

CONCURRENT EXECUTION

The modules described in this chapter provide support for concurrent execution of code. The appropriate choice of tool will depend on the task to be executed (CPU bound vs IO bound) and preferred style of development (event driven cooperative multitasking vs preemptive multitasking). Here's an overview:

17.1 `threading` — Thread-based parallelism

Source code: [Lib/threading.py](#)

This module constructs higher-level threading interfaces on top of the lower level `_thread` module. See also the `queue` module.

Changed in version 3.7: This module used to be optional, it is now always available.

Note: While they are not listed below, the `camelCase` names used for some methods and functions in this module in the Python 2.x series are still supported by this module.

This module defines the following functions:

`threading.active_count()`

Return the number of `Thread` objects currently alive. The returned count is equal to the length of the list returned by `enumerate()`.

`threading.current_thread()`

Return the current `Thread` object, corresponding to the caller's thread of control. If the caller's thread of control was not created through the `threading` module, a dummy thread object with limited functionality is returned.

`threading.get_ident()`

Return the 'thread identifier' of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

New in version 3.3.

`threading.enumerate()`

Return a list of all `Thread` objects currently alive. The list includes daemon threads, dummy thread objects created by `current_thread()`, and the main thread. It excludes terminated threads and threads that have not yet been started.

`threading.main_thread()`

Return the main `Thread` object. In normal conditions, the main thread is the thread from which the Python interpreter was started.

New in version 3.4.

`threading.settrace(func)`

Set a trace function for all threads started from the `threading` module. The *func* will be passed to `sys.settrace()` for each thread, before its `run()` method is called.

`threading.setprofile(func)`

Set a profile function for all threads started from the `threading` module. The *func* will be passed to `sys.setprofile()` for each thread, before its `run()` method is called.

`threading.stack_size([size])`

Return the thread stack size used when creating new threads. The optional *size* argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32 KiB). If *size* is not specified, 0 is used. If changing the thread stack size is unsupported, a `RuntimeError` is raised. If the specified stack size is invalid, a `ValueError` is raised and the stack size is unmodified. 32 KiB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32 KiB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4 KiB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information).

Availability: Windows, systems with POSIX threads.

This module also defines the following constant:

`threading.TIMEOUT_MAX`

The maximum value allowed for the *timeout* parameter of blocking functions (`Lock.acquire()`, `RLock.acquire()`, `Condition.wait()`, etc.). Specifying a timeout greater than this value will raise an `OverflowError`.

New in version 3.2.

This module defines a number of classes, which are detailed in the sections below.

The design of this module is loosely based on Java's threading model. However, where Java makes locks and condition variables basic behavior of every object, they are separate objects in Python. Python's `Thread` class supports a subset of the behavior of Java's `Thread` class; currently, there are no priorities, no thread groups, and threads cannot be destroyed, stopped, suspended, resumed, or interrupted. The static methods of Java's `Thread` class, when implemented, are mapped to module-level functions.

All of the methods described below are executed atomically.

17.1.1 Thread-Local Data

Thread-local data is data whose values are thread specific. To manage thread-local data, just create an instance of `local` (or a subclass) and store attributes on it:

```
mydata = threading.local()
mydata.x = 1
```

The instance's values will be different for separate threads.

`class threading.local`

A class that represents thread-local data.

For more details and extensive examples, see the documentation string of the `_threading_local` module.

17.1.2 Thread Objects

The `Thread` class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the `run()` method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, *only* override the `__init__()` and `run()` methods of this class.

Once a thread object is created, its activity must be started by calling the thread's `start()` method. This invokes the `run()` method in a separate thread of control.

Once the thread's activity is started, the thread is considered 'alive'. It stops being alive when its `run()` method terminates – either normally, or by raising an unhandled exception. The `is_alive()` method tests whether the thread is alive.

Other threads can call a thread's `join()` method. This blocks the calling thread until the thread whose `join()` method is called is terminated.

A thread has a name. The name can be passed to the constructor, and read or changed through the `name` attribute.

A thread can be flagged as a “daemon thread”. The significance of this flag is that the entire Python program exits when only daemon threads are left. The initial value is inherited from the creating thread. The flag can be set through the `daemon` property or the `daemon` constructor argument.

Note: Daemon threads are abruptly stopped at shutdown. Their resources (such as open files, database transactions, etc.) may not be released properly. If you want your threads to stop gracefully, make them non-daemonic and use a suitable signalling mechanism such as an [Event](#).

There is a “main thread” object; this corresponds to the initial thread of control in the Python program. It is not a daemon thread.

There is the possibility that “dummy thread objects” are created. These are thread objects corresponding to “alien threads”, which are threads of control started outside the threading module, such as directly from C code. Dummy thread objects have limited functionality; they are always considered alive and daemonic, and cannot be `join()`ed. They are never deleted, since it is impossible to detect the termination of alien threads.

```
class threading.Thread(group=None, target=None, name=None, args=(), kwargs={}, *, dae-
                     mon=None)
```

This constructor should always be called with keyword arguments. Arguments are:

`group` should be `None`; reserved for future extension when a `ThreadGroup` class is implemented.

`target` is the callable object to be invoked by the `run()` method. Defaults to `None`, meaning nothing is called.

`name` is the thread name. By default, a unique name is constructed of the form “Thread-*N*” where *N* is a small decimal number.

`args` is the argument tuple for the target invocation. Defaults to `()`.

`kwargs` is a dictionary of keyword arguments for the target invocation. Defaults to `{}`.

If not `None`, `daemon` explicitly sets whether the thread is daemonic. If `None` (the default), the daemonic property is inherited from the current thread.

If the subclass overrides the constructor, it must make sure to invoke the base class constructor (`Thread.__init__()`) before doing anything else to the thread.

Changed in version 3.3: Added the `daemon` argument.

```
start()
```

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's `run()` method to be invoked in a separate thread of control.

This method will raise a `RuntimeError` if called more than once on the same thread object.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object's constructor as the `target` argument, if any, with sequential and keyword arguments taken from the `args` and `kwargs` arguments, respectively.

join(timeout=None)

Wait until the thread terminates. This blocks the calling thread until the thread whose `join()` method is called terminates – either normally or through an unhandled exception – or until the optional timeout occurs.

When the `timeout` argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As `join()` always returns `None`, you must call `is_alive()` after `join()` to decide whether a timeout happened – if the thread is still alive, the `join()` call timed out.

When the `timeout` argument is not present or `None`, the operation will block until the thread terminates.

A thread can be `join()`ed many times.

`join()` raises a `RuntimeError` if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to `join()` a thread before it has been started and attempts to do so raise the same exception.

name

A string used for identification purposes only. It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

getName()**setName()**

Old getter/setter API for `name`; use it directly as a property instead.

ident

The 'thread identifier' of this thread or `None` if the thread has not been started. This is a nonzero integer. See the `get_ident()` function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

is_alive()

Return whether the thread is alive.

This method returns `True` just before the `run()` method starts until just after the `run()` method terminates. The module function `enumerate()` returns a list of all alive threads.

daemon

A boolean value indicating whether this thread is a daemon thread (`True`) or not (`False`). This must be set before `start()` is called, otherwise `RuntimeError` is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to `daemon = False`.

The entire Python program exits when no alive non-daemon threads are left.

isDaemon()**setDaemon()**

Old getter/setter API for `daemon`; use it directly as a property instead.

CPython implementation detail: In CPython, due to the *Global Interpreter Lock*, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this

limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use *multiprocessing* or *concurrent.futures.ProcessPoolExecutor*. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

17.1.3 Lock Objects

A primitive lock is a synchronization primitive that is not owned by a particular thread when locked. In Python, it is currently the lowest level synchronization primitive available, implemented directly by the *_thread* extension module.

A primitive lock is in one of two states, “locked” or “unlocked”. It is created in the unlocked state. It has two basic methods, *acquire()* and *release()*. When the state is unlocked, *acquire()* changes the state to locked and returns immediately. When the state is locked, *acquire()* blocks until a call to *release()* in another thread changes it to unlocked, then the *acquire()* call resets it to locked and returns. The *release()* method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, a *RuntimeError* will be raised.

Locks also support the *context management protocol*.

When more than one thread is blocked in *acquire()* waiting for the state to turn to unlocked, only one thread proceeds when a *release()* call resets the state to unlocked; which one of the waiting threads proceeds is not defined, and may vary across implementations.

All methods are executed atomically.

class *threading.Lock*

The class implementing primitive lock objects. Once a thread has acquired a lock, subsequent attempts to acquire it block, until it is released; any thread may release it.

Note that *Lock* is actually a factory function which returns an instance of the most efficient version of the concrete *Lock* class that is supported by the platform.

acquire(*blocking=True, timeout=-1*)

Acquire a lock, blocking or non-blocking.

When invoked with the *blocking* argument set to **True** (the default), block until the lock is unlocked, then set it to locked and return **True**.

When invoked with the *blocking* argument set to **False**, do not block. If a call with *blocking* set to **True** would block, return **False** immediately; otherwise, set the lock to locked and return **True**.

When invoked with the floating-point *timeout* argument set to a positive value, block for at most the number of seconds specified by *timeout* and as long as the lock cannot be acquired. A *timeout* argument of **-1** specifies an unbounded wait. It is forbidden to specify a *timeout* when *blocking* is false.

The return value is **True** if the lock is acquired successfully, **False** if not (for example if the *timeout* expired).

Changed in version 3.2: The *timeout* parameter is new.

Changed in version 3.2: Lock acquisition can now be interrupted by signals on POSIX if the underlying threading implementation supports it.

release()

Release a lock. This can be called from any thread, not only the thread which has acquired the lock.

When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

When invoked on an unlocked lock, a *RuntimeError* is raised.

There is no return value.

17.1.4 RLock Objects

A reentrant lock is a synchronization primitive that may be acquired multiple times by the same thread. Internally, it uses the concepts of “owning thread” and “recursion level” in addition to the locked/unlocked state used by primitive locks. In the locked state, some thread owns the lock; in the unlocked state, no thread owns it.

To lock the lock, a thread calls its `acquire()` method; this returns once the thread owns the lock. To unlock the lock, a thread calls its `release()` method. `acquire()/release()` call pairs may be nested; only the final `release()` (the `release()` of the outermost pair) resets the lock to unlocked and allows another thread blocked in `acquire()` to proceed.

Reentrant locks also support the *context management protocol*.

class `threading.RLock`

This class implements reentrant lock objects. A reentrant lock must be released by the thread that acquired it. Once a thread has acquired a reentrant lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

Note that `RLock` is actually a factory function which returns an instance of the most efficient version of the concrete `RLock` class that is supported by the platform.

acquire(*blocking=True, timeout=-1*)

Acquire a lock, blocking or non-blocking.

When invoked without arguments: if this thread already owns the lock, increment the recursion level by one, and return immediately. Otherwise, if another thread owns the lock, block until the lock is unlocked. Once the lock is unlocked (not owned by any thread), then grab ownership, set the recursion level to one, and return. If more than one thread is blocked waiting until the lock is unlocked, only one at a time will be able to grab ownership of the lock. There is no return value in this case.

When invoked with the *blocking* argument set to true, do the same thing as when called without arguments, and return true.

When invoked with the *blocking* argument set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

When invoked with the floating-point *timeout* argument set to a positive value, block for at most the number of seconds specified by *timeout* and as long as the lock cannot be acquired. Return true if the lock has been acquired, false if the timeout has elapsed.

Changed in version 3.2: The *timeout* parameter is new.

release()

Release a lock, decrementing the recursion level. If after the decrement it is zero, reset the lock to unlocked (not owned by any thread), and if any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling thread.

Only call this method when the calling thread owns the lock. A `RuntimeError` is raised if this method is called when the lock is unlocked.

There is no return value.

17.1.5 Condition Objects

A condition variable is always associated with some kind of lock; this can be passed in or one will be created by default. Passing one in is useful when several condition variables must share the same lock. The lock is part of the condition object: you don't have to track it separately.

A condition variable obeys the *context management protocol*: using the `with` statement acquires the associated lock for the duration of the enclosed block. The `acquire()` and `release()` methods also call the corresponding methods of the associated lock.

Other methods must be called with the associated lock held. The `wait()` method releases the lock, and then blocks until another thread awakens it by calling `notify()` or `notify_all()`. Once awakened, `wait()` re-acquires the lock and returns. It is also possible to specify a timeout.

The `notify()` method wakes up one of the threads waiting for the condition variable, if any are waiting. The `notify_all()` method wakes up all threads waiting for the condition variable.

Note: the `notify()` and `notify_all()` methods don't release the lock; this means that the thread or threads awakened will not return from their `wait()` call immediately, but only when the thread that called `notify()` or `notify_all()` finally relinquishes ownership of the lock.

The typical programming style using condition variables uses the lock to synchronize access to some shared state; threads that are interested in a particular change of state call `wait()` repeatedly until they see the desired state, while threads that modify the state call `notify()` or `notify_all()` when they change the state in such a way that it could possibly be a desired state for one of the waiters. For example, the following code is a generic producer-consumer situation with unlimited buffer capacity:

```
# Consume one item
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# Produce one item
with cv:
    make_an_item_available()
    cv.notify()
```

The `while` loop checking for the application's condition is necessary because `wait()` can return after an arbitrary long time, and the condition which prompted the `notify()` call may no longer hold true. This is inherent to multi-threaded programming. The `wait_for()` method can be used to automate the condition checking, and eases the computation of timeouts:

```
# Consume an item
with cv:
    cv.wait_for(an_item_is_available)
    get_an_available_item()
```

To choose between `notify()` and `notify_all()`, consider whether one state change can be interesting for only one or several waiting threads. E.g. in a typical producer-consumer situation, adding one item to the buffer only needs to wake up one consumer thread.

class `threading.Condition(lock=None)`

This class implements condition variable objects. A condition variable allows one or more threads to wait until they are notified by another thread.

If the `lock` argument is given and not `None`, it must be a `Lock` or `RLock` object, and it is used as the underlying lock. Otherwise, a new `RLock` object is created and used as the underlying lock.

Changed in version 3.3: changed from a factory function to a class.

acquire(*args)

Acquire the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.

release()

Release the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.

wait(timeout=None)

Wait until notified or until a timeout occurs. If the calling thread has not acquired the lock when this method is called, a *RuntimeError* is raised.

This method releases the underlying lock, and then blocks until it is awakened by a *notify()* or *notify_all()* call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.

When the *timeout* argument is present and not *None*, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

When the underlying lock is an *RLock*, it is not released using its *release()* method, since this may not actually unlock the lock when it was acquired multiple times recursively. Instead, an internal interface of the *RLock* class is used, which really unlocks it even when it has been recursively acquired several times. Another internal interface is then used to restore the recursion level when the lock is reacquired.

The return value is *True* unless a given *timeout* expired, in which case it is *False*.

Changed in version 3.2: Previously, the method always returned *None*.

wait_for(predicate, timeout=None)

Wait until a condition evaluates to true. *predicate* should be a callable which result will be interpreted as a boolean value. A *timeout* may be provided giving the maximum time to wait.

This utility method may call *wait()* repeatedly until the predicate is satisfied, or until a timeout occurs. The return value is the last return value of the predicate and will evaluate to *False* if the method timed out.

Ignoring the timeout feature, calling this method is roughly equivalent to writing:

```
while not predicate():
    cv.wait()
```

Therefore, the same rules apply as with *wait()*: The lock must be held when called and is re-acquired on return. The predicate is evaluated with the lock held.

New in version 3.2.

notify(n=1)

By default, wake up one thread waiting on this condition, if any. If the calling thread has not acquired the lock when this method is called, a *RuntimeError* is raised.

This method wakes up at most *n* of the threads waiting for the condition variable; it is a no-op if no threads are waiting.

The current implementation wakes up exactly *n* threads, if at least *n* threads are waiting. However, it's not safe to rely on this behavior. A future, optimized implementation may occasionally wake up more than *n* threads.

Note: an awakened thread does not actually return from its *wait()* call until it can reacquire the lock. Since *notify()* does not release the lock, its caller should.

notify_all()

Wake up all threads waiting on this condition. This method acts like `notify()`, but wakes up all waiting threads instead of one. If the calling thread has not acquired the lock when this method is called, a `RuntimeError` is raised.

17.1.6 Semaphore Objects

This is one of the oldest synchronization primitives in the history of computer science, invented by the early Dutch computer scientist Edsger W. Dijkstra (he used the names `P()` and `V()` instead of `acquire()` and `release()`).

A semaphore manages an internal counter which is decremented by each `acquire()` call and incremented by each `release()` call. The counter can never go below zero; when `acquire()` finds that it is zero, it blocks, waiting until some other thread calls `release()`.

Semaphores also support the *context management protocol*.

class threading.Semaphore(value=1)

This class implements semaphore objects. A semaphore manages an atomic counter representing the number of `release()` calls minus the number of `acquire()` calls, plus an initial value. The `acquire()` method blocks if necessary until it can return without making the counter negative. If not given, `value` defaults to 1.

The optional argument gives the initial `value` for the internal counter; it defaults to 1. If the `value` given is less than 0, `ValueError` is raised.

Changed in version 3.3: changed from a factory function to a class.

acquire(blocking=True, timeout=None)

Acquire a semaphore.

When invoked without arguments:

- If the internal counter is larger than zero on entry, decrement it by one and return true immediately.
- If the internal counter is zero on entry, block until awoken by a call to `release()`. Once awoken (and the counter is greater than 0), decrement the counter by 1 and return true. Exactly one thread will be awoken by each call to `release()`. The order in which threads are awoken should not be relied on.

When invoked with `blocking` set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

When invoked with a `timeout` other than `None`, it will block for at most `timeout` seconds. If acquire does not complete successfully in that interval, return false. Return true otherwise.

Changed in version 3.2: The `timeout` parameter is new.

release()

Release a semaphore, incrementing the internal counter by one. When it was zero on entry and another thread is waiting for it to become larger than zero again, wake up that thread.

class threading.BoundedSemaphore(value=1)

Class implementing bounded semaphore objects. A bounded semaphore checks to make sure its current value doesn't exceed its initial value. If it does, `ValueError` is raised. In most situations semaphores are used to guard resources with limited capacity. If the semaphore is released too many times it's a sign of a bug. If not given, `value` defaults to 1.

Changed in version 3.3: changed from a factory function to a class.

Semaphore Example

Semaphores are often used to guard resources with limited capacity, for example, a database server. In any situation where the size of the resource is fixed, you should use a bounded semaphore. Before spawning any worker threads, your main thread would initialize the semaphore:

```
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
```

Once spawned, worker threads call the semaphore's `acquire` and `release` methods when they need to connect to the server:

```
with pool_sema:
    conn = connectdb()
    try:
        # ... use connection ...
    finally:
        conn.close()
```

The use of a bounded semaphore reduces the chance that a programming error which causes the semaphore to be released more than it's acquired will go undetected.

17.1.7 Event Objects

This is one of the simplest mechanisms for communication between threads: one thread signals an event and other threads wait for it.

An event object manages an internal flag that can be set to true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true.

`class threading.Event`

Class implementing event objects. An event manages a flag that can be set to true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true. The flag is initially false.

Changed in version 3.3: changed from a factory function to a class.

`is_set()`

Return true if and only if the internal flag is true.

`set()`

Set the internal flag to true. All threads waiting for it to become true are awakened. Threads that call `wait()` once the flag is true will not block at all.

`clear()`

Reset the internal flag to false. Subsequently, threads calling `wait()` will block until `set()` is called to set the internal flag to true again.

`wait(timeout=None)`

Block until the internal flag is true. If the internal flag is true on entry, return immediately. Otherwise, block until another thread calls `set()` to set the flag to true, or until the optional timeout occurs.

