* [Skip to main content](https://superfastpython.com/thread-producer-consumer-pattern-in-python/" \l "genesis-content)
* [Skip to primary sidebar](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#genesis-sidebar-primary)
* [Skip to footer](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#footer)

**Additional menu**

[](https://superfastpython.com/)

**Super Fast Python**

making you awesome at concurrency

* [About](https://superfastpython.com/about/)
  + [About](https://superfastpython.com/about/)

* + [Contact](https://superfastpython.com/contact/)

* [Start Here](https://superfastpython.com/learning-paths/)

* [Topics](https://superfastpython.com/tutorials/)
  + [Threads](https://superfastpython.com/category/threading/)

* + [Multiprocessing](https://superfastpython.com/category/multiprocessing/)

* + [Asyncio](https://superfastpython.com/category/asyncio/)

* + [More…](https://superfastpython.com/python-concurrent-topics/)

* [Guides](https://superfastpython.com/python-concurrency-guides/)
  + [Threading Guide](https://superfastpython.com/threading-in-python/)

* + [Multiprocessing Guide](https://superfastpython.com/multiprocessing-in-python/)

* + [Asyncio Guide](https://superfastpython.com/python-asyncio/)

* + [More…](https://superfastpython.com/python-concurrency-guides/)

* [Products](https://superfastpython.com/products/)

* [**Free Book**](https://superfastpython.com/plip-menu)

**Thread Producer-Consumer Pattern in Python**

*April 15, 2022 by*[**Jason Brownlee**](https://superfastpython.com/about)*in*[**Python Threading**](https://superfastpython.com/category/threading/)

Last Updated on September 12, 2022

You can code a **producer-consumer pattern** using **[threading.Thread](https://superfastpython.com/threading-in-python/)** and a **queue.Queue**.

In this tutorial you will discover how to use the **producer-consumer pattern with threads in Python**.

Let’s get started.

[Download Now: Free Threading PDF Cheat Sheet](https://marvelous-writer-6152.ck.page/088fc51f28)

Table of Contents

* [What is the Producer-Consumer Pattern](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#What_is_the_Producer-Consumer_Pattern)
  + [Producer Tasks](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#Producer_Tasks)
  + [Consumer Tasks](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#Consumer_Tasks)
  + [Shared Buffer](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#Shared_Buffer)
  + [Pattern Configurations](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#Pattern_Configurations)
  + [Signal Shutdown](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#Signal_Shutdown)
* [How to Implement the Producer-Consumer Pattern](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#How_to_Implement_the_Producer-Consumer_Pattern)
  + [Producer Function](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#Producer_Function)
  + [Consumer Function](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#Consumer_Function)
  + [Shared Queue](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#Shared_Queue)
* [Example of One Producer and One Consumer](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#Example_of_One_Producer_and_One_Consumer)
  + [Producer Task](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#Producer_Task)
  + [Consumer Task](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#Consumer_Task)
  + [Create Queue and Threads](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#Create_Queue_and_Threads)
  + [Complete Example](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#Complete_Example)
* [Example of Multiple Producers and One Consumer](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#Example_of_Multiple_Producers_and_One_Consumer)
* [Example of One Producer and Multiple Consumers](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#Example_of_One_Producer_and_Multiple_Consumers)
* [Example of Multiple Producers and Multiple Consumers](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#Example_of_Multiple_Producers_and_Multiple_Consumers)
* [Further Reading](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#Further_Reading)
* [Takeaways](https://superfastpython.com/thread-producer-consumer-pattern-in-python/#Takeaways)

**What is the Producer-Consumer Pattern**

The [Producer-Consumer pattern](https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem) is a common concurrency programming design pattern.

It involves producer tasks, consumer tasks and a shared buffer or queue that connects these two types of tasks.

* **Producer Tasks**: Generate work that is added to the shared buffer.
* **Consumer Tasks**: Remove work from the shared buffer and process it in some way.
* **Shared Buffer**: Thread-safe data structure that connects producers and consumers that may be bounded.

**Producer Tasks**

Producer tasks are responsible for generating items or data and placing them on the shared buffer.

They produce work to be consumed.

Producers may run until there are no further items to generate.

**Consumer Tasks**

Consumer tasks are responsible for removing items or data off the shared buffer and processing them in some way.

If there are no items in the shared buffer, a consumer task may wait or block until items become available.

Consumer tasks may run until there are no further items to consume.

**Shared Buffer**

The shared buffer may or may not be bounded, meaning it may have a limited size. If so, producer tasks may have to wait or block until space becomes available to add new items.

Additionally, the shared buffer must be thread-safe, ensuring that items can be added and removed concurrently by multiple threads or processes without race condition or corruption of the internal data structures.

**Pattern Configurations**

There may be one or more producers and one or more consumer tasks operating on the same shared buffer concurrently.

For example, common configurations include:

* One Producer, One Consumer.
* Multiple Producers, One Consumer.
* One Producer, Multiple Consumers.
* Multiple Producers, Multiple Consumers.

**Signal Shutdown**

Producer tasks may signal to consumer tasks that no further items will be generated, in which case the system may shutdown. This is often achieved via a [sentinel object](https://en.wikipedia.org/wiki/Sentinel_value) that is added to the shared buffer by a producer task and read and interpreted by consumer tasks.

* **Sentinel Value**: Message added to the shared queue by producer tasks to indicate no further items, read by consumer tasks to signal a shutdown.

