

## Semaphores in Python:

- The Semaphore class of the Python threading module implements the concept of semaphore.
- It has a constructor and two methods `acquire()` and `release()`.
- The `acquire()` method decreases the semaphore count if the count is greater than zero. Else it blocks till the count is greater than zero.
- The `release()` method increases the semaphore count and wakes up one of the threads waiting on the semaphore.

`Class semaphore([value])`

- The optional argument gives the initial value for the internal counter; it defaults to `1`.

### **acquire()** (*blocking*)

Acquire a semaphore.

When invoked without arguments: if the internal counter is larger than zero on entry, decrement it by one and return immediately. If it is zero on entry, block, waiting until some other thread has called `release()` to make it larger than zero. This is done with proper interlocking so that if multiple `acquire()` calls are blocked, `release()` will wake exactly one of them up. The implementation may pick one at random, so the order in which blocked threads are awakened should not be relied on. There is no return value in this case.

When invoked with *blocking* set to true, do the same thing as when called without arguments, and return true.

When invoked with *blocking* set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

### **release()**

Release a semaphore, incrementing the internal counter by one. When it was zero on entry and another thread is waiting for it to become larger than zero again, wake up that thread.

```
import threading
from time import sleep
sem = threading.Semaphore()

def fun1():
    print("fun1 starting")
    sem.acquire()
    for loop in range(1,5):
        print("Fun1 Working in loop")
        sleep(1)
    sem.release()
    print("fun1 finished")
```

```
def fun2():
```

```

    print("fun2 starting")
    while not sem.acquire(blocking=False):
        print("Fun2 No Semaphore available")
        sleep(1)
    else:
        print("Got Semaphore")
        for loop in range(1, 5):
            print("Fun2 Working loop too")
            sleep(1)
    sem.release()

t1 = threading.Thread(target = fun1)
t2 = threading.Thread(target = fun2)
t1.start()
t2.start()
t1.join()
t2.join()
print("All Threads done Exiting")

```

### **Difference between Semaphore, BoundedSemaphore**

```

from threading import Semaphore, BoundedSemaphore

```

```

# Usually, you create a Semaphore that will allow a certain number of threads
# into a section of code. This one starts at 5.

```

```

s1 = Semaphore(5)

```

```

# When you want to enter the section of code, you acquire it first.
# That lowers it to 4. (Four more threads could enter this section.)

```

```

s1.acquire()

```

```

# Then you do whatever sensitive thing needed to be restricted to five threads.

```

```

# When you're finished, you release the semaphore, and it goes back to 5.

```

```

s1.release()

```

```

# That's all fine, but you can also release it without acquiring it first.

```

```

s1.release()

```

```

# The counter is now 6! That might make sense in some situations, but not in most.

```

```

print(s1._value) # => 6

```

```

# If that doesn't make sense in your situation, use a BoundedSemaphore.

```

```

s2 = BoundedSemaphore(5) # Start at 5.

```

```

s2.acquire() # Lower to 4.

```

```

s2.release() # Go back to 5.

```

```

print(s2._value)

```

```

try:

```

```

    s2.release() # Try to raise to 6, above starting value.

```

```

    print(s2._value)

```

```
except ValueError:  
    print('As expected, it complained.')
```