When the timeout argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

This method returns `true` if and only if the internal flag has been set to `true`, either before the wait call or after the wait starts, so it will always return `True` except if a timeout is given and the operation times out.

Changed in version 3.1: Previously, the method always returned `None`.

17.1.8 Timer Objects

This class represents an action that should be run only after a certain amount of time has passed — a timer. `Timer` is a subclass of `Thread` and as such also functions as an example of creating custom threads.

Timers are started, as with threads, by calling their `start()` method. The timer can be stopped (before its action has begun) by calling the `cancel()` method. The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user.

For example:

```
def hello():
    print("hello, world")

t = Timer(30.0, hello)
t.start()  # after 30 seconds, "hello, world" will be printed
```

`class threading.Timer(interval, function, args=None, kwargs=None)`

Create a timer that will run *function* with arguments *args* and keyword arguments *kwargs*, after *interval* seconds have passed. If *args* is `None` (the default) then an empty list will be used. If *kwargs* is `None` (the default) then an empty dict will be used.

Changed in version 3.3: changed from a factory function to a class.

`cancel()`

Stop the timer, and cancel the execution of the timer's action. This will only work if the timer is still in its waiting stage.

17.1.9 Barrier Objects

New in version 3.2.

This class provides a simple synchronization primitive for use by a fixed number of threads that need to wait for each other. Each of the threads tries to pass the barrier by calling the `wait()` method and will block until all of the threads have made their `wait()` calls. At this point, the threads are released simultaneously.

The barrier can be reused any number of times for the same number of threads.

As an example, here is a simple way to synchronize a client and server thread:

```
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
        process_server_connection(connection)

def client():
    b.wait()
```

(continues on next page)

(continued from previous page)

```
while True:
    connection = make_connection()
    process_client_connection(connection)
```

class `threading.Barrier`(*parties*, *action=None*, *timeout=None*)

Create a barrier object for *parties* number of threads. An *action*, when provided, is a callable to be called by one of the threads when they are released. *timeout* is the default timeout value if none is specified for the `wait()` method.

wait(*timeout=None*)

Pass the barrier. When all the threads party to the barrier have called this function, they are all released simultaneously. If a *timeout* is provided, it is used in preference to any that was supplied to the class constructor.

The return value is an integer in the range 0 to *parties* - 1, different for each thread. This can be used to select a thread to do some special housekeeping, e.g.:

```
i = barrier.wait()
if i == 0:
    # Only one thread needs to print this
    print("passed the barrier")
```

If an *action* was provided to the constructor, one of the threads will have called it prior to being released. Should this call raise an error, the barrier is put into the broken state.

If the call times out, the barrier is put into the broken state.

This method may raise a `BrokenBarrierError` exception if the barrier is broken or reset while a thread is waiting.

reset()

Return the barrier to the default, empty state. Any threads waiting on it will receive the `BrokenBarrierError` exception.

Note that using this function may can require some external synchronization if there are other threads whose state is unknown. If a barrier is broken it may be better to just leave it and create a new one.

abort()

Put the barrier into a broken state. This causes any active or future calls to `wait()` to fail with the `BrokenBarrierError`. Use this for example if one of the needs to abort, to avoid deadlocking the application.

It may be preferable to simply create the barrier with a sensible *timeout* value to automatically guard against one of the threads going awry.

parties

The number of threads required to pass the barrier.

n_waiting

The number of threads currently waiting in the barrier.

broken

A boolean that is `True` if the barrier is in the broken state.

exception `threading.BrokenBarrierError`

This exception, a subclass of `RuntimeError`, is raised when the `Barrier` object is reset or broken.

17.1.10 Using locks, conditions, and semaphores in the with statement

All of the objects provided by this module that have `acquire()` and `release()` methods can be used as context managers for a `with` statement. The `acquire()` method will be called when the block is entered, and `release()` will be called when the block is exited. Hence, the following snippet:

```
with some_lock:
    # do something...
```

is equivalent to:

```
some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()
```

Currently, *Lock*, *RLock*, *Condition*, *Semaphore*, and *BoundedSemaphore* objects may be used as `with` statement context managers.

17.2 multiprocessing — Process-based parallelism

Source code: [Lib/multiprocessing/](#)

17.2.1 Introduction

multiprocessing is a package that supports spawning processes using an API similar to the *threading* module. The *multiprocessing* package offers both local and remote concurrency, effectively side-stepping the *Global Interpreter Lock* by using subprocesses instead of threads. Due to this, the *multiprocessing* module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows.

The *multiprocessing* module also introduces APIs which do not have analogs in the *threading* module. A prime example of this is the *Pool* object which offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes (data parallelism). The following example demonstrates the common practice of defining such functions in a module so that child processes can successfully import that module. This basic example of data parallelism using *Pool*,

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

will print to standard output

```
[1, 4, 9]
```

The Process class

In *multiprocessing*, processes are spawned by creating a *Process* object and then calling its *start()* method. *Process* follows the API of *threading.Thread*. A trivial example of a multiprocessing program is

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

To show the individual process IDs involved, here is an expanded example:

```
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

For an explanation of why the `if __name__ == '__main__':` part is necessary, see *Programming guidelines*.

Contexts and start methods

Depending on the platform, *multiprocessing* supports three ways to start a process. These *start methods* are

spawn The parent process starts a fresh python interpreter process. The child process will only inherit those resources necessary to run the process objects *run()* method. In particular, unnecessary file descriptors and handles from the parent process will not be inherited. Starting a process using this method is rather slow compared to using *fork* or *forkserver*.

Available on Unix and Windows. The default on Windows.

fork The parent process uses *os.fork()* to fork the Python interpreter. The child process, when it begins, is effectively identical to the parent process. All resources of the parent are inherited by the child process. Note that safely forking a multithreaded process is problematic.

Available on Unix only. The default on Unix.

forkserver When the program starts and selects the *forkserver* start method, a server process is started. From then on, whenever a new process is needed, the parent process connects to the server and requests that it fork a new process. The fork server process is single threaded so it is safe for it to use `os.fork()`. No unnecessary resources are inherited.

Available on Unix platforms which support passing file descriptors over Unix pipes.

Changed in version 3.4: *spawn* added on all unix platforms, and *forkserver* added for some unix platforms. Child processes no longer inherit all of the parents inheritable handles on Windows.

On Unix using the *spawn* or *forkserver* start methods will also start a *semaphore tracker* process which tracks the unlinked named semaphores created by processes of the program. When all processes have exited the semaphore tracker unlinks any remaining semaphores. Usually there should be none, but if a process was killed by a signal there may be some “leaked” semaphores. (Unlinking the named semaphores is a serious matter since the system allows only a limited number, and they will not be automatically unlinked until the next reboot.)

To select a start method you use the `set_start_method()` in the `if __name__ == '__main__':` clause of the main module. For example:

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    mp.set_start_method('spawn')
    q = mp.Queue()
    p = mp.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

`set_start_method()` should not be used more than once in the program.

Alternatively, you can use `get_context()` to obtain a context object. Context objects have the same API as the multiprocessing module, and allow one to use multiple start methods in the same program.

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    ctx = mp.get_context('spawn')
    q = ctx.Queue()
    p = ctx.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

Note that objects related to one context may not be compatible with processes for a different context. In particular, locks created using the *fork* context cannot be passed to processes started using the *spawn* or *forkserver* start methods.

A library which wants to use a particular start method should probably use `get_context()` to avoid interfering with the choice of the library user.

Warning: The 'spawn' and 'forkserver' start methods cannot currently be used with “frozen” executables (i.e., binaries produced by packages like **PyInstaller** and **cx_Freeze**) on Unix. The 'fork' start method does work.

Exchanging objects between processes

multiprocessing supports two types of communication channel between processes:

Queues

The *Queue* class is a near clone of *queue.Queue*. For example:

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

Queues are thread and process safe.

Pipes

The *Pipe()* function returns a pair of connection objects connected by a pipe which by default is duplex (two-way). For example:

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())    # prints "[42, None, 'hello']"
    p.join()
```

The two connection objects returned by *Pipe()* represent the two ends of the pipe. Each connection object has *send()* and *recv()* methods (among others). Note that data in a pipe may become corrupted if two processes (or threads) try to read from or write to the *same* end of the pipe at the same time. Of course there is no risk of corruption from processes using different ends of the pipe at the same time.

Synchronization between processes

multiprocessing contains equivalents of all the synchronization primitives from *threading*. For instance one can use a lock to ensure that only one process prints to standard output at a time:

```

from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
    finally:
        l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()

```

Without using the lock output from the different processes is liable to get all mixed up.

Sharing state between processes

As mentioned above, when doing concurrent programming it is usually best to avoid using shared state as far as possible. This is particularly true when using multiple processes.

However, if you really do need to use some shared data then *multiprocessing* provides a couple of ways of doing so.

Shared memory

Data can be stored in a shared memory map using *Value* or *Array*. For example, the following code

```

from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])

```

will print

```

3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]

```

The 'd' and 'i' arguments used when creating `num` and `arr` are typecodes of the kind used by the *array* module: 'd' indicates a double precision float and 'i' indicates a signed integer.

These shared objects will be process and thread-safe.

For more flexibility in using shared memory one can use the `multiprocessing.sharedctypes` module which supports the creation of arbitrary ctypes objects allocated from shared memory.

Server process

A manager object returned by `Manager()` controls a server process which holds Python objects and allows other processes to manipulate them using proxies.

A manager returned by `Manager()` will support types `list`, `dict`, `Namespace`, `Lock`, `RLock`, `Semaphore`, `BoundedSemaphore`, `Condition`, `Event`, `Barrier`, `Queue`, `Value` and `Array`. For example,

```
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))

        p = Process(target=f, args=(d, l))
        p.start()
        p.join()

        print(d)
        print(l)
```

will print

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Server process managers are more flexible than using shared memory objects because they can be made to support arbitrary object types. Also, a single manager can be shared by processes on different computers over a network. They are, however, slower than using shared memory.

Using a pool of workers

The `Pool` class represents a pool of worker processes. It has methods which allows tasks to be offloaded to the worker processes in a few different ways.

For example:

```
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x
```

(continues on next page)

(continued from previous page)

```

if __name__ == '__main__':
    # start 4 worker processes
    with Pool(processes=4) as pool:

        # print "[0, 1, 4,..., 81]"
        print(pool.map(f, range(10)))

        # print same numbers in arbitrary order
        for i in pool.imap_unordered(f, range(10)):
            print(i)

        # evaluate "f(20)" asynchronously
        res = pool.apply_async(f, (20,))    # runs in *only* one process
        print(res.get(timeout=1))          # prints "400"

        # evaluate "os.getpid()" asynchronously
        res = pool.apply_async(os.getpid, ()) # runs in *only* one process
        print(res.get(timeout=1))           # prints the PID of that process

        # launching multiple evaluations asynchronously *may* use more processes
        multiple_results = [pool.apply_async(os.getpid, ()) for i in range(4)]
        print([res.get(timeout=1) for res in multiple_results])

        # make a single worker sleep for 10 secs
        res = pool.apply_async(time.sleep, (10,))
        try:
            print(res.get(timeout=1))
        except TimeoutError:
            print("We lacked patience and got a multiprocessing.TimeoutError")

        print("For the moment, the pool remains available for more work")

    # exiting the 'with'-block has stopped the pool
    print("Now the pool is closed and no longer available")

```

Note that the methods of a pool should only ever be used by the process which created it.

Note: Functionality within this package requires that the `__main__` module be importable by the children. This is covered in *Programming guidelines* however it is worth pointing out here. This means that some examples, such as the `multiprocessing.pool.Pool` examples will not work in the interactive interpreter. For example:

```

>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:

```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
```

(If you try this it will actually output three full tracebacks interleaved in a semi-random fashion, and then you may have to stop the master process somehow.)

17.2.2 Reference

The *multiprocessing* package mostly replicates the API of the *threading* module.

Process and exceptions

```
class multiprocessing.Process(group=None, target=None, name=None, args=(), kwargs={}, *,
                             daemon=None)
```

Process objects represent activity that is run in a separate process. The *Process* class has equivalents of all the methods of *threading.Thread*.

The constructor should always be called with keyword arguments. *group* should always be *None*; it exists solely for compatibility with *threading.Thread*. *target* is the callable object to be invoked by the *run()* method. It defaults to *None*, meaning nothing is called. *name* is the process name (see *name* for more details). *args* is the argument tuple for the target invocation. *kwargs* is a dictionary of keyword arguments for the target invocation. If provided, the keyword-only *daemon* argument sets the process *daemon* flag to *True* or *False*. If *None* (the default), this flag will be inherited from the creating process.

By default, no arguments are passed to *target*.

If a subclass overrides the constructor, it must make sure it invokes the base class constructor (*Process.__init__()*) before doing anything else to the process.

Changed in version 3.3: Added the *daemon* argument.

run()

Method representing the process's activity.

You may override this method in a subclass. The standard *run()* method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the *args* and *kwargs* arguments, respectively.

start()

Start the process's activity.

This must be called at most once per process object. It arranges for the object's *run()* method to be invoked in a separate process.

join([timeout])

If the optional argument *timeout* is *None* (the default), the method blocks until the process whose *join()* method is called terminates. If *timeout* is a positive number, it blocks at most *timeout* seconds. Note that the method returns *None* if its process terminates or if the method times out. Check the process's *exitcode* to determine if it terminated.

A process can be joined many times.

A process cannot join itself because this would cause a deadlock. It is an error to attempt to join a process before it has been started.

name

The process's name. The name is a string used for identification purposes only. It has no semantics. Multiple processes may be given the same name.

The initial name is set by the constructor. If no explicit name is provided to the constructor, a name of the form 'Process-N₁:N₂:...:N_k' is constructed, where each N_k is the N-th child of its parent.

is_alive()

Return whether the process is alive.

Roughly, a process object is alive from the moment the `start()` method returns until the child process terminates.

daemon

The process's daemon flag, a Boolean value. This must be set before `start()` is called.

The initial value is inherited from the creating process.

When a process exits, it attempts to terminate all of its daemon child processes.

Note that a daemon process is not allowed to create child processes. Otherwise a daemon process would leave its children orphaned if it gets terminated when its parent process exits. Additionally, these are **not** Unix daemons or services, they are normal processes that will be terminated (and not joined) if non-daemon processes have exited.

In addition to the `threading.Thread` API, `Process` objects also support the following attributes and methods:

pid

Return the process ID. Before the process is spawned, this will be `None`.

exitcode

The child's exit code. This will be `None` if the process has not yet terminated. A negative value `-N` indicates that the child was terminated by signal `N`.

authkey

The process's authentication key (a byte string).

When `multiprocessing` is initialized the main process is assigned a random string using `os.urandom()`.

When a `Process` object is created, it will inherit the authentication key of its parent process, although this may be changed by setting `authkey` to another byte string.

See *Authentication keys*.

sentinel

A numeric handle of a system object which will become "ready" when the process ends.

You can use this value if you want to wait on several events at once using `multiprocessing.connection.wait()`. Otherwise calling `join()` is simpler.

On Windows, this is an OS handle usable with the `WaitForSingleObject` and `WaitForMultipleObjects` family of API calls. On Unix, this is a file descriptor usable with primitives from the `select` module.

New in version 3.3.

terminate()

Terminate the process. On Unix this is done using the `SIGTERM` signal; on Windows `TerminateProcess()` is used. Note that exit handlers and finally clauses, etc., will not be executed.

Note that descendant processes of the process will *not* be terminated – they will simply become orphaned.

Warning: If this method is used when the associated process is using a pipe or queue then the pipe or queue is liable to become corrupted and may become unusable by other process. Similarly, if the process has acquired a lock or semaphore etc. then terminating it is liable to cause other processes to deadlock.

kill()

Same as `terminate()` but using the SIGKILL signal on Unix.

New in version 3.7.

close()

Close the `Process` object, releasing all resources associated with it. `ValueError` is raised if the underlying process is still running. Once `close()` returns successfully, most other methods and attributes of the `Process` object will raise `ValueError`.

New in version 3.7.

Note that the `start()`, `join()`, `is_alive()`, `terminate()` and `exitcode` methods should only be called by the process that created the process object.

Example usage of some of the methods of `Process`:

