```
import asyncio
import functools
from asyncio import Event


def trigger_event(event: Event):
    event.set()


async def do_work_on_event(event: Event):
    print('Waiting for event...')
    await event.wait()                                      1
    print('Performing work!')
    await asyncio.sleep(1)                                  2
    print('Finished work!')
    event.clear()                                           3


async def main():
    event = asyncio.Event()
    asyncio.get_running_loop().call_later(5.0, functools.par
    await asyncio.gather(do_work_on_event(event), do_work_on


asyncio.run(main())
```

1 Wait until the event occurs.

2 Once the event occurs, wait will no longer block, and we can do work.

❸ Reset the event, so future calls to wait will block.

❹ Trigger the event 5 seconds in the future.

In the preceding listing, we create a coroutine method `do_work_on_event`, this coroutine takes in an event and first calls its `wait` coroutine. This will block until someone calls the event's `set` method to indicate the event has happened. We also create a simple method `trigger_event`, which sets a given event. In our main coroutine, we create an event object and use `call_later` to trigger the event 5 seconds in the future. We then call `do_work_on_event` twice with `gather`, which will create two concurrent tasks for us. We'll see the two `do_work_on_event` tasks idle for 5 seconds until we trigger the event, after which we'll see them do their work, giving us the following output:

```
Waiting for event...
Waiting for event...
Triggering event!
Performing work!
Performing work!
Finished work!
Finished work!
```

This shows us the basics; waiting on an event will block one or more coroutines until we trigger an event, after which they can proceed to do work. Next, let's look at a more real-world example. Imagine we're building an API to accept file uploads from clients. Due to network latency and buffering, a file upload may take some time to complete. With this constraint, we want our API to have a coroutine to block until the file is fully uploaded. Callers of this coroutine can then wait for all the data to come in and do anything they want with it.

We can use an event to accomplish this. We’ll have a coroutine that listens for data from an upload and stores it in an internal buffer. Once we’ve reached the end of the file, we’ll trigger an event indicating the upload is finished. We’ll then have a coroutine method to grab the file contents, which will wait for the event to be set. Once the event is set, we can then return the fully formed uploaded data. Let’s create this API in a class called `FileUpload`:.

Listing 11.11 A file upload API

```
import asyncio
from asyncio import StreamReader, StreamWriter


class FileUpload:
    def __init__(self,
                 reader: StreamReader,
                 writer: StreamWriter):
        self._reader = reader
        self._writer = writer
        self._finished_event = asyncio.Event()
        self._buffer = b''
        self._upload_task = None

    def listen_for_uploads(self):
        self._upload_task = asyncio.create_task(self._accept

    async def _accept_upload(self):
        while data := await self._reader.read(1024):
            self._buffer = self._buffer + data
        self._finished_event.set()
        self._writer.close()
        await self._writer.wait_closed()

    async def get_contents(self):
        await self._finished_event.wait()
        return self._buffer
```

❶ Create a task to listen for the upload and append it to a buffer.

❷ Block until the finished event is set, then return the contents of the buffer.

Now let's create a file upload server to test out this API. Let's say that on every successful upload we want to dump contents to standard out. When a client connects, we'll create a `FileUpload` object and call `listen_for_uploads`. Then, we'll create a separate task that awaits the results of `get_contents`.

Listing 11.12 Using the API in a file upload server

```
import asyncio
from asyncio import StreamReader, StreamWriter
from chapter_11.listing_11_11 import FileUpload


class FileServer:

    def __init__(self, host: str, port: int):
        self.host = host
        self.port = port
        self.upload_event = asyncio.Event()

    async def start_server(self):
        server = await asyncio.start_server(self._client_con
                                            self.host,
                                            self.port)
        await server.serve_forever()

    async def dump_contents_on_complete(self, upload: FileUp
        file_contents = await upload.get_contents()
        print(file_contents)

    def _client_connected(self, reader: StreamReader, writer
        upload = FileUpload(reader, writer)
        upload.listen_for_uploads()
        asyncio.create_task(self.dump_contents_on_complete(u


async def main():
    server = FileServer('127.0.0.1', 9000)
    await server.start_server()


asyncio.run(main())
```

In the preceding listing, we create a `FileServer` class. Each time a client connects to our server we create an instance of the `FileUpload` class that we created in the previous listing, which starts listening for an upload from a connected client. We also concurrently create a task for the `dump_contents_on_complete` coroutine. This calls the `get_contents` coroutine (which will only return once the upload is complete) on the file upload and prints the file to standard out.

We can test this server out by using `netcat`. Pick a file on your filesystem, and run the following command, replacing `file` with the file of your choice:

```
cat file | nc localhost 9000
```

You should then see any file you upload printed to standard out once all the contents have fully uploaded.

One drawback to be aware of with events is that they may fire more frequently than your coroutines can respond to them. Suppose we're using a single event to wake up multiple tasks in a type of producer-consumer workflow. If our all our worker tasks are busy for a long time, the event could run while we're doing work, and we'll never see it. Let's create a dummy example to demonstrate this. We'll create two worker tasks each of which does 5 seconds of work. We'll also create a task that fires an event every second, outpacing the rate that our consumers can handle.

Listing 11.13 A worker falling behind an event

```
import asyncio
from asyncio import Event
from contextlib import suppress


async def trigger_event_periodically(event: Event):
    while True:
        print('Triggering event!')
        event.set()
        await asyncio.sleep(1)
```

```
async def do_work_on_event(event: Event):
    while True:
        print('Waiting for event...')
        await event.wait()
        event.clear()
        print('Performing work!')
        await asyncio.sleep(5)
        print('Finished work!')


async def main():
    event = asyncio.Event()
    trigger = asyncio.wait_for(trigger_event_periodically(ev

    with suppress(asyncio.TimeoutError):
        await asyncio.gather(do_work_on_event(event), do_wor


asyncio.run(main())
```

When we run the preceding listing, we’ll see our event fires and our two workers start their work concurrently. In the meantime, we keep triggering our event. Since our workers are busy, they won’t see that our event fired a second time until they finish their work and call `event.wait()` a second time. If you care about responding every time an event occurs, you’ll need to use a queueing mechanism, which we’ll learn about in the next chapter.

Events are useful for when we want to alert when a specific event happens, but what happens if we need a combination of waiting for an event alongside exclusive access to a shared resource, such as a database connection? Conditions can help us solve these types of workflows.

