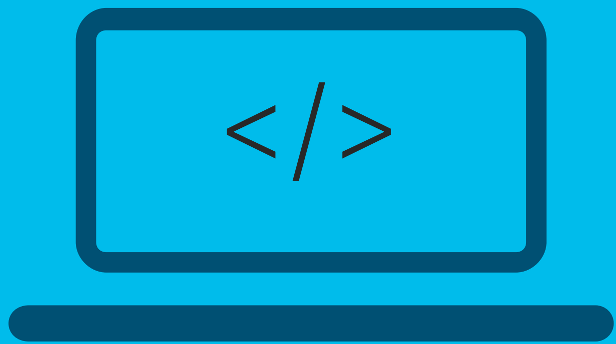
Coding Hour – Cisco EMEAR

[broadcasting from Frankfurt]



23 July 2020

DEEP PYTHON SERIES

[Concurrency in Python Using Concurrent.futures]

Today Speaker

Johannes Krohn @BU

23 July 2020

Agenda

Code, examples available at
https://github.com/jeokrohn/EMEAR_Coding_threads_processes

- Concurrency, …
- Threads
- Race conditions, locks
- concurrent.futures module
- “futures”
- Python Performance
- Threads vs Processes
- Summary

Concurrency, ...

Parallelism, Concurrency, ...

- Parallelism: performing multiple operations at (virtually) the same time
  - Multiprocessing: spreading tasks over CPUs/cores
  - Good for CPU bound tasks
- Concurrency: multiple tasks can run in an overlapping manner
  - Does not imply parallelism
- Threading: concurrent execution model, threads take turns
  - Even in multi-processor environments only one thread runs at any given time: Global Interpreter Lock (GIL)
  - Better for I/O bound tasks

Threads

`threading`: Starting a Thread

- Creating a thread: threading.Thread()
  - target: function to execute in thread context
  - args: tuple of arguments to pass to target for execution
  - kwargs: dictionary w/ keyword arguments to pass to target for execution

Demo: 01-threads.py

`threading`: Starting a Thread

- Threads start immediately, main thread suspended
- Main thread only terminates after all threads are done

```
10:24:52,443 [INFO] [MainThread] __main__: creating threads
10:24:52,444 [INFO] [MainThread] __main__: starting threads
10:24:52,444 [INFO] [Thread-1] __main__: wait_some_time(5): before sleep
10:24:52,444 [INFO] [Thread-2] __main__: wait_some_time(4): before sleep
10:24:52,445 [INFO] [Thread-3] __main__: wait_some_time(3): before sleep
10:24:52,445 [INFO] [Thread-4] __main__: wait_some_time(2): before sleep
10:24:52,445 [INFO] [Thread-5] __main__: wait_some_time(1): before sleep
10:24:52,445 [INFO] [MainThread] __main__: Done
10:24:53,450 [INFO] [Thread-5] __main__: wait_some_time(1): done
10:24:54,449 [INFO] [Thread-4] __main__: wait_some_time(2): done
10:24:55,449 [INFO] [Thread-3] __main__: wait_some_time(3): done
10:24:56,449 [INFO] [Thread-2] __main__: wait_some_time(4): done
10:24:57,449 [INFO] [Thread-1] __main__: wait_some_time(5): done
```

Race Conditions

- Execution can be switched between threads at any point
- Can lead to “race conditions” between threads
- Trivial example: Read-Update-Write on same resource from two threads

Demo: 02-race_condition.py

Race Conditions

- Waiting for thread to finish:

  `thread.join()`

- Parallel Read-Increment-Write operations
- Threads get interrupted between read and write
- Some increments get lost
- Even x += 1 can get interrupted

```
>>> import dis
>>> def update(x, v):
...     x += v
...
>>> dis.dis(update)
  2           0 LOAD_FAST                0 (x)
              2 LOAD_FAST                1 (v)
              4 INPLACE_ADD
              6 STORE_FAST               0 (x)
              8 LOAD_CONST               0
(None)
             10 RETURN_VALUE
```

```
10:36:42,453 [INFO] [MainThread] __main__: creating threads
10:36:42,453 [INFO] [MainThread] __main__: starting threads
10:36:42,454 [INFO] [Thread-1] __main__: doing some preparation
10:36:42,454 [INFO] [Thread-2] __main__: doing some preparation
10:36:42,454 [INFO] [Thread-3] __main__: doing some preparation
10:36:42,454 [INFO] [Thread-4] __main__: doing some preparation
10:36:42,454 [INFO] [Thread-5] __main__: doing some preparation
10:36:42,455 [INFO] [MainThread] __main__: threads started
10:36:42,584 [INFO] [Thread-4] __main__: previous value: 0
10:36:42,668 [INFO] [Thread-3] __main__: previous value: 0
10:36:43,072 [INFO] [Thread-1] __main__: previous value: 0
10:36:44,019 [INFO] [Thread-2] __main__: previous value: 0
10:36:44,174 [INFO] [Thread-3] __main__: Done, set new value: 1
10:36:44,180 [INFO] [Thread-4] __main__: Done, set new value: 1
10:36:44,584 [INFO] [Thread-1] __main__: Done, set new value: 1
10:36:44,743 [INFO] [Thread-5] __main__: previous value: 1
10:36:45,565 [INFO] [Thread-2] __main__: Done, set new value: 1
10:36:46,325 [INFO] [Thread-5] __main__: Done, set new value: 2
10:36:46,325 [INFO] [MainThread] __main__: Done, final value: 2
```

