*SuperFast*Python

API Mastery

Python Asyncio Mastery

Discover Modern Asynchronous Programming in Python With Asyncio

Jason Brownlee

Python Asyncio Mastery

Discover Modern Asynchronous Programming In Python With Asyncio

Jason Brownlee

2023

Praise for *SuperFast*Python

"*I'm reading this article now, and it is really well made (simple, concise but comprehensive). Thank you for the effort! Tech industry is going forward thanks also by people like you that diffuse knowledge.*"

– **Gabriele Berselli**, Python Developer.

"*I enjoy your postings and intuitive writeups - keep up the good work*"

– **Martin Gay**, Quantitative Developer at Lacima Group.

"*Great work. I always enjoy reading your knowledge based articles*"

– **Janath Manohararaj**, Director of Engineering.

"*Great as always!!!*"

– **Jadranko Belusic**, Software Developer at Crossvallia.

"*Thank you for sharing your knowledge. Your tutorials are one of the best I've read in years. Unfortunately, most authors, try to prove how clever they are and fail to educate. Yours are very much different. I love the simplicity of the examples on which more complex scenarios can be built on, but, the most important aspect in my opinion, they are easy to understand. Thank you again for all the time and effort spent on creating these tutorials.*"

– **Marius Rusu**, Python Developer.

"*Thanks for putting out excellent content Jason Brownlee, tis much appreciated*"

– **Bilal B.**, Senior Data Engineer.

"*Thank you for sharing. I've learnt a lot from your tutorials, and, I am still doing, thank you so much again. I wish you all the best.*"

– **Sehaba Amine**, Research Intern at LIRIS.

"*Wish I had this tutorial 7 yrs ago when I did my first multithreading software. Awesome Jason*"

– **Leon Marusa**, Big Data Solutions Project Leader at Elektro Celje.

"*This is awesome*"

– **Subhayan Ghosh**, Azure Data Engineer at Mercedes-Benz R&D.

Copyright

Disclaimer

The information contained within this book is strictly for educational purposes. If you wish to apply ideas contained in this book, you are taking full responsibility for your actions.

The author has made every effort to ensure the accuracy of the information within this book was correct at time of publication. The author does not assume and hereby disclaims any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from accident, negligence, or any other cause.

Preface

Asynchronous programming is built into Python.

The language directly supports coroutines as first-class objects with the `async` and `await` expressions for asynchronous programming. The `asyncio` module provides tools for creating and managing asynchronous task and for developing non-blocking I/O client and server programs.

Asyncio is not coming, it's here and has been here since Python 3.5.

Skills in asyncio are in demand and the demand is growing.

Asynchronous programming and asyncio is how we develop modern event-driven programs in Python. This paradigm dominates modern Python web development, API development, and network programming.

You need to learn asyncio and this begins with a through knowledge of the `asyncio` module API in the Python standard library.

The official documentation is terse at best and asyncio examples on the web are outdated. There are few places a motivated Python developer can turn to really learn the asyncio API in detail.

I wrote this book to solve this problem and to show you exactly how to use the modern high-level asyncio API for application development, by example.

Together, we can make Python code run faster.

Thank you for letting me guide you along this path.

Jason Brownlee, Ph.D.
2023.

Contents

Introduction

Chapter 1

Introduction

Welcome to *Python Asyncio Mastery*!

Asyncio refers to both support for coroutines in Python and the `asyncio` module in the standard library. Together, these elements are required to develop programs using the asynchronous programming paradigm in Python.

This book teaches you how to use the modern high-level asyncio API for application development.

It is not a dry, long-winded academic textbook. Instead, it is a deep-dive for Python developers that provides carefully designed tutorials with complete and working code examples that you can copy-paste into your project today and get results.

Before we dive into the tutorials, let's look at what is coming with a breakdown of this book.

1.1 Who Is This For

Before we dive in, let's make sure you're in the right place.

This book is designed for Python developers who want to discover how to use asyncio.

Specifically, this book is for:

- Developers that can write simple Python programs.
- Developers that want to learn asynchronous programming.
- Developers that are working with I/O-based tasks.

This book does not require that you are an expert in the Python programming language or concurrency.

Specifically:

- You do not need to be an expert Python developer.
- You do not need to be an expert in concurrency.

Next, let's take a look at what this book will cover.

1.2 Book Overview

This book is designed to bring you up-to-speed with the high-level asyncio API.

The coverage of the API is not exhaustive, but is pretty close. The goal was to show you how to use all of the high-level asyncio API, intended for application development, with worked code examples.

This book is divided into seven parts, they are:

- **Part I: Asyncio**. Tutorials on how to get started with asyncio.
- **Part II: Asyncio Tasks**. Tutorials on how to create and use asyncio tasks.
- **Part III: Multiple Tasks**. Tutorials on how to manage multiple asyncio tasks.
- **Part IV: More Tasks**. Tutorials on more advanced management of asyncio tasks.
- **Part V: Structures**. Tutorials on asyncio code and data structures.
- **Part VI: Synchronization**. Tutorials on coroutine synchronization patterns.
- **Part VII: Streams**. Tutorials on non-blocking I/O with subprocesses and streams.

The book is further divided into 35 tutorials across the seven parts, they are:

Part I: Asyncio:

- **Tutorial 01**: What is Asyncio
- **Tutorial 02**: What is a Coroutine
- **Tutorial 03**: Asyncio Hello World

Part II: Asyncio Tasks:

- **Tutorial 04**: What are Tasks
- **Tutorial 05**: Task Names
- **Tutorial 06**: Task Status
- **Tutorial 07**: Task Cancellation
- **Tutorial 08**: Task Results
- **Tutorial 09**: Task Exceptions
- **Tutorial 10**: Task Done Callbacks
- **Tutorial 11**: Main and Current Task
- **Tutorial 12**: All Tasks

Part III: Multiple Tasks:

- **Tutorial 13**: Gather Tasks
- **Tutorial 14**: Wait On Tasks
- **Tutorial 15**: Tasks As Completed
- **Tutorial 16**: Task Group
- **Tutorial 17**: Tasks With Timeouts

Part IV: More Tasks:

- **Tutorial 18**: Shield Tasks
- **Tutorial 19**: Sleep Tasks
- **Tutorial 20**: Wait For Tasks
- **Tutorial 21**: Blocking Tasks

Part V: Structures:

- **Tutorial 22**: Asynchronous Iterators
- **Tutorial 23**: Asynchronous Generators
- **Tutorial 24**: Asynchronous Context Managers
- **Tutorial 25**: Asynchronous Queues

Part VI: Synchronization:

- **Tutorial 26**: Locks
- **Tutorial 27**: Events
- **Tutorial 28**: Conditions
- **Tutorial 29**: Semaphores
- **Tutorial 30**: Barriers

Part VII: Streams:

- **Tutorial 31**: Subprocesses
- **Tutorial 32**: Client Sockets
- **Tutorial 33**: Check Website Status
- **Tutorial 34**: Server Sockets
- **Tutorial 35**: Chat Client and Server

Next, let's take a closer look at how the tutorials are structured.

1.3 Tutorial Structure

This book teaches you the asyncio API by example.

Each lesson has a specific learning outcome and is designed to be completed in less than one hour.

Each lesson is also designed to be self-contained so that you can read the lessons out of order if you choose, such as dipping into topics in the future to solve specific programming problems.

The lessons were written with some intentional repetition of key APIs and concepts. These gentle reminders are designed to help embed the common usage patterns in your mind so that they become second nature.

We Python developers learn best from real and working code examples.

Next, let's learn more about the code examples provided in this book.

1.4 Code Examples

All code examples use Python 3.

Python 2.7 is not supported because it reached its end of life in 2020.

I recommend using the most recent version of Python 3 available at the time you are reading this, although Python 3.12 or higher is sufficient to run all code examples in this book.

You do not require any specific Integrated Development Environment (IDE). I recommend typing code into a simple text editor like Sublime Text that runs on all modern operating systems. I'm a Sublime user myself, but any text editor will do. If you are familiar with an IDE, then, by all means, use it.

Each code example is complete and can be run as a standalone program. I recommend running code examples from the command line (also called the command prompt on Windows or terminal on macOS) to avoid any possible issues.

The procedure to run a Python program from the command line is as follows:

1. Save the code file to a directory of your choice with a `.py` extension.
2. Open your command line.
3. Change directory to the location where you saved the Python program.
4. Execute the program using the Python interpreter followed by the name of the program.

For example:

```
python my_program.py
```

I recommend running programs on the command line instead of notebooks. It is easy, it works for everyone, it avoids all kinds of problems that beginners have with notebooks and IDEs, and programs run fastest on the command line.

That being said, if you know what you're doing, you can run code examples within your IDE if you like. Editors like Sublime Text will let you run Python programs directly, and this is fine. I just can't help you debug any issues you might encounter because they're probably caused by your development environment.

Most tutorials in this book provide code examples. These are typically introduced first via snippets of code that begin with an ellipsis (`...`) to clearly indicate that they are not a complete code example. After the program is introduced via snippets, a complete code example is listed that includes all of the snippets tied together, with any additional glue code and import statements.

I recommend typing code examples from scratch to help you learn and memorize the APIs.

Beware of copy-pasting code from the EBook version of this book as you may accidentally lose or add white space, which may break the execution of the program.

A code file is provided for each complete example in the book organized by tutorial and example within each tutorial. You can execute these programs directly or use them as a reference.

You can download all code examples from here:

- Download Code Examples.
  https://SuperFastPython.com/pam-code

All code examples were tested on a POSIX machine by myself and my technical editors prior to publication.

APIs can change over time, functions can become deprecated, and idioms can change and be replaced. I keep this book up to date with changes to the Python standard library and you can email me any time to get the latest version. Nevertheless, if you encounter any warnings or problems with the code, please contact me immediately and I will fix them. I pride myself on having complete and working code examples in all of my tutorials.

Next, let's consider how we might approach working through this book.

1.5 How To Read

You can work at your own pace.

There's no rush and I recommend that you take your time.

This book is designed to be read linearly from start to finish, guiding you from being a Python developer at the start of the book to being a Python developer that can confidently used the asyncio API in your Python projects by the end of the book.

In order to avoid overload, I recommend completing one or two tutorials per day, such as in the evening or during your lunch break. This will allow you to complete the transformation in about one month.

I recommend maintaining a directory with all of the code you type from the tutorials in the book. This will allow you to use the directory as your own private code library, allowing you to copy-paste code into your projects in the future.

I recommend trying to adapt and extend the examples in the tutorials. Play with them. Break them. This will help you learn more about how the API works and why we follow specific usage patterns.

Next, let's review your newfound capabilities after completing this book.

1.6 Learning Outcomes

This book will transform you into a Python developer that can use the asyncio API in your Python projects.

1. You will confidently develop Python programs with the asynchronous programming paradigm, including:

1. How to structure and run an asyncio programs.

2. How to create and execute coroutines.
3. How to use the expressions added to the Python language to implement cooperative multitasking.

2. You will confidently create, run, and query asyncio tasks, including:

1. How to create and schedule coroutines as tasks and run them in the foreground or background.
2. How to query task status and assign them meaningful names.
3. How to cancel tasks, handle task cancellation, and check if a task has been canceled.
4. How to retrieve results from tasks and exceptions raised in tasks.
5. How to add and mange done callback functions executed by tasks.
6. How to introspect the current and all running tasks and know the role and importance of the main task.

3. You will confidently manage groups of asyncio tasks, including:

1. How to execute multiple tasks concurrently and retrieve their results once they are all done.
2. How to wait for a condition on a group of tasks, such as all done, first done, or first to fail.
3. How to process task results in the order that tasks are completed rather than the order they were issued.
4. How to define a related collection of tasks as a group and operate upon the group, such as cancel all if one fails.
5. How to execute tasks and blocks of asyncio code with timeouts.

4. You will confidently implement more complex task management, including:

1. How to shield tasks from cancellation.
2. How to yield control with sleep to allow scheduled tasks to execute.
3. How to wait for a single task to complete with a timeout.
4. How to execute a blocking task without stopping the asyncio event loop.

5. You will confidently use asynchronous control flow and data structures, including:

1. How to develop and traverse asynchronous iterators and generators
2. How to develop and use asynchronous context managers.
3. How to communicate between coroutines and tasks using queues.

6. You will confidently synchronize the behavior between tasks, including:

1. How to use mutex locks to avoid race conditions and deadlocks, ensuring coroutine safety.
2. How to synchronize behavior with shared events and use the wait/notify pattern with condition variables.
3. How to limit access to shared resources by tasks using semaphores.
4. How to coordinate behavior between tasks using barriers.

7. You will confidently communicate with other processes and networked programs efficiently, including:

1. How to create, manage and perform non-blocking reads and writes with subprocesses.
2. How to read and write with non-blocking network sockets.
3. How to develop an asynchronous website status checking application.
4. How to develop and manage asynchronous socket servers
5. How to develop an asynchronous group chat application.

Next, let's discover how we can get help when working through the book.

1.7 Getting Help

The tutorials in this book were designed to be easy to read and follow.

Nevertheless, sometimes we need a little extra help.

A list of further reading resources is provided at the end of each tutorial. These can be helpful if you are interested in learning more about the topic covered, such as fine-grained details of the standard library and API functions used.

The conclusions at the end of the book provide a complete list of websites and books that can help if you want to learn more about asyncio in Python and the relevant parts of the Python standard library. It also lists places where you can go online and ask questions about Python.

Finally, if you ever have questions about the tutorials or code in this book, you can contact me any time and I will do my best to help. My contact details are provided at the end of the book.

Now that we know what's coming, let's get started.

1.7.1 Next

Next up in the first tutorial, we will look at what exactly is asyncio.

Part I

Asyncio

Chapter 2

What Is Asyncio

Asyncio is new and challenging to understand for beginners.

The reason is because it requires a different way of thinking.

Asyncio programs are not like regular Python programs where we might have a linear sequence of steps. Instead, asyncio programs use an *asynchronous programming paradigm*.

Additionally, asyncio programs are different from other types of Python concurrency, such as `threading` and `multiprocessing` that use native threads and processes respectively. Instead, concurrency is achieved using *coroutines* that are baked into the language itself.

In this tutorial, you will discover what exactly asyncio is all about.

After completing this tutorial, you will know:

- That asyncio really refers to two things, support for coroutines in the language and the `asyncio` module in the standard library.
- What is asynchronous programming and how asyncio provides this capability in Python.
- When you should consider using asyncio in your Python programs.

Let's get started.

2.1 What Is Asyncio

Broadly, **asyncio** refers to the ability to implement asynchronous programming in Python.

Specifically, it refers to two elements:

1. The addition of `async/await` expressions to the Python language (circa Python 3.5).
2. The addition of the `asyncio` module to the Python standard library (circa Python 3.4).

Notice, the module came before the language changes. This is because there were different ways to realize the desired effects that were changed into their current form in Python version 3.5. We will get more into the history of asyncio in the next chapter.

Together, the module and changes to the language facilitate the development of Python programs that support asynchronous programming via coroutine-based concurrency, typically used for non-blocking I/O tasks.

Let's take a closer look at these two aspects of asyncio, starting with the changes to the language.

2.1.1 Python Support For Coroutines

The Python language was changed to accommodate asyncio with the addition of expressions and types.

More specifically, it was changed to support coroutines as first-class concepts. In turn, coroutines are the unit of concurrency used in asyncio programs.

A coroutine is a function that can be suspended and resumed.

A coroutine may suspend for many reasons, such as executing another coroutine, e.g. awaiting another task, or waiting for some external resources, such as a socket connection or process to return data.

Many coroutines can be created and executed at the same time. They have control over when they will suspend and resume, allowing them to cooperate as to when concurrent tasks are executed.

This is called cooperative multitasking and is different from the multitasking typically used with threads called preemptive multitasking.

A coroutine can be defined via the `async def` expression. It can take arguments and return a value, just like a function.

For example:

```
# define a coroutine
async def custom_coro():
    # ...
```

Calling a coroutine function will create a coroutine object.

It instantiates the coroutines. It does not execute the coroutine function.

For example:

```
...
# create a coroutine object
coro = custom_coro()
```

A coroutine can execute another coroutine via the `await` expression.

This suspends the caller and schedules the target for execution.

For example:

```
...
# suspend and schedule the target
await custom_coro()
```

Along with coroutines, we have asynchronous versions of programming constructs familiar to Python programmers, such as iterators, generators, and context managers.

An asynchronous iterator is an iterator that yields awaitables (things that can be awaited). It can be traversed using the `async for` expression.

For example:

```
...
# traverse an asynchronous iterator
async for item in AsyncIterator():
    print(item)
```

This does not execute the for-loop in parallel or concurrently.

Instead, the calling coroutine that executes the loop will suspend and internally await each awaitable yielded from the iterator.

Similarly, we can define an asynchronous generator as a coroutine that has at least one `yield` expression. Creating and traversing the generator will provide an asynchronous generator iterator that we can also navigate using the `async for` expression.

An asynchronous context manager is a context manager that can await the enter and exit methods. The `async with` expression is for creating and using asynchronous context managers.

The calling coroutine will suspend and await the context manager before entering the block for the context manager, and similarly when leaving the context manager block.

For example:

```
...
# use an asynchronous context manager
async with item in AsyncContextManager():
    # ...
```

These are the sum of the major changes to Python language to support coroutines.

This was just a high-level tour to get you familiar with the terrain, we will take a closer look at each aspect in later chapters.

Next, let's look at the `asyncio` module.

2.1.2 The `asyncio` Module

The `asyncio` module in the Python standard library provides functions and objects for developing coroutine-based programs using the asynchronous programming paradigm.

Specifically, it supports non-blocking I/O with subprocesses (for executing commands) and with streams (for socket programming).

The `asyncio` module is the mechanism that runs a coroutine-based program and implements cooperative multitasking between coroutines.

The `asyncio` module provides two APIs:

1. **High-level API**, intended for Python application developers (us!).
2. **Low-level API** , intended for framework developers, (not us, in most cases).

Most application programming use cases are satisfied using the high-level API that provides utilities for working with coroutines, streams, synchronization primitives, subprocesses, and queues.

These are all the interesting and useful things we will be exploring in this book.

The lower-level API provides the foundation for the high-level API and includes the internals of the event loop, transport protocols, policies, and more.

You know when you're using the low-level asyncio APIs when you are getting and passing around `loop` objects. Incidentally, this is also a sign that you are reading an out-dated asyncio tutorial online written before the high-level API matured.

The low-level API typically requires that you provide a reference to the asyncio event loop that you are using via a `loop` argument, or by method calls on the event loop itself.

What is the event loop? The event loop is the engine or runtime that executes asyncio programs. It is the central purpose for the `asyncio` module, to drive asyncio programs.

We create and start an asyncio event loop via the `asyncio.run()` module function in the high-level API.

This function takes a coroutine object as the entry point to the program.

For example:

```
...
# start the asyncio event loop
asyncio.run(main())
```

This will block the current thread until the asyncio program is completed. This means that a given Python thread can only run a single asyncio event loop at a time. Note, a coroutine is not a thread, we will learn more about this in the next chapter.

We can acquire the reference to the running event loop within our asyncio program via the `asyncio.get_running_loop()` module function.

For example:

```
...
# access the event loop
loop = asyncio.get_running_loop()
```

Sometimes we must interact with the event loop directly, as the high-level API does not yet provide a specific capability.

One example is to access the current time kept by the event loop via the `loop.time()` method. This may be needed when determining a delay for a timeout.

Another example is when we need to execute a blocking function in a separate thread pool or process pool via the `loop.run_in_executor()` method. We will learn more about how to do this in a later chapter.

For now, it is important to know that the event loop is the runtime that executes our asyncio programs and we rarely need to interact with it directly.

We now know about the two sides to asyncio, the language changes for coroutines and the module in the standard library.

Next, let's better understand what we mean by asynchronous programming.

2.2 Asynchronous Programming

Asynchronous means *not at the same time*, as opposed to synchronous or *at the same time*.

When programming, asynchronous means that the action is requested, although not performed at the time of the request. The requested action is performed later.

For example, we can make an asynchronous function call. This will issue the request to make the function call and will not wait around for the call to complete. We can choose to check on the status or result of the function call later.

Waiting around in the program for a result from a request is referred to as blocking. Asynchronous programming is a programming paradigm that focuses on the idea of not blocking. Instead, we fire off tasks that need to get done and get on with other tasks.

Requests and function calls are issued and executed *somehow* in the background at *some future time*. This frees the caller to perform other activities and handle the results of issued calls at a later time when results are available or when the caller is interested.

To be clear, it does not mean that asynchronous programming programs never block, instead the code an choose when to wait, which may not be at the time a request was made. Request and response are decoupled.

This may require specific programming patterns.

For example, issuing an asynchronous function call often results in some handle on the request that the caller can use to check on the status of the call or get results. This is often called a future or a promise.

Developing programs focused on asynchronous function calls and asynchronous tasks that make use of patterns like futures is referred to as asynchronous programming.

- **Asynchronous Programming**: The use of asynchronous techniques, such as issuing asynchronous tasks or function calls.

Therefore, asynchronous programming in Python broadly refers to making requests and not blocking to wait for them to complete.

Although there are other ways to achieve elements of asynchronous programming, full asynchronous programming in Python requires the use of coroutines and the `asyncio` module.

Asynchronous programming is commonly used with non-blocking I/O. In fact, they are so tightly coupled, that their usage is often referred to as asynchronous I/O. This is why in Python we call it *asyncio*, a contraction of *asynchronous I/O*.

Recall that Input/Output (I/O) means reading or writing from a resource.

In traditional programming, a read or write is requested and the caller waits for the request to complete. As such, these operations are commonly referred to as blocking I/O tasks.

Non-blocking I/O allows read and write calls to be made as asynchronous requests.

The underlying run time, or library, or operating system will handle the request and notify the calling program when the results are available.

We can develop Python programs using asynchronous programming using asyncio, broadly defined. Specifically, they will use coroutines and the `asyncio` module.

Why would we want to do this?

2.3 Asyncio Use Cases

A key reason we may need to use asyncio is to meet the scalability requirements of a project.

Asyncio allows us to develop programs that easily support many thousands (even tens or hundreds of thousands) of concurrent tasks, such as concurrent socket connections. This can be achieved with threads but requires significantly more resource overhead.

As such, asyncio is widely used across various domains due to its ability to handle asynchronous I/O operations efficiently.

Below are some of the more popular use cases for asyncio:

1. **Web Servers and Frameworks:** asyncio is frequently employed in web servers and frameworks like FastAPI, Quart, and Sanic to handle multiple concurrent client connections without blocking.
2. **Network Applications:** It's used in building network clients and servers where handling multiple connections simultaneously is crucial, such as chat applications, multiplayer games, or real-time communication systems.
3. **Data Streaming and Processing:** asyncio is useful for processing and handling data streams asynchronously, especially in scenarios like real-time analytics, log processing, and handling large volumes of data.

4. **Asynchronous APIs:** Building APIs that handle multiple requests concurrently without blocking using frameworks like Starlette or FastAPI. Client-side APIs may use asyncio that make HTTP and WebSocket connections with frameworks like AIOHTTP and HTTPx.
5. **IoT and Embedded Systems:** asyncio can be helpful in handling concurrent tasks in IoT devices or embedded systems where efficiency and non-blocking operations are essential. Common implementations make use of asyncio in MicroPython running on platforms such as Raspberry Pi.

These are just a few examples, but asyncio's versatility extends to various other domains wherever non-blocking I/O operations and concurrent task handling are necessary.

Importantly, asyncio has found a home in Python web programming.

Modern Python web application frameworks increasingly make an asyncio interface a preferred or only choice. Similarly, it is increasingly common for client API libraries that access remote endpoints to be implemented using an asyncio interface.

This means that to develop web applications or use web APIs, we likely have to develop asyncio programs. For this reason, learning asyncio is not a choice, but instead a requirement for modern Python developers.

Asyncio is new and interesting, and can be very powerful. Nevertheless, asyncio is not always the best choice for a project.

2.3.1 When Should We Not Use Asyncio

There are cases to be made for not using asyncio.

One reason to not use asyncio is that you do not require non-blocking network I/O. It might be reasonable to say that if you are not performing network programming of some kind in your Python application, then asyncio is probably not a good fit.

This is not technically true, as asyncio does support other forms of non-blocking I/O, such as with subprocesses, and can simulate non-blocking I/O with threads and processes for other common tasks, such as file I/O.

Another major reason to not use asyncio is that it does not deliver the benefit that we think it does.

Common misconceptions about asyncio include:

- Asyncio will work around the Global Interpreter Lock (GIL).
- Asyncio is faster than threads.
- Asyncio avoids the need for mutex locks and other synchronization primitives.
- Asyncio is easier to use than threads.

These are all false, for example:

- Coroutines run in a single thread making the GIL irrelevant.
- Tasks implemented using threads and coroutines run at the same speed.

- Coroutines can suffer race conditions just like threads and processes.
- We could make the case that coroutines are harder to use, given then use of the newer async/await syntax.

Choosing asyncio for one of these reasons is a bad idea.

That being said, performance is a big reason to choose coroutines and asyncio. Coroutines are faster to start than threads and do use less memory, allowing a given system to be able to run more coroutines concurrently than threads. We will touch on this again in the next chapter.

Another reason to not use asyncio is that we don't like asynchronous programming.

Asynchronous programming has been popular for some time now in a number of different programming communities, most notably the TypeScript and JavaScript communities. Developers coming to Python asyncio from these communities have an easier time, whereas more traditional developers trained in procedural and object-oriented programming require a shift in thinking and approach.

Asynchronous programming is different from procedural, object-oriented, and functional programming, and some developers just don't like it. If you feel disoriented, give it time and practice. Asyncio will grow on you, I promise.

2.4 Takeaways

You now know what asyncio is in Python.

Specifically, you know:

- That asyncio really refers to two things, support for coroutines and the `asyncio` module in the standard library.
- What is asynchronous programming and how asyncio provides this capability in Python.
- When you should consider using asyncio in your Python programs.

2.4.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

2.4.1.1 References

- Cooperative multitasking, Wikipedia.
  https://en.wikipedia.org/wiki/Cooperative_multitasking
- Asynchrony, Wikipedia.
  https://w.wiki/8T6c
- Asynchronous procedure call, Wikipedia.
  https://en.wikipedia.org/wiki/Asynchronous_procedure_call

2.4.1.2 APIs

- Python Glossary.
  https://docs.python.org/3/glossary.html
- PEP 492 – Asynchronous Context Managers and "async with".
  https://peps.python.org/pep-0492/
- asyncio – Asynchronous I/O.
  https://docs.python.org/3/library/asyncio.html
- Asyncio Event Loop.
  https://docs.python.org/3/library/asyncio-eventloop.html

2.4.2 Next

In the next tutorial, we will explore coroutines in Python.

Chapter 3

What Are Coroutines

Python provides first-class coroutines and the `asyncio` module for running and using them in Python applications.

Coroutines are used to develop concurrent applications, but are unlike thread-based and process-based concurrency commonly used in Python.

In this tutorial, you will discover coroutines in Python.

After completing this tutorial, you will know:

- What are coroutines, how do they work, and how do they compare to subroutines and generators.
- How to define, create, and run coroutines in Python.
- What is the history of coroutines in Python.

Let's get started.

3.1 What Is A Coroutine

A coroutine is a function that can be suspended and resumed.

It is often described as a generalized subroutine.

A subroutine can be executed, starting at one point and finishing at another point. Whereas, a coroutine can be executed then suspended and resumed many times before finally terminating.

> **coroutine**: Coroutines are a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points.

– Python Glossary.

Specifically, coroutines have control over when exactly they suspend (pause or block) their execution.

This typically involve the use of a specific expression, such as an `await` expression in Python, which is like a `yield` expression in a Python generator.

A coroutine may suspend for many reasons, such as executing another coroutine, e.g. awaiting another task, or waiting for some external resources, such as a socket connection or subprocess to return data.

Coroutines are used for concurrency.

Many coroutines can be created and executed at the same time. They have control over when they will suspend and resume, allowing them to cooperate as to when concurrent tasks are executed. This is called cooperative multitasking.

> ... in order to run multiple applications concurrently, processes voluntarily yield control periodically or when idle or logically blocked. This type of multitasking is called cooperative because all programs must cooperate for the scheduling scheme to work.

– Cooperative multitasking, Wikipedia.

Preemptive multitasking involves the operating system choosing what threads to suspend and resume and when to do so. The scheduler in the operating system actively intervenes and interrupts and suspends threads, and selects and resumes other threads. This is contrasted to the coroutines or tasks themselves making these decisions in the case of cooperative multitasking.

Coroutines are not threads, they are different units of concurrency, but it is helpful to mention threads when introducing coroutines to contrast the manner in which their concurrent multitasking is managed, top-down (threads) versus bottom-up (coroutines).

Now that we have some idea of what a coroutine is, let's deepen this understanding by comparing them to other familiar programming constructs.

3.2 Comparing Coroutines

We can better understand a coroutine by comparing it to other familiar programming concepts.

Let's dive in.

3.2.1 Coroutine Versus Routine And Subroutine

A *routine* and *subroutine* often refer to the same thing in modern programming.

Perhaps more correctly, a routine is a program, whereas a subroutine is a function in the program.

A routine has subroutines.

It is a discrete module of expressions that is assigned a name, may take arguments and may return a value. In Python, we call them functions or methods.

- **Subroutine**: A module of instructions that can be executed on demand, typically named, and may take arguments and return a value. also called a function.

A subroutine is executed, runs through the expressions, and returns somehow. Typically, a subroutine is called by another subroutine.

A coroutine is an generalization of a subroutine. This means that a subroutine is a special type of a coroutine.

A coroutine is like a subroutine in many ways, such as:

- They both are discrete named collections of expressions.
- They both can take arguments, or not.
- They both can return a value, or not.

The main difference is that a coroutine may choose to suspend and resume its execution one or more times before returning and exiting.

Both coroutines and subroutines can call other examples of themselves. A subroutine can call other subroutines. A coroutine call (suspend and schedule) other coroutines. However, a coroutine can also execute other subroutines but a subroutine (a Python function) can never await a coroutine.

The table below summaries the call relationships between subroutines (functions, methods, etc.) and coroutines.

Type	Func/Method/etc.	Coroutine
Function	Yes (can call)	No (cannot call)
Method	Yes	No
Lambda	Yes	No
Coroutine	Yes	Yes

When a coroutine executes another coroutine, it must suspend its execution and allow the other coroutine to run, then resume once the other coroutine has completed.

This is like a subroutine calling another subroutine. The difference is the suspension of the coroutine may allow any number of other coroutines an opportunity to execute as well. A function can only call another function and only the called function can make progress.

How? The asyncio event loop keeps track of all scheduled and running coroutines and allows each an opportunity to make progress, one at a time, when the current coroutine suspends itself.

This makes a coroutine calling another coroutine more powerful than a subroutine calling another subroutine. It is central to the cooperative multitasking facilitated by coroutines.

3.2.2 Coroutine Versus Generator

A generator is a special function that can suspend its execution.

> **generator**: A function which returns a generator iterator. It looks like a normal function except that it contains `yield` expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function.

– Python Glossary.

A generator function can be defined like a normal function although it uses a `yield` expression at the point it will suspend its execution and return a value.

A generator function will return a generator iterator object that can be traversed, such as via a for-loop. Each time the generator is executed, it runs from the last point it was suspended to the next `yield` statement.

> **generator iterator**: An object created by a generator function. Each yield temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator iterator resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

– Python Glossary.

A coroutine can suspend or "yield" to another coroutine using an `await` expression. It will then resume from this point once the awaited coroutine has been completed.

We might think of a generator as a special type of coroutine and cooperative multitasking used in loops.

> Generators, also known as semicoroutines, are a subset of coroutines.

– Coroutine, Wikipedia.

Before coroutines were developed, generators were extended so that they might be used like coroutines in Python programs.

This required a lot of technical knowledge of generators and the development of custom task schedulers.

This was made possible via changes to the generators and the introduction of the `yield from` expression.

These were later deprecated in favor of the modern `await` expression.

3.2.3 Coroutine Versus Task

A subroutine and a coroutine may represent a *task* in a program.

However, in Python, there is a specific object called an `asyncio.Task` object.

> A Future-like object that runs a Python coroutine. [...] Tasks are used to run coroutines in event loops.

– Coroutines and Tasks.

A coroutine can be wrapped in a `asyncio.Task` object and executed independently, as opposed to being executed directly within a coroutine.

The `Task` object provides a handle on the asynchronously executed coroutine.

- **Task**: A wrapped coroutine that can be executed independently.

This allows the wrapped coroutine to execute in the background. The calling coroutine can continue executing instructions rather than awaiting another coroutine.

A `Task` cannot exist on its own, it must wrap a coroutine.

Therefore a `Task` is a coroutine, but a coroutine is not a task.

We will take a longer look at `asyncio.Task` and how to use them in later chapters.

3.2.4 Coroutine Versus Thread

A coroutine is more lightweight than a thread.

We might say that a coroutine is "just" a function, whereas a thread is a Python object that maps onto a facility provided in the underlying operating system, called a native thread.

- **Thread**: heavyweight compared to a coroutine.
- **Coroutine**: lightweight compared to a thread.

A coroutine is defined as a function.

A thread is an object created and managed by the underlying operating system and represented in Python as a `threading.Thread` object.

- **Thread**: Managed by the operating system, represented by a Python object.

This means that coroutines are typically faster to create and start executing and take up less memory. Conversely, threads are slower than coroutines to create and start and take up more memory. This differences means that a system with fixed memory resources may execute more concurrent coroutines than threads, typically an order of magnitude more.

It is important to understand that a coroutine is not a thread, although both may be used to execute tasks concurrently.

Coroutines execute within one thread, therefore a single thread may execute many coroutines. More specifically, an asyncio event loop that runs coroutines operates within one thread.

Now that we better understand coroutines by comparing them to familiar programming constructs, let's take a closer look at how to create and use them.

3.3 How To Use Coroutines

Python coroutines are a little tricky in the beginning.

They are defined like a function, however, calling a coroutine like a function does not execute it. Instead, it creates an instance of a coroutine object. The coroutine object can then be scheduled for execution within the asyncio event loop.

Let's take a closer look at how to define, create, and run a coroutine.

3.3.1 Define A Coroutine

A coroutine can be defined via the `async def` expression.

This is an extension of the `def` expression for defining functions and methods.

For example:

```
# define a coroutine
async def custom_coro():
    # ...
```

A coroutine defined with the `async def` expression is referred to as a *coroutine function*.

A coroutine can then use coroutine-specific expressions within it, such as `await`, `async for`, and `async with`.

For example:

```
# define a coroutine
async def custom_coro():
    # await another coroutine
    await asyncio.sleep(1)
```

These coroutine-specific expressions cannot be used outside of a coroutine. Attempting to do so, such as using an `await` expression in a Python function or method, will result in a syntax error.

Next, let's look at how to create a coroutine.

3.3.2 Create A Coroutine

Once a coroutine is defined, it can be created.

This looks like calling a subroutine.

For example:

```
...
# create a coroutine
coro = custom_coro()
```

This does not execute the coroutine.

It constructs the coroutine that we defined and returns a handle to a `coroutine` object.

A `coroutine` Python object has methods, such as `send()` and `close()` (which we will never call). It is a type.

We can demonstrate this by creating an instance of a coroutine and calling the `type()` built-in function on it in order to report its type details.

For example:

```
# SuperFastPython.com
# check the type of a coroutine

# define a coroutine
async def custom_coro():
    pass

# create the coroutine
coro = custom_coro()
# check the type of the coroutine
print(type(coro))
```

Running the example reports that the created coroutine is a `coroutine` class.

We also get a `RuntimeWarning` because the coroutine was created but never executed. Creating and not running a coroutine is a common error made by developers new to asyncio, which is why we get a warning any time we do this.

```
<class 'coroutine'>
RuntimeWarning: coroutine 'custom_coro' was never awaited
```

A coroutine object is an awaitable.

This means it is a Python type that implements the `__await__()` magic method (dunder method).

Objects that implement the `__await__()` method, like a coroutine object can be used in an `await` expression, one way to run a coroutine.

Next, let's take a look at the ways we can run a created coroutine object.

3.3.3 Run A Coroutine

Coroutines can be defined and created, but they can only be executed within an event loop.

The event loop is a runtime that manages one or more running coroutines. It is the heart of an asyncio program, like the Python interpreter is the heart of a regular Python program.

The event loop manages the cooperative multitasking between coroutines.

Typically, an asyncio program has one thread that runs the event loop and ends when all coroutines running in the event look are done.

The typical way to start a coroutine event loop is via the `asyncio.run()` function.

This function takes one coroutine object and returns the value returned by the coroutine, if any.

For example:

```
...
# create a coroutine object
coro = main()
# start the event loop and execute a coroutine
result = asyncio.run(coro)
```

We almost never create and assign a coroutine object and then use it separately in asyncio programs. It is helpful in the beginning to separate these concerns so we can see what is going on.

Typically, we know that "calling" a coroutine constructs and returns a coroutine object, so we just use the returned object directly, such as in the call to `asyncio.run()`.

For example:

```
...
# start the event loop and execute a coroutine
asyncio.run(main())
```

This starts the asyncio event loop and schedules the coroutine for execution. It then executes it as soon as it is able, which is pretty much immediately.

The trick is that a coroutine used to start an asyncio program then creates and runs more coroutines.

This is achieved by using the `await` expression with coroutine objects, and other awaitables, like `asyncio.Task` instances (that we will learn more about later).

For example:

```
# define a coroutine
async def main():
    # await another coroutine
    await asyncio.sleep(1)
```

This creates a coroutine object from `asyncio.sleep()` which is passed to the `await` expression.

The `await` expression schedules the new `asyncio.sleep()` coroutine instance to run, then suspends the current coroutine called `main()`.

The event loop then gives all other running and scheduled coroutines a chance to run. It is only after the coroutine used in the `await` expression is done that the `main()` coroutine is able to resume its execution.

So we can run a coroutine from a regular Python program with `asyncio.run()` and we can execute coroutines at will from within coroutines using `await` expressions.

Most of the time, we will be defining our coroutines with `async def` expressions and running them with `await` expression, hence the common reference to asynchronous programming in Python as `async/await`, referring to the Python expressions directly.

> ... the async/await pattern is a syntactic feature of many programming languages that allows an asynchronous, non-blocking function to be structured in a way similar to an ordinary synchronous function.

– Async/await, Wikipedia.

This is just a first taste of coroutines, we will go over all of this again in the next chapter with a hello world example.

Before then, let's take a moment to review the brief history of coroutines in Python. This is needed, because we may come across the older syntax in our journeys and it is good to know what it is, and not use it.

3.4 When Were Coroutines Added To Python

Generators have slowly been migrating towards becoming first-class coroutines for a long time.

We can explore some of the major changes to Python to add coroutines, which we might consider a subset of the addition of asyncio to the Python language and standard library.

New methods like `send()` and `close()` were added to generator objects to allow them to act more like coroutines.

These were added in Python version 2.5 and described in PEP 342.

> This PEP proposes some enhancements to the API and syntax of generators, to make them usable as simple coroutines.

– PEP 342 – Coroutines via Enhanced Generators.

Later, allowing generators to emit a suspension exception as well as a stop exception was described in PEP 334.

> This PEP proposes a limited approach to coroutines based on an extension to the iterator protocol. Currently, an iterator may raise a StopIteration exception to indicate that it is done producing values. This proposal adds another exception to this protocol, SuspendIteration, which indicates that the given iterator may have more values to produce, but is unable to do so at this time.

– PEP 334 – Simple Coroutines via SuspendIteration.

The vast majority of the capabilities for working with modern coroutines in Python via the `asyncio` module were described in PEP 3156, added in Python version 3.3.

> This is a proposal for asynchronous I/O in Python 3, starting at Python 3.3. Consider this the concrete proposal that is missing from PEP 3153. The proposal includes a pluggable event loop, transport and protocol abstractions similar to those in Twisted, and a higher-level scheduler based on yield from (PEP 380). The proposed package name is asyncio.

– PEP 3156 – Asynchronous IO Support Rebooted.

A second approach to coroutines, based on generators, was added to Python 3.4 generally as an extension to Python generators themselves.

In this case, a coroutine was defined as a function that used the `@asyncio.coroutine` function decorator, a precursor to the `async def` expression.

Coroutines were executed using an asyncio event loop, provided in the `asyncio` module, like today.

A coroutine could suspend and execute another coroutine via the `yield from` expression, a precursor to the `await` expression.

For example:

```
# define a custom coroutine in Python 3.4
@asyncio.coroutine
def main():
    # suspend and execute coroutine in Python 3.4
    yield from asyncio.sleep(1)
```

The `yield from` expression was defined in PEP 380.

> A syntax is proposed for a generator to delegate part of its operations to another generator. This allows a section of code containing 'yield' to be factored out and placed in another generator.

– PEP 380 – Syntax for Delegating to a Subgenerator.

The `yield from` expression is still available for use in generators, although is a deprecated approach to suspending execution in coroutines, in favor of the `await` expression.

> Note: Support for generator-based coroutines is deprecated and is removed in Python 3.11. Generator-based coroutines predate async/await syntax. They are Python generators that use yield from expressions to await on `Futures` and other coroutines.

– Coroutines and Tasks.

Coroutines were added as first-class objects to Python in version 3.5.

This included changes to the Python language, such as the `async def` and `await` expression we are familiar with, as well as the `async with` and `async for` expressions that we will learn more about later.

These changes were described in PEP 492.

> It is proposed to make coroutines a proper standalone concept in Python, and introduce new supporting syntax. The ultimate goal is to help establish a common, easily approachable, mental model of asynchronous programming in Python and make it as close to synchronous programming as possible.

– PEP 492 – Coroutines with async and await syntax.

And that brings us to modern day. Changes since Python 3.5 have mostly focused on making the `asyncio` module easier to use.

3.5 Takeaways

You now know about coroutines in Python.

Specifically, you know:

- What are coroutines, how do they work, and how do they compare to subroutines and generators.
- How to define, create, and run coroutines in Python.
- What is the history of coroutines in Python.

3.5.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

3.5.1.1 References

- Coroutine, Wikipedia.
  https://en.wikipedia.org/wiki/Coroutine
- Cooperative multitasking, Wikipedia.
  https://en.wikipedia.org/wiki/Cooperative_multitasking
- Async/await, Wikipedia.
  https://en.wikipedia.org/wiki/Async/await

3.5.1.2 PEPs

- PEP 342 – Coroutines via Enhanced Generators.
  https://peps.python.org/pep-0342/
- PEP 334 – Simple Coroutines via SuspendIteration.
  https://peps.python.org/pep-0334/
- PEP 3156 – Asynchronous IO Support Rebooted.
  https://peps.python.org/pep-3156/
- PEP 380 – Syntax for Delegating to a Subgenerator.
  https://peps.python.org/pep-0380/

- PEP 492 – Coroutines with async and await syntax.
  https://peps.python.org/pep-0492/

3.5.1.3 APIs

- Python Glossary.
  https://docs.python.org/3/glossary.html
- Python Data model.
  https://docs.python.org/3/reference/datamodel.html
- Python Compound Statements.
  https://docs.python.org/3/reference/compound_stmts.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html
- Asyncio Event Loop.
  https://docs.python.org/3/library/asyncio-eventloop.html

3.5.2 Next

In the next tutorial, we will explore how to develop a "hello world" asyncio program.

Chapter 4

Asyncio Hello World

The first program we typically develop when starting with a new programming language or programming paradigm is called "hello world".

Asyncio programs are different from *regular* Python programs, they use the asynchronous programming paradigm. It's almost like learning a new programming language.

Therefore, to understand asyncio, we need to develop an asyncio "hello world" program.

In this tutorial, you will discover how to develop a hello world program using asyncio.

After completing this tutorial, you will know:

- How to develop the simplest asyncio "hello world" program.
- How to read and understand how the asyncio "hello world" program works in detail and how it might go wrong.
- How to develop a slightly more complex "hello world" program that helps us understand coroutines.

Let's get started.

4.1 Asyncio Hello World

Let's write the simplest "hello world" program for asyncio.

