Codio Activity: The Producer-Consumer Mechanism (Unit 5)

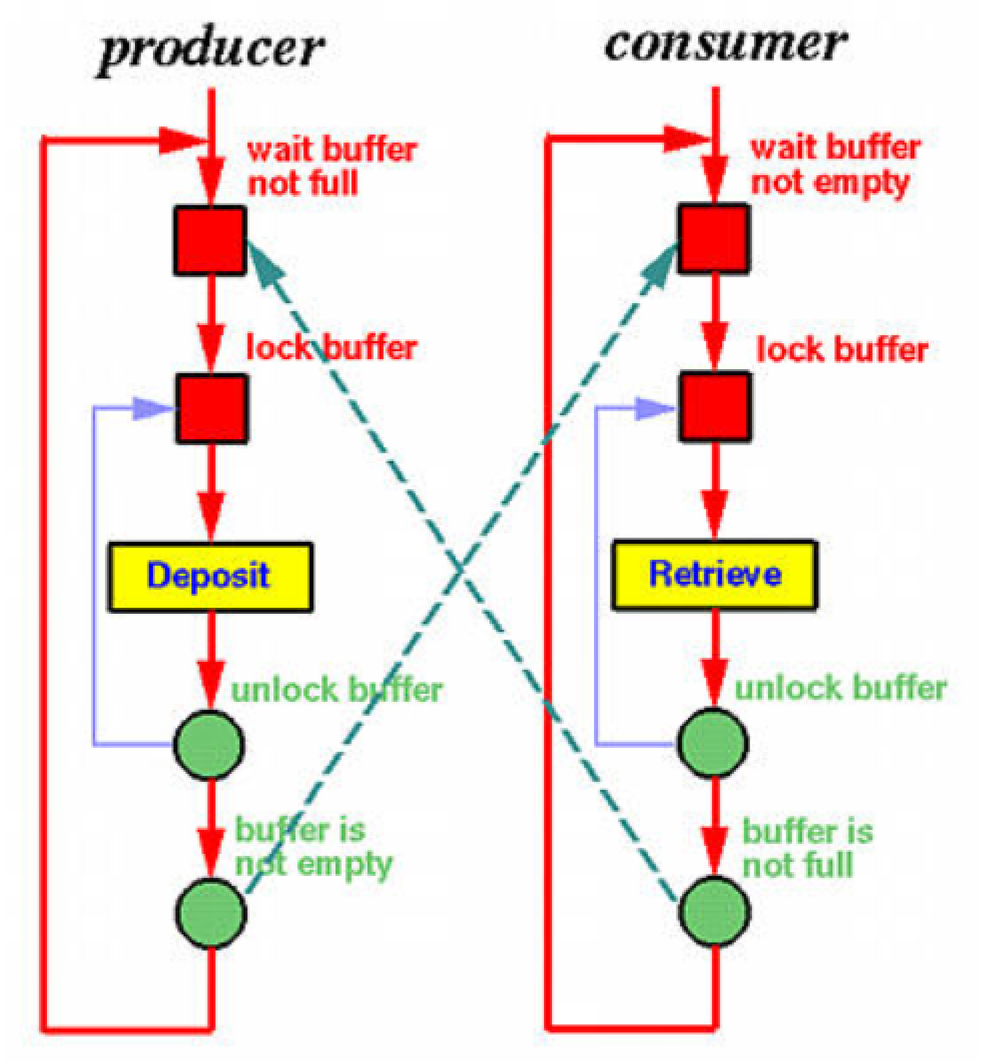
Producer/Consumer Problem (also known as the ‘bounded buffer’ problem):

- A ‘producer’ is producing items at a particular (unknown and sometimes unpredictable) rate.
- A ‘consumer’ is consuming the items – again, at some rate.

For example, a producer-consumer scenario models an application producing a listing that must be consumed by a printer process, as well as a keyboard handler producing a line of data that will be consumed by an application program. This is shown in the picture below (Shene, 2014).

Items are placed in a buffer when produced, so:

- Consumer should wait if there isn’t an item to consume
- Producer shouldn’t ‘overwrite’ an item in the buffer



A picture showing a diagram of the Producer-Consumer Mechanism.

