

# PYTHON THREADING 2

## GOAL

The purpose of this lab is to demonstrate the use of locks with threads in Python. The following subjects will be covered:

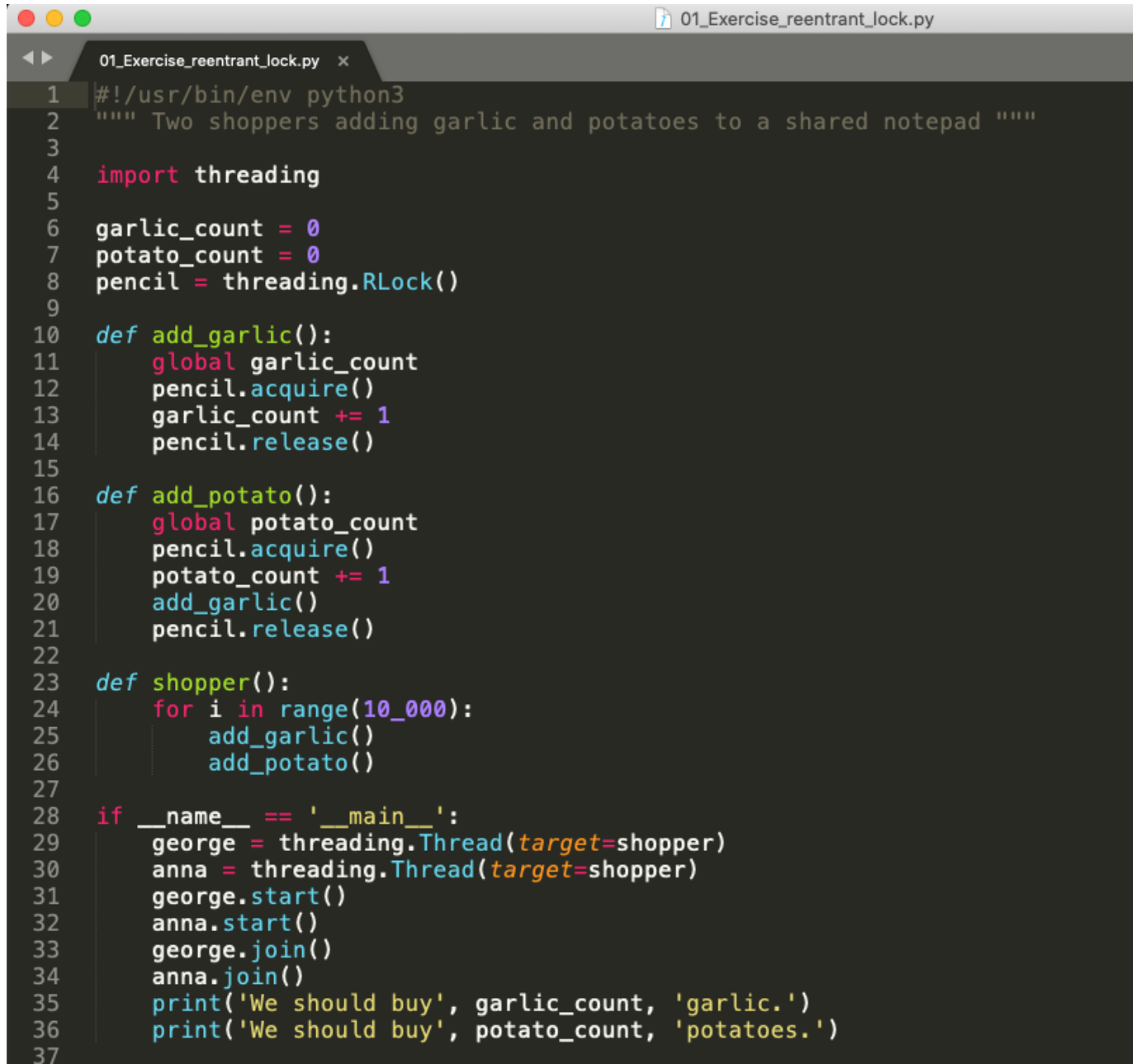
- **Re-entrant lock in Python (Rlock)**
- **Try Lock: a non-blocking version of the acquire method**
- **Read-Write Lock**
- **Deadlock**
- **Abandoned Lock**

Proceed through the pages of this document until you have gone through all the exercises. Run the python scripts using Python 3.