Thread Synchronization: Locks

- Lock: mutually exclusive ownership
- only one thread can “own” a lock
  - .acquire(): get ownership, waits until lock becomes available
  - .release(): release ownership, allows another thread to acquire ownership

Demo: 03-lock.py

Thread Synchronization: Locks

- Nesting critical operation between acquire() and release() prevents race conditions
- Locks can act as context manager

```
log.info('acquiring lock')
with counter_lock:
    log.info('acquired lock')

    val = counter
    log.info(f'previous value: {val}')

    time.sleep(random.uniform(1.5, 1.6))
    val += 1
    counter = val
    log.info(f'Done, set new value: {val}')

    log.info('releasing lock')
```

```
10:54:44,049 [INFO] [MainThread] __main__: creating threads
10:54:44,050 [INFO] [MainThread] __main__: starting threads
10:54:44,050 [INFO] [Thread-1] __main__: doing some preparation
10:54:44,050 [INFO] [Thread-2] __main__: doing some preparation
10:54:44,051 [INFO] [Thread-3] __main__: doing some preparation
10:54:44,051 [INFO] [Thread-4] __main__: doing some preparation
10:54:44,051 [INFO] [Thread-5] __main__: doing some preparation
10:54:44,051 [INFO] [MainThread] __main__: threads started
10:54:44,064 [INFO] [Thread-3] __main__: acquiring lock
10:54:44,064 [INFO] [Thread-3] __main__: acquired lock
10:54:44,064 [INFO] [Thread-3] __main__: previous value: 0
10:54:44,097 [INFO] [Thread-1] __main__: acquiring lock
10:54:45,631 [INFO] [Thread-3] __main__: Done, set new value: 1
10:54:45,631 [INFO] [Thread-3] __main__: releasing lock
10:54:45,631 [INFO] [Thread-1] __main__: acquired lock
10:54:45,631 [INFO] [Thread-1] __main__: previous value: 1
10:54:46,555 [INFO] [Thread-4] __main__: acquiring lock
10:54:46,661 [INFO] [Thread-2] __main__: acquiring lock
10:54:46,959 [INFO] [Thread-5] __main__: acquiring lock
10:54:47,142 [INFO] [Thread-1] __main__: Done, set new value: 2
10:54:47,142 [INFO] [Thread-1] __main__: releasing lock
10:54:47,142 [INFO] [Thread-4] __main__: acquired lock
10:54:47,142 [INFO] [Thread-4] __main__: previous value: 2
10:54:48,692 [INFO] [Thread-4] __main__: Done, set new value: 3
10:54:48,692 [INFO] [Thread-4] __main__: releasing lock
10:54:48,692 [INFO] [Thread-2] __main__: acquired lock
10:54:48,692 [INFO] [Thread-2] __main__: previous value: 3
10:54:50,291 [INFO] [Thread-2] __main__: Done, set new value: 4
10:54:50,292 [INFO] [Thread-2] __main__: releasing lock
10:54:50,292 [INFO] [Thread-5] __main__: acquired lock
10:54:50,292 [INFO] [Thread-5] __main__: previous value: 4
10:54:51,842 [INFO] [Thread-5] __main__: Done, set new value: 5
10:54:51,842 [INFO] [Thread-5] __main__: releasing lock
10:54:51,842 [INFO] [MainThread] __main__: Done, final value: 5
```

Returning Results from Threads

- How can a thread return a result to the main thread?
- `thread.join()` always returns None
- Option: pass immutable (list, dictionary) to thread and update in thread

Demo: 04-return_values.py

`concurrent.futures`: PoolExecutor

- High level interface for concurrent execution of tasks
- Execution based on
  - Thread
  - Process
- Core concept: “Executor”
  - ThreadPoolExecutor
  - ProcessPoolExecutor
- `submit()` – schedule execution of a task