```
>>> import multiprocessing, time, signal
>>> p = multiprocessing.Process(target=time.sleep, args=(1000,))
>>> print(p, p.is_alive())
<Process(Process-1, initial)> False
>>> p.start()
>>> print(p, p.is_alive())
<Process(Process-1, started)> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print(p, p.is_alive())
<Process(Process-1, stopped[SIGTERM])> False
>>> p.exitcode == -signal.SIGTERM
True
```

exception multiprocessing.ProcessError

The base class of all `multiprocessing` exceptions.

exception multiprocessing.BufferTooShort

Exception raised by `Connection.recv_bytes_into()` when the supplied buffer object is too small for the message read.

If `e` is an instance of `BufferTooShort` then `e.args[0]` will give the message as a byte string.

exception multiprocessing.AuthenticationError

Raised when there is an authentication error.

exception multiprocessing.TimeoutError

Raised by methods with a timeout when the timeout expires.

Pipes and Queues

When using multiple processes, one generally uses message passing for communication between processes and avoids having to use any synchronization primitives like locks.

For passing messages one can use `Pipe()` (for a connection between two processes) or a queue (which allows multiple producers and consumers).

The *Queue*, *SimpleQueue* and *JoinableQueue* types are multi-producer, multi-consumer FIFO queues modelled on the *queue.Queue* class in the standard library. They differ in that *Queue* lacks the *task_done()* and *join()* methods introduced into Python 2.5's *queue.Queue* class.

If you use *JoinableQueue* then you **must** call *JoinableQueue.task_done()* for each task removed from the queue or else the semaphore used to count the number of unfinished tasks may eventually overflow, raising an exception.

Note that one can also create a shared queue by using a manager object – see *Managers*.

Note: *multiprocessing* uses the usual *queue.Empty* and *queue.Full* exceptions to signal a timeout. They are not available in the *multiprocessing* namespace so you need to import them from *queue*.

Note: When an object is put on a queue, the object is pickled and a background thread later flushes the pickled data to an underlying pipe. This has some consequences which are a little surprising, but should not cause any practical difficulties – if they really bother you then you can instead use a queue created with a *manager*.

- (1) After putting an object on an empty queue there may be an infinitesimal delay before the queue's *empty()* method returns *False* and *get_nowait()* can return without raising *queue.Empty*.
 - (2) If multiple processes are enqueueing objects, it is possible for the objects to be received at the other end out-of-order. However, objects enqueued by the same process will always be in the expected order with respect to each other.
-

Warning: If a process is killed using *Process.terminate()* or *os.kill()* while it is trying to use a *Queue*, then the data in the queue is likely to become corrupted. This may cause any other process to get an exception when it tries to use the queue later on.

Warning: As mentioned above, if a child process has put items on a queue (and it has not used *JoinableQueue.cancel_join_thread()*), then that process will not terminate until all buffered items have been flushed to the pipe.

This means that if you try joining that process you may get a deadlock unless you are sure that all items which have been put on the queue have been consumed. Similarly, if the child process is non-daemonic then the parent process may hang on exit when it tries to join all its non-daemonic children.

Note that a queue created using a manager does not have this issue. See *Programming guidelines*.

For an example of the usage of queues for interprocess communication see *Examples*.

multiprocessing.Pipe(*[duplex]*)

Returns a pair (*conn1*, *conn2*) of *Connection* objects representing the ends of a pipe.

If *duplex* is *True* (the default) then the pipe is bidirectional. If *duplex* is *False* then the pipe is unidirectional: *conn1* can only be used for receiving messages and *conn2* can only be used for sending messages.

class multiprocessing.Queue(*[maxsize]*)

Returns a process shared queue implemented using a pipe and a few locks/semaphores. When a process first puts an item on the queue a feeder thread is started which transfers objects from a buffer into the pipe.

The usual `queue.Empty` and `queue.Full` exceptions from the standard library's `queue` module are raised to signal timeouts.

`Queue` implements all the methods of `queue.Queue` except for `task_done()` and `join()`.

qsize()

Return the approximate size of the queue. Because of multithreading/multiprocessing semantics, this number is not reliable.

Note that this may raise `NotImplementedError` on Unix platforms like Mac OS X where `sem_getvalue()` is not implemented.

empty()

Return `True` if the queue is empty, `False` otherwise. Because of multithreading/multiprocessing semantics, this is not reliable.

full()

Return `True` if the queue is full, `False` otherwise. Because of multithreading/multiprocessing semantics, this is not reliable.

put(obj[, block[, timeout]])

Put `obj` into the queue. If the optional argument `block` is `True` (the default) and `timeout` is `None` (the default), block if necessary until a free slot is available. If `timeout` is a positive number, it blocks at most `timeout` seconds and raises the `queue.Full` exception if no free slot was available within that time. Otherwise (`block` is `False`), put an item on the queue if a free slot is immediately available, else raise the `queue.Full` exception (`timeout` is ignored in that case).

put_nowait(obj)

Equivalent to `put(obj, False)`.

get([block[, timeout]])

Remove and return an item from the queue. If optional args `block` is `True` (the default) and `timeout` is `None` (the default), block if necessary until an item is available. If `timeout` is a positive number, it blocks at most `timeout` seconds and raises the `queue.Empty` exception if no item was available within that time. Otherwise (`block` is `False`), return an item if one is immediately available, else raise the `queue.Empty` exception (`timeout` is ignored in that case).

get_nowait()

Equivalent to `get(False)`.

`multiprocessing.Queue` has a few additional methods not found in `queue.Queue`. These methods are usually unnecessary for most code:

close()

Indicate that no more data will be put on this queue by the current process. The background thread will quit once it has flushed all buffered data to the pipe. This is called automatically when the queue is garbage collected.

join_thread()

Join the background thread. This can only be used after `close()` has been called. It blocks until the background thread exits, ensuring that all data in the buffer has been flushed to the pipe.

By default if a process is not the creator of the queue then on exit it will attempt to join the queue's background thread. The process can call `cancel_join_thread()` to make `join_thread()` do nothing.

cancel_join_thread()

Prevent `join_thread()` from blocking. In particular, this prevents the background thread from being joined automatically when the process exits – see `join_thread()`.

A better name for this method might be `allow_exit_without_flush()`. It is likely to cause enqueued data to be lost, and you almost certainly will not need to use it. It is really only there if

you need the current process to exit immediately without waiting to flush enqueued data to the underlying pipe, and you don't care about lost data.

Note: This class's functionality requires a functioning shared semaphore implementation on the host operating system. Without one, the functionality in this class will be disabled, and attempts to instantiate a *Queue* will result in an *ImportError*. See [bpo-3770](#) for additional information. The same holds true for any of the specialized queue types listed below.

class multiprocessing.SimpleQueue

It is a simplified *Queue* type, very close to a locked *Pipe*.

empty()

Return *True* if the queue is empty, *False* otherwise.

get()

Remove and return an item from the queue.

put(item)

Put *item* into the queue.

class multiprocessing.JoinableQueue([maxsize])

JoinableQueue, a *Queue* subclass, is a queue which additionally has *task_done()* and *join()* methods.

task_done()

Indicate that a formerly enqueued task is complete. Used by queue consumers. For each *get()* used to fetch a task, a subsequent call to *task_done()* tells the queue that the processing on the task is complete.

If a *join()* is currently blocking, it will resume when all items have been processed (meaning that a *task_done()* call was received for every item that had been *put()* into the queue).

Raises a *ValueError* if called more times than there were items placed in the queue.

join()

Block until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer calls *task_done()* to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, *join()* unblocks.

Miscellaneous

multiprocessing.active_children()

Return list of all live children of the current process.

Calling this has the side effect of “joining” any processes which have already finished.

multiprocessing.cpu_count()

Return the number of CPUs in the system.

This number is not equivalent to the number of CPUs the current process can use. The number of usable CPUs can be obtained with `len(os.sched_getaffinity(0))`

May raise *NotImplementedError*.

See also:

`os.cpu_count()`

`multiprocessing.current_process()`

Return the *Process* object corresponding to the current process.

An analogue of *threading.current_thread()*.

`multiprocessing.freeze_support()`

Add support for when a program which uses *multiprocessing* has been frozen to produce a Windows executable. (Has been tested with **py2exe**, **PyInstaller** and **cx_Freeze**.)

One needs to call this function straight after the `if __name__ == '__main__':` line of the main module. For example:

```
from multiprocessing import Process, freeze_support

def f():
    print('hello world!')

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

If the `freeze_support()` line is omitted then trying to run the frozen executable will raise *RuntimeError*.

Calling `freeze_support()` has no effect when invoked on any operating system other than Windows. In addition, if the module is being run normally by the Python interpreter on Windows (the program has not been frozen), then `freeze_support()` has no effect.

`multiprocessing.get_all_start_methods()`

Returns a list of the supported start methods, the first of which is the default. The possible start methods are 'fork', 'spawn' and 'forkserver'. On Windows only 'spawn' is available. On Unix 'fork' and 'spawn' are always supported, with 'fork' being the default.

New in version 3.4.

`multiprocessing.get_context(method=None)`

Return a context object which has the same attributes as the *multiprocessing* module.

If *method* is *None* then the default context is returned. Otherwise *method* should be 'fork', 'spawn', 'forkserver'. *ValueError* is raised if the specified start method is not available.

New in version 3.4.

`multiprocessing.get_start_method(allow_none=False)`

Return the name of start method used for starting processes.

If the start method has not been fixed and *allow_none* is false, then the start method is fixed to the default and the name is returned. If the start method has not been fixed and *allow_none* is true then *None* is returned.

The return value can be 'fork', 'spawn', 'forkserver' or *None*. 'fork' is the default on Unix, while 'spawn' is the default on Windows.

New in version 3.4.

`multiprocessing.set_executable()`

Sets the path of the Python interpreter to use when starting a child process. (By default *sys.executable* is used). Embedders will probably need to do some thing like

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

before they can create child processes.

Changed in version 3.4: Now supported on Unix when the 'spawn' start method is used.

`multiprocessing.set_start_method(method)`

Set the method which should be used to start child processes. *method* can be 'fork', 'spawn' or 'forkserver'.

Note that this should be called at most once, and it should be protected inside the `if __name__ == '__main__':` clause of the main module.

New in version 3.4.

Note: `multiprocessing` contains no analogues of `threading.active_count()`, `threading.enumerate()`, `threading.settrace()`, `threading.setprofile()`, `threading.Timer`, or `threading.local`.

Connection Objects

Connection objects allow the sending and receiving of picklable objects or strings. They can be thought of as message oriented connected sockets.

Connection objects are usually created using *Pipe* – see also *Listeners and Clients*.

`class multiprocessing.connection.Connection`

`send(obj)`

Send an object to the other end of the connection which should be read using `recv()`.

The object must be picklable. Very large pickles (approximately 32 MiB+, though it depends on the OS) may raise a `ValueError` exception.

`recv()`

Return an object sent from the other end of the connection using `send()`. Blocks until there is something to receive. Raises `EOFError` if there is nothing left to receive and the other end was closed.

`fileno()`

Return the file descriptor or handle used by the connection.

`close()`

Close the connection.

This is called automatically when the connection is garbage collected.

`poll([timeout])`

Return whether there is any data available to be read.

If *timeout* is not specified then it will return immediately. If *timeout* is a number then this specifies the maximum time in seconds to block. If *timeout* is `None` then an infinite timeout is used.

Note that multiple connection objects may be polled at once by using `multiprocessing.connection.wait()`.

`send_bytes(buffer[, offset[, size]])`

Send byte data from a *bytes-like object* as a complete message.

If *offset* is given then data is read from that position in *buffer*. If *size* is given then that many bytes will be read from *buffer*. Very large buffers (approximately 32 MiB+, though it depends on the OS) may raise a `ValueError` exception

`recv_bytes([maxlength])`

Return a complete message of byte data sent from the other end of the connection as a string.

Blocks until there is something to receive. Raises `EOFError` if there is nothing left to receive and the other end has closed.

If *maxlength* is specified and the message is longer than *maxlength* then `OSError` is raised and the connection will no longer be readable.

Changed in version 3.3: This function used to raise `IOError`, which is now an alias of `OSError`.

`recv_bytes_into(buffer[, offset])`

Read into *buffer* a complete message of byte data sent from the other end of the connection and return the number of bytes in the message. Blocks until there is something to receive. Raises `EOFError` if there is nothing left to receive and the other end was closed.

buffer must be a writable *bytes-like object*. If *offset* is given then the message will be written into the buffer from that position. Offset must be a non-negative integer less than the length of *buffer* (in bytes).

If the buffer is too short then a `BufferTooShort` exception is raised and the complete message is available as `e.args[0]` where `e` is the exception instance.

Changed in version 3.3: Connection objects themselves can now be transferred between processes using `Connection.send()` and `Connection.recv()`.

New in version 3.3: Connection objects now support the context management protocol – see *Context Manager Types*. `__enter__()` returns the connection object, and `__exit__()` calls `close()`.

For example:

```
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes(b'thank you')
>>> a.recv_bytes()
b'thank you'
>>> import array
>>> arr1 = array.array('i', range(5))
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

Warning: The `Connection.recv()` method automatically unpickles the data it receives, which can be a security risk unless you can trust the process which sent the message.

Therefore, unless the connection object was produced using `Pipe()` you should only use the `recv()` and `send()` methods after performing some sort of authentication. See *Authentication keys*.

Warning: If a process is killed while it is trying to read or write to a pipe then the data in the pipe is likely to become corrupted, because it may become impossible to be sure where the message boundaries lie.

Synchronization primitives

Generally synchronization primitives are not as necessary in a multiprocess program as they are in a multi-threaded program. See the documentation for [threading](#) module.

Note that one can also create synchronization primitives by using a manager object – see [Managers](#).

class multiprocessing.Barrier(*parties*[, *action*[, *timeout*]])

A barrier object: a clone of [threading.Barrier](#).

New in version 3.3.

class multiprocessing.BoundedSemaphore([*value*])

A bounded semaphore object: a close analog of [threading.BoundedSemaphore](#).

A solitary difference from its close analog exists: its `acquire` method's first argument is named *block*, as is consistent with [Lock.acquire\(\)](#).

Note: On Mac OS X, this is indistinguishable from [Semaphore](#) because `sem_getvalue()` is not implemented on that platform.

class multiprocessing.Condition([*lock*])

A condition variable: an alias for [threading.Condition](#).

If *lock* is specified then it should be a [Lock](#) or [RLock](#) object from [multiprocessing](#).

Changed in version 3.3: The `wait_for()` method was added.

class multiprocessing.Event

A clone of [threading.Event](#).

class multiprocessing.Lock

A non-recursive lock object: a close analog of [threading.Lock](#). Once a process or thread has acquired a lock, subsequent attempts to acquire it from any process or thread will block until it is released; any process or thread may release it. The concepts and behaviors of [threading.Lock](#) as it applies to threads are replicated here in [multiprocessing.Lock](#) as it applies to either processes or threads, except as noted.

Note that [Lock](#) is actually a factory function which returns an instance of `multiprocessing.synchronize.Lock` initialized with a default context.

[Lock](#) supports the [context manager](#) protocol and thus may be used in `with` statements.

acquire(*block=True*, *timeout=None*)

Acquire a lock, blocking or non-blocking.

With the *block* argument set to **True** (the default), the method call will block until the lock is in an unlocked state, then set it to locked and return **True**. Note that the name of this first argument differs from that in [threading.Lock.acquire\(\)](#).

With the *block* argument set to **False**, the method call does not block. If the lock is currently in a locked state, return **False**; otherwise set the lock to a locked state and return **True**.

When invoked with a positive, floating-point value for *timeout*, block for at most the number of seconds specified by *timeout* as long as the lock can not be acquired. Invocations with a negative value for *timeout* are equivalent to a *timeout* of zero. Invocations with a *timeout* value of **None** (the default) set the timeout period to infinite. Note that the treatment of negative or **None** values for *timeout* differs from the implemented behavior in [threading.Lock.acquire\(\)](#). The *timeout* argument has no practical implications if the *block* argument is set to **False** and is thus ignored. Returns **True** if the lock has been acquired or **False** if the timeout period has elapsed.

release()

Release a lock. This can be called from any process or thread, not only the process or thread which originally acquired the lock.

Behavior is the same as in `threading.Lock.release()` except that when invoked on an unlocked lock, a `ValueError` is raised.

class multiprocessing.RLock

A recursive lock object: a close analog of `threading.RLock`. A recursive lock must be released by the process or thread that acquired it. Once a process or thread has acquired a recursive lock, the same process or thread may acquire it again without blocking; that process or thread must release it once for each time it has been acquired.

Note that `RLock` is actually a factory function which returns an instance of `multiprocessing.synchronize.RLock` initialized with a default context.

`RLock` supports the *context manager* protocol and thus may be used in `with` statements.

acquire(block=True, timeout=None)

Acquire a lock, blocking or non-blocking.

When invoked with the `block` argument set to `True`, block until the lock is in an unlocked state (not owned by any process or thread) unless the lock is already owned by the current process or thread. The current process or thread then takes ownership of the lock (if it does not already have ownership) and the recursion level inside the lock increments by one, resulting in a return value of `True`. Note that there are several differences in this first argument's behavior compared to the implementation of `threading.RLock.acquire()`, starting with the name of the argument itself.

When invoked with the `block` argument set to `False`, do not block. If the lock has already been acquired (and thus is owned) by another process or thread, the current process or thread does not take ownership and the recursion level within the lock is not changed, resulting in a return value of `False`. If the lock is in an unlocked state, the current process or thread takes ownership and the recursion level is incremented, resulting in a return value of `True`.

Use and behaviors of the `timeout` argument are the same as in `Lock.acquire()`. Note that some of these behaviors of `timeout` differ from the implemented behaviors in `threading.RLock.acquire()`.

release()

Release a lock, decrementing the recursion level. If after the decrement the recursion level is zero, reset the lock to unlocked (not owned by any process or thread) and if any other processes or threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling process or thread.

Only call this method when the calling process or thread owns the lock. An `AssertionError` is raised if this method is called by a process or thread other than the owner or if the lock is in an unlocked (unowned) state. Note that the type of exception raised in this situation differs from the implemented behavior in `threading.RLock.release()`.

class multiprocessing.Semaphore([value])

A semaphore object: a close analog of `threading.Semaphore`.

A solitary difference from its close analog exists: its `acquire` method's first argument is named `block`, as is consistent with `Lock.acquire()`.

Note: On Mac OS X, `sem_timedwait` is unsupported, so calling `acquire()` with a timeout will emulate that function's behavior using a sleeping loop.

Note: If the SIGINT signal generated by Ctrl-C arrives while the main thread is blocked by a call to `BoundedSemaphore.acquire()`, `Lock.acquire()`, `RLock.acquire()`, `Semaphore.acquire()`, `Condition.acquire()` or `Condition.wait()` then the call will be immediately interrupted and `KeyboardInterrupt` will be raised.

This differs from the behaviour of `threading` where SIGINT will be ignored while the equivalent blocking calls are in progress.

Note: Some of this package’s functionality requires a functioning shared semaphore implementation on the host operating system. Without one, the `multiprocessing.synchronize` module will be disabled, and attempts to import it will result in an `ImportError`. See [bpo-3770](#) for additional information.

Shared ctypes Objects

It is possible to create shared objects using shared memory which can be inherited by child processes.

`multiprocessing.Value(typecode_or_type, *args, lock=True)`

Return a `ctypes` object allocated from shared memory. By default the return value is actually a synchronized wrapper for the object. The object itself can be accessed via the `value` attribute of a `Value`.

`typecode_or_type` determines the type of the returned object: it is either a `ctypes` type or a one character typecode of the kind used by the `array` module. `*args` is passed on to the constructor for the type.

If `lock` is `True` (the default) then a new recursive lock object is created to synchronize access to the value. If `lock` is a `Lock` or `RLock` object then that will be used to synchronize access to the value. If `lock` is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Operations like `+=` which involve a read and write are not atomic. So if, for instance, you want to atomically increment a shared value it is insufficient to just do

```
counter.value += 1
```

Assuming the associated lock is recursive (which it is by default) you can instead do

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

Note that `lock` is a keyword-only argument.

`multiprocessing.Array(typecode_or_type, size_or_initializer, *, lock=True)`

Return a `ctypes` array allocated from shared memory. By default the return value is actually a synchronized wrapper for the array.

`typecode_or_type` determines the type of the elements of the returned array: it is either a `ctypes` type or a one character typecode of the kind used by the `array` module. If `size_or_initializer` is an integer, then it determines the length of the array, and the array will be initially zeroed. Otherwise, `size_or_initializer` is a sequence which is used to initialize the array and whose length determines the length of the array.

If `lock` is `True` (the default) then a new lock object is created to synchronize access to the value. If `lock` is a `Lock` or `RLock` object then that will be used to synchronize access to the value. If `lock` is

`False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Note that `lock` is a keyword only argument.

Note that an array of `ctypes.c_char` has `value` and `raw` attributes which allow one to use it to store and retrieve strings.

The `multiprocessing.sharedctypes` module

The `multiprocessing.sharedctypes` module provides functions for allocating `ctypes` objects from shared memory which can be inherited by child processes.

Note: Although it is possible to store a pointer in shared memory remember that this will refer to a location in the address space of a specific process. However, the pointer is quite likely to be invalid in the context of a second process and trying to dereference the pointer from the second process may cause a crash.

`multiprocessing.sharedctypes.RawArray(typecode_or_type, size_or_initializer)`

Return a `ctypes` array allocated from shared memory.

`typecode_or_type` determines the type of the elements of the returned array: it is either a `ctypes` type or a one character typecode of the kind used by the `array` module. If `size_or_initializer` is an integer then it determines the length of the array, and the array will be initially zeroed. Otherwise `size_or_initializer` is a sequence which is used to initialize the array and whose length determines the length of the array.

Note that setting and getting an element is potentially non-atomic – use `Array()` instead to make sure that access is automatically synchronized using a lock.

`multiprocessing.sharedctypes.RawValue(typecode_or_type, *args)`

Return a `ctypes` object allocated from shared memory.

`typecode_or_type` determines the type of the returned object: it is either a `ctypes` type or a one character typecode of the kind used by the `array` module. `*args` is passed on to the constructor for the type.

Note that setting and getting the value is potentially non-atomic – use `Value()` instead to make sure that access is automatically synchronized using a lock.

Note that an array of `ctypes.c_char` has `value` and `raw` attributes which allow one to use it to store and retrieve strings – see documentation for `ctypes`.

`multiprocessing.sharedctypes.Array(typecode_or_type, size_or_initializer, *, lock=True)`

The same as `RawArray()` except that depending on the value of `lock` a process-safe synchronization wrapper may be returned instead of a raw `ctypes` array.

If `lock` is `True` (the default) then a new lock object is created to synchronize access to the value. If `lock` is a `Lock` or `RLock` object then that will be used to synchronize access to the value. If `lock` is `False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Note that `lock` is a keyword-only argument.

`multiprocessing.sharedctypes.Value(typecode_or_type, *args, lock=True)`

The same as `RawValue()` except that depending on the value of `lock` a process-safe synchronization wrapper may be returned instead of a raw `ctypes` object.

If `lock` is `True` (the default) then a new lock object is created to synchronize access to the value. If `lock` is a `Lock` or `RLock` object then that will be used to synchronize access to the value. If `lock` is

`False` then access to the returned object will not be automatically protected by a lock, so it will not necessarily be “process-safe”.

Note that *lock* is a keyword-only argument.

`multiprocessing.sharedctypes.copy(obj)`

Return a ctypes object allocated from shared memory which is a copy of the ctypes object *obj*.

`multiprocessing.sharedctypes.synchronized(obj[, lock])`

Return a process-safe wrapper object for a ctypes object which uses *lock* to synchronize access. If *lock* is `None` (the default) then a `multiprocessing.RLock` object is created automatically.

A synchronized wrapper will have two methods in addition to those of the object it wraps: `get_obj()` returns the wrapped object and `get_lock()` returns the lock object used for synchronization.

Note that accessing the ctypes object through the wrapper can be a lot slower than accessing the raw ctypes object.

Changed in version 3.5: Synchronized objects support the *context manager* protocol.

The table below compares the syntax for creating shared ctypes objects from shared memory with the normal ctypes syntax. (In the table `MyStruct` is some subclass of `ctypes.Structure`.)

ctypes	sharedctypes using type	sharedctypes using typecode
<code>c_double(2.4)</code>	<code>RawValue(c_double, 2.4)</code>	<code>RawValue('d', 2.4)</code>
<code>MyStruct(4, 6)</code>	<code>RawValue(MyStruct, 4, 6)</code>	
<code>(c_short * 7)()</code>	<code>RawArray(c_short, 7)</code>	<code>RawArray('h', 7)</code>
<code>(c_int * 3)(9, 2, 8)</code>	<code>RawArray(c_int, (9, 2, 8))</code>	<code>RawArray('i', (9, 2, 8))</code>

Below is an example where a number of ctypes objects are modified by a child process:

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value *= 2
    x.value *= 2
    s.value = s.value.upper()
    for a in A:
        a.x *= 2
        a.y *= 2

if __name__ == '__main__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', b'hello world', lock=lock)
    A = Array(Point, [(1.875,-6.25), (-5.75,2.0), (2.375,9.5)], lock=lock)

    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()
```

(continues on next page)

(continued from previous page)

```
print(n.value)
print(x.value)
print(s.value)
print([(a.x, a.y) for a in A])
```

The results printed are

```
49
0.1111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
```

Managers

Managers provide a way to create data which can be shared between different processes, including sharing over a network between processes running on different machines. A manager object controls a server process which manages *shared objects*. Other processes can access the shared objects by using proxies.