## EXERCISE 01: RE-ENTRANT LOCK

[Filename: **01\_Exercise\_reentrant\_lock.py**]

To demonstrate how to use a reentrant lock in Python, we follow the example used to demonstrate a data race and mutual exclusion with two shoppers that are concurrently incrementing the amount of items to buy.



```
1 #!/usr/bin/env python3
2 """ Two shoppers adding garlic and potatoes to a shared notepad """
3
4 import threading
5
6 garlic_count = 0
7 potato_count = 0
8 pencil = threading.RLock()
9
10 def add_garlic():
11     global garlic_count
12     pencil.acquire()
13     garlic_count += 1
14     pencil.release()
15
16 def add_potato():
17     global potato_count
18     pencil.acquire()
19     potato_count += 1
20     add_garlic()
21     pencil.release()
22
23 def shopper():
24     for i in range(10_000):
25         add_garlic()
26         add_potato()
27
28 if __name__ == '__main__':
29     george = threading.Thread(target=shopper)
30     anna = threading.Thread(target=shopper)
31     george.start()
32     anna.start()
33     george.join()
34     anna.join()
35     print('We should buy', garlic_count, 'garlic.')
36     print('We should buy', potato_count, 'potatoes.')
37
```

- On line 8, note that we use the constructor method with RLock, which is Python's implementation of a reentrant lock that can be acquired multiple times before being released.
- One interesting difference between the regular lock and Rlock in Python is that the regular lock can be released by different threads than the one that acquired it.
- The reentrant lock must be released by the same thread that acquired it. And of course, it must be released by that thread as many times as it was acquired before it will be available for another thread to take.

## EXERCISE 02: TRY LOCK

[Filename: **02\_Exercise\_nonblocking\_acquire.py**]

Try Lock, or Try Enter, is a non-blocking version of the lock or acquire method. It returns immediately and one of two things will happen:

1. **If the mutex you're trying to lock is available**, it will get locked and the method will return **TRUE**.
2. **If the mutex is already possessed by another thread**, the Try Lock method will immediately return **FALSE**.

The returned value of true or false lets the thread know whether or not it was successful in acquiring the lock.

The order of the statements on line 15 on each side of the and operator are evaluated from left to right. Only if the left side of the and is true, will it evaluate the right side, and execute the non-blocking acquire method.

The non-blocking acquire method is configured by setting the optional blocking parameter to be false: *pencil.acquire(blocking=False)*

**If the lock is available**, then calling acquire will lock it, and return true. The program will then execute the code between line 16 to 20 to add items to the shared notepad.

**If the lock is not available**, then the non-blocking acquire method will immediately return false, and that thread will execute the else clause from lines 22 to 24.

Using the non-blocking acquire method, a program is executed much faster.

```

02_Exercise_nonblocking_acquire.py
1  #!/usr/bin/env python3
2  """ Two shoppers adding items to a shared notepad """
3
4  import threading
5  import time
6
7  items_on_notepad = 0
8  pencil = threading.Lock()
9
10 def shopper():
11     global items_on_notepad
12     name = threading.current_thread().getName()
13     items_to_add = 0
14     while items_on_notepad <= 20:
15         if items_to_add and pencil.acquire(blocking=False): # add item(s)
16             items_on_notepad += items_to_add
17             print(name, 'added', items_to_add, 'item(s) to notepad.')
18             items_to_add = 0
19             time.sleep(0.3) # time spent writing
20             pencil.release()
21         else: # look for other things to buy
22             time.sleep(0.1) # time spent searching
23             items_to_add += 1
24             print(name, 'found something else to buy.')
25
26 if __name__ == '__main__':
27     george = threading.Thread(target=shopper, name='george')
28     anna = threading.Thread(target=shopper, name='anna')
29     start_time = time.perf_counter()
30     george.start()
31     anna.start()
32     george.join()
33     anna.join()
34     elapsed_time = time.perf_counter() - start_time
35     print('Elapsed Time: {:.2f} seconds'.format(elapsed_time))
36

```

## EXERCISE 03: READ-WRITE LOCK

[Filename: **03\_Exercise\_readwrite\_lock.py**]

A reader-writer lock, or shared mutex, can be locked in one of two ways.

