python学习笔记之多线程中的daemon与join特性

**none daemon与daemon**

　　在python 3中，threading属于标准库中的模块，在version 3.3中增加了daemon参数，值是一个bool值，默认为False。

　　如果一个线程的daemon=False，即我们说这个线程是一个none daemon的线程，那主线程永远会等待子线程全部退出后自己才退出。

看下边的例子，下边的例子全部都在pytharm的IDE中进行：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import threading  import logging  import time    logging.basicConfig(level=logging.DEBUG, format='%(asctime)s %(levelname)s [%(threadName)s] %(message)s')      def worker(message):  time.sleep(5)  logging.debug("worker is started, {0}".format(message))    if \_\_name\_\_ == '\_\_main\_\_':  t = threading.Thread(target=worker, name='worker', kwargs={'message': 'ha,ha'})  t.daemon = False  t.start()  logging.debug("main thread exiting") |

当运行这段代码时，先打印出了主线程的日志信息，如下：

|  |  |
| --- | --- |
| 1 | 2016-08-14 14:06:24,358 DEBUG [MainThread] main thread exiting |

主线程被阻塞，然后等待5秒后再打印出了子线程的日志信息，如下：

|  |  |
| --- | --- |
| 1  2 | 2016-08-14 14:06:24,358 DEBUG [MainThread] main thread exiting # 被阻塞  2016-08-14 14:06:29,359 DEBUG [worker] worker is started, ha,ha # 5秒后才打印出来 |

所以当线程是一个none daemon线程时，主线程会被阻塞，直到全部子线程退出。

当线程是一个daemon线程时呢？把上边的t.daemon = False修改成t.daemon = True后再次运行代码，结果如下：

|  |  |
| --- | --- |
| 1  2  3 | 2016-08-14 14:12:39,644 DEBUG [MainThread] main thread exiting    Process finished with exit code 0 |

从上边输出的结果可知，主线程没有等待子线程运行结束，而主线程自己就退出了。而当主线程退出后子线程也会跟着退出。

小结：

　　在none daemon的线程中主线程永远会等待子线程退出后才退出，在daemon的线程中主线程不会等待子线程退出后自己才退出，在none daemon线程中子线程因程序bug或其他原因而不能正常的退出时，那主线程就一直被阻塞，一直等待着子线程退出，这样线程所占用的资源就一直不能被释放。join函数能解决这样的问题。

**join( )**

　　 线程调用join函数后，主线程就会等待子线程退出后才退出，而子线程执行的代码如果不能正常退出时那主线程也一直会等待，如果在join时加上一个超时时间，那主线程只等待设置的超时时间后主线程就退出。

* none daemon中的join()

看下边的例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | import threading  import logging  import time    logging.basicConfig(level=logging.DEBUG, format='%(asctime)s %(levelname)s [%(threadName)s] %(message)s')      def worker(message):  print('exec worker')  time.sleep(5)  logging.debug("worker is started, {0}".format(message))    if \_\_name\_\_ == '\_\_main\_\_':  t = threading.Thread(target=worker, name='worker', kwargs={'message': 'ha,ha'})  t.daemon = False  t.start()  t.join()  logging.debug("main thread exiting") |

当运行上边代码时，先输出了：

|  |  |
| --- | --- |
| 1 | exec worker |

再等待5秒后输出了下边的日志信息：

|  |  |
| --- | --- |
| 1  2 | 2016-08-14 15:41:16,925 DEBUG [worker] worker is started, ha,ha  2016-08-14 15:41:16,925 DEBUG [MainThread] main thread exiting |

这说明当一个线程是none daemon时，并调用了join()函数，那主线程会等待子线程执行完成。这和不调用join()函数的none daemon线程是有区别的，调用join()函数时是主线程等待子线程完成，不调用join()函数时是主线程被阻塞后，一直等待子线程执行完成，从打印的日志可看出两者的区别。

join()函数可以给定一个超时时间，如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | import threading  import logging  import time    logging.basicConfig(level=logging.DEBUG, format='%(asctime)s %(levelname)s [%(threadName)s] %(message)s')      def worker(message):  print('exec worker')  time.sleep(5)  logging.debug("worker is started, {0}".format(message))    if \_\_name\_\_ == '\_\_main\_\_':  t = threading.Thread(target=worker, name='worker', kwargs={'message': 'ha,ha'})  t.daemon = False  t.start()  t.join(timeout=2)  logging.debug("main thread exiting") |

运行上边的代码后，首先输出exec worker，等待2秒，再输出2016-08-14 16:10:59,695 DEBUG [MainThread] main thread exiting,再等待3秒，最后输出2016-08-14 16:11:02,692 DEBUG [worker] worker is started, ha,ha，如下：

|  |  |
| --- | --- |
| 1  2  3 | exec worker # 2秒后输出下一行  2016-08-14 16:10:59,695 DEBUG [MainThread] main thread exiting # 3秒后输出下一行  2016-08-14 16:11:02,692 DEBUG [worker] worker is started, ha,ha |