The complete example is listed below.

Type the example and run it. Or copy-paste it.

This is your first serious step on the asyncio journey.

```
# SuperFastPython.com
# hello world program for asyncio
import asyncio

# define a coroutine
```

```
async def  main():
    # report a message
    print('Hello world')

# start the asyncio event loop
asyncio.run(main())
```

Running the example reports the *hello world* message.

```
Hello world
```

Did it run okay for you? Ensure you are using Python 3.7 or higher.

Full of questions? Good!

Now, let's slow everything down and understand what this example does.

4.2 Asyncio Hello World In Gory Detail

We know how to type and run a hello world for asyncio, now let's understand what it does.

What should a hello world program for asyncio do?

If asyncio is for coroutine-based concurrency, we should create and run a coroutine.

If we were exploring thread-based concurrency in the `threading` module, we would create and run a thread. This would be the same if we were exploring process-based concurrency in the `multiprocessing` module.

Recall, a routine is a function. For example, we can define a function to print "hello world" as follows:

```
# custom routine
def main():
    # report a message
    print('Hello world')
```

A coroutine is a function that can be suspended. More correctly, a function that can suspend itself.

We can define a coroutine just like a normal function, except it has the added `async` keyword before the `def` keyword, e.g. the `async def` expression.

For example:

```
# define a coroutine
async def main():
    # report a message
    print('Hello world')
```

We saw this in the previous chapter on coroutines. So far, so good.

We cannot execute a coroutine like a routine.

If we call our `main()` directly, we get a `RuntimeWarning` that look like an error.

For example:

```
# SuperFastPython.com
# example of calling a coroutine directly
import asyncio

# define a coroutine
async def main():
    # report a message
    print('Hello world')

# call the coroutine directly
main() # error
```

Running the example results in warning messages that look scary.

The first `RuntimeWarning` says that we never awaited our coroutine, meaning it was created and never run.

The second `RuntimeWarning` gives a very technical suggestion about how to track down the coroutine that was created and not run.

```
...
RuntimeWarning: coroutine 'main' was never awaited
RuntimeWarning: Enable tracemalloc to get the object...
```

As we touched on in the previous chapter, we cannot call a coroutine directly.

Instead, we must have the routine called for us by the asyncio runtime, called the event loop.

This can be achieved by calling the `asyncio.run()` module function.

This function will start the asyncio event loop in the current thread, and we can only have one of these running in a thread at a time.

It takes a coroutine as an argument.

Not the name of the coroutine, like the `target` argument of a `threading.Thread` or `multiprocessing.Process`, instead it takes an instance of a coroutine. A coroutine object.

We can create an instance of a coroutine just like creating a Python object, and it looks like we are calling the coroutine.

For example:

```
...
# execute the coroutine
asyncio.run(main())
```

This will start the event loop and execute the coroutine and print the message.

But, let's unpack this further.

If we are creating an instance of a coroutine and passing it to the `run()` module function to execute, why not assign it first?

For example:

```
...
# create the coroutine and assign it to a variable
coro = main()
# execute the coroutine
asyncio.run(coro)
```

This makes more sense now.

We can clearly see the creation of the coroutine and it is passed to the `run()` function for execution.

There's one more thing.

If we don't execute an instance of a coroutine, we will get a `RuntimeWarning`.

This means that if we create and assign an instance of our custom coroutine and do not pass it to `asyncio.run()`, a warning message is reported, as we saw above.

For example:

```
# SuperFastPython.com
# example of calling a coroutine directly
import asyncio

# define a coroutine
async def main():
    # report a message
    print('Hello world')

# create the coroutine and assign it to a variable
coro = main() # warning
```

Running the example creates the coroutine, but does not do anything with it.

The Python interpreter then reports this as a warning message, similar to what we saw when we "called" the coroutine directly.

```
RuntimeWarning: coroutine '...' was never awaited
```

This is in the Python specification.

For example:

> When a native coroutine is garbage collected, a RuntimeWarning is raised if it was never awaited on

– PEP 492 – Coroutines with async and await syntax.

I suspect it's aimed to be helpful to new programmers using the new and confusing asyncio syntax, highlighting they created a thing that was never used.

Strange though.

It would be like creating a `threading.Thread()` and never calling the `start()` method, if you're familiar with Python threads. It does nothing, and there's little reason to do it, but we don't need to raise a runtime warning, do we?

So now we know what our hello world example does. It creates a coroutine objects, then runs it using a new asyncio event loop.

Let's do one more thing to make it more interesting.

4.3 Slightly Better Asyncio Hello World

The thing about coroutines is that they can be suspended.

Technically threads can be suspended too. The operating system can suspend a thread at any time and resume it again later, called a context switch.

But coroutines are different. We can specify the point at which they will be suspended and the suspending and resuming are controlled within the asyncio event loop.

A coroutine can suspend and wait for some condition via the `await` keyword.

Specifically, a coroutine can await an awaitable. An awaitable is another coroutine object or an `asyncio.Task` or technically anything that implements the `__await__()` method.

We can suspend a coroutine and do nothing via the `asyncio.sleep()` function. This will return a coroutine that does nothing for a fixed number of seconds. This turns out to be invaluable and we will learn more about it in a later chapter.

So let's update our hello world program so that our coroutine performs a sleep for one second.

For example:

```
# define a coroutine
async def main():
    # sleep for one second
    asyncio.sleep(1)
    # report a message
    print('Hello world')
```

But we cannot just call the `asyncio.sleep()` function.

Doing so results in another warning message, as you might expect.

For example:

```
# SuperFastPython.com
# hello world program for asyncio that tries to sleep
import asyncio

# define a coroutine
async def main():
    # block
    asyncio.sleep(1) # warning
    # report a message
    print('Hello world')

# start the asyncio event loop
asyncio.run(main())
```

Running the example reports our `RuntimeWarning` message as before and the example does not sleep.

```
RuntimeWarning: coroutine 'sleep' was never awaited
RuntimeWarning: Enable tracemalloc to get the object...
Hello world
```

In fact, the `asyncio.sleep()` is creating and returning a coroutine object.

The coroutine is not being used, it gets garbage collected and the Python interpreter tells us we did not use it for some reason.

We can make this clear if we assign the return value from `asyncio.sleep()`, which we would never do in practice by the way.

```
...
# create and store a sleep coroutine
sleeper = asyncio.sleep(1)
```

We do not need to pass it to `asyncio.run()`, because our `main()` coroutine is already being executed within the asyncio event loop. No need to start a second event loop for one coroutine.

Instead, we can request that the current coroutine suspend, execute the new coroutine, and wait for the new coroutine to finish.

This can be achieved using the `await` expression within a coroutine.

For example:

```
...
# create and store a sleep coroutine
sleeper = asyncio.sleep(1)
# pause the coroutine and execute the new coroutine
await sleeper
```

Or, we do this on one line and await the return value from `asyncio.sleep()` directly.

```
...
# block
await asyncio.sleep(1)
```

Great!

The complete example is listed below.

```
# SuperFastPython.com
# hello world program for asyncio that sleeps
import asyncio

# define a coroutine
async def main():
    # block
    await asyncio.sleep(1)
    # report a message
    print('Hello world')

# start the asyncio event loop
asyncio.run(main())
```

Running the example prints the message, as before, but does more.

The `run()` function starts the asyncio runtime.

We create an instance of our custom coroutine and pass it to the asyncio event loop to execute.

It's executed, and the first thing it does is it creates another coroutine and asks it to be executed by the event loop, and waits for it to finish.

The `sleep()` coroutine is created and scheduled and the `main()` coroutine is suspended.

The event loop gives all scheduled and running coroutines an opportunity to run, starting with the newly scheduled `sleep()` coroutine. It is executed and suspends itself until one second in the future.

A second passes. The `sleep()` coroutine resumes and finishes, then our `main()` coroutine resumes.

The message is reported, then the asyncio event loop is shut down.

```
Hello world
```

One final point, we cannot await our custom coroutine in order to execute it.

For example:

```
...
# execute the coroutine
await main()
```

The `await` expression can only be used within a coroutine to execute and wait on an awaitable.

Put another way, the `await` expression must be used within code that is executed by an asyncio event loop. To bootstrap the coroutine runtime, typically the `asyncio.run()` function is used as the entry point into an asyncio program.

Asyncio "hello world" achieved.

4.4 Takeaways

You now know how to develop a "hello world" program using asyncio.

Specifically, you know:

- How to develop the simplest asyncio "hello world" program.
- How to read and understand how the asyncio "hello world" program works in detail and how it might go wrong.
- How to develop a slightly more complex "hello world" program that helps us understand coroutines.

4.4.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

4.4.1.1 References

- "Hello, World!" program, Wikipedia.
  https://w.wiki/JAS

4.4.1.2 APIs

- PEP 492 – Coroutines with async and await syntax.
  https://peps.python.org/pep-0492/
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

4.4.2 Next

In the next tutorial, we will explore asyncio tasks in detail.

Part II

Asyncio Tasks

Chapter 5

What Are Tasks

An asyncio task is a scheduled and independently managed coroutine.

Asyncio tasks provide a handle on independently running coroutines and allow them to be queried, canceled, and results and exceptions to be retrieved later.

We can create `Task` objects from coroutines in asyncio programs using factory functions in the asyncio API.

In this tutorial, you will discover how to create and use asyncio tasks.

After completing this tutorial, you will know:

- How to create an asyncio task using a coroutine.
- Understand what creating a task does and when an asyncio task runs.
- How to schedule and execute an asyncio task to run in the background.

Let's get started.

5.1 What Is A Task

An `asyncio.Task` is a class that schedules and independently runs an asyncio coroutine.

It provides a handle on a scheduled coroutine that an asyncio program can query and use to interact with.

A task is created from a coroutine. It requires a coroutine object, wraps the coroutine, schedules it for execution, and provides ways to interact with it.

This means that a task cannot exist on its own, it requires a coroutine.

A task is executed independently. This means it is scheduled in the asyncio event loop and will execute regardless of what else happens in the coroutine that created it.

This is different from executing a coroutine directly, where the caller must `await` it and is suspended until the target coroutine is complete.

The `asyncio.Task` class extends the `asyncio.Future` class and is an awaitable, meaning it can be used in an `await` expression.

A `Future` is a lower-level class that represents a result that will eventually arrive.

We typically do not create or use `Future` objects, but it is helpful to know that `Task` extends a `Future`. This is because classes that extend the `Future` class are often referred to as `Future`-like.

Because a task is awaitable it means that a coroutine can suspend and wait for a task to be done using the `await` expression.

Now that we know what an asyncio task is, let's look at how we might create and use them.

5.2 How To Create A Task

Generally, we do not create an `asyncio.Task` object from a coroutine object directly.

Instead, we use a factory function that takes a coroutine and returns an `asyncio.Task` object.

There are 3 ways you can create an asyncio `Task` from a coroutine, they are:

1. Create a Task with `asyncio.create_task()` (recommended)
2. Create a Task with `asyncio.ensure_future()` (low-level)
3. Create a Task with `loop.create_task()` (low-level)

I mention all three, because you may see their usage elsewhere.

Generally, it is recommended to stick with the high-level asyncio API and to only create tasks using the `asyncio.create_task()` function.

Let's take a closer look.

5.2.1 Recommended Way To Create A Task

The `asyncio.create_task()` function takes a coroutine object and an optional name for the task and returns an `asyncio.Task` object.

For example:

```
...
# create and schedule a task
task = asyncio.create_task(task_coroutine())
```

We can specify a name for the task when it is created, and we will take a closer look at this in the next chapter.

The `create_task()` function does not block, instead it returns immediately.

The returned task object is awaitable. This means we can choose to await it immediately, if we want.

For example:

```
...
# create and schedule a task
task = asyncio.create_task(task_coroutine())
# suspend and wait for task to be done
await task
```

This is the same as awaiting the coroutine directly, which is simpler and preferred:

```
...
# suspend and wait for coroutine to be done
await task_coroutine()
```

It is generally a best practice to assign and keep a reference to the created task to avoid it from being garbage collected.

This means, it is not a good idea to assign new tasks intended to run in the background to the underscore variable name, for example:

```
...
# create and schedule a background task (don't do this)
_ = asyncio.create_task(task_coroutine())
```

It also means that it is not a good idea to call `create_task()` without assigning its return value.

```
...
# create and schedule a background task (don't do this)
asyncio.create_task(task_coroutine())
```

Instead, assign tasks to a meaningful variable name.

We can also create a task using the `TaskGroup` which will keep a reference to our task for us, and we will take a closer look at this in a later chapter.

Now that we know how to create an asyncio task, let's consider what creating a task actually does.

5.2.2 What Does Creating A Task Do?

Creating a task does a few things.

1. It wraps the provided coroutine in an `asyncio.Task` instance.
2. It schedules the task for execution in the event loop.
3. It returns the `Task` object that provides a handle on the scheduled coroutine.

The coroutine that is wrapped in the task is free to execute independently of the coroutine that created and scheduled it.

The wrapped coroutine does not run immediately. Instead, it is scheduled and will run as soon as the event loop finds an opportunity to execute the task.

Next, let's better understand when the scheduled task will run.

5.2.3 When Does A Task Run?

Although we can schedule a coroutine to run independently as a task with the `create_task()` function, it may not run immediately.

In fact, the task will not execute until the event loop has an opportunity to run it.

This will not happen until all other coroutines are not running and it is the task's turn to run.

It means that the caller (that created and scheduled the task) will block the new task from executing until it either terminates or suspends execution. If the caller terminates, then this responsibility rises to the next coroutine up in the callgraph (e.g. its caller).

This is an important point. A task will not run until the event loop is given an opportunity to run it. Sometimes this means explicitly giving the event loop that opportunity to at least start running the new task.

We will explore this more in a later chapter on sleeping coroutines.

5.2.4 Run The Task In The Background

Generally, we use tasks because we require coroutines to run concurrently, in the background.

A created and scheduled task will run in the background by default. It is independent of the coroutine that created and scheduled it.

This generally means that the coroutine that created it does not have to await it.

Even as a background task, we should find a way to store the returned task object so that it is not garbage collected.

This might mean assigning it to a global variable.

For example:

```
...
# create and schedule a task from a coroutine
MY_TASK = asyncio.create_task(task_coroutine())
```

The `Task` API documentation suggests storing background tasks in a global collection, such as a set, and removing them from the set once the task is done, such as via a done-callback function. We will explore done callback functions for tasks in a later chapter.

A modern solution is to use a `TaskGroup`, also explored in a later chapter.

Now that we know how to create asyncio tasks, let's look at some worked examples.

5.3 Example Of Creating A Task

We can explore how to create a task using the `asyncio.create_task()` function.

In this example, we will define a task coroutine that reports a message and sleeps for a moment.

The main coroutine will create a task from the task coroutine and then wait for the task to complete.

The complete example is listed below.

```
# SuperFastPython.com
# example of creating an asyncio task
import asyncio

# define a coroutine for a task
async def task_coroutine():
    # report a message
    print('Executing the task')
    # block for a moment
    await asyncio.sleep(1)

# custom coroutine
async def main():
    # report a message
    print('Main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # wait for the task to complete
    await task
    # report a final message
    print('Main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and executes it as the entry point into the asyncio program.

Just like using a `main()` function in regular Python programs, using a `main()` coroutine to drive an asyncio program is a common idiom. We will use it in the examples in this book going forward.

The `main()` coroutine runs, first printing a message. It then creates an instance of the `task_coroutine()` coroutine and uses it in a call to `asyncio.create_task()` to create a `asyncio.Task` object.

The new `asyncio.Task` instance wraps the coroutine, schedules it for execution, and returns

the task instance which is assigned to the local variable `task`.

The `main()` coroutine then awaits the `task` variable and is suspended, waiting for the task to be done.

The task coroutine is given an opportunity to run, reporting a message then sleeping for a moment. It resumes and then terminates.

The `main()` coroutine resumes, reports a message, then terminates.

This highlights how we can use the `create_task()` function to create a task from a custom coroutine and how the caller must suspend to allow the task to execute, in this case by awaiting the task directly.

```
Main coroutine started
Executing the task
Main coroutine done
```

Next, let's take a look at how we might create many tasks.

5.3.1 Example Of Creating Many Tasks

We can explore how we might create many tasks from a coroutine.

In this example, we will update the task coroutine to take an integer argument that is then reported.

The main coroutine will create many `Task` objects from the task coroutine in a loop, using a list comprehension. This creates and schedules the tasks and creates a list of the returned task instances.

The complete example is listed below.

```
# SuperFastPython.com
# example of creating many asyncio tasks
import asyncio

# define a coroutine for a task
async def task_coroutine(number):
    # report a message
    print(f'>Executing the task {number}')
    # block for a moment
    await asyncio.sleep(1)

# custom coroutine
async def main():
    # report a message
    print('Main coroutine started')
    # create and schedule many tasks
    tasks = [asyncio.create_task(
```

```
        task_coroutine(i)) for i in range(20)]
    # wait for each task to complete
    for task in tasks:
        await task
    # report a final message
    print('Main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and executes it as the entry point into the asyncio program.

The `main()` coroutine runs and prints a message.

It then loops and creates 20 tasks in a list comprehension. This provides a unique integer to each coroutine as it is created.

Each iteration creates and schedules a coroutine for execution as a task and collects the `Task` object in a list.

The `main()` coroutine then loops over the list of `Task` instances and awaits each in turn.

The task coroutines are given an opportunity to run, reporting a message with their integer argument then sleeping for a moment.

Only one task executes at a time and sequentially in order. This is because the tasks were scheduled sequentially and the event loop is honoring the scheduling order.

All tasks are completed and the `main()` coroutine resumes, reports a message, then terminates.

This highlights how we might create a list of tasks from a coroutine. The list comprehension idiom for creating lists of tasks is common and we will see it many times going forward.

```
Main coroutine started
>Executing the task 0
>Executing the task 1
>Executing the task 2
>Executing the task 3
>Executing the task 4
>Executing the task 5
>Executing the task 6
>Executing the task 7
>Executing the task 8
>Executing the task 9
>Executing the task 10
>Executing the task 11
>Executing the task 12
>Executing the task 13
```

```
>Executing the task 14
>Executing the task 15
>Executing the task 16
>Executing the task 17
>Executing the task 18
>Executing the task 19
Main coroutine done
```

Next, let's take a look at some common errors when using tasks.

5.4 3 Common Errors Creating A Task

There are handful of common errors when using tasks, and it's helpful to touch on them so that they can be avoided.

1. Tasks must be created from coroutines.

We cannot call `asyncio.create_task()` and pass it a function name or a lambda.

Attempting to call `create_task()` with anything other than a coroutine will result in a TypeError.

2. Tasks must be run in the event loop.

We cannot create and use tasks outside of an asyncio program.

Create and use tasks in asyncio programs. We cannot use them as a workaround to run coroutines in regular Python programs.

Technically, we can create tasks in a regular Python program, but not running them will result in a `RuntimeWarning`. We would then have to access a running event loop or create a new event loop to run our task.

3. Tasks must be assigned and the reference must be kept.

As we mentioned above, we must keep a reference to all tasks created via the `create_task()` function.

This is to avoid the garbage collection of tasks and in turn unexpected behavior.

Sometimes the solution is to store background tasks in a global variable or collection.

A better and more modern solution is to create tasks using the `TaskGroup` that will keep track of the tasks for us. We will learn more about how to create tasks with the `TaskGroup` in a later chapter.

5.5 Takeaways

You now know how to create an asyncio task.

Specifically, you know:

- How to create an asyncio task using a coroutine.
- Understand what creating a task does and when an asyncio task runs.
- How to schedule and execute an asyncio task to run in the background.

5.5.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- PEP 3156 – Asynchronous IO Support Rebooted.
  https://peps.python.org/pep-3156/
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html
- Asyncio Futures.
  https://docs.python.org/3/library/asyncio-future.html
- Asyncio Event Loop.
  https://docs.python.org/3/library/asyncio-eventloop.html

5.5.2 Next

In the next tutorial, we will explore how to set and use names for asyncio tasks.

Chapter 6

Task Names

Asyncio tasks are assigned default names when they are created.

Nevertheless, it can be beneficial to give tasks meaningful names. This can help when introspecting the asyncio event loop, when logging, and when debugging.

Task names can be made more meaningful by including task-specific data, such as identifiers.

In this tutorial, you will discover how to set and get the name of an asyncio task.

After completing this tutorial, you will know:

- Why setting a name for a task is helpful in asyncio programs.
- How to set a meaningful name for an asyncio task.
- How to retrieve the name of an asyncio task.

Let's get started.

6.1 Meaningful Task Names

Coroutines are how we define behavior in our asyncio programs, but tasks is how they are executed.

A modest asyncio program may create hundreds or thousands of tasks, many of which may be running at the same time.

As such, it is helpful to have some way to tell our asyncio tasks apart, especially when things go wrong.

This might be when we are logging details of our tasks via the `logging` module infrastructure.

It may also be when debugging our program in our favorite IDE that might introspect the asyncio event loop for us.

We may also develop debugging functions of our own that introspect the event loop and report all running tasks or all tasks in which we are interested. We will learn more about

how to introspect the event loop in a later chapter.

But, if we are printing the details of all tasks in a for loop, we need a way to tell one task from another.

The answer is to give tasks meaningful names.

6.2 How To Set The Task Name

A task may have a name.

We can assign and retrieve names for our tasks in asyncio.

6.2.1 Set A Task Name

The name of a task can be set when the task is created from a coroutine via the `name` argument to the `asyncio.create_task()` function.

For example:

```
...
# create a task from a coroutine
task = asyncio.create_task(coro(), name='MyTask')
```

The name for the task can also be set via the `set_name()` method.

For example:

```
...
# set the name of the task
task.set_name('MyTask')
```

6.2.2 Get A Task Name

We can retrieve the name of a task via the `get_name()` method.

For example:

```
...
# get the name of a task
name = task.get_name()
```

It is interesting to note that we can set and change the task name while the task is running.

This may be helpful if we would like the task name to include some state about the task, such as some program-specific state.

Now that we are familiar with how to set and get the task name, let's look at some examples.

6.3 Example Of Checking Default Task Names

Tasks are assigned a default name when they are created.

The default names have the format `'Task-%d'`, where `'%d'` refers to the sequential task number created in the event loop.

For example, the first few tasks will have the default names `Task-1`, `Task-2, Task-3`, and so on.

We can explore the default name assigned to an asyncio task.

In this example, we will define a coroutine for a task that will report a message and block for a moment.

We will then define the main coroutine that will be used as the entry point to the asyncio program. The main coroutine will report a message, then create and schedule the task coroutine. It will then wait for the task to be completed.

Once completed, the main coroutine will print the task and its name.

The complete example is listed below.

```
# SuperFastPython.com
# example of getting the default task name
import asyncio

# define a coroutine for a task
async def task_coroutine():
    # report a message
    print('executing the task')
    # block for a moment
    await asyncio.sleep(1)

# custom coroutine
async def main():
    # report a message
    print('main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # wait for the task to complete
    await task
    # report the task
    print(task)
    # report the task name
    print(f'name: {task.get_name()}')
    # report a final message
    print('main coroutine done')
```

```
# start the asyncio event loop
asyncio.run(main())
```

Running the example starts the asyncio event loop and executes the `main()` coroutine.

The `main()` coroutine reports a message, then creates and schedules the task coroutine.

It then suspends and awaits the task to be complete.

The task runs, reports a message, and sleeps for a moment before terminating normally.

The `main()` coroutine resumes.

It prints a string representation of the task, which we can see shows the name of the task as `'Task-2'`.

Notice that the string representation of the task also includes additional details such as the coroutine used to run it (e.g. `task_coroutine()`) and its current status (e.g. `done`).

It then reports just the name of the task, which we can see matches the name included in the string representation of the task.

This example highlights that tasks are assigned a name by default that follows a predictable naming convention.

```
main coroutine started
executing the task
<Task finished name='Task-2'
    coro=<task_coroutine() done, defined at ...>
    result=None>
name: Task-2
main coroutine done
```

Next, let's look at how we might set the name of a task when it is created.

6.4 Example Of Setting The Task Name

We can set the name of a task when it is created and scheduled via an argument to the `create_task()` function.

In this case, we can update the previous example to supply a name for the task when it is created in the `main()` coroutine.

```
# create and schedule the task
task = asyncio.create_task(
    task_coroutine(), name='MyTask')
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of setting the task name when it is created
import asyncio

# define a coroutine for a task
async def task_coroutine():
    # report a message
    print('executing the task')
    # block for a moment
    await asyncio.sleep(1)

# custom coroutine
async def main():
    # report a message
    print('main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(
        task_coroutine(), name='MyTask')
    # wait for the task to complete
    await task
    # report the task
    print(task)
    # report the task name
    print(f'name: {task.get_name()}')
    # report a final message
    print('main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example starts the asyncio event loop and executes the `main()` coroutine.

The `main()` coroutine reports a message, then creates and schedules the task coroutine with a custom name.

It then suspends and awaits the task to be completed.

The task runs, reports a message, and sleeps for a moment before terminating normally.

The `main()` coroutine resumes.

It prints a string representation of the task, which we can see shows the custom name of `'MyTask'`.

It then reports just the name of the task, which we can see matches the name included in the string representation of the task.

This example highlights that we can assign a task a distinct name when it is created and

access it again later.

```
main coroutine started
executing the task
<Task finished name='MyTask'
    coro=<task_coroutine() done, defined at ...>
    result=None>
name: MyTask
main coroutine done
```

6.5 Takeaways

You now know how to set and get the name of an asyncio task.

Specifically, you know:

- Why setting a name for a task is helpful in asyncio programs.
- How to set a meaningful name for an asyncio task.
- How to retrieve the name of an asyncio task.

6.5.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

6.5.2 Next

In the next tutorial, we will explore how to check the status of asyncio tasks.

Chapter 7

Task Status

An asyncio `Task` is an object that schedules and independently runs an asyncio coroutine.

It provides a handle on a scheduled coroutine that an asyncio program can query and use to interact with the coroutine.

Because tasks are executed asynchronously, we often need to check on the status of the task, such as whether it is done, is still running, or was canceled.

In this tutorial, you will discover how to check the status of an asyncio task.

After completing this tutorial, you will know:

- How to check if an asyncio task is running or is done.
- How to check if a task was canceled or completed normally.
- Understand the conditions under which we may consider that a task is done.

Let's get started.

7.1 How To Check Task Status

After a `Task` is created, we can check its status.

There are two statuses we might want to check, they are:

- Whether the task is done, or is still running.
- Whether the task is done because it was canceled, or not.

Let's take a closer look at each in turn.

7.1.1 Check If A Task Is Done

We can check if a task is done via the `done()` method.

The method returns `True` if the task is done, or `False` otherwise.

For example:

```
...
# check if a task is done
if task.done():
    print('The task is DONE')
else:
    print('The task is RUNNING')
```

A task is done if it has had the opportunity to run and is now no longer running.

A task that has been scheduled is not done, it is not really running either, but we don't have a clean way to determined a *scheduled-but-not-started* state.

Similarly, a task that is running is not done.

The idea of a *done* task is important and we will dig into it more in a moment.

7.1.2 Check If A Task Is Canceled

Tasks can be canceled. The whole next chapter will focus on this. For now, just know that we can cancel tasks.

We can check if a task is canceled via the `cancelled()` method.

The method returns `True` if the task was canceled, or `False` otherwise.

For example:

```
...
# check if a task was canceled
if task.cancelled():
    # ...
```

A task is canceled if the `cancel()` method was called on the task and completed successfully, e.g. `cancel()` returned `True`.

A task is not canceled if the `cancel()` method was not called, or if the `cancel()` method was called but failed to cancel the task.

A canceled task will be done, but a non-canceled task may be done or may still be running.

7.1.3 When Is An Asyncio Task Done

A done task means the task is finished, but finished may or may not mean successful.

It means that the task had at least one opportunity to execute and is no longer able to execute.

A task is done if any of the following things happened:

1. The task's coroutine finished normally and terminated.
2. The task's coroutine returned explicitly and terminated (e.g. a `return` statement).

3. An error or exception is raised in the task's coroutine that was not handled and terminated the coroutine.
4. The task was canceled, raising a special `CancelledError` exception, terminating the coroutine.

The common theme is that a task is done when the task's coroutine has terminated.

Also, keep in mind that:

- A scheduled task is not done (it is running but technically not yet executing).
- A suspended task is not done (it is running).
- A sleeping task is not done (it is running and suspended).

Now that we know how to check the status of a task, let's look at some worked examples.

7.2 Example Of Checking If A Task Is Done

We can explore how to check if a task is done.

In this example, we will define a task coroutine that reports a message and sleeps for a moment. We will then create and schedule the task coroutine from the main coroutine.

We will check the status of the task immediately after it is created before it has run.

We will then give the task an opportunity to run, then check the status of the task while it is running.

Finally, we will check the status of the task after it is finished.

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of checking if an asyncio task is done
import asyncio

# define a coroutine for a task
async def task_coroutine():
    # report a message
    print('executing the task')
    # block for a moment
    await asyncio.sleep(1)

# custom coroutine
async def main():
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # check if it is done
    print(f'>task done: {task.done()}')
    # wait a moment
```

```
    await asyncio.sleep(0.1)
    # check if it is done
    print(f'>task done: {task.done()}')
    # wait for the task to complete
    await task
    # check if it is done
    print(f'>task done: {task.done()}')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and runs it as the entry point to the asyncio program.

The `main()` coroutine runs, first creating an instance of the `task_coroutine()` coroutine and then using it to create and schedule a task.

The `main()` coroutine then checks if the new task is done, which it is not because it has just been scheduled. It then sleeps, giving the new task an opportunity to run.

The task begins executing, reporting a message, and then starts sleeping.

The `main()` coroutine then resumes and checks if the coroutine is done. At this point, the `task_coroutine()` is still running, yet has suspended in a call to `sleep()`.

The `main()` coroutine then awaits the task, waiting until it is done.

The `task_coroutine()` finishes its sleep, resumes, and then terminates.

The `main()` coroutine then resumes and checks the done status one final time. This time the task is done as it has completely finished.

This example highlights that a scheduled and suspended task is not done and that a task is not done until the coroutine is has terminated.

```
>task done: False
executing the task
>task done: False
>task done: True
```

Next, let's explore the done status of a task that fails with an exception.

7.3 Example Exception That Causes A Task To Be Done

We can explore the status of a task where the wrapped coroutine fails with an unhandled exception.

In this example, we will update the task coroutine to sleep for less time and to then raise an `Exception`.

This will terminate the coroutine and cause the task to be marked as done.

The complete example is listed below.

```
# SuperFastPython.com
# example of checking the status of a failed task
import asyncio

# define a coroutine for a task
async def task_coroutine():
    # report a message
    print('executing the task')
    # block for a moment
    await asyncio.sleep(0.5)
    # raise an exception
    raise Exception('Something bad happened')

# custom coroutine
async def main():
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # check if it is done
    print(f'>task done: {task.done()}')
    # wait a moment
    await asyncio.sleep(0.1)
    # check if it is done
    print(f'>task done: {task.done()}')
    # wait for the task to complete
    await asyncio.sleep(0.5)
    # check if it is done
    print(f'>task done: {task.done()}')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and runs it as the entry point to the asyncio program.

The `main()` coroutine runs, first creating an instance of the `task_coroutine()` coroutine and then using it to create and schedule a task.

The `main()` coroutine then checks if the new task is done, which it is not because it has just been scheduled. It then sleeps, giving the task an opportunity to run.

The task begins executing, reporting a message, and then sleeping.

The `main()` coroutine then resumes and checks if the coroutine is done. At this point, the `task_coroutine()` is still running, yet suspended in a call to sleep.

The `main()` coroutine then sleeps for a little longer. We don't await the task, because it will raise an exception that will need to be handled in our `main()` coroutine.

The `task_coroutine()` finishes its sleep and then resumes raising an exception. This terminates the task and marks it as done.

The `main()` coroutine then continues on and checks the done status one final time. This time the task is done.

The event loop is exited and notices that an exception was raised in a task that was never retrieved. It then logs the exception using a default logger, e.g. to standard output.

This example highlights that an exception raised in a coroutine wrapped in a task will cause the task to be done.

```
>task done: False
executing the task
>task done: False
>task done: True
Task exception was never retrieved
Exception: Something bad happened
```

Next, let's look at how we might check if a task is canceled.

7.4 Example Of Checking If A Task Was Canceled

We can explore the case of explicitly canceling a task, then confirming that the status of the task is marked as canceled.

Calling the `cancel()` method on a task will request that the task cancel, and in most cases will cancel the task. We will explore this in detail in the next chapter.

In this example, we can update the previous example to cancel the task while it is running, wait for the task to be done after being canceled, then check its status.

We expect that the task will not be marked as canceled before we cancel it, and will be marked as canceled and done after it is canceled.

The complete example is listed below.

```
# SuperFastPython.com
# example of checking if a task was canceled
import asyncio

# define a coroutine for a task
async def task_coroutine():
    # report a message
```

```
    print('executing the task')
    # block for a moment
    await asyncio.sleep(1)

# custom coroutine
async def main():
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # check if it is canceled
    print(f'>task canceled: {task.cancelled()}')
    # give the task a chance to run
    await asyncio.sleep(0.1)
    # cancel the task
    task.cancel()
    # wait for the task to be done
    await asyncio.sleep(0.1)
    # check if the task is canceled
    print(f'>task canceled: {task.cancelled()}')
    # check if the task is done
    print(f'>task done: {task.done()}')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and runs it as the entry point to the asyncio program.

The `main()` coroutine runs, first creating an instance of the `task_coroutine()` coroutine and then using it to create and schedule a task.

The `main()` coroutine then checks if the new task is canceled, which it is not because it has just been scheduled. It is then suspended, sleeping for a fraction of a second.

This gives the task an opportunity to execute, report its message and sleep itself.

The `main()` coroutine resumes and then cancels the task.

It then sleeps again, suspending and allowing the task to execute and handle the request to be canceled.

The task resumes and completes its cancellation, e.g. a special `CancelledError` exception is raised in the coroutine and is not handled. The task is both done and marked as canceled.

The `main()` coroutine then checks the canceled status one final time, confirming that indeed the done task is now marked as canceled. It also reports the done status, confirming that the task is also done.

This highlights that a task is only marked as canceled if it is requested to cancel and this

request to cancel succeeds in terminating the task, marking it as done.

```
>task canceled: False
executing the task
>task canceled: True
>task done: True
```

7.5 Takeaways

You now know how to check the status of an asyncio task.

Specifically, you know:

- How to check if an asyncio task is running or is done.
- How to check if a task was canceled or completed normally.
- Understand the conditions under which we may consider that a task is done.

7.5.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

7.5.2 Next

In the next tutorial, we will explore how to cancel asyncio tasks.

Chapter 8

Task Cancellation

Asynchronous tasks run independently in the asyncio event loop.

We may need to stop or cancel a task from executing after it has started. This may be for many reasons, such as we no longer require the result, or the running task may negatively impact the program.

In this tutorial, you will discover how to cancel asyncio tasks.

After completing this tutorial, you will know:

- How to request that a task cancel.
- What happens when a task is canceled and the conditions under which a canceled task may not stop.
- How a message can be provided to a canceled task.

Let's get started.

8.1 How To Cancel A Task

A running task in asyncio can be canceled.

Once canceled, we may need to handle the `CancelledError` exception raised to cancel the task.

8.1.1 Cancel A Task

We can cancel an asyncio task via the `cancel()` method on the `asyncio.Task` instance.

The `cancel()` method returns `True` if the request to cancel was successful, or `False` otherwise.

For example:

```
...
# request a task to cancel
request_successful = task.cancel()
```

The `cancel()` method can also take a message argument which will be used in the content of the `CancelledError` raised in the target task.

For example:

```
...
# request a task to cancel
request_successful = task.cancel('No longer needed')
```

If the task is already done, it cannot be canceled and the `cancel()` method will return `False` and the task will not have the status of canceled.

If the task is not done, the `cancel()` method will succeed and return `True`, the task itself may or may not actually cancel after this point.

8.1.2 Handle Cancellation Exception

The next time the canceled task is given an opportunity to run by the event loop, a `CancelledError` exception will be raised at the point the task resumes.

If the `CancelledError` exception is not handled within the task's coroutine, the exception will bubble-up and the task will terminate with a canceled status. It will be canceled. Otherwise, if the `CancelledError` exception is handled within the task's coroutine, the task will not be canceled.

It is generally bad practice for a task to consume a `CancelledError` exception, as it breaks the contract with the `cancel()` method that assumes the task will be canceled.

If a task is required to clean-up when it is canceled, this can be achieved with a try-finally block within the body of a task, or the `CancelledError` exception can be handled and re-raised.

The `CancelledError` exception will be propagated to the caller if the caller awaits the task or calls the `result()` method. This is probably the most appropriate place to consume or suppress the `CancelledError` exception.

Now that we know how to cancel a task, let's look at some worked examples.

8.2 Example Of Canceling A Running Task

We can explore how to cancel a running task.

In this example, we define a task coroutine that reports a message and then blocks for a moment.

We then define the main coroutine that is used as the entry point into the asyncio program. It reports a message, creates and schedules the task, then waits for the task to cancel.

This is a common idiom when canceling a task, called *cancel and wait*.

The complete example is listed below.

```
# SuperFastPython.com
# example of canceling a running task
import asyncio

# define a coroutine for a task
async def task_coroutine():
    # report a message
    print('executing the task')
    # block for a moment
    await asyncio.sleep(1)

# custom coroutine
async def main():
    # report a message
    print('main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # wait a moment, allow the task to run
    await asyncio.sleep(0.1)
    # cancel the task
    was_cancelled = task.cancel()
    # report whether the cancel request was successful
    print(f'was canceled: {was_cancelled}')
    try:
        # wait for the task to cancel
        await task
    except asyncio.CancelledError:
        pass
    # check the status of the task
    print(f'canceled: {task.cancelled()}')
    # report a final message
    print('main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example starts the asyncio event loop and executes the `main()` coroutine.

The `main()` coroutine reports a message, then creates and schedules the task coroutine.

It then suspends and awaits a moment to allow the task coroutine to begin running.

The task runs, reports a message and sleeps for a while.

The `main()` coroutine resumes and cancels the task. It reports that the request to cancel the task was successful. It then awaits the task to be done.

The `task_coroutine()` resumes and a `CancelledError` exception is raised that causes the task to fail and be done.

The `main()` coroutine resumes. The `CancelledError` is propagated to the main() coroutine, is consumed, and suppressed with the try-except structure. The `main()` coroutine then reports whether the task has the status of canceled. In this case, it does.

This example highlights the normal case of canceling a running task and the use of the *cancel and wait* idiom.

We can see that if we adopt this idiom, we assume that the target task will be done sometime after a request to cancel. If a task consumes the `CancelledError` exception, it breaks the expectation made by the task's `cancel()` method.

```
main coroutine started
executing the task
was canceled: True
canceled: True
main coroutine done
```

Next, let's take a look at how we might cancel a scheduled task.

8.3 Example Of Canceling A Scheduled Task

A scheduled task is a task that has been created and scheduled in the event loop but has not yet had an opportunity to run.

That is, the task is not yet running.

A scheduled task can be canceled in the same manner as a running task.

If a scheduled task is canceled, it will raise a `CancelledError` exception when it is given an opportunity to run, but before the body of the task is executed.

This means that a task cannot consume a request to cancel that is made prior to the task's first opportunity to execute. This is a subtle point and can trip up beginners wondering why their tasks are failing to consume a request to cancel.

The example below updates the previous example so that the scheduled task is canceled before it has an opportunity to run.

```
# SuperFastPython.com
# example of canceling a scheduled task
import asyncio

# define a coroutine for a task
async def task_coroutine():
    # report a message
    print('executing the task')
```

```
    # block for a moment
    await asyncio.sleep(1)

# custom coroutine
async def main():
    # report a message
    print('main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # cancel the task before it has had a chance to run
    was_cancelled = task.cancel()
    # report whether the cancel request was successful
    print(f'was canceled: {was_cancelled}')
    # wait a moment
    await asyncio.sleep(0.1)
    # check the status of the task
    print(f'canceled: {task.cancelled()}')
    # report a final message
    print('main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example starts the asyncio event loop and executes the `main()` coroutine.

The `main()` coroutine reports a message, then creates and schedules the task coroutine.

The `main()` coroutine then cancels the task before it has had an opportunity to execute. It reports that the request to cancel the task was successful.

The `main()` coroutine then sleeps for a moment to allow the task to respond to the request to be canceled.

The `task_coroutine()` runs and a `CancelledError` exception is raised before the body of the task is reached. This exception causes the task to fail and be done.

The `main()` coroutine resumes and reports whether the task has the status of canceled. In this case it does.

This example highlights the case of canceling a scheduled task. It highlights that both a scheduled task can be canceled in the same manner as a running task.

```
main coroutine started
was canceled: True
canceled: True
main coroutine done
```

Next, we will look at how to handle a request to be canceled within a task.

8.4 Example Of A Task Handling The Request To Cancel

A task can handle a request to being canceled.

This can be achieved by expecting and handling a `CancelledError` exception.

If a task's coroutine handles the `CancelledError`, then the task will not be canceled, may continue to run, and will not be marked with the canceled status meaning the `cancelled()` method will return `False`.

This means that although the `cancel()` method may return `True` indicating that the request to cancel was successful, it does not mean that will be canceled, only that the task has the potential to be canceled.

The example below gives an example of canceling a task while it is running and the task handling the request for canceling by reporting a message.

```
# SuperFastPython.com
# example of a task handling the request to be canceled
import asyncio

# define a coroutine for a task
async def task_coroutine():
    try:
        # report a message
        print('executing the task')
        # block for a moment
        await asyncio.sleep(1)
    except asyncio.CancelledError:
        print('Received a request to cancel')

# custom coroutine
async def main():
    # report a message
    print('main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # wait a moment
    await asyncio.sleep(0.1)
    # cancel the task
    was_cancelled = task.cancel()
    # report whether the cancel request was successful
    print(f'was canceled: {was_cancelled}')
    # wait a moment
    await asyncio.sleep(0.1)
    # check the status of the task
```

```
    print(f'canceled: {task.cancelled()}')
    # report a final message
    print('main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example starts the asyncio event loop and executes the `main()` coroutine.

The `main()` coroutine reports a message, then creates and schedules the task coroutine.

It then suspends and waits a moment to allow the task coroutine to begin running.

The task runs, reports a message, and sleeps for a while.

The `main()` coroutine resumes and cancels the task. It reports that the request to cancel the task was successful.

It then sleeps for a moment to allow the task to respond to the request to be canceled.

The `task_coroutine()` resumes and a `CancelledError` exception is raised. The task catches the exception and reports a message. It could then continue running. In this case, there are no further operations to perform

The `main()` coroutine resumes and reports whether the task has the status of canceled.

In this case, the task is not marked as canceled. This is because the `CancelledError` exception did not unravel the coroutine, instead, it was caught and handled.

This example highlights that the `cancel()` method only requests a task cancel, it does not ensure that it will be canceled.

```
main coroutine started
executing the task
was canceled: True
Received a request to cancel
canceled: False
main coroutine done
```

Next, we will look at how to pass a message to a task when requesting it to cancel.

8.5 Example Of A Canceling A Task With A Message

The `cancel()` method allows a message to be passed to the task that is being canceled.

This allows the caller to interact with the callee. For example, the string message could provide some indication of why the task must be canceled. The target coroutine could catch the `CancelledError` exception and decide whether to cancel or not based on that information.

We can explore how to send a cancel message to a task via the `cancel()` method.

