How to use the Python Threading Lock to Prevent Race Conditions

Summary: in this tutorial, you'll learn about the race conditions and how to use the Python threading Lock object to prevent them.

What is a race condition

A race condition occurs when two threads (https://www.pythontutorial.net/advanced-python/python-threading/) try to access a shared variable simultaneously.

The first thread reads the value from the shared variable. The second thread also reads the value from the same shared variable.

Then both threads try to change the value of the shared variable. And they race to see which thread writes a value to the variable last.

The value from the thread that writes to the shared variable last is preserved because it overwrites the value that the previous thread wrote.

Race condition example

The following example illustrates a race condition:

```python
from threading import Thread
from time import sleep


counter = 0

def increase(by):
    global counter

    local_counter = counter
    local_counter += by

    sleep(0.1)

    counter = local_counter
    print(f'counter={counter}')


# create threads
t1 = Thread(target=increase, args=(10,))
t2 = Thread(target=increase, args=(20,))

# start the threads
t1.start()
t2.start()


# wait for the threads to complete
t1.join()
t2.join()
```

```
print(f'The final counter is {counter}')
```

In this program, both threads try to modify the value of the `counter` variable at the same time. The value of the `counter` variable depends on which thread completes last.

If the thread `t1` completes before the thread `t2`, you'll see the following output:

```
counter=10
counter=20
The counter is 20
```

Otherwise, you'll see the following output:

```
counter=20
counter=10
The final counter is 10
```

How it works.

First, import `Thread` class from the `threading` module and the `sleep()` function from the `time` module:

```
from threading import Thread
from time import sleep
```

Second, define a global variable called `counter` whose value is zero:

```
counter = 0
```

Third, define a function that increases the value of the `counter` variable by a number:

```
def increase(by):
    global counter

    local_counter = counter
    local_counter += by

    sleep(0.1)

    counter = local_counter
    print(f'counter={counter}')
```

Fourth, create two threads. The first thread increases the `counter` by 10 while the second thread increases the `counter` by 20:

```
t1 = Thread(target=increase, args=(10,))
t2 = Thread(target=increase, args=(20,))
```

Fifth, start the threads:

```
t1.start()
t2.start()
```

Sixth, from the main thread, wait for the threads t1 and t2 to complete:

```
t1.join()
t2.join()
```

Finally, show the final value of the `counter` variable:

```
print(f'The final counter is {counter}')
```

Using Lock to prevent the race condition

To prevent race conditions, you can use the `Lock` class from the `threading` (https://www.pythontutorial.net/advanced-python/python-threading/) module. A lock has two states: locked and unlocked.

First, create an instance the `Lock` class:

```
lock = Lock()
```

By default, the lock has the unlocked status until you acquire it.

Second, acquire a lock by calling the `acquire()` method:

```
lock.acquire()
```

Third, release the lock once the thread completes changing the shared variable:

```
lock.release()
```

The following example shows how to use the `Lock` object to prevent the race condition in the previous program:

```
from threading import Thread, Lock
from time import sleep


counter = 0


def increase(by, lock):
    global counter

    lock.acquire()
```

```
        local_counter = counter
        local_counter += by

        sleep(0.1)

        counter = local_counter
        print(f'counter={counter}')

        lock.release()


    lock = Lock()

    # create threads
    t1 = Thread(target=increase, args=(10, lock))
    t2 = Thread(target=increase, args=(20, lock))

    # start the threads
    t1.start()
    t2.start()



    # wait for the threads to complete
    t1.join()
    t2.join()



    print(f'The final counter is {counter}')
```

Output:

```
counter=10
counter=30
The final counter is 30
```

How it works.

- First, add a second parameter to the `increase()` function.
- Second, create an instance of the `Lock` class.
- Third, acquire a lock before accessing the `counter` variable and release it after updating the new value.

The following illustrates how to define a `Counter` class that uses the `Lock` object:

```
from threading import Thread, Lock
from time import sleep


class Counter:
    def __init__(self):
        self.value = 0
        self.lock = Lock()

    def increase(self, by):
        self.lock.acquire()

        current_value = self.value
        current_value += by

        sleep(0.1)

        self.value = current_value
        print(f'counter={self.value}')
        self.lock.release()


counter = Counter()

# create threads
t1 = Thread(target=counter.increase, args=(10, ))
t2 = Thread(target=counter.increase, args=(20, ))

# start the threads
t1.start()
t2.start()


# wait for the threads to complete
t1.join()
t2.join()


print(f'The final counter is {counter.value}')
```

Summary

- A race condition occurs when two threads access a shared variable at the same time.
- Use a `Lock` object to prevent the race condition
- Call the `acquire()` method of a lock object to acquire a lock.
- Call the `release()` method of a lock object to release the previously acquired lock.