Synchronisation is necessary because:

- If the consumer has not taken out the current value in the buffer, then the producer should not replace it with another.
- Similarly, the consumer should not consume the same value twice.

Task

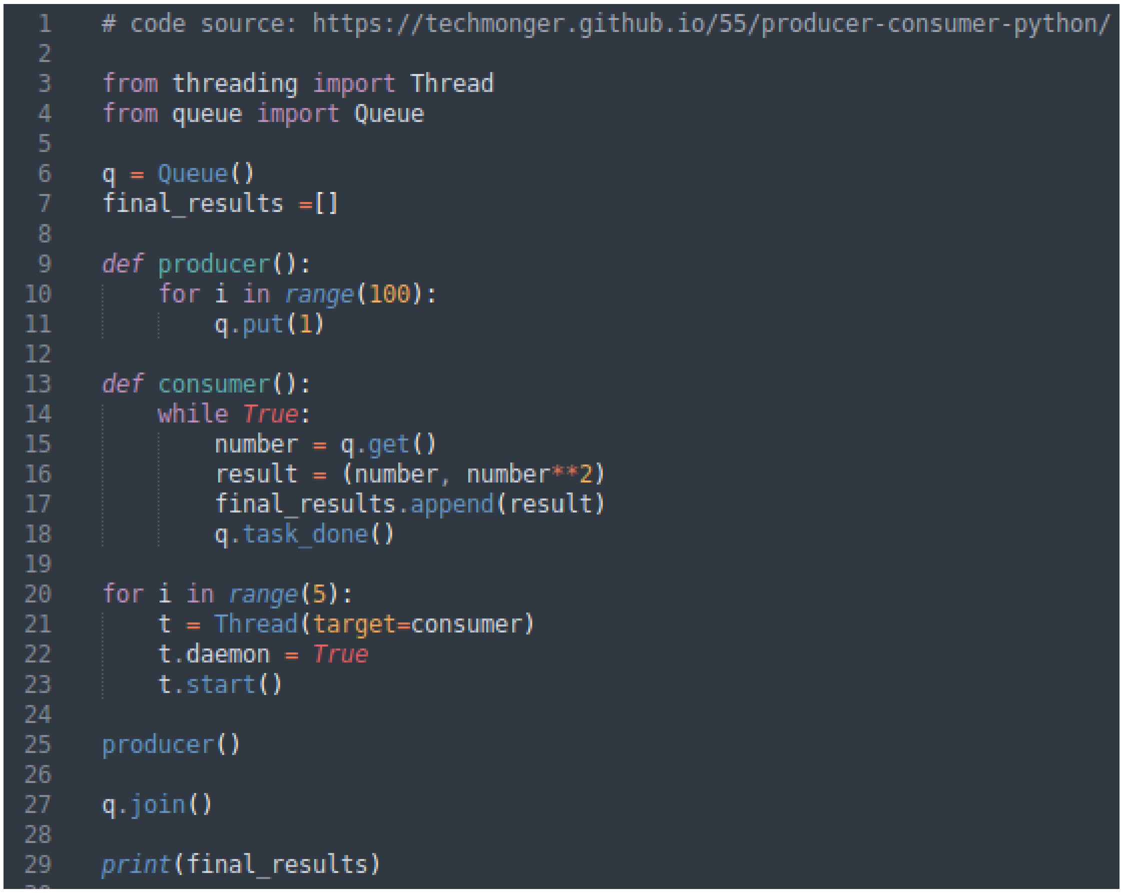
Run producer-consumer.py in the provided Codio workspace (Producer-Consumer Mechanism), where the queue data structure is used.

A copy of the code is available here for you.