`multiprocessing.Manager()`

Returns a started *SyncManager* object which can be used for sharing objects between processes. The returned manager object corresponds to a spawned child process and has methods which will create shared objects and return corresponding proxies.

Manager processes will be shutdown as soon as they are garbage collected or their parent process exits. The manager classes are defined in the `multiprocessing.managers` module:

`class multiprocessing.managers.BaseManager([address[, authkey]])`

Create a BaseManager object.

Once created one should call `start()` or `get_server().serve_forever()` to ensure that the manager object refers to a started manager process.

address is the address on which the manager process listens for new connections. If *address* is `None` then an arbitrary one is chosen.

authkey is the authentication key which will be used to check the validity of incoming connections to the server process. If *authkey* is `None` then `current_process().authkey` is used. Otherwise *authkey* is used and it must be a byte string.

`start([initializer[, initargs]])`

Start a subprocess to start the manager. If *initializer* is not `None` then the subprocess will call `initializer(*initargs)` when it starts.

`get_server()`

Returns a `Server` object which represents the actual server under the control of the Manager. The `Server` object supports the `serve_forever()` method:

```
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=(' ', 50000), authkey=b'abc')
>>> server = manager.get_server()
>>> server.serve_forever()
```

`Server` additionally has an *address* attribute.

`connect()`

Connect a local manager object to a remote manager process:

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 5000), authkey=b'abc')
>>> m.connect()
```

shutdown()

Stop the process used by the manager. This is only available if *start()* has been used to start the server process.

This can be called multiple times.

register(*typeid*[, *callable*[, *proxytype*[, *exposed*[, *method_to_typeid*[, *create_method*]]]])

A classmethod which can be used for registering a type or callable with the manager class.

typeid is a “type identifier” which is used to identify a particular type of shared object. This must be a string.

callable is a callable used for creating objects for this type identifier. If a manager instance will be connected to the server using the *connect()* method, or if the *create_method* argument is *False* then this can be left as *None*.

proxytype is a subclass of *BaseProxy* which is used to create proxies for shared objects with this *typeid*. If *None* then a proxy class is created automatically.

exposed is used to specify a sequence of method names which proxies for this *typeid* should be allowed to access using *BaseProxy._callmethod()*. (If *exposed* is *None* then *proxytype._exposed_* is used instead if it exists.) In the case where no exposed list is specified, all “public methods” of the shared object will be accessible. (Here a “public method” means any attribute which has a *__call__()* method and whose name does not begin with *'_'*.)

method_to_typeid is a mapping used to specify the return type of those exposed methods which should return a proxy. It maps method names to *typeid* strings. (If *method_to_typeid* is *None* then *proxytype._method_to_typeid_* is used instead if it exists.) If a method’s name is not a key of this mapping or if the mapping is *None* then the object returned by the method will be copied by value.

create_method determines whether a method should be created with name *typeid* which can be used to tell the server process to create a new shared object and return a proxy for it. By default it is *True*.

BaseManager instances also have one read-only property:

address

The address used by the manager.

Changed in version 3.3: Manager objects support the context management protocol – see *Context Manager Types*. *__enter__()* starts the server process (if it has not already started) and then returns the manager object. *__exit__()* calls *shutdown()*.

In previous versions *__enter__()* did not start the manager’s server process if it was not already started.

class multiprocessing.managers.SyncManager

A subclass of *BaseManager* which can be used for the synchronization of processes. Objects of this type are returned by *multiprocessing.Manager()*.

Its methods create and return *Proxy Objects* for a number of commonly used data types to be synchronized across processes. This notably includes shared lists and dictionaries.

Barrier(*parties*[, *action*[, *timeout*]])

Create a shared *threading.Barrier* object and return a proxy for it.

New in version 3.3.

BoundedSemaphore(*[value]*)

Create a shared *threading.BoundedSemaphore* object and return a proxy for it.

Condition(*[lock]*)

Create a shared *threading.Condition* object and return a proxy for it.

If *lock* is supplied then it should be a proxy for a *threading.Lock* or *threading.RLock* object.

Changed in version 3.3: The *wait_for()* method was added.

Event()

Create a shared *threading.Event* object and return a proxy for it.

Lock()

Create a shared *threading.Lock* object and return a proxy for it.

Namespace()

Create a shared *Namespace* object and return a proxy for it.

Queue(*[maxsize]*)

Create a shared *queue.Queue* object and return a proxy for it.

RLock()

Create a shared *threading.RLock* object and return a proxy for it.

Semaphore(*[value]*)

Create a shared *threading.Semaphore* object and return a proxy for it.

Array(*typecode, sequence*)

Create an array and return a proxy for it.

Value(*typecode, value*)

Create an object with a writable *value* attribute and return a proxy for it.

dict()

dict(*mapping*)

dict(*sequence*)

Create a shared *dict* object and return a proxy for it.

list()

list(*sequence*)

Create a shared *list* object and return a proxy for it.

Changed in version 3.6: Shared objects are capable of being nested. For example, a shared container object such as a shared list can contain other shared objects which will all be managed and synchronized by the *SyncManager*.

class multiprocessing.managers.Namespace

A type that can register with *SyncManager*.

A namespace object has no public methods, but does have writable attributes. Its representation shows the values of its attributes.

However, when using a proxy for a namespace object, an attribute beginning with '_' will be an attribute of the proxy and not an attribute of the referent:

```
>>> manager = multiprocessing.Manager()
>>> Global = manager.Namespace()
>>> Global.x = 10
>>> Global.y = 'hello'
>>> Global._z = 12.3      # this is an attribute of the proxy
>>> print(Global)
Namespace(x=10, y='hello')
```

Customized managers

To create one's own manager, one creates a subclass of *BaseManager* and uses the *register()* classmethod to register new types or callables with the manager class. For example:

```
from multiprocessing.managers import BaseManager

class MathsClass:
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
    pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    with MyManager() as manager:
        maths = manager.Maths()
        print(maths.add(4, 3))           # prints 7
        print(maths.mul(7, 8))          # prints 56
```

Using a remote manager

It is possible to run a manager server on one machine and have clients use it from other machines (assuming that the firewalls involved allow it).

Running the following commands creates a server for a single shared queue which remote clients can access:

```
>>> from multiprocessing.managers import BaseManager
>>> from queue import Queue
>>> queue = Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda:queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

One client can access the server as follows:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.put('hello')
```

Another client can also use it:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'
```

Local processes can also access that queue, using the code from above on the client to access it remotely:

```
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super(Worker, self).__init__()
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
...
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

Proxy Objects

A proxy is an object which *refers* to a shared object which lives (presumably) in a different process. The shared object is said to be the *referent* of the proxy. Multiple proxy objects may have the same referent.

A proxy object has methods which invoke corresponding methods of its referent (although not every method of the referent will necessarily be available through the proxy). In this way, a proxy can be used just like its referent can:

```
>>> from multiprocessing import Manager
>>> manager = Manager()
>>> l = manager.list([i*i for i in range(10)])
>>> print(l)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print(repr(l))
<ListProxy object, typeid 'list' at 0x...>
>>> l[4]
16
>>> l[2:5]
[4, 9, 16]
```

Notice that applying `str()` to a proxy will return the representation of the referent, whereas applying `repr()` will return the representation of the proxy.

An important feature of proxy objects is that they are picklable so they can be passed between processes. As such, a referent can contain *Proxy Objects*. This permits nesting of these managed lists, dicts, and other *Proxy Objects*:

```
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)           # referent of a now contains referent of b
>>> print(a, b)
[<ListProxy object, typeid 'list' at ...>] []
>>> b.append('hello')
>>> print(a[0], b)
['hello'] ['hello']
```

Similarly, dict and list proxies may be nested inside one another:

```
>>> l_outer = manager.list([ manager.dict() for i in range(2) ])
>>> d_first_inner = l_outer[0]
>>> d_first_inner['a'] = 1
>>> d_first_inner['b'] = 2
>>> l_outer[1]['c'] = 3
>>> l_outer[1]['z'] = 26
>>> print(l_outer[0])
{'a': 1, 'b': 2}
>>> print(l_outer[1])
{'c': 3, 'z': 26}
```

If standard (non-proxy) *list* or *dict* objects are contained in a referent, modifications to those mutable values will not be propagated through the manager because the proxy has no way of knowing when the values contained within are modified. However, storing a value in a container proxy (which triggers a `__setitem__` on the proxy object) does propagate through the manager and so to effectively modify such an item, one could re-assign the modified value to the container proxy:

```
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
d = lproxy[0]
d['a'] = 1
d['b'] = 2
# at this point, the changes to d are not yet synced, but by
# updating the dictionary, the proxy is notified of the change
lproxy[0] = d
```

This approach is perhaps less convenient than employing nested *Proxy Objects* for most use cases but also demonstrates a level of control over the synchronization.

Note: The proxy types in *multiprocessing* do nothing to support comparisons by value. So, for instance, we have:

```
>>> manager.list([1,2,3]) == [1,2,3]
False
```

One should just use a copy of the referent instead when making comparisons.

`class multiprocessing.managers.BaseProxy`

Proxy objects are instances of subclasses of *BaseProxy*.

`_callmethod(methodname[, args[, kwds]])`

Call and return the result of a method of the proxy's referent.

If proxy is a proxy whose referent is obj then the expression

```
proxy._callmethod(methodname, args, kwds)
```

will evaluate the expression

```
getattr(obj, methodname)(*args, **kwds)
```

in the manager's process.

The returned value will be a copy of the result of the call or a proxy to a new shared object – see documentation for the *method_to_typeid* argument of *BaseManager.register()*.

If an exception is raised by the call, then is re-raised by `_callmethod()`. If some other exception is raised in the manager's process then this is converted into a `RemoteError` exception and is raised by `_callmethod()`.

Note in particular that an exception will be raised if *methodname* has not been *exposed*.

An example of the usage of `_callmethod()`:

```
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
>>> l._callmethod('__getitem__', (slice(2, 7),)) # equivalent to l[2:7]
[2, 3, 4, 5, 6]
>>> l._callmethod('__getitem__', (20,))          # equivalent to l[20]
Traceback (most recent call last):
...
IndexError: list index out of range
```

`_getvalue()`

Return a copy of the referent.

If the referent is unpicklable then this will raise an exception.

`__repr__()`

Return a representation of the proxy object.

`__str__()`

Return the representation of the referent.

Cleanup

A proxy object uses a weakref callback so that when it gets garbage collected it deregisters itself from the manager which owns its referent.

A shared object gets deleted from the manager process when there are no longer any proxies referring to it.

Process Pools

One can create a pool of processes which will carry out tasks submitted to it with the *Pool* class.

```
class multiprocessing.pool.Pool([processes[, initializer[, initargs[, maxtasksperchild[, context]]]])
```

A process pool object which controls a pool of worker processes to which jobs can be submitted. It supports asynchronous results with timeouts and callbacks and has a parallel map implementation.

processes is the number of worker processes to use. If *processes* is `None` then the number returned by `os.cpu_count()` is used.

If *initializer* is not `None` then each worker process will call `initializer(*initargs)` when it starts.

maxtasksperchild is the number of tasks a worker process can complete before it will exit and be replaced with a fresh worker process, to enable unused resources to be freed. The default *maxtasksperchild* is `None`, which means worker processes will live as long as the pool.

context can be used to specify the context used for starting the worker processes. Usually a pool is created using the function `multiprocessing.Pool()` or the `Pool()` method of a context object. In both cases *context* is set appropriately.

Note that the methods of the pool object should only be called by the process which created the pool.

New in version 3.2: *maxtasksperchild*

New in version 3.4: *context*

Note: Worker processes within a `Pool` typically live for the complete duration of the Pool's work queue. A frequent pattern found in other systems (such as Apache, `mod_wsgi`, etc) to free resources held by workers is to allow a worker within a pool to complete only a set amount of work before being exiting, being cleaned up and a new process spawned to replace the old one. The *maxtasksperchild* argument to the `Pool` exposes this ability to the end user.

```
apply(func[, args[, kwds]])
```

Call *func* with arguments *args* and keyword arguments *kwds*. It blocks until the result is ready. Given this blocks, `apply_async()` is better suited for performing work in parallel. Additionally, *func* is only executed in one of the workers of the pool.

```
apply_async(func[, args[, kwds[, callback[, error_callback]]]])
```

A variant of the `apply()` method which returns a result object.

If *callback* is specified then it should be a callable which accepts a single argument. When the result becomes ready *callback* is applied to it, that is unless the call failed, in which case the *error_callback* is applied instead.

If *error_callback* is specified then it should be a callable which accepts a single argument. If the target function fails, then the *error_callback* is called with the exception instance.

Callbacks should complete immediately since otherwise the thread which handles the results will get blocked.

```
map(func, iterable[, chunksize])
```

A parallel equivalent of the `map()` built-in function (it supports only one *iterable* argument though). It blocks until the result is ready.

This method chops the iterable into a number of chunks which it submits to the process pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer.

```
map_async(func, iterable[, chunksize[, callback[, error_callback]]])
```

A variant of the `map()` method which returns a result object.

If *callback* is specified then it should be a callable which accepts a single argument. When the result becomes ready *callback* is applied to it, that is unless the call failed, in which case the *error_callback* is applied instead.

If *error_callback* is specified then it should be a callable which accepts a single argument. If the target function fails, then the *error_callback* is called with the exception instance.

Callbacks should complete immediately since otherwise the thread which handles the results will get blocked.

imap(*func*, *iterable*[, *chunksize*])

A lazier version of *map()*.

The *chunksize* argument is the same as the one used by the *map()* method. For very long iterables using a large value for *chunksize* can make the job complete **much** faster than using the default value of 1.

Also if *chunksize* is 1 then the *next()* method of the iterator returned by the *imap()* method has an optional *timeout* parameter: *next(timeout)* will raise *multiprocessing.TimeoutError* if the result cannot be returned within *timeout* seconds.

imap_unordered(*func*, *iterable*[, *chunksize*])

The same as *imap()* except that the ordering of the results from the returned iterator should be considered arbitrary. (Only when there is only one worker process is the order guaranteed to be “correct”.)

starmap(*func*, *iterable*[, *chunksize*])

Like *map()* except that the elements of the *iterable* are expected to be iterables that are unpacked as arguments.

Hence an *iterable* of [(1,2), (3, 4)] results in [func(1,2), func(3,4)].

New in version 3.3.

starmap_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])

A combination of *starmap()* and *map_async()* that iterates over *iterable* of iterables and calls *func* with the iterables unpacked. Returns a result object.

New in version 3.3.

close()

Prevents any more tasks from being submitted to the pool. Once all the tasks have been completed the worker processes will exit.

terminate()

Stops the worker processes immediately without completing outstanding work. When the pool object is garbage collected *terminate()* will be called immediately.

join()

Wait for the worker processes to exit. One must call *close()* or *terminate()* before using *join()*.

New in version 3.3: Pool objects now support the context management protocol – see *Context Manager Types*. *__enter__()* returns the pool object, and *__exit__()* calls *terminate()*.

class multiprocessing.pool.AsyncResult

The class of the result returned by *Pool.apply_async()* and *Pool.map_async()*.

get([*timeout*])

Return the result when it arrives. If *timeout* is not *None* and the result does not arrive within *timeout* seconds then *multiprocessing.TimeoutError* is raised. If the remote call raised an exception then that exception will be reraised by *get()*.

wait([*timeout*])

Wait until the result is available or until *timeout* seconds pass.

ready()

Return whether the call has completed.

successful()

Return whether the call completed without raising an exception. Will raise *AssertionError* if the result is not ready.

The following example demonstrates the use of a pool:

```
from multiprocessing import Pool
import time

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(processes=4) as pool:          # start 4 worker processes
        result = pool.apply_async(f, (10,)) # evaluate "f(10)" asynchronously in a
        ↪ single process
        print(result.get(timeout=1))        # prints "100" unless your computer is
        ↪ *very* slow

        print(pool.map(f, range(10)))      # prints "[0, 1, 4, ..., 81]"

        it = pool.imap(f, range(10))
        print(next(it))                    # prints "0"
        print(next(it))                    # prints "1"
        print(it.next(timeout=1))          # prints "4" unless your computer is *very*
        ↪ slow

        result = pool.apply_async(time.sleep, (10,))
        print(result.get(timeout=1))        # raises multiprocessing.TimeoutError
```

Listeners and Clients

Usually message passing between processes is done using queues or by using *Connection* objects returned by *Pipe()*.

However, the *multiprocessing.connection* module allows some extra flexibility. It basically gives a high level message oriented API for dealing with sockets or Windows named pipes. It also has support for *digest authentication* using the *hmac* module, and for polling multiple connections at the same time.

multiprocessing.connection.deliver_challenge(connection, authkey)

Send a randomly generated message to the other end of the connection and wait for a reply.

If the reply matches the digest of the message using *authkey* as the key then a welcome message is sent to the other end of the connection. Otherwise *AuthenticationError* is raised.

multiprocessing.connection.answer_challenge(connection, authkey)

Receive a message, calculate the digest of the message using *authkey* as the key, and then send the digest back.

If a welcome message is not received, then *AuthenticationError* is raised.

multiprocessing.connection.Client(address[, family[, authkey]])

Attempt to set up a connection to the listener which is using address *address*, returning a *Connection*.

The type of the connection is determined by *family* argument, but this can generally be omitted since it can usually be inferred from the format of *address*. (See [Address Formats](#))

If *authkey* is given and not `None`, it should be a byte string and will be used as the secret key for an HMAC-based authentication challenge. No authentication is done if *authkey* is `None`. [AuthenticationError](#) is raised if authentication fails. See [Authentication keys](#).

class `multiprocessing.connection.Listener`(`[address[, family[, backlog[, authkey]]]]`)

A wrapper for a bound socket or Windows named pipe which is ‘listening’ for connections.

address is the address to be used by the bound socket or named pipe of the listener object.

Note: If an address of ‘0.0.0.0’ is used, the address will not be a connectable end point on Windows. If you require a connectable end-point, you should use ‘127.0.0.1’.

family is the type of socket (or named pipe) to use. This can be one of the strings ‘AF_INET’ (for a TCP socket), ‘AF_UNIX’ (for a Unix domain socket) or ‘AF_PIPE’ (for a Windows named pipe). Of these only the first is guaranteed to be available. If *family* is `None` then the family is inferred from the format of *address*. If *address* is also `None` then a default is chosen. This default is the family which is assumed to be the fastest available. See [Address Formats](#). Note that if *family* is ‘AF_UNIX’ and *address* is `None` then the socket will be created in a private temporary directory created using [tempfile.mkstemp\(\)](#).

If the listener object uses a socket then *backlog* (1 by default) is passed to the [listen\(\)](#) method of the socket once it has been bound.

If *authkey* is given and not `None`, it should be a byte string and will be used as the secret key for an HMAC-based authentication challenge. No authentication is done if *authkey* is `None`. [AuthenticationError](#) is raised if authentication fails. See [Authentication keys](#).

accept()

Accept a connection on the bound socket or named pipe of the listener object and return a [Connection](#) object. If authentication is attempted and fails, then [AuthenticationError](#) is raised.

close()

Close the bound socket or named pipe of the listener object. This is called automatically when the listener is garbage collected. However it is advisable to call it explicitly.

Listener objects have the following read-only properties:

address

The address which is being used by the Listener object.

last_accepted

The address from which the last accepted connection came. If this is unavailable then it is `None`.

New in version 3.3: Listener objects now support the context management protocol – see [Context Manager Types](#). `__enter__()` returns the listener object, and `__exit__()` calls [close\(\)](#).

multiprocessing.connection.wait(*object_list*, *timeout=None*)

Wait till an object in *object_list* is ready. Returns the list of those objects in *object_list* which are ready. If *timeout* is a float then the call blocks for at most that many seconds. If *timeout* is `None` then it will block for an unlimited period. A negative timeout is equivalent to a zero timeout.

For both Unix and Windows, an object can appear in *object_list* if it is

- a readable [Connection](#) object;
- a connected and readable [socket.socket](#) object; or
- the [sentinel](#) attribute of a [Process](#) object.

A connection or socket object is ready when there is data available to be read from it, or the other end has been closed.

Unix: `wait(object_list, timeout)` almost equivalent `select.select(object_list, [], [], timeout)`. The difference is that, if `select.select()` is interrupted by a signal, it can raise `OSError` with an error number of `EINTR`, whereas `wait()` will not.

Windows: An item in `object_list` must either be an integer handle which is waitable (according to the definition used by the documentation of the Win32 function `WaitForMultipleObjects()`) or it can be an object with a `fileno()` method which returns a socket handle or pipe handle. (Note that pipe handles and socket handles are **not** waitable handles.)

New in version 3.3.

Examples

The following server code creates a listener which uses 'secret password' as an authentication key. It then waits for a connection and sends some data to the client:

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)    # family is deduced to be 'AF_INET'

with Listener(address, authkey=b'secret password') as listener:
    with listener.accept() as conn:
        print('connection accepted from', listener.last_accepted)

        conn.send([2.25, None, 'junk', float])

        conn.send_bytes(b'hello')

        conn.send_bytes(array('i', [42, 1729]))
```

The following code connects to the server and receives some data from the server:

```
from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)

with Client(address, authkey=b'secret password') as conn:
    print(conn.recv())           # => [2.25, None, 'junk', float]

    print(conn.recv_bytes())     # => 'hello'

    arr = array('i', [0, 0, 0, 0, 0])
    print(conn.recv_bytes_into(arr)) # => 8
    print(arr)                   # => array('i', [42, 1729, 0, 0, 0])
```

The following code uses `wait()` to wait for messages from multiple processes at once:

```
import time, random
from multiprocessing import Process, Pipe, current_process
from multiprocessing.connection import wait

def foo(w):
```

(continues on next page)

(continued from previous page)

```

for i in range(10):
    w.send((i, current_process().name))
w.close()

if __name__ == '__main__':
    readers = []

    for i in range(4):
        r, w = Pipe(duplex=False)
        readers.append(r)
        p = Process(target=foo, args=(w,))
        p.start()
        # We close the writable end of the pipe now to be sure that
        # p is the only process which owns a handle for it. This
        # ensures that when p closes its handle for the writable end,
        # wait() will promptly report the readable end as being ready.
        w.close()

    while readers:
        for r in wait(readers):
            try:
                msg = r.recv()
            except EOFError:
                readers.remove(r)
            else:
                print(msg)

```

Address Formats

- An 'AF_INET' address is a tuple of the form (hostname, port) where *hostname* is a string and *port* is an integer.
- An 'AF_UNIX' address is a string representing a filename on the filesystem.
- An 'AF_PIPE' address is a string of the form r'\\.pipe{PipeName}'. To use *Client()* to connect to a named pipe on a remote computer called *ServerName* one should use an address of the form r'\\ServerName\\pipe{PipeName}' instead.

Note that any string beginning with two backslashes is assumed by default to be an 'AF_PIPE' address rather than an 'AF_UNIX' address.

Authentication keys

When one uses *Connection.recv*, the data received is automatically unpickled. Unfortunately unpickling data from an untrusted source is a security risk. Therefore *Listener* and *Client()* use the *hmac* module to provide digest authentication.

An authentication key is a byte string which can be thought of as a password: once a connection is established both ends will demand proof that the other knows the authentication key. (Demonstrating that both ends are using the same key does **not** involve sending the key over the connection.)

If authentication is requested but no authentication key is specified then the return value of *current_process().authkey* is used (see *Process*). This value will be automatically inherited by any

Process object that the current process creates. This means that (by default) all processes of a multi-process program will share a single authentication key which can be used when setting up connections between themselves.

Suitable authentication keys can also be generated by using `os.urandom()`.

Logging

Some support for logging is available. Note, however, that the *logging* package does not use process shared locks so it is possible (depending on the handler type) for messages from different processes to get mixed up.

`multiprocessing.get_logger()`

Returns the logger used by *multiprocessing*. If necessary, a new one will be created.

When first created the logger has level `logging.NOTSET` and no default handler. Messages sent to this logger will not by default propagate to the root logger.

Note that on Windows child processes will only inherit the level of the parent process's logger – any other customization of the logger will not be inherited.

`multiprocessing.log_to_stderr()`

This function performs a call to `get_logger()` but in addition to returning the logger created by `get_logger`, it adds a handler which sends output to `sys.stderr` using format `'[% (levelname)s/ %(processName)s] %(message)s'`.

Below is an example session with logging turned on:

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pypm-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

For a full table of logging levels, see the *logging* module.

The `multiprocessing.dummy` module

multiprocessing.dummy replicates the API of *multiprocessing* but is no more than a wrapper around the *threading* module.

17.2.3 Programming guidelines

There are certain guidelines and idioms which should be adhered to when using *multiprocessing*.

All start methods

The following applies to all start methods.

Avoid shared state

As far as possible one should try to avoid shifting large amounts of data between processes.

It is probably best to stick to using queues or pipes for communication between processes rather than using the lower level synchronization primitives.

Picklability

Ensure that the arguments to the methods of proxies are picklable.

Thread safety of proxies

Do not use a proxy object from more than one thread unless you protect it with a lock.

(There is never a problem with different processes using the *same* proxy.)

Joining zombie processes

On Unix when a process finishes but has not been joined it becomes a zombie. There should never be very many because each time a new process starts (or `active_children()` is called) all completed processes which have not yet been joined will be joined. Also calling a finished process's `Process.is_alive` will join the process. Even so it is probably good practice to explicitly join all the processes that you start.

Better to inherit than pickle/unpickle

When using the `spawn` or `forkserver` start methods many types from `multiprocessing` need to be picklable so that child processes can use them. However, one should generally avoid sending shared objects to other processes using pipes or queues. Instead you should arrange the program so that a process which needs access to a shared resource created elsewhere can inherit it from an ancestor process.

Avoid terminating processes

Using the `Process.terminate` method to stop a process is liable to cause any shared resources (such as locks, semaphores, pipes and queues) currently being used by the process to become broken or unavailable to other processes.

Therefore it is probably best to only consider using `Process.terminate` on processes which never use any shared resources.

Joining processes that use queues

Bear in mind that a process that has put items in a queue will wait before terminating until all the buffered items are fed by the “feeder” thread to the underlying pipe. (The child process can call the `Queue.cancel_join_thread` method of the queue to avoid this behaviour.)

This means that whenever you use a queue you need to make sure that all items which have been put on the queue will eventually be removed before the process is joined. Otherwise you cannot be sure that processes which have put items on the queue will terminate. Remember also that non-daemonic processes will be joined automatically.

An example which will deadlock is the following:

```
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
```

(continues on next page)

(continued from previous page)

```
p.join()                # this deadlocks
obj = queue.get()
```

A fix here would be to swap the last two lines (or simply remove the `p.join()` line).

Explicitly pass resources to child processes

On Unix using the *fork* start method, a child process can make use of a shared resource created in a parent process using a global resource. However, it is better to pass the object as an argument to the constructor for the child process.

Apart from making the code (potentially) compatible with Windows and the other start methods this also ensures that as long as the child process is still alive the object will not be garbage collected in the parent process. This might be important if some resource is freed when the object is garbage collected in the parent process.

So for instance

```
from multiprocessing import Process, Lock

def f():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

should be rewritten as

```
from multiprocessing import Process, Lock

def f(l):
    ... do something using "l" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

Beware of replacing `sys.stdin` with a “file like object”

`multiprocessing` originally unconditionally called:

```
os.close(sys.stdin.fileno())
```

in the `multiprocessing.Process._bootstrap()` method — this resulted in issues with processes-in-processes. This has been changed to:

```
sys.stdin.close()
sys.stdin = open(os.open(os.devnull, os.O_RDONLY), closefd=False)
```

Which solves the fundamental issue of processes colliding with each other resulting in a bad file descriptor error, but introduces a potential danger to applications which replace `sys.stdin()` with a “file-like object” with output buffering. This danger is that if multiple processes call `close()` on this file-like object, it could result in the same data being flushed to the object multiple times, resulting in corruption.

If you write a file-like object and implement your own caching, you can make it fork-safe by storing the pid whenever you append to the cache, and discarding the cache when the pid changes. For example:

```
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

For more information, see [bpo-5155](#), [bpo-5313](#) and [bpo-5331](#)

The *spawn* and *forkserver* start methods

There are a few extra restriction which don't apply to the *fork* start method.

More picklability

Ensure that all arguments to `Process.__init__()` are picklable. Also, if you subclass *Process* then make sure that instances will be picklable when the *Process.start* method is called.

Global variables

Bear in mind that if code run in a child process tries to access a global variable, then the value it sees (if any) may not be the same as the value in the parent process at the time that *Process.start* was called.

However, global variables which are just module level constants cause no problems.

Safe importing of main module

Make sure that the main module can be safely imported by a new Python interpreter without causing unintended side effects (such a starting a new process).

For example, using the *spawn* or *forkserver* start method running the following module would fail with a *RuntimeError*:

```
from multiprocessing import Process

def foo():
    print('hello')

p = Process(target=foo)
p.start()
```

Instead one should protect the “entry point” of the program by using `if __name__ == '__main__':` as follows:

```
from multiprocessing import Process, freeze_support, set_start_method

def foo():
    print('hello')

if __name__ == '__main__':
    freeze_support()
    set_start_method('spawn')
```

(continues on next page)

(continued from previous page)

```
p = Process(target=foo)
p.start()
```

(The `freeze_support()` line can be omitted if the program will be run normally instead of frozen.)

This allows the newly spawned Python interpreter to safely import the module and then run the module's `foo()` function.

Similar restrictions apply if a pool or manager is created in the main module.

17.2.4 Examples

Demonstration of how to create and use customized managers and proxies:

```
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo:
    def f(self):
        print('you called Foo.f()')
    def g(self):
        print('you called Foo.g()')
    def _h(self):
        print('you called Foo._h()')

# A simple generator function
def baz():
    for i in range(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ['__next__']
    def __iter__(self):
        return self
    def __next__(self):
        return self._callmethod('__next__')

# Function to return the operator module
def get_operator_module():
    return operator

##

class MyManager(BaseManager):
    pass

# register the Foo class; make `f()` and `g()` accessible via proxy
```

(continues on next page)

(continued from previous page)

```

MyManager.register('Foo1', Foo)

# register the Foo class; make `g()` and `_h()` accessible via proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

# register the generator function baz; use `GeneratorProxy` to make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions accessible via proxy
MyManager.register('operator', get_operator_module)

##

def test():
    manager = MyManager()
    manager.start()

    print('-' * 20)

    f1 = manager.Foo1()
    f1.f()
    f1.g()
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print('-' * 20)

    f2 = manager.Foo2()
    f2.g()
    f2._h()
    assert not hasattr(f2, 'f')
    assert sorted(f2._exposed_) == sorted(['g', '_h'])

    print('-' * 20)

    it = manager.baz()
    for i in it:
        print('<%d>' % i, end=' ')
    print()

    print('-' * 20)

    op = manager.operator()
    print('op.add(23, 45) =', op.add(23, 45))
    print('op.pow(2, 94) =', op.pow(2, 94))
    print('op._exposed_ =', op._exposed_)

##

if __name__ == '__main__':
    freeze_support()
    test()

```

Using *Pool*:

```
import multiprocessing
import time
import random
import sys

#
# Functions used by test code
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % (
        multiprocessing.current_process().name,
        func.__name__, args, result
    )

def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5 * random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5 * random.random())
    return a + b

def f(x):
    return 1.0 / (x - 5.0)

def pow3(x):
    return x ** 3

def noop(x):
    pass

#
# Test code
#

def test():
    PROCESSES = 4
    print('Creating pool with %d processes\n' % PROCESSES)

    with multiprocessing.Pool(PROCESSES) as pool:
        #
        # Tests
        #

        TASKS = [(mul, (i, 7)) for i in range(10)] + \
            [(plus, (i, 8)) for i in range(10)]
```

(continues on next page)

(continued from previous page)

```

results = [pool.apply_async(calculate, t) for t in TASKS]
imap_it = pool.imap(calculatestar, TASKS)
imap_unordered_it = pool.imap_unordered(calculatestar, TASKS)

print('Ordered results using pool.apply_async():')
for r in results:
    print('\t', r.get())
print()

print('Ordered results using pool.imap():')
for x in imap_it:
    print('\t', x)
print()

print('Unordered results using pool.imap_unordered():')
for x in imap_unordered_it:
    print('\t', x)
print()

print('Ordered results using pool.map() --- will block till complete:')
for x in pool.map(calculatestar, TASKS):
    print('\t', x)
print()

#
# Test error handling
#

print('Testing error handling:')

try:
    print(pool.apply(f, (5,)))
except ZeroDivisionError:
    print('\tGot ZeroDivisionError as expected from pool.apply()')
else:
    raise AssertionError('expected ZeroDivisionError')

try:
    print(pool.map(f, list(range(10))))
except ZeroDivisionError:
    print('\tGot ZeroDivisionError as expected from pool.map()')
else:
    raise AssertionError('expected ZeroDivisionError')

try:
    print(list(pool.imap(f, list(range(10)))))
except ZeroDivisionError:
    print('\tGot ZeroDivisionError as expected from list(pool.imap())')
else:
    raise AssertionError('expected ZeroDivisionError')

```

(continues on next page)

(continued from previous page)

```

it = pool.imap(f, list(range(10)))
for i in range(10):
    try:
        x = next(it)
    except ZeroDivisionError:
        if i == 5:
            pass
    except StopIteration:
        break
    else:
        if i == 5:
            raise AssertionError('expected ZeroDivisionError')

assert i == 9
print('\tGot ZeroDivisionError as expected from IMapIterator.next()')
print()

#
# Testing timeouts
#

print('Testing ApplyResult.get() with timeout:', end=' ')
res = pool.apply_async(calculate, TASKS[0])
while 1:
    sys.stdout.flush()
    try:
        sys.stdout.write('\n\t%s' % res.get(0.02))
        break
    except multiprocessing.TimeoutError:
        sys.stdout.write('.')
print()
print()

print('Testing IMapIterator.next() with timeout:', end=' ')
it = pool.imap(calculatestar, TASKS)
while 1:
    sys.stdout.flush()
    try:
        sys.stdout.write('\n\t%s' % it.next(0.02))
    except StopIteration:
        break
    except multiprocessing.TimeoutError:
        sys.stdout.write('.')
print()
print()

if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()

```

An example showing how to use queues to feed tasks to a collection of worker processes and collect the results:

```
import time
import random

from multiprocessing import Process, Queue, current_process, freeze_support

#
# Function run by worker processes
#

def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

#
# Function used to calculate result
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % \
        (current_process().name, func.__name__, args, result)

#
# Functions referenced by tasks
#

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

#
#
#

def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # Create queues
    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)
```

(continues on next page)

(continued from previous page)

```

# Start worker processes
for i in range(NUMBER_OF_PROCESSES):
    Process(target=worker, args=(task_queue, done_queue)).start()

# Get and print results
print('Unordered results:')
for i in range(len(TASKS1)):
    print('\t', done_queue.get())

# Add more tasks using `put()`
for task in TASKS2:
    task_queue.put(task)

# Get and print some more results
for i in range(len(TASKS2)):
    print('\t', done_queue.get())

# Tell child processes to stop
for i in range(NUMBER_OF_PROCESSES):
    task_queue.put('STOP')

if __name__ == '__main__':
    freeze_support()
    test()

```

17.3 The concurrent package

Currently, there is only one module in this package:

- `concurrent.futures` – Launching parallel tasks

17.4 `concurrent.futures` — Launching parallel tasks

New in version 3.2.

Source code: `Lib/concurrent/futures/thread.py` and `Lib/concurrent/futures/process.py`

The `concurrent.futures` module provides a high-level interface for asynchronously executing callables.

The asynchronous execution can be performed with threads, using `ThreadPoolExecutor`, or separate processes, using `ProcessPoolExecutor`. Both implement the same interface, which is defined by the abstract `Executor` class.

17.4.1 Executor Objects

class `concurrent.futures.Executor`

An abstract class that provides methods to execute calls asynchronously. It should not be used directly, but through its concrete subclasses.

submit(*fn*, **args*, ***kwargs*)

Schedules the callable, *fn*, to be executed as *fn*(**args* ***kwargs*) and returns a *Future* object representing the execution of the callable.

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

map(*func*, **iterables*, *timeout=None*, *chunksize=1*)

Similar to *map(func, *iterables)* except:

- the *iterables* are collected immediately rather than lazily;
- *func* is executed asynchronously and several calls to *func* may be made concurrently.

The returned iterator raises a *concurrent.futures.TimeoutError* if *__next__()* is called and the result isn't available after *timeout* seconds from the original call to *Executor.map()*. *timeout* can be an int or a float. If *timeout* is not specified or *None*, there is no limit to the wait time.

If a *func* call raises an exception, then that exception will be raised when its value is retrieved from the iterator.

When using *ProcessPoolExecutor*, this method chops *iterables* into a number of chunks which it submits to the pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer. For very long iterables, using a large value for *chunksize* can significantly improve performance compared to the default size of 1. With *ThreadPoolExecutor*, *chunksize* has no effect.

Changed in version 3.5: Added the *chunksize* argument.

shutdown(*wait=True*)

Signal the executor that it should free any resources that it is using when the currently pending futures are done executing. Calls to *Executor.submit()* and *Executor.map()* made after shutdown will raise *RuntimeError*.

If *wait* is *True* then this method will not return until all the pending futures are done executing and the resources associated with the executor have been freed. If *wait* is *False* then this method will return immediately and the resources associated with the executor will be freed when all pending futures are done executing. Regardless of the value of *wait*, the entire Python program will not exit until all pending futures are done executing.

You can avoid having to call this method explicitly if you use the *with* statement, which will shutdown the *Executor* (waiting as if *Executor.shutdown()* were called with *wait* set to *True*):

```
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

17.4.2 ThreadPoolExecutor

ThreadPoolExecutor is an *Executor* subclass that uses a pool of threads to execute calls asynchronously.