在none daemon的线程中，如果给join()函数一个超时时间，当超过这个时间后，即使这个线程没有执行完成，程序就直接输出了2016-08-14 16:10:59,695 DEBUG [MainThread] main thread exiting，并且被阻塞在这里等待子线程的完成，因为线程是none daemon的。这种场景在现实的编程中应该不多。

* daemon中的join()

　　在实际的编程中，一般会把线程设置成daemon，并启用join()函数，并适当给一个超时时间，这样主线程即能等待子线程，又能兼顾子线程因一些原因被卡住后无法退出时导致主线程也无法退出的情况。如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | import threading  import logging  import time    logging.basicConfig(level=logging.DEBUG, format='%(asctime)s %(levelname)s [%(threadName)s] %(message)s')      def worker(message):  print('exec worker')  time.sleep(5)  logging.debug("worker is started, {0}".format(message))    if \_\_name\_\_ == '\_\_main\_\_':  t = threading.Thread(target=worker, name='worker', kwargs={'message': 'ha,ha'})  t.daemon = True  t.start()  t.join(timeout=6)  logging.debug("main thread exiting") |

运行上边的代码，先是输出exec worker，再等待5秒，依次输出2016-08-14 16:31:59,287 DEBUG [worker] worker is started, ha,ha和2016-08-14 16:31:59,287 DEBUG [MainThread] main thread exiting。这样，采用join的方式让主线程等待子线程正常退出，如果在调用worker函数时有bug，执行时一直退不出来，那到join的超时时间后，主线程同样能退出，把上边的t.join(timeout=6)修改成t.join(timeout=3)来模拟这样一个场景，如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | import threading  import logging  import time    logging.basicConfig(level=logging.DEBUG, format='%(asctime)s %(levelname)s [%(threadName)s] %(message)s')      def worker(message):  print('exec worker')  time.sleep(5)  logging.debug("worker is started, {0}".format(message))    if \_\_name\_\_ == '\_\_main\_\_':  t = threading.Thread(target=worker, name='worker', kwargs={'message': 'ha,ha'})  t.daemon = True  t.start()  t.join(timeout=3)  logging.debug("main thread exiting") |

执行上边的代码后，先是输出exec worker,再等待3秒后直接输出2016-08-14 16:36:23,608 DEBUG [MainThread] main thread exiting,worker函数里的内容没有被执行，如下：

|  |  |
| --- | --- |
| 1  2  3  4 | exec worker  2016-08-14 16:36:23,608 DEBUG [MainThread] main thread exiting    Process finished with exit code 0 |

# threading — Manage Concurrent Operations Within a Process

|  |  |
| --- | --- |
| **Purpose:** | Manage several threads of execution. |

Using threads allows a program to run multiple operations concurrently in the same process space.

## Thread Objects

The simplest way to use a Thread is to instantiate it with a target function and call start() to let it begin working.

*threading\_simple.py*

**import** **threading**

**def** worker():

*"""thread worker function"""*

print('Worker')

threads = []

**for** i **in** range(5):

t = threading.Thread(target=worker)

threads.append(t)

t.start()

The output is five lines with "Worker" on each.

$ python3 threading\_simple.py

Worker

Worker

Worker

Worker

Worker

It is useful to be able to spawn a thread and pass it arguments to tell it what work to do. Any type of object can be passed as argument to the thread. This example passes a number, which the thread then prints.

*threading\_simpleargs.py*

**import** **threading**

**def** worker(num):

*"""thread worker function"""*

print('Worker: *%s*' % num)

threads = []

**for** i **in** range(5):

t = threading.Thread(target=worker, args=(i,))

threads.append(t)

t.start()

The integer argument is now included in the message printed by each thread.

$ python3 threading\_simpleargs.py

Worker: 0

Worker: 1

Worker: 2

Worker: 3

Worker: 4

## Determining the Current Thread

Using arguments to identify or name the thread is cumbersome and unnecessary. Each Thread instance has a name with a default value that can be changed as the thread is created. Naming threads is useful in server processes with multiple service threads handling different operations.

*threading\_names.py*

**import** **threading**

**import** **time**

**def** worker():

print(threading.current\_thread().getName(), 'Starting')

time.sleep(0.2)

print(threading.current\_thread().getName(), 'Exiting')

**def** my\_service():

print(threading.current\_thread().getName(), 'Starting')

time.sleep(0.3)

print(threading.current\_thread().getName(), 'Exiting')

t = threading.Thread(name='my\_service', target=my\_service)

w = threading.Thread(name='worker', target=worker)

w2 = threading.Thread(target=worker) *# use default name*

w.start()

w2.start()

t.start()

The debug output includes the name of the current thread on each line. The lines with "Thread-1" in the thread name column correspond to the unnamed thread w2.