The example below updates the above example where the task handles the request to cancel and does not cancel. In this case, it reports the message that was passed by the caller.

```
# SuperFastPython.com
# example of canceling a running task with a message
import asyncio

# define a coroutine for a task
async def task_coroutine():
    try:
        # report a message
        print('executing the task')
        # block for a moment
        await asyncio.sleep(1)
    except asyncio.CancelledError as e:
        print(f'received request to cancel with: {e}')

# custom coroutine
async def main():
    # report a message
    print('main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # wait a moment
    await asyncio.sleep(0.1)
    # cancel the task
    was_cancelled = task.cancel('Stop Right Now')
    # report whether the cancel request was successful
    print(f'was canceled: {was_cancelled}')
    # wait a moment
    await asyncio.sleep(0.1)
    # check the status of the task
    print(f'canceled: {task.cancelled()}')
    # report a final message
    print('main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example starts the asyncio event loop and executes the `main()` coroutine.

The `main()` coroutine reports a message, then creates and schedules the task coroutine.

It then suspends and waits a moment to allow the task coroutine to begin running.

The task runs, reports a message, and sleeps for a while.

The `main()` coroutine resumes and cancels the task and passes it a unique message. It reports that the request to cancel the task was successful.

It then sleeps for a moment to allow the task to respond to the request to be canceled.

The `task_coroutine()` resumes and a `CancelledError` exception is raised. The task catches the exception and reports the exception. This includes the specific message provided by the caller in the request to cancel.

The `main()` coroutine resumes and reports whether the task has the status of canceled.

In this case, the task is not marked as canceled. This is because the `CancelledError` exception did not unravel the coroutine, instead, it was caught and handled.

This example highlights that we can pass custom messages to tasks when they are requested to cancel. This message could be used in various ways, not least logging.

```
main coroutine started
executing the task
was canceled: True
received request to cancel with: Stop Right Now
canceled: False
main coroutine done
```

8.6 Takeaways

You now know how to cancel asyncio tasks.

Specifically, you know:

- How to request that a task cancel.
- What happens when a task is canceled and the conditions under which a canceled task may not stop.
- How a message can be provided to a canceled task.

8.6.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html
- Asyncio Exceptions.
  https://docs.python.org/3/library/asyncio-exceptions.html

8.6.2 Next

In the next tutorial, we will explore how to get results from asyncio tasks.

Chapter 9

Task Results

A coroutine may return a result directly via a return value.

Asyncio tasks that execute a coroutine run asynchronously. Therefore we need a way to retrieve results from coroutines executed by independently run coroutines.

In this tutorial, you will discover how to get a result from an asyncio task.

After completing this tutorial, you will know:

- How to retrieve a result from an asyncio task.
- What happens to the result if the task fails or is canceled.
- What happens if we try to retrieve a result from a running task.

Let's get started.

9.1 How To Get A Task Result

The result of a task is the return value from the coroutine that it executes.

If the coroutine does not explicitly return a value, then the result of the task will be `None`.

9.1.1 Retrieve Task Result

We can get the result of a task via the `result()` method.

This method returns the return value of the coroutine.

For example:

```
...
# get the return value from the wrapped coroutine
value = task.result()
```

9.1.2 Retrieve Result From Failed Task

If the coroutine fails with an unhandled exception, it is re-raised when calling the `result()` method and may need to be handled.

For example:

```
...
try:
    # get the return value from the wrapped coroutine
    value = task.result()
except Exception:
    # task failed and there is no result
```

9.1.3 Retrieve Result From Canceled Task

If the task was canceled, then a `CancelledError` exception is raised when calling the `result()` method and may need to be handled.

For example:

```
...
try:
    # get the return value from the wrapped coroutine
    value = task.result()
except asyncio.CancelledError:
    # task was canceled
```

As such, it is a good idea to check if the task was canceled first.

For example:

```
...
# check if the task was not canceled
if not task.cancelled():
    # get the return value from the wrapped coroutine
    value = task.result()
else:
    # task was canceled
```

9.1.4 Retrieve Result From Running Task

If the task is not yet done, then an `InvalidStateError` exception is raised when calling the `result()` method and may need to be handled.

For example:

```
...
try:
    # get the return value from the wrapped coroutine
    value = task.result()
```

```
except asyncio.InvalidStateError:
    # task is not yet done
```

As such, it is a good idea to check if the task is done first.

For example:

```
...
# check if the task is not done
if not task.done():
    await task
# get the return value from the wrapped coroutine
value = task.result()
```

Notice, we awaited the task first, then retrieved the result.

We can do this in a single step as awaiting a task will retrieve its result once it is done.

The `await` expression on a task automatically retrieves the task's result.

For example:

```
...
# get task result and wait for it to be done if needed
value = await task
```

Now that we know how to get the return value from a task, let's look at some worked examples.

9.2 Example Of Getting A Result From A Done Task

We can explore how to get a return value result from a successfully done task.

In this example, we define a task coroutine that reports a message, suspends for a moment to simulate effort, then returns a value.

We then define the main coroutine that is used as the entry point into the asyncio program. It reports a message, creates and schedules the task, then awaits the task to be completed. Once completed, it retrieves the result from the task and reports it.

The complete example is listed below.

```
# SuperFastPython.com
# example of getting a result from a done task
import asyncio

# define a coroutine for a task
async def task_coroutine():
    # report a message
    print('executing the task')
    # block for a moment
```

```
    await asyncio.sleep(1)
    # return a value
    return 99

# custom coroutine
async def main():
    # report a message
    print('main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # wait for the task to complete
    await task
    # get the result
    value = task.result()
    # report the task result
    print(f'result: {value}')
    # report a final message
    print('main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example starts the asyncio event loop and executes the `main()` coroutine.

The `main()` coroutine reports a message, then creates and schedules the task coroutine.

It then suspends and awaits the task to be completed.

The task runs, reports a message, and sleeps for a moment before returning a value and terminating normally.

The `main()` coroutine resumes and retrieves the return value result from the task, which is then reported.

This example highlights the normal case of retrieving a return value from a successful task.

```
main coroutine started
executing the task
result: 99
main coroutine done
```

Note, that we could achieve a similar result in the `main()` coroutine by awaiting the task and assigning the return value in one line.

For example:

```
# custom coroutine
async def main():
    # report a message
```

```
    print('main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # wait for the task to complete and get result
    value = await task
    # report the task result
    print(f'result: {value}')
    # report a final message
    print('main coroutine done')
```

The difference is that it requires that the caller suspend until the task is complete and assigns the result in a single line, as opposed to splitting these concerns.

We would prefer to use the `result()` method on the task when we know that the task is already done and the waiting was performed by other means, such as the `asyncio.wait()` function, which we will learn more about in a later chapter.

Next, we will look at attempting to get a result from a task that fails with an exception.

9.3 Example Of Getting A Result From A Failed Task

If the task fails with an unhandled exception, the exception will be re-raised when calling the `result()` method on the task to get the result.

As such, we may need to handle possible exceptions when getting task results.

We can explore getting a result from a task that failed with an unhandled exception.

In this example, we can update the task coroutine to explicitly raise an exception that is not handled.

This will cause the task's coroutine to fail.

The main coroutine will sleep to wait for the task to be completed. This is to avoid using the `await` expression which will propagate the exception back to the caller.

Once the task is done, the main coroutine will attempt to retrieve the return value and handle the exception that is re-raised.

The complete example is listed below.

```
# SuperFastPython.com
# example of getting a result from a failed task
import asyncio

# define a coroutine for a task
async def task_coroutine():
    # report a message
    print('executing the task')
```

```
    # block for a moment
    await asyncio.sleep(1)
    # fail with an exception
    raise Exception('Something bad happened')
    # return a value (never reached)
    return 99

# custom coroutine
async def main():
    # report a message
    print('main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # wait for the task to complete
    await asyncio.sleep(1.1)
    try:
        # get the result
        value = task.result()
        print(f'result: {value}')
    except Exception as e:
        print(f'Failed with: {e}')
    # report a final message
    print('main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example starts the asyncio event loop and executes the `main()` coroutine.

The `main()` coroutine reports a message, then creates and schedules the task coroutine.

It then suspends and awaits the task to be completed.

The task runs, reports a message, and sleeps for a moment. The task resumes and raises an exception.

The exception does not terminate the application or the asyncio event loop.

Instead, the exception is captured by the asyncio event loop and stored in the task.

The `main()` coroutine resumes and then attempts to retrieve the result from the task. This fails and the exception that was raised and not handled in the Task's wrapped coroutine is re-raised in the caller.

The `main()` coroutine catches the exception and reports its details.

```
main coroutine started
executing the task
```

```
Failed with: Something bad happened
main coroutine done
```

Next, we can look at the case of attempting to get a task result from a canceled task.

9.4 Example Of Getting A Result From A Canceled Task

We cannot retrieve a result from a canceled task.

Although a canceled task is done, a result will not be available and cannot be retrieved.

Instead, a `CancelledError` exception is raised when calling the `result()` method if the task was canceled.

The example below updates the previous example to create and schedule the task as before, then wait a moment. It then cancels the task, waits a moment for the task to be canceled, then attempts to get the result.

This is expected to fail as the `result()` method will re-raise the `CancelledError` exception from the wrapped coroutine that was used to cancel the task.

The complete example is listed below.

```
# SuperFastPython.com
# example of getting a result from a canceled task
import asyncio

# define a coroutine for a task
async def task_coroutine():
    # report a message
    print('executing the task')
    # block for a moment
    await asyncio.sleep(1)
    # return a value (never reached)
    return 99

# custom coroutine
async def main():
    # report a message
    print('main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # wait for the task to complete
    await asyncio.sleep(0.1)
    # cancel the task
```

```
    task.cancel()
    # wait a moment for the task to be canceled
    await asyncio.sleep(0.1)
    # get the result
    value = task.result()
    print(f'result: {value}')
    # report a final message
    print('main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example starts the asyncio event loop and executes the `main()` coroutine.

The `main()` coroutine reports a message, then creates and schedules the task coroutine.

It then suspends and sleeps for a moment.

The `main()` coroutine resumes and cancels the task. It then suspends and waits a moment for the task to respond to the request for being canceled.

The task is canceled by raising a `CancelledError` within the wrapped coroutine.

The `main()` coroutine resumes and attempts to retrieve the result.

This fails and the `CancelledError` exception is re-raised in the caller.

This breaks the event loop in this case.

```
main coroutine started
executing the task
Traceback (most recent call last):
  ...
asyncio.exceptions.CancelledError
During handling of the above exception,
    another exception occurred:
Traceback (most recent call last):
  ...
asyncio.exceptions.CancelledError
During handling of the above exception,
    another exception occurred:
Traceback (most recent call last):
  ...
asyncio.exceptions.CancelledError
```

This highlights the importance of either checking if a task was canceled before getting the result, or getting the result in a manner that anticipates a `CancelledError`.

For example:

```
# custom coroutine
async def main():
    # report a message
    print('main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # wait for the task to complete
    await asyncio.sleep(0.1)
    # cancel the task
    task.cancel()
    # wait a moment for the task to be canceled
    await asyncio.sleep(0.1)
    # get the result
    try:
        value = task.result()
        print(f'result: {value}')
    except asyncio.CancelledError:
        print('Unable to get the result, canceled')
    # report a final message
    print('main coroutine done')
```

9.5 Takeaways

You now know how to get a result from an asyncio task.

Specifically, you know:

- How to retrieve a result from an asyncio task.
- What happens to the result if the task fails or is canceled.
- What happens if we try to retrieve a result from a running task.

9.5.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

9.5.2 Next

In the next tutorial, we will explore how to retrieve exceptions from asyncio tasks.

Chapter 10

Task Exceptions

An asyncio task may fail with an unhandled exception.

The exception will unwind the task, although may not impact other tasks or the broader asyncio program.

As such, we need a way of checking if a task has failed and a way of retrieving any unhandled exceptions in the task, if they occurred.

In this tutorial, you will discover exception handling in asyncio tasks.

After completing this tutorial, you will know:

- How to check if a task failed due to an unhandled exception.
- How to retrieve the exception from a task and what happens if we get the exception while the task is running.
- How and when the exception in the task may be propagated to another coroutine or task.

Let's get started.

10.1 How To Check For Exceptions In Tasks

A coroutine within a task may raise an exception that is not handled.

This will cause the task to fail and terminate.

We are able to check if a task has failed with an unhandled exception.

10.1.1 Check For An Exception

We can retrieve an unhandled exception in a task via the `exception()` method.

For example:

```
...
# get the exception raised by a task
exception = task.exception()
```

If an unhandled exception was not raised in the task, then a value of `None` is returned.

10.1.2 Check For An Exception In Canceled Task

Asyncio tasks can be canceled, which raises a `CancelledError` exception in the task. These exceptions are not returned when calling the `exception()` method.

This means that if the task was canceled, then a `CancelledError` exception will be re-raised when calling the `exception()` method and may need to be handled.

For example:

```
...
try:
    # get the exception raised by a task
    exception = task.exception()
except asyncio.CancelledError:
    # task was canceled
```

As such, it is a good idea to check if the task was canceled first.

For example:

```
...
# check if the task was not canceled
if not task.cancelled():
    # get the exception raised by a task
    exception = task.exception()
else:
    # task was canceled
```

10.1.3 Check For An Exception In Running Task

If the task is not yet done, then an `InvalidStateError` exception is raised when calling the `exception()` method and may need to be handled.

For example:

```
...
try:
    # get the exception raised by a task
    exception = task.exception()
except asyncio.InvalidStateError:
    # task is not yet done
```

As such, it is a good idea to check if the task is done first.

For example:

```
...
# check if the task is not done
if not task.done():
    await task
# get the exception raised by a task
exception = task.exception()
```

10.1.4 Never-Retrieved Exceptions

If a task fails with an unhandled exception and the exception is not retrieved via the `exception()` method, then it is generally referred to as a "*never-retrieved*" exception.

The asyncio event loop will automatically report never-retrieved exceptions as part of shutting down an asyncio program.

The event loop also provides a way to handle never retrieved exceptions via the `loop.set_exception_handler()` method.

We can define a handler function that takes an event loop and context as arguments from which an exception can be retrieved.

For example:

```
# define an exception handler
def exception_handler(loop, context):
    # get the exception
    ex = context['exception']
    # log details
    print(f'Got exception {ex}')
```

We can configure the asyncio event loop to call this function for each never-retrieved exception, as the event loop is being shut down.

For example:

```
...
# get the event loop
loop = asyncio.get_running_loop()
# set the exception handler
loop.set_exception_handler(exception_handler)
```

Next, let's look at when an unhandled exception in a task is propagated to the caller.

10.2 When Are Task Exceptions Propagated To The Caller

Exceptions that occur within a task can be propagated to the caller.

This can happen in two situations, they are:

1. When the caller awaits the task.
2. When the caller gets the result from the task.

When a coroutine awaits a task that raises an unhandled exception, the exception is propagated to the caller.

For example:

```
...
# wait for the task to finish
await task
```

Therefore, if an unhandled exception is possible in a task's coroutine, it may need to be handled when awaiting the task.

For example:

```
...
try:
    # wait for the task to finish
    await task
except Exception as e:
    # ...
```

Similarly, if the task is done and the caller tempts to retrieve the return value from the task via the `result()` method, an unhandled exception is propagated.

For example:

```
...
# get the return value from the task
value = task.result()
```

Therefore, if an unhandled exception is possible in a task's coroutine, it may need to be handled when awaiting the task.

For example:

```
...
try:
    # get the return value from the task
    value = task.result()
except Exception as e:
    # ...
```

Now that we know when exceptions in tasks are propagated, let's look at some worked examples of checking for and handling exceptions in tasks.

10.3 Example Of Checking For A Task Exception

We can explore getting an exception from a task that failed with an unhandled exception.

This is the intended use case for the `exception()` method. In this example, we can define a task coroutine to explicitly raise an exception that is not handled.

This will cause the task coroutine to fail.

The main coroutine will sleep to wait for the task to be completed. This is to avoid using the `await` expression which will propagate the exception back to the caller.

Once the task is done, the main coroutine will retrieve and report the exception raised in the task.

The complete example is listed below.

```
# SuperFastPython.com
# example of getting an exception from a failed task
import asyncio

# define a coroutine for a task
async def task_coroutine():
    # report a message
    print('executing the task')
    # block for a moment
    await asyncio.sleep(1)
    # raise an exception
    raise Exception('Something bad happened')

# custom coroutine
async def main():
    # report a message
    print('main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # wait for the task to complete
    await asyncio.sleep(1.1)
    # get the exception
    ex = task.exception()
    # report the details of the exception
    print(f'exception: {ex}')
    # report a final message
    print('main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example starts the asyncio event loop and executes the `main()` coroutine.

The `main()` coroutine reports a message, then creates and schedules the task coroutine.

It then suspends and awaits the task to be completed.

The task runs, reports a message, and sleeps for a moment. The task resumes and raises an exception.

The exception does not terminate the application or the asyncio event loop.

Instead, the exception is captured by the asyncio event loop and stored in the task.

The `main()` coroutine resumes and then retrieves the exception from the task, which is reported.

```
main coroutine started
executing the task
exception: Something bad happened
main coroutine done
```

Next, let's look at how we might handle an exception propagated by awaiting a task.

10.4 Example Of Handling A Task Exception

Awaiting a task that fails with an exception will cause the exception to be propagated to the caller.

As such, awaiting a task may require that the possible exceptions be handled.

The example below demonstrates this with a task that fails with an exception that is awaited in a main coroutine that expects and then handles the exception.

```
# SuperFastPython.com
# example of handling a task exception when await
import asyncio

# define a coroutine for a task
async def task_coroutine():
    # report a message
    print('executing the task')
    # block for a moment
    await asyncio.sleep(1)
    # fail with an exception
    raise Exception('Something bad happened')

# custom coroutine
async def main():
    # report a message
    print('main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
```

```
    try:
        # wait for the task to complete
        await task
    except Exception as e:
        # report the exception
        print(f'Failed with: {e}')
    # report a final message
    print('main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example starts the asyncio event loop and executes the `main()` coroutine.

The `main()` coroutine reports a message, then creates and schedules the task coroutine.

It then suspends and awaits the task to be completed. Importantly, the `main()` coroutine awaits the task within a try-except block.

The task runs, reports a message and sleeps for a moment, and then fails with an exception.

The `main()` coroutine resumes and handles the exception that was raised in the wrapped coroutine. The exception is propagated to the caller, caught, and the details are reported.

This highlights that we may need to handle unhandled exceptions because they can be propagated back to any coroutines waiting on the task.

```
main coroutine started
executing the task
Failed with: Something bad happened
main coroutine done
```

Next, we will look at how to handle task exceptions propagated to the caller when getting task results.

10.5 Example Of Handling A Task Exception With Result

We can get the return value from a task via the `result()` method.

Care must be taken with this method because any exception that was raised in the task's coroutine that was not handled will be propagated back and re-raised in the caller.

We can demonstrate this with a worked example.

The task coroutine returns a value, but the line is never reached because it fails with an exception.

The main coroutine attempts to retrieve the result from the task and handles the exception that may be raised and propagated.

```
# SuperFastPython.com
# example of handling an exception when getting result
import asyncio

# define a coroutine for a task
async def task_coroutine():
    # report a message
    print('executing the task')
    # block for a moment
    await asyncio.sleep(1)
    # fail with an exception
    raise Exception('Something bad happened')
    # return a value (never reached)
    return 100

# custom coroutine
async def main():
    # report a message
    print('main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # wait for the task to complete
    await asyncio.sleep(1.1)
    try:
        # get the result
        value = task.result()
    except Exception as e:
        print(f'Failed with: {e}')
    # report a final message
    print('main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example starts the asyncio event loop and executes the `main()` coroutine.

The `main()` coroutine reports a message, then creates and schedules the task coroutine.

It then suspends and sleeps a moment to allow the task to be completed.

The task runs, reports a message and sleeps for a moment, and then fails with an exception.

The `main()` coroutine resumes and attempts to retrieve the return value from the task.

This fails and the unhandled exception raised in the task's coroutine is re-raised in the caller.

The `main()` coroutine handles the exception, catching it and reporting the details.

This highlights that we may need to handle unhandled exceptions when getting task results because they can be propagated back to any coroutines waiting on the task.

```
main coroutine started
executing the task
Failed with: Something bad happened
main coroutine done
```

10.6 Takeaways

You now know how to handle exceptions in asyncio tasks.

Specifically, you know:

- How to check if a task failed due to an unhandled exception.
- How to retrieve the exception from a task and what happens if we get the exception while the task is running.
- How and when the exception in the task may be propagated to another coroutine or task.

10.6.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html
- Asyncio Exceptions.
  https://docs.python.org/3/library/asyncio-exceptions.html

10.6.2 Next

In the next tutorial, we will explore how to add and use done callback functions on asyncio tasks.

Chapter 11

Task Done Callbacks

We typically need to perform some activity when an asyncio task is done.

This may be to collect and store results or to clean up some resource that is no longer needed.

Performing these actions can be challenging given the asynchronous nature of tasks. The solution is to use a callback function, triggered when a task is done.

In this tutorial, you will discover how to use asyncio task done callback functions.

After completing this tutorial, you will know:

- How to add a done callback function to an asyncio task.
- How to add multiple callback functions and how to remove functions that are no longer needed.
- What happens if a callback function fails with an exception or is added after a task is already done.

Let's get started.

11.1 How To Use Callback With A Task

We can configure a task to automatically execute one or more regular Python functions once the task is done.

These are called callback functions. Because they are only executed when the task is finished executing, they are referred to as "done" callback functions.

Recall, a task is done if it finishes normally, returns a value, is canceled, or fails with an unhandled exception.

11.1.1 Add Done Callback Function

We can add a done callback function to a task via the `add_done_callback()` method.

This method takes the name of a function to call when the task is done.

The callback function must take the `Task` instance as an argument, which will be the task instance on which the callback was added.

For example:

```
# done callback function
def handle(task):
    print(task)

...
# register a done callback function
task.add_done_callback(handle)
```

Notice that the callback function is a regular Python function, not a coroutine. It is not awaited by the event loop, but instead called after the task is done.

Recall that a task may be done when its coroutine finishes normally, when an unhandled exception is raised, or when the task is canceled.

The `add_done_callback()` method can be used to add or register as many done callback functions as we like.

If a done callback function is added to a task that is already done, then the callback will execute immediately.

If an exception occurs within a done callback function it does not impact the task or other done callback functions that also may have been registered. The exception is handled by the event loop's default exception handler.

11.1.2 Remove Done Callback Function

We can also remove or de-register a callback function via the `remove_done_callback()` function.

For example:

```
...
# remove a done callback function
task.remove_done_callback(handle)
```

Now that we know how to add and remove done callback functions, let's look at some worked examples.

11.2 Example Of Adding A Done Callback Function

We can explore how to add a done callback function to a task.

In this example, we will define a task coroutine that reports a message and sleeps for a moment.

We will then define a main coroutine that we will use as the entry point to the program. It will report a message, create and schedule the task, then add a done callback function to the task that reports a simple message.

The main coroutine then waits for the task to be completed.

The complete example is listed below.

```
# SuperFastPython.com
# example of adding a done callback function to a task
import asyncio

# custom done callback function
def callback(task):
    # report a message
    print('Task is done')

# define a coroutine for a task
async def task_coroutine():
    # report a message
    print('executing the task')
    # block for a moment
    await asyncio.sleep(1)

# custom coroutine
async def main():
    # report a message
    print('main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # add a done callback function
    task.add_done_callback(callback)
    # wait for the task to complete
    await task
    # report a final message
    print('main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example starts the asyncio event loop and executes the `main()` coroutine.

The `main()` coroutine reports a message, then creates and schedules the task coroutine.

It then adds the done callback function to the task to be executed when the task is finished.

The `main()` coroutine then suspends and awaits the task to be completed.

The task runs, reports a message, and sleeps for a moment before terminating normally.

After the task completes, the done callback function is called by the event loop, reporting a message.

The `main()` coroutine resumes, reports its own final message and the program ends.

This example highlights how a done callback can be added to a task and that it is called automatically when the task is done.

```
main coroutine started
executing the task
Task is done
main coroutine done
```

Next, let's look at the case where we may want to add more than one done callback function.

11.3 Example Of Adding More Than One Callback

A task may have more than one done callback function registered to be called when it completes.

This can be achieved by calling the `add_done_callback()` function for each function to be registered.

We can explore how to register multiple callback functions to be called when a task finishes.

The example below updates the above example to define two callback functions. The first reports a generic message, and the second reports the details of the task itself.

Both callback functions are added to the task after it is created and scheduled, then called automatically once the task is done.

The complete example is listed below.

```
# SuperFastPython.com
# example of adding more than one done callback function
import asyncio

# custom done callback function
def callback1(task):
    # report a message
    print('Task is done')

# another custom done callback function
def callback2(task):
    # report a message
    print(f'Task: {task}')
```

```
# define a coroutine for a task
async def task_coroutine():
    # report a message
    print('executing the task')
    # block for a moment
    await asyncio.sleep(1)

# custom coroutine
async def main():
    # report a message
    print('main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # add a done callback function
    task.add_done_callback(callback1)
    # add another done callback function
    task.add_done_callback(callback2)
    # wait for the task to complete
    await task
    # report a final message
    print('main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example starts the asyncio event loop and executes the `main()` coroutine.

The `main()` coroutine reports a message, then creates and schedules the task coroutine.

It then adds both done callback functions to the task. We expect the callbacks to be executed by the asyncio event loop in the order that they are added.

The `main()` coroutine then suspends and awaits the task to be completed.

The task runs, reports a message, and sleeps for a moment before terminating normally.

After the task completes, the first done callback function is called by the event loop, reporting a message. Then the second callback function is called, reporting the details of the task itself.

The `main()` coroutine resumes, reports its own final message and the program ends.

This example highlights how we can add multiple done callback functions to a task.

```
main coroutine started
executing the task
Task is done
Task: <Task finished name='Task-2'
    coro=<task_coroutine() done, defined at ...>
```

```
    result=None>
main coroutine done
```

Next, let's look at an example of adding, then removing a done callback function.

11.4 Example Of Removing A Done Callback

A done callback function can be removed from a task.

This may be required if callbacks need to be executed conditionally and in some cases not executed at all.

The example below adds a callback function to a task after it is created but before it is running.

Then, after the task has been running for a moment, the done callback function is removed to ensure it is not executed when the task finishes.

The complete example is listed below.

```
# SuperFastPython.com
# example of adding and removing callback functions
import asyncio

# custom done callback function
def callback(task):
    # report a message
    print('Task is done')

# define a coroutine for a task
async def task_coroutine():
    # report a message
    print('executing the task')
    # block for a moment
    await asyncio.sleep(1)

# custom coroutine
async def main():
    # report a message
    print('main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # add a done callback function
    task.add_done_callback(callback)
    # wait a moment
    await asyncio.sleep(0.1)
```

```
    # remove the done callback function
    task.remove_done_callback(callback)
    # wait for the task to complete
    await task
    # report a final message
    print('main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example starts the asyncio event loop and executes the `main()` coroutine.

The `main()` coroutine reports a message, then creates and schedules the task coroutine.

It then adds the done callback function to the task to be executed when the task is finished.

The `main()` coroutine then suspends and awaits the task to be completed.

The task runs, reports a message, and sleeps for a moment. This gives the task time to start running.

It then resumes and removes the done callback function from the running task.

The `main()` coroutine then awaits the task to be completed.

The task completes and the done callback function is not executed, as we intended.

The `main()` coroutine resumes, reports its own final message and the program ends.

This example highlights how a done callback function can be removed from a running task before it is done.

```
main coroutine started
executing the task
main coroutine done
```

Next, let's look at what happens if we attempt to remove a callback that was not added.

11.5 Example Of Adding A Callback To A Done Task

A done callback function can be added after the task has already been completed.

The effect is that the done callback function will execute as soon as it is able.

The example below demonstrates this. The main coroutine creates and schedules a task, then waits for it to complete. Once the task is done, the main coroutine then adds a done callback function.

The complete example is listed below.

```
# SuperFastPython.com
# example of adding a callback to a done task
import asyncio

# custom done callback function
def callback(task):
    # report a message
    print('Task is done')

# define a coroutine for a task
async def task_coroutine():
    # report a message
    print('executing the task')
    # block for a moment
    await asyncio.sleep(1)

# custom coroutine
async def main():
    # report a message
    print('main coroutine started')
    # create and schedule the task
    task = asyncio.create_task(task_coroutine())
    # wait for the task to complete
    await task
    # add a done callback function
    task.add_done_callback(callback)
    # report a final message
    print('main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example starts the asyncio event loop and executes the `main()` coroutine.

The `main()` coroutine reports a message, then creates and schedules the task coroutine. It then suspends and awaits the task to be completed.

The task runs, reports a message, and sleeps for a moment before terminating normally.

After the task completes, the `main()` coroutine adds the done callback and then terminates.

Before the asyncio event loop terminates, it executes the done callback function added to the task, reporting its message.

This example highlights that we can add a done callback function to a task after it has been completed.

```
main coroutine started
executing the task
main coroutine done
Task is done
```

11.6 Takeaways

You now know how to use asyncio task done callback functions.

Specifically, you know:

- How to add a done callback function to an asyncio task.
- How to add multiple callback functions and how to remove functions that are no longer needed.
- What happens if an callback function fails with an exception or is added after a task is already done.

11.6.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

11.6.2 Next

In the next tutorial, we will explore the main task in asyncio and how to get the current task.

Chapter 12

Main And Current Task

The entry point into an asyncio program is called the main task.

This is an important and different task from other tasks, similar to the idea of the main thread or the main process of a Python program.

The `asyncio` module provides introspection tools to access the currently executing task. This can be used to retrieve details for the main task.

In this tutorial, you will discover the main task and how to retrieve the current task.

After completing this tutorial, you will know:

- What is the "main task" in an asyncio program.
- The special properties of the main task in asyncio programs and how to identify it.
- How to retrieve the currently executing asyncio task.

Let's get started.

12.1 What Is The Main Task (Main Coroutine)

An asyncio program is executed by explicitly starting the asyncio event loop.

Typically, this involves calling the `asyncio.run()` module function and providing a coroutine object to execute as the entry point to the program.

For example:

```
...
# start the event loop
asyncio.run(main())
```

The coroutine that is provided to `asyncio.run()` is called the "main coroutine".

- **Main Coroutine**: The coroutine provided to `asyncio.run()` when starting the asyncio event loop.

The coroutine is wrapped in an `asyncio.Task` object and scheduled for execution.

As such, it is also common to refer to the main coroutine as the "main task".

The main coroutine is synonymous with the *main thread* and the *main process*.

Recall that the main thread is the entry point into a program or the default thread, and has the name `MainThread`.

Also, recall that the main process is the process that encloses the main thread and is the first or main parent process created when the Python interpreter is started and has the name `MainProcess`.

Both the main thread and the main process have special properties, and so does the main coroutine. This is why we are interested in it.

12.1.1 Special Properties Of The Main Coroutine

The main coroutine has special properties compared to other asyncio tasks and coroutines.

- The main coroutine is the entry point into the asyncio event loop.
- When the main coroutine exits, all other tasks are canceled and the asyncio event loop is terminated.
- When a signal interrupt (SIGINT) is received by the program, the main task is canceled via the `cancel()` method.

The main coroutine is different in some ways from the main thread and main process.

For example:

- The main coroutine does not have a special name e.g. it is typically called `Task-1` as the first task created.
- The main coroutine does not have a special type, e.g. it is an `asyncio.Task` and can be canceled like any other task.

12.1.2 Most Important Property Of The Main Coroutine

The most important detail of the main coroutine is that when it is done, the event loop is closed.

Recall, that a coroutine can be "done" in a few ways:

- It may be completed normally.
- It may return early.
- It may raise an unhandled exception.
- It may be canceled.
- It may receive a signal.

Once the main coroutine is done, the asyncio event loop is terminated.

As part of the shutdown, all other tasks in the event loop are canceled.

This is important because if we want to immediately shut down the event loop on demand, we must do so by completing the main coroutine.

Now that we are familiar with the main coroutine, let's look at how we might access it via introspection tools.

12.2 How To Get The Current Task

The `asyncio` module provides introspection tools, including the ability to retrieve a handle on the currently executing task in the event loop.

12.2.1 Get Current Task

We can get the current task in an asyncio program via the `asyncio.current_task()` function.

This function will return a `Task` object for the task that is currently running.

For example:

```
...
# get the current task
task = asyncio.current_task()
```

This task may be:

- The main coroutine passed to `asyncio.run()`.
- A task created and scheduled within the asyncio program via `create_task()`.

A task may create and run another coroutine (e.g. not wrapped in a task). Getting the current task from within a coroutine will return a `Task` object for the running task, but not the coroutine that is currently running.

This means that we may always want to wrap coroutines in tasks to ensure that introspection tools like `current_task()` can "see" and access them.

Getting the current task can be helpful if a coroutine or task requires details about itself, such as the task name for logging.

It may also be helpful if a task wants to do something to itself, such as change or amend its name.

12.2.2 Get Current Coroutine

If we need access to the current coroutine, we can access the current task, then call the `get_coro()` method.

For example:

```
...
# get the current task
task = asyncio.current_task()
```

```
# get the current coroutine
coro = task.get_coro()
```

Now that we know how to get the current task, let's look at some worked examples.

12.3 Example Of Getting The Main Task

We can explore how to get a `Task` instance for the main coroutine used to start an asyncio program.

The example below defines a coroutine used as the entry point into the program. It reports a message, then gets the current task and reports its details.

This is an important example, as it highlights that all coroutines can be accessed as tasks within the asyncio event loop.

The complete example is listed below.

```
# SuperFastPython.com
# example of getting the task for the main coroutine
import asyncio

# define a main coroutine
async def main():
    # report a message
    print('main coroutine started')
    # get the current task
    task = asyncio.current_task()
    # report its details
    print(task)

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the main coroutine and uses it to start the asyncio program.

The `main()` coroutine runs and first reports a message.

It then retrieves the current task, which is a `Task` object that represents itself, the currently running coroutine.

It then reports the details of the currently running task.

We can see that the task has the default name for the first task, `'Task-1'` and is executing the `main()` coroutine, the currently running coroutine.

This highlights that we can use the `asyncio.current_task()` function to access a `Task` object for the currently running coroutine, that is automatically wrapped in a Task object.

```
main coroutine started
<Task pending name='Task-1'
    coro=<main() running at ...>
    cb=[_run_until_complete_cb() at ...]>
```

12.4 Example Of A Coroutine Accessing The Task

The `asyncio.current_task()` function can also be used to retrieve a `Task` object for the current task from another coroutine that is currently running.

We can explore running a second coroutine in an asyncio program and getting the current `Task` object from within it.

In this example, the main coroutine is used as the entry point into the program. It is converted into a `Task` automatically.

The main coroutine creates and runs a second coroutine and this second coroutine retrieves the current task, which refers to the main coroutine, not the currently running coroutine.

The complete example is listed below.

```
# SuperFastPython.com
# example of getting the task from another coroutine
import asyncio

# define another coroutine
async def another_coroutine():
    # report a message
    print('executing the coroutine')
    # get the current task
    task = asyncio.current_task()
    # report its details
    print(task)

# define a main coroutine
async def main():
    # report a message
    print('main coroutine started')
    # wait another coroutine
    await another_coroutine()
    # report a final message
    print('main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine reports a message, then creates and runs a second coroutine, then suspends waiting for it to be done.

The second coroutine runs, first reporting a message. It then gets the current `Task` object for itself and reports its details.

We can see that the `asyncio.current_task()` returns a `Task` instance for the `main()` coroutine that was wrapped in a task when it was used to start the program.

It does not refer to the coroutine that is currently running.

This highlights that the coroutine used as the entry point is turned into a task within the event loop, but arbitrary coroutines executing within a task are not themselves turned into a task and cannot be referenced directly as tasks.

```
main coroutine started
executing the coroutine
<Task pending name='Task-1'
    coro=<main() running at ...>
    cb=[_run_until_complete_cb() at ...]>
main coroutine done
```

Next, we will look at getting a task object for a new separate task.

12.5 Example Of A New Task Accessing Itself

We can create and schedule a new task and the new task can get a `Task` object to reference itself.

In this example, we will define a coroutine that we will then wrap in a new `Task`. The task will get itself and report its details.

The complete example is listed below.

```
# SuperFastPython.com
# example of getting the task object from another task
import asyncio

# define another coroutine
async def another_coroutine():
    # report a message
    print('executing the task')
    # get the current task
    task = asyncio.current_task()
    # report its details
    print(task)
```

```
# define a main coroutine
async def main():
    # report a message
    print('main coroutine started')
    # create another task
    task = asyncio.create_task(another_coroutine())
    # await the task
    await task
    # report a final message
    print('main coroutine done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine reports a message. It then creates and schedules a task and suspends it until it is done.

The new task runs. It reports a message, then gets a `Task` object for the currently running task and reports its details.

We can see that a new second task was created and used to execute the second coroutine. It has the name `'Task-2'`, compared to `'Task-1'` for the entry point to the program that we saw in the above example. We can also see that the coroutine executed by the task matches our new second coroutine.

This highlights that we can get a `Task` object for new tasks that are created and scheduled within our program.

```
main coroutine started
executing the task
<Task pending name='Task-2'
    coro=<another_coroutine() running at ...>
    cb=[Task.task_wakeup()]>
main coroutine done
```

12.6 Takeaways

You now know about the main task and the current task in asyncio.

Specifically, you now know:

- What is the “main task” in an asyncio program.
- The special properties of the main task in asyncio programs and how to identify it.
- How to retrieve the currently executing asyncio task.

12.6.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

12.6.2 Next

In the next tutorial, we will explore how to introspect an asyncio program and access all tasks running in the event loop.

Chapter 13

All Tasks

The asyncio event loop is a program for executing asyncio tasks and coroutines.

It runs our asyncio programs, but it also provides tools for introspecting the tasks that are running.

One of these tools is the ability to access all of the tasks that are currently running and are not yet done.

In this tutorial, you will discover how to get and access all asyncio tasks.

After completing this tutorial, you will know:

- How to access all tasks running in the asyncio event loop.
- How we might develop a coroutine to wait for all currently running tasks to complete.
- The dangers of a task that attempts to wait upon itself.

Let's get started.

13.1 How To Get All Asyncio Tasks

We may need to get access to all tasks in an asyncio program.

This may be for many reasons, such as:

- To introspect the current status or complexity of the program.
- To log the details of all running tasks or to cancel all tasks.
- To find a task that can be queried or canceled.

We can get a collection of all scheduled and running (not yet done) tasks in an asyncio program via the `asyncio.all_tasks()` function.

For example:

```
...
# get all tasks
tasks = asyncio.all_tasks()
```

This will return all tasks in the asyncio program. It is a set datatype so that each task is only represented once.

A task will be included if:

- The task has been scheduled but is not yet running.
- The task is currently running (e.g. but is currently suspended).

The set will also include a task instance for the currently running task, e.g. the task that is executing the coroutine that calls the `asyncio.all_tasks()` function.

Recall that coroutines are only represented by a task if an explicit call to `create_task()` is made, otherwise the coroutine will not be represented in the set of all tasks.

Also, recall that the `asyncio.run()` method that is used to start an asyncio program will wrap the provided coroutine in a task. This means that the set of all tasks will include the task for the entry point of the program called the "main task".

Now that we know how to get all tasks in an asyncio program, let's look at some worked examples.

13.2 Example Of Getting All Tasks

We can explore the case where we have many tasks within an asyncio program and then get a set of all tasks.

In this example, we first create 10 tasks, each wrapping and running the same coroutine.

The main coroutine then gets a set of all tasks scheduled or running in the program and reports their details.

The complete example is listed below.

```
# SuperFastPython.com
# example of starting and getting many tasks
import asyncio

# coroutine for a task
async def task_coroutine(value):
    # report a message
    print(f'task {value} is running')
    # block for a moment
    await asyncio.sleep(1)

# define a main coroutine
async def main():
    # report a message
    print('main coroutine started')
    # start many tasks
```

```
    started_tasks = [asyncio.create_task(
        task_coroutine(i)) for i in range(10)]
    # allow some of the tasks time to start
    await asyncio.sleep(0.1)
    # get all tasks
    tasks = asyncio.all_tasks()
    # report all tasks
    for task in tasks:
        print(f'> {task.get_name()}, {task.get_coro()}')
    # wait for all tasks to complete
    for task in started_tasks:
        await task

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the main coroutine and uses it to start the asyncio program.

The `main()` coroutine runs and first reports a message.

It then creates and schedules 10 tasks that wrap the custom coroutine,

The `main()` coroutine then blocks for a moment to allow the tasks to begin running.

The tasks start running and each reports a message and then sleeps.

The `main()` coroutine resumes and gets a list of all tasks in the program.

It then reports the name and coroutine of each.

Finally, it enumerates the list of tasks that were created and awaits each, allowing them to be completed.

This highlights that we can get a set of all tasks in an asyncio program that includes both the tasks that were created as well as the task that represents the entry point into the program.

```
main coroutine started
task 0 is running
task 1 is running
task 2 is running
task 3 is running
task 4 is running
task 5 is running
task 6 is running
task 7 is running
task 8 is running
task 9 is running
> Task-9, <coroutine object task_coroutine at ...>
> Task-2, <coroutine object task_coroutine at ...>
```

```
> Task-11, <coroutine object task_coroutine at ...>
> Task-7, <coroutine object task_coroutine at ...>
> Task-4, <coroutine object task_coroutine at ...>
> Task-10, <coroutine object task_coroutine at ...>
> Task-8, <coroutine object task_coroutine at ...>
> Task-5, <coroutine object task_coroutine at ...>
> Task-1, <coroutine object main at ...>
> Task-3, <coroutine object task_coroutine at ...>
> Task-6, <coroutine object task_coroutine at ...>
```

Next, let's look at how we might wait on all tasks in the program.

13.3 Example Of Waiting On All Tasks

A common use case for getting all tasks in an asyncio program is to wait for all tasks to be done.

This allows the caller to suspend and wait for all tasks to execute, such as before exiting the program.

The problem is, if we get all tasks and naively await each, it will result in an error. The reason is that the set of all tasks includes the current task and the current task cannot await itself.

We can demonstrate this with an example.

The example below first creates 10 tasks, it waits a moment for the tasks to start, then gets the set of all tasks and awaits each in turn.

The result is a `RuntimeException`.

The complete example is listed below.

```
# SuperFastPython.com
# example of starting many tasks and awaiting them
import asyncio

# coroutine for a task
async def task_coroutine(value):
    # report a message
    print(f'task {value} is running')
    # block for a moment
    await asyncio.sleep(1)

# define a main coroutine
async def main():
    # report a message
```

```
    print('main coroutine started')
    # start many tasks
    for i in range(10):
        asyncio.create_task(task_coroutine(i))
    # allow some of the tasks time to start
    await asyncio.sleep(0.1)
    # get all tasks
    tasks = asyncio.all_tasks()
    # wait for all tasks to complete
    for task in tasks:
        await task

# start the asyncio event loop
asyncio.run(main()) # error
```

Running the example first creates the main coroutine and uses it to start the asyncio program.

The `main()` coroutine runs and first reports a message.

It then creates and schedules 10 tasks that wrap the custom coroutine,

The `main()` coroutine then blocks for a moment to allow the tasks to begin running.

The tasks start running and each reports a message and then sleeps.

The `main()` coroutine resumes and gets a set of all tasks in the program.

It traverses the set of all tasks and awaits each.

This fails with a `RuntimeException`.

It fails because the set of all tasks contains a task for the currently running task and a task cannot await itself.

```
main coroutine started
task 0 is running
task 1 is running
task 2 is running
task 3 is running
task 4 is running
task 5 is running
task 6 is running
task 7 is running
task 8 is running
task 9 is running
Traceback (most recent call last):
  ...
RuntimeError: Task cannot await on itself:
    <Task pending name='Task-1'
```

```
    coro=<main() running at ...>
    cb=[_run_until_complete_cb() at ...]>
```

Instead, the example must be updated so that the running task does not await itself.

This can be achieved by first getting the `Task` object for the currently running task via the `asyncio.current_task()` function (explored in the previous chapter).

We then await each task as long as it does not represent the currently running task.

For example:

```
...
# get all tasks
tasks = asyncio.all_tasks()
# get the current task
current = asyncio.current_task()
# wait for all tasks to complete
for task in tasks:
    # skip the current task
    if task is current:
        continue
    # await the task
    await task
```

We can simplify this by removing the current task from the set of all tasks, then traverse the set of the remaining tasks and await each in turn.

For example:

```
...
# get all tasks
tasks = asyncio.all_tasks()
# get the current task
current = asyncio.current_task()
# remove the current task from the set
tasks.remove(current)
# wait for all tasks to complete
for task in tasks:
    # await the task
    await task
```

We can use this same pattern in the case where we want to explicitly cancel all tasks in the asyncio program, except the current task.

Tying this together, the example of correctly awaiting all running tasks in an asyncio program is listed below.

