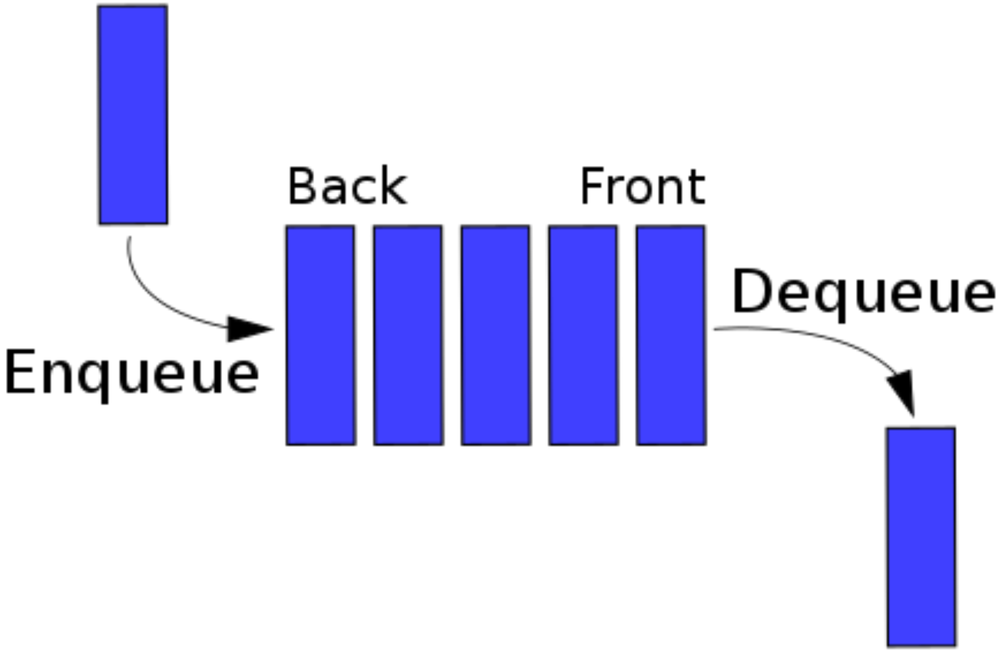
Asyncio Queue

https://github.com/hanattaw/class8-asyncio.git

Queue

- Queue is a data structure
  - items can be added by a call to **put( )**
  - items can be retrieved by a call to **get( )**



How to share data between coroutines using queues

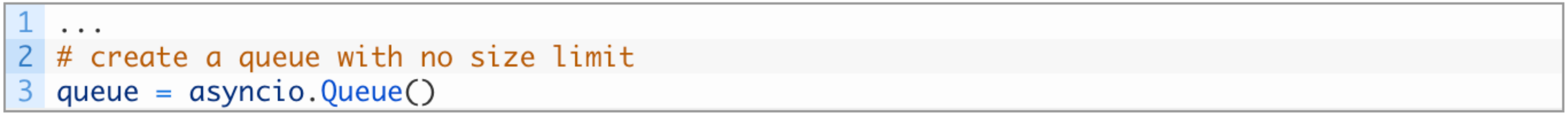
- Asyncio.Queue provides a **FIFO** queue
- Asyncio.Queue cannot be used outside of an asyncio program

How to Use and Asyncio Queue

- How to **create** and configure an instance
- How to **add** items
- How to **remove** items
- **Query** the properties of the queue and **manage** tasks.

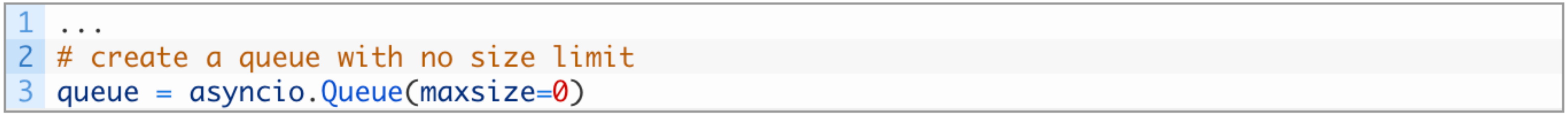
Create an Asyncio Queue

- Queue will not be limited in capacity



```python
...
# create a queue with no size limit
queue = asyncio.Queue()
```

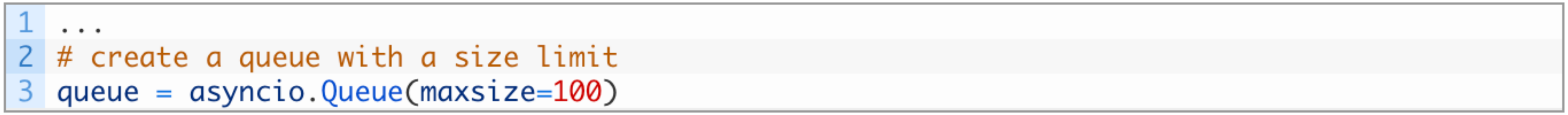
- Queue with “maxsize” set to zero (no limit) by default



```python
...
# create a queue with no size limit
queue = asyncio.Queue(maxsize=0)
```

Create an Asyncio Queue

- Queue with limit size



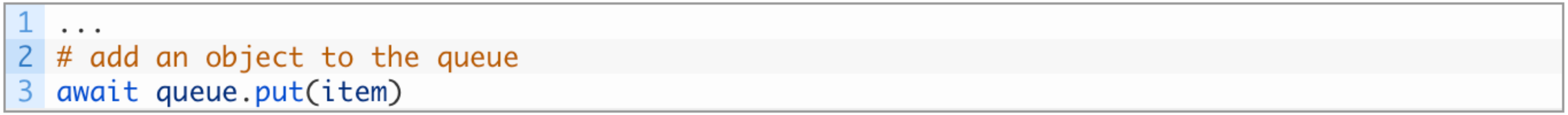
```python
...
# create a queue with a size limit
queue = asyncio.Queue(maxsize=100)
```

- When the queue is **full** and coroutines attempt to add an object, they will **block** until space becomes available, or fail if a non-blocking method is used