Now that we are familiar with the producer-consumer pattern for concurrency programming, let’s look at how we might implement it in Python.

Run loops using all CPUs, [download your FREE book](https://superfastpython.com/plip-incontent) to learn how.

**How to Implement the Producer-Consumer Pattern**

The producer and consumer tasks can be implemented using threads and the shared buffer can be implemented using a queue.

**Producer Function**

A producer task can be defined using a custom function. The function will take the shared buffer as an argument and other application-specific details. The task will loop and be responsible for generating items and adding them to the shared buffer each iteration.

For example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | # producer task  def producer(buffer):  while True:  # generate an item  item = ...  # add it to the buffer  buffer.put(item) |

**Consumer Function**

Similarly, the consumer task can be defined using a custom function that takes the shared buffer. The task will loop and read one item from the queue each iteration.

For example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | # consumer task  def consumer(buffer):  while True:  # retrieve an item  item = buffer.get()  # process the item in some way  # ... |

**Shared Queue**

The shared buffer can be implemented using a **[queue.Queue](https://docs.python.org/3/library/queue.html)**[data structure](https://docs.python.org/3/library/queue.html).

This is a thread-safe FIFO (first-in, first-out) queue.

For example:

|  |  |
| --- | --- |
| 1  2  3 | ...  # create a shared queue  buffer = queue.Queue() |

The queue may be bounded to a fixed size by specifying the “**maxsize**” argument.

For example:

|  |  |
| --- | --- |
| 1  2  3 | ...  # create a bounded shared queue  buffer = queue.Queue(maxsize=100) |

You can learn more about the queue.Queue in this tutorial:

* [Thread-Safe Queue in Python](https://superfastpython.com/thread-queue)

Run Producers and Consumers in Threads

Both the producer and consumer tasks may be executed by a new thread in Python.

The task functions can be executed directly by new threads. This can be achieved by creating new **threading.Thread** instances and specifying the function to execute via the “**target**” argument and any arguments to the function via the “**args**” argument.

Once created and configured, each thread can then be started.

For example, we may configure and start a producer thread:

|  |  |
| --- | --- |
| 1  2  3  4  5 | ...  # create a producer thread  producer\_thread = threading.Thread(target=producer, args=(buffer,))  # start the producer thread  producer\_thread.start() |

We may then create and start a consumer thread:

|  |  |
| --- | --- |
| 1  2  3  4  5 | ...  # create a consumer thread  consumer\_thread = threading.Thread(target=consumer, args=(buffer,))  # start the consumer thread  consumer\_thread.start() |

This process of creating, configuring, and starting new threads can be repeated as many times as needed for the desired number of concurrent producers and consumers.

You can learn more about executing functions in a new thread in this tutorial:

* [How to Run a Function in a New Thread in Python](https://superfastpython.com/run-function-in-new-thread/)

Now that we know how to implement the producer-consumer pattern using threads, let’s look at some worked examples.

[Download Now: Free Threading PDF Cheat Sheet](https://marvelous-writer-6152.ck.page/088fc51f28)

**Example of One Producer and One Consumer**

We can explore an example of one producer and one consumer.

In this example we will have one producer that generates ten items and adds them to the shared queue. Each item will be a random number between 0 and 1, the producer will block for that fraction of a second to simulate effort, then add the generated number and the task number as a tuple to the queue. Once finished, the producer will signal that no further items are expected.

The consumer will read items from the queue. If the item represents the sentinel value then the task will stop. Otherwise, the consumer will block for a fraction of a second defined in the item, to simulate work. It will then report the value.

The main thread will create the shared queue, create and configure each thread, then wait for all threads to finish.

Let’s take a closer look at each element.

**Producer Task**

The producer task will be defined in a new function named **producer()**.

The function will take the shared queue as an argument.

|  |  |
| --- | --- |
| 1  2  3  4 | # producer task  def producer(queue):      print('Producer: Running')      # ... |

The task will loop for a fixed number of iterations, in this case ten.

|  |  |
| --- | --- |
| 1  2  3  4 | ...  # generate items  for i in range(10):  # ... |

Each iteration, the task will generate a new random number via call to **random.random()**, block for this fraction of a second, create a tuple from the task number and the value and add the item to the shared queue.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | ...  # generate a value  value = random()  # block, to simulate effort  sleep(value)  # create a tuple  item = (i, value)  # add to the queue  queue.put(item)  # report progress  print(f'>producer added {item}') |

After all items have been added to the queue, the producer will send the sentinel value to indicate that no further items will be generated. In this case, we will use the object None.

|  |  |
| --- | --- |
| 1  2  3  4 | ...  # signal that there are no further items  queue.put(None)  print('Producer: Done') |

Tying this together, the complete **producer()** function is listed below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | # producer task  def producer(queue):      print('Producer: Running')      # generate items      for i in range(10):          # generate a value          value = random()          # block, to simulate effort          sleep(value)          # create a tuple          item = (i, value)          # add to the queue          queue.put(item)          # report progress          print(f'>producer added {item}')      # signal that there are no further items      queue.put(None)      print('Producer: Done') |

Next, let’s develop a consumer task.