```
# SuperFastPython.com
# example of starting many tasks and awaiting them
```

```
import asyncio

# coroutine for a task
async def task_coroutine(value):
    # report a message
    print(f'task {value} is running')
    # block for a moment
    await asyncio.sleep(1)

# define a main coroutine
async def main():
    # report a message
    print('main coroutine started')
    # start many tasks
    for i in range(10):
        asyncio.create_task(task_coroutine(i))
    # allow some of the tasks time to start
    await asyncio.sleep(0.1)
    # get all tasks
    tasks = asyncio.all_tasks()
    # get the current task
    current = asyncio.current_task()
    # remove the current task from the set
    tasks.remove(current)
    # wait for all tasks to complete
    for task in tasks:
        # await the task
        await task

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the main coroutine and uses it to start the asyncio program.

The `main()` coroutine runs and first reports a message.

It then creates and schedules 10 tasks that wrap the custom coroutine,

The `main()` coroutine then blocks for a moment to allow the tasks to begin running.

The tasks start running and each reports a message and then sleeps.

The `main()` coroutine resumes and gets a set of all tasks in the program. It also gets the `Task` object for the currently running task (e.g. the entry point) and removes it from the set of all tasks.

It then traverses the remaining set of all tasks and awaits each in turn.

This highlights how we can get a set of all tasks in an asyncio program and await each before exiting the program.

```
main coroutine started
task 0 is running
task 1 is running
task 2 is running
task 3 is running
task 4 is running
task 5 is running
task 6 is running
task 7 is running
task 8 is running
task 9 is running
```

13.4 Takeaways

You now know about how to access all tasks that are running in the asyncio event loop.

Specifically, you know:

- How to access all tasks running in the asyncio event loop.
- How we might develop a coroutine to wait for all currently running tasks to complete.
- The dangers of a task that attempts to wait upon itself.

13.4.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

13.4.2 Next

In the next tutorial, we will explore how to wait for and retrieve results from multiple asyncio tasks.

Part III

Multiple Tasks

Chapter 14

Gather Tasks

The `asyncio.gather()` module function is perhaps the most commonly used way of executing a group of tasks concurrently.

It takes one or more coroutines or tasks, then waits until they are all done and returns an iterable of return values.

In this tutorial, you will discover how to handle a group of tasks concurrently with `gather()`.

After completing this tutorial, you will know:

- That the `gather()` function will wait for a collection of tasks to complete and retrieve all return values.
- How to use `gather()` with collections of coroutines and collections of tasks.
- How to use `gather()` with tasks that can be canceled and fail with an unhandled exception.

Let's get started.

14.1 How To Use Asyncio Gather

The `asyncio.gather()` module function allows the caller to group multiple awaitables together.

Once grouped, the awaitables can be executed concurrently and awaited.

It is a helpful utility function for both grouping and executing multiple coroutines or multiple tasks concurrently.

In this section, we will take a closer look at how to use the `asyncio.gather()` function.

14.1.1 Gather Takes Tasks And Coroutines

The `asyncio.gather()` function takes one or more awaitables as arguments.

Recall an awaitable may be a coroutine or an `asyncio.Task`.

Therefore, we can call the `gather()` function with:

- Multiple task objects.
- Multiple coroutine objects.
- Mixture of tasks and coroutines.

For example:

```
...
# execute multiple coroutines
asyncio.gather(coro1(), coro2())
```

If task objects are provided to `gather()`, they will already be scheduled. Recall that tasks are scheduled as part of being created.

If coroutines are provided to `gather()`, they are wrapped in `Task` objects and scheduled automatically.

The `asyncio.gather()` function takes awaitables as positional arguments.

This means that we cannot create a list or collection of awaitables and provide it to gather, as this will result in an error.

For example:

```
...
# create a list of awaitables
awaitables = [coro1(), coro2()]
# cannot provide a list of awaitables directly
asyncio.gather(awaitables) # error
```

A list of awaitables can be provided if it is first unpacked into separate expressions using the star operator (`*`).

For example:

```
...
# create a list of awaitables
awaitables = [coro1(), coro2()]
# gather with an unpacked list of awaitables
asyncio.gather(*awaitables)
```

14.1.2 Gather Returns An Awaitable

The `gather()` function does not block.

Instead, it returns an awaitable object that represents the group of awaitables.

For example:

```
...
# get a awaitable that represents multiple awaitables
group = asyncio.gather(coro1(), coro2())
```

Once the awaitable object is created it is scheduled automatically within the event loop.

The awaitable represents the group, and all awaitables in the group will execute as soon as they are able.

This means that if the caller did nothing else, the scheduled group of awaitables will run (assuming the caller suspends).

It also means that we do not have to await the awaitable that is returned from `gather()`.

For example:

```
...
# get a awaitable that represents multiple awaitables
group = asyncio.gather(coro1(), coro2())
# suspend and wait a while, the group may be executing..
await asyncio.sleep(10)
```

This is interesting, although not the common use case for `asyncio.gather()`.

The returned awaitable object can be awaited which will wait for all awaitables in the group to be done.

For example:

```
...
# run the group of awaitables
await group
```

Awaiting the awaitable returned from `gather()` will return a list of return values from the awaitables provided to `gather()`.

If the awaitables do not return a value, then this list will contain the default `None` return value.

For example:

```
...
# run the group of awaitables and get return values
results = await group
```

This is more commonly performed in one line.

For example:

```
...
# run tasks and get results on one line
results = await asyncio.gather(coro1(), coro2())
```

14.1.3 Gather And Exceptions

If an awaitable fails with an exception, the exception is re-raised in the caller and may need to be handled.

Similarly, if a task is canceled, it will re-raise a `CancelledError` exception in the caller and may need to be handled.

By default, `gather()` will attempt to retrieve the return value from each awaitable. This means that if a task in the group fails with an exception or is canceled, the exception will be re-raised in the caller.

We can set the `return_exceptions` argument to `gather()` to `True` which will catch these exceptions and provide them as return values instead of re-raising them in the caller.

Helpfully, this applies to both exceptions raised in awaitables, as well as `CancelledError` exceptions if the awaitables are canceled.

For example:

```
...
# run tasks and retrieve exceptions as a result
results = await asyncio.gather(coro1(), coro2(),
    return_exceptions=True)
```

Now that we know how to use `asyncio.gather()`, let's look at some worked examples.

14.2 Example Of Gather With A List Of Coroutines

It is common to create multiple coroutines beforehand and then gather them later.

This allows a program to prepare the tasks that are to be executed concurrently and then trigger their concurrent execution all at once and wait for them to complete.

We can collect many coroutines together into a list either manually or using a list comprehension.

For example:

```
...
# create a list of coroutines
coros = [task_coro(i) for i in range(10)]
```

This could just as easily be a list of tasks.

For example:

```
...
# create a list of tasks
coros = [asyncio.create_task(
    task_coro(i)) for i in range(10)]
```

We can then call `gather()` with all coroutines in the list.

The list of coroutines cannot be provided directly to the `gather()` function as this will result in an error.

Instead, the `gather()` function requires each awaitable to be provided as a separate positional argument.

This can be achieved by unwrapping the list into separate expressions and passing them to the `gather()` function. The star operator (`*`) will perform this operation for us.

For example:

```
...
# execute and wait for all tasks to be done
await asyncio.gather(*coros)
```

Here, we are ignoring the iterable of return values because our target coroutine does not return anything.

We could ignore it explicitly if we wanted, for example:

```
...
# execute and wait for all tasks to be done
_ = await asyncio.gather(*coros)
```

Tying this together, the complete example of running a list of pre-prepared coroutines with `gather()` is listed below.

```
# SuperFastPython.com
# example of gather for many coroutines in a list
import asyncio

# coroutine used for a task
async def task_coro(value):
    # report a message
    print(f'>task {value} executing')
    # sleep for a moment
    await asyncio.sleep(1)

# coroutine used for the entry point
async def main():
    # report a message
    print('main starting')
    # create a list of coroutines
    coros = [task_coro(i) for i in range(10)]
    # execute and wait for all tasks to be done
    await asyncio.gather(*coros)
    # report a message
    print('main done')
```

```
# start the asyncio event loop
asyncio.run(main())
```

Running the example executes the `main()` coroutine as the entry point to the program.

The `main()` coroutine then creates a list of 10 coroutine objects using a list comprehension.

This list is then provided to the `gather()` function and unpacked into 10 separate expressions using the star operator.

The `main()` coroutine then awaits the awaitable object returned from the call to `gather()`, suspending and waiting for all scheduled coroutines to complete their execution.

The coroutines run as soon as they are able, reporting their unique messages and sleeping before terminating.

Only after all coroutines in the group are complete does the `main()` coroutine resume and report its final message.

This highlights how we might prepare a collection of coroutines and provide them as separate expressions to the `gather()` function.

```
main starting
>task 0 executing
>task 1 executing
>task 2 executing
>task 3 executing
>task 4 executing
>task 5 executing
>task 6 executing
>task 7 executing
>task 8 executing
>task 9 executing
main done
```

14.3 Example Of Gather With Return Values

We may execute coroutines that return a value.

Using the `gather()` function to execute multiple tasks that return values allows the results of all tasks to be gathered together at one point for use, giving the `gather()` function its name.

Recall that the `gather()` function does not block, but returns immediately with an awaitable object.

When the awaitable object that is returned from `gather()` is awaited, it will return a list of return values from the grouped tasks.

The example below demonstrates this by updating the task coroutine to return a value and gathering and reporting the list of return values from all tasks in the main coroutine.

```
# SuperFastPython.com
# example of gather many coroutines that return values
import asyncio

# coroutine used for a task
async def task_coro(value):
    # report a message
    print(f'>task {value} executing')
    # sleep for a moment
    await asyncio.sleep(1)
    # return a value
    return value * 10

# coroutine used for the entry point
async def main():
    # report a message
    print('main starting')
    # create many tasks
    tasks = [asyncio.create_task(
        task_coro(i)) for i in range(10)]
    # run the tasks
    values = await asyncio.gather(*tasks)
    # report the values
    print(values)
    # report a message
    print('main done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates and runs the `main()` coroutine as the entry point into the program.

A list of tasks is then created, each with a different argument.

The gather function is called and the list of tasks is then unpacked into separate expressions using the star operator.

The main coroutine suspends and waits for all tasks in the group to complete.

Each task runs, reporting a message, sleeping, and returning a value that is a multiple of 10 of the input argument.

All coroutines are completed and the awaitable returned from `gather()` provides a list of return values.

The `main()` coroutine then reports the list of return values.

This highlights how we can retrieve return values from multiple coroutines that return values.

```
main starting
>task 0 executing
>task 1 executing
>task 2 executing
>task 3 executing
>task 4 executing
>task 5 executing
>task 6 executing
>task 7 executing
>task 8 executing
>task 9 executing
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
main done
```

14.4 Example Of Gather With Returned Exceptions

We can prevent an exception in the group of awaitables from being re-raised in the caller by setting the `return_exceptions` argument to `True` when calling `gather()`.

For example:

```
# run the tasks
results = await asyncio.gather(*tasks,
    return_exceptions=True)
```

This will trap the exception and return it as a return value for the awaitable.

We can then update our coroutine to fail with an exception if it is provided an argument with a given value.

```
# coroutine used for a task
async def task_coro(value):
    # report a message
    print(f'>task {value} executing')
    # sleep for a moment
    await asyncio.sleep(1)
    # check for failure
    if value == 0:
        raise Exception('Something bad happened')
    # return a value
    return value * 10
```

Tying this together, the example below demonstrate `gather()` that will return exceptions as a result and a task that will fail.

```
# SuperFastPython.com
# example of gather with returned exceptions
import asyncio

# coroutine used for a task
async def task_coro(value):
    # report a message
    print(f'>task {value} executing')
    # sleep for a moment
    await asyncio.sleep(1)
    # check for failure
    if value == 0:
        raise Exception('Something bad happened')
    # return a value
    return value * 10

# coroutine used for the entry point
async def main():
    # report a message
    print('main starting')
    # create many coroutines
    tasks = [asyncio.create_task(
        task_coro(i)) for i in range(10)]
    # run the tasks
    results = await asyncio.gather(*tasks,
        return_exceptions=True)
    # report results
    print(results)
    # report a message
    print('main done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example creates and runs the `main()` coroutine as the entry point into the program.

The `main()` coroutine then creates 10 task coroutines in a list comprehension. These are then provided to a call to `gather()`.

All coroutines are created as tasks scheduled for execution along with the awaitable returned from `gather()`.

The `main()` coroutine then suspends and waits for the tasks to complete.

All tasks execute, reporting a message and suspending. One task then raises an exception.

The exception does not impact any other coroutines in the group and is not re-raised in the caller.

The caller retrieves a list of return values from all awaitables. It then reports the values.

We can see that all awaitables return an integer, except the one that failed, where we can see the `Exception` object that was returned.

This highlights that exceptions can be prevented from being re-raised and can be retrieved as a return value from a failed awaitable.

```
main starting
>task 0 executing
>task 1 executing
>task 2 executing
>task 3 executing
>task 4 executing
>task 5 executing
>task 6 executing
>task 7 executing
>task 8 executing
>task 9 executing
[Exception('Something bad happened'),
    1, 2, 3, 4, 5, 6, 7, 8, 9]
main done
```

14.5 Takeaways

You now know how to await asyncio tasks concurrently with `gather()`.

Specifically, you know:

- That the `gather()` function will wait for a collection of tasks to complete and retrieve all return values.
- How to use `gather()` with collections of coroutines and collections of tasks.
- How to use `gather()` with tasks that can be canceled and fail with an unhandled exception.

14.5.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

14.5.2 Next

In the next tutorial, we will explore how to wait on a condition with a collection of asyncio tasks.

Chapter 15

Wait On Tasks

After issuing many tasks on asyncio, we may need to wait for a specific condition to occur in the group.

For example, we may want to wait until all tasks are complete, or for the first task to complete or fail and know which task it was.

This can be achieved via the `asyncio.wait()` function.

In this tutorial, you will discover how to wait on a group of tasks for a specific condition.

After completing this tutorial, you will know:

- How to use the `wait()` function to wait on a collection of asyncio tasks.
- How to configure the condition waited for by the `wait()` function.
- How to wait for a target condition with a timeout.

Let's get started.

15.1 How To Use Asyncio Wait

The `asyncio` module provides a utility function to wait for all tasks in a collection to be done.

This is preferred over `asyncio.gather()` if we don't require the results from the tasks.

It also supports other wait conditions on the collection of tasks.

15.1.1 Wait For All Tasks

The `asyncio.wait()` function takes a collection of awaitables, typically `Task` objects.

This could be a `list`, `dict` or `set` of task objects that we have created, such as via calls to the `asyncio.create()` task function in a list comprehension.

For example:

```
...
# create many tasks
tasks = [asyncio.create_task(
    task_coro(i)) for i in range(10)]
```

The `asyncio.wait()` will not return until some condition on the collection of tasks is met.

By default, the condition is that all tasks are completed.

The `wait()` function returns a tuple of two sets. The first set contains all task objects that meet the condition, and the second contains all other task objects that do not yet meet the condition.

These sets are referred to as the *done* set and the *pending* set.

For example:

```
...
# wait for all tasks to complete
done, pending = await asyncio.wait(tasks)
```

Technically, the `asyncio.wait()` is a coroutine function that returns a coroutine.

We can then await this coroutine which will return the tuple of sets.

For example:

```
...
# create the wait coroutine
wait_coro = asyncio.wait(tasks)
# await the wait coroutine
tuple = await wait_coro
```

The condition waited for can be specified by the `return_when` argument which is set to `asyncio.ALL_COMPLETED` by default.

For example:

```
...
# wait for all tasks to complete
done, pending = await asyncio.wait(tasks,
    return_when=asyncio.ALL_COMPLETED)
```

15.1.2 Wait For First Done Task

We can wait for the first task to be completed by setting `return_when` to `FIRST_COMPLETED`.

For example:

```
...
# wait for the first task to be completed
done, pending = await asyncio.wait(tasks,
    return_when=asyncio.FIRST_COMPLETED)
```

When the first task is complete and returned in the done set, the remaining tasks are not canceled and continue to execute concurrently.

15.1.3 Wait For First Failed Task

We can wait for the first task to fail with an exception by setting `return_when` to `FIRST_EXCEPTION`.

For example:

```
...
# wait for the first task to fail
done, pending = await asyncio.wait(tasks,
    return_when=asyncio.FIRST_EXCEPTION)
```

In this case, the done set will contain the first task that failed with an exception. If no task fails with an exception, the done set will contain all tasks and `wait()` will return only after all tasks are completed.

15.1.4 Wait With Timeout

We can specify how long we are willing to wait for the given condition via a `timeout` argument in seconds.

If the timeout expires before the condition is met, the tuple of tasks is returned with whatever subset of tasks do meet the condition at that time, e.g. the subset of tasks that are completed if waiting for all tasks to complete.

An exception is not raised and running tasks are not canceled.

For example:

```
...
# wait for all tasks to complete with a timeout
done, pending = await asyncio.wait(tasks, timeout=3)
```

Now that we know how to use the `asyncio.wait()` function, let's look at some worked examples.

15.2 Example Of Waiting For All Tasks

We can explore how to wait for all tasks using `asyncio.wait()`.

In this example, we will define a simple task coroutine that generates a random value, sleeps for a fraction of a second, then reports a message with the generated value.

The main coroutine will then create many tasks in a list comprehension with the coroutine and then wait for all tasks to complete.

The complete example is listed below.

```
# SuperFastPython.com
# example of waiting for all tasks to complete
from random import random
import asyncio

# coroutine to execute in a new task
async def task_coro(arg):
    # generate a random value between 0 and 1
    value = random()
    # block for a moment
    await asyncio.sleep(value)
    # report the value
    print(f'>task {arg} done with {value}')

# main coroutine
async def main():
    # create many tasks
    tasks = [asyncio.create_task(
        task_coro(i)) for i in range(10)]
    # wait for all tasks to complete
    done,pending = await asyncio.wait(tasks)
    # report results
    print('All done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine then creates a list of ten tasks in a list comprehension, each providing a unique integer argument from 0 to 9.

The `main()` coroutine is then suspended and waits for all tasks to complete.

The tasks execute. Each generates a random value, sleeps for a moment, then reports its generated value.

After all tasks have been completed, the `main()` coroutine resumes and reports a final message.

This example highlights how we can use the `wait()` function to wait for a collection of tasks to be completed.

This is perhaps the most common usage of the function.

Note, that the results will differ each time the program is run given the use of random numbers.

```
>task 5 done with 0.0591009105682192
>task 8 done with 0.10453715687017351
>task 0 done with 0.15462838864295925
>task 6 done with 0.4103492027393125
>task 9 done with 0.45567100006991623
>task 2 done with 0.6984682905809402
>task 7 done with 0.7785363531316224
>task 3 done with 0.827386088873161
>task 4 done with 0.9481344994700972
>task 1 done with 0.9577302665040541
All done
```

Next, let's look at how we might wait for the first task to complete.

15.3 Example Of Waiting For First Task

We can explore how to wait for the first task to complete using `asyncio.wait()`.

In this example, we will reuse the same simple task coroutine that generates a random value, sleeps for a fraction of a second, then reports a message with the generated value.

The main coroutine will then create many tasks in a list comprehension with the coroutine as before. It will then wait for the first task to complete.

```
# wait for the first task to complete
done,pending = await asyncio.wait(tasks,
    return_when=asyncio.FIRST_COMPLETED)
```

Once a task is completed, it is retrieved from the set of done tasks and its details are reported.

It is a `set` datatype, so we can call the `pop()` method to retrieve the done task.

```
# get the first task to complete
first = done.pop()
```

The complete example is listed below.

```
# SuperFastPython.com
# example of waiting for the first task to complete
from random import random
import asyncio

# coroutine to execute in a new task
async def task_coro(arg):
    # generate a random value between 0 and 1
    value = random()
    # block for a moment
    await asyncio.sleep(value)
```

```
    # report the value
    print(f'>task {arg} done with {value}')

# main coroutine
async def main():
    # create many tasks
    tasks = [asyncio.create_task(
        task_coro(i)) for i in range(10)]
    # wait for the first task to complete
    done,pending = await asyncio.wait(
        tasks, return_when=asyncio.FIRST_COMPLETED)
    # report result
    print('Done')
    # get the first task to complete
    first = done.pop()
    print(first)

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine then creates a list of ten tasks in a list comprehension, each providing a unique integer argument from 0 to 9.

The `main()` coroutine is then suspended and waits for the first task to complete

The tasks execute. Each generates a random value, sleeps for a moment, then reports its generated value.

As soon as the first task completes, the `wait()` function returns and the `main()` coroutine resumes.

The *done* set contains a single task that is finished, whereas the *pending* set contains all other tasks that were provided in the collection to the `wait()` function.

The single finished task is then retrieved from the *done* set and is reported. The printed status of the task confirms that indeed it is done.

The other tasks are not canceled and continue to run concurrently. Their execution is cut short because the asyncio program is terminated and all remaining running tasks are automatically canceled.

This example highlights how we can use the `wait()` function to wait for the first task to complete.

Note, that the results will differ each time the program is run given the use of random numbers.

```
>task 9 done with 0.04034360933451242
Done
<Task finished name='Task-11'
    coro=<task_coro() done, defined at ...>
    result=None>
```

Next, let's look at how we might wait for the first task to fail with an exception.

15.4 Example Of Waiting For First Task Failure

We can explore how to wait for the first task to fail using `asyncio.wait()`.

In this example, we will reuse the same simple task coroutine that generates a random value, sleeps for a fraction of a second, then reports a message with the generated value.

The task coroutine is updated so that conditionally it will fail with an exception if the generated value is below a threshold.

The main coroutine will then create many tasks. It will then wait for the first task to fail with an exception.

```
# wait for the first task to fail, or all done
done,pending = await asyncio.wait(tasks,
    return_when=asyncio.FIRST_EXCEPTION)
```

Once a task has failed, it is retrieved from the set of done tasks and its details are reported.

The complete example is listed below.

```
# SuperFastPython.com
# example of waiting for the first task to fail
from random import random
import asyncio

# coroutine to execute in a new task
async def task_coro(arg):
    # generate a random value between 0 and 1
    value = random()
    # block for a moment
    await asyncio.sleep(value)
    # report the value
    print(f'>task {arg} done with {value}')
    # conditionally fail
    if value < 0.5:
        raise Exception(
            f'Something bad happened in {arg}')
```

```
# main coroutine
async def main():
    # create many tasks
    tasks = [asyncio.create_task(
        task_coro(i)) for i in range(10)]
    # wait for the first task to fail, or all done
    done,pending = await asyncio.wait(
        tasks, return_when=asyncio.FIRST_EXCEPTION)
    # report result
    print('Done')
    # get the first task to fail
    first = done.pop()
    print(first)

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine then creates a list of ten tasks in a list comprehension, each providing a unique integer argument from 0 to 9.

The `main()` coroutine is then suspended and waits for the first task to complete

The tasks execute. Each generates a random value, sleeps for a moment, then reports its generated value.

Conditionally, some tasks raise an exception if their generated value is above a threshold value. It is possible that all tasks generate a value below the threshold and none raise an exception, although this situation is extremely unlikely.

As soon as the first task fails with an exception, the `wait()` function returns and the `main()` coroutine resumes.

The *done* set contains a single task that failed first, whereas the *pending* set contains all other tasks that were provided in the collection to the `wait()` function.

The single failed task is then retrieved from the *done* set and is reported. The printed status of the task confirms that it indeed failed with an exception in this case.

The other tasks are not canceled and continue to run concurrently. Their execution is cut short because the asyncio program is terminated.

This example highlights how we can use the `wait()` function to wait for the first task to fail.

Note, that the results will differ each time the program is run given the use of random numbers.

```
>task 5 done with 0.13168449673381888
Done
<Task finished name='Task-7'
    coro=<task_coro() done, defined at ...>
    exception=Exception('Something bad happened in 5')>
```

Next, let's look at how we might wait for tasks with a timeout.

15.5 Example Of Waiting With A Timeout

We can explore how to wait for all tasks to complete with a timeout using `asyncio.wait()`.

In this example, we will reuse the same simple task coroutine that generates a random value, sleeps for a fraction of a second, then reports a message with the generated value.

The main coroutine will then create many tasks and wait for all tasks to complete. In this case, a `timeout` is specified, setting an upper limit on how long the caller is willing to wait.

```
# wait for all tasks to complete
done,pending = await asyncio.wait(tasks, timeout=5)
```

Once all tasks are complete, or the timeout expires, the sets of done and pending tasks are returned. The total number of tasks completed within the timeout is then reported.

The complete example is listed below.

```
# SuperFastPython.com
# example of waiting for all tasks with a timeout
from random import random
import asyncio

# coroutine to execute in a new task
async def task_coro(arg):
    # generate a random value between 0 and 10
    value = random() * 10
    # block for a moment
    await asyncio.sleep(value)
    # report the value
    print(f'>task {arg} done with {value}')

# main coroutine
async def main():
    # create many tasks
    tasks = [asyncio.create_task(
        task_coro(i)) for i in range(10)]
    # wait for all tasks to complete
    done,pending = await asyncio.wait(tasks, timeout=5)
```

```
    # report results
    print(f'Done, {len(done)} tasks completed in time')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine then creates a list of ten tasks in a list comprehension, each providing a unique integer argument from 0 to 9.

The `main()` coroutine is then suspended and waits for all tasks to complete.

It specifies a timeout of 5 seconds. This is about half the duration of the longest tasks that could be generated.

The tasks execute. Each generates a random value between 0 and 10, sleeps for the generated number of seconds, then reports its generated value.

The timeout expires and the `main()` coroutine resumes. It reports a message indicating the number of tasks that were completed within the timeout, which was 6 in this case.

This example highlights how we can use the `wait()` function to wait for a collection of tasks to be completed with a timeout.

Note, that the results will differ each time the program is run given the use of random numbers.

```
>task 7 done with 0.16485206249285955
>task 0 done with 0.73241529734688
>task 4 done with 1.1137310743743878
>task 6 done with 2.396915437441108
>task 5 done with 3.375537014759735
>task 2 done with 4.821848023696365
Done, 6 tasks completed in time
```

15.6 Takeaways

You now know how to wait for asyncio tasks to complete.

Specifically, you know:

- How to use the `wait()` function to wait on a collection of asyncio tasks.
- How to configure the condition waited for by the `wait()` function.
- How to wait for a target condition with a timeout.

15.6.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- `random` – Generate pseudo-random numbers.
  https://docs.python.org/3/library/random.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

15.6.2 Next

In the next tutorial, we will explore how to handle asyncio tasks in the order they are completed.

Chapter 16

Tasks As Completed

It is common to issue many tasks at once, then need to process the results from each task as the tasks are completed.

This can be more efficient than waiting for all tasks to complete before handling the results.

We can achieve this and iterate tasks in the order they are done using the `as_completed()` function.

In this tutorial, you will discover how to use the `as_completed()` function in asyncio.

After completing this tutorial, you will know:

- How to use the `as_completed()` function to iterate over a collection of tasks in the order they are completed.
- How to retrieve results from coroutines and tasks that are traversed in completion order.
- How to set and use a timeout when iterating tasks in completion order.

Let's get started.

16.1 How To Use Asyncio As Completed

The `asyncio` module provides a utility function that creates a generator that will yield tasks from a collection that we provide in the order they are completed.

This is helpful for reporting results from tasks in the order the tasks are completed, rather than the order tasks were issued, or after all tasks are done.

16.1.1 Tasks As Completed

The `asyncio.as_completed()` function is called with a collection of awaitables.

This may be a `list`, `dict`, or `set`, and may contain `asyncio.Task` objects, coroutines, or other awaitables.

Any coroutines provided to `as_completed()` will be wrapped in a `Task` object for independent execution.

Importantly, the `asyncio.as_completed()` function does not return an iterable of return values of provided awaitables. Instead, it returns an iterable that when traversed will yield awaitables (or wrapped awaitables) in the provided list.

These can be awaited by the caller in order to get results in the order that tasks are completed, e.g. get the result from the next task to complete.

For example:

```
...
# iterate over awaitables
for task in asyncio.as_completed(tasks):
    # get the next result
    result = await task
```

16.1.2 As Completed With Timeout

The `as_completed()` function also takes a `timeout` argument.

This specifies how long the caller is willing to wait for all awaitables to be done.

For example:

```
...
# iterate over awaitables with a timeout
for task in asyncio.as_completed(tasks, timeout=10):
    # get the next result
    result = await task
```

If the timeout elapses before all awaitables are done, a `asyncio.TimeoutError` is raised and may need to be handled.

For example, we could handle it within the loop and processing results:

```
...
# iterate over awaitables with a timeout
for task in asyncio.as_completed(tasks, timeout=10):
    # handle a timeout
    try:
        # get the next result
        result = await task
    except asyncio.TimeoutError:
        # ...
```

This is not desirable because once the timeout has elapsed, an `asyncio.TimeoutError` will be raised each time `next()` is called on the generator.

Therefore, it is better to wrap the entire loop in a try-except block.

For example:

```
...
# handle a timeout
try:
    # iterate over awaitables with a timeout
    for task in asyncio.as_completed(tasks, timeout=10):
        # get the next result
        result = await task
except asyncio.TimeoutError:
    # ...
```

Now that we know how to use the `as_completed()` function, let's take a moment to consider how it works.

16.2 How Does As Completed Work

The `as_completed()` function works by providing a generator that yields coroutines, where each coroutine will return a result of a provided awaitable.

The `asyncio.as_completed()` function does not block (suspend the caller) but instead returns a generator.

For example:

```
...
# generator that returns awaitables in completion order
generator = asyncio.as_completed(tasks)
```

Calling the `next()` built-in function on the generator does not block, but instead yields a coroutine.

The returned coroutine is not one of the provided awaitables, but rather an internal coroutine from the `as_completed()` function that manages and monitors which issued task will return a result next.

For example:

```
...
# get the next coroutine
coro = next(generator)
```

It is not until one of the returned coroutines is awaited that the caller will block.

For example:

```
...
# get a result from the next task to complete
result = await coro
```

Note, the `asyncio.as_completed()` function returns a regular Python generator that in turn can be iterated with a regular Python `for` loop. It is not an asynchronous generator

and cannot be traversed via an `async for` expression.

Now that we have an idea of how to use the `as_completed()` function and how it works, let's look at some worked examples.

16.3 Example Of As Completed With Tasks

We can explore how to execute `asyncio.Task` objects concurrently and gets coroutine results as tasks are done with the `asyncio.as_completed()` function.

In this example, we will create a list of `asyncio.Task` objects from a custom coroutine that generates a random result, suspends a moment, then returns a result.

The complete example is listed below.

```
# SuperFastPython.com
# example of getting task results as completed
from random import random
import asyncio

# coroutine to execute in a new task
async def task_coro(arg):
    # generate a random value between 0 and 1
    value = random()
    # block for a moment
    await asyncio.sleep(value)
    # return the result
    return arg * value

# main coroutine
async def main():
    # create many tasks
    coros = [asyncio.create_task(
        task_coro(i)) for i in range(10)]
    # get results as tasks are completed
    for coro in asyncio.as_completed(coros):
        # get the result from the next task to complete
        result = await coro
        # report the result
        print(f'>got {result}')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and then uses this as the entry point into the asyncio program.

The `main()` coroutine runs and creates a list of `Task` objects, instead of coroutines.

Each task is scheduled for independent execution after it is created.

The `main()` coroutine then passes this list to the `as_completed()` function which returns a generator. The generator is traversed in a for loop and each iteration yields a coroutine.

The coroutine is awaited. This suspends the `main()` coroutine.

The tasks begin executing, generating a random value, and sleeping. A task finishes and returns a value.

The `main()` coroutine resumes, receives the return value, and reports it.

This continues until all coroutines in the provided list are completed. Notice that the task results are reported in ascending order as the time that each task suspends is proportional to the random number that was generated.

This example highlights that the `as_completed()` function can be used with `Task` objects as well as coroutines that we saw in the previous example.

Results will differ each time the example is run given the use of random numbers.

```
>got 0.3453975438487853
>got 0.5871779512599952
>got 0.0
>got 0.6767468346945187
>got 0.9905365580643459
>got 1.9274714575238143
>got 0.5127918804896211
>got 1.308261888746516
>got 4.12075681907987
>got 7.128076208913287
```

16.4 Example Of As Completed With A Timeout

We can explore how to execute coroutines concurrently with the `as_completed()` function and a timeout and handle cases where results take longer than the expected duration.

In this example, we call the `as_completed()` function with a timeout that is not long enough for all tasks to be completed.

This is intentional as we want to explore how to handle the case that a collection of tasks takes too long to be done.

The complete example is listed below.

```
# SuperFastPython.com
# example of getting coroutine results with a timeout
from random import random
```

```
import asyncio

# coroutine to execute in a new task
async def task_coro(arg):
    # generate a random value between 0 and 1
    value = random()
    # block for a moment
    await asyncio.sleep(value)
    # return the result
    return arg * value

# main coroutine
async def main():
    # create many coroutines
    coros = [task_coro(i) for i in range(10)]
    # handle a timeout
    try:
        # get results as coroutines are completed
        for coro in asyncio.as_completed(
            coros, timeout=0.5):
            # get the result from the next to complete
            result = await coro
            # report the result
            print(f'>got {result}')
    except asyncio.TimeoutError:
        print('Gave up after timeout')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and then uses this as the entry point into the asyncio program.

The `main()` coroutine runs and creates a list of coroutines.

It then passes this list to the `as_completed()` function with a timeout.

A generator is returned and traversed in a for loop and each iteration yields a coroutine.

The coroutine is awaited. This suspends the `main()` coroutine.

The tasks begin executing, generating a random value, and sleeping. A task finishes and returns a value.

The `main()` coroutine resumes, receives the return value, and reports it.

This continues until the timeout elapses, after which a `TimeoutError` exception is raised.

This breaks the loop and is caught, reporting a message.

This example highlights how we can impose a limit on how long the caller is willing to wait for all tasks to be completed and handle the case where the timeout elapses before all results are retrieved.

Results will differ each time the example is run given the use of random numbers.

```
>got 1.1794285132416094
>got 0.2733126502808916
>got 0.9659219853827947
>got 1.8523793561409736
>got 0.0
>got 3.4477334386446854
Gave up after timeout
```

16.5 Example Of As Completed With An Exception

We can explore how to execute coroutines concurrently with `as_completed()` and one of the tasks fails with an exception.

In this example, we update the above example so that one of the task coroutines intentionally fails with an exception.

Tasks can fail and it is important to be aware of this and to consider handling a task failure when traversing tasks in completed order.

The complete example is listed below.

```
# SuperFastPython.com
# example of getting coroutine results with an exception
from random import random
import asyncio

# coroutine to execute in a new task
async def task_coro(arg):
    # generate a random value between 0 and 1
    value = random()
    # block for a moment
    await asyncio.sleep(value)
    # check if the task should fail
    if value > 0.5:
        raise Exception('Something bad happened')
    # return the result
    return arg * value

# main coroutine
```

```
async def main():
    # create many coroutines
    coros = [task_coro(i) for i in range(10)]
    # get results as coroutines are completed
    for coro in asyncio.as_completed(coros):
        # get the result from the next to complete
        result = await coro
        # report the result
        print(f'>got {result}')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and then uses this as the entry point into the asyncio program.

The `main()` coroutine runs and creates a list of coroutines.

It then passes this list to the `as_completed()` function which returns a generator. The generator is traversed in a for loop and each iteration yields a coroutine.

The coroutine is awaited. This suspends the `main()` coroutine.

The tasks begin executing, generating a random value, and sleeping. A task finishes and returns a value.

The `main()` coroutine resumes, receives the return value, and reports it.

This continues for a number of iterations.

Then, one of the tasks fails with an exception. When it is awaited, the exception is re-raised by the caller.

The exception is not expected and therefore it unwinds the asyncio program and terminates the program.

This example highlights what happens if a task in the collection fails with an exception. It reminds us to be careful when awaiting results from a coroutine and to potentially handle exceptions that could be raised.

Results will differ each time the example is run given the use of random numbers.

```
>got 0.05191041233869953
>got 0.3949931321512907
>got 0.1518973920395813
>got 2.125424835725659
>got 0.0
Traceback (most recent call last):
  ...
Exception: Something bad happened
```

16.6 Takeaways

You now know how to use the `as_completed()` function in asyncio.

Specifically, you know:

- How to use the `as_completed()` function to iterate over a collection of tasks in the order they are completed.
- How to retrieve results from coroutines and tasks that are traversed in completion order.
- How to set and use a timeout when iterating tasks in completion order.

16.6.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- `random` – Generate pseudo-random numbers.
  https://docs.python.org/3/library/random.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

16.6.2 Next

In the next tutorial, we will explore how to manage asyncio tasks as part of a group.

Chapter 17

Task Group

A problem with tasks is that it is a good idea to assign and keep track of the `asyncio.Task` objects.

The reason is that if we don't the tasks may be garbage collected, terminating the task.

A helpful solution is to use a `TaskGroup` to create and manage a collection of tasks. It has a number of benefits, such as canceling all tasks in the group if one task fails.

In this tutorial, you will discover how to use the `TaskGroup` in asyncio.

After completing this tutorial, you will know:

- How to use the `TaskGroup` to create and manage a collection of tasks.
- How to wait for a collection of tasks to complete using the `TaskGroup` context manager interface.
- How to cancel all tasks if one task in the `TaskGroup` fails.

Let's get started.

17.1 How To Use Asyncio Task Groups

The `asyncio.TaskGroup` class is intended as a replacement for the `asyncio.create_task()` function for creating tasks and the `asyncio.gather()` function for waiting on a group of tasks. It was introduced in Python 3.11.

Historically, we create and issue a coroutine as an `asyncio.Task` using the `asyncio.create_task()` function.

For example:

```
...
# create and issue coroutine as task
task = asyncio.create_task(coro())
```

This creates a new `asyncio.Task` object and issues it to the asyncio event loop for execution as soon as it is able.

We can then choose to await the task and wait for it to be completed.

For example:

```
...
# wait for task to complete
result = await task
```

As we have seen in previous tutorials, the `asyncio.gather()` function is used to create and issue many coroutines simultaneously as `asyncio.Task` objects to the event loop, allowing the caller to treat them together as a group.

The most common usage is to wait for all issued tasks to complete.

For example:

```
...
# issue coroutines as tasks and wait to complete
results = await asyncio.gather(coro1(), coro2(), coro2)
```

The `asyncio.TaskGroup` can perform both of these activities and is now the preferred approach.

17.1.1 Create A Task Group

An `asyncio.TaskGroup` object implements the asynchronous context manager interface, and this is the preferred usage of the class.

This means that an instance of the class is created and is used via the `async with` expression.

For example:

```
...
# create a taskgroup
async with asyncio.TaskGroup() as group:
    # ...
```

An asynchronous context manager implements the `__aenter__()` and `__aexit__()` methods which can be awaited. We will explore this pattern more in a later chapter.

In the case of the `asyncio.TaskGroup()`, the `__aexit__()` method which is called automatically when the context manager block is exited will await all tasks created by the `asyncio.TaskGroup`.

This means that exiting the `TaskGroup` object's block normally or via an exception will automatically await until all group tasks are done.

```
...
# create a taskgroup
async with asyncio.TaskGroup() as group:
```

```
    # ...
# wait for all group tasks are done
```

17.1.2 Create Tasks With A Task Group

We can create a task in the task group via the `create_task()` method on the `asyncio.TaskGroup` object.

For example:

```
...
# create a taskgroup
async with asyncio.TaskGroup() as group:
    # create and issue a task
    task = group.create_task(coro())
```

This will create an `asyncio.Task` object and issue it to the asyncio event loop for execution, just like the `asyncio.create_task()` function, except that the task is associated with the group.

We can await the task directly if we choose and get results.

For example:

```
...
# create a taskgroup
async with asyncio.TaskGroup() as group:
    # create and issue a task
    result = await group.create_task(coro())
```

The benefit of using the `asyncio.TaskGroup` is that we can issue multiple tasks in the group and execute code in between, such as checking results or gathering more data.

17.1.3 Wait On Tasks With A Task Group

We can wait on all tasks in the group by exiting the asynchronous context manager block.

As such, the tasks are awaited automatically and nothing additional is required.

For example:

```
...
# create a taskgroup
async with asyncio.TaskGroup() as group:
    # ...
# wait for all group tasks are done
```

If this behavior is not preferred, then we must ensure all tasks are *done* (finished, canceled, or failed) before exiting the context manager.

17.1.4 Cancel All Tasks If One Task Fails

If one task in the group fails with an exception, then all non-done tasks remaining in the group will be canceled.

This is performed automatically and does not require any additional code.

For example:

```
...
# handle the failure of any tasks in the group
try:
    ...
    # create a taskgroup
    async with asyncio.TaskGroup() as group:
        # create and issue a task
        task1 = group.create_task(coro1())
        # create and issue a task
        task2 = group.create_task(coro2())
        # create and issue a task
        task3 = group.create_task(coro3())
    # wait for all group tasks are done
except:
    # all non-done tasks are cancelled
    pass
```

If this behavior is not preferred, then the failure of each task must be managed within the tasks themselves, e.g. by a try-except block within the coroutine.

Now that we know how to use the `asyncio.TaskGroup`, let's look at some worked examples.

17.2 Example Of Waiting On Multiple Tasks

We can explore the case of creating multiple tasks within an `asyncio.TaskGroup` and then waiting for all tasks to complete.

This can be achieved by first defining a suite of different coroutines that represent the tasks we want to complete.

In this case, we will define 3 coroutines that each report a different message and then sleep for one second.

```
# coroutine task
async def task1():
    # report a message
    print('Hello from coroutine 1')
    # sleep to simulate waiting
    await asyncio.sleep(1)
```

```
# coroutine task
async def task2():
    # report a message
    print('Hello from coroutine 2')
    # sleep to simulate waiting
    await asyncio.sleep(1)

# coroutine task
async def task3():
    # report a message
    print('Hello from coroutine 3')
    # sleep to simulate waiting
    await asyncio.sleep(1)
```

Next, we can define a `main()` coroutine that creates the `asyncio.TaskGroup` via the context manager interface.

```
# asyncio entry point
async def main():
    # create task group
    async with asyncio.TaskGroup() as group:
    # ...
```

We can then create and issue each coroutine as a task into the event loop, although collected together as part of the group.

```
...
# run first task
group.create_task(task1())
# run second task
group.create_task(task2())
# run third task
group.create_task(task3())
```

Notice that we don't need to keep a reference to the `asyncio.Task` objects as the `asyncio.TaskGroup` will keep track of them for us.

Also, notice that we don't need to await the tasks because when we exit the context manager block for the `asyncio.TaskGroup` we will await all tasks in the group automatically.

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of asyncio task group
import asyncio

# coroutine task
async def task1():
```

```
    # report a message
    print('Hello from coroutine 1')
    # sleep to simulate waiting
    await asyncio.sleep(1)

# coroutine task
async def task2():
    # report a message
    print('Hello from coroutine 2')
    # sleep to simulate waiting
    await asyncio.sleep(1)

# coroutine task
async def task3():
    # report a message
    print('Hello from coroutine 3')
    # sleep to simulate waiting
    await asyncio.sleep(1)

# asyncio entry point
async def main():
    # create task group
    async with asyncio.TaskGroup() as group:
        # run first task
        group.create_task(task1())
        # run second task
        group.create_task(task2())
        # run third task
        group.create_task(task3())
    # wait for all tasks to complete...
    print('Done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first executes the `main()` coroutine, starting a new event loop.

The `main()` coroutine runs and creates an `asyncio.TaskGroup`.

All three coroutines are then created as `asyncio.Task` objects and issued to the event loop via the `asyncio.TaskGroup`.

The context manager block for the `asyncio.TaskGroup` is exited which automatically awaits all three tasks.

The tasks report their message and sleep.

Once all tasks are completed the `main()` coroutine resumes and reports a final message.

