Chapter 30
Threading

30.1 Introduction

Threading is one of the ways in which Python allows you to write programs that multitask; that is *appearing* to do more than one thing at a time.

This chapter presents the `threading` module and uses a short example to illustrate how these features can be used.

30.2 Threads

In Python the `Thread` class from the `threading` module represents an activity that is run in a separate thread of execution within a single process. These threads of execution are lightweight, pre-emptive execution threads. A thread is *lightweight* because it does not possess its own address space and it is not treated as a separate entity by the host operating system; it is not a process. Instead, it exists within a single machine process using the same address space as other threads.

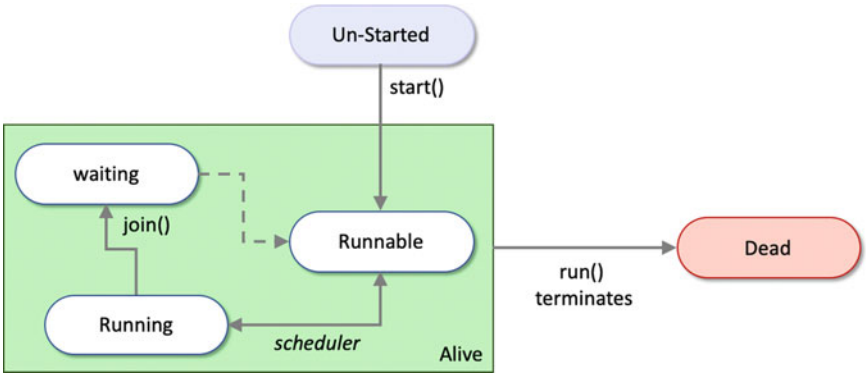
30.3 Thread States

When a thread object is first created it exists, but it is not yet runnable; it must be started. Once it has been started it is then runnable; that is, it is eligible to be scheduled for execution. It may switch back and forth between running and being runnable under the control of the scheduler. The scheduler is responsible for managing multiple threads that all wish to grab some execution time.

A thread object remains runnable or running until its `run()` method terminates; at which point it has finished its execution and it is now dead. All states between

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_30

un-started and dead are considered to indicate that the Thread is alive (and therefore may run at some point). This is shown below:



A Thread may also be in the *waiting* state; for example, when it is waiting for another thread to finish its work before continuing (possibly because it needs the results produced by that thread to continue). This can be achieved using the `join()` method and is also illustrated above. Once the second thread completes the waiting thread will again become runnable.

The thread which is currently executing is termed the *active* thread.

There are a few points to note about thread states:

- A thread is considered to be alive unless its `run()` method terminates after which it can be considered dead.
- A *live* thread can be running, runnable, waiting, etc.
- The `runnable` state indicates that the thread can be executed by the processor, but it is not currently executing. This is because an equal or higher priority process is already executing, and the thread must wait until the processor becomes free. Thus the diagram shows that the scheduler can move a thread between the running and runnable state. In fact, this could happen many times as the thread executes for a while, is then removed from the processor by the scheduler and added to the waiting queue, before being returned to the processor again at a later date.

30.4 Creating a Thread

There are two ways in which to initiate a new thread of execution:

- Pass a reference to a callable object (such as a function or method) into the Thread class constructor. This reference acts as the target for the Thread to execute.

- Create a subclass of the `Thread` class and redefine the `run()` method to perform the set of actions that the thread is intended to do.

We will look at both approaches.

As a thread is an object, it can be treated just like any other object: it can be sent messages, it can have instance variables and it can provide methods. Thus, the multi-threaded aspects of Python all conform to the object-oriented model. This greatly simplifies the creation of multi-threaded systems as well as the maintainability and clarity of the resulting software.

Once a new instance of a thread is created, it must be started. Before it is started, it cannot run, although it exists.