Add items to Asyncio Queue

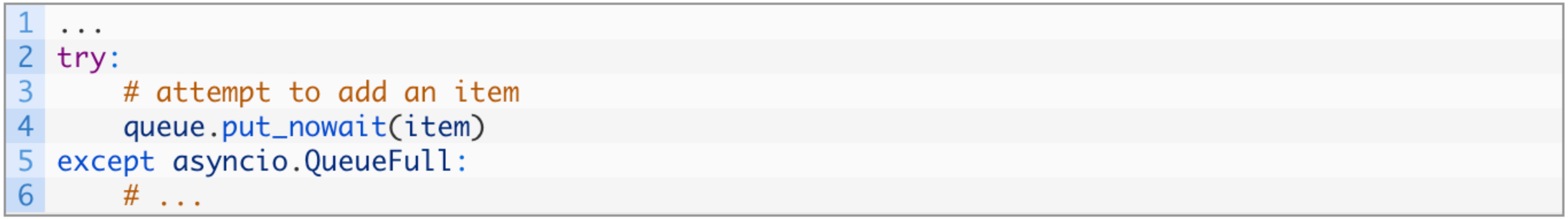
Via put( ) method

- Queue may block if queue is full.



```
...
# add an object to the queue
await queue.put(item)
```

- added to the queue without blocking via the **put_nowait**( ) method, This method is not a coroutine and will either add the item or return immediately or fail with an **asyncio.QueueFull exception** if the queue is full and the item cannot be added

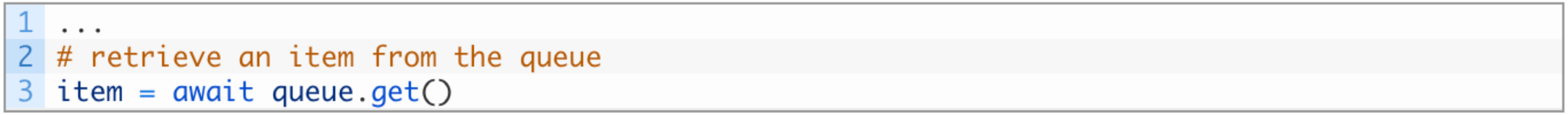


```
...
try:
    # attempt to add an item
    queue.put_nowait(item)
except asyncio.QueueFull:
    # ...
```

Get Items from Asyncio Queue

retrieved from the queue by calling the get() method

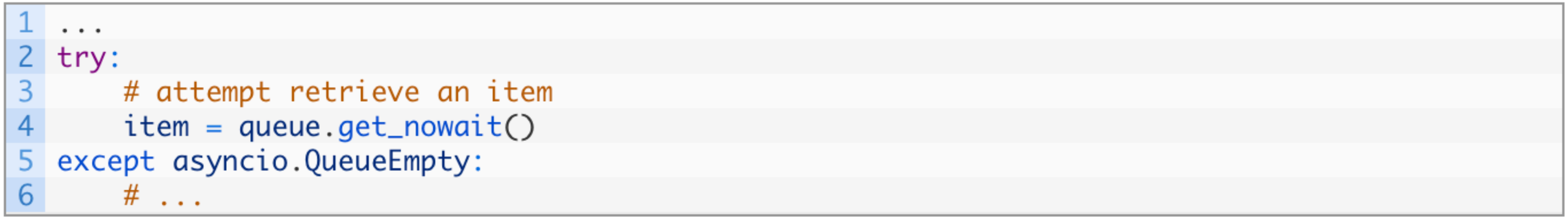
- The item retrieved will be the oldest item added, e.g. FIFO



```
...
# retrieve an item from the queue
item = await queue.get()
```

the calling coroutine may need to block until an item becomes available

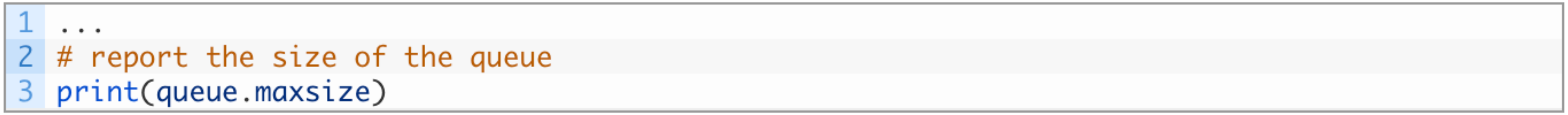
- Return an item immediately if available, otherwise will fail with exception



```
...
try:
    # attempt retrieve an item
    item = queue.get_nowait()
except asyncio.QueueEmpty:
    # ...
```

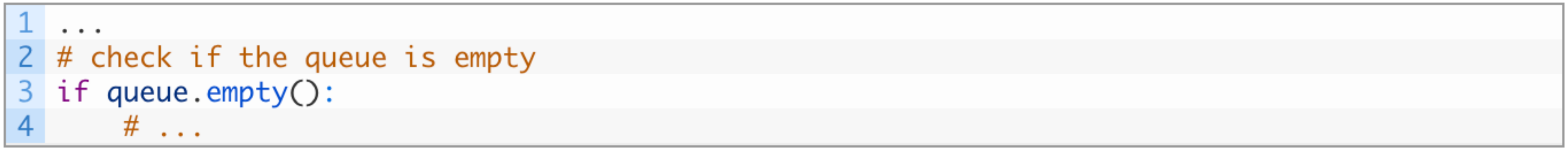
Query Asyncio Queue size

- Queue size



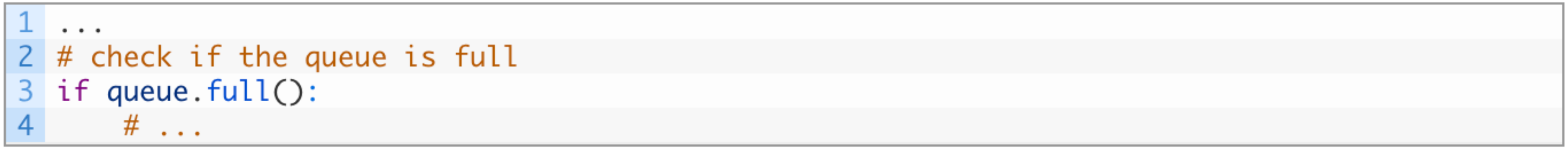
```python
...
# report the size of the queue
print(queue.maxsize)
```