$ python3 threading\_names.py

worker Starting

Thread-1 Starting

my\_service Starting

worker Exiting

Thread-1 Exiting

my\_service Exiting

Most programs do not use print to debug. The [logging](https://pymotw.com/3/logging/index.html#module-logging) module supports embedding the thread name in every log message using the formatter code %(threadName)s. Including thread names in log messages makes it possible to trace those messages back to their source.

*threading\_names\_log.py*

**import** **logging**

**import** **threading**

**import** **time**

**def** worker():

logging.debug('Starting')

time.sleep(0.2)

logging.debug('Exiting')

**def** my\_service():

logging.debug('Starting')

time.sleep(0.3)

logging.debug('Exiting')

logging.basicConfig(

level=logging.DEBUG,

format='[*%(levelname)s*] (*%(threadName)-10s*) *%(message)s*',

)

t = threading.Thread(name='my\_service', target=my\_service)

w = threading.Thread(name='worker', target=worker)

w2 = threading.Thread(target=worker) *# use default name*

w.start()

w2.start()

t.start()

[logging](https://pymotw.com/3/logging/index.html#module-logging) is also thread-safe, so messages from different threads are kept distinct in the output.

$ python3 threading\_names\_log.py

[DEBUG] (worker ) Starting

[DEBUG] (Thread-1 ) Starting

[DEBUG] (my\_service) Starting

[DEBUG] (worker ) Exiting

[DEBUG] (Thread-1 ) Exiting

[DEBUG] (my\_service) Exiting

## Daemon vs. Non-Daemon Threads

Up to this point, the example programs have implicitly waited to exit until all threads have completed their work. Sometimes programs spawn a thread as a daemon that runs without blocking the main program from exiting. Using daemon threads is useful for services where there may not be an easy way to interrupt the thread, or where letting the thread die in the middle of its work does not lose or corrupt data (for example, a thread that generates “heart beats” for a service monitoring tool). To mark a thread as a daemon, pass daemon=True when constructing it or call its set\_daemon() method with True. The default is for threads to not be daemons.

*threading\_daemon.py*

**import** **threading**

**import** **time**

**import** **logging**

**def** daemon():

logging.debug('Starting')

time.sleep(0.2)

logging.debug('Exiting')

**def** non\_daemon():

logging.debug('Starting')

logging.debug('Exiting')

logging.basicConfig(

level=logging.DEBUG,

format='(*%(threadName)-10s*) *%(message)s*',

)

d = threading.Thread(name='daemon', target=daemon, daemon=**True**)

t = threading.Thread(name='non-daemon', target=non\_daemon)

d.start()

t.start()

The output does not include the "Exiting" message from the daemon thread, since all of the non-daemon threads (including the main thread) exit before the daemon thread wakes up from the sleep() call.

$ python3 threading\_daemon.py

(daemon ) Starting

(non-daemon) Starting

(non-daemon) Exiting

To wait until a daemon thread has completed its work, use the join() method.

*threading\_daemon\_join.py*

**import** **threading**

**import** **time**

**import** **logging**

**def** daemon():

logging.debug('Starting')

time.sleep(0.2)

logging.debug('Exiting')

**def** non\_daemon():

logging.debug('Starting')

logging.debug('Exiting')

logging.basicConfig(

level=logging.DEBUG,

format='(*%(threadName)-10s*) *%(message)s*',

)

d = threading.Thread(name='daemon', target=daemon, daemon=**True**)

t = threading.Thread(name='non-daemon', target=non\_daemon)

d.start()

t.start()

d.join()

t.join()

Waiting for the daemon thread to exit using join() means it has a chance to produce its "Exiting" message.

$ python3 threading\_daemon\_join.py

(daemon ) Starting

(non-daemon) Starting

(non-daemon) Exiting

(daemon ) Exiting

By default, join() blocks indefinitely. It is also possible to pass a float value representing the number of seconds to wait for the thread to become inactive. If the thread does not complete within the timeout period, join() returns anyway.

*threading\_daemon\_join\_timeout.py*

**import** **threading**

**import** **time**

**import** **logging**

**def** daemon():

logging.debug('Starting')

time.sleep(0.2)

logging.debug('Exiting')

**def** non\_daemon():

logging.debug('Starting')

logging.debug('Exiting')

logging.basicConfig(

level=logging.DEBUG,

format='(*%(threadName)-10s*) *%(message)s*',

)

d = threading.Thread(name='daemon', target=daemon, daemon=**True**)

t = threading.Thread(name='non-daemon', target=non\_daemon)

d.start()

t.start()

d.join(0.1)

print('d.isAlive()', d.isAlive())

t.join()

Since the timeout passed is less than the amount of time the daemon thread sleeps, the thread is still “alive” after join() returns.