1. It can be locked in a **shared read mode** that allows multiple threads that only need to read simultaneously to lock it.
  2. It can be locked in an **exclusive write mode** that limits access to only one thread at a time, allowing that thread to safely write to the shared resource.
- A thread trying to acquire the lock in write mode can't do so as long as it's still being held by any other threads in the read mode.
  - Since only one thread can have the write lock at a time, all other threads wanting to read or write will have to wait until the lock becomes available again.
  - Recognizing when to use a read-write lock is just as important as knowing how to use it. In certain scenarios, read-write locks can improve a program's performance versus using a standard mutex but they are more complicated to implement, and they typically use more resources to keep track of the number of readers.
  - As general rule of thumb: it makes sense to use a shared reader-writer lock **when you have a lot more threads that will be reading from the shared data than the number of threads that will be writing to it**, such as certain types of database applications.
  - If the majority of your threads are writing, then there's not much, if any, advantage to using a read-write lock.

Reader-writer locks are a common feature in many programming languages that support concurrency, however **they're not included by default in Python**.

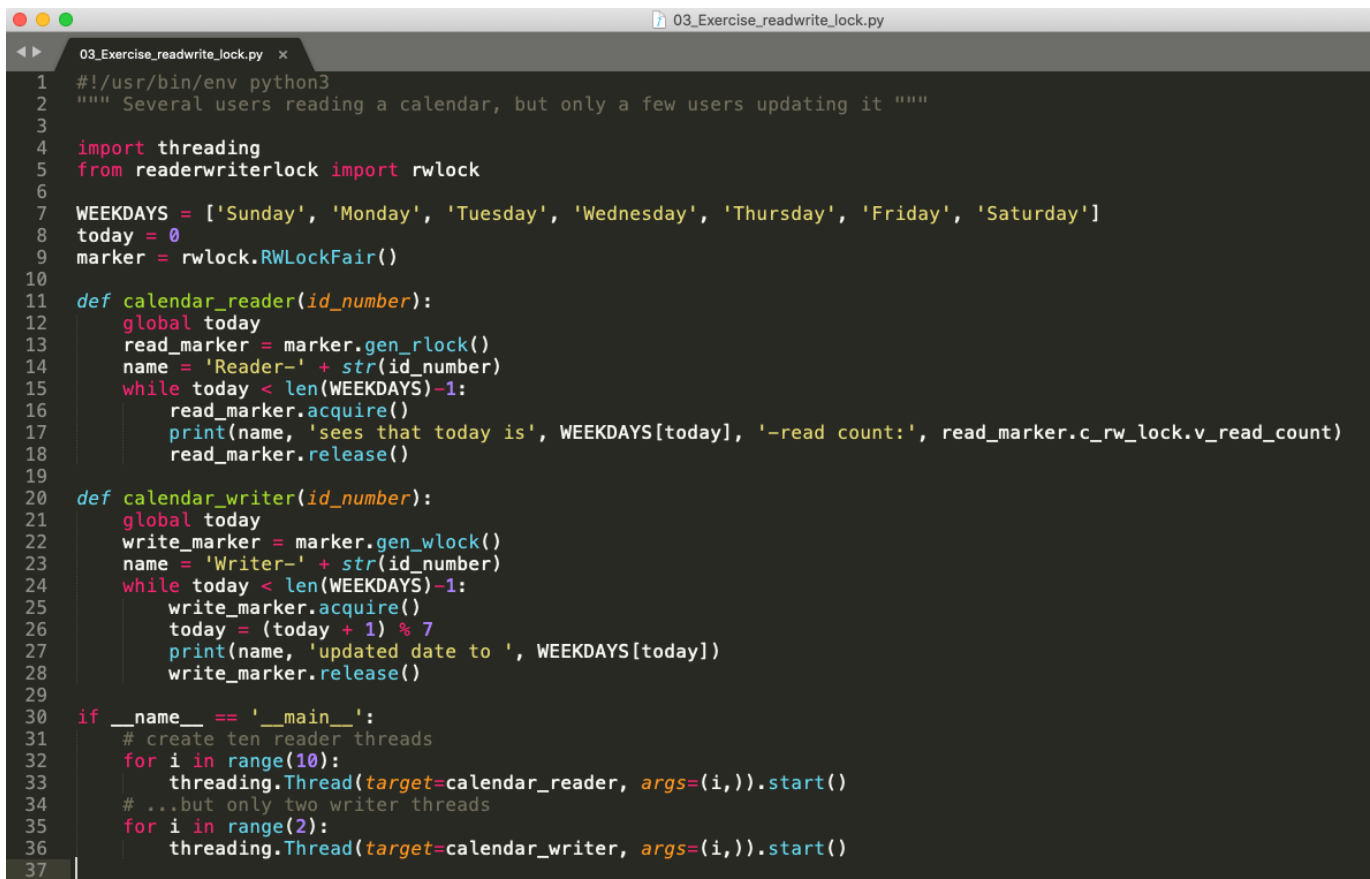
Before running the following script, make sure you have installed the readerwriterlock by following the link: <https://pypi.org/project/readerwriterlock/>