```
Hello from coroutine 1
Hello from coroutine 2
Hello from coroutine 3
Done
```

Next, let's look at an example of canceling all tasks in the group if one task fails.

17.3 Example Of Canceling All Tasks If A Task Fails

We can explore the case of canceling all tasks in the `asyncio.TaskGroup` if one task fails.

A failed task means that a coroutine is executed in an `asyncio.Task` object that raises an exception that is not handled in the coroutine, meaning that it bubbles up to the task and causes the task to be terminated.

It is common to issue many tasks and cancel all tasks if one or more of the tasks fails.

The `asyncio.TaskGroup` will perform this action automatically for us.

In this case, we will define 3 different coroutines that report a message and sleep. The second coroutine will then fail with an uncaught exception.

```
# coroutine task
async def task1():
    # report a message
    print('Hello from coroutine 1')
    # sleep to simulate waiting
    await asyncio.sleep(1)

# coroutine task
async def task2():
    # report a message
    print('Hello from coroutine 2')
    # sleep to simulate waiting
    await asyncio.sleep(0.5)
    # fail with an exception
    raise Exception('Something bad happened')

# coroutine task
async def task3():
    # report a message
    print('Hello from coroutine 2')
    # sleep to simulate waiting
    await asyncio.sleep(1)
```

Note that the second task sleeps less than the other two tasks before raising an exception.

This is to ensure that the other two tasks are still running at the point that the second task fails so that we can see if they are canceled as we expect.

The `main()` coroutine will issue all tasks via the `asyncio.TaskGroup` and then report the done and cancel status of each in turn once all tasks are done.

Recall a done task is a task that is finished normally, canceled, or failed with an exception.

```
# asyncio entry point
async def main():
    # handle exceptions
    try:
        # create task group
        async with asyncio.TaskGroup() as group:
            # run first task
            t1 = group.create_task(task1())
            # run second task
            t2 = group.create_task(task2())
            # run third task
            t3 = group.create_task(task3())
    except:
        pass
    # check the status of each task
    print(f't1: done={t1.done()}, can={t1.cancelled()}')
    print(f't2: done={t2.done()}, can={t2.cancelled()}')
    print(f't3: done={t3.done()}, can={t3.cancelled()}')
```

Notice that we wrap the entire `asyncio.TaskGroup` in a try-except block as any uncaught exception that occurs in a task is re-raised by the `asyncio.TaskGroup`

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of asyncio task group with a failed task
import asyncio

# coroutine task
async def task1():
    # report a message
    print('Hello from coroutine 1')
    # sleep to simulate waiting
    await asyncio.sleep(1)

# coroutine task
async def task2():
    # report a message
    print('Hello from coroutine 2')
```

```
    # sleep to simulate waiting
    await asyncio.sleep(0.5)
    # fail with an exception
    raise Exception('Something bad happened')

# coroutine task
async def task3():
    # report a message
    print('Hello from coroutine 2')
    # sleep to simulate waiting
    await asyncio.sleep(1)

# asyncio entry point
async def main():
    # handle exceptions
    try:
        # create task group
        async with asyncio.TaskGroup() as group:
            # run first task
            t1 = group.create_task(task1())
            # run second task
            t2 = group.create_task(task2())
            # run third task
            t3 = group.create_task(task3())
    except:
        pass
    # check the status of each task
    print(f't1: done={t1.done()}, can={t1.cancelled()}')
    print(f't2: done={t2.done()}, can={t2.cancelled()}')
    print(f't3: done={t3.done()}, can={t3.cancelled()}')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first executes the `main()` coroutine, starting a new event loop.

The `main()` coroutine runs and creates an `asyncio.TaskGroup`.

The three coroutines are then issued as tasks via the `asyncio.TaskGroup` and the `asyncio.Task` objects are kept in local variables for later.

The `asyncio.TaskGroup` context manager block is exited and the `main()` coroutine then awaits all three tasks.

The tasks run, report a message and sleep. The second coroutine then fails with an unhandled exception.

The `asyncio.TaskGroup` handles the exception and cancels all remaining not-done tasks. The exception is then re-raised at the top level and ignored.

The done and canceled status of each task is then reported. We can see that all tasks are done and that the two tasks (1 and 3) that were running at the time task 2 failed with an exception were indeed canceled.

This highlights how all running tasks in the group will be canceled if a task in the group fails with an unhandled exception.

```
Hello from coroutine 1
Hello from coroutine 2
Hello from coroutine 2
t1: done=True, can=True
t2: done=True, can=False
t3: done=True, can=True
```

17.4 Takeaways

You now know how to use the `TaskGroup` in asyncio.

Specifically, you know:

- How to use the `TaskGroup` to create and manage a collection of tasks.
- How to wait for a collection of tasks to complete using the `TaskGroup` context manager interface.
- How to cancel all tasks if one task in the `TaskGroup` fails.

17.4.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- What's New In Python 3.11.
  https://docs.python.org/3/whatsnew/3.11.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

17.4.2 Next

In the next tutorial, we will explore how to use timeouts within asyncio programs.

Chapter 18

Tasks With Timeouts

We often need to execute long-running tasks in asyncio.

For example, we may need to wait for a response from a remote server, for something to change, or for input from another system.

It is a best practice to use a timeout when waiting for a long-running task. If the task does not complete within a given time limit, it can then be canceled and perhaps tried again, or an error raised.

The `asyncio` module provides the `timeout()` asynchronous context manager to address exactly this problem.

In this tutorial, you will discover how to wait for coroutines and tasks using the `timeout()` asynchronous context manager.

After completing this tutorial, you will know:

- What is `timeout()` and how can we use it to wait for tasks to complete with a timeout.
- How to manage the case when the timeout is exceeded before the required tasks are complete.
- How to configure the timeout and how to extend the timeout if more time is needed.

Let's get started.

18.1 How To Use An Asyncio Timeout

The `asyncio` module provides a utility that imposes a timeout on a block of code in an asyncio program.

This is helpful if we require a block of code, e.g. one or more awaited tasks to be completed within a given time limit or before a given future time.

18.1.1 Create a Timeout

The `asyncio.timeout()` is an asynchronous context manager, introduced in Python 3.11.

As such, the entry and exit methods are coroutines that can be awaited and it must be used via the `async with` expression. We will learn more about asynchronous context managers in a later chapter, for now we will focus on how to use this specific one.

Entering the `asyncio.timeout()` context manager will set a time in the future when the context manager will give up waiting.

The timeout is provided in seconds, e.g. 5 seconds in the future, as recorded by the clock within the asyncio event loop.

For example:

```
...
# wait for 5 seconds
async with asyncio.timeout(5):
    # ...
```

We can then explicitly await long-running coroutines within the block of the `asyncio.timeout()` context manager.

For example:

```
...
# set a timeout
async with asyncio.timeout(5):
    # execute long running task
    result = await task()
```

The benefit of `asyncio.timeout()` being a context manager, means that we can await many coroutines, e.g. many sub-tasks, within the body and only cancel the one task that takes too long.

For example:

```
...
# set a timeout
async with asyncio.timeout(5):
    # execute many tasks
    await task1()
    await task2()
    await task3()
    await task4()
    await task5()
```

18.1.2 Handle Timeout Exception

If the wait time elapsed before the context manager is exited, any awaitables (e.g. coroutines and tasks) awaited within the block will be canceled and an `asyncio.TimeoutError` will be

raised.

We can handle the `asyncio.TimeoutError` and report and perform an action, like report a message of the timeout and cancellation of the task.

For example:

```
...
# handle timeout
try:
    # set a timeout
    async with asyncio.timeout(5):
        # execute long running task
        result = await task(1)
        # report the result
        print(result)
except asyncio.TimeoutError:
    print(f'Timeout waiting')
```

18.1.3 Reschedule Timeout

The `asyncio.timeout()` function creates an instance of the `asyncio.Timeout` class that has three methods:

- `asyncio.Timeout.when()`: Return the current deadline
- `asyncio.Timeout.reschedule()`: Change the current deadline to a new time.
- `asyncio.Timeout.when()`: `expired()` Return `True` if the timeout has expired, `False` otherwise.

It is possible to extend the timeout further into the future.

This can be used to either set no timeout initially, and then set a timeout later while running, or to extend a predefined timeout further into the future.

For example, we can set a timeout by passing `None` for the delay, then calling the `reschedule()` method on the `asyncio.Timeout` object created via the asynchronous context manager with a time in the future.

```
...
# set no timeout
async with asyncio.timeout(None) as timeout:
    # do something else
    # ...
    # calculate a deadline 5 seconds in the future
    loop = asyncio.get_running_loop()
    deadline = loop.time() + 5
    # set the new deadline
    timeout.reschedule(deadline)
    # execute long running task
```

```
    result = await task()
```

We can also set an initial timeout delay, then extend it into the future using the same approach.

For example:

```
...
# set a 5 second timeout
async with asyncio.timeout(5) as timeout:
    # do something else
    # ...
    # calculate a deadline 10 seconds in the future
    loop = asyncio.get_running_loop()
    deadline = loop.time() + 10
    # set the new deadline
    timeout.reschedule(deadline)
    # execute long running task
    result = await task()
```

Now that we know how to use `asyncio.timeout()`, let's look at some worked examples.

18.2 Example Of Timeout With A Long Running Task

We can explore an example of waiting for a long-running `asyncio.Task` with a timeout.

In this case, we will define a coroutine that sleeps for a long time and returns a value.

```
# long running task
async def task(value):
    # sleep to simulate waiting
    await asyncio.sleep(10)
    # return value
    return value * 100
```

We can then schedule this coroutine as a background task.

```
...
# schedule the task
running_task = asyncio.create_task(task(1))
```

Next, we can also allow the task to start running.

```
...
# allow the task to run
await asyncio.sleep(0)
```

We can then open the context manager and await this already scheduled task with a timeout that is not long enough to allow the task to complete.

The timeout will expire and a `asyncio.TimeoutError` will be raised that we can handle.

```
...
# handle timeout
try:
    # set a timeout
    async with asyncio.timeout(5):
        # wait for the task to complete
        result = await running_task
        # report the result
        print(result)
except asyncio.TimeoutError:
    print(f'Timeout waiting')
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of waiting for a task with a timeout
import asyncio

# long running task
async def task(value):
    # sleep to simulate waiting
    await asyncio.sleep(10)
    # return value
    return value * 100

# asyncio entry point
async def main():
    # schedule the task
    running_task = asyncio.create_task(task(1))
    # allow the task to run
    await asyncio.sleep(0)
    # handle timeout
    try:
        # set a timeout
        async with asyncio.timeout(5):
            # wait for the task to complete
            result = await running_task
            # report the result
            print(result)
    except asyncio.TimeoutError:
        print(f'Timeout waiting')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `task()` coroutine and schedules it as a task.

It then suspends the `main()` coroutine and allows the task to start running, beginning its sleep.

The `main()` coroutine resumes and opens the timeout context manager and awaits the task.

After 5 seconds, the `asyncio.timeout()` context manager resumes and cancels the `asyncio.Task` and raises an `asyncio.TimeoutError`.

This exception is handled and a failure message is reported.

This example highlights how we can wait for an already executing long-running `asyncio.Task` with a timeout, then cancel it after the timeout expires and handle the termination exception.

```
Timeout waiting
```

18.3 Example Of A Timeout Delay Set Later

We can explore an example of waiting for a long-running coroutine without an initial timeout, then adding a timeout later.

In this case, we can set no initial timeout.

```
...
# set a timeout
async with asyncio.timeout(None) as timeout:
    # ...
```

We can then perform some work and determine a timeout in the future and set this into the `asyncio.timeout()` context manager.

This can be achieved by first determining the current time as defined by the asyncio event loop, and adding a number of seconds to this time.

We can do this by first accessing the asyncio event loop object via the `asyncio.get_running_loop()` function, then calling the `time()` method on the event loop to access the current time.

```
...
# set a timeout
async with asyncio.timeout(None) as timeout:
    # wait a moment
    await asyncio.sleep(1)
    # set a deadline 5 seconds in the future
    loop = asyncio.get_running_loop()
    deadline = loop.time() + 5
    # set the new deadline
    timeout.reschedule(deadline)
```

This allows the timeout to be set later, on demand.

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of setting a timeout delay later
import asyncio

# long running task
async def task(value):
    # sleep to simulate waiting
    await asyncio.sleep(10)
    # return value
    return value * 100

# asyncio entry point
async def main():
    # handle timeout
    try:
        # set a timeout
        async with asyncio.timeout(None) as timeout:
            # wait a moment
            await asyncio.sleep(1)
            # set a deadline 5 seconds in the future
            loop = asyncio.get_running_loop()
            deadline = loop.time() + 5
            # set the new deadline
            timeout.reschedule(deadline)
            # execute long running task
            result = await task(1)
            # report the result
            print(result)
    except asyncio.TimeoutError:
        print(f'Timeout waiting')

# start the asyncio event loop
asyncio.run(main())
```

Running the example opens the `asyncio.timeout()` context manager without a timeout.

The `main()` coroutine sleeps to simulate other work.

A timeout of 5 seconds in the future is then determined and the timeout is set into the `asyncio.timeout()` context manager.

The `task()` coroutine is then issued and awaited.

After 5 seconds, the `asyncio.timeout()` context manager resumes and cancels the `task()` coroutine and raises an `asyncio.TimeoutError`.

The exception is handled and a message is reported.

This example highlights how we can set a timeout after we have entered the block and that this timeout has the normal effect of canceling the blocked task and raising an `asyncio.TimeoutError` exception.

```
Timeout waiting
```

18.4 Example Of Extending A Timeout

We can explore an example of waiting for a long-running coroutine with a timeout and then extending the timeout further into the future.

In this case, we will update the example with the long-running coroutine. A first timeout will be set to 5 seconds on the context manager, not enough time for the task to complete.

The program will perform other tasks, and then increase the timeout by another 11 seconds into the future.

It will then execute the long-running task, which will have enough time to complete.

```
...
# set a timeout
async with asyncio.timeout(5) as timeout:
    # wait a moment
    await asyncio.sleep(4)
    # set a deadline 5 seconds in the future
    loop = asyncio.get_running_loop()
    deadline = loop.time() + 11
    # set the new deadline
    timeout.reschedule(deadline)
    # execute long running task
    result = await task(1)
    # report the result
    print(result)
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of extending a timeout delay
import asyncio

# long running task
async def task(value):
    # sleep to simulate waiting
    await asyncio.sleep(10)
    # return value
    return value * 100

# asyncio entry point
```

```
async def main():
    # handle timeout
    try:
        # set a timeout
        async with asyncio.timeout(5) as timeout:
            # wait a moment
            await asyncio.sleep(4)
            # set a deadline 5 seconds in the future
            loop = asyncio.get_running_loop()
            deadline = loop.time() + 11
            # set the new deadline
            timeout.reschedule(deadline)
            # execute long running task
            result = await task(1)
            # report the result
            print(result)
    except asyncio.TimeoutError:
        print(f'Timeout waiting')

# start the asyncio event loop
asyncio.run(main())
```

Running the example opens the asyncio.`timeout()` context manager with a timeout of 5 seconds, not long enough for our task.

The `main()` coroutine sleeps for 4 seconds to simulate other work.

The `main()` coroutine resumes and updates the timeout to 11 seconds into the future. This is enough time to complete the task.

The `task()` coroutine is then issued and awaited.

The task completes normally and the return value is reported.

This example highlights how we can set a timeout on the context manager, then update that time out later as more information becomes available.

```
100
```

18.5 Takeaways

You now know how to wait for coroutines and tasks using the `timeout()` asynchronous context manager.

Specifically, you know:

- What is `timeout()` and how can we use it to wait for tasks to complete with a timeout.

- How to manage the case when the timeout is exceeded before the required tasks are complete.
- How to configure the timeout and how to extend the timeout if more time is needed.

18.5.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- What's New In Python 3.11.
  https://docs.python.org/3/whatsnew/3.11.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

18.5.2 Next

In the next tutorial, we will explore how to shield asyncio programs from cancellation.

Part IV

More Tasks

Chapter 19

Shield Tasks

Asyncio tasks can be canceled at any time.

This can cause a running task to stop mid-execution, which can cause problems if we expect a task or subtask to complete as an atomic operation.

Asyncio provides a way to shield tasks from cancellation, making them directly immune from being canceled.

In this tutorial, you will discover how to shield asyncio tasks from cancellation.

After completing this tutorial, you will know:

- How to shield asyncio tasks from cancellation.
- How shields work and how we can work around them if we really need to.
- How to develop worked examples that use shields to protect tasks from cancellation.

Let's get started.

19.1 How To Shield Tasks From Cancellation

A task can be protected from cancellation by a shield.

This is a task that wraps the target and absorbs requests to cancel.

19.1.1 Shield Tasks and Coroutines

The `asyncio.shield()` function will protect a target `Task` or coroutine from being canceled.

It takes an awaitable as an argument and returns an awaitable object.

The returned awaitable object can then be awaited directly or passed to another task or coroutine.

For example:

```
...
# shield a task from cancellation
shielded = asyncio.shield(task)
# await the shielded task
await shielded
```

19.1.2 Shield Absorbs Cancel Requests

The returned awaitable can be canceled by calling the `cancel()` method.

If the inner task is running, the request will be reported as successful.

For example:

```
...
# cancel a shielded task
was_canceld = shielded.cancel()
```

Any coroutines awaiting the returned awaitable object will raise an `asyncio.CancelledError` exception, which may need to be handled.

For example:

```
...
try:
    # await the shielded task
    await asyncio.shield(task)
except asyncio.CancelledError:
    # ...
```

Importantly, the request for cancellation made on the returned awaitable object is not propagated to the inner awaitable.

This means that the request for cancellation is absorbed by the shield.

For example:

```
...
# create a task
task = asyncio.create_task(coro())
# create a shield
shield = asyncio.shield(task)
# cancel the shield (does not cancel the task)
shield.cancel()
```

19.1.3 Inner Task Can Still Be Canceled

If the task that is being shielded is canceled, the cancellation request will be propagated up to the shield, which will also be canceled.

This provides a way to cancel a shielded task, if needed.

For example:

```
...
# create a task
task = asyncio.create_task(coro())
# create a shield
shield = asyncio.shield(task)
# cancel the task (also cancels the shield)
task.cancel()
```

Now that we know how to use the `asyncio.shield()` function, let's look at some worked examples.

19.2 Example Shielding A Task From Cancellation

We can explore how to protect a task from cancellation using `asyncio.shield()`.

In this example, we define a simple coroutine task that takes an integer argument, sleeps for a second, then returns the argument. The coroutine can then be created and scheduled as a `Task`.

We can define a second coroutine that takes a task, sleeps for a fraction of a second, then cancels the provided task.

In the main coroutine, we can then shield the first task and pass it to the second task, then await the shielded task.

The expectation is that the shield will be canceled and leave the inner task intact. The cancellation will disrupt the main coroutine. We can check the status of the inner task at the end of the program and we expect it to have been completed normally, regardless of the request to cancel made on the shield.

The complete example is listed below.

```
# SuperFastPython.com
# example of shielding a task from cancellation
import asyncio

# define a simple asynchronous
async def simple_task(number):
    # block for a moment
    await asyncio.sleep(1)
    # return the argument
    return number

# cancel the given task after a moment
async def cancel_task(task):
    # block for a moment
```

```
    await asyncio.sleep(0.2)
    # cancel the task
    was_cancelled = task.cancel()
    print(f'cancelled: {was_cancelled}')

# define a simple coroutine
async def main():
    # create the coroutine
    coro = simple_task(1)
    # create a task
    task = asyncio.create_task(coro)
    # created the shielded task
    shielded = asyncio.shield(task)
    # create the task to cancel the previous task
    asyncio.create_task(cancel_task(shielded))
    # handle cancellation
    try:
        # await the shielded task
        result = await shielded
        # report the result
        print(f'>got: {result}')
    except asyncio.CancelledError:
        print('shielded was cancelled')
    # wait a moment
    await asyncio.sleep(1)
    # report the details of the tasks
    print(f'shielded: {shielded}')
    print(f'task: {task}')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the program.

The task coroutine is created, then it is wrapped and scheduled in a `Task`.

The task is then shielded from cancellation.

The shielded task is then passed to the `cancel_task()` coroutine which is wrapped in a task and scheduled.

The main coroutine then awaits the shielded task, which expects a `CancelledError` exception.

The task runs for a moment then sleeps. The cancellation task runs for a moment, sleeps, resumes then cancels the shielded task. The request to cancel reports that it was successful.

This raises a `CancelledError` exception in the shielded awaitable, although not in the inner task.

The `main()` coroutine resumes and responds to the `CancelledError` exception, reporting a message. It then sleeps for a while longer.

The task resumes, finishes, and returns a value.

Finally, the `main()` coroutine resumes, and reports the status of the shielded awaitable and the inner task. We can see that the shielded awaitable is marked as canceled and yet the inner task is marked as finished normally and provides a return value.

This example highlights how a shield can be used to successfully protect an inner task from cancellation.

```
cancelled: True
shielded was cancelled
shielded: <Future cancelled>
task: <Task finished name='Task-2'
    coro=<simple_task() done, defined at ...>
    result=1>
```

Next, let's look at shielding a coroutine, instead of a task, from cancellation.

19.3 Example Shielding A Coroutine From Cancellation

We can explore how to protect a coroutine from cancellation using `asyncio.shield()`.

In this example, we update the above example to shield a coroutine directly, instead of a task.

The expectation is that the coroutine will be wrapped in a task immediately and scheduled for execution. The request for cancellation made on the shield will protect the inner coroutine from cancellation as it does for tasks.

At the end of the program, we will locate the `asyncio.Task` associated with the shielded coroutine and report its status. The expectation is that it will have finished normally and was not canceled.

The complete example is listed below.

```
# SuperFastPython.com
# example of shielding a coroutine from cancellation
import asyncio

# define a simple asynchronous
async def simple_task(number):
    # block for a moment
    await asyncio.sleep(1)
```

```
    # return the argument
    return number

# cancel the given task after a moment
async def cancel_task(task):
    # block for a moment
    await asyncio.sleep(0.2)
    # cancel the task
    was_cancelled = task.cancel()
    print(f'cancelled: {was_cancelled}')

# define a simple coroutine
async def main():
    # create the coroutine
    coro = simple_task(1)
    # created the shielded task
    shielded = asyncio.shield(coro)
    # create the task to cancel the previous task
    asyncio.create_task(cancel_task(shielded))
    # handle cancellation
    try:
        # await the shielded task
        result = await shielded
        # report the result
        print(f'>got: {result}')
    except asyncio.CancelledError:
        print('shielded was cancelled')
    # get all tasks
    tasks = asyncio.all_tasks()
    # wait a moment
    await asyncio.sleep(1)
    # report the details of the tasks
    print(f'shielded: {shielded}')
    # report the task for the coroutine
    for task in tasks:
        if task.get_coro() is coro:
            print(f'task: {task}')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the program.

The task coroutine is created. It is then shielded from cancellation. Internally this wraps the

coroutine in a `Task` object and schedules it for execution. We will retrieve this `Task` object later.

The shielded task is then passed to the `cancel_task()` coroutine which is wrapped in a task and scheduled.

The main coroutine then awaits the shielded task, which expects a `CancelledError` exception.

The task runs for a moment then sleeps. The cancellation task runs for a moment, sleeps, resumes, then cancels the shielded task. The request to cancel reports that it was successful.

This raises a `CancelledError` exception in the shielded awaitable, although not in the inner task or coroutine.

The `main()` coroutine resumes and responds to the `CancelledError` exception, reporting a message. It then sleeps for a while longer.

The inner task resumes, finishes, and returns a value.

Finally, the `main()` coroutine resumes, reporting the status of the shielded awaitable. It then locates the `Task` associated with the inner coroutine and reports its status.

We can see that the shielded awaitable is marked as canceled and yet the inner task for the shielded coroutine is marked as finished normally and provides a return value.

This example highlights how a shield can be used to successfully protect an inner coroutine from cancellation, and that the `shield()` function will create an `asyncio.Task` object for a provided coroutine.

```
cancelled: True
shielded was cancelled
shielded: <Future cancelled>
task: <Task finished name='Task-2'
    coro=<simple_task() done, defined at ...>
    result=1>
```

Next, let's look at what happens when we cancel the task within a shield.

19.4 Example Of Canceling Shielded Inner Task

We can explore what happens when the inner shielded task is canceled.

In this example, we will pass the inner task to the cancellation coroutine.

The expectation is that the inner task will be canceled and that this request for cancellation will be propagated out to the shield and then impact the main coroutine.

The complete example is listed below.

```
# SuperFastPython.com
# example of canceling a shielded task directly
```

```
import asyncio

# define a simple asynchronous
async def simple_task(number):
    # block for a moment
    await asyncio.sleep(1)
    # return the argument
    return number

# cancel the given task after a moment
async def cancel_task(task):
    # block for a moment
    await asyncio.sleep(0.2)
    # cancel the task
    was_cancelled = task.cancel()
    print(f'cancelled: {was_cancelled}')

# define a simple coroutine
async def main():
    # create the coroutine
    coro = simple_task(1)
    # create a task
    task = asyncio.create_task(coro)
    # created the shielded task
    shielded = asyncio.shield(task)
    # create the task to cancel the previous task
    asyncio.create_task(cancel_task(task))
    # handle cancellation
    try:
        # await the shielded task
        result = await shielded
        # report the result
        print(f'>got: {result}')
    except asyncio.CancelledError:
        print('shielded was cancelled')
    # wait a moment
    await asyncio.sleep(1)
    # report the details of the tasks
    print(f'shielded: {shielded}')
    print(f'task: {task}')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the application.

The task coroutine is created, then it is wrapped and scheduled in a `Task`.

The task is then shielded from cancellation.

The task itself, not the shielded task, is then passed to the `cancel_task()` coroutine that is wrapped in a task and scheduled.

The main coroutine then awaits the shielded task, which expects a `CancelledError` exception.

The task runs for a moment then sleeps. The cancellation task runs for a moment, sleeps, resumes then cancels the task directly. The request to cancel reports that it was successful.

This raises a `CancelledError` exception in the task itself which cancels. The `CancelledError` exception is then raised in the shielded awaitable object, which also cancels.

The `main()` coroutine resumes and responds to the `CancelledError` exception, reporting a message. It then sleeps for a while longer.

Finally, the `main()` coroutine resumes, and reports the status of the shielded awaitable and the inner task.

We can see that both the shielded awaitable object and the inner `Task` are marked as canceled.

This example highlights that although a `Task` can be shielded, it can still be canceled directly.

```
cancelled: True
shielded was cancelled
shielded: <Future cancelled>
task: <Task cancelled name='Task-2'
    coro=<simple_task() done, defined at ...>>
```

19.5 Takeaways

You now know how to shield asyncio tasks from cancellation.

Specifically, you know:

- How to shield asyncio tasks from cancellation.
- How shields work and how we can work around them if we really need to.
- How to develop worked examples that use shields to protect tasks from cancellation.

19.5.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Exceptions.
  https://docs.python.org/3/library/asyncio-exceptions.html

19.5.2 Next

In the next tutorial, we will explore how to suspend the execution of asyncio tasks with sleep.

Chapter 20

Sleep Tasks

Asyncio tasks will run until they choose to suspend and yield the control of execution.

This can be a problem in some tasks that call regular Python functions and methods, as they may not give an opportunity for other tasks in the asyncio event loop to run.

Asyncio provides the ability for tasks to explicitly yield control on demand by sleeping.

In this tutorial, you will discover how to sleep tasks in asyncio.

After completing this tutorial, you will know:

- How to sleep tasks in asyncio programs.
- The difference between `asyncio.sleep()` coroutine and the regular `time.sleep()` function.
- How we can yield control within a task without sleeping for any time.

Let's get started.

20.1 How To Use Asyncio Sleep

A coroutine can suspend itself by sleeping.

20.1.1 Sleep A Coroutine

The `asyncio.sleep()` function is a module function that will suspend the caller for a given number of seconds.

The `sleep()` function takes two arguments.

The first argument is `delay` and defines the time in seconds that the current task or coroutine will suspend.

For example:

```
...
# sleep for 1 second
await asyncio.sleep(1)
```

20.1.2 Return A Value After Sleep

The second argument to `asyncio.sleep()` is `result` which will return the provided value after a given the sleep has been completed.

For example:

```
...
# sleep for one second and return 100
value = await asyncio.sleep(1, result=100)
```

Returning a value can be helpful within an expression, providing a way to delay the release of a value.

For example:

```
...
# delay a conditional expression
if value > asyncio.sleep(5, 100):
    # ...
```

20.1.3 Sleep Returns An Awaitable

The `asyncio.sleep()` function itself does not block, but instead, it returns an awaitable object that can be awaited.

Specifically, the `asyncio.sleep()` function is a coroutine function that returns a coroutine object.

This coroutine can be stored in a variable and awaited at a later time.

For example:

```
...
# get the sleep coroutine
coro = asyncio.sleep(1)
...
# sleep by awaiting the sleep coroutine
await coro
```

20.1.4 Sleep For Milliseconds

The `asyncio.sleep()` function can be used to sleep the current task or coroutine for a given number of milliseconds.

Recall that one second is 1,000 milliseconds.

We can specify the number of milliseconds to sleep as a fraction of a second.

For example, 0.100 seconds equals 100 milliseconds or 1/10th of a second.

```
...
# sleep for 100 milliseconds
await asyncio.sleep(0.100)
```

20.1.5 Sleep For Seconds

The `asyncio.sleep()` function takes a time to suspend in seconds, as we have seen.

We can sleep for a given number of seconds, such as 1, 10, or 60.

For example:

```
...
# sleep for 10 seconds
await asyncio.sleep(10)
```

We can also sleep for zero seconds.

This is a special sleep period. It means that the current task or coroutine will be suspended and the asyncio event loop will allow other tasks or coroutines to execute.

The current coroutine or task will then resume as soon as it is able, which may be more than 0 seconds.

For example:

```
...
# sleep for zero seconds
await asyncio.sleep(0)
```

It is a way that one task can give up control for a moment to allow other tasks to execute.

20.1.6 Sleep For Minutes

The `asyncio.sleep()` function can be used to block the current task for minutes.

This may be helpful for a long-running task or a task that is required to perform a task periodically, such as every minute or every hour.

This can be achieved by determining the number of minutes to sleep and multiplying the value by 60 to convert it to seconds, before providing it to the `sleep()` function as an argument.

For example:

```
...
# sleep for 15 minutes
await asyncio.sleep(15 * 60)
```

Now that we know how to use the `asyncio.sleep()` function, let's consider when we might use it.

20.2 When To Use Sleep

The `sleep()` function has many uses.

In practice, it can be used to allow a task to wait a fixed time before performing an action.

This can be helpful in order to avoid overloading a resource, such as a remote server or database.

For example:

```
...
# wait a moment before querying again
await asyncio.sleep(0.5)
```

A sleep can also be used to wait for a condition in a program, such as for another task to complete or for a variable to be set to `True`.

This is called a wait loop and allows the waiting task to perform actions while also waiting for a condition.

For example:

```
...
# run forever
while True:
    # check condition
    if condition():
        # stop waiting
        break
    # otherwise sleep for a moment
    await asyncio.sleep(0.2)
```

Another important use for sleep in asyncio programs is to suspend the current task and allow other coroutines to execute.

It is important because although a task or coroutine can easily schedule new tasks via the `create_task()` or `gather()` function, the scheduled tasks will not begin executing until the current task is suspended.

Sleeping for zero seconds will suspend the current task and give an opportunity to other tasks to run.

For example:

```
...
# allow other tasks to run for a moment
await asyncio.sleep(0)
```

This is not a hack, it is a way that we can allow recently scheduled tasks an opportunity to begin, a voluntary suspend.

Finally, a good use for sleep is to simulate non-blocking tasks in a concurrent program.

Using calls to `asyncio.sleep()` in place of real file or socket I/O allows us to develop, test, configure, and refine concurrent real-world programs without the need to use resources.

Now that we know when to use the `sleep()` function, let's look at how it compares to other sleep functions.

20.3 Asyncio Sleep Versus Time Sleep

Another sleep function in Python is provided by the `time` module called `time.sleep()`.

For example:

```
...
# sleep for one seconds
time.sleep(1)
```

Like `asyncio.sleep()`, the `time.sleep()` takes a time in seconds and returns after the given interval.

Both functions suspend or block, but the difference is the level of concurrency that is blocked.

For example:

- `time.sleep()`: blocks the current thread. While blocked, other threads may run.
- `asyncio.sleep()`: blocks the current coroutine or task. While blocked, other tasks may run.

Recall that one thread may run many coroutines, in fact, many thousands of coroutines. In turn, a process may run many threads, often many thousands of threads.

- One system runs many processes (multiprocessing concurrency).
- One process runs many threads (threading concurrency).
- One thread runs many coroutines (asyncio concurrency).

Blocking the thread will block all coroutines running in that thread, if that thread is running an asyncio event loop.

We do not want to block the thread in an asyncio program, it will halt all coroutine-based concurrency, which would be undesirable.

Therefore we do not want to call `time.sleep()` within an asyncio program, at least not directly.

We may want to call `time.sleep()` within a multithreaded program to allow other threads in the program to run.

Another important difference is that calling `time.sleep()` blocks immediately. It calls a sleep routine in the C code and ultimately a sleep routine in the underlying operating system.

The `asyncio.sleep()` function technically does not block. In fact, it returns immediately. The value it returns is a coroutine object that a program can choose to await or not.

- `time.sleep()`: Blocks the current thread, returns nothing.
- `asyncio.sleep()`: Returns a coroutine object that can be awaited.

The coroutine object returned from `asyncio.sleep()` is scheduled to resume at a time in the future. In the mean time, the event loop can continue to progress any other running tasks.

Now that we know all about the `asyncio.sleep()` function, let's look at some worked examples.

20.4 Example Of Sleeping A Task

We can call sleep from a Task.

Recall that an asyncio task is just a coroutine that is wrapped in an `asyncio.Task` object so that it can be scheduled and executed independently within the asyncio event loop.

The example below creates a task that reports a message, sleeps for a moment then reports a final message before terminating.

The task is created in the main coroutine which then waits for it to be done.

The complete example is listed below.

```
# SuperFastPython.com
# example of sleeping a task
import asyncio

# custom coroutine
async def custom_coro():
    # report a message
    print('task running')
    # block for a moment
    await asyncio.sleep(1)
    # report a message
    print('task done')

# entry point coroutine
async def main():
    # execute another task
    await asyncio.create_task(custom_coro())

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point to the program.

The `main()` coroutine then creates our custom coroutine and uses it to create a task and

suspends, waiting for the task completed.

The task runs. It reports a message, sleeps for one second, then reports a final message.

This highlights how we can sleep within an asyncio task, just like we do in a coroutine.

```
task running
task done
```

Next, let's look at the return value from the `sleep()` function.

20.5 Example Sleep With A Return Value

The `asyncio.sleep()` function returns a value.

Although, this may not be clear from the usage of the `sleep()` function in most programs.

Specifically, the `sleep()` function returns immediately with a coroutine object. This object can then be awaited.

Before awaiting it, we can inspect it, such as reporting its type and details.

This is a good practice in order to better understand what exactly the sleep function is doing.

The example below explores the coroutine object returned from the `sleep()` function.

```
# SuperFastPython.com
# example of the return value from sleep
import asyncio

# entry point coroutine
async def main():
    # get the sleep awaitable
    awaitable = asyncio.sleep(0.1)
    # report the awaitable
    print(type(awaitable))
    print(awaitable)
    # await the awaitable
    await awaitable

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point to the program.

The `main()` coroutine then calls the sleep function which does not block, but instead returns immediately with an awaitable.

We then report the type of the returned awaitable. We can see that it is a coroutine.

When they report the details of the object and can see that it is a coroutine created from the `sleep()` coroutine definition, as we expected.

Finally, we await the sleep coroutine, suspending the main coroutine for a moment and allowing the sleep coroutine to run, which does nothing other than suspend the caller, but does allow other coroutines to run, if they were present.

The sleep coroutine returns, then our `main()` coroutine terminates.

This example highlights the non-blocking coroutine nature of the sleep function.

```
<class 'coroutine'>
<coroutine object sleep at ...>
```

Next, let's explore the case of sleeping for zero seconds.

20.6 Example Of Sleeping Zero Seconds

A common asyncio idiom is to sleep for zero seconds.

For example:

```
...
# sleep for zero seconds
asyncio.sleep(0)
```

This idiom is common because it suspends the current coroutine and allows the event loop to execute one or more other coroutines that may have been scheduled or suspended.

It is commonly used right after scheduling one or more coroutines as tasks, such as after calling `asyncio.create_task()` or calling `asyncio.gather()`.

The example below demonstrates this.

A task is created which schedules it for execution. The calling coroutine then sleeps for zero seconds, which allows the scheduled task to begin running and sleep itself.

The main coroutine resumes then awaits the task to complete entirely.

```
# SuperFastPython.com
# example of sleeping for zero seconds
import asyncio

# custom coroutine
async def custom_coro():
    # report a message
    print('task running')
    # block for a moment
    await asyncio.sleep(1)
    # report a message
    print('task done')
```

```
# entry point coroutine
async def main():
    # execute another coroutine
    task = asyncio.create_task(custom_coro())
    print('main is blocking now')
    # sleep while the task is running
    await asyncio.sleep(0)
    # report a message
    print('main is done blocking')
    # wait for the task to complete
    await task

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point to the program.

The `main()` coroutine then creates our custom coroutine and schedules it as a task. It then sleeps for zero seconds and allows the scheduled task to begin executing.

The task begins, first reporting a message, then sleeping for one second.

The `main()` coroutine resumes, ropers a message of its own, and then suspends again, awaiting the task.

The task resumes, reports a message then terminates.

The example highlights how we might allow scheduled tasks to begin their execution and interleave their execution with other tasks.

```
main is blocking now
task running
main is done blocking
task done
```

Next, let's explore how we might create a periodic task using the `sleep()` function.

20.7 Example Of Periodic Task With Sleep

The `asyncio.sleep()` function can be used to create a task that executes periodically.

This can be achieved by creating a loop that runs forever, or a fixed time, that performs a custom operation each iteration and then sleeps for a fixed interval.

This way, a custom operation can be performed every fraction of a second, every second, every minute, or whatever time interval is required for the application.

The example below demonstrates this.

A periodic task is defined that runs every 200 milliseconds. The operation performed is to just report a message, but it could be anything we like, such as checking a resource or refreshing some data.

The main coroutine creates and schedules the task and then simulates going on with other tasks before finally terminating the program.

```
# SuperFastPython.com
# example of a sleep in a periodic task
import asyncio

# periodic task
async def periodic():
    # loop forever
    while True:
        # perform operation
        print('>task is running')
        # block for an interval
        await asyncio.sleep(0.2)

# entry point coroutine
async def main():
    # report a message
    print('Main is starting')
    # start the periodic task
    _ = asyncio.create_task(periodic())
    # report a message
    print('Main is resuming with work...')
    # wait a while for some reason
    await asyncio.sleep(3)
    # report a message
    print('Main is done')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point to the application.

The `main()` coroutine then creates and schedules the periodic task.

It then continues on with its simulated work, choosing to report a message and sleep for a while.

The task runs, looping forever.

Each iteration it performs its task which is to report a message and then sleeps for about 200 milliseconds.

The `main()` coroutine resumes after 3 seconds or about 15-16 periods of the task and reports a final message before terminating the application.

```
Main is starting
Main is resuming with work...
>task is running
>task is running
>task is running
>task is running
>task is running
>task is running
>task is running
>task is running
>task is running
>task is running
>task is running
>task is running
>task is running
>task is running
>task is running
Main is done
```

20.8 Takeaways

You now know how to use sleep in asyncio.

Specifically, you know:

- How to sleep tasks in asyncio programs.
- The difference between `asyncio.sleep()` coroutine and the regular `time.sleep()` function.
- How we can yield control within a without sleeping for any time.

20.8.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- `time` – Time access and conversions.
  https://docs.python.org/3/library/time.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

20.8.2 Next

In the next tutorial, we will explore how to wait on another asyncio task with a timeout.

Chapter 21

Wait For Tasks

It is a good practice that any waiting performed in an asyncio program be limited to a timeout.

This is because something may go wrong when waiting on a resource and the program may be blocked for an extended period of time. Instead, we should wait for a reasonable amount of time and then take action, such as retry or give up.

Asyncio provides a way to wait on another task with a timeout via the `asyncio.wait_for()` function. If the timeout elapses before the task completes, the task is canceled and the caller receives an exception.

In this tutorial, you will discover how to wait for an asyncio task with a timeout.

After completing this tutorial, you will know:

- How to wait on another task with a timeout.
- How to handle a timeout while waiting on another task.
- How to handle the failure or cancellation of task that is being waited on with a timeout.

Let's get started.

21.1 How To Use Asyncio Wait For

The `asyncio.wait_for()` function takes an `awaitable` and a `timeout` as arguments.

The awaitable may be a coroutine or a task.

A `timeout` must be specified and may be `None` for no timeout, or floating point number of seconds.

The `wait_for()` function returns a coroutine that is not executed until it is explicitly awaited or scheduled as a task.

For example:

```
...
# wait for a task to complete
await asyncio.wait_for(coro, timeout=10)
```

If a coroutine is provided as an argument, it will be converted to the task when the `wait_for()` coroutine is executed.

If the `timeout` elapses before the task is completed, the task is canceled, and an `asyncio.TimeoutError` is raised, which may need to be handled.

For example:

```
...
# execute a task with a timeout
try:
    # wait for a task to complete
    await asyncio.wait_for(coro, timeout=1)
except asyncio.TimeoutError:
    # ...
```

If the awaited task fails with an unhandled exception, the exception will be propagated back to the caller that is awaiting on the `wait_for()` coroutine, in which case it may need to be handled.

For example

```
...
# execute a task that may fail
try:
    # wait for a task to complete
    await asyncio.wait_for(coro, timeout=1)
except asyncio.TimeoutError:
    # ...
except Exception:
    # ...
```

Next, let's look at some worked examples.

21.2 Example Of Waiting With A Timeout

We can explore how to wait for a coroutine with a timeout that elapses before the task is completed.

In this example, we execute a coroutine as above, except the caller waits a fixed timeout of 0.2 seconds or 200 milliseconds.

Recall that one second is equal to 1,000 milliseconds.

The task coroutine is modified so that it sleeps for more than one second, ensuring that the timeout always expires before the task is complete.

The complete example is listed below.

```
# SuperFastPython.com
# example of waiting for a coroutine with a timeout
from random import random
import asyncio

# coroutine to execute in a new task
async def task_coro(arg):
    # generate a random value between 0 and 1
    value = 1 + random()
    # report message
    print(f'>task got {value}')
    # block for a moment
    await asyncio.sleep(value)
    # report all done
    print('>task done')

# main coroutine
async def main():
    # create a task
    task = task_coro(1)
    # execute and wait for the task without a timeout
    try:
        await asyncio.wait_for(task, timeout=0.2)
    except asyncio.TimeoutError:
        print('Gave up waiting, task canceled')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine creates the task coroutine. It then calls `wait_for()` and passes the task coroutine and sets the timeout to 0.2 seconds.

The `main()` coroutine is suspended and the `task_coro()` is executed. It reports a message and sleeps for a moment.

The `main()` coroutine resumes after the timeout has elapsed. The `wait_for()` coroutine cancels the `task_coro()` coroutine and the `main()` coroutine is suspended.

The `task_coro()` runs again and responds to the request to be terminated. It raises a `TimeoutError` exception and terminates.

The `main()` coroutine resumes and handles the `TimeoutError` raised by the `task_coro()`.

This highlights how we can call the `wait_for()` function with a timeout and to cancel a task if it is not completed within a timeout.

The output from the program will differ each time it is run given the use of random numbers.