$ python3 threading\_daemon\_join\_timeout.py

(daemon ) Starting

(non-daemon) Starting

(non-daemon) Exiting

d.isAlive() True

## Enumerating All Threads

It is not necessary to retain an explicit handle to all of the daemon threads in order to ensure they have completed before exiting the main process. enumerate() returns a list of active Thread instances. The list includes the current thread, and since joining the current thread introduces a deadlock situation, it must be skipped.

*threading\_enumerate.py*

**import** **random**

**import** **threading**

**import** **time**

**import** **logging**

**def** worker():

*"""thread worker function"""*

pause = random.randint(1, 5) / 10

logging.debug('sleeping *%0.2f*', pause)

time.sleep(pause)

logging.debug('ending')

logging.basicConfig(

level=logging.DEBUG,

format='(*%(threadName)-10s*) *%(message)s*',

)

**for** i **in** range(3):

t = threading.Thread(target=worker, daemon=**True**)

t.start()

main\_thread = threading.main\_thread()

**for** t **in** threading.enumerate():

**if** t **is** main\_thread:

**continue**

logging.debug('joining *%s*', t.getName())

t.join()

Because the worker is sleeping for a random amount of time, the output from this program may vary.

$ python3 threading\_enumerate.py

(Thread-1 ) sleeping 0.20

(Thread-2 ) sleeping 0.30

(Thread-3 ) sleeping 0.40

(MainThread) joining Thread-1

(Thread-1 ) ending

(MainThread) joining Thread-3

(Thread-2 ) ending

(Thread-3 ) ending

(MainThread) joining Thread-2

## Subclassing Thread

At start-up, a Thread does some basic initialization and then calls its run() method, which calls the target function passed to the constructor. To create a subclass of Thread, override run() to do whatever is necessary.

*threading\_subclass.py*

**import** **threading**

**import** **logging**

**class** **MyThread**(threading.Thread):

**def** run(self):

logging.debug('running')

logging.basicConfig(

level=logging.DEBUG,

format='(*%(threadName)-10s*) *%(message)s*',

)

**for** i **in** range(5):

t = MyThread()

t.start()

The return value of run() is ignored.

$ python3 threading\_subclass.py

(Thread-1 ) running

(Thread-2 ) running

(Thread-3 ) running

(Thread-4 ) running

(Thread-5 ) running

Because the args and kwargs values passed to the Thread constructor are saved in private variables using names prefixed with '\_\_', they are not easily accessed from a subclass. To pass arguments to a custom thread type, redefine the constructor to save the values in an instance attribute that can be seen in the subclass.

*threading\_subclass\_args.py*

**import** **threading**

**import** **logging**

**class** **MyThreadWithArgs**(threading.Thread):

**def** \_\_init\_\_(self, group=**None**, target=**None**, name=**None**,

args=(), kwargs=**None**, \*, daemon=**None**):

super().\_\_init\_\_(group=group, target=target, name=name,

daemon=daemon)

self.args = args

self.kwargs = kwargs

**def** run(self):

logging.debug('running with *%s* and *%s*',

self.args, self.kwargs)

logging.basicConfig(

level=logging.DEBUG,

format='(*%(threadName)-10s*) *%(message)s*',

)

**for** i **in** range(5):

t = MyThreadWithArgs(args=(i,), kwargs={'a': 'A', 'b': 'B'})

t.start()

MyThreadWithArgs uses the same API as Thread, but another class could easily change the constructor method to take more or different arguments more directly related to the purpose of the thread, as with any other class.

$ python3 threading\_subclass\_args.py

(Thread-1 ) running with (0,) and {'b': 'B', 'a': 'A'}

(Thread-2 ) running with (1,) and {'b': 'B', 'a': 'A'}

(Thread-3 ) running with (2,) and {'b': 'B', 'a': 'A'}

(Thread-4 ) running with (3,) and {'b': 'B', 'a': 'A'}

(Thread-5 ) running with (4,) and {'b': 'B', 'a': 'A'}

## Timer Threads

One example of a reason to subclass Thread is provided by Timer, also included in threading. A Timer starts its work after a delay, and can be canceled at any point within that delay time period.

*threading\_timer.py*

**import** **threading**

**import** **time**

**import** **logging**

**def** delayed():

logging.debug('worker running')

logging.basicConfig(

level=logging.DEBUG,

format='(*%(threadName)-10s*) *%(message)s*',

)

t1 = threading.Timer(0.3, delayed)

t1.setName('t1')

t2 = threading.Timer(0.3, delayed)

t2.setName('t2')

logging.debug('starting timers')

t1.start()

t2.start()

logging.debug('waiting before canceling *%s*', t2.getName())

time.sleep(0.2)

logging.debug('canceling *%s*', t2.getName())

t2.cancel()

logging.debug('done')

The second timer in this example is never run, and the first timer appears to run after the rest of the main program is done. Since it is not a daemon thread, it is joined implicitly when the main thread is done.