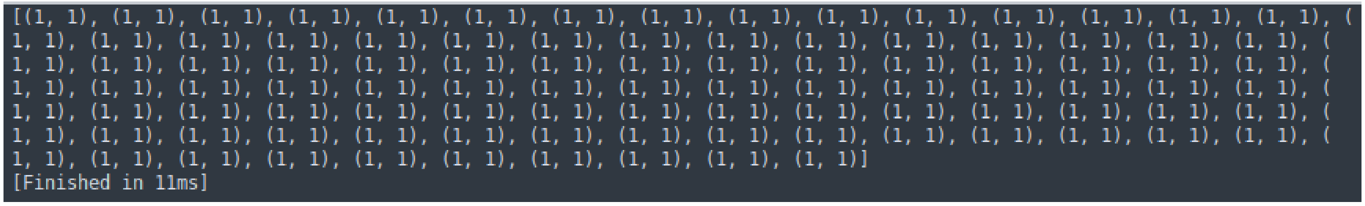
```
# code source: https://techmonger.github.io/55/producer-consumer-python/
```

```
from threading import Thread
from queue import Queue

q = Queue()
final_results = []

def producer():
    for i in range(100):
        q.put(i)


def consumer():
    while True:
        number = q.get()
        result = (number, number**2)
        final_results.append(result)
        q.task_done()


for i in range(5):
    t = Thread(target=consumer)
    t.daemon = True
    t.start()

producer()

q.join()

print (final_results)
```

Code Output

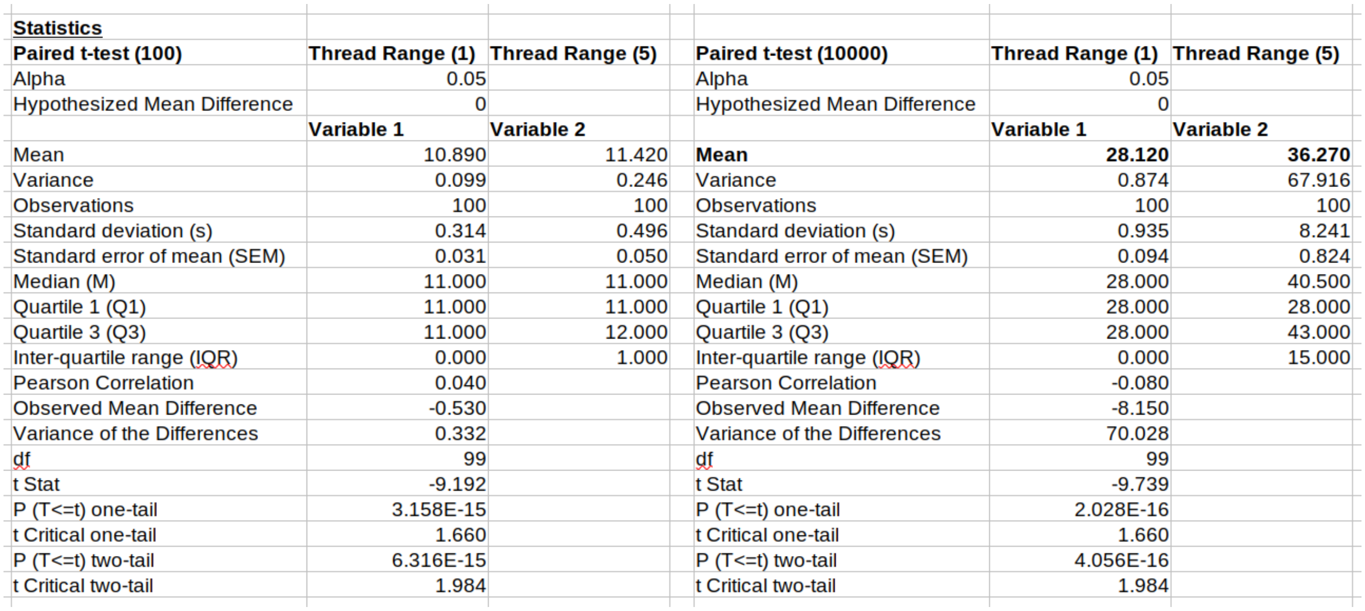


```python
# code source: https://techmonger.github.io/55/producer-consumer-python/

from threading import Thread
from queue import Queue

q = Queue()
final_results =[]

def producer():
    for i in range(100):
        q.put(1)

def consumer():
    while True:
        number = q.get()
        result = (number, number**2)
        final_results.append(result)
        q.task_done()

for i in range(5):
    t = Thread(target=consumer)
    t.daemon = True
    t.start()

producer()

q.join()

print(final_results)
```



```
[(1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (
1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (
1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (
1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (
1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (
1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (
1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1)]
[Finished in 11ms]
```

It should be noted that, upon further investigation (Berenson et al., 2015), the possible increased processing speed derived from a thread range of 5 was found to have a slight disadvantage when compared to a thread range of 1 within an *i* range of 100. Within an *i* range of 10000, a thread range of 5 was found to have a significant disadvantage in terms of processing time when compared to a thread range of 1. While this may be useful for a light program such as this, a scaled program may need additional parameters to not significantly slow the processing time.