**Consumer Task**

We can create a custom function for the consumer task named **consumer()**.

|  |  |
| --- | --- |
| 1  2  3  4 | # consumer task  def consumer(queue):      print('Consumer: Running')      # ... |

The task will loop forever.

|  |  |
| --- | --- |
| 1  2  3  4 | ...  # consume items  while True:      # ... |

Each iteration of the task loop, an item from the queue will be retrieved. If there are no items on the queue, then the call will block until an item becomes available.

|  |  |
| --- | --- |
| 1  2  3 | ...  # get a unit of work  item = queue.get() |

If the item represents the sentinel value, then the task loop will be broken and the task function will return.

|  |  |
| --- | --- |
| 1  2  3  4 | ...  # check for stop  if item is None:      break |

Otherwise, the task will block for a fraction of a second to simulate computational effort via a call to **time.sleep()** and will then report the value.

|  |  |
| --- | --- |
| 1  2  3  4  5 | ...  # block, to simulate effort  sleep(item[1])  # report  print(f'>consumer got {item}') |

Tying this together, the complete **consumer()** function is listed below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | # consumer task  def consumer(queue):      print('Consumer: Running')      # consume items      while True:          # get a unit of work          item = queue.get()          # check for stop          if item is None:              break          # block, to simulate effort          sleep(item[1])          # report          print(f'>consumer got {item}')      # all done      print('Consumer: Done') |

Next, let’s create the shared queue and threads.

**Create Queue and Threads**

The main thread will drive the application.

First, we can create the queue instance that will be shared between the threads.

|  |  |
| --- | --- |
| 1  2  3 | ...  # create the shared queue  queue = Queue() |

Next, we can create and configure the consumer threads to execute our **consumer()** function and pass in the shared queue as an argument. Once created, the thread can be started immediately and the task will wait on the queue for new items to arrive.

|  |  |
| --- | --- |
| 1  2  3  4 | ...  # start the consumer  consumer = Thread(target=consumer, args=(queue,))  consumer.start() |

We can then create and confer a producer thread to execute our **producer()** function and take the shared queue as an argument, then start it immediately.

|  |  |
| --- | --- |
| 1  2  3  4 | ...  # start the producer  producer = Thread(target=producer, args=(queue,))  producer.start() |

Finally, the main thread can join the producer thread, then the consumer thread, blocking until each thread terminates.

|  |  |
| --- | --- |
| 1  2  3  4 | ...  # wait for all threads to finish  producer.join()  consumer.join() |

If you are new to waiting for threads to terminate via **join()**, you can learn more in this tutorial:

* [How to Join a Thread in Python](https://superfastpython.com/join-a-thread-in-python/)

**Complete Example**

Tying this together, the complete example is listed below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | # SuperFastPython.com  # example of one producer and one consumer with threads  from time import sleep  from random import random  from threading import Thread  from queue import Queue    # producer task  def producer(queue):      print('Producer: Running')      # generate items      for i in range(10):          # generate a value          value = random()          # block, to simulate effort          sleep(value)          # create a tuple          item = (i, value)          # add to the queue          queue.put(item)          # report progress          print(f'>producer added {item}')      # signal that there are no further items      queue.put(None)      print('Producer: Done')    # consumer task  def consumer(queue):      print('Consumer: Running')      # consume items      while True:          # get a unit of work          item = queue.get()          # check for stop          if item is None:              break          # block, to simulate effort          sleep(item[1])          # report          print(f'>consumer got {item}')      # all done      print('Consumer: Done')    # create the shared queue  queue = Queue()  # start the consumer  consumer = Thread(target=consumer, args=(queue,))  consumer.start()  # start the producer  producer = Thread(target=producer, args=(queue,))  producer.start()  # wait for all threads to finish  producer.join()  consumer.join() |

Running the example first creates the shared queue.

Next, the consumer thread is configured and started, followed by the producer thread. The main thread then blocks until both threads terminate.

The consumer thread waits until items are added to the queue.

Each iteration, the producer thread generates a work item, blocks, then adds it to the shared queue. The consumer thread reads these items as fast as it is able, blocks, then reports their values.

Once the producer has generated and added all ten items to the shared queue, it exits its task loop and sends the sentinel value. The producer thread then terminates. The main thread notices, then continues on to block on the consumer thread.

The consumer thread processes all items in the queue until it reaches the sentinel value. The consumer thread then breaks the task loop, returns from the function and its thread terminates.

The main thread notices that the consumer thread has terminated, then terminates itself.

This highlights how to develop a single producer and single consumer pattern with threads and a queue.

A sample of the output from the program is listed below. Note, your specific results will differ given the use of random numbers.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | Consumer: Running  Producer: Running  >producer added (0, 0.1773209157029404)  >producer added (1, 0.12768228631118883)  >producer added (2, 0.04267713210595103)  >consumer got (0, 0.1773209157029404)  >consumer got (1, 0.12768228631118883)  >consumer got (2, 0.04267713210595103)  >producer added (3, 0.6854336941642073)  >consumer got (3, 0.6854336941642073)  >producer added (4, 0.7528676639191877)  >producer added (5, 0.700167348662827)  >consumer got (4, 0.7528676639191877)  >producer added (6, 0.565060022931722)  >consumer got (5, 0.700167348662827)  >producer added (7, 0.39424897042984364)  >producer added (8, 0.121766083440221)  >consumer got (6, 0.565060022931722)  >producer added (9, 0.5824679674664152)  Producer: Done  >consumer got (7, 0.39424897042984364)  >consumer got (8, 0.121766083440221)  >consumer got (9, 0.5824679674664152)  Consumer: Done |

Next, let’s look at an example with multiple producers and a single consumer.