Deadlocks can occur when the callable associated with a *Future* waits on the results of another *Future*. For example:

```
import time
def wait_on_b():
    time.sleep(5)
    print(b.result()) # b will never complete because it is waiting on a.
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result()) # a will never complete because it is waiting on b.
    return 6

executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
```

And:

```
def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # This will never complete because there is only one worker thread and
    # it is executing this function.
    print(f.result())

executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```

```
class concurrent.futures.ThreadPoolExecutor(max_workers=None, thread_name_prefix="",
                                             initializer=None, initargs=())
```

An *Executor* subclass that uses a pool of at most *max_workers* threads to execute calls asynchronously.

initializer is an optional callable that is called at the start of each worker thread; *initargs* is a tuple of arguments passed to the initializer. Should *initializer* raise an exception, all currently pending jobs will raise a *BrokenThreadPool*, as well as any attempt to submit more jobs to the pool.

Changed in version 3.5: If *max_workers* is *None* or not given, it will default to the number of processors on the machine, multiplied by 5, assuming that *ThreadPoolExecutor* is often used to overlap I/O instead of CPU work and the number of workers should be higher than the number of workers for *ProcessPoolExecutor*.

New in version 3.6: The *thread_name_prefix* argument was added to allow users to control the *threading.Thread* names for worker threads created by the pool for easier debugging.

Changed in version 3.7: Added the *initializer* and *initargs* arguments.

ThreadPoolExecutor Example

```
import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/']
```

(continues on next page)

(continued from previous page)

```

    'http://some-made-up-domain.com/']

# Retrieve a single page and report the URL and contents
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

# We can use a with statement to ensure threads are cleaned up promptly
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    # Start the load operations and mark each future with its URL
    future_to_url = {executor.submit(load_url, url, 60): url for url in URLs}
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
        else:
            print('%r page is %d bytes' % (url, len(data)))

```

17.4.3 ProcessPoolExecutor

The *ProcessPoolExecutor* class is an *Executor* subclass that uses a pool of processes to execute calls asynchronously. *ProcessPoolExecutor* uses the *multiprocessing* module, which allows it to side-step the *Global Interpreter Lock* but also means that only picklable objects can be executed and returned.

The `__main__` module must be importable by worker subprocesses. This means that *ProcessPoolExecutor* will not work in the interactive interpreter.

Calling *Executor* or *Future* methods from a callable submitted to a *ProcessPoolExecutor* will result in deadlock.

```
class concurrent.futures.ProcessPoolExecutor(max_workers=None, mp_context=None, ini-
                                             tializer=None, initargs=())
```

An *Executor* subclass that executes calls asynchronously using a pool of at most *max_workers* processes. If *max_workers* is *None* or not given, it will default to the number of processors on the machine. If *max_workers* is lower or equal to 0, then a *ValueError* will be raised. *mp_context* can be a multiprocessing context or *None*. It will be used to launch the workers. If *mp_context* is *None* or not given, the default multiprocessing context is used.

initializer is an optional callable that is called at the start of each worker process; *initargs* is a tuple of arguments passed to the initializer. Should *initializer* raise an exception, all currently pending jobs will raise a *BrokenProcessPool*, as well any attempt to submit more jobs to the pool.

Changed in version 3.3: When one of the worker processes terminates abruptly, a *BrokenProcessPool* error is now raised. Previously, behaviour was undefined but operations on the executor or its futures would often freeze or deadlock.

Changed in version 3.7: The *mp_context* argument was added to allow users to control the `start_method` for worker processes created by the pool.

Added the *initializer* and *initargs* arguments.

ProcessPoolExecutor Example

```
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print('%d is prime: %s' % (number, prime))

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

17.4.4 Future Objects

The *Future* class encapsulates the asynchronous execution of a callable. *Future* instances are created by *Executor.submit()*.

class concurrent.futures.Future

Encapsulates the asynchronous execution of a callable. *Future* instances are created by *Executor.submit()* and should not be created directly except for testing.

cancel()

Attempt to cancel the call. If the call is currently being executed and cannot be cancelled then the method will return **False**, otherwise the call will be cancelled and the method will return **True**.

cancelled()

Return **True** if the call was successfully cancelled.

running()

Return **True** if the call is currently being executed and cannot be cancelled.

done()

Return **True** if the call was successfully cancelled or finished running.

result(timeout=None)

Return the value returned by the call. If the call hasn't yet completed then this method

will wait up to *timeout* seconds. If the call hasn't completed in *timeout* seconds, then a `concurrent.futures.TimeoutError` will be raised. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If the future is cancelled before completing then `CancelledError` will be raised.

If the call raised, this method will raise the same exception.

exception(*timeout=None*)

Return the exception raised by the call. If the call hasn't yet completed then this method will wait up to *timeout* seconds. If the call hasn't completed in *timeout* seconds, then a `concurrent.futures.TimeoutError` will be raised. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If the future is cancelled before completing then `CancelledError` will be raised.

If the call completed without raising, `None` is returned.

add_done_callback(*fn*)

Attaches the callable *fn* to the future. *fn* will be called, with the future as its only argument, when the future is cancelled or finishes running.

Added callables are called in the order that they were added and are always called in a thread belonging to the process that added them. If the callable raises an `Exception` subclass, it will be logged and ignored. If the callable raises a `BaseException` subclass, the behavior is undefined.

If the future has already completed or been cancelled, *fn* will be called immediately.

The following `Future` methods are meant for use in unit tests and `Executor` implementations.

set_running_or_notify_cancel()

This method should only be called by `Executor` implementations before executing the work associated with the `Future` and by unit tests.

If the method returns `False` then the `Future` was cancelled, i.e. `Future.cancel()` was called and returned `True`. Any threads waiting on the `Future` completing (i.e. through `as_completed()` or `wait()`) will be woken up.

If the method returns `True` then the `Future` was not cancelled and has been put in the running state, i.e. calls to `Future.running()` will return `True`.

This method can only be called once and cannot be called after `Future.set_result()` or `Future.set_exception()` have been called.

set_result(*result*)

Sets the result of the work associated with the `Future` to *result*.

This method should only be used by `Executor` implementations and unit tests.

set_exception(*exception*)

Sets the result of the work associated with the `Future` to the `Exception` *exception*.

This method should only be used by `Executor` implementations and unit tests.

17.4.5 Module Functions

concurrent.futures.wait(*fs*, *timeout=None*, *return_when=ALL_COMPLETED*)

Wait for the `Future` instances (possibly created by different `Executor` instances) given by *fs* to complete. Returns a named 2-tuple of sets. The first set, named `done`, contains the futures that completed (finished or were cancelled) before the wait completed. The second set, named `not_done`, contains uncompleted futures.

timeout can be used to control the maximum number of seconds to wait before returning. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

return_when indicates when this function should return. It must be one of the following constants:

Constant	Description
<code>FIRST_COMPLETED</code>	The function will return when any future finishes or is cancelled.
<code>FIRST_EXCEPTION</code>	The function will return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to <code>ALL_COMPLETED</code> .
<code>ALL_COMPLETED</code>	The function will return when all futures finish or are cancelled.

`concurrent.futures.as_completed(fs, timeout=None)`

Returns an iterator over the *Future* instances (possibly created by different *Executor* instances) given by *fs* that yields futures as they complete (finished or were cancelled). Any futures given by *fs* that are duplicated will be returned once. Any futures that completed before `as_completed()` is called will be yielded first. The returned iterator raises a `concurrent.futures.TimeoutError` if `__next__()` is called and the result isn't available after *timeout* seconds from the original call to `as_completed()`. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

See also:

PEP 3148 – futures - execute computations asynchronously The proposal which described this feature for inclusion in the Python standard library.

17.4.6 Exception classes

exception `concurrent.futures.CancelledError`

Raised when a future is cancelled.

exception `concurrent.futures.TimeoutError`

Raised when a future operation exceeds the given timeout.

exception `concurrent.futures.BrokenExecutor`

Derived from *RuntimeError*, this exception class is raised when an executor is broken for some reason, and cannot be used to submit or execute new tasks.

New in version 3.7.

exception `concurrent.futures.thread.BrokenThreadPool`

Derived from *BrokenExecutor*, this exception class is raised when one of the workers of a *ThreadPoolExecutor* has failed initializing.

New in version 3.7.

exception `concurrent.futures.process.BrokenProcessPool`

Derived from *BrokenExecutor* (formerly *RuntimeError*), this exception class is raised when one of the workers of a *ProcessPoolExecutor* has terminated in a non-clean fashion (for example, if it was killed from the outside).

New in version 3.3.

17.5 subprocess — Subprocess management

Source code: [Lib/subprocess.py](#)

The *subprocess* module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This module intends to replace several older modules and functions:

```
os.system
os.spawn*
```

Information about how the `subprocess` module can be used to replace these modules and functions can be found in the following sections.

See also:

[PEP 324](#) – PEP proposing the subprocess module

17.5.1 Using the subprocess Module

The recommended approach to invoking subprocesses is to use the `run()` function for all use cases it can handle. For more advanced use cases, the underlying `Popen` interface can be used directly.

The `run()` function was added in Python 3.5; if you need to retain compatibility with older versions, see the *Older high-level API* section.

```
subprocess.run(args, *, stdin=None, input=None, stdout=None, stderr=None, capture_output=False, shell=False, cwd=None, timeout=None, check=False, encoding=None, errors=None, text=None, env=None, universal_newlines=None)
```

Run the command described by `args`. Wait for command to complete, then return a `CompletedProcess` instance.

The arguments shown above are merely the most common ones, described below in *Frequently Used Arguments* (hence the use of keyword-only notation in the abbreviated signature). The full function signature is largely the same as that of the `Popen` constructor - most of the arguments to this function are passed through to that interface. (`timeout`, `input`, `check`, and `capture_output` are not.)

If `capture_output` is true, `stdout` and `stderr` will be captured. When used, the internal `Popen` object is automatically created with `stdout=PIPE` and `stderr=PIPE`. The `stdout` and `stderr` arguments may not be used as well.

The `timeout` argument is passed to `Popen.communicate()`. If the timeout expires, the child process will be killed and waited for. The `TimeoutExpired` exception will be re-raised after the child process has terminated.

The `input` argument is passed to `Popen.communicate()` and thus to the subprocess's `stdin`. If used it must be a byte sequence, or a string if `encoding` or `errors` is specified or `text` is true. When used, the internal `Popen` object is automatically created with `stdin=PIPE`, and the `stdin` argument may not be used as well.

If `check` is true, and the process exits with a non-zero exit code, a `CalledProcessError` exception will be raised. Attributes of that exception hold the arguments, the exit code, and `stdout` and `stderr` if they were captured.

If `encoding` or `errors` are specified, or `text` is true, file objects for `stdin`, `stdout` and `stderr` are opened in text mode using the specified `encoding` and `errors` or the `io.TextIOWrapper` default. The `universal_newlines` argument is equivalent to `text` and is provided for backwards compatibility. By default, file objects are opened in binary mode.

If `env` is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process' environment. It is passed directly to `Popen`.

Examples:

```
>>> subprocess.run(["ls", "-l"]) # doesn't capture output
CompletedProcess(args=['ls', '-l'], returncode=0)
```

(continues on next page)

(continued from previous page)

```

>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1

>>> subprocess.run(["ls", "-l", "/dev/null"], capture_output=True)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n', stderr=b'')

```

New in version 3.5.

Changed in version 3.6: Added *encoding* and *errors* parameters

Changed in version 3.7: Added the *text* parameter, as a more understandable alias of *universal_newlines*. Added the *capture_output* parameter.

class subprocess.CompletedProcess

The return value from `run()`, representing a process that has finished.

args

The arguments used to launch the process. This may be a list or a string.

returncode

Exit status of the child process. Typically, an exit status of 0 indicates that it ran successfully.

A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).

stdout

Captured stdout from the child process. A bytes sequence, or a string if `run()` was called with an encoding, errors, or `text=True`. `None` if stdout was not captured.

If you ran the process with `stderr=subprocess.STDOUT`, stdout and stderr will be combined in this attribute, and `stderr` will be `None`.

stderr

Captured stderr from the child process. A bytes sequence, or a string if `run()` was called with an encoding, errors, or `text=True`. `None` if stderr was not captured.

check_returncode()

If `returncode` is non-zero, raise a `CalledProcessError`.

New in version 3.5.

subprocess.DEVNULL

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to `Popen` and indicates that the special file `os.devnull` will be used.

New in version 3.3.

subprocess.PIPE

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to `Popen` and indicates that a pipe to the standard stream should be opened. Most useful with `Popen.communicate()`.

subprocess.STDOUT

Special value that can be used as the *stderr* argument to `Popen` and indicates that standard error should go into the same handle as standard output.

exception subprocess.SubprocessError

Base class for all other exceptions from this module.

New in version 3.3.

exception subprocess.TimeoutExpired

Subclass of *SubprocessError*, raised when a timeout expires while waiting for a child process.

cmd

Command that was used to spawn the child process.

timeout

Timeout in seconds.

output

Output of the child process if it was captured by *run()* or *check_output()*. Otherwise, *None*.

stdout

Alias for output, for symmetry with *stderr*.

stderr

Stderr output of the child process if it was captured by *run()*. Otherwise, *None*.

New in version 3.3.

Changed in version 3.5: *stdout* and *stderr* attributes added

exception subprocess.CallProcessError

Subclass of *SubprocessError*, raised when a process run by *check_call()* or *check_output()* returns a non-zero exit status.

returncode

Exit status of the child process. If the process exited due to a signal, this will be the negative signal number.

cmd

Command that was used to spawn the child process.

output

Output of the child process if it was captured by *run()* or *check_output()*. Otherwise, *None*.

stdout

Alias for output, for symmetry with *stderr*.

stderr

Stderr output of the child process if it was captured by *run()*. Otherwise, *None*.

Changed in version 3.5: *stdout* and *stderr* attributes added

Frequently Used Arguments

To support a wide variety of use cases, the *Popen* constructor (and the convenience functions) accept a large number of optional arguments. For most typical use cases, many of these arguments can be safely left at their default values. The arguments that are most commonly needed are:

args is required for all calls and should be a string, or a sequence of program arguments. Providing a sequence of arguments is generally preferred, as it allows the module to take care of any required escaping and quoting of arguments (e.g. to permit spaces in file names). If passing a single string, either *shell* must be *True* (see below) or else the string must simply name the program to be executed without specifying any arguments.

stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are *PIPE*, *DEVNULL*, an existing file descriptor (a positive integer), an existing file object, and *None*. *PIPE* indicates that a new pipe to the child should be created. *DEVNULL* indicates that the special file *os.devnull* will be used. With the default settings of *None*, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, *stderr* can be *STDOUT*, which indicates that the stderr data from the child process should be captured into the same file handle as for *stdout*.

If *encoding* or *errors* are specified, or *text* (also known as *universal_newlines*) is true, the file objects *stdin*, *stdout* and *stderr* will be opened in text mode using the *encoding* and *errors* specified in the call or the defaults for *io.TextIOWrapper*.

For *stdin*, line ending characters '\n' in the input will be converted to the default line separator *os.linesep*. For *stdout* and *stderr*, all line endings in the output will be converted to '\n'. For more information see the documentation of the *io.TextIOWrapper* class when the *newline* argument to its constructor is *None*.

If text mode is not used, *stdin*, *stdout* and *stderr* will be opened as binary streams. No encoding or line ending conversion is performed.

New in version 3.6: Added *encoding* and *errors* parameters.

New in version 3.7: Added the *text* parameter as an alias for *universal_newlines*.

Note: The *newlines* attribute of the file objects *Popen.stdin*, *Popen.stdout* and *Popen.stderr* are not updated by the *Popen.communicate()* method.

If *shell* is *True*, the specified command will be executed through the shell. This can be useful if you are using Python primarily for the enhanced control flow it offers over most system shells and still want convenient access to other shell features such as shell pipes, filename wildcards, environment variable expansion, and expansion of ~ to a user's home directory. However, note that Python itself offers implementations of many shell-like features (in particular, *glob*, *fnmatch*, *os.walk()*, *os.path.expandvars()*, *os.path.expanduser()*, and *shutil*).

Changed in version 3.3: When *universal_newlines* is *True*, the class uses the encoding *locale.getpreferredencoding(False)* instead of *locale.getpreferredencoding()*. See the *io.TextIOWrapper* class for more information on this change.

Note: Read the *Security Considerations* section before using *shell=True*.

These options, along with all of the other options, are described in more detail in the *Popen* constructor documentation.

Popen Constructor

The underlying process creation and management in this module is handled by the *Popen* class. It offers a lot of flexibility so that developers are able to handle the less common cases not covered by the convenience functions.

```
class subprocess.Popen(args, bufsize=-1, executable=None, stdin=None, stdout=None,
                        stderr=None, preexec_fn=None, close_fds=True, shell=False, cwd=None,
                        env=None, universal_newlines=None, startupinfo=None, creationflags=0,
                        restore_signals=True, start_new_session=False, pass_fds=(), *, encoding=None, errors=None, text=None)
```

Execute a child program in a new process. On POSIX, the class uses *os.execvp()*-like behavior to execute the child program. On Windows, the class uses the Windows *CreateProcess()* function. The arguments to *Popen* are as follows.

args should be a sequence of program arguments or else a single string. By default, the program to execute is the first item in *args* if *args* is a sequence. If *args* is a string, the interpretation is platform-dependent and described below. See the *shell* and *executable* arguments for additional differences from the default behavior. Unless otherwise stated, it is recommended to pass *args* as a sequence.

On POSIX, if *args* is a string, the string is interpreted as the name or path of the program to execute. However, this can only be done if not passing arguments to the program.

Note: `shlex.split()` can be useful when determining the correct tokenization for `args`, especially in complex cases:

```
>>> import shlex, subprocess
>>> command_line = input()
/bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
>>> args = shlex.split(command_line)
>>> print(args)
['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd', 'echo '
↪ '$MONEY'']
>>> p = subprocess.Popen(args) # Success!
```

Note in particular that options (such as `-input`) and arguments (such as `eggs.txt`) that are separated by whitespace in the shell go in separate list elements, while arguments that need quoting or backslash escaping when used in the shell (such as filenames containing spaces or the `echo` command shown above) are single list elements.

On Windows, if `args` is a sequence, it will be converted to a string in a manner described in [Converting an argument sequence to a string on Windows](#). This is because the underlying `CreateProcess()` operates on strings.

