





Select lesson text to explore the Explain with AI Premium Offering

# **Thread Safety**

This lessons discusses the concept of thread safety.

## **Thread Safety**

The primary motivation behind using multiple threads is improving program performance that may be measured with metrics such as throughput, responsiveness, latency, etc. Whenever threads are introduced in a program, the shared state amongst the threads becomes vulnerable to corruption. If a class or a program has immutable state then the class is necessarily thread-safe. Similarly, the shared state in an application where the same thread mutates the state using an operation that translates into an atomic bytecode instruction can be safely read by multiple reader threads. In contrast, a sole writer thread mutating the shared state using several atomic bytecode instructions isn't a thread-safe scenario for reader threads. Most multi-threaded setups require caution when interacting with shared state. As a corollary, the composition of two thread-safe classes doesn't guarantee thread-safety.

### **Atomicity**

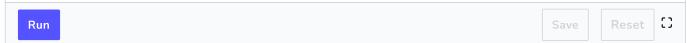
Consider the below snippet:

```
count = 0

def increment():
    global count
    count += 1
```

The above code will work flawlessly when it is executed by a single thread. However, if there are two or more threads involved, things get tricky. The key to realize is that the statement count += 1 isn't atomic. A thread can't increment the variable atomically, i.e. there doesn't exist a single bytecode instruction that can increment the count variable. Let's examine the bytecode generated for our snippet above.

```
import dis
1
3
    count = 0
4
5
    def increment():
6
        global count
7
        count += 1
8
   # prints the bytecode
9
10
    dis.dis(increment)
11
```



#### Select lesson text to explore the Explain with AI Premium Offering

Output			0.56s
7	<pre>0 LOAD_GLOBAL 3 LOAD_CONST 6 INPLACE_ADD</pre>	0 (count) 1 (1)	
	7 STORE_GLOBAL 10 LOAD_CONST 13 RETURN_VALUE	0 (count) 0 (None)	
			•

#### **Generated Byte Code**

7 0 LOAD\_GLOBAL 0 (count)

3 LOAD\_CONST 1 (1)

6 INPLACE\_ADD

7 STORE\_GLOBAL 0 (count)

10 LOAD\_CONST 0 (None)

13 RETURN\_VALUE

The seemingly single line statement expands into multiple bytecode instructions. When two threads invoke the <code>increment()</code> method it is possible that the first thread is switched out by the Python interpreter just before the third <code>INPLACE\_ADD</code> instruction is executed. Now the second thread comes along and executes all the six bytecode instructions in one go. When the first thread is rescheduled by the interpreter, it executes the third line but the value the thread holds is stale causing it to incorrectly update the <code>count</code> variable.

Programming languages provide constructs such as mutexes and locks to help developers guard sections of code that must be executed sequentially by multiple threads. Guarding shared data is one aspect of multi-threaded programs. The other aspect is coordination and cooperation amongst threads. Again, languages provide mechanisms to facilitate threads to work cooperatively towards a common goal. These include semaphores, barriers etc.

#### Thread Unsafe Class

Take a minute to go through the following program. It increments an object of class **Counter** using 5 threads. Each thread increments the object a hundred thousand times. The final value of the counter should be half a million (500,000). If you run the program enough times, you'll sometimes get the correct summation, and at others, you'll get an incorrect value.

1 from threading import Thread

2 import sys

3

4 class Counter:

5







```
def __init__(self):
    self.count = 0
```

Select lesson text to explore the Explain with AI Premium Offering

```
SCII.COUIIC I- I
12
13
    if __name__ == "__main__":
14
15
        # Sets the thread switch interval
16
        sys.setswitchinterval(0.005)
17
18
19
        numThreads = 5
        threads = [0] * numThreads
20
21
        counter = Counter()
22
23
        for i in range(0, numThreads):
24
            threads[i] = Thread(target=counter.increment)
25
        for i in range(0, numThreads):
26
27
            threads[i].start()
28
29
        for i in range(0, numThreads):
30
             threads[i].join()
                                                                                                                []
 Run
                                                                                                            X
Output
                                                                                                        0.63s
  count = 352061
```

## Fixing Thread Unsafe Class

We fix the above example using the equivalent of a mutex in Python called a Lock. For now, don't worry about how the example below works, but observe how the count always sums up to half a million.

```
1 from threading import Thread
 2
    from threading import Lock
 3
   import sys
 4
 5
   class Counter:
 6
 7
        def __init__(self):
            self.count = 0
 8
 9
            self.lock = Lock()
10
        def increment(self):
11
12
            for _ in range(100000):
13
                self.lock.acquire()
14
                self.count += 1
15
                self.lock.release()
16
17
18
   if __name__ == "__main__":
```