**Free Python Threading Course**

Download your **FREE threading PDF cheat sheet** and get BONUS access to my free 7-day crash course on the threading API.

Discover how to use the Python threading module including how to create and start new threads and how to use a mutex locks and semaphores

[**Learn more**](https://marvelous-writer-6152.ck.page/088fc51f28)

**Example of Multiple Producers and One Consumer**

We can explore how to have multiple producers and a single consumer.

Each producer can execute the same task, but in a separate thread and adding items to the same shared queue.

A problem when having multiple producers is how to signal to the consumer that there are no more tasks expected from any producers.

For example, if each producer sent a sentinel item on the queue when they finished, then the consumer would stop processing items as soon as the first sentinel item was observed. THe problem with this approach is that it may leave unprocessed items on the queue as not all producer tasks would finish at the same time.

A solution might involve the consumer keeping track of how many producers have finished and only break the task loop after the expected number of producers send a sentinel item.

An alternate solution involves using a barrier, provided by the **threading.Barrier** class.

Each producer would wait on the barrier when finishing producing items. Once all producers reach the barrier, then a single producer would send a sentinel item to the consumer to indicate that no further items are expected.

If you are new to barriers, you can learn more in this tutorial:

* [How to Use a Thread Barrier in Python](https://superfastpython.com/thread-barrier-in-python/)

We can update the above example to achieve this.

Firstly, we can update the **producer()** function.

The function will take a barrier instance shared among all producers, the shared queue instance, and a unique integer identifier assigned to the producer. The producer task will use its unique id when reporting progress.

|  |  |
| --- | --- |
| 1  2  3  4 | # producer task  def producer(barrier, queue, identifier):      print(f'Producer {identifier}: Running')      # ... |

Once the task has finished, the producer can wait on the barrier for all other producers to finish.

|  |  |
| --- | --- |
| 1  2  3 | ...  # wait for all producers to finish  barrier.wait() |

Once all producers make it to the barrier, it will be released and we know that no further items will be added to the shared queue.

At this point a single producer can add a sentinel item, such as the producer task with the identifier of zero.

|  |  |
| --- | --- |
| 1  2  3  4 | ...  # signal that there are no further items  if identifier == 0:      queue.put(None) |

The updated **producer()** function with these changes is listed below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | # producer task  def producer(barrier, queue, identifier):      print(f'Producer {identifier}: Running')      # generate items      for i in range(5):          # generate a value          value = random()          # block, to simulate effort          sleep(value)          # create a tuple          item = (i, value)          # add to the queue          queue.put(item)      # wait for all producers to finish      barrier.wait()      # signal that there are no further items      if identifier == 0:          queue.put(None)      print(f'Producer {identifier}: Done') |

In the main thread, we can create a new **threading.Barrier** instance configured with the expected number of producer threads.

In this case, we will limit the number of producers to three.

|  |  |
| --- | --- |
| 1  2  3  4 | ...  # create the shared barrier  n\_producers = 3  barrier = Barrier(n\_producers) |

We can then create each producer thread, configured to call the same **producer()** function with the same shared barrier and queue. Each thread will be assigned a unique integer from 0 to 2.

This can be achieved in a list comprehension, for example:

|  |  |
| --- | --- |
| 1  2  3 | ...  # start the producers  producers = [Thread(target=producer, args=(barrier,queue,i)) for i in range(n\_producers)] |

The producer threads can then be started in a loop.

|  |  |
| --- | --- |
| 1  2  3  4 | ...  # start the producers  for producer in producers:      producer.start() |

Finally, the main thread can wait for each producer thread to finish, also in a loop.

|  |  |
| --- | --- |
| 1  2  3  4 | ...  # wait for all threads to finish  for producer in producers:      producer.join() |

Tying this together, the complete example is listed below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | # SuperFastPython.com  # example of multiple producers and one consumer with threads  from time import sleep  from random import random  from threading import Thread  from threading import Barrier  from queue import Queue    # producer task  def producer(barrier, queue, identifier):      print(f'Producer {identifier}: Running')      # generate items      for i in range(5):          # generate a value          value = random()          # block, to simulate effort          sleep(value)          # create a tuple          item = (i, value)          # add to the queue          queue.put(item)      # wait for all producers to finish      barrier.wait()      # signal that there are no further items      if identifier == 0:          queue.put(None)      print(f'Producer {identifier}: Done')    # consumer task  def consumer(queue):      print('Consumer: Running')      # consume items      while True:          # get a unit of work          item = queue.get()          # check for stop          if item is None:              break          # block, to simulate effort          sleep(item[1])          # report          print(f'>consumer got {item}')      # all done      print('Consumer: Done')    # create the shared queue  queue = Queue()  # create the shared barrier  n\_producers = 3  barrier = Barrier(n\_producers)  # start the consumer  consumer = Thread(target=consumer, args=(queue,))  consumer.start()  # start the producers  producers = [Thread(target=producer, args=(barrier,queue,i)) for i in range(n\_producers)]  # start the producers  for producer in producers:      producer.start()  # wait for all threads to finish  for producer in producers:      producer.join()  consumer.join() |

Running the example first creates the shared queue and the barrier to be shared among all producer threads.

The single consumer thread is created and started, awaiting items to be added to the queue. The three producer threads are then created and started in a loop. The main thread then blocks waiting for all threads to finish.

Each producer executes the **producer()** function, looping and each adding five items to the queue.

