threading – Manage concurrent threads

Purpose:	Builds on the **thread** module to more easily manage several threads of execution.
Available In:	1.5.2 and later

The **threading** module builds on the low-level features of **thread** to make working with threads even easier and more *pythonic*. 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.

```
import threading

def worker():
    """thread worker function"""
    print 'Worker'
    return

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:

```
$ python threading_simple.py

Worker
Worker
Worker
Worker
Worker
```

It useful to be able to spawn a thread and pass it arguments to tell it what work to do. This example passes a number, which the thread then prints.

```
import threading

def worker(num):
    """thread worker function"""
    print 'Worker: %s' % num
```

```
    return

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:

```
    $ python -u 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.

```
import threading
import time

def worker():
    print threading.currentThread().getName(), 'Starting'
    time.sleep(2)
    print threading.currentThread().getName(), 'Exiting'

def my_service():
    print threading.currentThread().getName(), 'Starting'
    time.sleep(3)
    print threading.currentThread().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**.

```
$ python -u threading_names.py

worker Thread-1 Starting
my_service Starting
Starting
Thread-1worker Exiting
 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 easier to trace those messages back to their source.

```
import logging
import threading
import time

logging.basicConfig(level=logging.DEBUG,
                    format='[%(levelname)s] (%(threadName)-10s)
%(message)s',
                    )

def worker():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

def my_service():
    logging.debug('Starting')
    time.sleep(3)
    logging.debug('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()
```

logging is also thread-safe, so messages from different threads are kept

distinct in the output.

```
$ python 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, call its **`setDaemon()`** method with a boolean argument. The default is for threads to not be daemons, so passing True turns the daemon mode on.

```
import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

def daemon():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

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

def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')
```

```
t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()
```

Notice that 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 its two second sleep.

```
$ python 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.

```
import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

def daemon():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

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

def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')

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.

```
$ python 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 timeout argument (a float 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.

```
import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

def daemon():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

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

def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')

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

d.start()
t.start()

d.join(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.

```
$ python 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 is not allowed (it introduces a deadlock situation), it must be skipped.

```
import random
import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

def worker():
    """thread worker function"""
    t = threading.currentThread()
    pause = random.randint(1,5)
    logging.debug('sleeping %s', pause)
    time.sleep(pause)
    logging.debug('ending')
    return

for i in range(3):
    t = threading.Thread(target=worker)
    t.setDaemon(True)
    t.start()

main_thread = threading.currentThread()
for t in threading.enumerate():
    if t is main_thread:
        continue
    logging.debug('joining %s', t.getName())
    t.join()
```

Since the worker is sleeping for a random amount of time, the output from this

program may vary. It should look something like this:

```
$ python threading_enumerate.py

(Thread-1  ) sleeping 3
(Thread-2  ) sleeping 2
(Thread-3  ) sleeping 5
(MainThread) joining Thread-1
(Thread-2  ) ending
(Thread-1  ) ending
(MainThread) joining Thread-3
(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.

```
import threading
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

class MyThread(threading.Thread):

    def run(self):
        logging.debug('running')
        return

for i in range(5):
    t = MyThread()
    t.start()
```

The return value of **`run()`** is ignored.

```
$ python 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, 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.

```
import threading
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

class MyThreadWithArgs(threading.Thread):

    def __init__(self, group=None, target=None, name=None,
                 args=(), kwargs=None, verbose=None):
        threading.Thread.__init__(self, group=group,
target=target, name=name,
                                  verbose=verbose)
        self.args = args
        self.kwargs = kwargs
        return

    def run(self):
        logging.debug('running with %s and %s', self.args,
self.kwargs)
        return

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.

```
$ python threading_subclass_args.py

(Thread-1  ) running with (0,) and {'a': 'A', 'b': 'B'}
(Thread-2  ) running with (1,) and {'a': 'A', 'b': 'B'}
(Thread-3  ) running with (2,) and {'a': 'A', 'b': 'B'}
(Thread-4  ) running with (3,) and {'a': 'A', 'b': 'B'}
(Thread-5  ) running with (4,) and {'a': 'A', 'b': 'B'}
```

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.

```
import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

def delayed():
    logging.debug('worker running')
    return

t1 = threading.Timer(3, delayed)
t1.setName('t1')
t2 = threading.Timer(3, delayed)
t2.setName('t2')

logging.debug('starting timers')
t1.start()
t2.start()

logging.debug('waiting before canceling %s', t2.getName())
time.sleep(2)
logging.debug('canceling %s', t2.getName())
t2.cancel()
logging.debug('done')
```

Notice that the second timer 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.

```
$ python 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 spin separate operations off to run concurrently, there are times when it is important to be able to synchronize the operations in two or more threads. A simple way to communicate between threads is using **`Event`** objects. An **`Event`** manages an internal flag that callers can either **`set()`** or **`clear()`**. Other threads can **`wait()`** for the flag to be **`set()`**, effectively blocking progress until allowed to continue.