- Check the queue is empty via the **empty( )**



```python
...
# check if the queue is empty
if queue.empty():
    # ...
```

- **C**heck queue is full via full( )

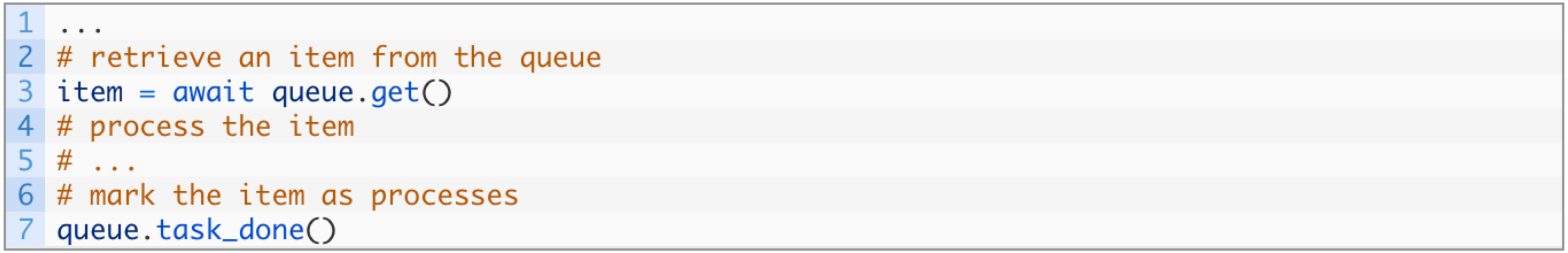


```python
...
# check if the queue is full
if queue.full():
    # ...
```

Asyncio Queue Join and Task done

- Item on the queue can be treated as **tasks** that can be marked as processes by consumer coroutines

- Once processed marking them via the task_done( )



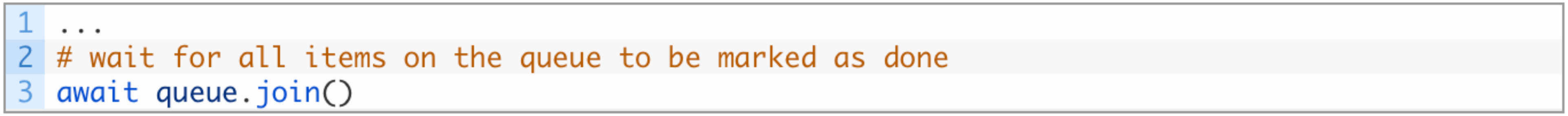
```
...
# retrieve an item from the queue
item = await queue.get()
# process the item
# ...
# mark the item as processes
queue.task_done()
```

Asyncio Queue Join and Task done

- Other **coroutines** may be interest to know **when all items added to the queue have been retrieved** and **marked** as done

- This can be achieved by the coroutine awaiting the **join( )**

- The **join( )** coroutine will not return until all items added to the queue prior to the call have been marked as done

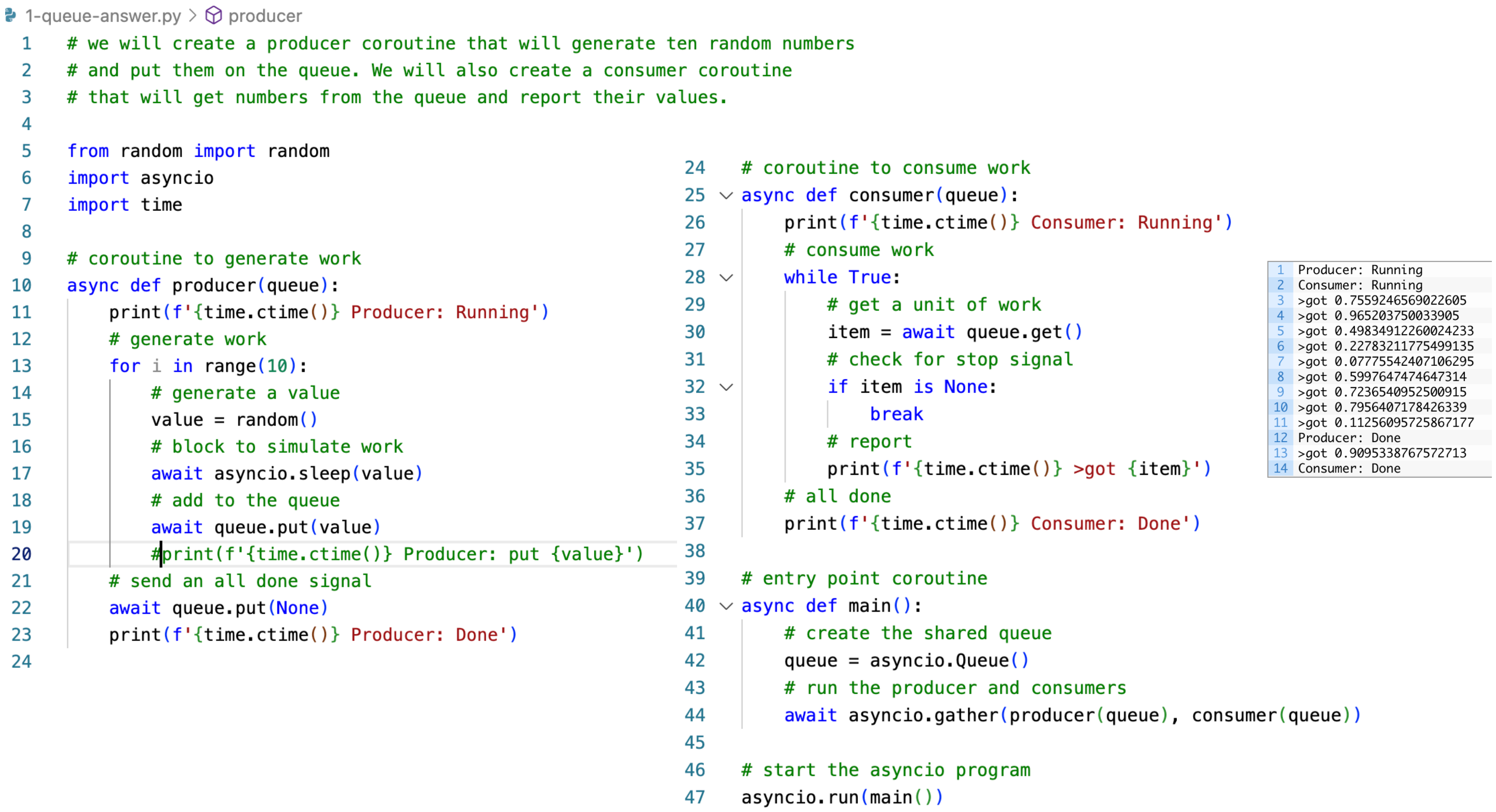


```
...
# wait for all items on the queue to be marked as done
await queue.join()
```