The consumer retrieves items from the queue as fast as it is able, reporting each item in turn.

As each producer thread finishes it reaches the barrier and waits for all other producers to finish. After all producers reach the barrier, the barrier is released. The thread with the identifier of zero sends the sentinel value into the queue, and all producer threads terminate.

The main thread notices that the producer threads have all terminated, then proceeds to block on the consumer thread until it terminates.

The consumer thread receives the single sentinel value, breaks the task loop and terminates.

Finally the main thread stops blocking and also terminates.

This demonstrates how to develop a producer-consumer pattern with multiple consumers and a barrier to coordinate the shutdown of the system.

A sample of the output from the program is listed below. Note, your specific results will differ given the use of random numbers.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | Consumer: Running  Producer 0: Running  Producer 1: Running  Producer 2: Running  >consumer got (0, 0.7575027685720114)  >consumer got (0, 0.7863614605516718)  Producer 1: Done  Producer 0: Done  Producer 2: Done  >consumer got (0, 0.8042275144987115)  >consumer got (1, 0.04813013635540675)  >consumer got (1, 0.2955931977923576)  >consumer got (1, 0.533589747825717)  >consumer got (2, 0.36505264173769114)  >consumer got (2, 0.8385568324386631)  >consumer got (2, 0.5629464266889235)  >consumer got (3, 0.11311883387252708)  >consumer got (3, 0.40376008090688753)  >consumer got (4, 0.2715376798261393)  >consumer got (3, 0.9574661544383317)  >consumer got (4, 0.35128687833420313)  >consumer got (4, 0.5709439705809225)  Consumer: Done |

Next, let’s look at an example with a single producer and multiple consumers.

**Overwhelmed by the python concurrency APIs?**  
Find relief, download my FREE [Python Concurrency Mind Maps](https://marvelous-writer-6152.ck.page/8f23adb076)

**Example of One Producer and Multiple Consumers**

We can explore an example of the producer-consumer pattern with a single producer and multiple consumers.

A key concern in this configuration is how a single producer can signal to all consumer threads that no further items are expected.

When using a single producer and single consumer, the producer signals to the consumer that there are no further items via a sentinel value. If there are multiple consumers, then this approach would only stop a single consumer.

One solution is to use an event, via the **threading.Event** class. All threads could share the instance. The producer can then set the event once finished, and all consumers can check the status of the event each iteration. Consumers will block on the queue for new items and may not notice the change in the status of the event. Therefore, the consumers would have to be updated to use a timeout on the call to **get()** in a type of busy waiting.

You can learn more about busy waiting in this tutorial:

* [How to Use Busy Waiting in Python](https://superfastpython.com/thread-busy-waiting-in-python/)

An alternate solution is to use a sentinel value, and for the consumer task to be configured to add the sentinel back to the queue before breaking the task loop.

This will allow each consumer task to retrieve the value and respond by shutting down, without the need for a busy wait loop.

This can be achieved by updating the above single producer, single consumer example.

First, the **consumer()** function can be updated to take a unique integer identifier, used in reporting.

|  |  |
| --- | --- |
| 1  2  3  4 | # consumer task  def consumer(queue, identifier):      print(f'Consumer {identifier}: Running')      # ... |

When checking if the value retrieved from the queue is the sentinel value, the task can re-add the item if the sentinel is identified before terminating the task.

For example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | ...  # check for stop  if item is None:      # add the signal back for other consumers      queue.put(item)      # stop running      break |

Tying this together, the **consumer()** function with these changes is listed below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | # consumer task  def consumer(queue, identifier):      print(f'Consumer {identifier}: Running')      # consume items      while True:          # get a unit of work          item = queue.get()          # check for stop          if item is None:              # add the signal back for other consumers              queue.put(item)              # stop running              break          # block, to simulate effort          sleep(item[1])          # report          print(f'>consumer {identifier} got {item}')      # all done      print(f'Consumer {identifier}: Done') |

In the main thread, we can then create a number of consumer threads, each configured to execute our **consumer()** function with a unique integer for each thread.

This can be achieved in a list comprehension.

|  |  |
| --- | --- |
| 1  2  3 | ...  # start the consumers  consumers = [Thread(target=consumer, args=(queue,i)) for i in range(3)] |

The consumer threads can then be started in a loop.

|  |  |
| --- | --- |
| 1  2  3 | ...  for consumer in consumers:      consumer.start() |

Finally, the main thread can block on each consumer thread in a loop.

|  |  |
| --- | --- |
| 1  2  3 | ...  for consumer in consumers:      consumer.join() |

Tying this together, the complete example is listed below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | # SuperFastPython.com  # example of one producer and multiple consumers with threads  from time import sleep  from random import random  from threading import Thread  from queue import Queue    # producer task  def producer(queue):      print('Producer: Running')      # generate items      for i in range(10):          # generate a value          value = random()          # block, to simulate effort          sleep(value)          # create a tuple          item = (i, value)          # add to the queue          queue.put(item)      # signal that there are no further items      queue.put(None)      print('Producer: Done')    # consumer task  def consumer(queue, identifier):      print(f'Consumer {identifier}: Running')      # consume items      while True:          # get a unit of work          item = queue.get()          # check for stop          if item is None:              # add the signal back for other consumers              queue.put(item)              # stop running              break          # block, to simulate effort          sleep(item[1])          # report          print(f'>consumer {identifier} got {item}')      # all done      print(f'Consumer {identifier}: Done')    # create the shared queue  queue = Queue()  # start the consumers  consumers = [Thread(target=consumer, args=(queue,i)) for i in range(3)]  for consumer in consumers:      consumer.start()  # start the producer  producer = Thread(target=producer, args=(queue,))  producer.start()  # wait for all threads to finish  producer.join()  for consumer in consumers:      consumer.join() |

Running the example first creates the shared queue.