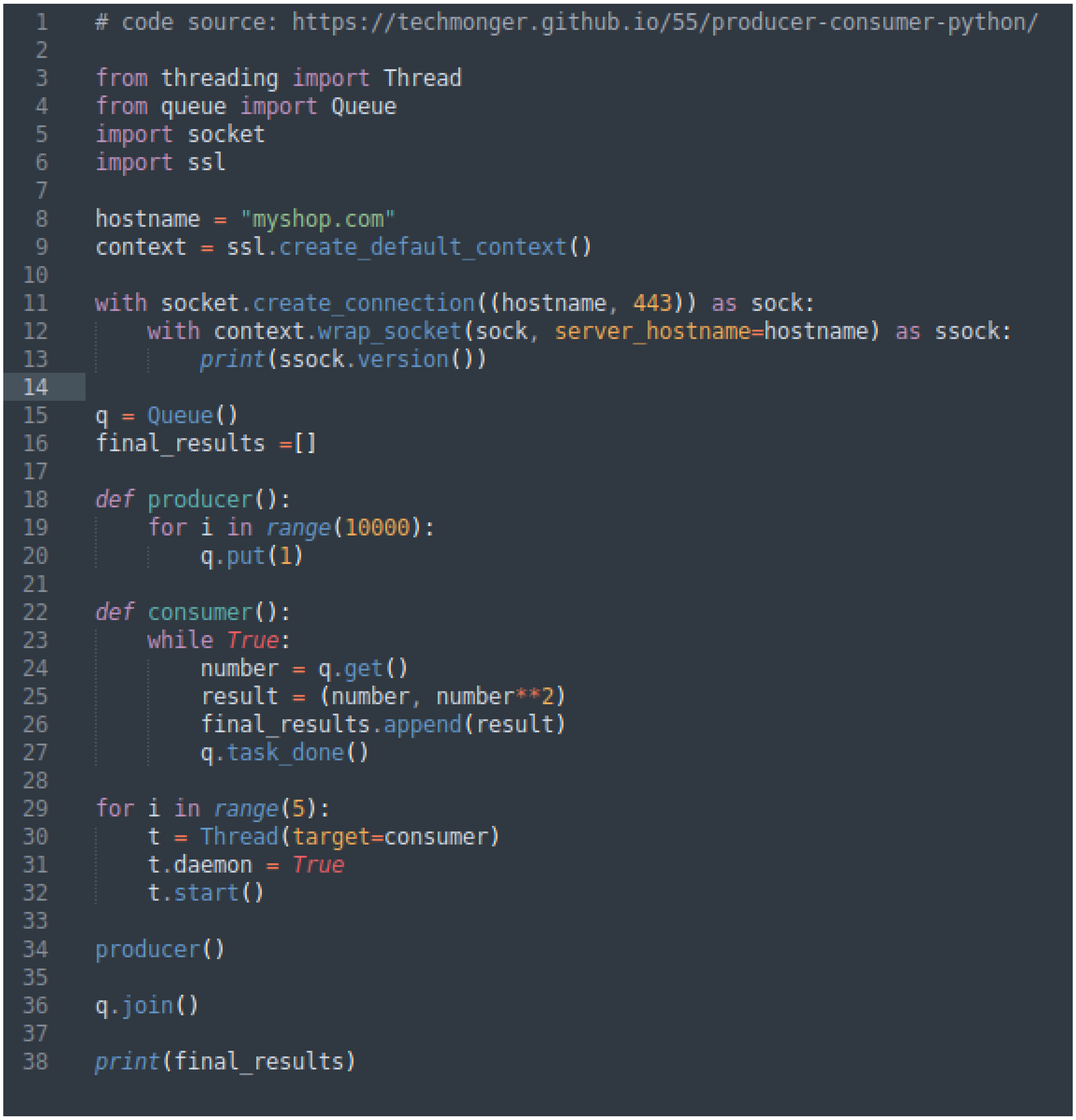


Statistics		
Paired t-test (100)	**Thread Range (1)**	**Thread Range (5)**
Alpha	0.05	
Hypothesized Mean Difference	0	
	Variable 1	**Variable 2**
Mean	10.890	11.420
Variance	0.099	0.246
Observations	100	100
Standard deviation (s)	0.314	0.496
Standard error of mean (SEM)	0.031	0.050
Median (M)	11.000	11.000
Quartile 1 (Q1)	11.000	11.000
Quartile 3 (Q3)	11.000	12.000
Inter-quartile range (IQR)	0.000	1.000
Pearson Correlation	0.040	
Observed Mean Difference	-0.530	
Variance of the Differences	0.332	
df	99	
t Stat	-9.192	
P (T<=t) one-tail	3.158E-15	
t Critical one-tail	1.660	
P (T<=t) two-tail	6.316E-15	
t Critical two-tail	1.984	

Paired t-test (10000)	**Thread Range (1)**	**Thread Range (5)**
Alpha	0.05	
Hypothesized Mean Difference	0	
	Variable 1	**Variable 2**
Mean	**28.120**	**36.270**
Variance	0.874	67.916
Observations	100	100
Standard deviation (s)	0.935	8.241
Standard error of mean (SEM)	0.094	0.824
Median (M)	28.000	40.500
Quartile 1 (Q1)	28.000	28.000
Quartile 3 (Q3)	28.000	43.000
Inter-quartile range (IQR)	0.000	15.000
Pearson Correlation	-0.080	
Observed Mean Difference	-8.150	
Variance of the Differences	70.028	
df	99	
t Stat	-9.739	
P (T<=t) one-tail	2.028E-16	
t Critical one-tail	1.660	
P (T<=t) two-tail	4.056E-16	
t Critical two-tail	1.984	

Answer the following questions:

1. How is the queue data structure used to achieve the purpose of the code?
   Queue follows the FICO rule: First In First Out. This rule means oldest item is removed first in a queue lineup. It is used to implement first in, first served (GeeksforGeeks, n.d)
2. What is the purpose of q.put(x)?
   Puts an item into the queue. If the queue is full, the next item must wait to be added. (GeeksforGeeks, n.d)