- If the queue is **empty** or all items have already been **marked** as **done**, then join( ) will **return immediately**

Example of Asyncio Queue

- we will create a **producer** coroutine that will generate ten random numbers and **put** them on the **queue.** We will also create a **consumer** coroutine that will **get** numbers from the queue and report their values**.**
- The **asyncio.Queue** provides a way to allow these producer and consumer coroutines to **communicate** data with each other.

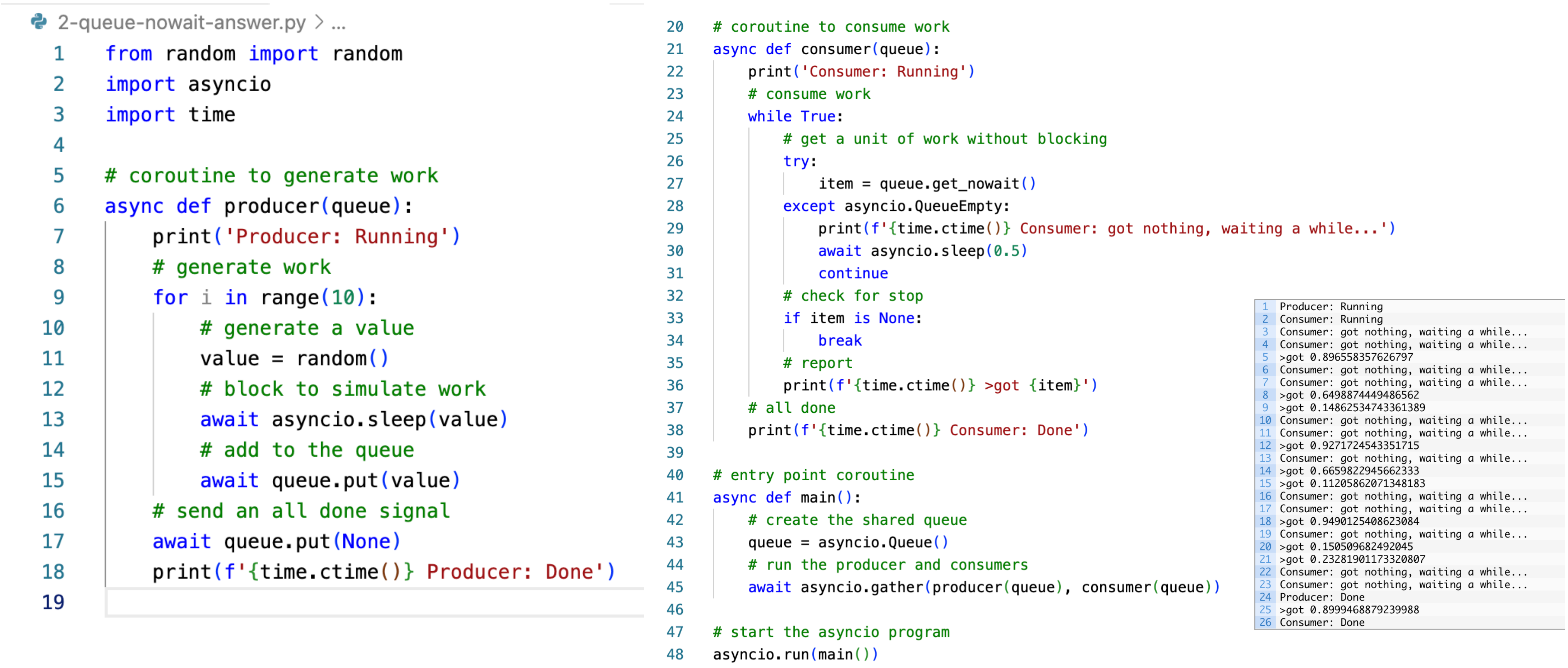


```python
# we will create a producer coroutine that will generate ten random numbers
# and put them on the queue. We will also create a consumer coroutine
# that will get numbers from the queue and report their values.

from random import random
import asyncio
import time

# coroutine to generate work
async def producer(queue):
    print(f'{time.ctime()} Producer: Running')
    # generate work
    for i in range(10):
        # generate a value
        value = random()
        # block to simulate work
        await asyncio.sleep(value)
        # add to the queue
        await queue.put(value)
        #print(f'{time.ctime()} Producer: put {value}')
    # send an all done signal
    await queue.put(None)
    print(f'{time.ctime()} Producer: Done')

# coroutine to consume work
async def consumer(queue):
    print(f'{time.ctime()} Consumer: Running')
    # consume work
    while True:
        # get a unit of work
        item = await queue.get()
        # check for stop signal
        if item is None:
            break
        # report
        print(f'{time.ctime()} >got {item}')
    # all done
    print(f'{time.ctime()} Consumer: Done')

# entry point coroutine
async def main():
    # create the shared queue
    queue = asyncio.Queue()
    # run the producer and consumers
    await asyncio.gather(producer(queue), consumer(queue))

# start the asyncio program
asyncio.run(main())
```

```
Producer: Running
Consumer: Running
>got 0.7559246569022605
>got 0.965203750033905
>got 0.49834912260024233
>got 0.22783211775499135
>got 0.07775542407106295
>got 0.5997647474647314
>got 0.7236540952500915
>got 0.7956407178426339
>got 0.11256095725867177
Producer: Done
>got 0.9095338767572713
Consumer: Done
```

Example of Asyncio Queue Without Blocking

- This can be achieved by calling the **get_nowait()** method.
- The **get_nowait()** function will return immediately. If there is a **value** in the **queue** to **retrieve**, then it is returned.
- Otherwise, if the queue is **empty**, then an **asyncio.QueueEmpty** exception is raised, which can be handled.