The `shell` argument (which defaults to `False`) specifies whether to use the shell as the program to execute. If `shell` is `True`, it is recommended to pass `args` as a string rather than as a sequence.

On POSIX with `shell=True`, the shell defaults to `/bin/sh`. If `args` is a string, the string specifies the command to execute through the shell. This means that the string must be formatted exactly as it would be when typed at the shell prompt. This includes, for example, quoting or backslash escaping filenames with spaces in them. If `args` is a sequence, the first item specifies the command string, and any additional items will be treated as additional arguments to the shell itself. That is to say, `Popen` does the equivalent of:

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

On Windows with `shell=True`, the `COMSPEC` environment variable specifies the default shell. The only time you need to specify `shell=True` on Windows is when the command you wish to execute is built into the shell (e.g. `dir` or `copy`). You do not need `shell=True` to run a batch file or console-based executable.

Note: Read the [Security Considerations](#) section before using `shell=True`.

`bufsize` will be supplied as the corresponding argument to the `open()` function when creating the `stdin/stdout/stderr` pipe file objects:

- 0 means unbuffered (read and write are one system call and can return short)
- 1 means line buffered (only usable if `universal_newlines=True` i.e., in a text mode)
- any other positive value means use a buffer of approximately that size
- negative `bufsize` (the default) means the system default of `io.DEFAULT_BUFFER_SIZE` will be used.

Changed in version 3.3.1: `bufsize` now defaults to -1 to enable buffering by default to match the behavior that most code expects. In versions prior to Python 3.2.4 and 3.3.1 it incorrectly defaulted to 0 which was unbuffered and allowed short reads. This was unintentional and did not match the behavior of Python 2 as most code expected.

The *executable* argument specifies a replacement program to execute. It is very seldom needed. When *shell=False*, *executable* replaces the program to execute specified by *args*. However, the original *args* is still passed to the program. Most programs treat the program specified by *args* as the command name, which can then be different from the program actually executed. On POSIX, the *args* name becomes the display name for the executable in utilities such as *ps*. If *shell=True*, on POSIX the *executable* argument specifies a replacement shell for the default */bin/sh*.

stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are *PIPE*, *DEVNULL*, an existing file descriptor (a positive integer), an existing *file object*, and *None*. *PIPE* indicates that a new pipe to the child should be created. *DEVNULL* indicates that the special file *os.devnull* will be used. With the default settings of *None*, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, *stderr* can be *STDOUT*, which indicates that the *stderr* data from the applications should be captured into the same file handle as for *stdout*.

If *preexec_fn* is set to a callable object, this object will be called in the child process just before the child is executed. (POSIX only)

Warning: The *preexec_fn* parameter is not safe to use in the presence of threads in your application. The child process could deadlock before *exec* is called. If you must use it, keep it trivial! Minimize the number of libraries you call into.

Note: If you need to modify the environment for the child use the *env* parameter rather than doing it in a *preexec_fn*. The *start_new_session* parameter can take the place of a previously common use of *preexec_fn* to call *os.setsid()* in the child.

If *close_fds* is true, all file descriptors except 0, 1 and 2 will be closed before the child process is executed. Otherwise when *close_fds* is false, file descriptors obey their inheritable flag as described in *Inheritance of File Descriptors*.

On Windows, if *close_fds* is true then no handles will be inherited by the child process unless explicitly passed in the *handle_list* element of *STARTUPINFO.lpAttributeList*, or by standard handle redirection.

Changed in version 3.2: The default for *close_fds* was changed from *False* to what is described above.

Changed in version 3.7: On Windows the default for *close_fds* was changed from *False* to *True* when redirecting the standard handles. It's now possible to set *close_fds* to *True* when redirecting the standard handles.

pass_fds is an optional sequence of file descriptors to keep open between the parent and child. Providing any *pass_fds* forces *close_fds* to be *True*. (POSIX only)

New in version 3.2: The *pass_fds* parameter was added.

If *cwd* is not *None*, the function changes the working directory to *cwd* before executing the child. *cwd* can be a *str* and *path-like object*. In particular, the function looks for *executable* (or for the first item in *args*) relative to *cwd* if the executable path is a relative path.

Changed in version 3.6: *cwd* parameter accepts a *path-like object*.

If *restore_signals* is true (the default) all signals that Python has set to SIG_IGN are restored to SIG_DFL in the child process before the *exec*. Currently this includes the SIGPIPE, SIGXFZ and SIGXFSZ signals. (POSIX only)

Changed in version 3.2: *restore_signals* was added.

If *start_new_session* is true the `setsid()` system call will be made in the child process prior to the execution of the subprocess. (POSIX only)

Changed in version 3.2: *start_new_session* was added.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process' environment.

Note: If specified, *env* must provide any variables required for the program to execute. On Windows, in order to run a [side-by-side assembly](#) the specified *env* **must** include a valid `SystemRoot`.

If *encoding* or *errors* are specified, or *text* is true, the file objects *stdin*, *stdout* and *stderr* are opened in text mode with the specified encoding and *errors*, as described above in [Frequently Used Arguments](#). The *universal_newlines* argument is equivalent to *text* and is provided for backwards compatibility. By default, file objects are opened in binary mode.

New in version 3.6: *encoding* and *errors* were added.

New in version 3.7: *text* was added as a more readable alias for *universal_newlines*.

If given, *startupinfo* will be a `STARTUPINFO` object, which is passed to the underlying `CreateProcess` function. *creationflags*, if given, can be one or more of the following flags:

- `CREATE_NEW_CONSOLE`
- `CREATE_NEW_PROCESS_GROUP`
- `ABOVE_NORMAL_PRIORITY_CLASS`
- `BELOW_NORMAL_PRIORITY_CLASS`
- `HIGH_PRIORITY_CLASS`
- `IDLE_PRIORITY_CLASS`
- `NORMAL_PRIORITY_CLASS`
- `REALTIME_PRIORITY_CLASS`
- `CREATE_NO_WINDOW`
- `DETACHED_PROCESS`
- `CREATE_DEFAULT_ERROR_MODE`
- `CREATE_BREAKAWAY_FROM_JOB`

Popen objects are supported as context managers via the `with` statement: on exit, standard file descriptors are closed, and the process is waited for.

```
with Popen(["ifconfig"], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

Changed in version 3.2: Added context manager support.

Changed in version 3.6: Popen destructor now emits a [ResourceWarning](#) warning if the child process is still running.

Exceptions

Exceptions raised in the child process, before the new program has started to execute, will be re-raised in the parent.

The most common exception raised is [OSError](#). This occurs, for example, when trying to execute a non-existent file. Applications should prepare for [OSError](#) exceptions.

A `ValueError` will be raised if `Popen` is called with invalid arguments.

`check_call()` and `check_output()` will raise `CalledProcessError` if the called process returns a non-zero return code.

All of the functions and methods that accept a `timeout` parameter, such as `call()` and `Popen.communicate()` will raise `TimeoutExpired` if the timeout expires before the process exits.

Exceptions defined in this module all inherit from `SubprocessError`.

New in version 3.3: The `SubprocessError` base class was added.

17.5.2 Security Considerations

Unlike some other `popen` functions, this implementation will never implicitly call a system shell. This means that all characters, including shell metacharacters, can safely be passed to child processes. If the shell is invoked explicitly, via `shell=True`, it is the application's responsibility to ensure that all whitespace and metacharacters are quoted appropriately to avoid [shell injection](#) vulnerabilities.

When using `shell=True`, the `shlex.quote()` function can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

17.5.3 Popen Objects

Instances of the `Popen` class have the following methods:

`Popen.poll()`

Check if child process has terminated. Set and return `returncode` attribute. Otherwise, returns `None`.

`Popen.wait(timeout=None)`

Wait for child process to terminate. Set and return `returncode` attribute.

If the process does not terminate after `timeout` seconds, raise a `TimeoutExpired` exception. It is safe to catch this exception and retry the wait.

Note: This will deadlock when using `stdout=PIPE` or `stderr=PIPE` and the child process generates enough output to a pipe such that it blocks waiting for the OS pipe buffer to accept more data. Use `Popen.communicate()` when using pipes to avoid that.

Note: The function is implemented using a busy loop (non-blocking call and short sleeps). Use the `asyncio` module for an asynchronous wait: see `asyncio.create_subprocess_exec`.

Changed in version 3.3: `timeout` was added.

`Popen.communicate(input=None, timeout=None)`

Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate. The optional `input` argument should be data to be sent to the child process, or `None`, if no data should be sent to the child. If streams were opened in text mode, `input` must be a string. Otherwise, it must be bytes.

`communicate()` returns a tuple (`stdout_data`, `stderr_data`). The data will be strings if streams were opened in text mode; otherwise, bytes.

Note that if you want to send data to the process's stdin, you need to create the `Popen` object with `stdin=PIPE`. Similarly, to get anything other than `None` in the result tuple, you need to give `stdout=PIPE` and/or `stderr=PIPE` too.

If the process does not terminate after *timeout* seconds, a *TimeoutExpired* exception will be raised. Catching this exception and retrying communication will not lose any output.

The child process is not killed if the timeout expires, so in order to cleanup properly a well-behaved application should kill the child process and finish communication:

```
proc = subprocess.Popen(...)
try:
    outs, errs = proc.communicate(timeout=15)
except TimeoutExpired:
    proc.kill()
    outs, errs = proc.communicate()
```

Note: The data read is buffered in memory, so do not use this method if the data size is large or unlimited.

Changed in version 3.3: *timeout* was added.

Popen.send_signal(*signal*)

Sends the signal *signal* to the child.

Note: On Windows, SIGTERM is an alias for *terminate()*. CTRL_C_EVENT and CTRL_BREAK_EVENT can be sent to processes started with a *creationflags* parameter which includes *CREATE_NEW_PROCESS_GROUP*.

Popen.terminate()

Stop the child. On Posix OSs the method sends SIGTERM to the child. On Windows the Win32 API function *TerminateProcess()* is called to stop the child.

Popen.kill()

Kills the child. On Posix OSs the function sends SIGKILL to the child. On Windows *kill()* is an alias for *terminate()*.

The following attributes are also available:

Popen.args

The *args* argument as it was passed to *Popen* – a sequence of program arguments or else a single string. New in version 3.3.

Popen.stdin

If the *stdin* argument was *PIPE*, this attribute is a writeable stream object as returned by *open()*. If the *encoding* or *errors* arguments were specified or the *universal_newlines* argument was *True*, the stream is a text stream, otherwise it is a byte stream. If the *stdin* argument was not *PIPE*, this attribute is *None*.

Popen.stdout

If the *stdout* argument was *PIPE*, this attribute is a readable stream object as returned by *open()*. Reading from the stream provides output from the child process. If the *encoding* or *errors* arguments were specified or the *universal_newlines* argument was *True*, the stream is a text stream, otherwise it is a byte stream. If the *stdout* argument was not *PIPE*, this attribute is *None*.

Popen.stderr

If the *stderr* argument was *PIPE*, this attribute is a readable stream object as returned by *open()*. Reading from the stream provides error output from the child process. If the *encoding* or *errors* arguments were specified or the *universal_newlines* argument was *True*, the stream is a text stream, otherwise it is a byte stream. If the *stderr* argument was not *PIPE*, this attribute is *None*.

Warning: Use `communicate()` rather than `.stdin.write`, `.stdout.read` or `.stderr.read` to avoid deadlocks due to any of the other OS pipe buffers filling up and blocking the child process.

`Popen.pid`

The process ID of the child process.

Note that if you set the `shell` argument to `True`, this is the process ID of the spawned shell.

`Popen.returncode`

The child return code, set by `poll()` and `wait()` (and indirectly by `communicate()`). A `None` value indicates that the process hasn't terminated yet.

A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).

17.5.4 Windows Popen Helpers

The `STARTUPINFO` class and following constants are only available on Windows.

```
class subprocess.STARTUPINFO(*, dwFlags=0, hStdInput=None, hStdOutput=None, hStdError=None, wShowWindow=0, lpAttributeList=None)
```

Partial support of the Windows `STARTUPINFO` structure is used for `Popen` creation. The following attributes can be set by passing them as keyword-only arguments.

Changed in version 3.7: Keyword-only argument support was added.

`dwFlags`

A bit field that determines whether certain `STARTUPINFO` attributes are used when the process creates a window.

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_USESHOWWINDOW
```

`hStdInput`

If `dwFlags` specifies `STARTF_USESTDHANDLES`, this attribute is the standard input handle for the process. If `STARTF_USESTDHANDLES` is not specified, the default for standard input is the keyboard buffer.

`hStdOutput`

If `dwFlags` specifies `STARTF_USESTDHANDLES`, this attribute is the standard output handle for the process. Otherwise, this attribute is ignored and the default for standard output is the console window's buffer.

`hStdError`

If `dwFlags` specifies `STARTF_USESTDHANDLES`, this attribute is the standard error handle for the process. Otherwise, this attribute is ignored and the default for standard error is the console window's buffer.

`wShowWindow`

If `dwFlags` specifies `STARTF_USESHOWWINDOW`, this attribute can be any of the values that can be specified in the `nCmdShow` parameter for the `ShowWindow` function, except for `SW_SHOWDEFAULT`. Otherwise, this attribute is ignored.

`SW_HIDE` is provided for this attribute. It is used when `Popen` is called with `shell=True`.

`lpAttributeList`

A dictionary of additional attributes for process creation as given in `STARTUPINFOEX`, see `UpdateProcThreadAttribute`.

Supported attributes:

handle_list Sequence of handles that will be inherited. *close_fds* must be true if non-empty.

The handles must be temporarily made inheritable by `os.set_handle_inheritable()` when passed to the `Popen` constructor, else `OSError` will be raised with Windows error `ERROR_INVALID_PARAMETER` (87).

Warning: In a multithreaded process, use caution to avoid leaking handles that are marked inheritable when combining this feature with concurrent calls to other process creation functions that inherit all handles such as `os.system()`. This also applies to standard handle redirection, which temporarily creates inheritable handles.

New in version 3.7.

Windows Constants

The `subprocess` module exposes the following constants.

`subprocess.STD_INPUT_HANDLE`

The standard input device. Initially, this is the console input buffer, `CONIN$`.

`subprocess.STD_OUTPUT_HANDLE`

The standard output device. Initially, this is the active console screen buffer, `CONOUT$`.

`subprocess.STD_ERROR_HANDLE`

The standard error device. Initially, this is the active console screen buffer, `CONOUT$`.

`subprocess.SW_HIDE`

Hides the window. Another window will be activated.

`subprocess.STARTF_USESTDHANDLES`

Specifies that the `STARTUPINFO.hStdInput`, `STARTUPINFO.hStdOutput`, and `STARTUPINFO.hStdError` attributes contain additional information.

`subprocess.STARTF_USESHOWWINDOW`

Specifies that the `STARTUPINFO.wShowWindow` attribute contains additional information.

`subprocess.CREATE_NEW_CONSOLE`

The new process has a new console, instead of inheriting its parent's console (the default).

`subprocess.CREATE_NEW_PROCESS_GROUP`

A `Popen` `creationflags` parameter to specify that a new process group will be created. This flag is necessary for using `os.kill()` on the subprocess.

This flag is ignored if `CREATE_NEW_CONSOLE` is specified.

`subprocess.ABOVE_NORMAL_PRIORITY_CLASS`

A `Popen` `creationflags` parameter to specify that a new process will have an above average priority.

New in version 3.7.

`subprocess.BELOW_NORMAL_PRIORITY_CLASS`

A `Popen` `creationflags` parameter to specify that a new process will have a below average priority.

New in version 3.7.

`subprocess.HIGH_PRIORITY_CLASS`

A `Popen` `creationflags` parameter to specify that a new process will have a high priority.

New in version 3.7.

subprocess.IDLE_PRIORITY_CLASS

A *Popen* `creationflags` parameter to specify that a new process will have an idle (lowest) priority.

New in version 3.7.

subprocess.NORMAL_PRIORITY_CLASS

A *Popen* `creationflags` parameter to specify that a new process will have an normal priority. (default)

New in version 3.7.

subprocess.REALTIME_PRIORITY_CLASS

A *Popen* `creationflags` parameter to specify that a new process will have realtime priority. You should almost never use `REALTIME_PRIORITY_CLASS`, because this interrupts system threads that manage mouse input, keyboard input, and background disk flushing. This class can be appropriate for applications that “talk” directly to hardware or that perform brief tasks that should have limited interruptions.

New in version 3.7.

subprocess.CREATE_NO_WINDOW

A *Popen* `creationflags` parameter to specify that a new process will not create a window

New in version 3.7.

subprocess.DETACHED_PROCESS

A *Popen* `creationflags` parameter to specify that a new process will not inherit its parent’s console. This value cannot be used with `CREATE_NEW_CONSOLE`.

New in version 3.7.

subprocess.CREATE_DEFAULT_ERROR_MODE

A *Popen* `creationflags` parameter to specify that a new process does not inherit the error mode of the calling process. Instead, the new process gets the default error mode. This feature is particularly useful for multithreaded shell applications that run with hard errors disabled.

New in version 3.7.

subprocess.CREATE_BREAKAWAY_FROM_JOB

A *Popen* `creationflags` parameter to specify that a new process is not associated with the job.

New in version 3.7.

17.5.5 Older high-level API

Prior to Python 3.5, these three functions comprised the high level API to `subprocess`. You can now use *run()* in many cases, but lots of existing code calls these functions.

**subprocess.call(*args*, *, *stdin*=None, *stdout*=None, *stderr*=None, *shell*=False, *cwd*=None, *time-*
out=None)**

Run the command described by *args*. Wait for command to complete, then return the *returncode* attribute.

This is equivalent to:

```
run(...).returncode
```

(except that the *input* and *check* parameters are not supported)

The arguments shown above are merely the most common ones. The full function signature is largely the same as that of the *Popen* constructor - this function passes all supplied arguments other than *timeout* directly through to that interface.

Note: Do not use `stdout=PIPE` or `stderr=PIPE` with this function. The child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from.

Changed in version 3.3: *timeout* was added.

```
subprocess.check_call(args, *, stdin=None, stdout=None, stderr=None, shell=False, cwd=None,
                     timeout=None)
```

Run command with arguments. Wait for command to complete. If the return code was zero then return, otherwise raise `CalledProcessError`. The `CalledProcessError` object will have the return code in the *returncode* attribute.

This is equivalent to:

```
run(..., check=True)
```

(except that the *input* parameter is not supported)

The arguments shown above are merely the most common ones. The full function signature is largely the same as that of the `Popen` constructor - this function passes all supplied arguments other than *timeout* directly through to that interface.

Note: Do not use `stdout=PIPE` or `stderr=PIPE` with this function. The child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from.

Changed in version 3.3: *timeout* was added.

```
subprocess.check_output(args, *, stdin=None, stderr=None, shell=False, cwd=None, encoding=None,
                       errors=None, universal_newlines=None, timeout=None,
                       text=None)
```

Run command with arguments and return its output.

If the return code was non-zero it raises a `CalledProcessError`. The `CalledProcessError` object will have the return code in the *returncode* attribute and any output in the *output* attribute.

This is equivalent to:

```
run(..., check=True, stdout=PIPE).stdout
```

The arguments shown above are merely the most common ones. The full function signature is largely the same as that of `run()` - most arguments are passed directly through to that interface. However, explicitly passing `input=None` to inherit the parent's standard input file handle is not supported.

By default, this function will return the data as encoded bytes. The actual encoding of the output data may depend on the command being invoked, so the decoding to text will often need to be handled at the application level.

This behaviour may be overridden by setting *universal_newlines* to `True` as described above in *Frequently Used Arguments*.