$ python3 threading\_timer.py

(MainThread) starting timers

(MainThread) waiting before canceling t2

(MainThread) canceling t2

(MainThread) done

(t1 ) worker running

## Signaling Between Threads

Although the point of using multiple threads is to run separate operations concurrently, there are times when it is important to be able to synchronize the operations in two or more threads. Event objects are a simple way to communicate between threads safely. An Event manages an internal flag that callers can control with the set() and clear() methods. Other threads can use wait() to pause until the flag is set, effectively blocking progress until allowed to continue.

*threading\_event.py*

**import** **logging**

**import** **threading**

**import** **time**

**def** wait\_for\_event(e):

*"""Wait for the event to be set before doing anything"""*

logging.debug('wait\_for\_event starting')

event\_is\_set = e.wait()

logging.debug('event set: *%s*', event\_is\_set)

**def** wait\_for\_event\_timeout(e, t):

*"""Wait t seconds and then timeout"""*

**while** **not** e.is\_set():

logging.debug('wait\_for\_event\_timeout starting')

event\_is\_set = e.wait(t)

logging.debug('event set: *%s*', event\_is\_set)

**if** event\_is\_set:

logging.debug('processing event')

**else**:

logging.debug('doing other work')

logging.basicConfig(

level=logging.DEBUG,

format='(*%(threadName)-10s*) *%(message)s*',

)

e = threading.Event()

t1 = threading.Thread(

name='block',

target=wait\_for\_event,

args=(e,),

)

t1.start()

t2 = threading.Thread(

name='nonblock',

target=wait\_for\_event\_timeout,

args=(e, 2),

)

t2.start()

logging.debug('Waiting before calling Event.set()')

time.sleep(0.3)

e.set()

logging.debug('Event is set')

The wait() method takes an argument representing the number of seconds to wait for the event before timing out. It returns a Boolean indicating whether or not the event is set, so the caller knows why wait() returned. The is\_set() method can be used separately on the event without fear of blocking.

In this example, wait\_for\_event\_timeout() checks the event status without blocking indefinitely. The wait\_for\_event() blocks on the call to wait(), which does not return until the event status changes.

$ python3 threading\_event.py

(block ) wait\_for\_event starting

(nonblock ) wait\_for\_event\_timeout starting

(MainThread) Waiting before calling Event.set()

(MainThread) Event is set

(nonblock ) event set: True

(nonblock ) processing event

(block ) event set: True

## Controlling Access to Resources

In addition to synchronizing the operations of threads, it is also important to be able to control access to shared resources to prevent corruption or missed data. Python’s built-in data structures (lists, dictionaries, etc.) are thread-safe as a side-effect of having atomic byte-codes for manipulating them (the global interpreter lock used to protect Python’s internal data structures is not released in the middle of an update). Other data structures implemented in Python, or simpler types like integers and floats, do not have that protection. To guard against simultaneous access to an object, use a Lock object.

*threading\_lock.py*

**import** **logging**

**import** **random**

**import** **threading**

**import** **time**

**class** **Counter**:

**def** \_\_init\_\_(self, start=0):

self.lock = threading.Lock()

self.value = start

**def** increment(self):

logging.debug('Waiting for lock')

self.lock.acquire()

**try**:

logging.debug('Acquired lock')

self.value = self.value + 1

**finally**:

self.lock.release()

**def** worker(c):

**for** i **in** range(2):

pause = random.random()

logging.debug('Sleeping *%0.02f*', pause)

time.sleep(pause)

c.increment()

logging.debug('Done')

logging.basicConfig(

level=logging.DEBUG,

format='(*%(threadName)-10s*) *%(message)s*',

)

counter = Counter()

**for** i **in** range(2):

t = threading.Thread(target=worker, args=(counter,))

t.start()

logging.debug('Waiting for worker threads')

main\_thread = threading.main\_thread()

**for** t **in** threading.enumerate():

**if** t **is** **not** main\_thread:

t.join()

logging.debug('Counter: *%d*', counter.value)

In this example, the worker() function increments a Counter instance, which manages a Lock to prevent two threads from changing its internal state at the same time. If the Lock was not used, there is a possibility of missing a change to the value attribute.

$ python3 threading\_lock.py

(Thread-1 ) Sleeping 0.18

(Thread-2 ) Sleeping 0.93

(MainThread) Waiting for worker threads

(Thread-1 ) Waiting for lock

(Thread-1 ) Acquired lock

(Thread-1 ) Sleeping 0.11

(Thread-1 ) Waiting for lock

(Thread-1 ) Acquired lock

(Thread-1 ) Done

(Thread-2 ) Waiting for lock

(Thread-2 ) Acquired lock

(Thread-2 ) Sleeping 0.81

(Thread-2 ) Waiting for lock

(Thread-2 ) Acquired lock

(Thread-2 ) Done

(MainThread) Counter: 4

To find out whether another thread has acquired the lock without holding up the current thread, pass False for the blocking argument to acquire(). In the next example, worker() tries to acquire the lock three separate times and counts how many attempts it has to make to do so. In the mean time, lock\_holder() cycles between holding and releasing the lock, with short pauses in each state used to simulate load.