Demo: 05-thread_pool.py

Concurrent.futures.ThreadPoolExecutor

- Number of workers determines the number of threads
  - Tasks get assigned to workers
- Can act as context manager
  - Waits for all tasks to terminate
  - Clean up threads

```
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    log.info('creating tasks')
    for i in range(THREADS):
        executor.submit(update_counter_context, results, i)
    log.info('tasks created')

log.info(f'Done, final value: {counter}')
```

Futures

- ”Future” represents an eventual result of an asynchronous operation
- `PoolExecutor.submit()` returns a future
- Calling `result()` on a future waits until the asynchronous operation is finished

Demo: 06-futures.py

`PoolExecutor.map()`: Schedule Tasks, Get Results

- Scheduling tasks and collecting results can be done in one step

Demo: 07-futures_map.py

- `map()`
  - Call some code asynchronously passing provided parameters one by one
  - Each call is scheduled as a separate task
  - Returns an iterator which returns the results of the futures in order

Demo: 08-futures_map_enumerate.py

- Note: although other tasks might already be done, iterating over `map()` always waits for the next task on order to finish

`as_completed()`: I Want My Results Now!

- concurrent.futures.as_completed()
  - Takes a list of futures
  - Iterator
  - Returns futures as they complete

Demo: 09-futures_as_completed.py

- Main thread can work on results as they become available
  - Not necessarily in the order they were scheduled
  - Tasks can return context to main thread

```
return i, val
…
i, r = completed_future.result()
results[i] = r
```

`as_completed()`: Future Map

- When using `as_completed()` futures are returned in order of completion not in scheduled order
- Typical pattern to determine context for future: mapping (dictionary)

  10-as_completed_future_map.py

- Futures are hashable
- Main thread can create a dictionary to map from future to context

Python Performance

Python Performance

- Depending on the workload Python code can be slower than other language like for example Java
- A lot of workloads are not really CPU bound
  - Other factors: network, queries, ...
  - Faster running code does not really help
- Python execution speed does not really matter when working on I/O bound problems

Practical Example: Chess Exhibition (serial)

- Assumption:
  - 24 opponents
  - Master moves in 5 seconds
  - Other players move in 55 seconds
  - Game averages at 30 move pairs
- Each game runs 30 minutes
- 24 sequential games: 12 hours

Asynchronous Python for the Complete Beginner, Miguel Grinberg, Pycon 2017: https://www.youtube.com/watch?v=iG6fr81xHKA

Practical Example: Chess Exhibition (async)

- Master moves on first game
- Then moves to 2nd, 3rd, 4th, ..
- A move on all 24 games takes the master: 24 x 5 sec = 2 min
- After two minutes the 1st game is again ready for her move
- 24 games are completed in 2 min x 30 = 1 h

Asynchronous Python for the Complete Beginner, Miguel Grinberg, Pycon 2017: https://www.youtube.com/watch?v=iG6fr81xHKA

Boosting I/O Performance Using Threads

I/O Performance

Demo: 11-field notices.py

- I/O takes ”forever”
  - web server response time
  - Network propagation
  - Data transfer
- Threading allows to issue multiple requests w/o having to wait for each request to complete
- There can be too many threads: processing the responses might take too long → server errors
  - PoolExecutor offers simple way to limit the number of threads

Threads vs Processes

Thread vs. Process

- Multi-Threading boosts performance of I/O bound tasks
- What about CPU bound tasks?

Demo: 12-thread_vs_process.py

- Threads don't offer performance benefit if tasks are CPU bound
- `concurrent.futures.ProcessPoolExecutor` to the rescue!

Threads vs. Processes

- ProcessPoolExecutor:
  - One Python process per task
  - Single thread per process
  - More memory intensive
- ThreadPoolExecutor:
  - Single Python process for all tasks
  - One thread per task

Process Name	Memory	Threads	Ports	PID	User
Python	7,4 MB	1	14	11760	jkrohn
Python	7,4 MB	3	16	11733	jkrohn
Python	7,4 MB	1	14	11764	jkrohn
Python	7,4 MB	1	14	11761	jkrohn
Python	7,4 MB	1	14	11762	jkrohn
Python	7,3 MB	1	14	11763	jkrohn
Python	5,8 MB	1	14	11759	jkrohn

Process Name	Memory	Threads	Ports	PID	User
Python	5,3 MB	6	19	11733	jkrohn

Summary

Summary

- Concurrency is easy to use in Python
- Multi-Threading perfect to speed up I/O bound tasks
- CPU bound tasks can be sped up using multiple processes
- `concurrent.futures`
  - High-level interface for concurrent execution of tasks
  - Supports threads and processes

References

- https://realpython.com/python-concurrency/
- https://docs.python.org/3/library/threading.html
- https://docs.python.org/3/library/concurrent.futures.html

CISCO