### RWLock Variations:

- RWLockFair – fair priority for readers/writers
- RWLockRead – readers get priority
- RWLockWrite – writers get priority

## RWLock Methods:

- `gen_rlock()` – generates a reader lock object, which can be held by multiple threads at once
- `gen_wlock()` – generates a writer lock object, which can only be held by a single thread at once



```

1  #!/usr/bin/env python3
2  """ Several users reading a calendar, but only a few users updating it """
3
4  import threading
5  from readerwriterlock import rwlock
6
7  WEEKDAYS = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
8  today = 0
9  marker = rwlock.RWLockFair()
10
11 def calendar_reader(id_number):
12     global today
13     read_marker = marker.gen_rlock()
14     name = 'Reader-' + str(id_number)
15     while today < len(WEEKDAYS)-1:
16         read_marker.acquire()
17         print(name, 'sees that today is', WEEKDAYS[today], '-read count:', read_marker.c_rw_lock.v_read_count)
18         read_marker.release()
19
20 def calendar_writer(id_number):
21     global today
22     write_marker = marker.gen_wlock()
23     name = 'Writer-' + str(id_number)
24     while today < len(WEEKDAYS)-1:
25         write_marker.acquire()
26         today = (today + 1) % 7
27         print(name, 'updated date to ', WEEKDAYS[today])
28         write_marker.release()
29
30 if __name__ == '__main__':
31     # create ten reader threads
32     for i in range(10):
33         threading.Thread(target=calendar_reader, args=(i,)).start()
34     # ...but only two writer threads
35     for i in range(2):
36         threading.Thread(target=calendar_writer, args=(i,)).start()
37

```

Line 17: `read_marker.c_rw_lock.v_read_count` is used to display the number of threads that currently hold the read marker.

## EXERCISE 04: DEADLOCK

[Filename: **04\_Exercise\_deadlock.py**]

**Deadlock:** Each thread of a group of threads is waiting for another thread to take action

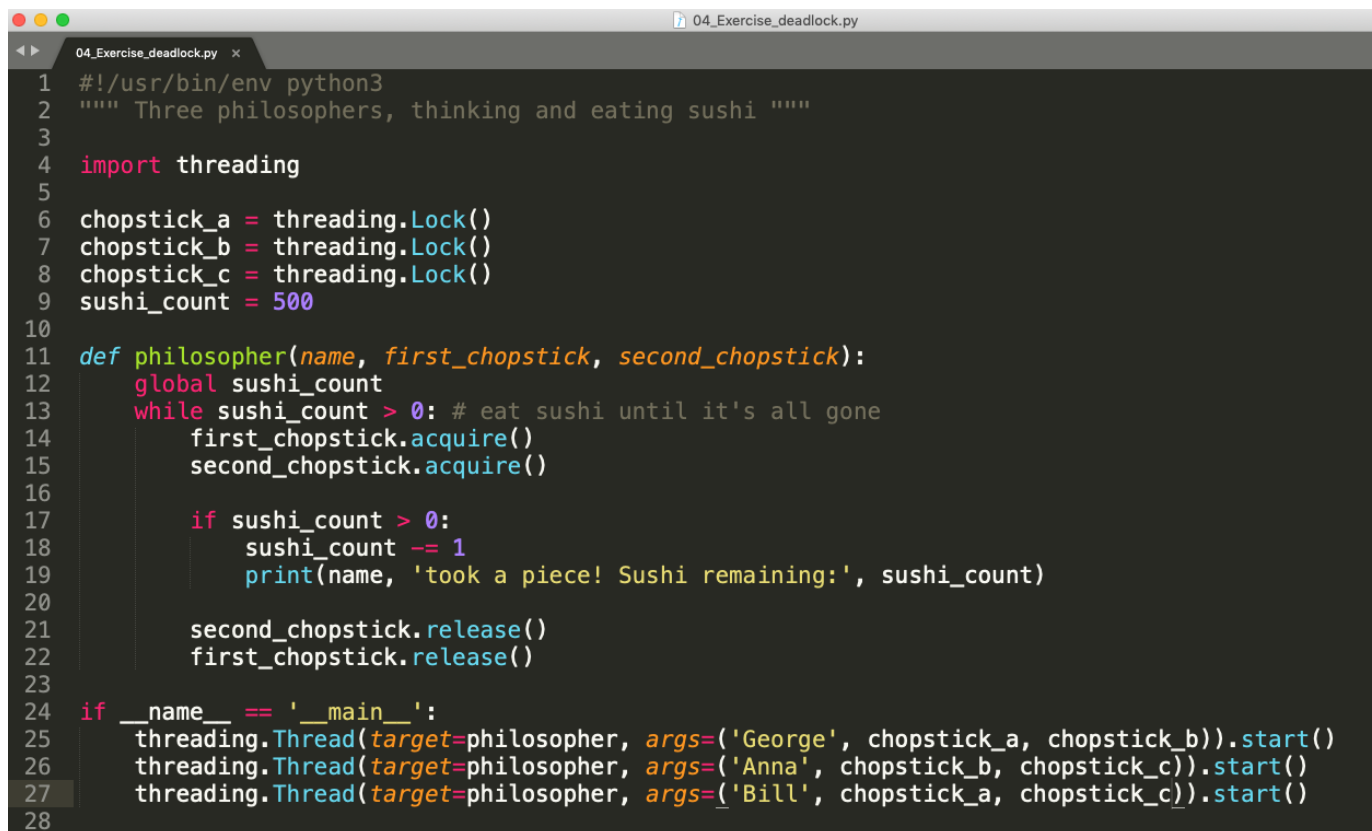
**Liveness:** We want our programs to be free from deadlock to guarantee liveness. So we need:

