Runs the string command *cmd* in a new process *P* and returr a file-like object *f* that wraps a pipe to *P*'s standard input or from *P*'s standard output (depending on `mode`); *f* uses text streams in both directions rather than raw bytes. `mode` and `buffering` have the same meaning as for Python's `open` function, covered in **"Creating a File Object with open"**.<br>When `mode` is `'r'` (the default), *f* is read-only and wraps *P*'s standard output. When `mode` is `'w'`, *f* is write-only and wrap *P*'s standard input.<br>The key difference of *f* from other file-like objects is the behavior of method *f*`.close`. *f*`.close` waits for *P* to termina and returns **None**, as `close` methods of file-like objects normally do, when *P*'s termination is successful. However, i the operating system associates an integer error code *c* with *P*'s termination, indicating that *P*'s termination was unsuccessful, *f*`.close` returns *c*. On Windows systems, `c` is a signed integer return code from the child process. |
| spawnv, spawnve | `spawnv(`*mode*`, `*path*`, `*args*`)`,<br>`spawnve(`*mode*`, `*path*`, `*args*`, `*env*`)`<br>These functions run the program indicated by *path* in a new process *P*, with the arguments passed as sequence *args*. `spawnve` uses mapping *env* as *P*'s environment (both keys an values must be strings), while `spawnv` uses `os.environ` for this purpose. On Unix-like platforms only, there are other |

variations of `os.spawn`, corresponding to variations of `os.exec`, but `spawnv` and `spawnve` are the only two that also exist on Windows.

*mode* must be one of two attributes supplied by the `os` module: `os.P_WAIT` indicates that the calling process waits until the new process terminates, while `os.P_NOWAIT` indicates that the calling process continues executing simultaneously with the new process. When *mode* is `os.P_WAIT`, the function returns the termination code *c* of *P*: indicates successful termination, *c* < `0` indicates *P* was kille by a *signal*, and *c* > `0` indicates normal but unsuccessful termination. When *mode* is `os.P_NOWAIT`, the function returr *P*'s process ID (or, on Windows, *P*'s process handle). There is no cross-platform way to use *P*'s ID or handle; platform-specific ways (not covered further in this book) include `os.waitpid` on Unix-like platforms, and third-party extensic package **pywin32** on Windows.

For example, suppose you want your interactive program tc give the user a chance to edit a text file that your program is about to read and use. You must have previously determine the full path to the user's favorite text editor, such as *c:||windows||notepad.exe* on Windows or */usr/bin/vim* on a Unix-like platform. Say that this path string is bound to the variable `editor`, and the path of the text file you want to let the user edit is bound to `textfile`:

```
import os
os.spawnv(os.P_WAIT, editor, (editor, textfile))
```

The first item of the argument *args* is passed to the program being spawned as "the name under which the program is being invoked." Most programs don't look at this, so you car usually place just about any string here. Just in case the editor program does look at this special first argument (som versions of Vim, for example, do), passing the same string

| | |
|---|---|
| | editor that is used as the second argument to `os.spawnv` is t<br>simplest and most effective approach. |
| `system` | `system(`*`cmd`*`)`<br>Runs the string command *`cmd`* in a new process and returns<br>when the new process terminates successfully. When the ne<br>process terminates unsuccessfully, `system` returns an intege<br>error code not equal to `0`. (Exactly what error codes may be<br>returned depends on the command you're running: there's<br>no widely accepted standard for this.) |

# The mmap Module

The `mmap` module supplies memory-mapped file objects. An `mmap` object behaves similarly to a bytestring, so you can often pass an `mmap` object where a bytestring is expected. However, there are differences:

- An `mmap` object does not supply the methods of a string object.
- An `mmap` object is mutable, like a `bytearray`, while `bytes` objects are immutable.
- An `mmap` object also corresponds to an open file, and behaves polymorphically to a Python file object (as covered in **"File-Like Objects and Polymorphism"**).

An `mmap` object *`m`* can be indexed or sliced, yielding bytestrings. Since *`m`* is mutable, you can also assign to an indexing or slicing of *`m`*. However, when you assign to a slice of *`m`*, the righthand side of the assignment statement must be a bytestring of exactly the same length as the slice you're assigning to. Therefore, many of the useful tricks available with list slice assignment (covered in **"Modifying a list"**) do not apply to `mmap` slice assignment.