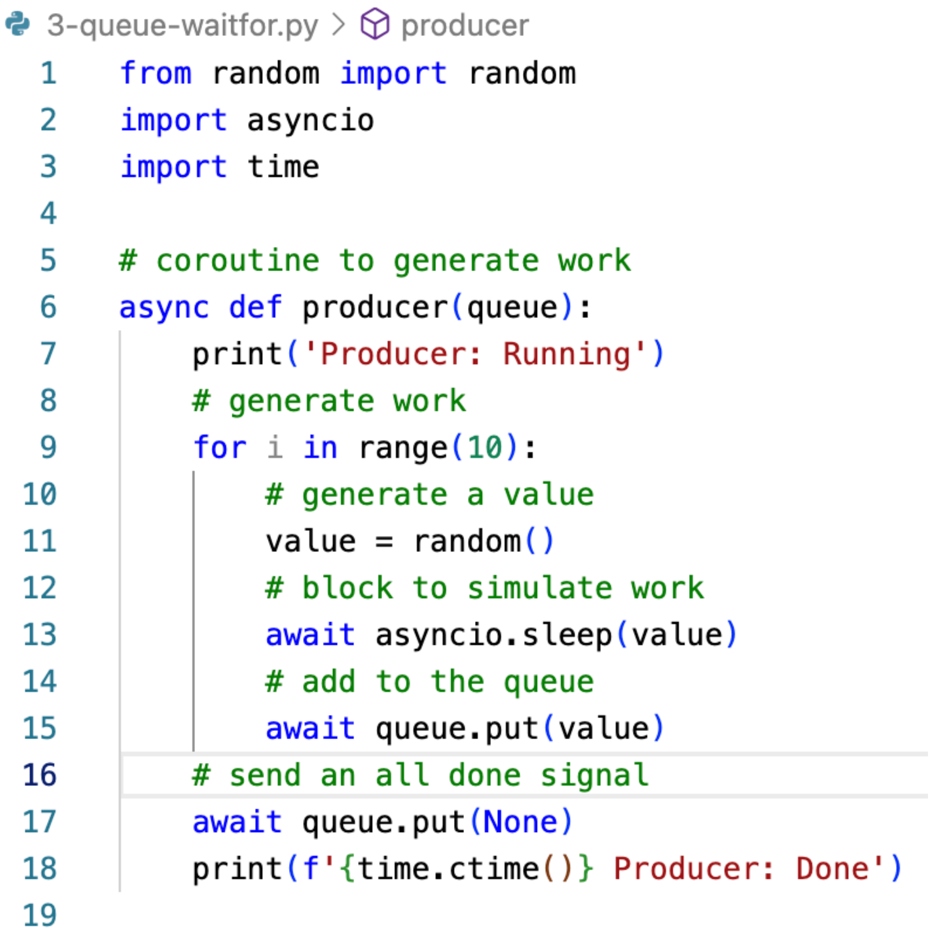
2-queue-nowait-answer.py > ...

```python
from random import random
import asyncio
import time

# coroutine to generate work
async def producer(queue):
    print('Producer: Running')
    # generate work
    for i in range(10):
        # generate a value
        value = random()
        # block to simulate work
        await asyncio.sleep(value)
        # add to the queue
        await queue.put(value)
    # send an all done signal
    await queue.put(None)
    print(f'{time.ctime()} Producer: Done')

# coroutine to consume work
async def consumer(queue):
    print('Consumer: Running')
    # consume work
    while True:
        # get a unit of work without blocking
        try:
            item = queue.get_nowait()
        except asyncio.QueueEmpty:
            print(f'{time.ctime()} Consumer: got nothing, waiting a while...')
            await asyncio.sleep(0.5)
            continue
        # check for stop
        if item is None:
            break
        # report
        print(f'{time.ctime()} >got {item}')
    # all done
    print(f'{time.ctime()} Consumer: Done')

# entry point coroutine
async def main():
    # create the shared queue
    queue = asyncio.Queue()
    # run the producer and consumers
    await asyncio.gather(producer(queue), consumer(queue))

# start the asyncio program
asyncio.run(main())
```

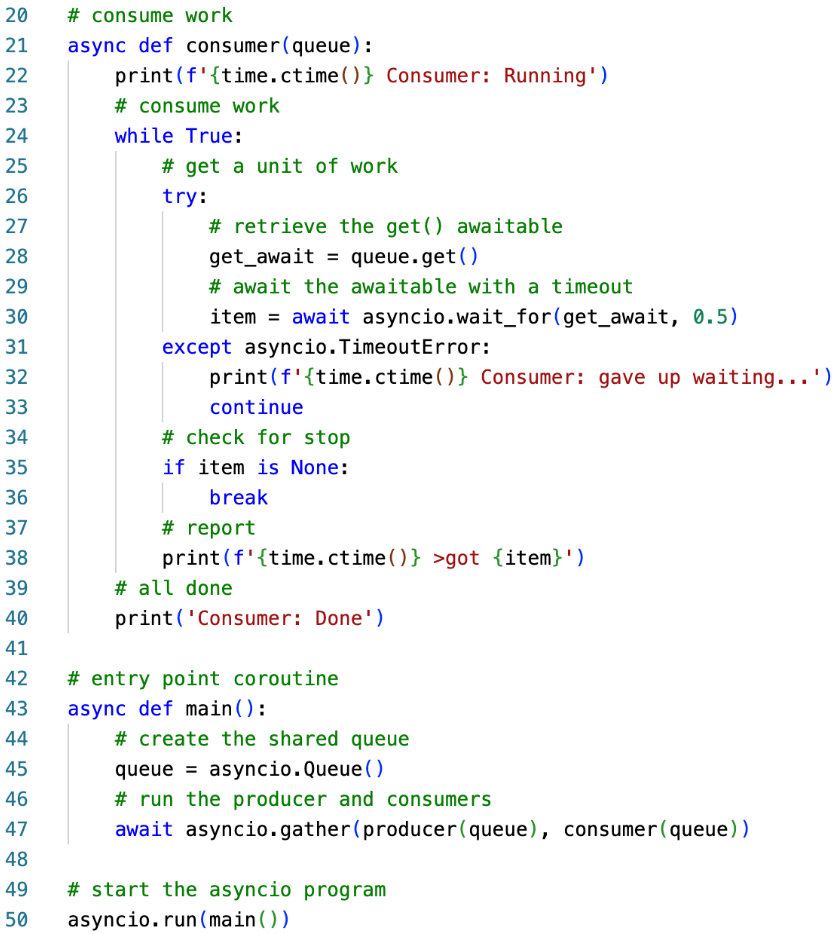
```
Producer: Running
Consumer: Running
Consumer: got nothing, waiting a while...
Consumer: got nothing, waiting a while...
>got 0.896558357626797
Consumer: got nothing, waiting a while...
Consumer: got nothing, waiting a while...
>got 0.6498874449486562
>got 0.14862534743361389
Consumer: got nothing, waiting a while...
Consumer: got nothing, waiting a while...
>got 0.9271724543351715
Consumer: got nothing, waiting a while...
>got 0.6659822945662333
>got 0.11205862071348183
Consumer: got nothing, waiting a while...
Consumer: got nothing, waiting a while...
>got 0.9490125408623084
Consumer: got nothing, waiting a while...
>got 0.150509682492045
>got 0.23281901173320807
Consumer: got nothing, waiting a while...
Consumer: got nothing, waiting a while...
Producer: Done
>got 0.8999468879239988
Consumer: Done
```