- Properties that require a system to make progress
- Threads may have to “take turns” in critical section

In this program, each thread is competing for two of the three chopsticks placed around the table, labeled here as A, B and C. In our example Python program, we instantiate those three lock objects on lines 6 through 8 and name them chopstick A, B, and C.

We also create a variable named `sushi_count` to represent the amount of sushi left between the philosophers.

The dining philosophers' problem can be solved by giving priorities. In this case, chopstick A has the highest priority, B is second and C is third. Each philosopher should always acquire their highest priority chopstick first. We can see on line 25 that George acquires A before B, Anna acquires B before C and Bill acquires A first and then C.



```
1  #!/usr/bin/env python3
2  """ Three philosophers, thinking and eating sushi """
3
4  import threading
5
6  chopstick_a = threading.Lock()
7  chopstick_b = threading.Lock()
8  chopstick_c = threading.Lock()
9  sushi_count = 500
10
11 def philosopher(name, first_chopstick, second_chopstick):
12     global sushi_count
13     while sushi_count > 0: # eat sushi until it's all gone
14         first_chopstick.acquire()
15         second_chopstick.acquire()
16
17         if sushi_count > 0:
18             sushi_count -= 1
19             print(name, 'took a piece! Sushi remaining:', sushi_count)
20
21         second_chopstick.release()
22         first_chopstick.release()
23
24 if __name__ == '__main__':
25     threading.Thread(target=philosopher, args=('George', chopstick_a, chopstick_b)).start()
26     threading.Thread(target=philosopher, args=('Anna', chopstick_b, chopstick_c)).start()
27     threading.Thread(target=philosopher, args=('Bill', chopstick_a, chopstick_c)).start()
28
```

Try running the program replacing the lines 25 to 27 with the following to see how the deadlock appears:

```
threading.Thread(target=philosopher, args=('George', chopstick_a, chopstick_b)).start()
threading.Thread(target=philosopher, args=('Anna', chopstick_b, chopstick_c)).start()
threading.Thread(target=philosopher, args=('Bill', chopstick_c, chopstick_a)).start()
```