The three consumer threads are created, each waiting on the queue for items to be added. Then the single producer is configured and started.

The producer then loops, generating and adding items to the queue.

The consumers each retrieve values as fast as they can from the queue, reporting values as they are retrieved.

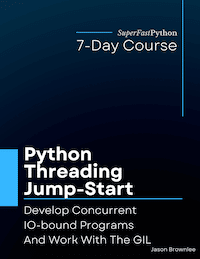
The producer finishes adding all ten items then adds the sentinel value before terminating.

The consumer tasks read the sentinel value, re-add it to the queue then exit the task loop. This allows the sentinel value to pass among the consumer threads until they all terminate.

Finally, the main thread unblocks and terminates.

A sample of the output from the program is listed below. Note, your specific results will differ given the use of random numbers.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | Consumer 0: Running  Consumer 1: Running  Consumer 2: Running  Producer: Running  >consumer 0 got (0, 0.2835145997612327)  >consumer 2 got (2, 0.27930380418463674)  >consumer 1 got (1, 0.566373500953346)  >consumer 0 got (3, 0.5297369117447445)  >consumer 2 got (4, 0.4227063111625722)  >consumer 1 got (5, 0.6635175225918853)  >consumer 0 got (6, 0.4275806337971696)  >consumer 1 got (8, 0.4479799894214017)  >consumer 2 got (7, 0.9443461021850552)  Producer: Done  Consumer 0: Done  Consumer 2: Done  >consumer 1 got (9, 0.7236589490262505)  Consumer 1: Done |

[](https://superfastpython.com/ptj-incontent)

**Loving The Tutorials?**

Why not take the next step? Get the book.

[**Learn more**](https://superfastpython.com/ptj-incontent)

**Example of Multiple Producers and Multiple Consumers**

We can explore how to develop an example of the producer-consumer pattern with multiple producers and multiple consumers.

This requires a way for the producers to coordinate in the shutdown of the system, signaling to the consumers that no further tasks are expected. It also requires that the consumers coordinate in their shutdown, ensuring that all consumers shutdown together.

The mechanisms developed above may be used. Specifically, the use of a barrier to coordinate the producers and the use of consumers re-adding the sentinel value in order to coordinate consumers.

We can update the previous examples above and reuse the solutions in order to achieve this.

In this case, we will have three producers, each generating and adding ten items, or thirty items in total onto the shared queue. We will then have five consumers, each retrieving and processing items from the shared queue.

Tying this together, the complete example is listed below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67 | # SuperFastPython.com  # example of multiple producers and multiple consumers with threads  from time import sleep  from random import random  from threading import Thread  from threading import Barrier  from queue import Queue    # producer task  def producer(barrier, queue, identifier):      print(f'Producer {identifier}: Running')      # generate items      for i in range(10):          # generate a value          value = random()          # block, to simulate effort          sleep(value)          # create a tuple          item = (i, value)          # add to the queue          queue.put(item)      # wait for all producers to finish      barrier.wait()      # signal that there are no further items      if identifier == 0:          queue.put(None)      print(f'Producer {identifier}: Done')    # consumer task  def consumer(queue, identifier):      print(f'Consumer {identifier}: Running')      # consume items      while True:          # get a unit of work          item = queue.get()          # check for stop          if item is None:              # add the signal back for other consumers              queue.put(item)              # stop running              break          # block, to simulate effort          sleep(item[1])          # report          print(f'>consumer {identifier} got {item}')      # all done      print(f'Consumer {identifier}: Done')    # create the shared queue  queue = Queue()  # create the shared barrier  n\_producers = 3  barrier = Barrier(n\_producers)  # start the consumers  consumers = [Thread(target=consumer, args=(queue,i)) for i in range(5)]  for consumer in consumers:      consumer.start()  # start the producers  producers = [Thread(target=producer, args=(barrier,queue,i)) for i in range(n\_producers)]  # start the producers  for producer in producers:      producer.start()  # wait for all threads to finish  for producer in producers:      producer.join()  for consumer in consumers:      consumer.join() |

Running the example first creates the shared queue and shared barrier instance.

The five consumer threads are created and started, each waiting on the queue for items to be added. The three producer threads are then created and started in a loop. The main thread then blocks waiting for all threads to finish.

Each producer executes the **producer()** function, looping and each adding ten items to the queue per thread.

The consumers each retrieve values as fast as they can from the queue, reporting values as they are retrieved.

As each producer thread finishes it reaches the barrier and waits for all other producers to finish. After all producers reach the barrier, the barrier is released. The thread with the identifier of zero sends the sentinel value into the queue, and all producer threads terminate.

After all producers reach the barrier, the barrier is released. The thread with the identifier of zero sends the sentinel value into the queue, and all producer threads terminate.

The consumer tasks read the sentinel value, re-add it to the queue then exit the task loop. This allows the sentinel value to pass among the consumer threads until they all terminate.

Finally, the main thread unblocks and terminates.