30.5 Instantiating the Thread Class

The `Thread` class can be found in the `threading` module and therefore must be imported prior to use. The class `Thread` defines a single constructor that takes up to six optional arguments:

```
class threading.Thread(group=None,
                       target=None,
                       name=None,
                       args=(),
                       kwargs={},
                       daemon=None)
```

The Thread constructor should *always* be called using *keyword* arguments; the meaning of these arguments is:

- `group` should be None; reserved for future extension when a ThreadGroup class is implemented.
- `target` is the callable object to be invoked by the `run()` method. Defaults to `None`, meaning nothing is called.
- `name` is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is an integer.
- `args` is the argument tuple for the target invocation. Defaults to `()`. If a single argument is provided the tuple is not required. If multiple arguments are provided then each argument is an element within the tuple.
- `kwargs` is a dictionary of keyword arguments for the target invocation. Defaults to `{}`.
- `daemon` indicates whether this thread runs as a daemon thread or not. If not `None`, daemon explicitly sets whether the thread is daemonic. If None (the default), the daemonic property is inherited from the current thread.

Once a `Thread` is created it must be started to become eligible for execution using the `Thread.start()` method.

The following illustrates a very simple program that creates a `Thread` that will run the `simple_worker()` function:

```
from threading import Thread

def simple_worker():
    print('hello')

# Create a new thread and start it
# The thread will run the function simple_worker
t1 = Thread(target=simple_worker)
t1.start()
```

In this example, the thread `t1` will execute the function `simple_worker`. The main code will be executed by the *main* thread that is present when the program starts; there are thus two threads used in the above program; *main* and *t1*.

30.6 The Thread Class

The `Thread` class defines all the facilities required to create an object that can execute within its own lightweight process. The key methods are:

- `start()` Start the thread's activity. It must be called at *most* once per thread object. It arranges for the object's `run()` method to be invoked in a separate thread of control. This method will raise a `RuntimeError` if called more than once on the same thread object.
- `run()` Method representing the thread's activity. You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object's constructor as the target argument, if any, with positional and keyword arguments taken from the `args` and `kwargs` arguments, respectively. You should *not* call this method directly.
- `join(timeout = None)` Wait until the thread sent this message terminates. This blocks the calling thread until the thread whose `join()` method is called terminates. When the timeout argument is present and not None, it should be a floating-point number specifying a timeout for the operation in seconds (or fractions thereof). A thread can be join()ed many times.
- `name` A string used for identification purposes only. It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor. Giving a thread a name can be useful for debugging purposes.
- `ident` The 'thread identifier' of this thread or `None` if the thread has not been started. This is a nonzero integer.

- `is_alive()` Return whether the thread is alive. This method returns `True` just before the `run()` method starts until just after the `run()` method terminates. The module function `threading.enumerate()` returns a list of all alive threads.
- `daemon` A boolean value indicating whether this thread is a daemon thread (`True`) or not (`False`). This must be set before `start()` is called, otherwise a `RuntimeError` is raised. Its default value is inherited from the creating thread. The entire Python program exits when no alive non-daemon threads are left.

An example illustrating using some of these methods is given below:

```
from threading import Thread

def simple_worker():
    print('hello')

t1 = Thread(target=simple_worker)
t1.start()

print(t1.getName())
print(t1.ident)
print(t1.is_alive())
```

This produces:

```
hello
Thread-1
123145441955840
True
```

The `join()` method can cause one thread to wait for another to complete. For example, if we want the *main* thread to wait until a thread completes before it prints the done message; then we can make it *join* athatthread:

```
from threading import Thread
from time import sleep

def worker():
    for i in range(0, 10):
        print('.', end='', flush=True)
        sleep(1)

print('Starting')

# Create read object with reference to worker function
t = Thread(target=worker)
# Start the thread object
```

```
t.start()
# Wait for the thread to complete

t.join()

print('\nDone')
```

Now the 'Done' message should not be printed out until after the worker thread has finished as shown below:

```
Starting
..........
Done
```

30.7 The Threading Module Functions

There are a set of `threading` module functions which support working with threads; these functions include:

- `threading.active_count()` Return the number of Thread objects currently alive. The returned count is equal to the length of the list returned by `enumerate()`.
- `threading.current_thread()` Return the current Thread object, corresponding to the caller's thread of control. If the caller's thread of control was not created through the `threading` module, a dummy thread object with limited functionality is returned.
- `threading.get_ident()` Return the 'thread identifier' of the current thread. This is a nonzero integer. Thread identifiers may be recycled when a thread exits and another thread is created.
- `threading.enumerate()` Return a list of all Thread objects currently alive. The list includes daemon threads, dummy thread objects created by `current_thread()` and the *main* thread. It excludes terminated threads and threads that have not yet been started.
- `threading.main_thread()` Return the *main* Thread object.

