17.2. `threading` — Higher-level threading interface¶

This module constructs higher-level threading interfaces on top of the lower level `thread` module. See also the `mutex` and `Queue` modules.

The `dummy_threading` module is provided for situations where `threading` cannot be used because `thread` is missing.

Note

Starting with Python 2.6, this module provides PEP 8 compliant aliases and properties to replace the `camelCase` names that were inspired by Java's threading API. This updated API is compatible with that of the `multiprocessing` module. However, no schedule has been set for the deprecation of the `camelCase` names and they remain fully supported in both Python 2.x and 3.x.

Note

Starting with Python 2.5, several Thread methods raise `RuntimeError` instead of `AssertionError` if called erroneously.

This module defines the following functions and objects:

`threading.active_count`()¶

`threading.activeCount`()¶

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

`threading.Condition`()

A factory function that returns a new condition variable object. A condition variable allows one or more threads to wait until they are notified by another thread.

`threading.current_thread`()¶

`threading.currentThread`()¶

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.enumerate`()¶

Return a list of all `Thread` objects currently alive. The list includes daemonic 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.Event`()

A factory function that returns a new event object. 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.

class `threading.local`¶

A class that represents thread-local data. Thread-local data are 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.

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

New in version 2.4.

`threading.Lock`()¶

A factory function that returns a new primitive lock object. Once a thread has acquired it, subsequent attempts to acquire it block, until it is released; any thread may release it.

`threading.RLock`()¶

A factory function that returns a new reentrant lock object. 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.

`threading.Semaphore`([*value*])

A factory function that returns a new semaphore object. A semaphore manages a 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.

`threading.BoundedSemaphore`([*value*])¶
A factory function that returns a new bounded semaphore object. 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.

class `threading.Thread`¶
A class that represents a thread of control. This class can be safely subclassed in a limited fashion.

class `threading.Timer`¶
A thread that executes a function after a specified interval has passed.

`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.

New in version 2.3.

`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.

New in version 2.3.

`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 (32kB). If changing the thread stack size is unsupported, a `ThreadError` is raised. If the specified stack size is invalid, a `ValueError` is raised and the stack size is unmodified. 32kB 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 > 32kB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4kB 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.

New in version 2.5.

Detailed interfaces for the objects are documented 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.

17.2.1. Thread Objects¶

This 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.

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={}*)

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

`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 `RuntimeException` 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*])¶

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 `isAlive()` 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 raises the same exception.

`getName`()¶

`setName`()¶

Old API for `name`.

`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.

`ident`¶

The ‘thread identifier’ of this thread or `None` if the thread has not been started. This is a nonzero integer. See the `thread.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.

New in version 2.6.

`is_alive`()¶

`isAlive`()¶

Return whether the thread is alive.

Roughly, a thread is alive from the moment the `start()` method returns until its `run()` method terminates. The module function `enumerate()` returns a list of all alive threads.

`isDaemon`()¶

`setDaemon`()¶

Old API for `daemon`.

`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.

17.2.2. 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.

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.

`Lock.acquire`([*blocking=1*])¶

Acquire a lock, blocking or non-blocking.

When invoked without arguments, block until the lock is unlocked, then set it to locked, and return true.

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.

`Lock.release`()¶

Release a 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.

Do not call this method when the lock is unlocked.

There is no return value.

17.2.3. 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.

`RLock.acquire`([*blocking=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.

`RLock.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.

17.2.4. 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.)

A condition variable has `acquire()` and `release()` methods that call the corresponding methods of the associated lock. It also has a `wait()` method, and `notify()` and `notifyAll()` methods. These three must only be called when the calling thread has acquired the lock, otherwise a `RuntimeError` is raised.

The `wait()` method releases the lock, and then blocks until it is awakened by a `notify()` or `notifyAll()` call for the same condition variable in another thread. Once awakened, it 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 `notifyAll()` method wakes up all threads waiting for the condition variable.

Note: the `notify()` and `notifyAll()` 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 `notifyAll()` finally relinquishes ownership of the lock.

Tip: 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 `notifyAll()` 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
cv.acquire()
while not an_item_is_available():
   cv.wait()
get_an_available_item()
cv.release()

# Produce one item
cv.acquire()
make_an_item_available()
cv.notify()
cv.release()
```

To choose between `notify()` and `notifyAll()`, 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*])¶

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.

`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*])¶

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 `notifyAll()` 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.

`notify`()¶

Wake up a 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 one of the threads waiting for the condition variable, if any are waiting; it is a no-op if no threads are waiting.

The current implementation wakes up exactly one thread, if any are waiting. However, it's not safe to rely on this behavior. A future, optimized implementation may occasionally wake up more than one thread.

Note: the 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()`¶
`notifyAll()`¶
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.

17.2.5. 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 `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()`.

class `threading.Semaphore`([*value*])¶

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.

`acquire`([*blocking*])¶

Acquire a semaphore.

When invoked without arguments: if the internal counter is larger than zero on entry, decrement it by one and return immediately. If it is zero on entry, block, waiting until some other thread has called `release()` to make it larger than zero. This is done with proper interlocking so that if multiple `acquire()` calls are blocked, `release()` will wake exactly one of them up. The implementation may pick one at random, so the order in which blocked threads are awakened should not be relied on. There is no return value in this case.

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

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.

`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.

17.2.5.1. `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 size 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:

```
pool_sema.acquire()
conn = connectdb()
... use connection ...
conn.close()
pool_sema.release()
```

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.

17.2.6. 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`¶

The internal flag is initially false.

`is_set`()¶

`isSet`()¶

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*])¶

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 the internal flag on exit, so it will always return `True` except if a timeout is given and the operation times out.

Changed in version 2.7: Previously, the method always returned `None`.

17.2.7. 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*=[], *kwargs*={})

Create a timer that will run *function* with arguments *args* and keyword arguments *kwargs*, after *interval* seconds have passed.

`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.

17.2.8. 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.

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

```
import threading

some_rlock = threading.RLock()

with some_rlock:
   print "some_rlock is locked while this executes"
```

17.2.9. Importing in threaded code¶

While the import machinery is thread safe, there are two key restrictions on threaded imports due to inherent limitations in the way that thread safety is provided:

- Firstly, other than in the main module, an import should not have the side effect of spawning a new thread and then waiting for that thread in any way. Failing to abide by this restriction can lead to a deadlock if the spawned thread directly or indirectly attempts to import a module.
- Secondly, all import attempts must be completed before the interpreter starts shutting itself down. This can be most easily achieved by only performing imports from non-daemon threads created through the threading module. Daemon threads and threads created directly with the thread module will require some other form of synchronization to ensure they do not attempt imports after system shutdown has commenced. Failure to abide by this restriction will lead to intermittent exceptions and crashes during interpreter shutdown (as the late imports attempt to access machinery which is no longer in a valid state).

Table Of Contents

17.2. `threading` — Higher-level threading interface

Previous topic

17.1. `select` — Waiting for I/O completion

Next topic

17.3. `thread` — Multiple threads of control

This Page

- Show Source

Navigation

- index
- modules |
- next |
- previous |
- Python v2.6.4 documentation »
- The Python Standard Library »
- 17. Optional Operating System Services »