*threading\_lock\_noblock.py*

**import** **logging**

**import** **threading**

**import** **time**

**def** lock\_holder(lock):

logging.debug('Starting')

**while** **True**:

lock.acquire()

**try**:

logging.debug('Holding')

time.sleep(0.5)

**finally**:

logging.debug('Not holding')

lock.release()

time.sleep(0.5)

**def** worker(lock):

logging.debug('Starting')

num\_tries = 0

num\_acquires = 0

**while** num\_acquires < 3:

time.sleep(0.5)

logging.debug('Trying to acquire')

have\_it = lock.acquire(0)

**try**:

num\_tries += 1

**if** have\_it:

logging.debug('Iteration *%d*: Acquired',

num\_tries)

num\_acquires += 1

**else**:

logging.debug('Iteration *%d*: Not acquired',

num\_tries)

**finally**:

**if** have\_it:

lock.release()

logging.debug('Done after *%d* iterations', num\_tries)

logging.basicConfig(

level=logging.DEBUG,

format='(*%(threadName)-10s*) *%(message)s*',

)

lock = threading.Lock()

holder = threading.Thread(

target=lock\_holder,

args=(lock,),

name='LockHolder',

daemon=**True**,

)

holder.start()

worker = threading.Thread(

target=worker,

args=(lock,),

name='Worker',

)

worker.start()

It takes worker() more than three iterations to acquire the lock three separate times.

$ python3 threading\_lock\_noblock.py

(LockHolder) Starting

(LockHolder) Holding

(Worker ) Starting

(LockHolder) Not holding

(Worker ) Trying to acquire

(Worker ) Iteration 1: Acquired

(LockHolder) Holding

(Worker ) Trying to acquire

(Worker ) Iteration 2: Not acquired

(LockHolder) Not holding

(Worker ) Trying to acquire

(Worker ) Iteration 3: Acquired

(LockHolder) Holding

(Worker ) Trying to acquire

(Worker ) Iteration 4: Not acquired

(LockHolder) Not holding

(Worker ) Trying to acquire

(Worker ) Iteration 5: Acquired

(Worker ) Done after 5 iterations

### Re-entrant Locks

Normal Lock objects cannot be acquired more than once, even by the same thread. This can introduce undesirable side-effects if a lock is accessed by more than one function in the same call chain.

*threading\_lock\_reacquire.py*

**import** **threading**

lock = threading.Lock()

print('First try :', lock.acquire())

print('Second try:', lock.acquire(0))

In this case, the second call to acquire() is given a zero timeout to prevent it from blocking because the lock has been obtained by the first call.

$ python3 threading\_lock\_reacquire.py

First try : True

Second try: False

In a situation where separate code from the same thread needs to “re-acquire” the lock, use an RLock instead.

*threading\_rlock.py*

**import** **threading**

lock = threading.RLock()

print('First try :', lock.acquire())

print('Second try:', lock.acquire(0))

The only change to the code from the previous example was substituting RLock for Lock.

$ python3 threading\_rlock.py

First try : True

Second try: True

### Locks as Context Managers

Locks implement the context manager API and are compatible with the with statement. Using with removes the need to explicitly acquire and release the lock.

*threading\_lock\_with.py*

**import** **threading**

**import** **logging**

**def** worker\_with(lock):

**with** lock:

logging.debug('Lock acquired via with')

**def** worker\_no\_with(lock):

lock.acquire()

**try**:

logging.debug('Lock acquired directly')

**finally**:

lock.release()

logging.basicConfig(

level=logging.DEBUG,

format='(*%(threadName)-10s*) *%(message)s*',

)

lock = threading.Lock()

w = threading.Thread(target=worker\_with, args=(lock,))

nw = threading.Thread(target=worker\_no\_with, args=(lock,))

w.start()

nw.start()

The two functions worker\_with() and worker\_no\_with() manage the lock in equivalent ways.

$ python3 threading\_lock\_with.py

(Thread-1 ) Lock acquired via with

(Thread-2 ) Lock acquired directly

## Synchronizing Threads

In addition to using Events, another way of synchronizing threads is through using a Condition object. Because the Condition uses a Lock, it can be tied to a shared resource, allowing multiple threads to wait for the resource to be updated. In this example, the consumer() threads wait for the Condition to be set before continuing. The producer() thread is responsible for setting the condition and notifying the other threads that they can continue.