To also capture standard error in the result, use `stderr=subprocess.STDOUT`:

```
>>> subprocess.check_output(
...     "ls non_existent_file; exit 0",
...     stderr=subprocess.STDOUT,
...     shell=True)
'ls: non_existent_file: No such file or directory\n'
```

New in version 3.1.

Changed in version 3.3: *timeout* was added.

Changed in version 3.4: Support for the *input* keyword argument was added.

Changed in version 3.6: *encoding* and *errors* were added. See [run\(\)](#) for details.

New in version 3.7: *text* was added as a more readable alias for *universal_newlines*.

17.5.6 Replacing Older Functions with the subprocess Module

In this section, “a becomes b” means that b can be used as a replacement for a.

Note: All “a” functions in this section fail (more or less) silently if the executed program cannot be found; the “b” replacements raise *OSError* instead.

In addition, the replacements using *check_output()* will fail with a *CalledProcessError* if the requested operation produces a non-zero return code. The output is still available as the *output* attribute of the raised exception.

In the following examples, we assume that the relevant functions have already been imported from the *subprocess* module.

Replacing `/bin/sh` shell backquote

```
output=`mycmd myarg`
```

becomes:

```
output = check_output(["mycmd", "myarg"])
```

Replacing shell pipeline

```
output=`dmesg | grep hda`
```

becomes:

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close() # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
```

The `p1.stdout.close()` call after starting the `p2` is important in order for `p1` to receive a `SIGPIPE` if `p2` exits before `p1`.

Alternatively, for trusted input, the shell’s own pipeline support may still be used directly:

```
output=`dmesg | grep hda`
```

becomes:

```
output=check_output("dmesg | grep hda", shell=True)
```

Replacing `os.system()`

```
sts = os.system("mycmd" + " myarg")
# becomes
sts = call("mycmd" + " myarg", shell=True)
```

Notes:

- Calling the program through the shell is usually not required.

A more realistic example would look like this:

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print("Child was terminated by signal", -retcode, file=sys.stderr)
    else:
        print("Child returned", retcode, file=sys.stderr)
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```

Replacing the `os.spawn` family

`P_NOWAIT` example:

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

`P_WAIT` example:

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

Vector example:

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

Environment example:

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

Replacing `os.popen()`, `os.popen2()`, `os.popen3()`

```
(child_stdin, child_stdout) = os.popen2(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

```
(child_stdin,
 child_stdout,
 child_stderr) = os.popen3(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin,
 child_stdout,
 child_stderr) = (p.stdin, p.stdout, p.stderr)
```

```
(child_stdin, child_stdout_and_stderr) = os.popen4(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```

Return code handling translates as follows:

```
pipe = os.popen(cmd, 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print("There were some errors")
==>
process = Popen(cmd, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print("There were some errors")
```

Replacing functions from the popen2 module

Note: If the `cmd` argument to `popen2` functions is a string, the command is executed through `/bin/sh`. If it is a list, the command is directly executed.

```
(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize, mode)
==>
p = Popen("somestring", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

```
(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"], bufsize, mode)
==>
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

`popen2.Popen3` and `popen2.Popen4` basically work as [*subprocess.Popen*](#), except that:

- *Popen* raises an exception if the execution fails.
- the *capturestderr* argument is replaced with the *stderr* argument.

- `stdin=PIPE` and `stdout=PIPE` must be specified.
- `popen2` closes all file descriptors by default, but you have to specify `close_fds=True` with *Popen* to guarantee this behavior on all platforms or past Python versions.

17.5.7 Legacy Shell Invocation Functions

This module also provides the following legacy functions from the 2.x `commands` module. These operations implicitly invoke the system shell and none of the guarantees described above regarding security and exception handling consistency are valid for these functions.

`subprocess.getstatusoutput(cmd)`

Return `(exitcode, output)` of executing `cmd` in a shell.

Execute the string `cmd` in a shell with `Popen.check_output()` and return a 2-tuple `(exitcode, output)`. The locale encoding is used; see the notes on *Frequently Used Arguments* for more details.

A trailing newline is stripped from the output. The exit code for the command can be interpreted as the return code of `subprocess`. Example:

```
>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(1, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(127, 'sh: /bin/junk: not found')
>>> subprocess.getstatusoutput('/bin/kill $$')
(-15, '')
```

Availability: POSIX & Windows.

Changed in version 3.3.4: Windows support was added.

The function now returns `(exitcode, output)` instead of `(status, output)` as it did in Python 3.3.3 and earlier. `exitcode` has the same value as *returncode*.

`subprocess.getoutput(cmd)`

Return output (`stdout` and `stderr`) of executing `cmd` in a shell.

Like *getstatusoutput()*, except the exit code is ignored and the return value is a string containing the command's output. Example:

```
>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'
```

Availability: POSIX & Windows.

Changed in version 3.3.4: Windows support added

17.5.8 Notes

Converting an argument sequence to a string on Windows

On Windows, an *args* sequence is converted to a string that can be parsed using the following rules (which correspond to the rules used by the MS C runtime):

1. Arguments are delimited by white space, which is either a space or a tab.

2. A string surrounded by double quotation marks is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.
3. A double quotation mark preceded by a backslash is interpreted as a literal double quotation mark.
4. Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
5. If backslashes immediately precede a double quotation mark, every pair of backslashes is interpreted as a literal backslash. If the number of backslashes is odd, the last backslash escapes the next double quotation mark as described in rule 3.

See also:

`shlex` Module which provides function to parse and escape command lines.

17.6 sched — Event scheduler

Source code: [Lib/sched.py](#)

The `sched` module defines a class which implements a general purpose event scheduler:

class `sched.scheduler`(*timefunc*=`time.monotonic`, *delayfunc*=`time.sleep`)

The `scheduler` class defines a generic interface to scheduling events. It needs two functions to actually deal with the “outside world” — *timefunc* should be callable without arguments, and return a number (the “time”, in any units whatsoever). If `time.monotonic` is not available, the *timefunc* default is `time.time` instead. The *delayfunc* function should be callable with one argument, compatible with the output of *timefunc*, and should delay that many time units. *delayfunc* will also be called with the argument 0 after each event is run to allow other threads an opportunity to run in multi-threaded applications.

Changed in version 3.3: *timefunc* and *delayfunc* parameters are optional.

Changed in version 3.3: `scheduler` class can be safely used in multi-threaded environments.

Example:

```
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(a='default'):
...     print("From print_time", time.time(), a)
...
>>> def print_some_times():
...     print(time.time())
...     s.enter(10, 1, print_time)
...     s.enter(5, 2, print_time, argument=('positional',))
...     s.enter(5, 1, print_time, kwargs={'a': 'keyword'})
...     s.run()
...     print(time.time())
...
>>> print_some_times()
930343690.257
From print_time 930343695.274 positional
From print_time 930343695.275 keyword
From print_time 930343700.273 default
930343700.276
```

17.6.1 Scheduler Objects

`scheduler` instances have the following methods and attributes:

`scheduler.enterabs(time, priority, action, argument=(), kwargs={})`

Schedule a new event. The *time* argument should be a numeric type compatible with the return value of the *timefunc* function passed to the constructor. Events scheduled for the same *time* will be executed in the order of their *priority*. A lower number represents a higher priority.

Executing the event means executing `action(*argument, **kwargs)`. *argument* is a sequence holding the positional arguments for *action*. *kwargs* is a dictionary holding the keyword arguments for *action*.

Return value is an event which may be used for later cancellation of the event (see `cancel()`).

Changed in version 3.3: *argument* parameter is optional.

New in version 3.3: *kwargs* parameter was added.

`scheduler.enter(delay, priority, action, argument=(), kwargs={})`

Schedule an event for *delay* more time units. Other than the relative time, the other arguments, the effect and the return value are the same as those for `enterabs()`.

Changed in version 3.3: *argument* parameter is optional.

New in version 3.3: *kwargs* parameter was added.

`scheduler.cancel(event)`

Remove the event from the queue. If *event* is not an event currently in the queue, this method will raise a `ValueError`.

`scheduler.empty()`

Return true if the event queue is empty.

`scheduler.run(blocking=True)`

Run all scheduled events. This method will wait (using the `delayfunc()` function passed to the constructor) for the next event, then execute it and so on until there are no more scheduled events.

If *blocking* is false executes the scheduled events due to expire soonest (if any) and then return the deadline of the next scheduled call in the scheduler (if any).

Either *action* or *delayfunc* can raise an exception. In either case, the scheduler will maintain a consistent state and propagate the exception. If an exception is raised by *action*, the event will not be attempted in future calls to `run()`.

If a sequence of events takes longer to run than the time available before the next event, the scheduler will simply fall behind. No events will be dropped; the calling code is responsible for canceling events which are no longer pertinent.

New in version 3.3: *blocking* parameter was added.

`scheduler.queue`

Read-only attribute returning a list of upcoming events in the order they will be run. Each event is shown as a *named tuple* with the following fields: time, priority, action, argument, kwargs.

17.7 queue — A synchronized queue class

Source code: [Lib/queue.py](#)

The `queue` module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. The `Queue` class in this

module implements all the required locking semantics. It depends on the availability of thread support in Python; see the *threading* module.

The module implements three types of queue, which differ only in the order in which the entries are retrieved. In a FIFO queue, the first tasks added are the first retrieved. In a LIFO queue, the most recently added entry is the first retrieved (operating like a stack). With a priority queue, the entries are kept sorted (using the *heapq* module) and the lowest valued entry is retrieved first.

Internally, those three types of queues use locks to temporarily block competing threads; however, they are not designed to handle reentrancy within a thread.

In addition, the module implements a “simple” FIFO queue type, *SimpleQueue*, whose specific implementation provides additional guarantees in exchange for the smaller functionality.

The *queue* module defines the following classes and exceptions:

class *queue.Queue*(*maxsize=0*)

Constructor for a FIFO queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

class *queue.LifoQueue*(*maxsize=0*)

Constructor for a LIFO queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

class *queue.PriorityQueue*(*maxsize=0*)

Constructor for a priority queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

The lowest valued entries are retrieved first (the lowest valued entry is the one returned by `sorted(list(entries))[0]`). A typical pattern for entries is a tuple in the form: (*priority_number*, *data*).

If the *data* elements are not comparable, the data can be wrapped in a class that ignores the data item and only compares the priority number:

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

class *queue.SimpleQueue*

Constructor for an unbounded FIFO queue. Simple queues lack advanced functionality such as task tracking.

New in version 3.7.

exception *queue.Empty*

Exception raised when non-blocking *get()* (or *get_nowait()*) is called on a *Queue* object which is empty.

exception *queue.Full*

Exception raised when non-blocking *put()* (or *put_nowait()*) is called on a *Queue* object which is full.

17.7.1 Queue Objects

Queue objects (*Queue*, *LifoQueue*, or *PriorityQueue*) provide the public methods described below.

Queue.qsize()

Return the approximate size of the queue. Note, `qsize() > 0` doesn't guarantee that a subsequent `get()` will not block, nor will `qsize() < maxsize` guarantee that `put()` will not block.

Queue.empty()

Return **True** if the queue is empty, **False** otherwise. If `empty()` returns **True** it doesn't guarantee that a subsequent call to `put()` will not block. Similarly, if `empty()` returns **False** it doesn't guarantee that a subsequent call to `get()` will not block.

Queue.full()

Return **True** if the queue is full, **False** otherwise. If `full()` returns **True** it doesn't guarantee that a subsequent call to `get()` will not block. Similarly, if `full()` returns **False** it doesn't guarantee that a subsequent call to `put()` will not block.

Queue.put(item, block=True, timeout=None)

Put *item* into the queue. If optional args *block* is true and *timeout* is **None** (the default), block if necessary until a free slot is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *Full* exception if no free slot was available within that time. Otherwise (*block* is false), put an item on the queue if a free slot is immediately available, else raise the *Full* exception (*timeout* is ignored in that case).

Queue.put_nowait(item)

Equivalent to `put(item, False)`.

Queue.get(block=True, timeout=None)

Remove and return an item from the queue. If optional args *block* is true and *timeout* is **None** (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *Empty* exception if no item was available within that time. Otherwise (*block* is false), return an item if one is immediately available, else raise the *Empty* exception (*timeout* is ignored in that case).

Queue.get_nowait()

Equivalent to `get(False)`.

Two methods are offered to support tracking whether enqueued tasks have been fully processed by daemon consumer threads.

Queue.task_done()

Indicate that a formerly enqueued task is complete. Used by queue consumer threads. For each *get()* used to fetch a task, a subsequent call to *task_done()* tells the queue that the processing on the task is complete.

If a *join()* is currently blocking, it will resume when all items have been processed (meaning that a *task_done()* call was received for every item that had been *put()* into the queue).

Raises a *ValueError* if called more times than there were items placed in the queue.

Queue.join()

Blocks until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls *task_done()* to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, *join()* unblocks.

Example of how to wait for enqueued tasks to be completed:

```

def worker():
    while True:
        item = q.get()
        if item is None:
            break
        do_work(item)
        q.task_done()

q = queue.Queue()
threads = []
for i in range(num_worker_threads):
    t = threading.Thread(target=worker)
    t.start()
    threads.append(t)

for item in source():
    q.put(item)

# block until all tasks are done
q.join()

# stop workers
for i in range(num_worker_threads):
    q.put(None)
for t in threads:
    t.join()

```

17.7.2 SimpleQueue Objects

SimpleQueue objects provide the public methods described below.

SimpleQueue.qsize()

Return the approximate size of the queue. Note, `qsize() > 0` doesn't guarantee that a subsequent `get()` will not block.

SimpleQueue.empty()

Return `True` if the queue is empty, `False` otherwise. If `empty()` returns `False` it doesn't guarantee that a subsequent call to `get()` will not block.

SimpleQueue.put(item, block=True, timeout=None)

Put *item* into the queue. The method never blocks and always succeeds (except for potential low-level errors such as failure to allocate memory). The optional args *block* and *timeout* are ignored and only provided for compatibility with *Queue.put()*.

CPython implementation detail: This method has a C implementation which is reentrant. That is, a `put()` or `get()` call can be interrupted by another `put()` call in the same thread without deadlocking or corrupting internal state inside the queue. This makes it appropriate for use in destructors such as `__del__` methods or *weakref* callbacks.

SimpleQueue.put_nowait(item)

Equivalent to `put(item)`, provided for compatibility with *Queue.put_nowait()*.

SimpleQueue.get(block=True, timeout=None)

Remove and return an item from the queue. If optional args *block* is true and *timeout* is `None` (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the *Empty* exception if no item was available within that time. Otherwise

(*block* is false), return an item if one is immediately available, else raise the *Empty* exception (*timeout* is ignored in that case).

`SimpleQueue.get_nowait()`
Equivalent to `get(False)`.

See also:

Class `multiprocessing.Queue` A queue class for use in a multi-processing (rather than multi-threading) context.

`collections.deque` is an alternative implementation of unbounded queues with fast atomic `append()` and `popleft()` operations that do not require locking.

The following are support modules for some of the above services:

17.8 _thread — Low-level threading API

This module provides low-level primitives for working with multiple threads (also called *light-weight processes* or *tasks*) — multiple threads of control sharing their global data space. For synchronization, simple locks (also called *mutexes* or *binary semaphores*) are provided. The `threading` module provides an easier to use and higher-level threading API built on top of this module.

Changed in version 3.7: This module used to be optional, it is now always available.

This module defines the following constants and functions:

exception `_thread.error`
Raised on thread-specific errors.

Changed in version 3.3: This is now a synonym of the built-in `RuntimeError`.

`_thread.LockType`
This is the type of lock objects.

`_thread.start_new_thread(function, args[, kwargs])`
Start a new thread and return its identifier. The thread executes the function *function* with the argument list *args* (which must be a tuple). The optional *kwargs* argument specifies a dictionary of keyword arguments. When the function returns, the thread silently exits. When the function terminates with an unhandled exception, a stack trace is printed and then the thread exits (but other threads continue to run).

`_thread.interrupt_main()`
Raise a `KeyboardInterrupt` exception in the main thread. A subthread can use this function to interrupt the main thread.

`_thread.exit()`
Raise the `SystemExit` exception. When not caught, this will cause the thread to exit silently.

`_thread.allocate_lock()`
Return a new lock object. Methods of locks are described below. The lock is initially unlocked.

`_thread.get_ident()`
Return the ‘thread identifier’ of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

`_thread.stack_size([size])`
Return the thread stack size used when creating new threads. The optional *size* argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured

default) or a positive integer value of at least 32,768 (32 KiB). If *size* is not specified, 0 is used. If changing the thread stack size is unsupported, a `RuntimeError` is raised. If the specified stack size is invalid, a `ValueError` is raised and the stack size is unmodified. 32 KiB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32 KiB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4 KiB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information).

Availability: Windows, systems with POSIX threads.

`_thread.TIMEOUT_MAX`

The maximum value allowed for the *timeout* parameter of `Lock.acquire()`. Specifying a timeout greater than this value will raise an `OverflowError`.

New in version 3.2.

Lock objects have the following methods:

`lock.acquire(waitflag=1, timeout=-1)`

Without any optional argument, this method acquires the lock unconditionally, if necessary waiting until it is released by another thread (only one thread at a time can acquire a lock — that's their reason for existence).

If the integer *waitflag* argument is present, the action depends on its value: if it is zero, the lock is only acquired if it can be acquired immediately without waiting, while if it is nonzero, the lock is acquired unconditionally as above.

If the floating-point *timeout* argument is present and positive, it specifies the maximum wait time in seconds before returning. A negative *timeout* argument specifies an unbounded wait. You cannot specify a *timeout* if *waitflag* is zero.

The return value is `True` if the lock is acquired successfully, `False` if not.

Changed in version 3.2: The *timeout* parameter is new.

Changed in version 3.2: Lock acquires can now be interrupted by signals on POSIX.

`lock.release()`

Releases the lock. The lock must have been acquired earlier, but not necessarily by the same thread.

`lock.locked()`

Return the status of the lock: `True` if it has been acquired by some thread, `False` if not.

In addition to these methods, lock objects can also be used via the `with` statement, e.g.:

```
import _thread

a_lock = _thread.allocate_lock()

with a_lock:
    print("a_lock is locked while this executes")
```

Caveats:

- Threads interact strangely with interrupts: the `KeyboardInterrupt` exception will be received by an arbitrary thread. (When the `signal` module is available, interrupts always go to the main thread.)
- Calling `sys.exit()` or raising the `SystemExit` exception is equivalent to calling `_thread.exit()`.
- It is not possible to interrupt the `acquire()` method on a lock — the `KeyboardInterrupt` exception will happen after the lock has been acquired.

- When the main thread exits, it is system defined whether the other threads survive. On most systems, they are killed without executing `try ... finally` clauses or executing object destructors.
- When the main thread exits, it does not do any of its usual cleanup (except that `try ... finally` clauses are honored), and the standard I/O files are not flushed.

17.9 `_dummy_thread` — Drop-in replacement for the `_thread` module

Source code: `Lib/_dummy_thread.py`

Deprecated since version 3.7: Python now always has threading enabled. Please use `_thread` (or, better, `threading`) instead.

This module provides a duplicate interface to the `_thread` module. It was meant to be imported when the `_thread` module was not provided on a platform.

Be careful to not use this module where deadlock might occur from a thread being created that blocks waiting for another thread to be created. This often occurs with blocking I/O.

17.10 `dummy_threading` — Drop-in replacement for the `threading` module

Source code: `Lib/dummy_threading.py`

Deprecated since version 3.7: Python now always has threading enabled. Please use `threading` instead.

This module provides a duplicate interface to the `threading` module. It was meant to be imported when the `_thread` module was not provided on a platform.

Be careful to not use this module where deadlock might occur from a thread being created that blocks waiting for another thread to be created. This often occurs with blocking I/O.