## onditions

Events are good for simple notifications when something happens, but what about more complex use cases? Imagine needing to gain access to a shared resource that requires a lock on an event, waiting for a more complex set of facts to be true before proceeding or waking up only a certain number of tasks instead of all of them. Conditions can be useful in these types of situations. They are by far the most complex synchronization primitives we've encountered so far, and as such, you likely won't need to use them all that often.

A *condition* combines aspects of a lock and an event into one synchronization primitive, effectively wrapping the behavior of both. We first acquire the condition's lock, giving our coroutine exclusive access to any shared resource, allowing us to safely change any state we need. Then, we wait for a specific event to happen with the `wait` or `wait_for` coroutine. These coroutines release the lock and block until the event happens, and once it does it reacquires the lock giving us exclusive access.

Since this is a bit confusing, let's create a dummy example to understand how to use conditions. We'll create two worker tasks that each attempt to acquire the condition lock and then wait for an event notification. Then, after a few seconds, we'll trigger the condition, which will wake up the two worker tasks and allow them to do work.

Listing 11.14 Condition basics

```
import asyncio
from asyncio import Condition
```

```
async def do_work(condition: Condition):
    while True:
        print('Waiting for condition lock...')
        async with condition:
            print('Acquired lock, releasing and waiting for
            await condition.wait()
            print('Condition event fired, re-acquiring lock
            await asyncio.sleep(1)
        print('Work finished, lock released.')

async def fire_event(condition: Condition):
    while True:
        await asyncio.sleep(5)
        print('About to notify, acquiring condition lock...'
        async with condition:
            print('Lock acquired, notifying all workers.')
            condition.notify_all()
        print('Notification finished, releasing lock.')


async def main():
    condition = Condition()

    asyncio.create_task(fire_event(condition))
    await asyncio.gather(do_work(condition), do_work(conditi


asyncio.run(main())
```

❶ Wait to acquire the condition lock; once acquired, release the lock.

❷ Wait for the event to fire; once it does, reacquire the condition lock.

❸ Once we exit the async with block, release the condition lock.

❹ Notify all tasks that the event has happened.

In the preceding listing, we create two coroutine methods: `do_work` and `fire_event`. The `do_work` method acquires the condition, which is analogous to acquiring a lock, and then calls the condition's `wait` coroutine method. The `wait` coroutine method will block until someone calls the condition's `notify_all` method.

The `fire_event` coroutine method sleeps for a little bit and then acquires the condition and calls the `notify_all` method, which will wake up any tasks that are currently waiting on the condition. Then, in our main coroutine we create one `fire_event` task and two `do_work` tasks and run them concurrently. When running this you'll see the following repeated if the application runs:

```
Worker 1: waiting for condition lock...
Worker 1: acquired lock, releasing and waiting for condition
Worker 2: waiting for condition lock...
Worker 2: acquired lock, releasing and waiting for condition
fire_event: about to notify, acquiring condition lock...
fire_event: Lock acquired, notifying all workers.
fire_event: Notification finished, releasing lock.
Worker 1: condition event fired, re-acquiring lock and doing
Worker 1: Work finished, lock released.
Worker 1: waiting for condition lock...
Worker 2: condition event fired, re-acquiring lock and doing
Worker 2: Work finished, lock released.
Worker 2: waiting for condition lock...
Worker 1: acquired lock, releasing and waiting for condition
Worker 2: acquired lock, releasing and waiting for condition
```

You'll notice that the two workers start right away and block waiting for the `fire_event` coroutine to call `notify_all`. Once `fire_event` calls `notify_all`, the worker tasks wake up and then proceed to do their work.

Conditions have an additional coroutine method called `wait_for`. Instead of blocking until someone notifies the condition, `wait_for` accepts a predicate (a no-argument function that returns a `Boolean`) and will block until that predicate returns `True`. This proves useful when there is a shared resource with some coroutines dependent on certain states becoming true.

As an example, let's pretend we're creating a class to wrap a database connection and run queries. We first have an underlying connection that can't run multiple queries at the same time, and the database connection may not be initialized before someone tries to run a query. The combination of a shared resource and an event we need to block gives us the right conditions to use a `Condition`. Let's simulate this with a mock database connection class. This class will run queries but will only do so once we've properly initialized a connection. We'll then use this mock connection class to try to run two queries concurrently before we've finished initializing the connection.

Listing 11.15 Using conditions to wait for specific states

```
import asyncio
from enum import Enum
```

```
class ConnectionState(Enum):
    WAIT_INIT = 0
    INITIALIZING = 1
    INITIALIZED = 2


class Connection:

    def __init__(self):
        self._state = ConnectionState.WAIT_INIT
        self._condition = asyncio.Condition()

    async def initialize(self):
        await self._change_state(ConnectionState.INITIALIZIN
        print('initialize: Initializing connection...')
        await asyncio.sleep(3)  # simulate connection startu
        print('initialize: Finished initializing connection'
        await self._change_state(ConnectionState.INITIALIZED

    async def execute(self, query: str):
        async with self._condition:
            print('execute: Waiting for connection to initia
            await self._condition.wait_for(self._is_initiali
            print(f'execute: Running {query}!!!')
            await asyncio.sleep(3)  # simulate a long query

    async def _change_state(self, state: ConnectionState):
        async with self._condition:
            print(f'change_state: State changing from {self.
            self._state = state
            self._condition.notify_all()

    def _is_initialized(self):
        if self._state is not ConnectionState.INITIALIZED:
            print(f'_is_initialized: Connection not finished
            return False
        print(f'_is_initialized: Connection is initialized!'
        return True
async def main():
    connection = Connection()
    query_one = asyncio.create_task(connection.execute('sele
    query_two = asyncio.create_task(connection.execute('sele
    asyncio.create_task(connection.initialize())
    await query_one
    await query_two


asyncio.run(main())
```

In the preceding listing, we create a connection class that contains a condition object and keeps track of an internal state that we initialize to `WAIT_INIT`, indicating we're waiting for initialization to happen. We also create a few methods on the `Connection` class. The first is `initialize`, which simulates creating a database connection. This method calls the _ `change_state` method to set the state to `INITIALIZING` when first called and then once the connection is initialized, it sets the state to `INITIALIZED`. Inside the _ `change_state` method, we set the internal state and then call the conditions `notify_` `all` method. This will wake up any tasks that are waiting for the condition.