```
>task got 0.685375224799321
Gave up waiting, task canceled
```

Next, let's look at what happens if we wait on a task that fails with an unhandled exception.

21.3 Example Of Waiting On A Task That Fails

We can explore how to wait for a coroutine with a timeout and a task that fails with an exception.

If a waited-for task fails with an unhandled exception, the exception is propagated back to the caller.

In this example, we update the task coroutine so that it fails with an unhandled exception after sleeping. The caller waits for the task to complete and the exception is propagated back to the caller and handled.

The complete example is listed below.

```
# SuperFastPython.com
# example of waiting for a coroutine that fails
from random import random
import asyncio

# coroutine to execute in a new task
async def task_coro(arg):
    # generate a random value between 0 and 1
    value = 1 * random()
    # report message
    print(f'>task got {value}')
    # block for a moment
    await asyncio.sleep(value)
    # fail with an exception
    raise Exception('Something bad happened')
    # report all done (never reached)
    print('>task done')

# main coroutine
async def main():
    # create a task
    task = task_coro(1)
    # execute and wait for the task without a timeout
```

```
    try:
        await asyncio.wait_for(task, timeout=2.0)
    except asyncio.TimeoutError:
        print('Gave up waiting, task canceled')
    except Exception as e:
        print(f'Task failed with: {e}')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine creates the task coroutine. It then calls `wait_for()` and passes the task coroutine and sets a timeout.

The `main()` coroutine is suspended and the `task_coro()` is executed. It reports a message, sleeps a moment, then raises an unhandled exception.

The `main()` coroutine resumes, catches the unhandled exception, and reports a message

This highlights how unhandled exceptions raised in the waited-for task or coroutine are propagated to the caller and may need to be handled.

The output from the program will differ each time it is run given the use of random numbers.

```
>task got 0.5142627201191862
Task failed with: Something bad happened
```

Next, let's look at what happens if we wait on a task that is canceled.

21.4 Example Waiting On A Task That Is Cancelled

We can explore how to wait for a task with a timeout and the waited-for task is canceled by another task.

If the waited-for task is canceled, the `wait_for()` coroutine stops waiting, and the `asyncio.CancelledError` exception is propagated to the caller.

In the example below we wrap and schedule our task coroutine in an `asyncio.Task` object so that it can be canceled.

We then pass the task to the `wait_for()` function and retrieve the coroutine, although do not await it yet.

We then create a second task for a coroutine that takes a task object, waits a moment, and cancels it. This task is scheduled.

Finally, the `wait_for()` coroutine is awaited with a timeout.

The waited-for task is canceled and the `asyncio.CancelledError` exception is propagated back to the caller and handled.

The complete example is listed below.

```
# SuperFastPython.com
# example of waiting for a task that is canceled
from random import random
import asyncio

# coroutine to execute in a new task
async def task_coro(arg):
    # generate a random value between 0 and 1
    value = 1 + random()
    # report message
    print(f'>task got {value}')
    # block for a moment
    await asyncio.sleep(value)
    # report all done
    print('>task done')

# another coroutine that cancels a task
async def task_cancel(other_task):
    # wait a moment
    await asyncio.sleep(0.3)
    # cancel the other task
    other_task.cancel()

# main coroutine
async def main():
    # create a task
    task = asyncio.create_task(task_coro(1))
    # create the wait for coroutine
    wait_coro = asyncio.wait_for(task, timeout=1)
    # create and run the cancel task
    asyncio.create_task(task_cancel(task))
    # await the wait-for coroutine
    try:
        await wait_coro
    except asyncio.TimeoutError:
        print('Gave up waiting, task canceled')
    except asyncio.CancelledError:
        print('Task was canceled externally')
        print(task)

# start the asyncio event loop
```

```
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine creates and schedules the `task_coro()` as a task.

The `wait_for()` is then called to wait on the task with a timeout of one second and the returned coroutine is assigned.

Next, the `task_cancel()` coroutine is wrapped in a task and scheduled for execution, and passed the task object for the `task_coro()` coroutine.

The `main()` coroutine is suspended and the `task_coro()` is executed.

The `task_coro()` task reports a message and then sleeps. The `task_cancel()` sleeps, then resumes and cancels the `task_coro()` task.

The `task_coro()` task resumes and raises an `asyncio.CancelledError` exception.

The `main()` coroutine resumes and the `asyncio.CancelledError` exception is re-raised and handled. The details of the waited-for task are reported and it shows that it is done and was canceled.

This highlights how a waited-for task can be canceled externally and that this may need to be handled.

The output from the program will differ each time it is run given the use of random numbers.

```
>task got 1.5848230431004935
Task was canceled externally
<Task cancelled name='Task-2' coro=<task_coro() done,
    defined at ...>>
```

21.5 Takeaways

You now know how to wait for an asyncio task with a timeout.

Specifically, you know:

- How to wait on another task with a timeout.
- How to handle a timeout while waiting on another task.
- How to handle the failure or cancellation of task that is being waited on with a timeout.

21.5.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- `random` – Generate pseudo-random numbers.
  https://docs.python.org/3/library/random.html

- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html
- Asyncio Exceptions.
  https://docs.python.org/3/library/asyncio-exceptions.html

21.5.2 Next

In the next tutorial, we will explore how to correctly execute and await blocking tasks in asyncio programs.

Chapter 22

Blocking Tasks

Calling regular functions and methods in asyncio programs will block the asyncio event loop.

This is typically fine for functions that execute briefly. It can be a problem for functions that block, such as performing regular (blocking) I/O or performing a long-running computation. This type of waiting will block the thread that is running the event loop and prevent all other tasks from running.

Luckily asyncio provides ways to execute blocking function calls in a way that simulates an asynchronous call that can be awaited, allowing other asyncio tasks to run and make progress.

In this tutorial, you will discover how to execute blocking functions in new threads separate from the asyncio event loop.

After completing this tutorial, you will know:

- The problem with executing blocking calls in asyncio programs.
- How to execute blocking function calls with a thread pool.
- How to choose to run a blocking task in a separate thread or process.

Let's get started.

22.1 Problem With Blocking The Event Loop

An issue in asyncio programs is that any time a regular Python function is executed, it blocks the event loop.

Instead of simply suspending the current coroutine, a blocking call will suspend the thread in which the coroutine is running. We discussed an example of this in the chapter on `asyncio.sleep()`.

This means that calling Python functions will stop all other asyncio tasks from progressing.

Typically this is not a problem for short functions that complete quickly. We don't want to transform all functions to be coroutines so that we support "*async all the way down*".

Nevertheless, blocking function calls can be a problem for longer running tasks that may block due to reading or writing synchronously rather than asynchronously, such as:

- Reading or writing a file.
- Reading or writing stdout, stderr, and stdin.
- Reading or writing a device, like a camera or mic.
- Reading or writing a blocking socket connection.

It may also be a problem when performing a CPU intensive operation, such as:

- Mathematical computation.
- Simulation.
- Model fitting or inference.
- Parsing data.

The solution is to run blocking function calls in a separate thread or process.

22.2 How To Execute Blocking Tasks In Asyncio

There are two main ways to run a blocking task in an asyncio program, they are:

1. With `asyncio.to_thread()`.
2. With `loop.run_in_executor()`.

Let's take a closer look at each in turn.

22.2.1 Run Task In Thread

We can execute blocking function calls in a separate thread using the `asyncio.to_thread()` function.

This function will execute a target function in a thread separate from the thread that is executing the asyncio event loop.

It does this using a thread pool behind the scenes and returns a coroutine object that can be awaited, allowing the blocking call to simulate and be treated like an asynchronous call.

The `to_thread()` function takes the name of a blocking function to execute and any arguments to the function. It then returns a coroutine that can be awaited to get the return value from the function, if any.

For example:

```
...
# create a coroutine for a blocking function
blocking_coro = asyncio.to_thread(blocking, arg1, arg2)
# await the coroutine and get return value
result = await blocking_coro
```

The blocking function will not be executed in a new thread until it is awaited or executed indirectly.

The coroutine can be wrapped in an `asyncio.Task` to execute the blocking function call independently.

For example:

```
...
# create a coroutine for a blocking function
blocking_coro = asyncio.to_thread(blocking)
# execute the blocking function independently
task = asyncio.create_task(blocking_coro)
```

This allows the blocking function call to be used like any other asyncio `Task`.

22.2.2 Run Task In Executor

The `asyncio.to_thread()` function is specifically designed to execute blocking I/O functions, not CPU-bound functions that might also block the asyncio event loop.

Instead, we must execute CPU-bound functions using process-based concurrency. This is because the Global Interpreter Lock (GIL) prevents more than one thread from executing at a time, except in some cases, such as when performing blocking I/O. No such limitation exists when executing code in separate processes.

Sadly, there is currently no `asyncio.to_process()` function in the `asyncio` module at the time of writing.

Instead, we can use the `run_in_executor()` method on the event loop.

This is part of the low-level asyncio API and first requires access to the event loop, such as via the `asyncio.get_running_loop()` function.

The `loop.run_in_executor()` function takes an executor and a function to execute.

If `None` is provided for the executor, then the default executor is used, which is a thread pool provided by the `ThreadPoolExecutor` class.

The `loop.run_in_executor()` function returns an awaitable that can be awaited directly. The task will begin executing immediately, so the returned awaitable does not need to be awaited or scheduled for the blocking call to start executing.

For example with the default a thread pool:

```
...
# get the event loop
loop = asyncio.get_running_loop()
# execute a function in a separate thread
await loop.run_in_executor(None, task)
```

For CPU-bound tasks we must create and supply a process pool via an instance of the `ProcessPoolExecutor` class.

The caller must manage the executor in this case, shutting it down once the caller is finished with it. This can be achieved by calling the `shutdown()` method or by using the context manager interface on the `ProcessPoolExecutor` that calls the shutdown method automatically on exit.

By default, the process pool will create one process worker for each logical CPU core in the current system.

For example:

```
...
# create a process pool
with ProcessPoolExecutor() as exe:
    # get the event loop
    loop = asyncio.get_running_loop()
    # execute a function in a separate thread
    await loop.run_in_executor(exe, task)
    # process pool is shutdown automatically...
```

Now that we know how to use the `asyncio.to_thread()` function, let's look at some worked examples.

22.3 Example Of Blocking The Asyncio Event Loop

Before we explore an example of running a blocking function in a new thread, let's look at an example of running the blocking function directly in the asyncio event loop.

In this example we will define a blocking function call that reports a message, sleeps, then reports a final message. Importantly, the `time.sleep()` function is used to block the current thread, including any currently running coroutines.

We also define a background coroutine that runs in a loop forever. Each iteration it reports a message and sleeps for half a second.

The main coroutine starts the background task independently. It then calls the blocking function.

The complete example is listed below.

```
# SuperFastPython.com
# example of running a blocking function call in asyncio
import time
import asyncio

# blocking function
def blocking_task():
```

```
    # report a message
    print('task is running')
    # block
    time.sleep(2)
    # report a message
    print('task is done')

# background coroutine task
async def background():
    # loop forever
    while True:
        # report a message
        print('>background task running')
        # sleep for a moment
        await asyncio.sleep(0.5)

# main coroutine
async def main():
    # run the background task
    _= asyncio.create_task(background())
    # execute the blocking call
    blocking_task()

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine runs. It creates the background coroutine and schedules it for execution as soon as it can.

The `main()` coroutine then calls the `blocking_task()` function.

This does not suspend the `main()` coroutine, instead, the call to the `blocking_task()` function is performed in the event loop and the call to `sleep()` blocks the current thread.

The background task coroutine cannot run.

Once the blocking call is finished, the background task is given an opportunity to run briefly before the `main()` coroutine is terminated.

This highlights that a blocking function call in a coroutine or task will not suspend and will block the entire event loop.

```
task is running
task is done
>background task running
```

Next, let's look at how we can update the example to run the blocking call in a new thread.

22.4 Example Of Running A Function In A Thread

We can explore how to execute the blocking call in a new thread and not stop the event loop.

In the example below we call `asyncio.to_thread()` to create a coroutine for the call to the `blocking_task()` function.

This coroutine is then awaited allowing the main coroutine to suspend and for the blocking function to execute in a new thread.

The complete example is listed below.

```
# SuperFastPython.com
# example of a blocking function call in a thread
import time
import asyncio

# blocking function
def blocking_task():
    # report a message
    print('task is running')
    # block
    time.sleep(2)
    # report a message
    print('task is done')

# background coroutine task
async def background():
    # loop forever
    while True:
        # report a message
        print('>background task running')
        # sleep for a moment
        await asyncio.sleep(0.5)

# main coroutine
async def main():
    # run the background task
    _= asyncio.create_task(background())
    # create a coroutine for the blocking function call
    coro = asyncio.to_thread(blocking_task)
    # execute the call in a new thread and await
    await coro
```

```
# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine runs. It creates the background coroutine and schedules it for execution as soon as it can.

The `main()` coroutine then creates a coroutine to run the background task in a new thread and then awaits this coroutine.

This does a couple of things.

Firstly, it suspends the `main()` coroutine, allowing any other coroutines in the event loop to run, such as the new coroutine for executing the blocking function in a new thread.

The new coroutine runs and starts a new thread and executes the blocking function in the new thread. This coroutine was also suspended.

The event loop is free and the background coroutine gets an opportunity to run, looping and reporting its messages.

The blocking call in the other thread finishes, suspends the background task, resumes the main thread, and terminates the program.

This highlights that running a blocking call in a new thread does not block the event loop, allowing other coroutines to run while the blocking call is being executed, suspending some thread other than the main event loop thread.

```
task is running
>background task running
>background task running
>background task running
>background task running
task is done
```

22.4.1 What Is The Difference?

Let's pause for a moment and consider the difference between this example and the previous example.

In both cases, the `main()` coroutine waits for the blocking call to finish.

- In the first case, the blocking call suspended the entire thread.
- In the second case, the blocking call is executed in a new thread and only the coroutine was suspended.

The first case prevents any other coroutines from running while the blocking call is blocked and waiting. The second case allows other coroutines to run while the blocking call is blocked

and waiting, as seen in the program output.

We now have a good idea of what the `to_thread()` function is doing.

Next, let's look at executing a CPU-bound blocking task using a new child process.

22.5 Example Of Running A Function In A Process

We can explore how to execute a CPU-bound task in an asyncio program.

In this example, we will use the `run_in_executor()` method on the currently running event loop.

We will create an instance of a `ProcessPoolExecutor`, configured with 4 worker child processes, appropriate for executing CPU-bound tasks. This process pool will then be passed to the `run_in_executor()` along with the target function.

Our target function will be a CPU-intensive task, in this case creating a list of 50 million squared integer values.

The task will begin executing immediately in the process pool, asynchronously. This frees the caller to continue with other activities, and the event loop to continue progressing asyncio tasks.

The complete example is listed below.

```
# SuperFastPython.com
# example of a blocking cpu-bound task in a process
from concurrent.futures import ProcessPoolExecutor
import asyncio
import math

# a blocking cpu-bound task
def blocking_task():
    # report a message
    print('Task starting', flush=True)
    # block for a while
    data = [math.sqrt(i) for i in range(50000000)]
    # report a message
    print('Task done', flush=True)

# main coroutine
async def main():
    # report a message
    print('Main running the blocking task')
    # get the event loop
    loop = asyncio.get_running_loop()
    # create the executor
```

```
    exe = ProcessPoolExecutor(4)
    # schedule the function to run
    awaitable = loop.run_in_executor(exe, blocking_task)
    # report a message
    print('Main doing other things')
    # sleep a moment
    await asyncio.sleep(1)
    # await the cpu-bound task
    await awaitable
    # close the process pool
    exe.shutdown()

# protect the entry point
if __name__ == '__main__':
    # start the asyncio event loop
    asyncio.run(main())
```

Running the example first creates the `main()` coroutine and runs it as the entry point into the asyncio program.

The `main()` coroutine runs and reports a message. It then gets access to the currently running event loop and creates a new `ProcessPoolExecutor` with 4 worker processes.

The `main()` coroutine then issues the blocking function call asynchronously using the process pool. This returns an awaitable and begins executing the task immediately using the provided executor.

The `main()` coroutine is free to continue with other activities. In this case, it reports a message and sleeps for a moment, then awaits the asynchronous task directly.

The blocking task runs in a new child process, reporting a message, sleeping for two seconds, and reporting a final message.

Once the task is completed, the `main()` coroutines then closes the process pool explicitly.

This highlights how a blocking CPU-bound task can be executed asynchronously in a separate thread using the low-level API.

```
Main running the blocking task
Main doing other things
Task starting
Task done
```

22.6 Takeaways

You now know how to execute blocking functions in new threads separate from the asyncio event loop.

Specifically, you know:

- The problem with executing blocking calls in asyncio programs.
- How to execute blocking function calls with a thread pool.
- How to choose to run a blocking task in a separate thread or process.

22.6.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- `time` – Time access and conversions.
  https://docs.python.org/3/library/time.html
- `threading` – Thread-based parallelism.
  https://docs.python.org/3/library/threading.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

22.6.2 Next

In the next tutorial, we will explore how to use asynchronous iterators in asyncio programs.

Part V

Structures

Chapter 23

Asynchronous Iterators

Iterators provide a way to traverse structures like lists of items in a linear way.

The problem is that conventional iterators are not well suited to asyncio programs. The reason is that we cannot have each item in the iterator retrieved asynchronously.

Instead, we can use asynchronous iterators along with the `async for` expression to automatically await the retrieval of the next item in the iteration.

In this tutorial, you will discover how to develop and use asynchronous iterators.

After completing this tutorial, you will know:

- The difference between regular iterators and asynchronous iterators.
- How to develop asynchronous iterators for use in asyncio.
- How to use asynchronous iterators with the `async for` expression in asyncio.

Let's get started.

23.1 What Are Asynchronous Iterators

An asynchronous iterator can be awaited each time it is stepped.

Before we take a close look at asynchronous iterators, let's review classical Python iterators.

23.1.1 Iterators

An iterator is a Python object that implements a specific interface.

Specifically, the `__iter__()` method that returns an instance of the iterator and the `__next__()` method that steps the iterator one cycle and returns a value.

> **iterator**: An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return

> successive items in the stream. When no more data are available a StopIteration exception is raised instead.

– Python Glossary.

An iterator can be stepped using the `next()` built-in function or traversed using a for loop.

Many Python objects are iterable, most notable are data structures such as lists.

Next, let's consider asynchronous iterators.

23.1.2 Asynchronous Iterators

An asynchronous iterator is a Python object that implements a specific interface.

> **asynchronous iterator**: An object that implements the `__aiter__()` and `__anext__()` methods.

– Python Glossary.

An asynchronous iterator must implement the `__aiter__()` and `__anext__()` methods.

- The `__aiter__()` method must return an instance of the iterator.
- The `__anext__()` method must return an awaitable that steps the iterator.

An asynchronous iterator may only be stepped or traversed in an asyncio program, such as within a coroutine.

An asynchronous iterator can be stepped using the `anext()` built-in function that returns an awaitable that executes one step of the iterator, e.g. one call to the `__anext__()` method.

An asynchronous iterator can be traversed using the `async for` expression that will automatically call `anext()` each iteration and await the returned awaitable in order to retrieve the return value.

Now that we know what an asynchronous iterator is, let's compare it to a classical iterator.

23.1.3 Iterators Versus Asynchronous Iterators

Both classical iterators and asynchronous iterators have a lot in common.

They both traverse a linear series one step at a time.

A classical iterator can be used generally anywhere in a Python program, whereas an asynchronous iterator can only be used in an asyncio Python program, such as called and used within a coroutine.

The classical iterator can be stepped using the built-in `next()` function, whereas asynchronous iterators are traversed using the `anext()` built-in function.

The classical iterator is traversed with a classical `for` loop, whereas the asynchronous iterator must be traversed using an `async for` loop expression.

Each iteration of the asynchronous iterator returns an awaitable that must be awaited in an asyncio program. Therefore mixing the syntax of classical and asynchronous iterators will result in errors.

We can summarize these differences using a table.

Differences	Classical Iterator	Async Iterator
Usage	Python and Asyncio	Asyncio
Step	`next()`	`anext()`
Traversal	`for` loop	`async for` loop
Returns	Value	Awaitable

Next, let's consider the relationship between asynchronous iterators and asynchronous generators.

23.1.4 Asynchronous Iterators And Generators

Asynchronous iterators and asynchronous generators are tightly related.

An asynchronous generator is a coroutine that makes use of the `yield` expression.

An asynchronous generator can be created which returns an *asynchronous generator iterator* which is a type of asynchronous iterator.

> **asynchronous generator iterator**: An object created by a asynchronous generator function. This is an asynchronous iterator which when called using the `__anext__()` method returns an awaitable object which will execute the body of the asynchronous generator function until the next `yield` expression.

– Python Glossary.

The asynchronous generator iterator can be used directly as an asynchronous iterator.

Now that we know about asynchronous iterators, let's look at how we can define and use them.

23.2 How To Use Asynchronous Iterators

In this section, we will take a close look at how to define, create, step, and traverse an asynchronous iterator in asyncio programs.

Let's start with how to define an asynchronous iterator.

23.2.1 Define An Asynchronous Iterator

We can define an asynchronous iterator by defining a class that implements the `__aiter__()` and `__anext__()` methods.

These methods are defined on a Python object as per normal.

Importantly, because the `__anext__()` function must return an awaitable, it must be defined using the `async def` expression.

When the iteration is complete, the `__anext__()` method must raise a `StopAsyncIteration` exception.

For example:

```
# define an asynchronous iterator
class AsyncIterator():
    # constructor, define some state
    def __init__(self):
        self.counter = 0

    # create an instance of the iterator
    def __aiter__(self):
        return self

    # return the next awaitable
    async def __anext__(self):
        # check for no further items
        if self.counter >= 10:
            raise StopAsyncIteration
        # increment the counter
        self.counter += 1
        # return the counter value
        return self.counter
```

Because the asynchronous iterator is a coroutine and each iterator returns an awaitable that is scheduled and executed in the asyncio event loop, we can execute and await awaitables within the body of the iterator.

For example:

```
...
# return the next awaitable
async def __anext__(self):
    # check for no further items
    if self.counter >= 10:
        raise StopAsyncIteration
    # increment the counter
    self.counter += 1
    # simulate work
    await asyncio.sleep(1)
    # return the counter value
    return self.counter
```

Next, let's look at how we might use an asynchronous iterator.

23.2.2 Create Asynchronous Iterator

To use an asynchronous iterator we must create the iterator.

This involves creating the Python object as per normal.

For example:

```
...
# create the iterator
it = AsyncIterator()
```

This returns an asynchronous iterable, which is an instance of an asynchronous iterator.

23.2.3 Step An Asynchronous Iterator

One step of the iterator can be traversed using the `anext()` built-in function, just like a classical iterator using the `next()` function.

The result is an awaitable that can be awaited via the `await` expression.

For example:

```
...
# get an awaitable for one step of the iterator
awaitable = anext(it)
# execute one step of the iterator and get the result
result = await awaitable
```

This can be achieved in one step.

For example:

```
...
# step the async iterator
result = await anext(it)
```

23.2.4 Traverse An Asynchronous Iterator

The asynchronous iterator can also be traversed in a loop using the `async for` expression that will await each iteration of the loop automatically.

For example:

```
...
# traverse an asynchronous iterator
async for result in AsyncIterator():
    # report resilt
    print(result)
```

We may also use an asynchronous list comprehension with the `async for` expression to collect the results of the iterator.

For example:

```
...
# async list comprehension with async iterator
results = [item async for item in AsyncIterator()]
```

Now that we know how to create and use asynchronous iterators, let's look at some worked examples.

23.3 Example Of One Step Of An Async Iterator

We can explore how to create an asynchronous iterator and step it one time.

This is a good approach to understanding more deeply what exactly is happening with the iterator when it is traversed using an `async for` expression.

In this example, we will first define an asynchronous iterator that loops a number of times, suspends each iteration, and returns an integer value. The constructor of the iterator defines a counter and initializes it to zero, this counter is updated each time the iterator is progressed with a call to `__anext__()`.

Awaiting within the asynchronous iterator highlights the coroutine nature of the iterator in that it is able to suspend and await other coroutines and tasks.

The main coroutine will create the iterator, get an awaitable that progresses the iterator one step, then await the awaitable in order to retrieve the yielded value.

The complete example is listed below.

```
# SuperFastPython.com
# example of one step of an asynchronous iterator
import asyncio

# define an asynchronous iterator
class AsyncIterator():
    # constructor, define some state
    def __init__(self):
        self.counter = 0

    # create an instance of the iterator
    def __aiter__(self):
        return self

    # return the next awaitable
    async def __anext__(self):
        # check for no further items
```

```
        if self.counter >= 10:
            raise StopAsyncIteration
        # increment the counter
        self.counter += 1
        # simulate work
        await asyncio.sleep(1)
        # return the counter value
        return self.counter

# main coroutine
async def main():
    # create the async iterator
    it = AsyncIterator()
    # step the iterator one iteration
    awaitable = anext(it)
    # get the result from one iteration
    result = await awaitable
    # report the result
    print(result)

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine runs and creates an instance of the asynchronous iterator. This initializes the internal counter to zero.

It then calls the `anext()` function on the iterator. This returns an awaitable, that if awaited will execute the iterator's `__anext__()` method.

The awaitable is awaited. This suspends the `main()` coroutine and executes the asynchronous iterator that itself suspends a moment and then returns an integer value.

The `main()` coroutine resumes and retrieves the returned value and reports it.

The program then terminates.

This highlights what is happening when an asynchronous iterator is executed, such as each iteration within an `async for` expression.

```
1
```

Next, let's look at how we might traverse an asynchronous iterator using the `async for` expression.

23.4 Example Of An Async Iterator With For Loop

We can explore how to traverse an asynchronous iterator using the `async for` expression.

In this example, we will update the previous example to traverse the iterator to completion using an `async for` loop.

This loop will automatically await each awaitable returned from the iterator, retrieve the returned value, and make it available within the loop body so that in this case it can be reported.

This is perhaps the most common usage pattern for asynchronous iterators.

The complete example is listed below.

```
# SuperFastPython.com
# example of an asynchronous iterator with async for
import asyncio

# define an asynchronous iterator
class AsyncIterator():
    # constructor, define some state
    def __init__(self):
        self.counter = 0

    # create an instance of the iterator
    def __aiter__(self):
        return self

    # return the next awaitable
    async def __anext__(self):
        # check for no further items
        if self.counter >= 10:
            raise StopAsyncIteration
        # increment the counter
        self.counter += 1
        # simulate work
        await asyncio.sleep(1)
        # return the counter value
        return self.counter

# main coroutine
async def main():
    # loop over async iterator with async for loop
    async for item in AsyncIterator():
        print(item)
```

```
# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine runs and starts the for loop.

An instance of the asynchronous iterator is created and the loop automatically steps it using the `anext()` function to return an awaitable. The loop then awaits the awaitable and retrieves a value which is made available to the body of the loop where it is reported.

This process is then repeated, suspending the `main()` coroutine, executing a step of the iterator and suspending, and resuming the `main()` coroutine until the iterator is exhausted.

Once the internal counter of the iterator reaches 10, a `StopAsyncIteration` is raised. This does not terminate the program. Instead, it is expected and handled by the `async for` expression and breaks the loop.

This highlights how an asynchronous iterator can be traversed using an `async for` expression.

```
1
2
3
4
5
6
7
8
9
10
```

Next, let's look at how we might use an asynchronous iterator with an asynchronous list comprehension.

23.5 Example Of Async Iterator With Comprehension

We can explore how to use an asynchronous iterator with an asynchronous list comprehension.

In this example, we will update the above example to traverse the asynchronous iterator to completion.

In this case, we will traverse it within an asynchronous list comprehension, rather than an `async for` loop. This will collect the returned values into a list that can be reported.

The complete example is listed below.

```
# SuperFastPython.com
# example of an iterator with async comprehension
```

```
import asyncio

# define an asynchronous iterator
class AsyncIterator():
    # constructor, define some state
    def __init__(self):
        self.counter = 0

    # create an instance of the iterator
    def __aiter__(self):
        return self

    # return the next awaitable
    async def __anext__(self):
        # check for no further items
        if self.counter >= 10:
            raise StopAsyncIteration
        # increment the counter
        self.counter += 1
        # simulate work
        await asyncio.sleep(1)
        # return the counter value
        return self.counter

# main coroutine
async def main():
    # loop over async iterator with async comprehension
    results = [item async for item in AsyncIterator()]
    # report results
    print(results)

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine runs and executes the asynchronous list comprehension.

This comprehension operates just like the `async for` loop in the previous section. Each iteration retrieves an awaitable, the awaitable is awaited and suspends the `main()` coroutine, and an iteration of the iterator executes and returns a value which is then collected in the resumed `main()` coroutine.

The process is repeated until all values are collected into a list.

Once the asynchronous list comprehension is completed, the values are then reported.

This highlights how we might traverse an asynchronous iterator using an asynchronous list comprehension.

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

23.6 Takeaways

You now know how to create and use asynchronous iterators.

Specifically, you know:

- The difference between regular iterators and asynchronous iterators.
- How to develop asynchronous iterators for use in asyncio.
- How to use asynchronous iterators with the `async for` expression in asyncio.

23.6.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- Python Glossary.
  https://docs.python.org/3/glossary.
- PEP 492 – Coroutines with async and await syntax.
  https://peps.python.org/pep-0492
- Python Data model.
  https://docs.python.org/3/reference/datamodel.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

23.6.2 Next

In the next tutorial, we will explore how to use asynchronous generators in asyncio programs.

Chapter 24

Asynchronous Generators

A generator is a function that has at least one `yield` expression.

Using a generator returns a generator iterator that can be traversed to yield the generated values. The problem is that conventional generators are not well suited to asyncio programs as we cannot await yielded values.

Instead, we can develop and use asynchronous generators defined using coroutines that yield values. The resulting asynchronous generator iterator can then be traversed, awaiting each yielded value automatically with the `async for` expression.

In this tutorial, you will discover how to develop and use asynchronous generators.

After completing this tutorial, you will know:

- The difference between conventional generators and asynchronous generators.
- How to develop asynchronous generators for use in asyncio.
- How to use asynchronous generators with the `async for` expression in asyncio.

Let's get started.

24.1 What Are Asynchronous Generators

An asynchronous generator is a coroutine that uses the `yield` expression.

Before we dive into the details of asynchronous generators, let's first review classical Python generators.

24.1.1 Generators

A generator is a Python function that returns a value via a `yield` expression.

For example:

```
# define a generator
def generator():
    for i in range(10):
        yield i
```

The generator is executed to the `yield` expression, after which a value is returned. This suspends the generator at that point. The next time the generator is executed it is resumed from the point it was resumed and runs until the next `yield` expression.

> **generator**: A function which returns a generator iterator. It looks like a normal function except that it contains `yield` expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function.

– Python Glossary.

Technically, a generator function creates and returns a generator iterator. The generator iterator executes the content of the generator function, yielding and resuming as needed.

> **generator iterator**: An object created by a generator function. Each yield temporarily suspends processing, remembering the location execution state [...] When the generator iterator resumes, it picks up where it left off ...

– Python Glossary.

A generator can be executed in steps by using the `next()` built-in function.

For example:

```
...
# create the generator
gen = generator()
# step the generator
result = next(gen)
```

Although, it is more common to iterate the generator to completion, such as using a for-loop or a list comprehension.

For example:

```
...
# traverse the generator and collect results
results = [item for item in generator()]
```

Next, let's take a closer look at asynchronous generators.

24.1.2 Asynchronous Generators

An asynchronous generator is a coroutine that uses the `yield` expression.

Unlike a function generator, the coroutine can schedule and await other coroutines and tasks.

> **asynchronous generator**: A function which returns an asynchronous generator iterator. It looks like a coroutine function defined with async def except that it contains `yield` expressions for producing a series of values usable in an `async for` loop.

– Python Glossary.

Like a classical generator, an asynchronous generator function can be used to create an asynchronous generator iterator that can be traversed using the built-in `anext()` function, instead of the `next()` function.

> **asynchronous generator iterator**: An object created by a asynchronous generator function. This is an asynchronous iterator which when called using the `__anext__()` method returns an awaitable object which will execute the body of the asynchronous generator function until the next `yield` expression.

– Python Glossary.

This means that the asynchronous generator iterator implements the `__anext__()` method and can be used with the `async for` expression.

This means that each iteration of the generator is scheduled and executed as awaitable. The `async for` expression will schedule and execute each iteration of the generator, suspending the calling coroutine and awaiting the result.

Next, let's consider the similarities and differences between classical and asynchronous generators in more detail.

24.1.3 Generators Versus Asynchronous Generators

Both classical generators and asynchronous generators have a lot in common.

Both are types of functions that have at least one `yield` expression.

A classical generator can be used generally anywhere in a Python program, whereas an asynchronous generator can only be used in an asyncio Python program, such as called and used within a coroutine.

The classical generator can be stepped using the built-in `next()` function, whereas asynchronous generators are traversed using the `anext()` built-in function.

The classical generator is traversed with a classical for loop, whereas the asynchronous generator must be traversed using an `async for` loop expression.

Each iteration of the asynchronous generator returns an awaitable that must be awaited in an asyncio program. Therefore mixing the syntax of classical and asynchronous generators will result in errors.

We can summarize these differences using a table.

Differences	Classical Generator	Async generator
Type	Function	Coroutine
Usage	Python and Asyncio	Asyncio
Step	`next()`	`anext()`
Traversal	`for` loop	`async for` loop
Yields	Value	Awaitable

Now that we know about asynchronous generators, let's look at how we can define and use them.

24.2 How To Use An Asynchronous Generator

In this section, we will take a close look at how to define, create, step, and traverse an asynchronous generator in asyncio programs.

Let's start with how to define an asynchronous generator.

24.2.1 Define An Asynchronous Generator

We can define an asynchronous generator by defining a coroutine that has at least one `yield` expression.

This means that the function is defined using the `async def` expression.

For example:

```
# define an asynchronous generator
async def async_generator():
    for i in range(10)
        yield i
```

Because the asynchronous generator is a coroutine and each iterator returns an awaitable that is scheduled and executed in the asyncio event loop, we can execute and await awaitables within the body of the generator.

For example:

```
# define an asynchronous generator that awaits
async def async_generator():
    for i in range(10)
        # suspend and sleep a moment
        await asyncio.sleep(1)
        # yield a value to the caller
        yield i
```

Next, let's look at how we might use an asynchronous generator.

24.2.2 Create Asynchronous Generator

To use an asynchronous generator we must create the generator.

This looks like calling it, but instead creates and returns an iterator object.

For example:

```
...
# create the iterator
it = async_generator()
```

This returns a type of asynchronous iterator called an asynchronous generator iterator.

24.2.3 Step An Asynchronous Generator

One step of the generator can be traversed using the `anext()` built-in function, just like a classical generator using the `next()` function.

The result is an awaitable that is awaited.

For example:

```
...
# get an awaitable for one step of the generator
awaitable = anext(gen)
# execute one step of the generator and get the result
result = await awaitable
```

This can be achieved in one step.

For example:

```
...
# step the async generator
result = await anext(gen)
```

24.2.4 Traverse An Asynchronous Generator

The asynchronous generator can also be traversed in a loop using the `async for` expression that will await each iteration of the loop automatically.

For example:

```
...
# traverse an asynchronous generator
async for result in async_generator():
    print(result)
```

We may also use an asynchronous list comprehension with the `async for` expression to collect the results of the generator.

For example:

```
...
# async list comprehension with async generator
results = [item async for item in async_generator()]
```

Now that we know how to create and use asynchronous generators, let's look at a worked example.

24.3 Example Of One Step Of An Async Generator

We can explore how to create an asynchronous generator and step it one time.

This is a good approach to understanding more deeply what exactly is happening with the generator when it is traversed using an `async for` expression.

In this example we will first define an asynchronous generator that iterates a number of times, suspends each iteration, and yields an integer value.

This highlights the coroutine nature of the asynchronous generator in that it is able to suspend and await other coroutines and tasks.

The main coroutine will create the generator, get an awaitable that steps the generator one iteration, then await the awaitable in order to retrieve the yielded value.

The complete example is listed below.

```
# SuperFastPython.com
# example of one step of an asynchronous generator
import asyncio

# define an asynchronous generator
async def async_generator():
    # normal loop
    for i in range(10):
        # block to simulate doing work
        await asyncio.sleep(1)
        # yield the result
        yield i

# main coroutine
async def main():
    # create the async generator
    gen = async_generator()
    # step the generator one iteration
    awaitable = anext(gen)
    # get the result from one iteration
    result = await awaitable
    # report the result
```

```
    print(result)

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine runs and creates an instance of the asynchronous generator.

It then calls the `anext()` function on the generator. This returns an awaitable, that if awaited will execute one iteration of the generator to the first `yield` statement.

The awaitable is awaited. This suspends the `main()` coroutine and executes the asynchronous generator one iteration to the first `yield` statement and a value is returned. The generator is suspended.

The `main()` coroutine resumes and retrieves the yielded value and reports it.

The program then terminates.

This highlights what is happening when an asynchronous generator is executed, such as each iteration within an `async for` expression.

```
0
```

Next, let's look at how we might traverse an asynchronous generator using the `async for` expression.

24.4 Example Of An Async Generator With For Loop

We can explore how to traverse an asynchronous generator using the `async for` expression.

In this example, we will update the previous example to traverse the generator to completion using an `async for` loop.

This loop will automatically await each awaitable returned from the generator, retrieve the yielded value, and make it available within the loop body so that in this case it can be reported.

This is perhaps the most common usage pattern for asynchronous generators.

The complete example is listed below.

```
# SuperFastPython.com
# example of asynchronous generator with async for loop
import asyncio

# define an asynchronous generator
async def async_generator():
```

```
    # normal loop
    for i in range(10):
        # block to simulate doing work
        await asyncio.sleep(1)
        # yield the result
        yield i

# main coroutine
async def main():
    # loop over async generator with async for loop
    async for item in async_generator():
        print(item)

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine runs and starts the for loop.

An instance of the asynchronous generator is created and the loop automatically steps it using the `anext()` function to return an awaitable. The loop then awaits the awaitable and retrieves a value which is made available to the body of the loop where it is reported.

This process is then repeated, suspending the `main()` coroutine, executing an iteration of the generator, and suspending, and resuming the `main()` coroutine until the generator is exhausted.

This highlights how an asynchronous generator can be traversed using an `async for` expression.

```
0
1
2
3
4
5
6
7
8
9
```

Next, let's look at how we might use an asynchronous generator with an asynchronous list comprehension.

24.5 Example Of Async Generator With Comprehension

We can explore how to use an asynchronous generator with an asynchronous list comprehension.

In this example, we will update the above example to traverse the asynchronous generator to completion.

In this case, we will traverse it within an asynchronous list comprehension, rather than a `async for` loop. This will collect the yielded values into a list that can be reported.

The complete example is listed below.

```
# SuperFastPython.com
# example of a generator with async comprehension
import asyncio

# define an asynchronous generator
async def async_generator():
    # normal loop
    for i in range(10):
        # block to simulate doing work
        await asyncio.sleep(1)
        # yield the result
        yield i

# main coroutine
async def main():
    # loop over async generator with async comprehension
    results = [item async for item in async_generator()]
    # report results
    print(results)

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine runs and executes the asynchronous list comprehension.

This comprehension operates just like the `async for` loop in the previous section. Each iteration retrieves an awaitable, the awaitable is awaited and suspends the `main()` coroutine, and an iteration of the generator executes and yields a value which is then collected in the resumed `main()` coroutine.

The process is repeated until all values are collected into a list.

Once the asynchronous list comprehension is completed, the values are then reported.

This highlights how we might traverse an asynchronous generator using an asynchronous list comprehension.

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

24.6 Takeaways

You now know how to create and use asynchronous generators.

Specifically, you know:

- The difference between conventional generators and asynchronous generators.
- How to develop asynchronous generators for use in asyncio.
- How to use asynchronous generators with the `async for` expression in asyncio.

24.6.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- Python Glossary.
  https://docs.python.org/3/glossary.html
- PEP 525 – Asynchronous Generators.
  https://peps.python.org/pep-0525/
- Python Built-in Functions.
  https://docs.python.org/3/library/functions.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

24.6.2 Next

In the next tutorial, we will explore how to use asynchronous context managers in asyncio programs.

Chapter 25

Asynchronous Context Managers

Context managers are a helpful construct for automatically executing common code before and after a block.

The problem is that we cannot suspend and await coroutines at the entry and exit of a regular context manager.

The solution is to use an asynchronous context manager where the entry and exit code can be defined using coroutines and awaited automatically via the `async with` expression.

In this tutorial, you will discover how to develop and use asynchronous context managers.

After completing this tutorial, you will know:

- The difference between conventional and asynchronous context managers.
- How to develop asynchronous context managers for use in asyncio.
- How to use asynchronous context managers with the `async with` expression in asyncio.

Let's get started.

25.1 What Is An Asynchronous Context Manager

An asynchronous context manager is like a regular context manager, except it can be awaited on entry and exit.

We have seen two asynchronous context managers in this book so far, specifically the `asyncio.TaskGroup` and the `asyncio.timeout()`.

Before we dive into the details of asynchronous context managers, let's review classical context managers.

25.1.1 Context Manager

A context manager is a Python object that implements the `__enter__()` and `__exit__()` methods.

The `__enter__()` method defines what happens at the beginning of a block, such as opening or preparing resources, like a file, socket or thread pool.

The `__exit__()` method defines what happens when the block is exited, such as closing a prepared resource.

A context manager is used via the `with` expression.

Typically the context manager object is created in the beginning of the `with` expression and the `__enter__()` method is called automatically. The body of the content makes use of the resource via the named context manager object, then the `__aexit__()` method is called automatically when the block is exited, normally or via an exception.

For example:

```
...
# open a context manager
with ContextManager() as manager:
    # ...
# closed automatically
```

This simplifies a try-finally construct that might call these same methods.

For example:

```
...
# create the object
manager = ContextManager()
try:
    manager.__enter__()
    # ...
finally:
    manager.__exit__()
```

Next, let's take a look at asynchronous context managers.

25.1.2 Asynchronous Context Manager

Asynchronous context managers provide a context manager that can be suspended when entering and exiting.

It must implement two methods that are defined as coroutines instead of Python functions, they are the `__aenter__()` and `__aexit__()` methods.

This is achieved using the `async with` expression.

As such, asynchronous context managers can only be used within asyncio programs, such as within calling coroutines.

Next, let's take a closer look at the difference between classical and asynchronous context managers.

25.1.3 Context Manager Versus Asynchronous Context Manager

Both classical context managers and asynchronous context managers have a lot in common.

They both attempt to achieve the same effect of a try-finally expression with minimal code and a well-defined interface.

Classical context managers use the **enter** and **exit** methods, whereas asynchronous context managers use the `__aenter__` and `__aexit__` methods.

Classical context managers may be used generally anywhere in Python programs, whereas asynchronous context managers may only be used within asyncio programs, such as within coroutines.

Classical context managers are used via the `with` expression, whereas asynchronous context managers are used via the `async with` expression.

We can summarize these differences using a table.

Differences	Context Mgmr	Async Ctx Mgmr
Implementation	Functions	Coroutines
Usage	Python and Asyncio	Asyncio
Expression	`with`	`async with`

Now that we know about asynchronous context managers, let's look at how we might use them.

25.2 How To Use Asynchronous Context Managers

In this section, we will explore how we can define, create, and use asynchronous context managers in our asyncio programs.

25.2.1 Define An Asynchronous Context Manager

We can define an asynchronous context manager as a Python object that implements the `__aenter__()` and `__aexit__()` methods.

Importantly, both methods must be defined as coroutines using the `async def` expression and therefore must return awaitables.

For example:

```
# define an asynchronous context manager
class AsyncContextManager:
    # enter the async context manager
    async def __aenter__(self):
        # report a message
```

```
        print('>entering the context manager')

    # exit the async context manager
    async def __aexit__(self, exc_type, exc, tb):
        # report a message
        print('>exiting the context manager')
```

Because each of the methods are coroutines, they may themselves await coroutines or tasks.

For example:

```
# define an asynchronous context manager
class AsyncContextManager:
    # enter the async context manager
    async def __aenter__(self):
        # report a message
        print('>entering the context manager')
        # block for a moment
        await asyncio.sleep(0.5)

    # exit the async context manager
    async def __aexit__(self, exc_type, exc, tb):
        # report a message
        print('>exiting the context manager')
        # block for a moment
        await asyncio.sleep(0.5)
```

25.2.2 Use An Asynchronous Context Manager

An asynchronous context manager is used via the `async with` expression.

This will automatically await the enter and exit coroutines, suspending the calling coroutine as needed.

For example:

```
...
# use an asynchronous context manager
async with AsyncContextManager() as manager:
    # ...
```

As such, the `async with` expression and asynchronous context managers more generally can only be used within asyncio programs, such as within coroutines.

Now that we know how to use asynchronous context managers, let's look at some worked examples.

25.3 Example Of Manual Async Context Manager

We can explore how to use an asynchronous context manager manually.

In this example, we will define an asynchronous context manager. The entry and exit methods of the manager will report a message and sleep for a moment. The messages will help us see when the methods are executed relative to our program code and then sleeps to simulate blocking I/O and show that these coroutines can be suspended when calling other coroutines.

The asynchronous context manager will be used manually.

That is, we will explicitly call the enter and exit methods, retrieve the awaitables from the coroutines, and await them directly.

This is a good example to show what exactly is happening automatically when we use more common usage patterns, such as the `async with` expression.

The complete example is listed below.