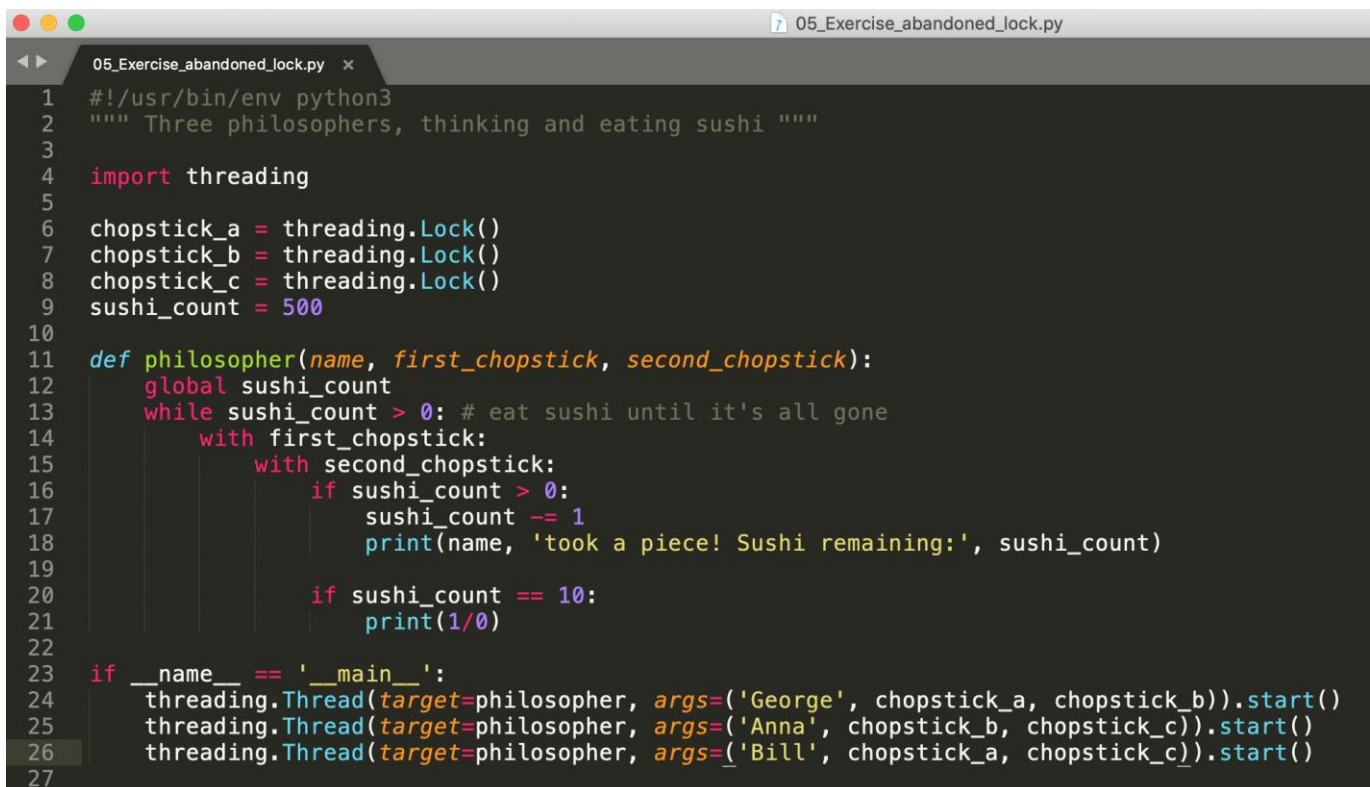
## EXERCISE 05: ABANDONED LOCK

[Filename: **05\_Exercise\_abandoned\_lock.py**]

If one thread or process acquires a lock and then terminates because of some unexpected reason it may not automatically release the lock before it disappears. That leaves other tasks stuck waiting for a lock that will never be released.

In the following program we intentionally create a crash condition by dividing by zero but we protect other threads by using context managers. We replace the acquire method on line 14 to say “with first\_chopstick:”, then nested within that goes another call “with second\_chopstick:”

When we run this program, the thread that finds `sushi_count == 10` will crash because it will run a division by zero. As locks work with context managers, the exception will not crash the whole program.



```
05_Exercise_abandoned_lock.py
1  #!/usr/bin/env python3
2  """ Three philosophers, thinking and eating sushi """
3
4  import threading
5
6  chopstick_a = threading.Lock()
7  chopstick_b = threading.Lock()
8  chopstick_c = threading.Lock()
9  sushi_count = 500
10
11 def philosopher(name, first_chopstick, second_chopstick):
12     global sushi_count
13     while sushi_count > 0: # eat sushi until it's all gone
14         with first_chopstick:
15             with second_chopstick:
16                 if sushi_count > 0:
17                     sushi_count -= 1
18                     print(name, 'took a piece! Sushi remaining:', sushi_count)
19
20                 if sushi_count == 10:
21                     print(1/0)
22
23 if __name__ == '__main__':
24     threading.Thread(target=philosopher, args=('George', chopstick_a, chopstick_b)).start()
25     threading.Thread(target=philosopher, args=('Anna', chopstick_b, chopstick_c)).start()
26     threading.Thread(target=philosopher, args=('Bill', chopstick_a, chopstick_c)).start()
27
```