*threading\_condition.py*

**import** **logging**

**import** **threading**

**import** **time**

**def** consumer(cond):

*"""wait for the condition and use the resource"""*

logging.debug('Starting consumer thread')

**with** cond:

cond.wait()

logging.debug('Resource is available to consumer')

**def** producer(cond):

*"""set up the resource to be used by the consumer"""*

logging.debug('Starting producer thread')

**with** cond:

logging.debug('Making resource available')

cond.notifyAll()

logging.basicConfig(

level=logging.DEBUG,

format='*%(asctime)s* (*%(threadName)-2s*) *%(message)s*',

)

condition = threading.Condition()

c1 = threading.Thread(name='c1', target=consumer,

args=(condition,))

c2 = threading.Thread(name='c2', target=consumer,

args=(condition,))

p = threading.Thread(name='p', target=producer,

args=(condition,))

c1.start()

time.sleep(0.2)

c2.start()

time.sleep(0.2)

p.start()

The threads use with to acquire the lock associated with the Condition. Using the acquire() and release() methods explicitly also works.

$ python3 threading\_condition.py

2016-07-10 10:45:28,170 (c1) Starting consumer thread

2016-07-10 10:45:28,376 (c2) Starting consumer thread

2016-07-10 10:45:28,581 (p ) Starting producer thread

2016-07-10 10:45:28,581 (p ) Making resource available

2016-07-10 10:45:28,582 (c1) Resource is available to consumer

2016-07-10 10:45:28,582 (c2) Resource is available to consumer

Barriers are another thread synchronization mechanism. A Barrier establishes a control point and all participating threads block until all of the participating “parties” have reached that point. It lets threads start up separately and then pause until they are all ready to proceed.

*threading\_barrier.py*

**import** **threading**

**import** **time**

**def** worker(barrier):

print(threading.current\_thread().name,

'waiting for barrier with *{}* others'.format(

barrier.n\_waiting))

worker\_id = barrier.wait()

print(threading.current\_thread().name, 'after barrier',

worker\_id)

NUM\_THREADS = 3

barrier = threading.Barrier(NUM\_THREADS)

threads = [

threading.Thread(

name='worker-*%s*' % i,

target=worker,

args=(barrier,),

)

**for** i **in** range(NUM\_THREADS)

]

**for** t **in** threads:

print(t.name, 'starting')

t.start()

time.sleep(0.1)

**for** t **in** threads:

t.join()

In this example, the Barrier is configured to block until three threads are waiting. When the condition is met, all of the threads are released past the control point at the same time. The return value from wait() indicates the number of the party being released, and can be used to limit some threads from taking an action like cleaning up a shared resource.

$ python3 threading\_barrier.py

worker-0 starting

worker-0 waiting for barrier with 0 others

worker-1 starting

worker-1 waiting for barrier with 1 others

worker-2 starting

worker-2 waiting for barrier with 2 others

worker-2 after barrier 2

worker-0 after barrier 0

worker-1 after barrier 1

The abort() method of Barrier causes all of the waiting threads to receive a BrokenBarrierError. This allows threads to clean up if processing is stopped while they are blocked on wait().

*threading\_barrier\_abort.py*

**import** **threading**

**import** **time**

**def** worker(barrier):

print(threading.current\_thread().name,

'waiting for barrier with *{}* others'.format(

barrier.n\_waiting))

**try**:

worker\_id = barrier.wait()

**except** threading.BrokenBarrierError:

print(threading.current\_thread().name, 'aborting')

**else**:

print(threading.current\_thread().name, 'after barrier',

worker\_id)

NUM\_THREADS = 3

barrier = threading.Barrier(NUM\_THREADS + 1)

threads = [

threading.Thread(

name='worker-*%s*' % i,

target=worker,

args=(barrier,),

)

**for** i **in** range(NUM\_THREADS)

]

**for** t **in** threads:

print(t.name, 'starting')

t.start()

time.sleep(0.1)

barrier.abort()

**for** t **in** threads:

t.join()

This example configures the Barrier to expect one more participating thread than is actually started so that processing in all of the threads is blocked. The abort() call raises an exception in each blocked thread.

$ python3 threading\_barrier\_abort.py

worker-0 starting

worker-0 waiting for barrier with 0 others

worker-1 starting

worker-1 waiting for barrier with 1 others

worker-2 starting

worker-2 waiting for barrier with 2 others

worker-0 aborting

worker-2 aborting

worker-1 aborting

## Limiting Concurrent Access to Resources

Sometimes it is useful to allow more than one worker access to a resource at a time, while still limiting the overall number. For example, a connection pool might support a fixed number of simultaneous connections, or a network application might support a fixed number of concurrent downloads. A Semaphoreis one way to manage those connections.