A sample of the output from the program is listed below. Note, your specific results will differ given the use of random numbers.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | Consumer 0: Running  Consumer 1: Running  Consumer 2: Running  Consumer 3: Running  Consumer 4: Running  Producer 0: Running  Producer 1: Running  Producer 2: Running  >consumer 0 got (0, 0.20092555155126923)  >consumer 4 got (2, 0.20457644408105113)  >consumer 1 got (0, 0.5724478360191372)  >consumer 2 got (1, 0.4761963602746929)  >consumer 1 got (2, 0.11432851526590049)  >consumer 4 got (1, 0.2340636925920332)  >consumer 0 got (1, 0.40858346196426076)  >consumer 3 got (0, 0.8439680058429783)  >consumer 2 got (3, 0.3974922482327897)  >consumer 1 got (2, 0.4634011848493308)  >consumer 0 got (4, 0.37054000908333773)  >consumer 4 got (3, 0.8314728938802879)  >consumer 3 got (3, 0.7198941330353915)  >consumer 2 got (5, 0.5759733508184562)  >consumer 0 got (5, 0.41901096962439854)  >consumer 1 got (4, 0.8485923753562503)  >consumer 2 got (7, 0.3046756277568494)  >consumer 4 got (4, 0.7881678301544097)  >consumer 3 got (6, 0.7132345803199032)  >consumer 2 got (8, 0.42674047473416477)  >consumer 1 got (5, 0.7094520643527811)  >consumer 0 got (6, 0.7740155089220191)  >consumer 4 got (7, 0.40037840046367257)  >consumer 4 got (8, 0.10490771681909827)  >consumer 2 got (7, 0.22082320027800584)  >consumer 3 got (6, 0.7249681823659646)  >consumer 1 got (8, 0.5651264978475339)  Producer 0: Done  Producer 2: Done  Producer 1: Done  Consumer 2: Done  Consumer 1: Done  >consumer 4 got (9, 0.33737564779536533)  Consumer 4: Done  >consumer 0 got (9, 0.8959814891654937)  Consumer 0: Done  >consumer 3 got (9, 0.7397967179285265)  Consumer 3: Done |

**Further Reading**

This section provides additional resources that you may find helpful.

**Python Threading Books**

* [Python Threading Jump-Start](https://superfastpython.com/ptj-further-reading), Jason Brownlee (**my book!**)
* [Threading API Interview Questions](https://superfastpython.com/python-threading-interview-questions/)
* [Threading Module API Cheat Sheet](https://marvelous-writer-6152.ck.page/088fc51f28)

I also recommend specific chapters in the following books:

* [Python Cookbook](https://amzn.to/3MSFzBv), David Beazley and Brian Jones, 2013.
  + See: *Chapter 12: Concurrency*
* [Effective Python](https://amzn.to/3GpopJ1), Brett Slatkin, 2019.
  + See: *Chapter 7: Concurrency and Parallelism*
* [Python in a Nutshell](https://amzn.to/3m7SLGD), Alex Martelli, et al., 2017.
  + See: *Chapter: 14: Threads and Processes*

**Guides**

* [Python Threading: The Complete Guide](https://superfastpython.com/threading-in-python/)
* [Python ThreadPoolExecutor: The Complete Guide](https://superfastpython.com/threadpoolexecutor-in-python/)
* [Python ThreadPool: The Complete Guide](https://superfastpython.com/threadpool-python/)

**APIs**

* [threading - Thread-based parallelism](https://docs.python.org/3/library/threading.html)
* [queue — A synchronized queue class](https://docs.python.org/3/library/queue.html)

**References**

* [Thread (computing), Wikipedia.](https://en.wikipedia.org/wiki/Thread_(computing))
* [Process (computing), Wikipedia.](https://en.wikipedia.org/wiki/Process_(computing))

**Takeaways**

You now know how to use the producer-consumer pattern with threads in Python.

**Do you have any questions?**  
Ask your questions in the comments below and I will do my best to answer.

Photo by [Glenn Hansen](https://unsplash.com/@hansenhouse?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText) on [Unsplash](https://unsplash.com/s/photos/dirtbike?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText)

**Share this:**

* Share

**Related Tutorials:**

[](https://superfastpython.com/threadpool-producer-consumer/?relatedposts_hit=1&relatedposts_origin=1179&relatedposts_position=0&relatedposts_hit=1&relatedposts_origin=1179&relatedposts_position=0&relatedposts_hit=1&relatedposts_origin=1179&relatedposts_position=0)

[ThreadPool Producer-Consumer Pattern in Python](https://superfastpython.com/threadpool-producer-consumer/?relatedposts_hit=1&relatedposts_origin=1179&relatedposts_position=0&relatedposts_hit=1&relatedposts_origin=1179&relatedposts_position=0&relatedposts_hit=1&relatedposts_origin=1179&relatedposts_position=0)

[](https://superfastpython.com/multiprocessing-joinablequeue-on-python/?relatedposts_hit=1&relatedposts_origin=1179&relatedposts_position=1&relatedposts_hit=1&relatedposts_origin=1179&relatedposts_position=1&relatedposts_hit=1&relatedposts_origin=1179&relatedposts_position=1)

[Multiprocessing JoinableQueue on Python](https://superfastpython.com/multiprocessing-joinablequeue-on-python/?relatedposts_hit=1&relatedposts_origin=1179&relatedposts_position=1&relatedposts_hit=1&relatedposts_origin=1179&relatedposts_position=1&relatedposts_hit=1&relatedposts_origin=1179&relatedposts_position=1)