```
# SuperFastPython.com
# example of using an async context manager manually
import asyncio

# define an asynchronous context manager
class AsyncContextManager:
    # enter the async context manager
    async def __aenter__(self):
        # report a message
        print('>entering the context manager')
        # block for a moment
        await asyncio.sleep(0.5)

    # exit the async context manager
    async def __aexit__(self, exc_type, exc, tb):
        # report a message
        print('>exiting the context manager')
        # block for a moment
        await asyncio.sleep(0.5)

# define a simple coroutine
async def custom_coroutine():
    # create and use the asynchronous context manager
    manager = AsyncContextManager()
    # get the awaitable for entering the manager
    enter_awaitable = manager.__aenter__()
    # await the entry
    await enter_awaitable
    # execute the body
```

```
    print(f'within the manager')
    # get the awaitable for exiting the manager
    exit_awaitable = manager.__aexit__(None, None, None)
    # await the exit
    await exit_awaitable

# start the asyncio event loop
asyncio.run(custom_coroutine())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine runs and creates an instance of our `AsyncContextManager` class.

Next, we call the `__aenter__()` coroutine to get an awaitable coroutine object.

The awaitable is then awaited, suspending the `main()` coroutine and executing the enter coroutine which reports a message and sleeps for a moment.

The `main()` coroutine resumes and reports a message.

It then calls the `__aexit__()` coroutine to get an awaitable coroutine object.

This second awaitable is awaited, suspending the `main()` coroutine, executing the coroutine that then reports a message and sleeps a moment.

The `main()` coroutine resumes and terminates the program.

This unrealistic example highlights what is happening automatically behind the scenes when we use a context manager with the `async with` expression, such as in the next section.

```
>entering the context manager
within the manager
>exiting the context manager
```

Next, we will look at the normal usage of asynchronous context manager.

25.4 Example Of An Asynchronous Context Manager

We can explore how to use an asynchronous context manager via the `async with` expression.

In this example, we will update the above example to use the context manager in a normal manner.

We will use an `async with` expression and on one line, create and enter the context manager. This will automatically await the enter method.

We can then make use of the manager within the inner block. In this case, we will just report a message.

Exiting the inner block will automatically await the exit method of the context manager.

Contrasting this example with the previous example shows how much heavy lifting the `async with` expression does for us in an asyncio program.

The complete example is listed below.

```
# SuperFastPython.com
# example of an async context manager via async with
import asyncio

# define an asynchronous context manager
class AsyncContextManager:
    # enter the async context manager
    async def __aenter__(self):
        # report a message
        print('>entering the context manager')
        # block for a moment
        await asyncio.sleep(0.5)

    # exit the async context manager
    async def __aexit__(self, exc_type, exc, tb):
        # report a message
        print('>exiting the context manager')
        # block for a moment
        await asyncio.sleep(0.5)

# define a simple coroutine
async def custom_coroutine():
    # create and use the asynchronous context manager
    async with AsyncContextManager() as manager:
        # report the result
        print(f'within the manager')

# start the asyncio event loop
asyncio.run(custom_coroutine())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine runs and creates an instance of our `AsyncContextManager` class in an `async with` expression.

This expression automatically calls the enter method and awaits the coroutine. A message is reported and the coroutine blocks for a moment.

The `main()` coroutine resumes and executes the body of the context manager, printing a message.

The block is exited and the exit method of the context manager is awaited automatically,

reporting a message and sleeping a moment.

This highlights the normal usage pattern for an asynchronous context manager in an asyncio program.

```
>entering the context manager
within the manager
>exiting the context manager
```

Next, we can look at what happens to an asynchronous context manager when an exception is raised.

25.5 Example Of Exception In Async Context Manager

We can explore what happens to an asynchronous context manager when an exception is raised in the inner block.

In this example, we will update the above example that uses the asynchronous context manager via the `async with` expression, and raise an exception within the inner block.

The goal of this example is to show that regardless of how the inner block of the context manager is exited, the `async with` expression will ensure that the exit coroutine is awaited and executed.

The complete example is listed below.

```
# SuperFastPython.com
# example of an async context manager with an exception
import asyncio

# define an asynchronous context manager
class AsyncContextManager:
    # enter the async context manager
    async def __aenter__(self):
        # report a message
        print('>entering the context manager')
        # block for a moment
        await asyncio.sleep(0.5)

    # exit the async context manager
    async def __aexit__(self, exc_type, exc, tb):
        # report a message
        print('>exiting the context manager')
        # block for a moment
        await asyncio.sleep(0.5)

# define a simple coroutine
```

```
async def custom_coroutine():
    # create and use the asynchronous context manager
    async with AsyncContextManager() as manager:
        # report the result
        print(f'within the manager')
        # fail with an exception
        raise Exception('Something bad happened')

# start the asyncio event loop
asyncio.run(custom_coroutine())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine runs and creates an instance of our `AsyncContextManager` class in an `async with` expression.

This expression automatically calls the enter method and awaits the coroutine. A message is reported and the coroutine blocks for a moment.

The `main()` coroutine resumes and executes the body of the context manager, printing a message, and raising an exception.

The raised exception causes the context manager block to be exited. The exit method of the context manager is awaited automatically, reporting a message and sleeping a moment.

The exception then terminates the event loop of the asyncio program.

This highlights that the exit coroutine of an asynchronous context manager is awaited, regardless of how the inner block of the context manager is exited, such as with an unhandled exception.

```
>entering the context manager
within the manager
>exiting the context manager
Traceback (most recent call last):
  ...
Exception: Something bad happened
```

25.5.1 Takeaways

You now know how to develop and use asynchronous context managers.

Specifically, you know:

- The difference between conventional and asynchronous context managers.
- How to develop asynchronous context managers for use in asyncio.
- How to use asynchronous context managers with the `async with` expression in asyncio.

25.5.2 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- Python Data Model.
  https://docs.python.org/3/reference/datamodel.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

25.5.3 Next

In the next tutorial, we will explore how to use asynchronous queues in asyncio programs.

Chapter 26

Asynchronous Queues

A queue is a helpful data structure where items can be added and removed.

Queues are generally used in concurrent programs to connect tasks, such as between threads and between processes.

Asyncio also provides a queue, specifically tailored to share data between coroutines and tasks where blocking operations such as placing items on the queue and removing them can be awaited, spending the caller.

In this tutorial, you will discover how to share data between coroutines using queues.

After completing this tutorial, you will know:

- How to create and use a queue to connect coroutines and tasks in asyncio.
- How to use a queue without blocking and with timeouts.
- How to join a queue and have tasks done, and limit the overall capacity of the queue.

Let's get started.

26.1 What Is An Asyncio Queue

A asynchronous queue provides a FIFO queue for use with coroutines.

Before we dive into the details of the asynchronous queue, let's take a quick look at queues more generally.

26.1.1 Queue

A queue is a data structure on which items can be added by a call to `put()` and from which items can be retrieved by a call to `get()`.

Python provides a thread-safe queue via the `queue.Queue` class that allows threads to share objects with each other.

A process-safe queue is provided via the `multiprocessing.Queue` class. This queue is both thread-safe and process-safe and allows processes to share data Python objects.

Importantly, queues can block when putting an item, in the case of a queue with a fixed capacity. A caller can block when retrieving an item from the queue, if the queue is empty at the time the get request was made.

The level of concurrency determines at what level the blocking call is made, e.g. blocking the thread or the process.

Next, let's take a look at the asyncio queue.

26.1.2 Asyncio Queue

The `asyncio.Queue` class provides a queue that allows coroutines to share data.

This queue is not thread-safe nor process-safe.

This means that it cannot be used by threads or processes to share Python objects and is only intended for use by coroutines within one Python thread, e.g. one event loop.

Like the `queue.Queue` and `multiprocessing.Queue` classes, the `asyncio.Queue` is a FIFO queue. This means that data is managed and retrieved in a first-in, first-out manner.

The `asyncio` module also provides additional queue classes such as the `asyncio.LifoQueue` and `asyncio.PriorityQueue` that offer the same capabilities as the `asyncio.Queue`, except they maintain the queued items with a different order.

The `asyncio.Queue` is intended for use within asyncio programs by coroutines. As such, some methods on the class are in fact coroutine functions and must be awaited.

This means that the `asyncio.Queue` cannot be used outside of an asyncio program.

Now that we know about the `asyncio.Queue`, let's look at how we might use it.

26.2 How To Use An Asyncio Queue

In this section, we will explore how to use the `asyncio.Queue` class, including how to create and configure an instance, how to add and remove items, query the properties of the queue and manage tasks.

26.2.1 Create A Queue

We can create an `asyncio.Queue` by creating an instance of the class.

By default, the queue will not be limited in capacity.

For example:

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

The `asyncio.Queue` takes one constructor argument which is `maxsize`, set to zero (no limit) by default.

For example:

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

We can set a size limit on the queue.

The effect of a size limit means that 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.

For example:

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

Because the `maxsize` argument is the first position argument, we don't need to specify it by name.

For example:

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

26.2.2 Add Items To The Queue

Python objects can be added to a queue via the `put()` method.

This is in fact a coroutine function that must be awaited. The reason is that the calling coroutine may block if the queue is full.

For example:

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

An item can also be added to the queue without blocking via the `put_nowait()` method.

This method is not a coroutine and will either add the item and return immediately or fail with an `asyncio.QueueFull` exception if the queue is full and the item cannot be added.

For example:

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

26.2.3 Get Items From The Queue

Items can be retrieved from the queue by calling the `get()` method.

This is in fact a coroutine that must be awaited. The reason is that the queue may not have any items to retrieve at the time, and the calling coroutine may need to block until an item becomes available.

For example:

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

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

An item can be retrieved from the queue without blocking via the `get_nowait()` method.

This method is not a coroutine and will return an item immediately if available, otherwise will fail with an `asyncio.QueueEmpty` exception.

For example:

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

26.2.4 Query Queue Size

We can retrieve the fixed size of the queue via the `maxsize` property.

For example:

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

We can check if the queue is empty via the `empty()` method which returns `True` if the queue contains no items or `False` otherwise.

For example:

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

We can also check if the queue is full via the `full()` method that returns `True` if the queue is at capacity or `False` otherwise.

For example:

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

26.2.5 Queue Join And Task Done

Items on the queue can be treated as tasks that can be marked as processes by consumer coroutines.

This can be achieved by consumer coroutines retrieving items from the queue via `get()` or `get_nowait()` and once processed marking them via the `task_done()` method.

For example:

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

Other coroutines may be interested 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()` coroutine on the queue.

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

For example:

```
...
# 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 the `join()` coroutine will return immediately.

Now that we know how to use the `asyncio.Queue`, let's look at some worked examples.

26.3 Example Of Asyncio Queue

We can explore how to use the `asyncio.Queue` class with a worked example.

In this example, 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.

First, we can define the coroutine to be executed by the producer.

The task will iterate ten times in a loop. Each iteration will generate a new random value between 0 and 1 via the `random.random()` function. It will then sleep for that fraction of a second to simulate work, then put the value on the queue.

Once the task is complete it will put the value `None` on the queue to signal to the consumer coroutine that there is no further work.

The `producer()` coroutine below implements this by taking the queue instance as an argument.

```
# 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('Producer: Done')
```

Next, we can define the coroutine to be executed by the consumer.

The task will loop forever. Each iteration, it will get an item from the queue and block if there is no item yet available.

If the item retrieved from the queue is the value `None`, then the task will break the loop and terminate the coroutine. Otherwise, the value is reported.

The `consumer()` coroutine below implements this and takes the queue instance as an argument.

```
# coroutine to consume work
async def consumer(queue):
    print('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'>got {item}')
    # all done
    print('Consumer: Done')
```

Finally, in the main coroutine, we can create the shared queue instance.

We can then create and run the producer and consumer coroutines and wait for them both to complete via the `asyncio.gather()` method.

```
# 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))
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of using an asyncio queue
from random import random
import asyncio

# 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('Producer: Done')

# coroutine to consume work
async def consumer(queue):
    print('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'>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 event loop
asyncio.run(main())
```

Running the example first creates the shared `asyncio.Queue` instance.

Next, the producer coroutine is created and passed the queue instance. Then the consumer coroutine is started and the main coroutine blocks until both coroutines terminate.

The producer coroutine generates a new random value for each iteration of the task, blocks, and adds it to the queue. The consumer coroutine waits on the queue for items to arrive, then consumes them one at a time, reporting their value.

Finally, the producer task finishes, a `None` value is put on the queue and the coroutine terminates. The consumer coroutine gets the `None` value, breaks its loop, and also terminates.

This highlights how the `asyncio.Queue` can be used to share data easily between producer and consumer coroutines.

Note that the program output will differ each time it is run given the use of random numbers.

```
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
```

Next, let's look at how we might get values from the queue without blocking.

26.4 Example Of Queue Without Blocking

We can get values from the `asyncio.Queue` without blocking.

This might be useful if we wish to use busy waiting in the consumer coroutine to check other state or perform other tasks while waiting for data to arrive on the queue.

We can update the example from the previous section to get items from the 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.

In this case, if there is no value to get from the queue, we report a message and sleep for a fraction of a second. We will then continue which will jump back to the start of the consumer busy waiting loop.

```
# 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('Consumer: got nothing, waiting...')
            await asyncio.sleep(0.5)
            continue
        # check for stop
        if item is None:
            break
        # report
        print(f'>got {item}')
    # all done
    print('Consumer: Done')
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of using an asyncio queue without blocking
from random import random
import asyncio

# 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('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('Consumer: got nothing, waiting...')
            await asyncio.sleep(0.5)
            continue
```

```
        # check for stop
        if item is None:
            break
        # report
        print(f'>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 event loop
asyncio.run(main())
```

Running the example creates the shared `asyncio.Queue`, then starts the consumer and producer coroutines, as before.

The producer coroutine generates, blocks, and adds items to the queue.

The consumer coroutine attempts to get a value from the queue. If there is no value to retrieve, an `asyncio.QueueEmpty` exception is raised and handled by reporting a message, sleeping for a fraction of a second then starting the busy wait loop again.

Otherwise, if there is a value in the queue, the consumer will retrieve it and report it as normal.

We can see the messages from the consumer coroutine busy waiting for new data to arrive in the queue.

This highlights how to get items from the `asyncio.Queue` without blocking.

Note that the program output will differ each time it is run given the use of random numbers.

```
Producer: Running
Consumer: Running
Consumer: got nothing, waiting...
Consumer: got nothing, waiting...
>got 0.896558357626797
Consumer: got nothing, waiting...
Consumer: got nothing, waiting...
>got 0.6498874449486562
>got 0.14862534743361389
Consumer: got nothing, waiting...
```

```
Consumer: got nothing, waiting...
>got 0.9271724543351715
Consumer: got nothing, waiting...
>got 0.6659822945662333
>got 0.11205862071348183
Consumer: got nothing, waiting...
Consumer: got nothing, waiting...
>got 0.9490125408623084
Consumer: got nothing, waiting...
>got 0.150509682492045
>got 0.23281901173320807
Consumer: got nothing, waiting...
Consumer: got nothing, waiting...
Producer: Done
>got 0.8999468879239988
Consumer: Done
```

Next, let's look at how we might wait on the queue and mark tasks as completed in the queue.

26.5 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`.

The producer coroutine can be updated to no longer send a `None` value into the queue to indicate no further tasks.

The updated version of the `producer()` function with this change is listed below.

```
# 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)
```

```
    print('Producer: Done')
```

The consumer coroutine can be updated to no longer check for `None` messages, and to mark each task as completed via a call to `task_done()`.

The updated version of the `consumer()` function with these changes is listed below.

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

The producer coroutine will run until there are no longer any tasks to add to the queue and will terminate. The consumer coroutine will now run forever.

We will start the consumer coroutine first as a separate standalone task.

Next, we will execute and await the producer coroutine until it is done and all items have been added to the queue.

Finally, we will wait for all items added to the queue to be marked as done.

The updated `main()` coroutine is listed below.

```
# 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()
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of using join and task_done with a queue
```

```
from random import random
import asyncio

# 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)
    print('Producer: Done')

# coroutine to consume work
async def consumer(queue):
    print('Consumer: Running')
    # consume work
    while True:
        # get a unit of work
        item = await queue.get()
        # report
        print(f'>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 event loop
asyncio.run(main())
```

Running the example creates the shared `asyncio.Queue`, then starts the consumer and producer coroutines, as before.

The producer coroutine generates, blocks, and adds items to the queue.

The consumer coroutine attempts to get values from the queue.

Once the producer coroutine has added all of its ten items, it will terminate. The consumer coroutine will run forever in the background.

Once the producer coroutine has terminated, the main coroutine will then block on the queue.

Once the consumer coroutine has processed all items that were added to the queue, the main coroutine will then resume and terminate.

The asyncio program then terminates with the consumer coroutine still running in the background, but with no work to process.

This highlights how to mark tasks as completed in the queue and how to wait on the queue for all work to be completed.

Note that the program output will differ each time it is run given the use of random numbers.

```
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
```

26.6 Takeaways

You now know how to share data between coroutines using queues.

Specifically, you know:

- How to create and use a queue to connect coroutines and tasks in asyncio.
- How to use a queue without blocking and with timeouts.
- How to join a queue and have tasks done, and limit the overall capacity of the queue.

26.6.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

26.6.1.1 References

- Queue (abstract data type), Wikipedia.
  https://en.wikipedia.org/wiki/Queue_(abstract_data_type)

26.6.1.2 APIs

- `random` – Generate pseudo-random numbers.
  https://docs.python.org/3/library/random.html
- `queue` – A synchronized queue class.
  https://docs.python.org/3/library/queue.html
- Asyncio Queues.
  https://docs.python.org/3/library/asyncio-queue.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

26.6.2 Next

In the next tutorial, we will explore how to use mutual exclusion locks to protect critical sections in asyncio programs.

Part VI

Synchronization

Chapter 27

Locks

Mutual exclusion locks or mutex locks for short can be used to protect critical sections of code from concurrent execution.

It is possible to suffer race conditions if two or more coroutines operate upon the same variables, or if tasks are executed out of order.

We can use locks to define atomic blocks of code where execution is serialized, e.g. made sequential.

In this tutorial, you will discover how to use mutex locks to protect critical sections in asyncio.

After completing this tutorial, you will know:

- The importance of mutex locks in concurrent programming.
- How we may suffer race conditions in asyncio with coroutines.
- How to use mutex locks in asyncio programs.

Let's get started.

27.1 What Is A Mutual Exclusion Lock

A mutual exclusion lock or mutex lock is a synchronization primitive intended to prevent a race condition.

A race condition is a concurrency failure case when two units of concurrency (processes, threads, or coroutines) run the same code and access or update the same resource (e.g. data variables, stream, etc.) leaving the resource in an unknown and inconsistent state.

Although race conditions and mutex locks are often described in the context of threads, they are just as relevant and applicable to coroutines in asyncio.

Race conditions often result in unexpected behavior of a program or corrupt data.

These sensitive parts of code that can be executed by multiple coroutines concurrently and may result in race conditions are called critical sections. A critical section may refer to

a single block of code, but it also refers to multiple accesses to the same data variable or resource from multiple functions.

A mutex lock can be used to ensure that only one coroutine at a time executes a critical section of code, while all other coroutines trying to execute the same code must wait until the currently executing coroutine is finished with the critical section and releases the lock.

Each coroutine must attempt to acquire the lock at the beginning of the critical section. If the lock has not been obtained, then a coroutine will acquire it and other coroutines must wait until the coroutine that acquired the lock releases it.

If the lock has not been acquired, we might refer to it as being in the "unlocked" state. Whereas if the lock has been acquired, we might refer to it as being in the "locked" state.

- **Unlocked**: The lock has not been acquired and can be acquired by the next coroutine that makes an attempt.
- **Locked**: The lock has been acquired by one coroutine and any coroutine that makes an attempt to acquire it must wait until it is released.

Locks are created in the unlocked state.

Now that we know what a mutex lock is, let's look at why we need them in an asyncio program.

27.2 Why Do We Need Locks?

All coroutines for an event loop run in one thread, and a thread runs in one process.

Therefore, coroutines are both thread-safe and process-safe, as long as they are using resources that are not used by other threads or processes.

Nevertheless, coroutines may access state or resources that are not coroutine-safe.

How can this be, especially considering only one coroutine can run at a time within an event loop?

Consider a block of code that may be executed by multiple coroutines concurrently and has at least a point where the coroutine is suspended, e.g. an `await` expression or similar.

This block of code may manipulate program state, and data, or access an external resource.

If multiple coroutines execute this block concurrently, then the program state, data, or external resource may be left in an inconsistent or corrupt state, or data may be lost.

That is, coroutines may have critical sections that span many lines of code and must be protected from concurrent execution.

An obvious example is multiple coroutines reading and writing to/from the same socket.

A less obvious example is multiple coroutines that update shared variables while executing some task.

A necessary condition for coroutines to suffer a race condition in an asyncio program is that the critical section must suspend, allowing one or more other coroutines to execute the same critical section, or manipulate the same state or resources.

Note that race conditions are a real problem in Python when using coroutines, even in the presence of the global interpreter lock (GIL). The refrain that "*there are no race conditions in Python because of the GIL*" is dangerously wrong.

Let's dig into this further.

27.2.1 Example Of An Asyncio Race Condition

Coroutines can suffer race conditions with shared memory used among multiple coroutines.

They are just more obvious than similar race conditions with Python threads.

For example, a common race condition with Python threads involves when adding or subtracting a value from an integer.

```
...
# add a value to an integer
value += 1
```

This is a possible source of race conditions because this expression is in fact at least three Python bytecode instructions:

1. Read the current value of the variable.
2. Calculate a new value for the variable.
3. Write a new value for the variable.

A Python thread may be context switched by the operating system in the middle of these operations, leaving the value in an inconsistent state.

Race conditions with this expression are not possible with asyncio coroutines.

The reason is that a coroutine is only context switched when we explicitly suspend via the `await` expression or similar expressions such as `async for` or `async with`.

Nevertheless, this type of race condition can happen in an asyncio program, as long as we space out the operations and suspend the coroutine between or more of the operations.

For example:

```
...
# retrieve the value
tmp = value
# suspend for a moment
await asyncio.sleep(0)
# update the tmp value
tmp = tmp + 1
# suspend for a moment
await asyncio.sleep(0)
```

```
# store the updated value
value = tmp
```

Executing this block among multiple coroutines where `value` is shared will result in a race condition.

The result will be a value of the shared variable that will be unknown and consistent from one run of the program to the next.

The example below demonstrates this.

```
# SuperFastPython.com
# example of asyncio race condition with shared memory
import asyncio

# task that operates on a shared variable
async def task():
    # declare global variable
    global value
    # retrieve the value
    tmp = value
    # suspend for a moment
    await asyncio.sleep(0)
    # update the tmp value
    tmp = tmp + 1
    # suspend for a moment
    await asyncio.sleep(0)
    # store the updated value
    value = tmp

# main coroutine
async def main():
    # declare the global variable
    global value
    # define the global variable
    value = 0
    # create many coroutines to update the global sate
    coros = [task() for _ in range(10000)]
    # execute all coroutines
    await asyncio.gather(*coros)
    # report the value of the counter
    print(value)

# entry point
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and executes it as the entry point of the program.

The `main()` coroutine runs and initializes the `value` global variable to zero.

Then a total 10,000 coroutines are then created and awaited using `gather()` allowing all coroutines to run concurrently. Each coroutine attempts to add one to the shared variable. Therefore, we expect the final value of the variable to be 10,000.

Each coroutine runs, copying the value of the global variable to a local variable, suspending, updating the temporary variable, sleeping, then storing the updated variable back into the global variable.

All the coroutines finish and the `main()` coroutine reports the value of the global variable.

The value is 1, instead of the expected 10,000.

This highlights a contrived, but very real example of a race condition in an asyncio program.

```
1
```

What happened?

We might have expected each coroutine to execute as an atomic block, but it didn't.

Each coroutine was suspended (like a context switch) two times, once after copying the global variable, and once after incrementing the temporary variable.

The first coroutine runs and suspends at the first point, the second coroutine runs and suspends at the first point, and so on. This continues on for all 10,000 coroutines.

They have all copied the current value of the global variable of zero into the temporary value.

The first coroutine resumes and increments the temporary variable from zero to one then suspends. The second coroutine does the same, and so on for all 10,000 coroutines.

They all have a value of one in their temporary variables.

Finally, the first coroutine copies the temporary variable into the global variable. The second coroutine does the same and so on for all 10,000 coroutines.

The global variable is set to one each time.

Therefore, when the `main()` coroutine reports the value of the global variable, it is one, instead of the expected 10,000.

Now that we know we need mutex locks in asyncio programs, let's look at how to use the `asyncio.Lock` class.

27.3 How To Use The Asyncio Lock

The `asyncio` module provides a mutex lock to protect critical sections in coroutines.

Only a single coroutine can hold the lock at any one time, and any other coroutines attempting to acquire the lock while it is held will wait and be suspended.

27.3.1 Create a Lock

Python provides a mutual exclusion lock for coroutines via the `asyncio.Lock` class.

This lock is neither thread-safe nor process-safe. It cannot be used to protect critical sections from multiple threads or multiple processes.

This means that we must be careful to ensure we only protect critical sections at the right level or unit of concurrency.

An instance of the lock must be created and shared among coroutines in order to protect a resource or critical sections.

For example:

```
...
# create a lock
lock = asyncio.Lock()
```

27.3.2 Acquire The Lock

The lock can be acquired by calling the `acquire()` coroutine.

This is done at the beginning of a critical section.

This call must be awaited because it may suspend if the lock is currently being held by another coroutine that is currently suspended.

This will suspend the calling coroutine and potentially allow the coroutine holding the lock to progress,

For example:

```
...
# acquire the lock
await lock.acquire()
```

27.3.3 Release The Lock

Once the critical section is completed, the lock can be released and made available again.

This can be achieved via the `release()` method.

This is not a blocking call.

```
...
# release the lock
lock.release()
```

27.3.4 Ensure Lock Is Released

The lock must always be released once we are finished with it.

If not, other coroutines awaiting it will never be able to acquire the lock and the program may be deadlocked (unable to progress).

As such, it may be a good practice to enclose the critical section in a try-finally structure, ensuring the `release()` method is always recalled regardless of how the critical section block is exited.

For example:

```
...
# acquire the lock
await lock.acquire()
try:
    # critical section
    # ...
finally:
    # release the lock
    lock.release()
```

We can achieve the same effect using the asynchronous context manager interface on the `Lock` class.

This is preferred as the lock always be released automatically as soon as the enclosed critical section block is exited.

Recall that we must use an asynchronous context manager via the `async with` expression.

For example:

```
...
# acquire the lock
async with lock:
    # critical section
    # ...
# lock is released automatically...
```

27.3.5 Check If Lock Is Held

Finally, we can also check if the lock is currently acquired by a coroutine via the `locked()` method.

For example:

```
...
# check if a lock is currently acquired
if lock.locked():
    # ...
```

The `asyncio.Lock` class is different from the `threading.Lock` and `multiprocessing.Lock` classes in that it does not support a timeout when acquiring the lock, or the ability to acquire the lock without blocking.

Now that we know how to use the `Lock` class, let's look at a worked example.

27.4 Example Of Using The Asyncio Lock

We can develop an example to demonstrate how to use the mutex lock.

First, we can define a target task coroutine that takes a lock as an argument and uses the lock to protect a contrived critical section.

In this case, the critical section involves reporting a message and blocking for a fraction of a second.

It is important that the critical section suspend in some way, in order to allow other coroutines to run and potentially interact with the same resources or critical sections that require the lock.

The `task()` coroutine below implements this.

```
# task coroutine with a critical section
async def task(lock, num, value):
    # acquire the lock to protect the critical section
    async with lock:
        # report a message
        print(f'>{num} got lock, sleeping for {value}')
        # block for a moment
        await asyncio.sleep(value)
```

We can then create one instance of the `asyncio.Lock` shared among the coroutines.

This can be achieved in the `main()` coroutine, used as the entry point to the program.

We can then create a large number of coroutines and pass the shared lock. Each coroutine will have a unique integer argument and a random floating point value between 0 and 1, which will be how long the coroutine will sleep while holding the lock.

The coroutines are created in a list comprehension and provided to the `asyncio.gather()` function. The `main()` coroutine will then block until all coroutines are complete.

The `main()` coroutine that implements this is listed below.

```
# entry point
async def main():
    # create a shared lock
    lock = asyncio.Lock()
    # create many concurrent coroutines
    coros = [task(lock, i, random()) for i in range(10)]
```

```
    # execute and wait for tasks to complete
    await asyncio.gather(*coros)
```

Tying this together, the complete example of using a lock is listed below.

```
# SuperFastPython.com
# example of an asyncio mutual exclusion (mutex) lock
from random import random
import asyncio

# task coroutine with a critical section
async def task(lock, num, value):
    # acquire the lock to protect the critical section
    async with lock:
        # report a message
        print(f'>{num} got lock, sleeping for {value}')
        # block for a moment
        await asyncio.sleep(value)

# entry point
async def main():
    # create a shared lock
    lock = asyncio.Lock()
    # create many concurrent coroutines
    coros = [task(lock, i, random()) for i in range(10)]
    # execute and wait for tasks to complete
    await asyncio.gather(*coros)

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine runs, first creating the shared lock.

It then creates a list of coroutines, each is passed the shared lock, a unique integer, and a random floating point value.

The list of coroutines is passed to the `gather()` function and the `main()` coroutine suspends until all coroutines are completed.

A task coroutine executes, acquires the lock, reports a message, then awaits the sleep, suspending.

Another coroutine resumes. It attempts to acquire the lock and is suspended, while it waits. This process is repeated with many if not all coroutines.

The first coroutines resumes, exits the block, and releases the lock automatically via the asynchronous context manager.

The first coroutine to wait on the lock resumes, acquires the lock, reports a message, and sleeps.

This process repeats until all coroutines are given an opportunity to acquire the lock, execute the critical section and terminate.

Once all tasks terminate, the `main()` coroutine resumes and terminates, closing the program.

The output from the program will differ each time it is run given the use of random numbers.

```
>0 got lock, sleeping for 0.5801066277650906
>1 got lock, sleeping for 0.9815199546438026
>2 got lock, sleeping for 0.9135704338693494
>3 got lock, sleeping for 0.5463432011778886
>4 got lock, sleeping for 0.23247917281544972
>5 got lock, sleeping for 0.6868532569722853
>6 got lock, sleeping for 0.1816689583467278
>7 got lock, sleeping for 0.05095982775540919
>8 got lock, sleeping for 0.12526189095920326
>9 got lock, sleeping for 0.7213748083743816
```

27.5 Example Fixing an Asyncio Race Condition

Remember the race condition example from earlier?

We can update it to use a mutex lock so that it no longer suffers the race condition.

This can be achieved by making the content of the `task()` coroutine an atomic block, executed only once the lock has been acquired.

This means that only one coroutine can hold the lock and execute the content of the coroutine at a time and the global variable is updated sequentially by coroutines as was intended.

The updated `task()` coroutine with this change is listed below.

```
# task that operates on a shared variable
async def task(lock):
    # acquire the lock
    async with lock:
        # declare global variable
        global value
        # retrieve the value
        tmp = value
        # suspend for a moment
        await asyncio.sleep(0)
        # update the tmp value
```

```
        tmp = tmp + 1
        # suspend for a moment
        await asyncio.sleep(0)
        # store the updated value
        value = tmp
```

The complete example with this change is listed below.

```
# SuperFastPython.com
# example of fixed race condition with shared memory
import asyncio

# task that operates on a shared variable
async def task(lock):
    # acquire the lock
    async with lock:
        # declare global variable
        global value
        # retrieve the value
        tmp = value
        # suspend for a moment
        await asyncio.sleep(0)
        # update the tmp value
        tmp = tmp + 1
        # suspend for a moment
        await asyncio.sleep(0)
        # store the updated value
        value = tmp

# main coroutine
async def main():
    # declare the global variable
    global value
    # define the global variable
    value = 0
    # create the shared lock
    lock = asyncio.Lock()
    # create many coroutines to update the global sate
    coros = [task(lock) for _ in range(10000)]
    # execute all coroutines
    await asyncio.gather(*coros)
    # report the value of the counter
    print(value)

# entry point
```

```
asyncio.run(main())
```

The example runs as before.

This time, the lock is passed to each task coroutine and must be acquired before the body is executed.

Only a single coroutine can acquire the lock at a time because acquiring the lock is mutually exclusive. All other coroutines must wait for the lock to be released, remaining suspended.

A single coroutine updates the global variable and releases the lock. The next coroutine acquires the lock and all other coroutines remain suspended, and so on.

The final value of the global variable is reported and has the expected value of 10,000.

This highlights how we can fix an asyncio race condition with shared data using a mutex lock.

```
10000
```

27.6 Takeaways

You now know how to use mutex locks to protect critical sections in asyncio programs.

Specifically, you know:

- The importance of mutex locks in concurrent programming.
- How we may suffer race conditions in asyncio with coroutines.
- How to use mutex locks in asyncio programs.

27.6.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

27.6.1.1 References

- Lock (computer science), Wikipedia.
  https://en.wikipedia.org/wiki/Lock_(computer_science)
- Mutual exclusion, Wikipedia.
  https://en.wikipedia.org/wiki/Mutual_exclusion

27.6.1.2 APIs

- `random` – Generate pseudo-random numbers.
  https://docs.python.org/3/library/random.html
- Asyncio Synchronization Primitives.
  https://docs.python.org/3/library/asyncio-sync.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html

- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

27.6.2 Next

In the next tutorial, we will explore how to use coroutine-safe event variables in asyncio programs.

Chapter 28

Events

It is common to need to share a variable between concurrent tasks that may be set and checked.

Asyncio provides a concurrency primitive that provides exactly this called an event. It is essentially a mutex lock and a boolean variable, but also offers the ability for a calling coroutine to wait for the event to be set, a capability not provided by a simple mutex.

In this tutorial, you will discover how to use an event concurrency primitive in asyncio.

After completing this tutorial, you will know:

- What is an event concurrency primitive and why we cannot just use a boolean variable.
- How to use asyncio event shared between coroutines and tasks.
- How an asyncio event might be used to trigger behavior across multiple concurrent tasks.

Let's get started.

28.1 What Is An Asyncio Event

An event provides a way to notify coroutines that something has happened.

This is achieved using a coroutine-safe manner that avoids race conditions.

An event manages an internal boolean flag that can be either set or not set.

Coroutines can check the status of the event, change the status of the event or wait on the event for it to be set.

Now that we know what an event is, let's look at how we might use it in an asyncio program.

28.2 How To Use An Asyncio Event

An event is a simple concurrency primitive that allows communication between coroutines.

It can be thought of like a coroutine-safe boolean variable or flag.

28.2.1 Create An Event

An `asyncio.Event` object wraps a boolean variable that can either be set (`True`) or not set (`False`).

First, an event object must be created and the event will be in the *not set* state.

```
...
# create an instance of an event
event = asyncio.Event()
```

28.2.2 Check If Event Is Set

Once created we can check if the event has been set via the `is_set()` method which will return `True` if the event is set, or `False` otherwise.

For example:

```
...
# check if the event is set
if event.is_set():
    # do something...
```

28.2.3 Set The Event

The event can be set via the `set()` method.

Any coroutines waiting on the event to be set will be notified.

For example:

```
...
# set the event
event.set()
```

28.2.4 Clear An Event

The event can be marked as not set (whether it is currently set or not) via the `clear()` method.

```
...
# mark the event as not set
event.clear()
```

28.2.5 Wait For Event To Be Set

Finally, coroutines can wait for the event to be set via the `wait()` method.

Calling this method will suspend until the event is marked as set (e.g. another coroutine calling the `set()` method). If the event is already set, the `wait()` method will return immediately.

```
...
# wait for the event to be set
await event.wait()
```

Now that we know how to use an `asyncio.Event`, let's look at a worked example.

28.3 Example Of An Asyncio Event

We can explore how to use an `asyncio.Event` object.

In this example, we will create a suite of coroutines that each will perform some processing and report a message. All coroutines will use an event to wait to be set before starting their work. The main coroutine will set the event and trigger the processing in all coroutines.

First, we can define a task coroutine that takes the shared `asyncio.Event` instance and a unique integer to identify the task.

```
# task coroutine
async def task(event, number):
    # ...
```

Next, we will wait for the event to be set before starting the processing work.

```
...
# wait for the event to be set
await event.wait()
```

Once triggered, the task will generate a random number, block for a moment and report a message.

```
...
# generate a random value between 0 and 1
value = random()
# block for a moment
await asyncio.sleep(value)
# report a message
print(f'Task {number} got {value}')
```

Tying this together, the complete task coroutine is listed below.

```
# task coroutine
async def task(event, number):
    # wait for the event to be set
    await event.wait()
    # generate a random value between 0 and 1
    value = random()
    # block for a moment
```

```
    await asyncio.sleep(value)
    # report a message
    print(f'Task {number} got {value}')
```

The main coroutine will first create the shared `asyncio.Event` instance, which will be in the not set state by default.

```
...
# create a shared event object
event = asyncio.Event()
```

Next, we can create and start five new coroutines specifying the `task()` coroutine with the event object and a unique integer as arguments.

```
...
# create and run the tasks
tasks = [asyncio.create_task(
    task(event, i)) for i in range(5)]
```

Finally, the main coroutine will block for a moment, then trigger the processing in all of the coroutines via the event object.

```
...
print('Main blocking...')
await asyncio.sleep(0)
# start processing in all tasks
print('Main setting the event')
event.set()
```

Finally, the main coroutine will block and wait for all tasks to complete via the `asyncio.wait()` function.

```
...
# await for all tasks  to terminate
_ = await asyncio.wait(tasks)
```

Tying this all together, the complete example is listed below.

```
# SuperFastPython.com
# example of using an asyncio event object
from random import random
import asyncio

# task coroutine
async def task(event, number):
    # wait for the event to be set
    await event.wait()
    # generate a random value between 0 and 1
    value = random()
    # block for a moment
```

```
    await asyncio.sleep(value)
    # report a message
    print(f'Task {number} got {value}')

# main coroutine
async def main():
    # create a shared event object
    event = asyncio.Event()
    # create and run the tasks
    tasks = [asyncio.create_task(
        task(event, i)) for i in range(5)]
    # allow the tasks to start
    print('Main blocking...')
    await asyncio.sleep(0)
    # start processing in all tasks
    print('Main setting the event')
    event.set()
    # await for all tasks  to terminate
    _ = await asyncio.wait(tasks)

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the asyncio program.

The `main()` coroutine runs and creates and schedules five task coroutines.

It then sleeps, suspending and allowing the tasks to run and start waiting on the event.

The main coroutine resumes, reports a message then sets the event to `True`. It then blocks and waits for all issued tasks to complete.

This triggers all five coroutines. They resume in turn perform their processing and report a message.

Note, results will vary each time the program is run given the use of random numbers.

This highlights how coroutines can wait for an event to be set and how we can signal to coroutines using an event.

```
Main blocking...
Main setting the event
Task 3 got 0.36705703414223256
Task 1 got 0.4852630342496812
Task 0 got 0.7251916806567016
Task 4 got 0.8104350284043036
Task 2 got 0.9726611709531982
```

28.4 Takeaways

You now know how to use an asyncio event.

Specifically, you know:

- What is an event concurrency primitive and why we cannot just use a boolean variable.
- How to use asyncio event shared between coroutines and tasks.
- How an asyncio event might be used to trigger behavior across multiple concurrent tasks.

28.4.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- `random` – Generate pseudo-random numbers.
  https://docs.python.org/3/library/random.html
- Asyncio Synchronization Primitives.
  https://docs.python.org/3/library/asyncio-sync.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

28.4.2 Next

In the next tutorial, we will explore how to coordinate coroutines with condition variables in asyncio.

Chapter 29

Conditions

We often need to coordinate the behavior between multiple concurrent tasks.

For example, it is common for one or more tasks to wait on an activity to be completed by another task.

Rather than requiring the task to complete and have other task wait upon it, we can use a concurrency primitive designed for this purpose called a monitor or a condition variable.

The condition variable implements the common wait/notify and wait/notify-all patterns for concurrent programming. Dependent tasks can wait on the condition variable and the target task can complete its work and notify all interested parties that a condition has been met and they are able to resume.

In this tutorial, you will discover how to use the condition variable in asyncio.

After completing this tutorial, you will know:

- What is a condition variable and the types of problems where it can be used.
- How to create and use a condition variable to signal between asyncio tasks and coroutines.
- How to develop examples that make use of the condition variable.

Let's get started.

29.1 What Is An Asyncio Condition Variable

In concurrent programming, a condition (also called a monitor) allows multiple coroutines to be notified about some result.

It combines both a mutual exclusion lock (mutex) and an event allowing exclusive access and notification.

A mutex can be used to protect a critical section, but it cannot be used to alert other coroutines that a condition has changed or been met.

An event can be used to notify other coroutines, but it cannot be used to protect a critical section and enforce mutual exclusion.

A condition can be acquired by a coroutine (like a mutex) after which it can wait to be notified by another coroutine that something has changed like an event. While waiting, the coroutine is blocked and releases the lock for other coroutines to acquire.

Another coroutine can then acquire the condition, make a change, and notify one, all, or a subset of coroutines waiting on the condition that something has changed. The waiting coroutine can then wake up (be scheduled by the operating system), re-acquire the condition (mutex), perform checks on any changed state, and perform required actions.

Now that we know what a condition is, let's look at how we might use it in an asyncio program.

29.2 How To Use An Asyncio Condition Variable

In this section, we will explore how to use the asyncio condition variable.

29.2.1 Create A Condition Variable

Python provides a condition for use with coroutines via the `asyncio.Condition` class.

To use a condition variable, we must create an instance of the class.

```
...
# create a new condition
condition = asyncio.Condition()
```

The condition object may then be shared and used among multiple asyncio coroutines.

29.2.2 Acquire And Release Condition Variable

In order for a coroutine to make use of the condition, it must acquire it and release it, like a mutex lock.

This can be achieved manually with the `acquire()` and `release()` methods.

Acquiring the condition via `acquire()` returns a coroutine object that requires that the caller use an `await` expression.

For example, we can acquire the condition, do something, then release the condition as follows:

```
...
# acquire the condition
await condition.acquire()
# do something
# ...
```

```
# release the condition
condition.release()
```

An alternative to calling the `acquire()` and `release()` methods directly is to use the asynchronous context manager, which will perform the acquire/release automatically for us.

This is the preferred usage of the condition variable, where appropriate.