Example of Asyncio Queue With Timeout

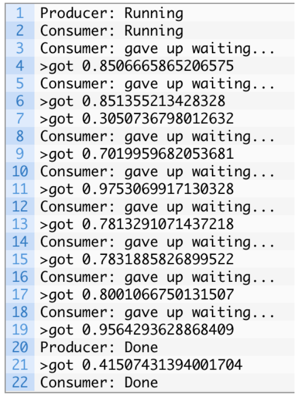
- We can get values from the asyncio.Queue by blocking but limited by a timeout
- Instead, we can wrap the **get()** coroutine in a **wait_for()** coroutine that supports a timeout. If the **timeout elapses before the get()** coroutine completes, an **asyncio.TimeoutError** exception is raised and can be handled



```python
from random import random
import asyncio
import time

# coroutine to generate work
async def producer(queue):
    print('Producer: Running')
    # generate work
    for i in range(10):
        # generate a value
        value = random()
        # block to simulate work
        await asyncio.sleep(value)
        # add to the queue
        await queue.put(value)
    # send an all done signal
    await queue.put(None)
    print(f'{time.ctime()} Producer: Done')

```



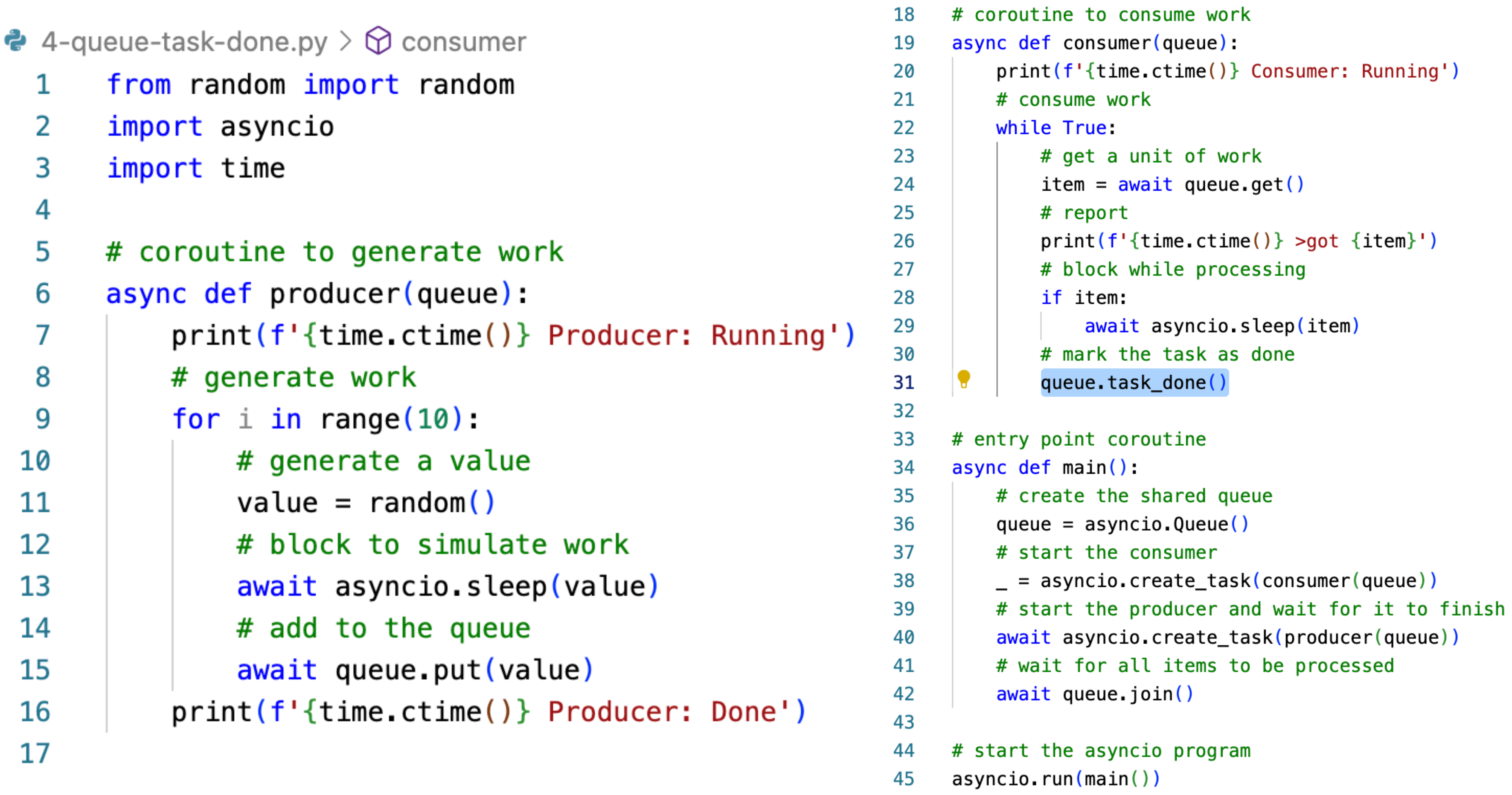
```python
# consume work
async def consumer(queue):
    print(f'{time.ctime()} Consumer: Running')
    # consume work
    while True:
        # get a unit of work
        try:
            # retrieve the get() awaitable
            get_await = queue.get()
            # await the awaitable with a timeout
            item = await asyncio.wait_for(get_await, 0.5)
        except asyncio.TimeoutError:
            print(f'{time.ctime()} Consumer: gave up waiting...')
            continue
        # check for stop
        if item is None:
            break
        # report
        print(f'{time.ctime()} >got {item}')
    # all done
    print('Consumer: Done')

# entry point coroutine
async def main():
    # create the shared queue
    queue = asyncio.Queue()
    # run the producer and consumers
    await asyncio.gather(producer(queue), consumer(queue))

# start the asyncio program
asyncio.run(main())
```