30.8 Passing Arguments to a Thread

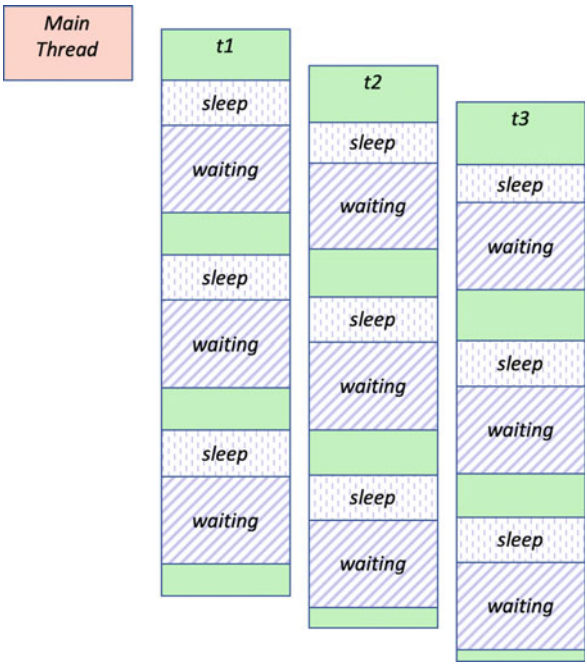
Many functions expect to be given a set of parameter values when they are run; these arguments still need to be passed to the function when they are run via a separate thread. These parameters can be passed to the function to be executed via the `args` parameter, for example:

```
from threading import Thread
from time import sleep

def worker(msg):
    for i in range(0, 10):
        print(msg, end='', flush=True)
        sleep(1)

print('Starting')
t1 = Thread(target=worker, args='A')
t2 = Thread(target=worker, args='B')
t3 = Thread(target=worker, args='C')
t1.start()
t2.start()
t3.start()

print('Done')
```

In this example, the `worker` function takes a message to be printed 10 times within a loop. Inside the loop the thread will print the message and then sleep for a second. This allows other threads to be executed as the `Thread` must wait for the sleep timeout to finish before again becoming runnable.

Three threads t1, t2 and t3 are then created each with a different message. Note that the `worker()` function can be reused with each `Thread` as each invocation of the function will have its own parameter values passed to it.

The three threads are then started. This means that at this point there is the *main* thread, and three worker threads that are *Runnable* (although only one thread will run at a time). The three worker threads each run the `worker()` function printing out either the letter A, B or C ten times. This means that once started each thread will print out a string, sleep for 1 s and then wait until it is selected to run again, this is illustrated in the following diagram:



The output generated by this program is illustrated below:

```
Starting
ABCDone
ABCACBABCABCCBAABCABCABCBAC
```

Notice that the *main* thread is finished after the worker threads have only printed out a single letter each; however as long as there is at least one *non-daemon* thread running the program will not terminate; as none of these threads are marked as a daemon thread the program continues until the last thread has finished printing out the tenth of its letters.

Also notice how each of the threads gets a chance to run on the processor before it sleeps again; thus we can see the letters A, B and C all mixed in together.

30.9 Extending the Thread Class

The second approach to creating a `Thread` mentioned earlier was to subclass the `Thread` class. To do this you must

1. Define a new subclass of `Thread`.
2. Override the `run()` method.
3. Define a new `__init__()` method that calls the parent class `__init__()` method to pass the required parameters up to the `Thread` class constructor.

This is illustrated below where the `WorkerThread` class passes the `name`, `target` and `daemon` parameters up to the Thread super class constructor.

```
from threading import Thread
from time import sleep

class WorkerThread(Thread):
    def __init__(self, daemon=None, target=None, name=None):
        super().__init__(daemon=daemon, target=target,
                         name=name)
    def run(self):
        for i in range(0, 10):
            print('.', end='', flush=True)
            sleep(1)
```

Once you have done this you can create an instance of the new `WorkerThread` class and then start that instance.

```
print('Starting')
t = WorkerThread()
t.start()
print('\nDone')
```

The output from this is:

```
Starting
.
Done
.........
```

Note that it is common to call any subclasses of the `Thread` class, *Something*`Thread`, to make it clear that it is a subclass of the `Thread` class and should be treated as if it was a Thread (which of course it is).