The `mmap` module supplies a factory function, slightly different on Unix-like systems and on Windows:

| | |
|---|---|
| `mmap` | *Windows:* `mmap(`*`filedesc`*`, `*`length`*`, tagname='', access=`**`None`**`, offset=`**`None`**`)`<br>*Unix:* `mmap(`*`filedesc`*`, `*`length`*`, flags=MAP_SHARED, prot=PROT_READ\|PROT_WRITE, access=`**`None`**`, offset=0)`<br>Creates and returns an `mmap` object *`m`* that maps into memory the first *`length`* bytes of the file indicated by file descriptor *`filedesc`*. *`filedesc`* must be a file descriptor opened for both reading and writing, except, on Unix-like platforms, when the argument `prot` requests only reading or only writing. (File descriptors are covered in **"File descriptor operations"**.) To get an `mmap` object *`m`* for a Python file object *`f`*, use *`m`*`=mmap.mmap(`*`f`*`.fileno(), `*`length`*`)`. *`filedesc`* can be `-1` to map anonymous memory. On Windows, all memory mappings are readable and writable, and shared among processes, so all processes with a memory mapping on a file can see changes made by other processes. On Windows only, you can pass a string `tagname` to give an explicit *tag name* for the memory mapping. This tag name lets you have several separate memory mappings on the same file, but this is rarely necessary. Calling `mmap` with only two arguments has the advantage of keeping your code portable between Windows and Unix-like platforms. |
| `mmap`<br>*(cont.)* | On Unix-like platforms only, you can pass `mmap.MAP_PRIVATE` as `flags` to get a mapping that is private to your process and copy-on-write. `mmap.MAP_SHARED`, the default, gets a mapping that is shared with other processes so that all processes mapping the file can see changes made by one process (the same as on Windows). You can pass `mmap.PROT_READ` as the `prot` argument to get a mapping that you can only read, not write. Passing `mmap.PROT_WRITE` gets a mapping that you can only write, not read. The default, the bitwise OR `mmap.PROT_READ\|mmap.PROT_WRITE`, gets a mapping you can both read and write. |

| | |
|---|---|
| | You can pass the named argument `access` instead of `flags` and `prot` (it's an error to pass both `access` and either or both of the other two arguments). The value for `access` can be one of `ACCESS_READ` (read-only), `ACCESS_WRITE` (write-through, the default on Windows), or `ACCESS_COPY` (copy-on-write).<br>You can pass the named argument `offset` to start the mapping after the beginning of the file; `offset` must be an `int >= 0`, a multiple of `ALLOCATIONGRANULARITY` (or, on Unix, of `PAGESIZE`). |

## Methods of mmap Objects

An `mmap` object *`m`* supplies the methods detailed in **Table 15-20**.

Table 15-20. Methods of an instance `m` of `mmap`

| | |
|---|---|
| `close` | *`m`*`.close()`<br>Closes *`m`*'s file. |
| `find` | *`m`*`.find(`*`sub`*`, start=0, end=None)`<br>Returns the lowest *`i`* `>= start` such that *`sub`* `==` *`m`*`[`*`i`*`:`*`i`*`+len(`*`sub`*`)]` (and *`i`*`+len(`*`sub`*`)-1 <= end`, when you pass `end`). If no such *`i`* exists, *`m`*`.find` returns `-1`. This is the same behavior as the `find` method of `str`, covered in **Table 9-1**. |
| `flush` | *`m`*`.flush([`*`offset`*`, `*`n`*`])`<br>Ensures that all changes made to *`m`* exist in *`m`*'s file. Until you call *`m`*`.flush`, it's unsure if the file reflects the current state of *`m`*. You can pass a starting byte offset *`offset`* and a byte count *`n`* to limit the flushing effect's guarantee to a slice of *`m`*. Pass both arguments, or neither: it's an error to call *`m`*`.flush` with just one argument. |

| | |
|---|---|
| `move` | *m*`.move(`*dstoff*`, `*srcoff*`, `*n*`)`<br>Like the slice assignment *m*`[`*dstoff*`:`*dstoff*`+`*n*`] = `*m*`[`*srcoff*`:`*srcoff*`+`*n*`]`, but potentially faster. The source and destination slices can overlap. Apart from such potential overlap, `move` does not affect the source slice (i.e., the `move` method *copies* bytes but does not *move* them, despite the method's name). |
| `read` | *m*`.read(`*n*`)`<br>Reads and returns a byte string *s* containing up to *n* bytes starting from *m*'s file pointer, then advances *m*'s file pointer by `len(`*s*`)`. If there are fewer than *n* bytes between *m*'s file pointer and *m*'s length, returns the bytes available. In particular, if *m*'s file pointer is at the end of *m*, returns the empty bytestring `b''`. |
| `read_byte` | *m*`.read_byte()`<br>Returns a byte string of length `1` containing the byte at *m*'s file pointer, then advances *m*'s file pointer by `1`. *m*`.read_byte()` is similar to *m*`.read(1)`. However, if *m*'s file pointer is at the end of *m*, *m*`.read(1)` returns the empty string `b''` and doesn't advance, while *m*`.read_byte()` raises a `ValueError` exception. |
| `readline` | *m*`.readline()`<br>Reads and returns, as a bytestring, one line from *m*'s file, from *m*'s current file pointer up to the next `'\n'`, included (or up to the end of *m* if there is no `'\n'`), then advances *m*'s file pointer to point just past the bytes just read. If *m*'s file pointer is at the end of *m*, `readline` returns the empty string `b''`. |
| `resize` | *m*`.resize(`*n*`)`<br>Changes the length of *m* so that `len(`*m*`)` becomes *n*. Does not affect the size of *m*'s file. *m*'s length and the file's size |