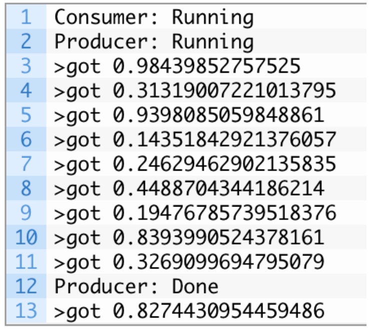
```
Producer: Running
Consumer: Running
Consumer: gave up waiting...
>got 0.8506665865206575
Consumer: gave up waiting...
>got 0.851355213428328
>got 0.3050736798012632
Consumer: gave up waiting...
>got 0.7019959682053681
Consumer: gave up waiting...
>got 0.9753069917130328
Consumer: gave up waiting...
>got 0.7813291071437218
Consumer: gave up waiting...
>got 0.7831885826899522
Consumer: gave up waiting...
>got 0.8001066750131507
Consumer: gave up waiting...
>got 0.9564293628868409
Producer: Done
>got 0.41507431394001704
Consumer: Done
```

Example of Asyncio Queue Join and Task Done

- In the previous examples, we have sent a special message (**None**) into the queue to indicate that all **tasks are done**.
- An alternative approach is to have **coroutines wait** on the **queue directly** and to have the **consumer coroutine** mark **tasks as done**.
- This can be achieved via the **join( )** and **task_done( )** functions on the **asyncio.Queue.**



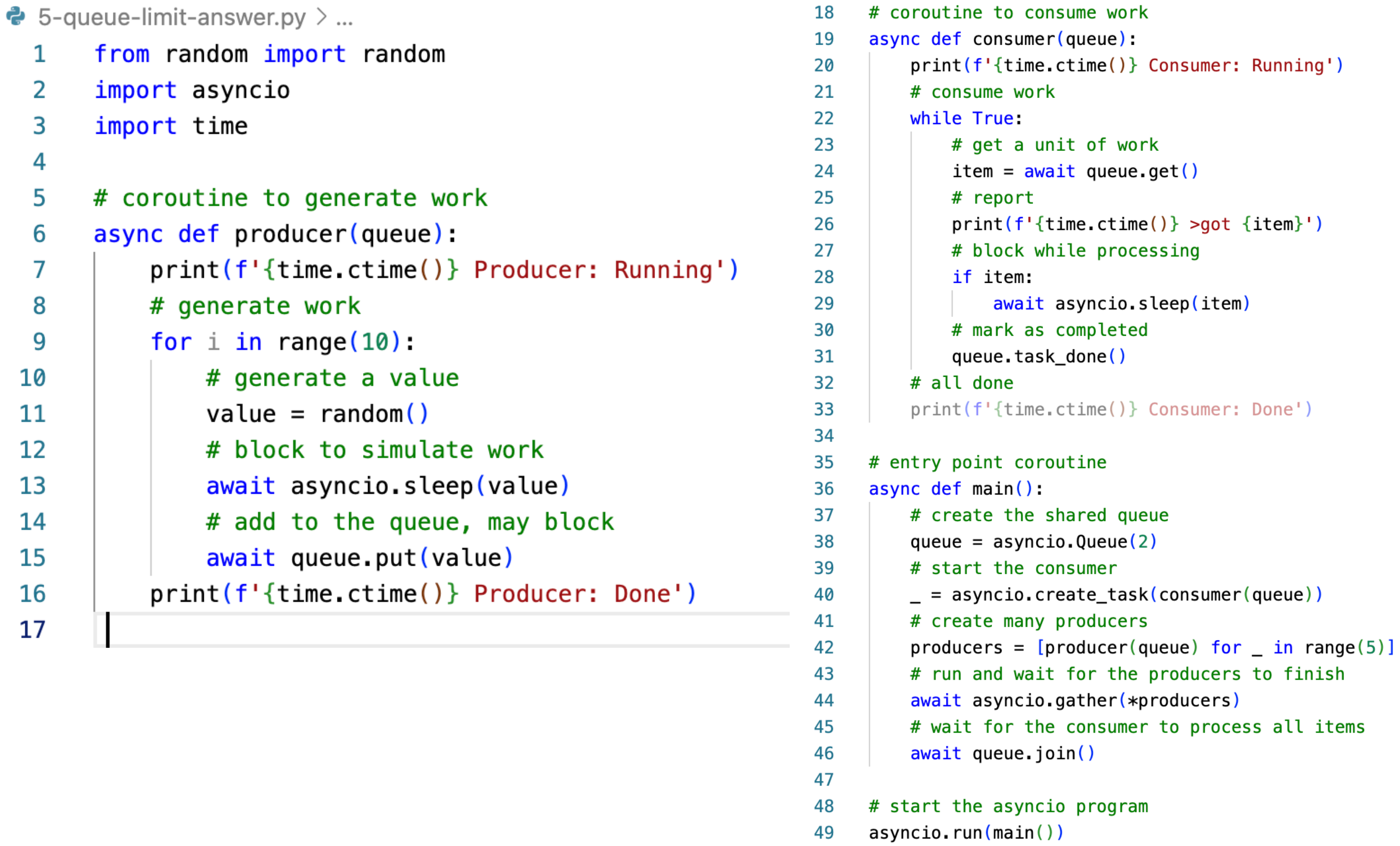
```python
from random import random
import asyncio
import time

# coroutine to generate work
async def producer(queue):
    print(f'{time.ctime()} Producer: Running')
    # generate work
    for i in range(10):
        # generate a value
        value = random()
        # block to simulate work
        await asyncio.sleep(value)
        # add to the queue
        await queue.put(value)
    print(f'{time.ctime()} Producer: Done')

# coroutine to consume work
async def consumer(queue):
    print(f'{time.ctime()} Consumer: Running')
    # consume work
    while True:
        # get a unit of work
        item = await queue.get()
        # report
        print(f'{time.ctime()} >got {item}')
        # block while processing
        if item:
            await asyncio.sleep(item)
        # mark the task as done
        queue.task_done()

# entry point coroutine
async def main():
    # create the shared queue
    queue = asyncio.Queue()
    # start the consumer
    _ = asyncio.create_task(consumer(queue))
    # start the producer and wait for it to finish
    await asyncio.create_task(producer(queue))
    # wait for all items to be processed
    await queue.join()

# start the asyncio program
asyncio.run(main())
```



```
Consumer: Running
Producer: Running
>got 0.98439852757525
>got 0.31319007221013795
>got 0.9398085059848861
>got 0.14351842921376057
>got 0.24629462902135835
>got 0.4488704344186214
>got 0.19476785739518376
>got 0.8393990524378161
>got 0.3269099694795079
Producer: Done
>got 0.8274430954459486
```

