Part 83
-
Multi-Tasking
-
Threading

Version: 2021-01-14

Threading

Threading is a higher-level interface and uses the standard thread module internally. Threading has been supported since Python 2.4 and provides higher-level support.
Threading uses all methods of the thread module and provides support for the following additional methods:

- `threading.activeCount()` : Returns the number of thread objects that are active
- `threading.currentThread()`: Returns the number of thread objects in the caller's thread
- `threading.enumerate()` : Returns a list of all thread objects that are currently active

The threading module also provides the following options:

- `run()` : entry point of a thread
- `start()` : starts a thread by calling the run method
- `join([time])` : waits for threads to terminate
- `isAlive()` : checks whether a thread is still executing
- `getName()` : returns the name of a thread
- `setName()` : sets the name of a thread
- `is_alive()` : returns whether the thread is alive

Single thread

In example program below, function `newprocess` is started as a new thread and displays the message `Hello from new process...` The thread is started with the function call `thread.start()`. Function `thread.join()` waits until the thread is complete and then displays the message `Hello from the creator...` The function `thread.join()` blocks indefinitely until all threads exit.

We can specify a floating-point timeout value, for example, `thread.join(2)` so that the program exits after 2 seconds even if the threads do not complete within this timeout value.

```
import threading

def newprocess():
   print("Hello from new process...")

thread = threading.Thread(target = newprocess, args=())
thread.start()
thread.join()
print("Hello from the creator...")
```

The following output will be displayed when the program is run:

```
python3 threading1.py
Hello from new process...
Hello from the creator...
```

We can give a name to a thread. For example, let us name the thread created by function newprocess TASK1 and then display the name of this thread as shown in the program below

```
import threading

def newprocess():
   thread.setName("Task1")
   name = thread.getName()
   print(name)
   print("Hello from new process...")

thread = threading.Thread(target = newprocess, args=())
thread.start()
thread.join()
print("Hello fro the creator...")
```

The following output will be displayed when the program is run:

```
python3 threading2.py
Task1
Hello from new process...
Hello from the creator...
```

Multiple threads
In the example program below multiple threads are created and data is passed to each thread. 5 threads are created in a loop and the loop count is passed as an argument to the created threads

```
import threading

def newprocess(j):
   print("Hello from new process %d" %j)

for i in range(5):
   t = threading.Thread(target = newprocess, args=(i, ))
   t.start()
```

The following output will be displayed when the program is run:

```
python3 threading3.py
Hello from new process 0
Hello from new process 1
Hello from new process 2
Hello from new process 3
Hello from new process 4
```

Thread lock objects
Lock objects are synchronization primitives used to synchronize threads. A lock can be in one of two states: locked or unlocked. It is created in the unlocked state and has two methods: `acquire()` and `release()`.
When the state is `locked()`, `acquire()` blocks until a call to `release()` in another thread changes it to unlocked. When the state is unlocked, `acquire()` changes the state to locked and immediately returns. The `release()` method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, an error will be raised.
`acquire()` has blocking or non-blocking as the first argument where by default it is blocking. An optional timeout can be specified as its second argument to specify the number of seconds to block.

In this example, a shared variable called `count` is used. Thread `Task1` locks the shared variable `count` and releases it after incrementing by 1. `Task2` locks the shared variable `count` and releases it after incrementing by 2.

```
import threading

lock = threading.Lock()
count = 0

def Task1():
  global count
  # Lock count
  lock.acquire()
  # Increment count
  count = count + 1
  # Release count
  lock.release()

def Task2():
  global count
  # Lock count
  lock.acquire()
  # Increment count
  count = count + 2
  # Release count
  lock.release()

t1 = threading.Thread(target = Task1, args=())
t2 = threading.Thread(target = Task2, args=())
t1.start()
t2.start()
t1.join()
t2.join()
print(count)
```

The following output will be displayed when the program is run:

```
python3 threading4.py
3
```

Semaphores
A semaphore is an advanced lock mechanism. It is used to limit access to a resource with limited capacity. A semaphore has an internal counter which is decremented when the semaphore is acquired and incremented when the semaphore is released. If the counter reaches zero when acquired, the acquiring thread will block. When the semaphore is incremented again, one of the blocking threads will run.
The semaphore counter must be initialized before used. If not initialized, a count of 1 is assumed by default. For example, to set the counter to 5, use the statements:

```
max = 5
semaphore = threading.BoundedSemaphore(max)
```

An example below shows the semaphore count is set to 1 and after acquiring the semaphore numbers 0 to 4 are displayed by `Task1`. After the semaphore is released, numbers 0 to 4 are displayed by `Task2`.

```
import threading

sema = threading.BoundedSemaphore(1)
count = 0

def Task1():
  # acuire semaphore
  sema.acquire()
  for i in range(4):
    print(i)
  # release semaphore
  sema.release()

def Task2():
  # acquire semaphore
  sema.acquire()
  for i in range(4):
    print(i)
  # release semaphore
  sema.release()

t1 = threading.Thread(target = Task1, args=())
t2 = threading.Thread(target = Task2, args=())
t1.start()
t2.start()
t1.join()
t2.join()
```

The following output will be displayed when the program is run:

```
python3 threading5.py
0
1
2
3
0
1
2
3
```

Events
Events are simple synchronization objects using internal flags. Threads can wait for the flag to be set, or set or clear the flags.

The following event operations are available

- `set()` : set the internal flag to true
- `clear()` : reset (clear) the internal flag to false
- `wait()` : block until the internal flag is set to true
- `is_set()` : returns True if the internal flag is set true

The example program below shows thread `Task1` waits until the event flag is set. Thread `Task2` prompts the user to enter a key. After entering a key, the event flag is set and `Task1` displays a message `Event occurred...`

```
import threading
event = threading.Event()

def Task1():
  # Wait for the event
  event.wait()
  print("Event occurred...")

def Task2():
   a = input("Enter a key to set the event:")
   event.set()

t1 = threading.Thread(target = Task1, args=())
t2 = threading.Thread(target = Task2, args=())
t1.start()
t2.start()
t1.join()
t2.join()
```

The following output will be displayed when the program is run:

```
python3 threading6.py
Enter a key to set the event:a
Event occurred...
```

Timer threading
Python also supports timer threads. A timer thread is a delayed thread that starts after the specified time delay. In the program example below `TimerThread` starts working 5 seconds after it is started by the `T.start()` function.

```
import threading
import time

def TimerThread():
   print("TimerThread started.")

print("TimerThread will start after 5 seconds.")
T = threading.Timer(5, TimerThread)
T.start()
```

The following output will be displayed when the program is run:

```
python3 threading7.py
TimerThread will start after 5 seconds.
TimerThread started.
```