| | |
|---|---|
| | are independent. To set *m*'s length to be equal to the file's size, call `m.resize(m.size())`. If *m*'s length is larger than the file's size, *m* is padded with null bytes (`\x00`). |
| `rfind` | `rfind(sub, start=0, end=None)`<br>Returns the highest `i >= start` such that *sub* == `m[i:i+len(sub)]` (and `i+len(sub)-1 <= end`, when you pass `end`). If no such `i` exists, `m.rfind` returns `-1`. This is the same as the `rfind` method of string objects, covered in **Table 9-1**. |
| `seek` | `m.seek(pos, how=0)`<br>Sets *m*'s file pointer to the integer byte offset *pos*, relative to the position indicated by `how`:<br>`0` *or* `os.SEEK_SET`<br>Offset is relative to start of *m*<br>`1` *or* `os.SEEK_CUR`<br>Offset is relative to *m*'s current file pointer<br>`2` *or* `os.SEEK_END`<br>Offset is relative to end of *m*<br>A `seek` trying to set *m*'s file pointer to a negative offset, or to an offset beyond *m*'s length, raises a `ValueError` exception. |
| `size` | `m.size()`<br>Returns the length (number of bytes) of *m*'s file (not the length of *m* itself). To get the length of *m*, use `len(m)`. |
| `tell` | `m.tell()`<br>Returns the current position of *m*'s file pointer, a byte offset within *m*'s file. |
| `write` | `m.write(b)`<br>Writes the bytes in bytestring *b* into *m* at the current |

position of *m*'s file pointer, overwriting the bytes that were there, and then advances *m*'s file pointer by `len(`*b*`)`. If there aren't at least `len(`*b*`)` bytes between *m*'s file pointer and the length of *m*, `write` raises a `ValueError` exception.

`write_byte` *m*`.write_byte(`*byte*`)`
Writes *byte*, which must be an `int`, into mapping *m* at the current position of *m*'s file pointer, overwriting the byte that was there, and then advances *m*'s file pointer by 1. *m*`.write_byte(`*x*`)` is similar to *m*`.write(`*x*`.to_bytes(1, 'little'))`. However, if *m*'s file pointer is at the end of *m*, *m*`.write_byte(`*x*`)` silently does nothing, while *m*`.write(`*x*`.to_bytes(1, 'little'))` raises a `ValueError` exception. Note that this is the reverse of the relationship between `read` and `read_byte` at end-of-file: `write` and `read_byte` may raise `ValueError`, while `read` and `write_byte` never do.

## Using mmap Objects for IPC

Processes communicate using `mmap` pretty much the same way they communicate using files: one process writes data, and another process later reads the same data back. Since an `mmap` object has an underlying file, you can have some processes doing I/O on the file (as covered in **"The io Module"**), while others use `mmap` on the same file. Choose between `mmap` and I/O on file objects on the basis of convenience: functionality is the same, performance is roughly equivalent. For example, here is a simple program that repeatedly uses file I/O to make the contents of a file equal to the last line interactively typed by the user:

```
fileob = open('xxx','wb')
while True:
    data = input('Enter some text:')
```

```
        fileob.seek(0)
        fileob.write(data.encode())
        fileob.truncate()
        fileob.flush()
```

And here is another simple program that, when run in the same directory as the former, uses `mmap` (and the `time.sleep` function, covered in Table 13-2) to check every second for changes to the file and print out the file's new contents, if there have been any changes:

```
import mmap, os, time
mx = mmap.mmap(os.open('xxx', os.O_RDWR), 1)
last = None
while True:
    mx.resize(mx.size())
    data = mx[:]
    if data != last:
        print(data)
        last = data
    time.sleep(1)
```

1 The best introductory work on async programming we have come across, though sadly now dated (as the async approach in Python keeps improving), is ***Using Asyncio in Python***, by Caleb Hattingh (O'Reilly). We recommend you also study **Brad Solomon's Asyncio walkthrough** on Real Python.

2 The online docs include an especially helpful **"Programming Guidelines" section** that lists a number of additional practical recommendations when using the `multiprocessing` module.

3 A race condition is a situation in which the relative timings of different events, which are usually unpredictable, can affect the outcome of a computation…never a good thing!

4 Just like `2>&1` would specify in a Unix-y shell command line.