In our `execute` method, we acquire the condition object in an `async` `with` block and then we call `wait_for` with a predicate that checks to see if the state is `INITIALIZED`. This will block until our database connection is fully initialized, preventing us from accidently issuing a query before the connection exists. Then, in our main coroutine, we create a connection class and create two tasks to run queries, followed by one task to initialize the connection. Running this code, you'll see the following output, indicating that our queries properly wait for the initialization task to finish before running the queries:

```
execute: Waiting for connection to initialize
_is_initialized: Connection not finished initializing, state
execute: Waiting for connection to initialize
_is_initialized: Connection not finished initializing, state
change_state: State changing from ConnectionState.WAIT_INIT
initialize: Initializing connection...
_is_initialized: Connection not finished initializing, state
_is_initialized: Connection not finished initializing, state
initialize: Finished initializing connection
change_state: State changing from ConnectionState.INITIALIZI
_is_initialized: Connection is initialized!
execute: Running select * from table!!!
_is_initialized: Connection is initialized!
execute: Running select * from other_table!!!
```

Conditions are useful in scenarios in which we need access to a shared resource and there are states that we need to be notified about before doing work. This is a somewhat complicated use case, and as such, you won't likely come across or need conditions in asyncio code.

## ary

- We’ve learned about single-threaded concurrency bugs and how they differ from concurrency bugs in multithreading and multiprocessing.
- We know how to use asyncio locks to prevent concurrency bugs and synchronize coroutines. This happens less often due to asyncio’s single-threaded nature, they can sometimes be needed when shared state could change during an `await`.
- We’ve learned how to use semaphores to control access to finite resources and limit concurrency, which can be useful in traffic-shaping.
- We know how to use events to trigger actions when something happens, such as initialization or waking up worker tasks.
- We know how to use conditions to wait for an action and, because of an action, gain access to a shared resource.

# 12 Asynchronous queues

This chapter covers

- Asynchronous queues
- Using queues for producer-consumer workflows
- Using queues with web applications
- Asynchronous priority queues
- Asynchronous LIFO queues

When designing applications to process events or other types of data, we often need a mechanism to store these events and distribute them to a set of workers. These workers can then do whatever we need to do based on these events concurrently, yielding time savings as opposed to processing events sequentially. asyncio provides an asynchronous queue implementation that lets us do this. We can add pieces of data into a queue and have several workers running concurrently, pulling data from the queue and processing it as it becomes available.

These are commonly referred to as *producer-consumer workflows*. Something produces data or events that we need to handle; processing these work items could take a long time. Queues can also help us transmit long-running tasks while keeping a responsive user interface. We put an item on the queue for later processing and inform the user that we’ve started this work in the background. Asynchronous queues also have an added benefit of providing a mechanism to limit concurrency, as each queue generally permits a finite amount of worker tasks. This can be used in cases in which we need to limit concurrency in a similar way to what we saw with semaphores in chapter 11.

In this chapter, we'll learn how to use asyncio queues to handle producer-consumer workflows. We'll master the basics first by building an example grocery store queue with cashiers as our consumers. We'll then apply this to an order management web API, demonstrating how to respond quickly to users while letting the queue process work in the background. We'll also learn how to process tasks in priority order, which is useful when one task is more important to process first, despite being put in the queue later. Finally, we'll look at LIFO (last in, first out) queues and understand the drawbacks of asynchronous queues.

## synchronous queue basics

Queues are a type of FIFO data structure. In other words, the first element in a queue is the first element to leave the queue when we ask for the next element. They're not much different from the queue you're a part of when checking out in a grocery store. You join the line at the end and wait for the cashier to check out anyone in front of you. Once they've checked someone out, you move up in the queue while someone who joins after you waits behind you. Then, when you're first in the queue you check out and leave the queue entirely.

The checkout queue as we have described it is a synchronous workflow. One cashier checks out one customer at a time. What if we reimagined the queue to better take advantage of concurrency and perform more like a supermarket checkout? Instead of one cashier, there would be multiple cashiers and a single queue. Whenever a cashier is available, they can flag down the next person to the checkout counter. This means there are multiple cashiers directing customers from the queue concurrently in addition to multiple cashiers concurrently checking out customers.

This is the core of what asynchronous queues let us do. We add multiple work items waiting to be processed into the queue. We then have multiple workers pull items from the queue when they are available to perform a task.

Let’s explore this by building our supermarket example. We’ll think of our worker tasks as cashiers, and our “work items” will be customers to check out. We’ll implement customers with individual lists of products that the cashier needs to scan. Some items take longer than others to scan; for instance, bananas must be weighed and have their SKU code entered. Alcoholic beverages require a manager to check the customer’s ID.

For our supermarket checkout scenario, we’ll implement a few data classes to represent products with integers used to represent the time (in seconds) they take for a cashier to check out. We’ll also build a customer class that has a random set of products they’d like to buy. Then, we’ll put these customers in an asyncio queue to represent our checkout line. We’ll also create several worker tasks to represent our cashiers. These tasks will pull customers from the queue, looping through all their products and sleeping for the time needed to check out their items to simulate the checkout process.

Listing 12.1 A supermarket checkout queue

```
import asyncio
from asyncio import Queue
from random import randrange
from typing import List


class Product:
    def __init__(self, name: str, checkout_time: float):
        self.name = name
        self.checkout_time = checkout_time


class Customer:
    def __init__(self, customer_id: int, products: List[Prod
        self.customer_id = customer_id
        self.products = products


async def checkout_customer(queue: Queue, cashier_number: in
    while not queue.empty():
        customer: Customer = queue.get_nowait()
        print(f'Cashier {cashier_number} '
              f'checking out customer '
              f'{customer.customer_id}')
        for product in customer.products:
            print(f"Cashier {cashier_number} "
                  f"checking out customer "
                  f"{customer.customer_id}'s {product.name}"
            await asyncio.sleep(product.checkout_time)
        print(f'Cashier {cashier_number} '
              f'finished checking out customer '
              f'{customer.customer_id}')
        queue.task_done()


async def main():
    customer_queue = Queue()

    all_products = [Product('beer', 2),
                      Product('bananas', .5),
                      Product('sausage', .2),
                      Product('diapers', .2)]

    for i in range(10):
        products = [all_products[randrange(len(all_products)
                      for _ in range(randrange(10))]
        customer_queue.put_nowait(Customer(i, products))
    cashiers = [asyncio.create_task(checkout_customer(custom
                  for i in range(3)]
    await asyncio.gather(customer_queue.join(), *cashiers)

asyncio.run(main())
```

❶ Keep checking out customers if there are any in the queue.