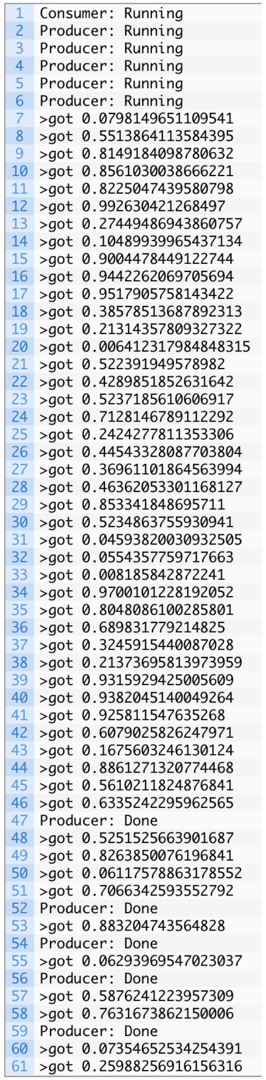
Example of Asyncio Queue With Limited Size

- We can limit the capacity of the queue.
- This can be helpful if we have a **large number of producers** or **slow consumers**. It allows us to **limit** the **number** of **tasks** that may be in memory at any one time, limiting the overall memory usage of the application.
- When the **queue is full**, calls to **put()** will **block until** a position becomes **available** to place another item on the queue.



5-queue-limit-answer.py > ...

```python
from random import random
import asyncio
import time

# coroutine to generate work
async def producer(queue):
    print(f'{time.ctime()} Producer: Running')
    # generate work
    for i in range(10):
        # generate a value
        value = random()
        # block to simulate work
        await asyncio.sleep(value)
        # add to the queue, may block
        await queue.put(value)
    print(f'{time.ctime()} Producer: Done')

# coroutine to consume work
async def consumer(queue):
    print(f'{time.ctime()} Consumer: Running')
    # consume work
    while True:
        # get a unit of work
        item = await queue.get()
        # report
        print(f'{time.ctime()} >got {item}')
        # block while processing
        if item:
            await asyncio.sleep(item)
        # mark as completed
        queue.task_done()
    # all done
    print(f'{time.ctime()} Consumer: Done')

# entry point coroutine
async def main():
    # create the shared queue
    queue = asyncio.Queue(2)
    # start the consumer
    _ = asyncio.create_task(consumer(queue))
    # create many producers
    producers = [producer(queue) for _ in range(5)]
    # run and wait for the producers to finish
    await asyncio.gather(*producers)
    # wait for the consumer to process all items
    await queue.join()

# start the asyncio program
asyncio.run(main())
```



```
Consumer: Running
Producer: Running
Producer: Running
Producer: Running
Producer: Running
Producer: Running
>got 0.0798149651109541
>got 0.5513864113584395
>got 0.8149184098780632
>got 0.8561030038666221
>got 0.8225047439580798
>got 0.992630421268497
>got 0.27449486943860757
>got 0.10489939965437134
>got 0.9004478449122744
>got 0.9442262069705694
>got 0.9517905758143422
>got 0.38578513687892313
>got 0.21314357809327322
>got 0.006412317984848315
>got 0.522391949578982
>got 0.4289851852631642
>got 0.5237185610606917
>got 0.7128146789112292
>got 0.2424277811353306
>got 0.44543328087703804
>got 0.36961101864563994
>got 0.46362053301168127
>got 0.853341848695711
>got 0.5234863755930941
>got 0.04593820030932505
>got 0.0554357759717663
>got 0.008185842872241
>got 0.9700101228192052
>got 0.8048086100285801
>got 0.689831779214825
>got 0.3245915440087028
>got 0.21373695813973959
>got 0.9315929425005609
>got 0.9382045140049264
>got 0.925811547635268
>got 0.6079025826247971
>got 0.1675603246130124
>got 0.8861271320774468
>got 0.5610211824876841
>got 0.6335242295962565
Producer: Done
>got 0.5251525663901687
>got 0.8263850076196841
>got 0.06117578863178552
>got 0.7066342593552792
Producer: Done
>got 0.883204743564828
Producer: Done
>got 0.06293969547023037
Producer: Done
>got 0.5876241223957309
>got 0.7631673862150006
Producer: Done
>got 0.07354652534254391
>got 0.25988256916156316
```