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` 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 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) 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()
```

```
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 `Semaphore` is 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 = 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
```

See also

- Standard library documentation for threading
- Python 2 to 3 porting notes for threading
- thread – Lower level thread API.
- Queue – Thread-safe queue, useful for passing messages between threads.
- multiprocessing – An API for working with processes that mirrors the threading API.

signal — Asynchronous System Events

multiprocessing — Manage Processes Like Threads

Quick Links

This page was last updated 2016-12-29.

Navigation

signal — Asynchronous System Events
multiprocessing — Manage Processes Like Threads

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

- Blog
- The Python Standard Library By Example