❷ Check out each customer's product.

❸ Create 10 customers with random products.

❹ Create three "cashiers" or worker tasks to check out customers.

In the preceding listing, we create two data classes: one for a product and one for a supermarket customer. A product consists of a product name and the amount of time (in seconds) it takes for a cashier to enter that item in the register. A customer has a number of products they are bringing to the cashier to buy. We also define a `checkout_` `customer` coroutine function, which does the work of checking out a customer. While our queue has customers in it, it pulls a customer from the front of the queue with `queue.get_nowait()` and simulates the time to scan a product with `asyncio.sleep`. Once a customer is checked out, we call `queue.task_done`. This signals to the queue that our worker has finished its current work item. Internally within the `Queue` class, when we get an item from the queue a counter is incremented by one to track the number of unfinished tasks remain. When we call `task_done`, we tell the queue that we've finished, and it decrements this count by one (why we need to do this will make sense shortly, when we talk about `join`).

In our main coroutine function, we create a list of available products and generate 10 customers, each with random products. We also create three worker tasks for the `checkout_customer` coroutine that are stored in a list called `cashiers`, which is analogous to three human cashiers working at our imaginary supermarket. Finally, we wait for the cashier `checkout_customer` tasks to finish alongside the `customer_queue.join()` coroutine using `gather`. We use `gather` so that any exceptions from our cashier tasks will rise up to our main coroutine function. The `join` coroutine blocks until the queue is empty and all customers have been checked out. The queue is considered empty when the internal counter of pending work items reaches zero. Therefore, it is important to call `task_done` in your workers. If you don't do this, the `join` coroutine may receive an incorrect view of the queue and may never terminate.

While the customer's items are randomly generated, you should see output similar to the following, showing that each worker task (cashier) is concurrently checking out customers from the queue:

```
Cashier 0 checking out customer 0
Cashier 0 checking out customer 0's sausage
Cashier 1 checking out customer 1
Cashier 1 checking out customer 1's beer
Cashier 2 checking out customer 2
Cashier 2 checking out customer 2's bananas
Cashier 0 checking out customer 0's bananas
Cashier 2 checking out customer 2's sausage
Cashier 0 checking out customer 0's sausage
Cashier 2 checking out customer 2's bananas
Cashier 0 finished checking out customer 0
Cashier 0 checking out customer 3
```

Our three cashiers start checking out customers from the queue concurrently. Once they’ve finished checking out one customer, they pull another from the queue until the queue is empty.

You may notice that our methods for putting items into the queue and retrieving them are oddly named: `get_nowait` and `put_nowait`. Why is there a `nowait` at the end of each of these methods? There are two ways of getting and retrieving an item from a queue: one that is a coroutine and blocks, and one that is nonblocking and is a regular method. The `get_nowait` and `put_nowait` variants instantly perform the non-blocking method calls and return. Why would we need a blocking queue insertion or retrieval?

The answer lies in how we want to handle the upper and lower bounds of our queue. This describes happens when there are too many items in the queue (the upper bound) and what happens when there are no items in the queue (the lower bound).

Going back to our supermarket queue example, let’s address two things that aren’t quite real-world about it, using the coroutine versions of `get` and `put`.

- It is unlikely we’ll just have one line of 10 customers who all show up at the same time, and once the line is empty the cashiers stop working altogether.

- Our customer queue probably shouldn't be unbounded; say, the latest desirable gaming console just came out, and you're the only store in town to carry it. Naturally, mass hysteria has ensued, and your store is flooded with customers. We probably couldn't fit 5,000 customers in the store, so we need a way to turn them away or make them wait outside.

For the first issue, let's say we wanted to refactor our application so that we randomly generate some customers every few seconds to simulate a realistic supermarket queue. In our current implementation of `checkout_customer`, we loop while the queue is not empty and grab a customer with `get_nowait`. Since our queue could be empty, we can't loop on `not queue.empty`, since our cashiers will be available even if no one is in line, so we'll need a `while True` in our worker coroutine. So what happens in this case when we call `get_nowait` and the queue is empty? This is easy to test out in a few lines of code; we just create an empty queue and call the method in question:

```
import asyncio
from asyncio import Queue


async def main():
    customer_queue = Queue()
    customer_queue.get_nowait()

asyncio.run(main())
```

Our method will throw an `asyncio.queues.QueueEmpty` exception. While we could wrap this in a `try catch` and ignore this exception, this wouldn't quite work, as whenever the queue is empty, we've made our worker task CPU-bound, spinning and catching exceptions. In this case, we can use the `get` coroutine method. This will block (in a non-CPU-bound fashion) until an item is in the queue to process and won't throw an exception. This is the equivalent of the worker tasks idling, standing by for some customer to come into the queue giving them work to do at the checkout counter.

To address our second issue of thousands of customers trying to get in line concurrently, we need to think about the bounds of our queue. By default, queues are unbounded, and they can grow to store an infinite amount of work items. In theory this is acceptable, but in the real world, systems have memory constraints, so placing an upper bound on our queue to prevent running out of memory is a good idea. In this case, we need to think through what we want our behavior to be when our queue is full. Let's see what happens when we create a queue that can only hold one item and try to add a second with `put_nowait`:

```
import asyncio
from asyncio import Queue



async def main():
    queue = Queue(maxsize=1)

    queue.put_nowait(1)
    queue.put_nowait(2)

asyncio.run(main())
```

In this case, much like `get_nowait`, `put_nowait` throws an exception of the type `asyncio.queues.QueueFull`. Like `get`, there is also a coroutine method called `put`. This method will block until there is room in the queue. With this in mind, let’s refactor our customer example to use the coroutine variants of `get` and `put`.