[](https://superfastpython.com/thread-queue-task-done-join/?relatedposts_hit=1&relatedposts_origin=1179&relatedposts_position=2&relatedposts_hit=1&relatedposts_origin=1179&relatedposts_position=2&relatedposts_hit=1&relatedposts_origin=1179&relatedposts_position=2)

[Queue task\_done() and join() in Python](https://superfastpython.com/thread-queue-task-done-join/?relatedposts_hit=1&relatedposts_origin=1179&relatedposts_position=2&relatedposts_hit=1&relatedposts_origin=1179&relatedposts_position=2&relatedposts_hit=1&relatedposts_origin=1179&relatedposts_position=2)



**About Jason Brownlee**

Hi, my name is Jason Brownlee, Ph.D. and I’m the guy behind this website. I am obsessed with Python Concurrency.

I help python developers learn concurrency, super fast.  
[Learn more](https://superfastpython.com/about).

**Parallel Loops in Python**

Discover how to run your loops in parallel, download your [free book](https://marvelous-writer-6152.ck.page/99ee689b9b) now:

Your free book "**Parallel Loops in Python**" includes complete and working code templates that you can modify and use right now in your own projects.

[Download Your FREE Book](https://marvelous-writer-6152.ck.page/99ee689b9b)

**Reader Interactions**

**Do you have any questions?**

Top of Form

Bottom of Form

**Primary Sidebar**

[](https://superfastpython.com/about)

**About Jason Brownlee**  
Hi, my name is Jason Brownlee, Ph.D. and welcome to *SuperFast*Python! My mission is to make Python developers awesome at concurrency. [Learn more](https://superfastpython.com/about).

**Join 10,000+ developers:**

**Search:**

Top of Form

Search this website

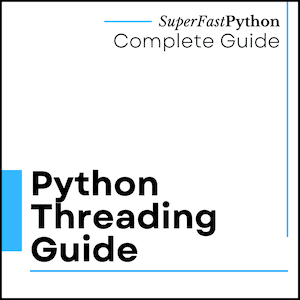
Bottom of Form

**Threading Cheat Sheet:**

Do you want to **Learn Threading Fast?**

Download your *Free PDF Cheat Sheet*.  
Get bonus access to a *7-day crash course*.  
  
[Get your PDF cheat sheet](https://marvelous-writer-6152.ck.page/088fc51f28)

**Python Threading Guide:**

[](https://superfastpython.com/ptg-sidebar)  
[Python Threading Guide](https://superfastpython.com/ptg-sidebar)

**Best Threading Resources**

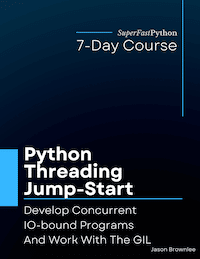
**Threading Resources:**

1. [Threading Tutorials](https://superfastpython.com/category/threading/)
2. [Threading Guide](https://superfastpython.com/ptg-sidebar)
3. [Threading Learning Path](https://superfastpython.com/learning-paths/#Threading_Learning_Path)
4. [Threading Cheat Sheet](https://marvelous-writer-6152.ck.page/088fc51f28)
5. [Threading Jump-Start Book](https://superfastpython.com/ptj-sidebar)
6. [Threading Interview Questions Book](https://superfastpython.com/ptiq-sidebar)

**Python Threading Jump-Start:**

**Loving the Tutorials?**

**[Get The Book:](https://superfastpython.com/ptj-sidebar)**

[](https://superfastpython.com/ptj-sidebar)  
[Learn More](https://superfastpython.com/ptj-sidebar)

**Python Concurrency Bookshelf:**

**Don't Dabble!**

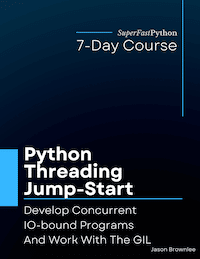
**[Learn All Of Python Concurrency](https://superfastpython.com/pjbs-sidebar)**

**No more idle CPUs**

[Learn more](https://superfastpython.com/pjbs-sidebar)

**Footer**

**Learn Threading Systematically**

[](https://superfastpython.com/ptj-footer)

What if you could develop Python programs that were concurrent from the start?

The **threading module** provides easy-to-use thread-based concurrency.

Introducing: "[Python Threading Jump-Start](https://superfastpython.com/ptj-footer)".

A new book designed to teach you the **threading module** step-by-step, super fast!

* [Privacy](https://superfastpython.com/privacy/)

* [Disclaimer](https://superfastpython.com/disclaimer/)

* [Site Map](https://superfastpython.com/site-map/)

* [About](https://superfastpython.com/about/)

* [Contact](https://superfastpython.com/contact/)

* [Praise](https://superfastpython.com/praise/)

Copyright © 2024 Super Fast Python  
[LinkedIn](https://www.linkedin.com/company/super-fast-python/) | [Twitter](https://twitter.com/SuperFastPython) | [Facebook](https://www.facebook.com/SuperFastPython) | [RSS](https://superfastpython.com/feed/)  
  
Super Fast Python Pty. Ltd.  
PO Box 206, Vermont, Victoria, 3133, Australia  
ABN: 65 654 877 433, ACN: 654 877 433  
Email: Jason@SuperFastPython.com