*threading\_semaphore.py*

**import** **logging**

**import** **random**

**import** **threading**

**import** **time**

**class** **ActivePool**:

**def** \_\_init\_\_(self):

super(ActivePool, self).\_\_init\_\_()

self.active = []

self.lock = threading.Lock()

**def** makeActive(self, name):

**with** self.lock:

self.active.append(name)

logging.debug('Running: *%s*', self.active)

**def** makeInactive(self, name):

**with** self.lock:

self.active.remove(name)

logging.debug('Running: *%s*', self.active)

**def** worker(s, pool):

logging.debug('Waiting to join the pool')

**with** s:

name = threading.current\_thread().getName()

pool.makeActive(name)

time.sleep(0.1)

pool.makeInactive(name)

logging.basicConfig(

level=logging.DEBUG,

format='*%(asctime)s* (*%(threadName)-2s*) *%(message)s*',

)

pool = ActivePool()

s = threading.Semaphore(2)

**for** i **in** range(4):

t = threading.Thread(

target=worker,

name=str(i),

args=(s, pool),

)

t.start()

In this example, the ActivePool class simply serves as a convenient way to track which threads are able to run at a given moment. A real resource pool would allocate a connection or some other value to the newly active thread, and reclaim the value when the thread is done. Here, it is just used to hold the names of the active threads to show that at most two are running concurrently.

$ python3 threading\_semaphore.py

2016-07-10 10:45:29,398 (0 ) Waiting to join the pool

2016-07-10 10:45:29,398 (0 ) Running: ['0']

2016-07-10 10:45:29,399 (1 ) Waiting to join the pool

2016-07-10 10:45:29,399 (1 ) Running: ['0', '1']

2016-07-10 10:45:29,399 (2 ) Waiting to join the pool

2016-07-10 10:45:29,399 (3 ) Waiting to join the pool

2016-07-10 10:45:29,501 (1 ) Running: ['0']

2016-07-10 10:45:29,501 (0 ) Running: []

2016-07-10 10:45:29,502 (3 ) Running: ['3']

2016-07-10 10:45:29,502 (2 ) Running: ['3', '2']

2016-07-10 10:45:29,607 (3 ) Running: ['2']

2016-07-10 10:45:29,608 (2 ) Running: []

## Thread-specific Data

While some resources need to be locked so multiple threads can use them, others need to be protected so that they are hidden from threads that do not own them. The local() class creates an object capable of hiding values from view in separate threads.

*threading\_local.py*

**import** **random**

**import** **threading**

**import** **logging**

**def** show\_value(data):

**try**:

val = data.value

**except** AttributeError:

logging.debug('No value yet')

**else**:

logging.debug('value=*%s*', val)

**def** worker(data):

show\_value(data)

data.value = random.randint(1, 100)

show\_value(data)

logging.basicConfig(

level=logging.DEBUG,

format='(*%(threadName)-10s*) *%(message)s*',

)

local\_data = threading.local()

show\_value(local\_data)

local\_data.value = 1000

show\_value(local\_data)

**for** i **in** range(2):

t = threading.Thread(target=worker, args=(local\_data,))

t.start()

The attribute local\_data.value is not present for any thread until it is set in that thread.

$ python3 threading\_local.py

(MainThread) No value yet

(MainThread) value=1000

(Thread-1 ) No value yet

(Thread-1 ) value=33

(Thread-2 ) No value yet

(Thread-2 ) value=74

To initialize the settings so all threads start with the same value, use a subclass and set the attributes in \_\_init\_\_().

*threading\_local\_defaults.py*

**import** **random**

**import** **threading**

**import** **logging**

**def** show\_value(data):

**try**:

val = data.value

**except** AttributeError:

logging.debug('No value yet')

**else**:

logging.debug('value=*%s*', val)

**def** worker(data):

show\_value(data)

data.value = random.randint(1, 100)

show\_value(data)

**class** **MyLocal**(threading.local):

**def** \_\_init\_\_(self, value):

super().\_\_init\_\_()

logging.debug('Initializing *%r*', self)

self.value = value

logging.basicConfig(

level=logging.DEBUG,

format='(*%(threadName)-10s*) *%(message)s*',

)

local\_data = MyLocal(1000)

show\_value(local\_data)

**for** i **in** range(2):

t = threading.Thread(target=worker, args=(local\_data,))

t.start()

\_\_init\_\_() is invoked on the same object (note the id() value), once in each thread to set the default values.

$ python3 threading\_local\_defaults.py

(MainThread) Initializing <\_\_main\_\_.MyLocal object at

0x101c6c288>

(MainThread) value=1000

(Thread-1 ) Initializing <\_\_main\_\_.MyLocal object at

0x101c6c288>

(Thread-1 ) value=1000

(Thread-1 ) value=18

(Thread-2 ) Initializing <\_\_main\_\_.MyLocal object at

0x101c6c288>

(Thread-2 ) value=1000

(Thread-2 ) value=77