Listing 12.2 Using coroutine queue methods

```
import asyncio
from asyncio import Queue
from random import randrange


class Product:
    def __init__(self, name: str, checkout_time: float):
        self.name = name
        self.checkout_time = checkout_time


class Customer:
    def __init__(self, customer_id, products):
        self.customer_id = customer_id
        self.products = products


async def checkout_customer(queue: Queue, cashier_number: in
    while True:
        customer: Customer = await queue.get()
        print(f'Cashier {cashier_number} '
              f'checking out customer '
              f'{customer.customer_id}')
        for product in customer.products:
            print(f"Cashier {cashier_number} "
                  f"checking out customer "
                  f"{customer.customer_id}'s {product.name}"
            await asyncio.sleep(product.checkout_time)
        print(f'Cashier {cashier_number} '
              f'finished checking out customer '
              f'{customer.customer_id}')
        queue.task_done()


def generate_customer(customer_id: int) -> Customer:    ❶
    all_products = [Product('beer', 2),
                    Product('bananas', .5),
                    Product('sausage', .2),
                    Product('diapers', .2)]
    products = [all_products[randrange(len(all_products))]
                for _ in range(randrange(10))]
    return Customer(customer_id, products)


async def customer_generator(queue: Queue):             ❷
    customer_count = 0

    while True:
        customers = [generate_customer(i)
                     for i in range(customer_count,
                     customer_count + randrange(5))]
        for customer in customers:
            print('Waiting to put customer in line...')
            await queue.put(customer)
            print('Customer put in line!')
        customer_count = customer_count + len(customers)
        await asyncio.sleep(1)
```

```
async def main():
    customer_queue = Queue(5)

    customer_producer = asyncio.create_task(customer_generat

    cashiers = [asyncio.create_task(checkout_customer(custom
                for i in range(3)]

    await asyncio.gather(customer_producer, *cashiers)


asyncio.run(main())
```

❶ Generate a random customer.

❷ Generate several random customers every second.

In the preceding listing, we create a `generate_customer` coroutine that creates a customer with a random list of products. Alongside this we create a `customer_generator` coroutine function that generates between one and five random customers every second and adds them to the queue with `put`. Because we use the coroutine `put`, if our queue is full, `customer_generator` will block until the queue has free spaces. Specifically, this means that if there are five customers in the queue and the *producer* tries to add a sixth, the queue will block, allowing that customer into the queue until there is a space freed up by a cashier checking someone out. We can think of `customer_` `generator` as our *producer*, as it produces customers for our cashiers to check out.

We also refactor `checkout_customer` to run forever, since our cashiers remain on call when the queue is empty. We then refactor `checkout_customer` to use the queue `get` coroutine, and the coroutine will block if the queue has no customers in it. Then, in our main coroutine we create a queue that allows five customers in line at a time and create three `checkout_customer` tasks running concurrently. We can think of the cashiers as our *consumers*; they consume customers to check out from the queue.

This code randomly generates customers, but at some point, the queue should fill up such that the cashiers aren't processing customers as fast as the producer is creating them. Thus, we'll see output similar to the following where the producer waits to add a customer into the line until a customer has finished checking out:

```
Waiting to put customer in line...
Cashier 1 checking out customer 7's sausage
Cashier 1 checking out customer 7's diapers
Cashier 1 checking out customer 7's diapers
Cashier 2 finished checking out customer 5
Cashier 2 checking out customer 9
Cashier 2 checking out customer 9's bananas
Customer put in line!
```

We now understand the basics of how asynchronous queues work, but since we're usually not building supermarket simulations in our day jobs, let's look at a few real-world scenarios to see how we would apply this in applications we really might build.

## Queues in web applications

Queues can be useful in web applications when we have a potentially time-consuming operation that we can run in the background. If we ran this operation in the main coroutine of the web request, we would block the response to the user until the operation finished, potentially leaving the end user with a slow, unresponsive page.

Imagine we're part of an e-commerce organization, and we're operating with a slow order management system. Processing an order can take several seconds, but we don't want to keep the user waiting for a response that their order has been placed. Furthermore, the order management system does not handle load well, so we'd like to limit how many requests we make to it concurrently. In this circumstance a queue can solve both problems. As we saw before, a queue can have a maximum number of elements we allow before adding more either blocks or throws an exception. This means if we have a queue with an upper limit, we'll at most have however many consumer tasks we create that are running concurrently. This provides a natural limit to concurrency.

A queue also solves the issue of the user waiting too long for a response. Putting an element on the queue happens instantly, meaning we can notify the user that their order has been placed right away, providing a fast user experience. In the real world, of course, this opens up the potential for the background task to fail without the user being notified, so you'll need some form of data persistence and logic to combat this.

To try this out, let's create a simple web application with aiohttp that employs a queue to run background tasks. We'll simulate interacting with a slow order management system by using `asyncio.sleep`. In a real world microservice architecture you'd likely be communicating over REST with aiohttp or a similar library, but we’ll use `sleep` for simplicity.

We’ll create an aiohttp startup hook to create our queue as well as a set of worker tasks that will interact with the slow service. We’ll also create a `HTTP` `POST` endpoint/ order that will place an order on the queue (here, we’ll just generate a random number for our worker task to `sleep` to simulate the slow service). Once the order is put on the queue, we’ll return a `HTTP` `200` and a message indicating the order has been placed.

We’ll also add some graceful shutdown logic in an aiohttp shutdown hook, since if our application shuts down, we might still have some orders being processed. In the shutdown hook, we’ll wait until any workers that are busy have finished.

Listing 12.3 Queues with a web application

```
import asyncio
from asyncio import Queue, Task
from typing import List
from random import randrange
from aiohttp import web
from aiohttp.web_app import Application
from aiohttp.web_request import Request
from aiohttp.web_response import Response

routes = web.RouteTableDef()

QUEUE_KEY = 'order_queue'
TASKS_KEY = 'order_tasks'
```

```
async def process_order_worker(worker_id: int, queue: Queue)
    while True:
        print(f'Worker {worker_id}: Waiting for an order...'
        order = await queue.get()
        print(f'Worker {worker_id}: Processing order {order}
        await asyncio.sleep(order)
        print(f'Worker {worker_id}: Processed order {order}'
        queue.task_done()


@routes.post('/order')
async def place_order(request: Request) -> Response:
    order_queue = app[QUEUE_KEY]
    await order_queue.put(randrange(5))
    return Response(body='Order placed!')
async def create_order_queue(app: Application):
    print('Creating order queue and tasks.')
    queue: Queue = asyncio.Queue(10)
    app[QUEUE_KEY] = queue
    app[TASKS_KEY] = [asyncio.create_task(process_order_work
                      for i in range(5)]


async def destroy_queue(app: Application):
    order_tasks: List[Task] = app[TASKS_KEY]
    queue: Queue = app[QUEUE_KEY]
    print('Waiting for pending queue workers to finish....')
    try:
        await asyncio.wait_for(queue.join(), timeout=10)
    finally:
        print('Finished all pending items, canceling worker
        [task.cancel() for task in order_tasks]


app = web.Application()
app.on_startup.append(create_order_queue)
app.on_shutdown.append(destroy_queue)

app.add_routes(routes)
web.run_app(app)
```

❶ Grab an order from the queue, and process it.