```
import logging
import threading
import time

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

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.isSet():
        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')


e = threading.Event()
t1 = threading.Thread(name='block',
                      target=wait_for_event,
                      args=(e,))
t1.start()

t2 = threading.Thread(name='non-block',
                      target=wait_for_event_timeout,
                      args=(e, 2))
```

```
t2.start()

logging.debug('Waiting before calling Event.set()')
time.sleep(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 **`isSet()`** 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.

```
$ python threading_event.py

(block     ) wait_for_event starting
(non-block ) wait_for_event_timeout starting
(MainThread) Waiting before calling Event.set()
(non-block ) event set: False
(non-block ) doing other work
(non-block ) wait_for_event_timeout starting
(MainThread) Event is set
(block     ) event set: True
(non-block ) event set: True
(non-block ) processing event
```

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 GIL is not released in the middle of an update). Other data structures implemented in Python, or simpler types like integers and floats, don't have that protection. To guard against simultaneous access to an object, use a **`Lock`** object.

```
import logging
import random
import threading
```

```
import time

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

class Counter(object):
    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')

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.currentThread()
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.

```
$ python threading_lock.py
```

```
(Thread-1  ) Sleeping 0.47
(Thread-2  ) Sleeping 0.65
(MainThread) Waiting for worker threads
(Thread-1  ) Waiting for lock
(Thread-1  ) Acquired lock
(Thread-1  ) Sleeping 0.90
(Thread-2  ) Waiting for lock
(Thread-2  ) Acquired lock
(Thread-2  ) Sleeping 0.11
(Thread-2  ) Waiting for lock
(Thread-2  ) Acquired lock
(Thread-2  ) Done
(Thread-1  ) Waiting for lock
(Thread-1  ) Acquired lock
(Thread-1  ) 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.

```
import logging
import threading
import time

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

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)
    return

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)


lock = threading.Lock()

holder = threading.Thread(target=lock_holder, args=(lock,),
name='LockHolder')
holder.setDaemon(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.

```
$ python threading_lock_noblock.py

(LockHolder) Starting
(LockHolder) Holding
(Worker    ) Starting
(LockHolder) Not holding
(Worker    ) Trying to acquire
(Worker    ) Iteration 1: Acquired
(Worker    ) Trying to acquire
(LockHolder) Holding
(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.

```
import threading

lock = threading.Lock()

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

In this case, since both functions are using the same global lock, and one calls the other, the second acquisition fails and would have blocked using the default arguments to **`acquire()`**.

```
$ python 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.

```
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`**.

```
$ python threading_rlock.py
```

```
First try : True
Second try: 1
```

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.

```
import threading
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

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()

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.

```
$ python 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. This allows 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.

```
import logging
import threading
import time

logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s (%(threadName)-2s) %
(message)s',
                    )

def consumer(cond):
    """wait for the condition and use the resource"""
    logging.debug('Starting consumer thread')
    t = threading.currentThread()
    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()

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(2)
c2.start()
time.sleep(2)
p.start()
```

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

```
$ python threading_condition.py

2013-02-21 06:37:49,549 (c1) Starting consumer thread
2013-02-21 06:37:51,550 (c2) Starting consumer thread
2013-02-21 06:37:53,551 (p ) Starting producer thread
2013-02-21 06:37:53,552 (p ) Making resource available
2013-02-21 06:37:53,552 (c2) Resource is available to consumer
2013-02-21 06:37:53,553 (c1) Resource is available to consumer
```

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.

```
import logging
import random
import threading
import time

logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s (%(threadName)-2s) %
(message)s',
                    )

class ActivePool(object):
    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.currentThread().getName()
        pool.makeActive(name)
        time.sleep(0.1)
        pool.makeInactive(name)

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 only five are running concurrently.

```
$ python threading_semaphore.py

2013-02-21 06:37:53,629 (0 ) Waiting to join the pool
2013-02-21 06:37:53,629 (1 ) Waiting to join the pool
2013-02-21 06:37:53,629 (0 ) Running: ['0']
2013-02-21 06:37:53,629 (2 ) Waiting to join the pool
2013-02-21 06:37:53,630 (3 ) Waiting to join the pool
2013-02-21 06:37:53,630 (1 ) Running: ['0', '1']
2013-02-21 06:37:53,730 (0 ) Running: ['1']
2013-02-21 06:37:53,731 (2 ) Running: ['1', '2']
2013-02-21 06:37:53,731 (1 ) Running: ['2']
2013-02-21 06:37:53,732 (3 ) Running: ['2', '3']
2013-02-21 06:37:53,831 (2 ) Running: ['3']
2013-02-21 06:37:53,833 (3 ) 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 view in threads that do not “own” them. The **`local()`** function creates an object capable of hiding values from view in separate threads.

```
import random
import threading
import logging

logging.basicConfig(level=logging.DEBUG,
```

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

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)
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()
```

Notice that `local_data.value` is not present for any thread until it is set in that thread.

```
$ python threading_local.py

(MainThread) No value yet
(MainThread) value=1000
(Thread-1  ) No value yet
(Thread-1  ) value=34
(Thread-2  ) No value yet
(Thread-2  ) value=7
```

To initialize the settings so all threads start with the same value, use a subclass and set the attributes in **`__init__()`**.

```
import random
import threading
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )
```

```
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):
        logging.debug('Initializing %r', self)
        self.value = value


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.

```
$ python threading_local_defaults.py

(MainThread) Initializing <__main__.MyLocal object at
0x100514390>
(MainThread) value=1000
(Thread-1  ) Initializing <__main__.MyLocal object at
0x100514390>
(Thread-1  ) value=1000
(Thread-2  ) Initializing <__main__.MyLocal object at
0x100514390>
(Thread-1  ) value=81
(Thread-2  ) value=1000
(Thread-2  ) value=54
```

See also

threading

Standard library documentation for this module.

`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.