30.10 Daemon Threads

A thread can be marked as a *daemon* thread by setting the daemon property to true either in the constructor or later via the accessor property.

For example:

```
from threading import Thread
from time import sleep

def worker(msg):
    for i in range(0, 10):
        print(msg, end='', flush=True)
        sleep(1)

print('Starting')

# Create a daemon thread
d = Thread(daemon=True, target=worker, args='C')
d.start()

sleep(5)
print('Done')
```

This creates a background daemon thread that will run the function `worker()`. Such threads are often used for house keeping tasks (such as background data backups etc.).

As mentioned above a daemon thread is not enough on its own to keep the current program from terminating. This means that the daemon thread will keep looping until the main thread finishes. As the main thread sleeps for 5 s that allows the daemon thread to print out about 5 strings before the main thread terminates. This is illustrated by the output below:

```
Starting
CCCCCDone
```

30.11 Naming Threads

Threads can be named; which can be very useful when debugging an application with multiple threads.

In the following example, three threads have been created; two have been explicitly given a name related to what they are doing while the middle one has been left with the default name. We then start all three threads and use the `threading.enumerate()` function to loop through all the currently live threads printing out their names:

```
import threading
from threading import Thread
from time import sleep

def worker(msg):
    for i in range(0, 10):
        print(msg, end='', flush=True)
        sleep(1)

t1 = Thread(name='worker', target=worker, args='A')
t2 = Thread(target=worker, args='B')  # use default name e.g.
Thread-1
d = Thread(daemon = True, name='daemon', target=worker,
args='C')

t1.start()
t2.start()
d.start()

print()
for t in threading.enumerate():
    print(t.getName())
```

The output from this program is given blow:

```
ABC
MainThread
worker
Thread-1
daemon
ABCBACACBCBACBAABCCBACBACBA
```

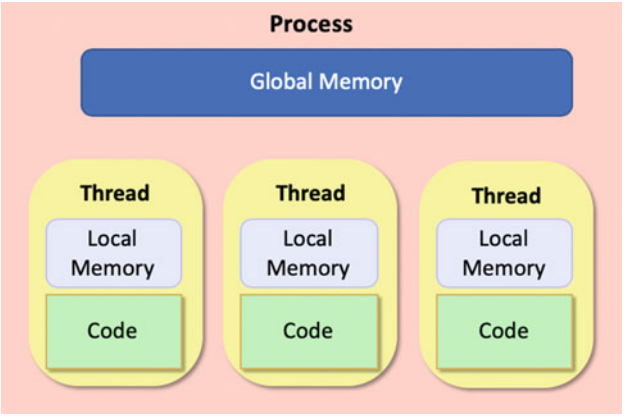
As you can see in addition to the worker thread and the daemon thread there is a `MainThread` (that initiates the whole program) and `Thread-1` which is the thread referenced by the variable `t2` and uses the default thread name.

30.12 Thread Local Data

In some situations each `Thread` requires its own copy of the data it is working with; this means that the shared (heap) memory is difficult to use as it is inherently shared between all threads.

To overcome this Python provides a concept known as *Thread-Local* data.

Thread-local data is data whose values are associated with a thread rather than with the shared memory. This idea is illustrated below:



To create thread-local data it is only necessary to create an instance of `threading.local` (or a subclass of this) and store attributes into it. The instances will be thread specific; meaning that one thread will not see the values stored by another thread.

For example:

```
from threading import Thread, local, currentThread
from random import randint

def show_value(data):
    try:
        val = data.value
    except AttributeError:
        print(currentThread().name, ' - No value yet')
    else:
        print(currentThread().name, ' - value =', val)

def worker(data):
    show_value(data)
    data.value = randint(1, 100)
    show_value(data)

print(currentThread().name, ' - Starting')
local_data = local()
show_value(local_data)
```

```
for i in range(2):
    t = Thread(name='W' + str(i),
               target=worker, args=[local_data])
    t.start()

show_value(local_data)
print(currentThread().name, ' - Done')
```

The output from this is

```
MainThread - Starting
MainThread - No value yet
W0 - No value yet
W0 - value = 20
W1 - No value yet
W1 - value = 90
MainThread - No value yet
MainThread - Done
```

The example presented above defines two functions.