❷ Put the order on the queue, and respond to the user immediately.

❸ Create a queue with a maximum of 10 elements, and create 5 worker tasks.

❹ Wait for any busy tasks to finish.

In the preceding listing, we first create a `process_order_worker` coroutine. This pulls an item from the queue, in this case an integer, and sleeps for that amount of time to simulate working with a slow order management system. This coroutine loops forever, continually pulling items from the queue and processing them.

We then create the coroutines to set up and tear down the queue, `create_order_queue` and `destroy_order_queue`, respectively. Creating the queue is straightforward, as we create an asyncio queue with a maximum of 10 elements and we create five worker tasks, storing them in our `Application` instance.

Destroying the queue is a bit more involved. We first wait for the queue to finish processing all its elements with `Queue.join`. Since our application is shutting down, it won't be serving any more HTTP requests, so no other orders can go into our queue. This means that anything already in the queue will be processed by a worker, and anything a worker is currently processing will finish as well. We also wrap `join` in a `wait_ for` with a timeout of 10 seconds as well. This is a good idea because we don't want a runaway task taking a long time preventing our application from shutting down.

Finally, we define our application `route`. We create a POST endpoint at `/order`. This endpoint creates a random delay and adds it to the queue. Once we've added the order to the queue, we respond to the user with a HTTP 200 status code and a short message. Note that we used the coroutine variant of `put`, which means that if our queue is full the request will block until the message is on the queue, which could take time. You may want to use the `put_nowait` variant and then respond with a HTTP 500 error or other error code asking the caller to try again later. Here, we've made a tradeoff of a request potentially taking some time so that our order always goes on the queue. Your application may require "fail fast" behavior, so responding with an error when the queue is full may the correct behavior for your use case.

Using this queue, our order endpoint will respond nearly instantly when our order was placed so long as the queue isn't full. This provides the end user with a quick and smooth ordering experience—one that hopefully keeps them coming back to buy more.

One thing to keep in mind when using asyncio queues in web applications is the failure modes of queues. What if one of our API instances crashed for some reason, such as running out of memory, or if we needed to restart the server for a redeploy of our application? In this case, we would lose any unprocessed orders that are in the queue, as they are only stored in memory. Sometimes, losing an item in a queue isn't a big deal, but in the case of a customer order, it probably is.

asyncio queues provide no out-of-the-box concept of task persistence or queue durability. If we want tasks in our queue to be robust against these types of failures, we need to introduce somewhere a method to save our tasks, such as a database. More correctly, however, is using a separate queue outside of asyncio that supports task persistence. Celery and RabbitMQ are two examples of task queues that can persist to disk.

Of course, using a separate architectural queue comes with added complexity. In the case of durable queues with persistent tasks, it also comes with a performance challenge of needing to persist to disk. To determine the best architecture for your application, you'll need to carefully weigh the tradeoffs of an in-memory-only asyncio queue versus a separate architectural component.

## A web crawler queue

Consumer tasks can also be producers if our consumer generates more work to put in the queue. Take for instance a web crawler that visits all links on a particular page. You can imagine one worker downloading and scanning a page for links. Once the worker has found links it can add them to a queue. This lets other available workers pull links onto the queue and visit them concurrently, adding any links they encounter back to the queue.

Let's build a crawler that does this. We'll create an unbounded queue (you may want to bound it if you're concerned about memory overruns) that will hold URLs to download. Then, our workers will pull URLs off the queue and use aiohttp to download them. Once we've downloaded them, we'll use a popular HTML parser, Beautiful Soup, to extract links to put back into the queue.

At least with this application, we don't want to scan the entire internet, so we'll only scan a set number of pages away from the root page. We'll call this our "maximum depth"; if our maximum depth is set to three, it means we'll only follow links three pages away from the root.

To get started, let's install Beautiful Soup version 4.9.3 with the following command:

```
pip install -Iv beautifulsoup4==4.9.3
```

We'll assume some knowledge of Beautiful Soup. You can read more in the documentation at https://www.crummy.com/software/BeautifulSoup/bs4/doc.

Our plan will be to create a worker coroutine that will pull a page from the queue and download it with aiohttp. Once we've done this, we'll use Beautiful Soup to get all the links of the form `<a` `href="url">` from the page, adding them back to the queue.

Listing 12.4 A queue-based crawler

```
import asyncio
import aiohttp
import logging
from asyncio import Queue
from aiohttp import ClientSession
from bs4 import BeautifulSoup
```

```
class WorkItem:
    def __init__(self, item_depth: int, url: str):
        self.item_depth = item_depth
        self.url = url


async def worker(worker_id: int, queue: Queue, session: Clie
    print(f'Worker {worker_id}')
    while True:
        work_item: WorkItem = await queue.get()
        print(f'Worker {worker_id}: Processing {work_item.ur
        await process_page(work_item, queue, session, max_de
        print(f'Worker {worker_id}: Finished {work_item.url}
        queue.task_done()


async def process_page(work_item: WorkItem, queue: Queue, se
    try:
        response = await asyncio.wait_for(session.get(work_i
        if work_item.item_depth == max_depth:
            print(f'Max depth reached, '
                  f'for {work_item.url}')
        else:
            body = await response.text()
            soup = BeautifulSoup(body, 'html.parser')
            links = soup.find_all('a', href=True)
            for link in links:
                queue.put_nowait(WorkItem(work_item.item_dep
                                          link['href']))
    except Exception as e:
        logging.exception(f'Error processing url {work_item.
async def main():
    start_url = 'http:/ /example.com'
    url_queue = Queue()
    url_queue.put_nowait(WorkItem(0, start_url))
    async with aiohttp.ClientSession() as session:
        workers = [asyncio.create_task(worker(i, url_queue,
                   for i in range(100)]
        await url_queue.join()
        [w.cancel() for w in workers]


asyncio.run(main())
```

❶ Grab a URL from the queue to process and then begin to download it.

❷ Download the URL contents, and parse all links from the page, putting them back on the queue.

❸ Create a queue and 100 worker tasks to process URLs.

In the preceding listing, we first define a `WorkItem` class. This is a simple data class to hold a URL and the depth of that URL. We then define our worker, which pulls a `WorkItem` from the queue and calls `process_page`. The `process_page` coroutine function downloads the contents of the URL if it can do so (a timeout or exception could occur, which we just log and ignore). It then uses Beautiful Soup to get all the links and adds them back to the queue for other workers to process.

