California State University Fullerton

CPSC-223P

Python Programming

Stephen T. May

Python Standard Library

Concurrent Execution - threading

https://docs.python.org/release/3.9.6/tutorial/index.html

Slide Notes

- Command typed at the Linux command prompt ($)

```
$ python3.9
```

- Command typed at the Python interpreter command prompt (>>>)

```
>>> Ctrl-D
```

- Python source code

```
print(“Hello world!”)
```

- Mixed example

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

`threading` module functions

- `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.enumerate()`

  Return a list of all Thread objects currently active. The list includes daemonic threads and dummy thread objects created by current_thread(). It excludes terminated threads and threads that have not yet been started. However, the main thread is always part of the result, even when terminated.

- `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.main_thread()`

  Return the main Thread object. In normal conditions, the main thread is the thread from which the Python interpreter was started.

`Thread` Objects

- The `Thread` class represents an activity that is run in a separate thread of control.
- There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the `run()` method in a subclass.
- No other methods (except for the constructor) should be overridden in a subclass.
- In other words, only override the `__init__()` and `run()` methods of this class.
- Once a thread object is created, its activity must be started by calling the thread's `start()` method.
- This invokes the `run()` method in a separate thread of control.
- Once the thread's activity is started, the thread is considered 'alive'.
- It stops being alive when its `run()` method terminates – either normally, or by raising an unhandled exception.
- The `is_alive()` method tests whether the thread is alive.
- Other threads can call a thread's `join()` method.
- This blocks the calling thread until the thread whose `join()` method is called is terminated.
- A thread has a `name`.
- The name can be passed to the constructor, and read or changed through the name attribute.
- If the `run()` method raises an exception, `threading.excepthook()` is called to handle it.
- By default, `threading.excepthook()` ignores silently `SystemExit`.

`Thread` class

- class `threading.Thread`(*group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None*)
- This constructor should always be called with keyword arguments.
- Arguments are:
- ***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 a small decimal number.
- ***args*** is the argument tuple for the target invocation. Defaults to ().
- ***kwargs*** is a dictionary of keyword arguments for the target invocation. Defaults to {}.
- If not None, ***daemon*** explicitly sets whether the thread is daemonic. If None (the default), the daemonic property is inherited from the current thread.
- If the subclass overrides the constructor, it must make sure to invoke the base class constructor (`Thread.__init__()`) before doing anything else to the thread.

`Thread` class methods and properties

- `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.

- `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.

- `join(timeout=None)`

  Wait until the thread terminates. This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception – or until the optional timeout occurs.

- `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 enumerate() returns a list of all alive threads.

`Lock` Objects

- A primitive lock is in one of two states, “locked” or “unlocked”.
- It is created in the unlocked state. It has two basic methods, `acquire()` and `release()`.
- When the state is unlocked, `acquire()` changes the state to locked and returns immediately.
- When the state is locked, `acquire()` blocks until a call to `release()` in another thread changes it to unlocked, then the `acquire()` call resets it to locked and 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, a `RuntimeError` will be raised.
- When more than one thread is blocked in `acquire()` waiting for the state to turn to unlocked, only one thread proceeds when a `release()` call resets the state to unlocked.

`Lock` class and methods

class `threading.Lock`

- Once a thread has acquired a lock, subsequent attempts to acquire it block, until it is released; any thread may release it.

`acquire(blocking=True, timeout=-1)`

- Acquire a lock, blocking or non-blocking.
- When invoked with the *blocking* argument set to `True` (the default), block until the lock is unlocked, then set it to locked and return `True`.
- When invoked with the *blocking* argument set to `False`, do not block. If a call with *blocking* set to `True` would block, return `False` immediately; otherwise, set the lock to locked and return `True`.
- When invoked with the floating-point *timeout* argument set to a positive value, block for at most the number of seconds specified by *timeout* and as long as the lock cannot be acquired.
- The return value is `True` if the lock is acquired successfully, `False` if not.

`release()`

- Release a lock. This can be called from any thread, not only the thread which has acquired the lock.
- When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.
- When invoked on an unlocked lock, a `RuntimeError` is raised.
- There is no return value.

`locked()`

- Return true if the lock is acquired.

`Condition` Objects

- A condition variable is always associated with some kind of lock; this can be passed in or one will be created by default.
- Passing one in is useful when several condition variables must share the same lock.
- The lock is part of the condition object: you don't have to track it separately.
- A condition variable obeys the *context management protocol*: using the with statement acquires the associated lock for the duration of the enclosed block.
- The `acquire()` and `release()` methods also call the corresponding methods of the associated lock.
- Other methods must be called with the associated lock held.
- The `wait()` method releases the lock, and then blocks until another thread awakens it by calling `notify()` or `notify_all()`.
- Once awakened, `wait()` re-acquires the lock and returns.
- It is also possible to specify a timeout.

- The `notify()` method wakes up one of the threads waiting for the condition variable, if any are waiting.
- The `notify_all()` method wakes up all threads waiting for the condition variable.
- Note: the `notify()` and `notify_all()` methods don't release the lock; this means that the thread or threads awakened will not return from their `wait()` call immediately, but only when the thread that called `notify()` or `notify_all()` finally relinquishes ownership of the lock.

`Condition` class and methods

class `threading.Condition(lock=None)`

- This class implements condition variable objects. A condition variable allows one or more threads to wait until they are notified by another thread.

`acquire(*args)`

- Acquire the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.

`release()`

- Release the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.

`wait(timeout=None)`

- Wait until notified or until a timeout occurs. If the calling thread has not acquired the lock when this method is called, a RuntimeError is raised.

`wait_for(predicate, timeout=None)`

- Wait until a condition evaluates to true. predicate should be a callable which result will be interpreted as a boolean value. A timeout may be provided giving the maximum time to wait.

`notify(n=1)`

- By default, wake up one thread waiting on this condition, if any. If the calling thread has not acquired the lock when this method is called, a RuntimeError is raised.

`notify_all()`

- Wake up all threads waiting on this condition. This method acts like notify(), but wakes up all waiting threads instead of one. If the calling thread has not acquired the lock when this method is called, a RuntimeError is raised.

`Event` Objects

- This is one of the simplest mechanisms for communication between threads: one thread signals an event and other threads wait for it.
- An event object manages an internal flag that can be set to true with the `set()` method and reset to false with the `clear()` method.
- The `wait()` method blocks until the flag is true.

`Event` class and methods

class `threading.Event`

- Class implementing event objects.
- An event manages a flag that can be set to true with the `set()` method and reset to false with the `clear()` method.
- The `wait()` method blocks until the flag is true. The flag is initially false.

`is_set()`

- Return `True` if and only if the internal flag is true.

`set()`

- Set the internal flag to true.
- All threads waiting for it to become true are awakened.
- Threads that call `wait()` once the flag is true will not block at all.

`clear()`

- Reset the internal flag to false.
- Subsequently, threads calling `wait()` will block until `set()` is called to set the internal flag to true again.

`wait(timeout=None)`

- Block until the internal flag is true.
- If the internal flag is true on entry, return immediately.
- Otherwise, block until another thread calls `set()` to set the flag to true, or until the optional timeout occurs.
- This method returns `True` if and only if the internal flag has been set to true, either before the wait call or after the wait starts, so it will always return True except if a timeout is given and the operation times out.