- The first function attempts to access a value in the thread local data object. If the value is not present an exception is raised (`AttributeError`). The `show_value()` function catches the exception or successfully processes the data.
- The worker function calls `show_value()` twice, once before it sets a value in the local data object and once after. As this function will be run by separate threads the `currentThread` name is printed by the `show_value()` function.

The main function crates a local data object using the `local()` function from the threading library. It then calls `show_value()` itself. Next it creates two threads to execute the worker function in passing the `local_data` object into them; each thread is then started. Finally, it calls `show_value()` again.

As can be seen from the output one thread cannot see the data set by another thread in the `local_data` object (even when the attribute name is the same).

30.13 Timers

The `Timer` class represents an action (or task) to run after a certain amount of time has elapsed. The `Timer` class is a subclass of `Thread` and as such also functions as an example of creating custom threads.

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

The signature of the `Timer` class constructor is:

```
Timer(interval, function, args = None, kwargs = None)
```

An example of using the `Timer` class is given below:

```
from threading import Timer

def hello():
    print('hello')

print('Starting')
t = Timer(5, hello)
t.start()

print('Done')
```

In this case the `Timer` will run the hello function after an initial delay of 5 s.

30.14 The Global Interpreter Lock

The Global Interpreter Lock (or the GIL) is a *global lock* within the underlying CPython interpreter that was designed to avoid potential deadlocks between multiple tasks. It is designed to protect access to Python objects by preventing multiple threads from executing at the same time.

For the most part you do not need to worry about the GIL as it is at a lower level than the programs you will be writing.

However, it is worth noting that the GIL is controversial because it prevents multithreaded Python programs from taking full advantage of multiprocessor systems in certain situations.

This is because in order to execute a thread must obtain the GIL and only one thread at a time can hold the GIL (that is the lock it represents). This means that Python acts like a single CPU machine; only one thing can run at a time. A Thread will only give up the GIL if it sleeps, has to wait for something (such as some I/O)

or it has held the GIL for a certain amount of time. If the maximum time that a thread can hold the GIL has been met the scheduler will release the GIL from that thread (resulting it stopping execution and now having to wait until it has the GIL returned to it) and will select another thread to gain the GIL and start to execute.

It is thus impossible for standard Python threads to take advantage of the multiple CPUs typically available on modern computer hardware.

One solution to this is to use the Python `multiprocessing` library described in the next chapter.

30.15 Online Resources

See the following online resources for information on the topics in this chapter:

- https://docs.python.org/3/library/threading.html The Python standard Library documentation on Threading.
- https://pymotw.com/3/threading/ The Python Module of the Week page on Threading.
- https://pythonprogramming.net/threading-tutorial-python/ Tutorial on Python's Threading module.

30.16 Exercise

Create a function called `printer()` that takes a message and a maximum value to use for a period to sleep.

Within the function create a loop that iterates 10 times. Within the loop

- generate a random number from 0 to the max period specified and then sleep for that period of time. You can use the `random.randint()` function for this.
- Once the sleep period has finished print out the message passed into the function.
- Then loop again until this has been repeated 10 times.

Now create five threads to run five invocations of the function you produced above and start all five threads. Each thread should have a different `max_sleep` time.

An example program to run the printer function five times via a set of Threads is given below:

```
t1 = Thread(target=printer, args=('A', 10))
t2 = Thread(target=printer, args=('B', 5))
t3 = Thread(target=printer, args=('C', 15))
t4 = Thread(target=printer, args=('D', 7))
t5 = Thread(target=printer, args=('E', 2))
t1.start()
```

```
t2.start()
t3.start()
t4.start()
t5.start()
```

An example of the sort of output this could generate is given below:

```
BAEAEABEDAEAEBEDCECBEEEADCDBBDABCADBBDABADCDCDCCCC
```