3. What is achieved by q.get()?
   Removes and returns an item from the queue. *Timeout* and *block* arguments can rate limit the speed of the command (Python, 2023a)
4. What functionality is provided by q.join()?
   Blocks until all items have been added and processed (Python, 2023a)
5. Extend this producer-consumer code to make the producer-consumer scenario available in a secure way. What technique(s) would be appropriate to apply?



```python
# code source: https://techmonger.github.io/55/producer-consumer-python/

from threading import Thread
from queue import Queue
import socket
import ssl

hostname = "myshop.com"
context = ssl.create_default_context()

with socket.create_connection((hostname, 443)) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())

q = Queue()
final_results =[]

def producer():
    for i in range(10000):
        q.put(1)

def consumer():
    while True:
        number = q.get()
        result = (number, number**2)
        final_results.append(result)
        q.task_done()

for i in range(5):
    t = Thread(target=consumer)
    t.daemon = True
    t.start()

producer()

q.join()

print(final_results)
```

Added: SSL wrapper for prep of setting up socket connection for client and producer (Python, 2023b).

References

GeeksforGeeks (n.d.) *Queue in Python*. geeksforgeeks.org [Available Online] https://www.geeksforgeeks.org/queue-in-python/

Python (2023a) *queue - A synchronized queue class* | Python. python.org [Available Online] https://docs.python.org/3/library/queue.html

Python (2023b) *ssl - TSL/SSL wrapper for socket objects* | Python. python.org [Available Online] https://docs.python.org/3/library/ssl.html#ssl-security