In our main coroutine, we create the queue and bootstrap it with our first `WorkItem`. In this example we hardcode example.com, and since it is our root page, its depth is

0. We then create an aiohttp session and create 100 workers, meaning we can download 100 URLs concurrently, and we set its max depth to 3. We then wait for the queue to empty and all workers to finish with `Queue.join`. Once the queue is finished processing, we cancel all our worker tasks. When you run this code, you should see 100 worker tasks fire up and start looking for links from each URL it downloads, giving you output like the following:

```
Found 1 links from http:/ / example .com
Worker 0: Finished http:/ / example .com
Worker 0: Processing https:/ /www .iana.org/domains/example
Found 68 links from https:/ /www .iana.org/domains/example
Worker 0: Finished https:/ /www .iana.org/domains/example
Worker 0: Processing /
Worker 2: Processing /domains
Worker 3: Processing /numbers
Worker 4: Processing /protocols
Worker 5: Processing /about
Worker 6: Processing /go/rfc2606
Worker 7: Processing /go/rfc6761
Worker 8: Processing http:/ /www .icann.org/topics/idn/
Worker 9: Processing http:/ /www .icann.org/
```

The workers will continue to download pages and process links, adding them to the queue until we reach the maximum depth we've specified.

We've now seen the basics of asynchronous queues by building a fake supermarket checkout line as well as by building an order management API and a web crawler. So far, our workers have given equal weight to each element in the queue, and they just pull whoever is at the front of the line out to work on. What if we wanted some tasks to happen sooner even if they're toward the back of the queue? Let's take a look at priority queues to see how to do this.

## riority queues

Our previous examples of queues processed items in FIFO, or first-in, first-out, ordering. Whoever was first in line gets processed first. This works well in many cases, both in software engineering and in life.

In certain applications, however, having all tasks be considered equal is not always desirable. Imagine we're building a data processing pipeline where each task is a long-running query that can take several minutes. Let's say two tasks come in at roughly the same time. The first task is a low priority data query, but the second is a mission-critical data update that should be processed as soon as possible. With simple queues, the first task will be processed, leaving the second, more important task waiting for the first one to finish. Imagine the first task takes hours, or if all our workers are busy, our second task could be waiting for a long time.

We can use a priority queue to solve this problem and make our workers work on our most important tasks first. Internally, priority queues are backed by *heaps* (using the `heapq` module) instead of Python lists like simple queues. To create an asyncio priority queue, we create an instance of `asyncio.PriorityQueue`.

We won't get too much into data structure specifics here, but a heap is a binary tree with the property that every parent node has a value less than all its children (see figure 12.1). This is unlike binary search trees typically used in sorting and searching problems where the only property is that a node's left-hand child is smaller than its parent and the node's right-hand child is larger. The property of heaps we take advantage of is that the topmost node is always the smallest element in the tree. If we always make the smallest node our highest priority one, then the high priority node will always be the first in the queue.

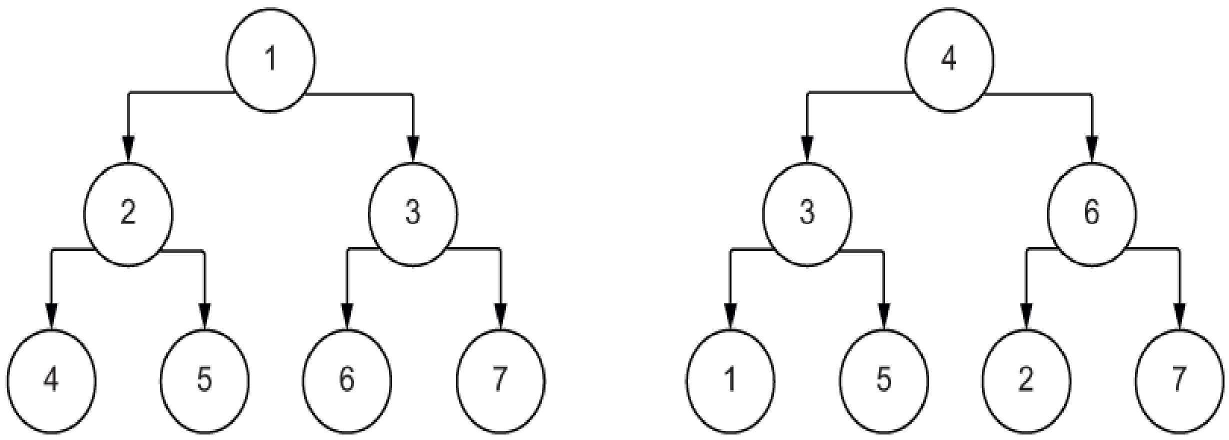

Figure 12.1 On the left, a binary tree that satisfies the heap property; on the right, a binary search tree that does not satisfy the heap property

It is unlikely the work items we put in our queue will be plain integers, so we'll need some way to construct a work item with a sensible priority rule. One way to do this is with a tuple, where the first element is an integer representing the priority and the second is any task data. The default queue implementation looks to the first value of the tuple to decide priority with the lowest numbers having the highest priority. Let's look at an example with tuples as work items to see the basics of how a priority queue works.

Listing 12.5 Priority queues with tuples

```
import asyncio
from asyncio import Queue, PriorityQueue
from typing import Tuple
```

```
async def worker(queue: Queue):
    while not queue.empty():
        work_item: Tuple[int, str] = await queue.get()
        print(f'Processing work item {work_item}')
        queue.task_done()


async def main():
    priority_queue = PriorityQueue()

    work_items = [(3, 'Lowest priority'),
                  (2, 'Medium priority'),
                  (1, 'High priority')]

    worker_task = asyncio.create_task(worker(priority_queue)

    for work in work_items:
        priority_queue.put_nowait(work)

    await asyncio.gather(priority_queue.join(), worker_task)


asyncio.run(main())
```

In the preceding listing, we create three work items: one with high priority, one with medium priority, and one with low priority. We then add them in the priority queue in reverse priority order, meaning we insert the lowest priority item first and the highest last. In a normal queue, this would mean we'd process the lowest priority item first, but if we run this code, we'll see the following output:

```
Processing work item (1, 'High priority')
Processing work item (2, 'Medium priority')
Processing work item (3, 'Lowest priority')
```

This indicates that we processed the work items in the order of their priority, not how they were inserted into the queue. Tuples work for simple cases, but if we have a lot of data in our work items, a tuple could get messy and confusing. Is there a way for us to create a class of some sort that will work the way we want with heaps? We can, in fact, and the tersest way to do this is by using a data class (we could also implement the proper *dunder* methods `__lt__`, `__le__`, `__gt__`, and `__ge__` if data classes aren't an option).

