Lecture #13: Threading and Queue

Presented by Jamal Bouajjaj

2023–10–02

For University of New Haven's Fall 2023 CSCIxx51 Course

Problem

Let's say you have process that waits on something, for example a delay.

```
import time
def do_long_thing():
    print("Doing a thing!")
    time.sleep(1)
    print("Done!")

print("Other thing")
do_long_thing()
print("Oh no, I am delayed!")
```

Problem

Or what if we want multiple things done together "at the same time"

```python
import time
def do_long_thing(i):
    print(f"Doing a thing for {i}")
    time.sleep(1)
    print("Done!")

print("Other thing")
for i in range(10):
    do_long_thing(i)
print("Oh no, I am delayed!")
```

threading

Solved!

Welcome to the threading module. This runs a function in a thread, allowing async functions to run while your main application is running.

```python
import threading
import time
def do_long_thing(i):
    print(f"Doing a thing for {i}")
    time.sleep(1)
    print("Done!")

print("Other thing")
for i in range(10):
    t = threading.Thread(target=do_long_thing, args=(i,
    ↪  ))
    t.start()
print("yay, not delayed!")
```

Making one

To make a thread, we call `Thread` to make a `Thread` object. Target is the function to run, and args are the arguments passed to the function given as a tuple.

```
t = threading.Thread(target=TARGET, args=())
```

This will return a Thread object

Class Methods

The following are the main methods to a thread object:

```
t.start()      # starts the thread
t.is_alive()   # gets a bool depending if the thread is
↪   alive
t.join()       # waits until the thread function is
↪   exited
```

Only Once

A Thread object can only be ran once:

```
t.start()
t.join()
t.start()   # This will fail
```

Actual Threads

When you think of a "thread", you are thinking it's a seperate process that uses another CPU core...right?

Actual Threads

When you think of a ”thread”, you are thinking it’s a seperate process that uses another CPU core...right?

What if I told you that is NOT the case!

Actual Threads

When you think of a "thread", you are thinking it's a seperate process that uses another CPU core...right?

What if I told you that is NOT the case!

One could also say...

Actual Threads

When you think of a "thread", you are thinking it's a seperate process that uses another CPU core...right?

What if I told you that is NOT the case!

One could also say...

An imposter thread

Actual Threads

When you think of a ”thread”, you are thinking it’s a seperate process that uses another CPU core...right?

What if I told you that is NOT the case!

One could also say...

An imposter thread AMONG US! (sorry)

Why?? GIL!

GIL

This because of the Python Global Intepreter Lock (GIL).

This internal mechanism ensures that the intepreter only executes one bytecode at a time.

This means that threading is not actually multi-CPU threaded, so your program will still run on one core.

Communication

Let's say you have a thread and GUI thread. How will you ensure nice communication between the thread and GUI?

You can just have a shared variable, but that is not thread safe, and can lead to race conditions.

Locks

Lock Object

A lock object, when called, will ensure the same lock is not executed elsewhere. It will hold the other process until the lock is released.

```
tl = threading.Lock()
tl.acquire()    # Get the lock
tl.release()    # Release it back
tl.locked()     # Get if the lock is locked
with tl:  # this will acquire and release for you!
  something()
```

Queue

Lock Object

If you want to send data back and forth, one useful thread-safe way to do so is with a Queue. This is a seperate module: queue.

A Queue is a FIFO buffer that can have stuff put into it, and stuff retreived from.

```
import queue
#q = queue.Queue(maxsize=0)  # maxsize is optional, can
↪  be set to limit size
q = queue.Queue()
q.put(123)
print(q.qsize())
print(q.empty())
print(q.full())   # if Queue was given a size
print(q.get())
q.join()  # Wait until all items have been grabbed.
```

End

The end