Listing 12.6 Priority queues with data classes

```
import asyncio
from asyncio import Queue, PriorityQueue
from dataclasses import dataclass, field


@dataclass(order=True)
class WorkItem:
    priority: int
    data: str = field(compare=False)


async def worker(queue: Queue):
    while not queue.empty():
        work_item: WorkItem = await queue.get()
        print(f'Processing work item {work_item}')
        queue.task_done()
```

```
async def main():
    priority_queue = PriorityQueue()

    work_items = [WorkItem(3, 'Lowest priority'),
                  WorkItem(2, 'Medium priority'),
                  WorkItem(1, 'High priority')]

    worker_task = asyncio.create_task(worker(priority_queue)

    for work in work_items:
        priority_queue.put_nowait(work)

    await asyncio.gather(priority_queue.join(), worker_task)


asyncio.run(main())
```

In the preceding listing, we create a `dataclass` with `ordered` set to `True`. We then add a priority integer and a string data field, excluding this from the comparison. This means that when we add these work items to the queue, they'll only be sorted by the priority field. Running the code above, we can see that this is processed in the proper order:

```
Processing work item WorkItem(priority=1, data='High priorit
Processing work item WorkItem(priority=2, data='Medium prior
Processing work item WorkItem(priority=3, data='Lowest prior
```

Now that we know the basics of priority queues, let’s translate this back into the earlier example of our order management API. Imagine we have some “power user” customers who spend a lot of money on our e-commerce site. We want to ensure that their orders always get processed first to ensure the best experience for them. Let’s adapt our earlier example to use a priority queue for these users.

Listing 12.7 A priority queue in a web application

```
import asyncio
from asyncio import Queue, Task
from dataclasses import field, dataclass
from enum import IntEnum
from typing import List
from random import randrange
from aiohttp import web
from aiohttp.web_app import Application
from aiohttp.web_request import Request
from aiohttp.web_response import Response

routes = web.RouteTableDef()

QUEUE_KEY = 'order_queue'
TASKS_KEY = 'order_tasks'


class UserType(IntEnum):
    POWER_USER = 1
    NORMAL_USER = 2


@dataclass(order=True)
```

```
class Order:
    user_type: UserType
    order_delay: int = field(compare=False)


async def process_order_worker(worker_id: int, queue: Queue)
    while True:
        print(f'Worker {worker_id}: Waiting for an order...'
        order = await queue.get()
        print(f'Worker {worker_id}: Processing order {order}
        await asyncio.sleep(order.order_delay)
        print(f'Worker {worker_id}: Processed order {order}'
        queue.task_done()


@routes.post('/order')
async def place_order(request: Request) -> Response:
    body = await request.json()
    user_type = UserType.POWER_USER if body['power_user'] ==
    order_queue = app[QUEUE_KEY]
    await order_queue.put(Order(user_type, randrange(5)))
    return Response(body='Order placed!')


async def create_order_queue(app: Application):
    print('Creating order queue and tasks.')
    queue: Queue = asyncio.PriorityQueue(10)
    app[QUEUE_KEY] = queue
    app[TASKS_KEY] = [asyncio.create_task(process_order_work
                      for i in range(5)]
```

```
async def destroy_queue(app: Application):
    order_tasks: List[Task] = app[TASKS_KEY]
    queue: Queue = app[QUEUE_KEY]
    print('Waiting for pending queue workers to finish....')
    try:
        await asyncio.wait_for(queue.join(), timeout=10)
    finally:
        print('Finished all pending items, canceling worker
        [task.cancel() for task in order_tasks]


app = web.Application()
app.on_startup.append(create_order_queue)
app.on_shutdown.append(destroy_queue)

app.add_routes(routes)
web.run_app(app)
```

❶ An order class to represent our work item with a priority based on user type.

❷ Parse the request into an order.

The preceding listing looks very similar to our initial API to interact with a slow order management system with the difference being that we use a priority queue and create an `Order` class to represent an incoming order. When we get an incoming order, we now expect it to have a payload with a “power user” flag set to `True` for VIP users and `False` for other users. We can hit this endpoint with cURL like so

```
curl -X POST -d '{"power_user":"False"}' localhost:8080/orde
```

passing in the desired power user value. If a user is a power user, their orders will always be processed by any available workers ahead of regular users.

One interesting corner case that can come up with priority queues is what happens when you add two work items with the same priority right after one another. Do they get processed by workers in the order they were inserted? Let’s make a simple example to test this out.

Listing 12.8 A work item priority tie

```
import asyncio
from asyncio import Queue, PriorityQueue
from dataclasses import dataclass, field


@dataclass(order=True)
class WorkItem:
    priority: int
    data: str = field(compare=False)


async def worker(queue: Queue):
    while not queue.empty():
        work_item: WorkItem = await queue.get()
        print(f'Processing work item {work_item}')
        queue.task_done()

async def main():
    priority_queue = PriorityQueue()

    work_items = [WorkItem(3, 'Lowest priority'),
                    WorkItem(3, 'Lowest priority second'),
                    WorkItem(3, 'Lowest priority third'),
                    WorkItem(2, 'Medium priority'),
                    WorkItem(1, 'High priority')]

    worker_task = asyncio.create_task(worker(priority_queue)

    for work in work_items:
        priority_queue.put_nowait(work)

    await asyncio.gather(priority_queue.join(), worker_task)


asyncio.run(main())
```

In the preceding listing, we put three low-priority tasks in the queue first. We might expect these to be processed in order of insertion, but we don't exactly get that behavior when we run this:

```
Processing work item WorkItem(priority=1, data='High priorit
Processing work item WorkItem(priority=2, data='Medium prior
Processing work item WorkItem(priority=3, data='Lowest prior
Processing work item WorkItem(priority=3, data='Lowest prior
Processing work item WorkItem(priority=3, data='Lowest prior
```

It turns out that we process the low-priority items in the reverse order we inserted them. This is happening because the underlying `heapsort` algorithm is not a stable sort algorithm, as equal items are not guaranteed to be in the same order of insertion. Order when there are ties in priority may not be an issue, but if you care about it,