For example:

```
...
# acquire the condition
async with condition:
    # do something
    # ...
```

29.2.3 Wait To Be Notified

Once the condition is acquired, we can wait on it.

This will suspend the calling coroutine until another coroutine notifies it via the condition with the `notify()` method (seen later).

This can be achieved via the `wait()` method that will return a coroutine and must be awaited.

For example:

```
...
# acquire the condition
async with condition:
    # wait to be notified
    await condition.wait()
```

Importantly, the condition is relinquished while waiting. This allows other coroutines to acquire it in order to also wait or to notify waiting coroutines.

29.2.4 Wait For Condition

The `asyncio.ondition` class also provides a `wait_for()` method that can be used to only unlock the waiting coroutine if a condition is met, such as calling a function that returns a boolean value.

The name of the function that returns a boolean value can be provided to the `wait_for()` method directly.

Like the `wait()` method, it returns a coroutine object that must be awaited.

For example:

```
...
# acquire the condition
async with condition:
```

```
    # wait to be notified and a function to return true
    await condition.wait_for(all_data_collected)
```

The condition is only checked when the coroutine is notified.

If the condition is not met, the coroutine may be notified many times and will not resume until the condition is met.

29.2.5 Notify Waiting Coroutines

We also must acquire the condition in a coroutine if we wish to notify waiting coroutines.

This too can be achieved directly with the acquire/release method calls or via the context manager.

We can notify a single waiting coroutine via the `notify()` method.

For example:

```
...
# acquire the condition
async with condition:
    # notify a waiting coroutine
    condition.notify()
```

The notified coroutine will resume as soon as it can re-acquire the mutex within the condition. This will be attempted automatically as part of its call to `wait()`, we do not need to do anything extra.

If there are more than one coroutines waiting on the condition, we will not know which coroutine will be notified.

We can also notify a subset of waiting coroutines by setting the `n` argument to an integer number of coroutines to notify, for example:

```
...
# acquire the condition
async with condition:
    # notify 3 waiting coroutines
    condition.notify(n=3)
```

29.2.6 Notify All Waiting Coroutines

Finally, we can notify all coroutines waiting on the condition via the `notify_all()` method.

```
...
# acquire the condition
async with condition:
    # notify all coroutines waiting on the condition
    condition.notify_all()
```

A reminder, a coroutine must acquire the condition before waiting on it or notifying waiting coroutines.

A failure to acquire the condition (the lock within the condition) before performing these actions will result in a `RuntimeError`.

Now that we know how to use the `asyncio.Condition` class, let's look at some worked examples.

29.3 Example Of Wait And Notify

In this section, we will explore using an `asyncio.Condition` to notify a waiting coroutine that something has happened.

We will use a task to prepare some data and notify a waiting coroutine, and in the main coroutine, we will create and schedule the new task and use the condition to wait for the work to be completed.

The complete example is listed below.

```
# SuperFastPython.com
# example of wait/notify with an asyncio condition
import asyncio

# task coroutine
async def task(condition, work_list):
    # block for a moment
    await asyncio.sleep(1)
    # add data to the work list
    work_list.append(33)
    # notify a waiting coroutine that the work is done
    print('Task sending notification...')
    async with condition:
        condition.notify()

# main coroutine
async def main():
    # create a condition
    condition = asyncio.Condition()
    # prepare the work list
    work_list = list()
    # wait to be notified that the data is ready
    print('Main waiting for data...')
    async with condition:
        # create and start the task
        _ = asyncio.create_task(
```

```
            task(condition, work_list))
        # wait to be notified
        await condition.wait()
    # we know the data is ready
    print(f'Got data: {work_list}')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine which is used as the entry point into the asyncio program.

The `main()` coroutine runs and creates the shared condition and the work list.

The `main()` coroutine then acquires the condition. A new task is created and scheduled, provided the shared condition and work list.

The `main()` coroutine then waits to be notified, suspending and calling the new scheduled task to run.

The `task()` coroutine runs. It first blocks for a moment to simulate effort, then adds work to the shared list. The condition is acquired and the waiting coroutine is notified, then releases the condition automatically. The task terminates.

The `main()` coroutine resumes and reports a final message, showing the updated content of the shared list.

This highlights how we can use a wait-notify pattern between coroutines using a condition variable.

```
Main waiting for data...
Task sending notification...
Got data: [33]
```

Next, let's look at how we might notify all waiting coroutines.

29.4 Example Of Wait And Notify All

We can explore how to notify all coroutines waiting on a condition.

In this example, we will start a suite of tasks that will wait on the condition to be notified before performing their processing and reporting a result.

The main coroutine will block for a moment and then notify all waiting coroutines that they can begin processing.

The complete example is listed below.

```
# SuperFastPython.com
# example of wait/notify all with an asyncio condition
```

```
from random import random
import asyncio

# task coroutine
async def task(condition, number):
    # report a message
    print(f'Task {number} waiting...')
    # acquire the condition
    async with condition:
        # wait to be notified
        await condition.wait()
    # generate a random number between 0 and 1
    value = random()
    # block for a moment
    await asyncio.sleep(value)
    # report a result
    print(f'Task {number} got {value}')

# main coroutine
async def main():
    # create a condition
    condition = asyncio.Condition()
    # create and start many tasks
    tasks = [asyncio.create_task(
        task(condition, i)) for i in range(5)]
    # allow the tasks to run
    await asyncio.sleep(1)
    # acquire the condition
    async with condition:
        # notify all waiting tasks
        condition.notify_all()
    # wait for all tasks to complete
    _ = await asyncio.wait(tasks)

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine which is used as the entry point into the asyncio program.

The `main()` coroutine runs and creates the shared condition.

It then creates and schedules five tasks, each providing the shared condition and a unique integer from 0 to 4 as arguments.

The `main()` coroutine then suspends, allowing the tasks to run.

The tasks run one by one. The tasks first report their message, then acquire the condition. Once acquired they wait to be notified.

The `main()` coroutine resumes. It acquires the condition and then notifies all coroutines waiting on the condition. It then releases the condition and waits for the issued tasks to terminate.

The tasks resume, one at a time. Each task first generates a random number, and blocks for a fraction of a second. Once resumed it reports a message and terminates.

This highlights how we may have many coroutines waiting to be notified and have them all notified.

```
Task 0 waiting...
Task 1 waiting...
Task 2 waiting...
Task 3 waiting...
Task 4 waiting...
Task 4 got 0.11036200324308998
Task 3 got 0.25497519885869324
Task 1 got 0.36215401216779797
Task 0 got 0.4277021597975379
Task 2 got 0.7073867691766996
```

Next, let's look at how we might wait for a specific result on the condition.

29.5 Example Of Wait For

We can explore how to use the `wait_for()` method on the condition.

The `wait_for()` method takes a callable, such as a function with no arguments or a lambda expression. The coroutine calling the `wait_for()` method will block until notified and the callable passed in as an argument returns a `True` value.

This might mean that the coroutine is notified many times by different coroutines, but will only unblock and continue execution once the condition in the callable is met.

In this example, will create a suite of tasks, each of which will calculate a value and add it to a shared list and notify the waiting coroutine.

The main coroutine will wait on the condition and will use a lambda expression in the `wait_for()` method to not continue on until a work list populated by the worker coroutines is fully populated.

The complete example is listed below.

```
# SuperFastPython.com
# example of wait for with a condition
from random import random
```

```
import asyncio

# task coroutine
async def task(condition, work_list):
    # acquire the condition
    async with condition:
        # generate a random value between 0 and 1
        value = random()
        # block for a moment
        await asyncio.sleep(value)
        # add work to the list
        work_list.append(value)
        print(f'Task added {value}')
        # notify the waiting coroutine
        condition.notify()

# main coroutine
async def main():
    # create a condition
    condition = asyncio.Condition()
    # define work list
    work_list = list()
    # create and start many tasks
    _ = [asyncio.create_task(
        task(condition, work_list)) for _ in range(5)]
    # acquire the condition
    async with condition:
        # wait to be notified
        await condition.wait_for(
            lambda : len(work_list)==5)
        # report final message
        print(f'Done, got: {work_list}')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine which is used as the entry point into the asyncio program.

The `main()` coroutine runs and first creates the shared condition variable and work list.

The `main()` coroutine then creates and schedules five tasks, passing the condition and work list.

The `main()` coroutine then acquires the condition and waits to be notified and for the lambda expression to be true, which in this case requires that the work list has a length of 5.

The tasks then execute one by one. Each task first acquires the condition, generates a random value, and blocks. Once it resumes it adds a value to the shared work list, reports a message, and notifies the waiting `main()` coroutine.

The `main()` coroutine is notified many times but does not resume until the condition is met, after which the final message is reported.

Importantly, only one coroutine can hold or acquire the condition at a time, although recall that the condition is released when waiting to be notified. In this case, the tasks do not wait, although they are suspended. The lock is held during this block, making the block of the condition context manager coroutine-safe, e.g. mutually exclusive.

This highlights how we may use the wait-for and notify pattern with coroutines.

```
Task added 0.45089927950158515
Task added 0.8931671797432676
Task added 0.06484378382773981
Task added 0.7649095074042099
Task added 0.9954569956651376
Done, got: [0.45089927950158515, 0.8931671797432676,
    0.06484378382773981, 0.7649095074042099,
    0.9954569956651376]
```

29.6 Takeaways

You now know how to use an asyncio condition variable.

Specifically, you know:

- What is a condition variable and the types of problems where it can be used.
- How to create and use a condition variable to signal between asyncio tasks and coroutines.
- How to develop examples that make use of the condition variable.

29.6.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

29.6.1.1 References

- Monitor (synchronization), Wikipedia.
  https://en.wikipedia.org/wiki/Monitor_(synchronization)

29.6.1.2 APIs

- `random` – Generate pseudo-random numbers.
  https://docs.python.org/3/library/random.html

- Asyncio Synchronization Primitives.
  https://docs.python.org/3/library/asyncio-sync.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

29.6.2 Next

In the next tutorial, we will explore how to synchronize coroutines with semaphores in asyncio.

Chapter 30

Semaphores

A semaphore is a concurrency primitive that is used to signal between concurrent tasks.

Semaphores are configurable and versatile, allowing them to be used like a mutex to protect a critical section, but also to be used as a coroutine-safe counter or a gate to protect a limited resource.

In this tutorial, you will discover how to use semaphores in asyncio.

After completing this tutorial, you will know:

- What is a semaphore and how can they be used in concurrent programming.
- How to use a semaphore among coroutines and tasks in asyncio.
- How to develop examples to explore the capabilities of semaphores with concurrent tasks.

Let's get started.

30.1 What Is An Asyncio Semaphore

A semaphore is a concurrency primitive that allows a limit on the number of coroutines that can acquire a lock protecting a critical section.

It is an extension of a mutual exclusion (mutex) lock that adds a count for the number of coroutines that can acquire the lock before additional coroutines will block.

Once at capacity (no more positions are available), new coroutines can only acquire a position on the semaphore once an existing coroutine holding the semaphore releases a position.

Internally, the semaphore maintains a counter protected by a mutex lock that is decremented each time the semaphore is acquired and incremented each time it is released.

When a semaphore is created, the upper limit on the counter is set. If it is set to 1, then the semaphore will operate like a mutex lock.

A semaphore provides a useful concurrency tool for limiting the number of coroutines that can access a resource concurrently. Some examples include:

- Limiting concurrent socket connections to a server.
- Limiting concurrent file operations on a hard drive.
- Limiting concurrent calculations.

Now that we know what a semaphore is, let's look at how we might use it in asyncio.

30.2 How To Use An Asyncio Semaphore

The `asyncio` module provides a semaphore for synchronizing coroutines.

30.2.1 Create A Semaphore

Python provides a semaphore via the `asyncio.Semaphore` class for use with coroutines.

The `asyncio.Semaphore` instance must be configured when it is created to set the limit on the internal counter. This limit will match the number of concurrent coroutines that can hold the semaphore at a time.

For example:

```
...
# create a semaphore with a limit of 100
semaphore = asyncio.Semaphore(100)
```

30.2.2 Acquire A Semaphore

The semaphore can be acquired by calling the `acquire()` method.

This returns a coroutine that must be awaited.

For example:

```
...
# acquire the semaphore
await semaphore.acquire()
```

30.2.3 Release A Semaphore

Once acquired, the semaphore can be released again by calling the `release()` method.

For example:

```
...
# release the semaphore
semaphore.release()
```

The `release()` method can be called many times, e.g. more times than the `acquire()` method was called.

Each time `release()` is called, it will increment the internal counter of the semaphore, regardless of the initial value. This can be used to increase the capacity of the semaphore dynamically.

For example:

```
...
# acquire the semaphore
semaphore.acquire()
# do something
# ...
# release the semaphore
semaphore.release()
# increase the capacity of the semaphore
for i in range(5):
    semaphore.release()
```

30.2.4 Ensure Semaphore Is Released

The `asyncio.Semaphore` class supports the asynchronous context manager interface, which will automatically acquire and release the semaphore for us.

As such it is the preferred usage, if appropriate for our program.

For example:

```
...
# acquire the semaphore
async with semaphore:
    # ...
```

30.2.5 Check If Semaphore Is At Capacity

Finally, a coroutine can check if the semaphore is current at capacity via the `locked()` method.

It will return `True` if the semaphore cannot be acquired at the time of the call, or `False` otherwise.

For example:

```
...
# check if there is space on the semaphore
if not semaphore.locked():
    # ...
```

Now that we know how to use the asyncio `Semaphore`, let's look at a worked example.

30.3 Example Of Using An Asyncio Semaphore

We can explore how to use an `asyncio.Semaphore` with a worked example.

We will develop an example with a suite of coroutines but a limit on the number of coroutines that can perform an action simultaneously.

A semaphore will be used to limit the number of concurrent coroutines which will be less than the total number of coroutines, allowing some coroutines to suspend, wait for a position, then be notified and acquire a position on the semaphore.

The complete example is listed below.

```
# SuperFastPython.com
# example of using an asyncio semaphore
from random import random
import asyncio

# task coroutine
async def task(semaphore, number):
    # acquire the semaphore
    async with semaphore:
        # generate a random value between 0 and 1
        value = random()
        # block for a moment
        await asyncio.sleep(value)
        # report a message
        print(f'Task {number} got {value}')

# main coroutine
async def main():
    # create the shared semaphore
    semaphore = asyncio.Semaphore(2)
    # create and schedule tasks
    tasks = [asyncio.create_task(
        task(semaphore, i)) for i in range(10)]
    # wait for all tasks to complete
    _ = await asyncio.wait(tasks)

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the `main()` coroutine that is used as the entry point into the asyncio program.

The `main()` coroutine runs and first creates the shared semaphore with an initial counter value of 2, meaning that two coroutines can hold the semaphore at once.

The `main()` coroutine then creates and schedules 10 tasks to execute our `task()` coroutine, passing the shared semaphore and a unique number between 0 and 9.

The `main()` coroutine then suspends and waits for all tasks to complete.

The tasks run one at a time.

Each task first attempts to acquire the semaphore. If there is a position available it proceeds with the task, otherwise it waits for a position to become available.

Once acquired, a task generates a random value, suspends for a moment, and then reports the generated value. It then releases the semaphore and terminates. The semaphore is not released while the task is suspended in the call to `sleep()`.

The body of the semaphore context manager is limited to two semaphores at a time.

This highlights how we can limit the number of coroutines to execute a block of code concurrently.

The output from the program will differ each time it is run given the use of random numbers.

```
Task 0 got 0.20369168197618748
Task 2 got 0.20640107131350838
Task 1 got 0.6855263719449817
Task 3 got 0.9396433586858612
Task 4 got 0.8039832235015294
Task 6 got 0.12266060196253203
Task 5 got 0.879466225105295
Task 7 got 0.6675244153844875
Task 8 got 0.11511060306129695
Task 9 got 0.9607702805925814
```

30.4 Takeaways

You now know how to use semaphores in asyncio programs.

Specifically, you know:

- What is a semaphore and how can they be used in concurrent programming.
- How to use an semaphore among coroutines and tasks in asyncio.
- How to develop examples to explore the capabilities of semaphores with concurrent tasks.

30.4.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

30.4.1.1 References

- Semaphore (programming), Wikipedia.
  https://en.wikipedia.org/wiki/Semaphore_(programming)

30.4.1.2 APIs

- `random` – Generate pseudo-random numbers.
  https://docs.python.org/3/library/random.html
- Asyncio Synchronization Primitives.
  https://docs.python.org/3/library/asyncio-sync.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

30.4.2 Next

In the next tutorial, we will explore how to coordinate coroutines with barriers in asyncio.

Chapter 31

Barriers

A barrier is another synchronization primitive.

It is used to coordinate the behavior of concurrent tasks at one point, on the barrier itself. Once all expected parties arrive at the barrier, the barrier is lifted and all waiting tasks are able to resume their activity.

In this tutorial, you will discover how to use the barrier in asyncio programs.

After completing this tutorial, you will know:

- What is a barrier and how can they be used in concurrent programming.
- How to use a barrier to coordinate among coroutines and tasks in asyncio.
- How to develop examples to explore the capabilities of barrier with concurrent tasks.

Let's get started.

31.1 What Is An Asyncio Barrier

A barrier is a synchronization primitive.

It allows multiple coroutines to wait on the same barrier object instance (e.g. at the same point in code) until a predefined fixed number of coroutines arrive (e.g. the barrier is full), after which all coroutines are then notified and released to continue their execution.

Internally, a barrier maintains a count of the number of coroutines waiting on the barrier and a configured maximum number of parties (coroutines) that are expected. Once the expected number of parties reaches the pre-defined maximum, all waiting coroutines are notified.

This provides a useful mechanism to coordinate actions between multiple coroutines.

Now that we know what a barrier is, let's look at how we might use it in asyncio programs.

31.2 How To Use The Asyncio Barrier

The `asyncio` module provides a barrier to safely coordinate the behavior of coroutines.

The asyncio barrier was introduced to the Python standard library in version 3.11.

31.2.1 Create A Barrier

Python provides an asyncio barrier via the `asyncio.Barrier` class.

A barrier instance must first be created and configured via the constructor specifying the number of parties (coroutines) that must arrive before the barrier will be lifted.

For example:

```
...
# create a barrier
barrier = asyncio.Barrier(10)
```

31.2.2 Wait On The Barrier

Once configured, the barrier instance can be shared between coroutines and used.

A coroutine can reach and wait on the barrier via the `wait()` method which must be awaited, for example:

```
...
# wait on all other tasks to complete
await barrier.wait()
```

This will suspend the caller until all parties reach and wait on the barrier.

31.2.3 Wait And Get Remaining Parties

The `wait()` method returns an integer indicating the number of parties remaining to arrive at the barrier.

If a coroutine was the last coroutine to arrive, then the return value will be zero. This is helpful if we want the last coroutine or one coroutine to perform an action after the barrier is released.

For example:

```
...
# wait on all other tasks to complete
position = await barrier.wait()
# perform action if last to leave the barrier
if position == 0:
    # perform some action
    # ...
```

31.2.4 Wait With Context Manager

Callers can also wait on the `asyncio.Barrier` via the asynchronous context manager interface.

This is helpful if some action needs to be performed after all coroutine reaches the barrier.

For example:

```
...
# wait on all other tasks to complete
async with barrier:
    # perform some action
    # ...
```

31.2.5 Abort The Barrier

We can also abort the barrier.

Aborting the barrier means that all coroutines waiting on the barrier via the `wait()` method will raise an `asyncio.BrokenBarrierError` and the barrier will be put in the broken state.

This might be helpful if we wish to cancel the coordination effort.

```
...
# abort the barrier
await barrier.abort()
```

A broken barrier can no longer be used. Calls to `wait()` will raise a `BrokenBarrierError`.

31.2.6 Reset A Barrier

An aborted and broken barrier can be fixed and made ready for use again by calling the `reset()` method.

This might be helpful if we cancel a coordination effort although we wish to retry it again with the same barrier instance.

```
...
# reset a broken barrier
await barrier.reset()
```

31.2.7 Barrier Attributes

Finally, the status of the barrier can be checked via attributes.

- `parties`: reports the configured number of parties that must reach the barrier for it to be lifted.
- `n_waiting`: reports the current number of coroutines waiting on the barrier.
- `broken`: attribute indicates whether the barrier is currently broken or not.

Now that we know how to use the barrier in asyncio, let's look at some worked examples.

31.3 Example Of Using An Asyncio Barrier

We can explore how to use an `asyncio.Barrier` with a worked example.

In this example, we will create a suite of coroutines, each required to perform some blocking calculation. We will use a barrier to coordinate all coroutines after they have finished their work and perform some action in the main coroutine. This is a good proxy for the types of coordination we may need to perform with a barrier.

First, let's define a target task function to execute by each coroutine. The coroutine will take the barrier as an argument as well as a unique identifier for the task.

The task will generate a random value between 0 and 10, sleep for that many seconds, report the result, then wait on the barrier for all other tasks to perform their computation.

The `work()` coroutine below implements this.

```
# coroutine that prepares work and waits on a barrier
async def work(barrier, number):
    # generate a unique value
    value = random() * 10
    # suspend for a moment to simulate work
    await asyncio.sleep(value)
    # report result
    print(f'Task {number} done, {value}, waiting...')
    # wait on all other tasks to complete
    await barrier.wait()
```

Next, in the main coroutine, we can create the barrier.

We need one party for each task we intend to create, five in this place, as well as an additional party for the main coroutine that will also wait for all coroutines to reach the barrier.

```
...
# create a barrier
n_tasks = 5
barrier = asyncio.Barrier(n_tasks + 1)
```

Next, we can create and schedule five `work()` tasks to run in the background, providing the barrier to each.

```
...
# issue all of the tasks
_ = [asyncio.create_task(
    work(barrier, i)) for i in range(n_tasks)]
```

The main coroutine can wait on the barrier for all tasks to perform their calculation.

We will use the context manager interface in this case.

Once the coroutines are finished, the barrier will be lifted and the worker coroutines will terminate and the main coroutine will report a message.

```
...
# wait for all tasks to finish
print('Main is waiting on all results...')
async with barrier:
    # report once all tasks are done
    print('All tasks have their result')
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# example of an asyncio barrier
from random import random
import asyncio

# coroutine that prepares work and waits on a barrier
async def work(barrier, number):
    # generate a unique value
    value = random() * 10
    # suspend for a moment to simulate work
    await asyncio.sleep(value)
    # report result
    print(f'Task {number} done, {value}, waiting...')
    # wait on all other tasks to complete
    await barrier.wait()

# main coroutine
async def main():
    # create a barrier
    n_tasks = 5
    barrier = asyncio.Barrier(n_tasks + 1)
    # issue all of the tasks
    _ = [asyncio.create_task(
        work(barrier, i)) for i in range(n_tasks)]
    # wait for all tasks to finish
    print('Main is waiting on all results...')
    async with barrier:
        # report once all tasks are done
        print('All tasks have their result')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first starts the event loop and runs the `main()` coroutine.

The `main()` coroutine runs and creates the barrier with 6 parties. It then creates and issues 5 tasks and prints a message and waits on the barrier.

The `work()` tasks run. Each generates a random number fraction of 10 and sleeps for that many seconds.

The `work()` tasks resume and report a message that includes their random value, then wait on the barrier.

Once all `work()` tasks make it to the barrier, the barrier is released.

The `work()` tasks resume and terminate as they have no further work to complete.

The `main()` coroutine resumes and reports a final message.

This highlights how we can coordinate multiple coroutines using a barrier.

```
Main is waiting on all results...
Task 2 done, 0.38453387908173875, waiting...
Task 1 done, 0.47381619708659506, waiting...
Task 3 done, 0.6937143847759375, waiting...
Task 0 done, 1.9407832412145531, waiting...
Task 4 done, 5.5488912595411986, waiting...
All tasks have their result
```

31.4 Takeaways

You now know how to use the barrier synchronization primitive in asyncio programs.

Specifically, you know:

- What is a barrier and how can they be used in concurrent programming.
- How to use a barrier to coordinate among coroutines and tasks in asyncio.
- How to develop examples to explore the capabilities of barrier with concurrent tasks.

31.4.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

31.4.1.1 References

- Barrier (computer science), Wikipedia.
  https://en.wikipedia.org/wiki/Barrier_(computer_science)

31.4.1.2 APIs

- `random` – Generate pseudo-random numbers.
  https://docs.python.org/3/library/random.html
- What's New In Python 3.11.
  https://docs.python.org/3/whatsnew/3.11.html
- Asyncio Synchronization Primitives.
  https://docs.python.org/3/library/asyncio-sync.html

- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

31.4.2 Next

In the next tutorial, we will explore how to run and communicate with commands executed in subprocesses with asyncio.

Part VII

Streams

Chapter 32

Subprocesses

Sometimes we need to start, run, and interact with other programs from within our Python program.

Asyncio provides a way to run commands in subprocesses and to read and write from the subprocesses without blocking. This can be helpful when porting Python programs that run local commands as subprocesses to asyncio and in developing new programs that are required to interact with separate programs.

In this tutorial, you will discover how to use subprocesses in asyncio programs.

After completing this tutorial, you will know:

- What are subprocesses and how we can run new commands in subprocesses from asyncio.
- How to perform non-blocking read and write communication with programs in subprocesses.
- How to send signals, retrieve data, and terminate programs run in subprocesses.

Let's get started.

32.1 What Are Subprocesses

We can execute commands and run separate programs from asyncio as subprocesses.

These subprocesses are represented by the `Process` class in the `asyncio.subprocess` module.

a `Process` provides a handle on a subprocess in asyncio programs, allowing actions to be performed on it, such as waiting and terminating it.

The API is very similar to the `multiprocessing.Process` class and perhaps more so with the `subprocess.Popen` class.

Specifically, it shares methods such as `wait()`, `communicate()`, and `send_signal()` and attributes such as `stdin`, `stdout`, and `stderr` with the `subprocess.Popen`.

Now that we know about the asyncio `Process` class, let's look at how we might use it in our asyncio programs.

32.2 How To Create Subprocesses

We do not create a `asyncio.subprocess.Process` directly.

Instead, an instance of the class is created for us when executing a subprocess in an asyncio program.

There are two ways to execute an external program as a subprocess and acquire a `Process` instance, they are:

- `asyncio.create_subprocess_exec()`
- `asyncio.create_subprocess_shell()`

Let's look at examples of each in turn.

32.2.1 Create Subprocess Directly

The `asyncio.create_subprocess_exec()` function can be called to execute a command in a subprocess.

It returns a `asyncio.subprocess.Process` as a handle on the subprocess.

The `create_subprocess_exec()` function is a coroutine and must be awaited. It will suspend the caller until the subprocess is started (not completed).

For example:

```
...
# create as a subprocess using create_subprocess_exec
process = await asyncio.create_subprocess_exec(
    'echo', 'Hello World')
```

We can configure the subprocess to receive input from the asyncio program or send output to the asyncio program by setting the `stdin`, `stdout`, and `stderr` attributes to the `asyncio.subprocess.PIPE` constant.

This will set the `stdin`, `stdout`, and `stderr` attributes on the `asyncio.subprocess.Process` to be a `StreamReader` or `StreamWriter` and allow coroutines to read or write from them via the `communicate()` method in the subprocess (described later).

For example:

```
...
# create as a subprocess using create_subprocess_exec
process = await asyncio.create_subprocess_exec(
    'echo', 'Hello World',
    stdout=asyncio.subprocess.PIPE)
```

We can explore how to get a `Process` instance by executing a command in a subprocess with the `create_subprocess_exec()` function.

The example below executes the `echo` command in a subprocess that prints out the provided string.

The subprocess is started, then the details of the subprocess are then reported.

The complete example is listed below.

```
# SuperFastPython.com
# example of creating a subprocess
import asyncio

# main coroutine
async def main():
    # create as a subprocess
    process = await asyncio.create_subprocess_exec(
        'echo', 'Hello World')
    # report the details of the subprocess
    print(f'subprocess: {process}')

# start the asyncio event loop
asyncio.run(main())
```

Running the example executes the `echo` command in a subprocess. Note, this command may not be available on some platforms.

A `asyncio.subprocess.Process` instance is returned and the details are reported, showing the unique process id (PID).

The `echo` command then reports the provided string on stdout.

```
subprocess: <Process 50598>
Hello World
```

Next, let's look at an example of creating a subprocess via the `create_subprocess_shell()` function.

32.2.2 Create Subprocess With Shell

The `asyncio.create_subprocess_shell()` function can be called to execute a command in a subprocess.

Unlike the `create_subprocess_exec()` function above, the `create_subprocess_shell()` function will execute the provided command via the shell. This is the command line interpreter used to execute commands on the system, such as `bash` or `zsh` on Linux and macOS or `cmd.exe` on Windows.

Executing a command via the shell allows the capabilities of the shell to be used in addition to executing the command, such as wildcards and shell variables.

The function returns a `asyncio.subprocess.Process` as a handle on the subprocess.

The `create_subprocess_shell()` function is a coroutine and must be awaited. It will suspend the caller until the subprocess is started (not completed).

We can explore how to get a `Process` instance by executing a command in a subprocess with the `create_subprocess_shell()` function.

The example below executes the `echo` command in a subprocess that prints out the provided string. Unlike the `create_subprocess_exec()` function, the entire command with arguments is provided as a single string.

The subprocess is started, then the details of the subprocess are then reported.

The complete example is listed below.

```
# SuperFastPython.com
# example of creating a subprocess with shell
import asyncio

# main coroutine
async def main():
    # create as a subprocess
    process = await asyncio.create_subprocess_shell(
        'echo Hello World')
    # report the details of the subprocess
    print(f'subprocess: {process}')

# start the asyncio event loop
asyncio.run(main())
```

Running the example executes the `echo` command in a subprocess.

A `asyncio.subprocess.Process` instance is returned and the details are reported, showing the unique process id (PID).

The `echo` command then reports the provided string on stdout.

```
subprocess: <Process 51822>
Hello World
```

Next, let's look at how we can use subprocesses

32.3 How To Use Subprocesses

Once we have created a subprocess, we can make use of it in our asyncio program.

There are specific ways we may need to use the process, such as:

1. Wait for the subprocess to complete.
2. Write data to the subprocess.
3. Read data from the subprocess.

Let's take a closer look at each in turn.

32.3.1 Wait For A Subprocess

We can wait for a subprocess to complete via the `wait()` method.

This is a coroutine that must be awaited.

The caller will be suspended until the subprocess is terminated, normally or otherwise.

If the subprocess is expecting input and is configured to receive input via asyncio with a pipe, then calling `wait()` can cause a deadlock as the caller cannot provide input to the subprocess if it is suspended.

We can explore how to wait for a subprocess to complete.

In the example below, we will execute the `sleep` command and sleep for three seconds.

The caller will then wait for the subprocess to complete before resuming.

The complete example is listed below.

```
# SuperFastPython.com
# example of waiting for a subprocess to finish
import asyncio

# main coroutine
async def main():
    # create as a subprocess
    process = await asyncio.create_subprocess_shell(
        'sleep 3')
    # wait for the subprocess to terminate
    await process.wait()

# start the asyncio event loop
asyncio.run(main())
```

Running the example executes the `sleep` command in a subprocess. This command may not be available on all platforms.

The coroutine then awaits the process to complete, which takes three seconds.

The subprocess closes and the coroutine resumes and terminates the programs.

This highlights how to wait for a subprocess in an asyncio program.

Next, let's look at how we might write data to the subprocess.

32.3.2 Write Data To A Subprocess

We can write data to the subprocess from an asyncio coroutine via the `communicate()` method.

This is a coroutine that must be awaited. Data is provided via the `input` argument as bytes.

Writing data to the subprocess via the `communicate()` method requires that the `stdin` argument in the `create_subprocess_shell()` or `create_subprocess_exec()` functions were set to `PIPE`.

We can explore how to write data to a subprocess in asyncio.

In the example below we will run the `cat` command that outputs the data provided to it. When called without an argument it will echo data from stdin.

We will write a string to the subprocess via the `communicate()` method.

The complete example is listed below.

```
# SuperFastPython.com
# example of writing to a subprocess
import asyncio

# main coroutine
async def main():
    # create a subprocess
    process = await asyncio.create_subprocess_exec(
        'cat', stdin=asyncio.subprocess.PIPE)
    # write data to the subprocess
    _ = await process.communicate(b'Hello World\n')

# start the asyncio event loop
asyncio.run(main())
```

Running the example executes the `cat` command in a subprocess and configures standard input to be redirected from an asyncio pipe. Note that this command may not be available on all platforms.

The command awaits input from stdin.

The coroutine then writes a string with a trailing new line to the subprocess in byte format.

The string is reported and the subprocess terminates.

This highlights how to write data to a subprocess in an asyncio program.

```
Hello World
```

Next, let's look at how to read data from a subprocess in asyncio.

32.3.3 Read Data From A Subprocess

We can read data from a subprocess in asyncio via the `communicate()` method.

Reading data from the subprocess requires that the `stdout` or `stderr` arguments of the `create_subprocess_shell()` or `create_subprocess_exec()` functions was set to `PIPE`.

No argument is provided and the method returns a tuple with input from stdout and stderr. Data is read until an end of file (EOF) character is received.

The `communicate()` method is a coroutine and must be awaited.

If no data can be read, the call will block until the subprocess has terminated.

We can explore how to read data from a subprocess.

In the example below, we execute the `echo` command to echo a string and configure stdout to redirect to an asyncio pipe.

We then read the data from the subprocess and report it.

The complete example is listed below.

```
# SuperFastPython.com
# example of reading from a subprocess
import asyncio

# main coroutine
async def main():
    # create a subprocess
    process = await asyncio.create_subprocess_shell(
        'echo Hello World',
        stdout=asyncio.subprocess.PIPE)
    # read data from the subprocess
    data, _ = await process.communicate()
    # report the data
    print(data)

# start the asyncio event loop
asyncio.run(main())
```

Running the example runs the `echo` command in a subprocess and configures stdout to redirect to a pipe.

The subprocess runs and outputs the string.

The coroutine reads data from the subprocess, keeping the data from stdout and ignoring stderr.

The returned string is then reported, showing that it was read as byte data.

```
b'Hello World\n'
```

This highlights how to read data from a subprocess from asyncio.

Next, let's look at how we might stop the subprocess once we are finished with it.

32.4 How To Stop Subprocesses

Once we are finished with the subprocess in our asyncio program, we can stop it.

Often the subprocess will finish gracefully, after the task is complete, but not always.

Sometimes we need to forcefully terminating the subprocess or even kill it.

Some ways we can stop a subprocess include:

1. Sending a signal to the subprocess.
2. Terminating the subprocess.
3. Killing the subprocess.

Once stopped, terminated or otherwise, we can then retrieve the return code and check if the subprocess completed normally or with an error.

Let's take a closer look at each in turn.

32.4.1 Send A Signal To A Subprocess

We can send a signal to the subprocess via the `send_signal()` method.

The method takes a specific signal type, such as `SIGINT` to interrupt the subprocess or `SIGKILL` to kill the subprocess.

We can explore sending a signal to a subprocess in asyncio.

In the example below, we run the `sleep` command in a subprocess that will sleep for 3 seconds.

The calling coroutine will wait for 1 second, then send a `SIGKILL` signal to the subprocess in order to terminate it immediately.

The complete example is listed below.

```
# SuperFastPython.com
# example of sending a signal to a subprocess
import asyncio
import signal

# main coroutine
async def main():
```

```
    # create a subprocess
    process = await asyncio.create_subprocess_shell(
        'sleep 3')
    # wait a moment
    await asyncio.sleep(1)
    # send a signal to the process
    process.send_signal(signal.SIGKILL)

# start the asyncio event loop
asyncio.run(main())
```

Running the example executes the `sleep` command in a subprocess.

The subprocess proceeded to sleep for 3 seconds.

The coroutine then blocks for 1 second. It then resumes and sends the `SIGKILL` to the subprocess.

This terminates the subprocess immediately, and the program closes.

This highlights how to send a signal to a subprocess from asyncio.

Next, let's look at how to terminate a subprocess from asyncio.

32.4.2 Terminate A Subprocess

We can stop a subprocess via the `terminate()` method.

This highlights how to send a signal to a subprocess from asyncio.

On most platforms, this sends a `SIGTERM` signal to the subprocess and terminates it immediately.

We can explore how to stop a subprocess from asyncio.

The example below starts a subprocess that executes the `sleep` command, then terminates it directly.

This achieves the same result as sending the `SIGTERM` to the subprocess manually, as we did in the previous example.

The complete example is listed below.

```
# SuperFastPython.com
# example of terminating a subprocess
import asyncio

# main coroutine
async def main():
    # create a subprocess
```

```
    process = await asyncio.create_subprocess_shell(
        'sleep 3')
    # wait a moment
    await asyncio.sleep(1)
    # terminate the subprocess
    process.terminate()

# start the asyncio event loop
asyncio.run(main())
```

Running the example executes the `sleep` command in a subprocess.

The calling coroutine then sleeps a moment to allow the subprocess to run.

It then calls the `terminate()` method. This stops the subprocess immediately, and the program exits.

This highlights how to terminate a subprocess from asyncio.

Next, we will look at how to kill a subprocess from an asyncio program.

32.4.3 Kill A Subprocess

We can kill a subprocess via the `kill()` method.

On most platforms, this will send the `SIGKILL` signal to the subprocess in order to stop it immediately.

Unlike the `terminate()` method that sends the `SIGTERM` signal, the `SIGKILL` signal cannot be handled by the subprocess. This means it is assured to stop the subprocess.

We can explore how to kill a subprocess.

In the example below, we will update the previous example that executes the `sleep` command in a subprocess.

Rather than terminating it after a short interval, the caller will kill it directly.

It has much the same effect in this case.

The complete example is listed below.

```
# SuperFastPython.com
# example of killing a subprocess
import asyncio

# main coroutine
async def main():
    # create a subprocess
    process = await asyncio.create_subprocess_shell(
```

```
        'sleep 3')
    # wait a moment
    await asyncio.sleep(1)
    # kill the subprocess
    process.kill()

# start the asyncio event loop
asyncio.run(main())
```

Running the example executes the `sleep` command in a subprocess.

The calling coroutine then sleeps a moment to allow the subprocess to run.

It then calls the `kill()` method. This stops the subprocess immediately, and the program exits.

This highlights how to kill a subprocess from asyncio.

Next, let's look at how to access the return code for a subprocess.

32.4.4 Get A Subprocess Return Code

We can retrieve the return code from a subprocess in asyncio via the `returncode` attribute.

The return code is only available after the process has terminated, otherwise, the value is `None`.

A return code of zero (0) indicates a successful exit. Any other value indicates an unsuccessful exit.

We can explore how to access the return code of a subprocess in asyncio.

The example below executes the `sleep` command in a subprocess. The calling coroutine then waits for the subprocess to terminate, then reports the exit code.

The complete example is listed below.

```
# SuperFastPython.com
# example of getting the return code of a subprocess
import asyncio

# main coroutine
async def main():
    # create a subprocess
    process = await asyncio.create_subprocess_shell(
        'sleep 1')
    # wait for the process to terminate
    await process.wait()
    # get the return code from the process
```

```
    print(process.returncode)

# start the asyncio event loop
asyncio.run(main())
```

Running the example executes the `sleep` command in a subprocess.

The coroutine then suspends and awaits the subprocess to terminate.

Once terminated, the coroutine then reports the return code for the subprocess.

A value of zero 0 is reported, showing a successful exit of the subprocess, as we might expect.

This highlights how to retrieve the return code for a subprocess in asyncio.

```
0
```

32.5 Takeaways

You now know how to use the run commands in subprocesses with asyncio programs.

Specifically, you know:

- What are subprocesses and how we can run new commands in subprocesses from asyncio.
- How to perform non-blocking read and write communication with programs in subprocesses.
- How to send signals, retrieve data, and terminate programs run in subprocesses.

32.5.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

- `signal` – Set handlers for asynchronous events.
  https://docs.python.org/3/library/signal.html
- Asyncio Subprocesses.
  https://docs.python.org/3/library/asyncio-subprocess.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

32.5.2 Next

In the next tutorial, we will explore how to perform non-blocking reads and writes with streams in asyncio.

Chapter 33

Client Sockets

A major benefit of asyncio is the ability to perform non-blocking reads and writes with sockets.

This can be used with network programming applications, such as client programs that read and write with remote servers via APIs, with programs that connect to servers using common protocols, and by developing server applications.

Central to socket programming in asyncio are streams for reading and writing data. A solid knowledge of streams is required in developing client-side asyncio programs, but also servers that we will explore in a later chapter.

In this tutorial, you will discover how to use asyncio streams.

After completing this tutorial, you will know:

- What are asyncio streams and how to open a TCP socket connection.
- How to read and write data using asyncio streams and ensure write buffers are emptied.
- How to close an open connection and ensure the connection is closed before moving on.

Let's get started.

33.1 Asyncio Streams

Asyncio provides non-blocking I/O socket programming.

This is provided via streams.

Sockets can be opened that provide access to a stream writer and a stream reader.

Data can then be written and read from the stream using coroutines, suspending when appropriate.

Once finished, the socket can be closed.

The asyncio streams capability is low-level meaning that any protocols required must be implemented manually, or used via a third-party library.

This might include common web protocols, such as:

- HTTP or HTTPS for interacting with web servers
- SMTP for interacting with email servers
- FTP for interacting with file servers.

The streams can also be used to create a server to handle requests using a standard protocol, or to develop our own application-specific protocol.

Now that we know what asyncio streams are, let's look at how to use them.

33.2 How To Open A Socket Connection

An asyncio TCP client socket connection can be opened using the `asyncio.open_connection()` function.

This is a coroutine that must be awaited and will return once the socket connection is open.

The function returns a `StreamReader` and `StreamWriter` object for interacting with the socket.

For example:

```
...
# open a connection
reader, writer = await asyncio.open_connection(...)
```

The `asyncio.open_connection()` function takes many arguments in order to configure the socket connection.

The two required arguments are the host and the port.

The host is a string that specifies the server to connect to, such as a domain name or an IP address.

The port is the socket port number, such as 80 for HTTP servers, 443 for HTTPS servers, 23 for SMTP and so on.

For example:

```
...
# open a connection to an http server
reader, writer = await asyncio.open_connection(
    'www.google.com', 80)
```

Encrypted socket connections are supported over the SSL protocol.

The most common example might be HTTPS which is replacing HTTP.

This can be achieved by setting the `ssl` argument to `True`.

For example:

```
...
# open a connection to an https server
reader, writer = await asyncio.open_connection(
    'www.google.com', 443, ssl=True)
```

33.3 How To Write Data To A Stream

We can write data to the socket using an `asyncio.StreamWriter`.

Data is written as bytes.

Byte data can be written to the socket using the `write()` method.

For example:

```
...
# write byte data
writer.write(byte_data)
```

Alternatively, multiple lines of byte data organized into a list or iterable can be written using the `writelines()` method.

For example:

```
...
# write lines of byte data
writer.writelines(byte_lines)
```

Neither method for writing data blocks or suspends the calling coroutine.

After writing byte data it is a good idea to empty the write buffer for the socket via the `drain()` method.

This is a coroutine and will suspend the caller until the bytes have been transmitted and the socket is ready.

For example:

```
...
# write byte data
writer.write(byte_data)
# wait for data to be transmitted
await writer.drain()
```

33.4 How To Read Data From A Stream

We can read data from the socket using an `asyncio.StreamReader`.

Data is read in byte format, therefore strings may need to be encoded before being used.

All read methods are coroutines that must be awaited.

An arbitrary number of bytes can be read via the `read()` method, which will read until the end of file (EOF) character is read.

```
...
# read byte data
byte_data = await reader.read()
```

Additionally, the number of bytes to read can be specified via the `n` argument.

This may be helpful if we know the number of bytes expected from the next response.

For example:

```
...
# read byte data
byte_data = await reader.read(n=100)
```

A single line of data can be read using the `readline()` method.

This will return bytes until a new line character `'\n'` is encountered, or an EOF character.

This is helpful when reading standard protocols that operate with lines of text.

```
...
# read a line data
byte_line = await reader.readline()
```

Additionally, there is a `readexactly()` method to read an exact number of bytes otherwise raise an exception, and a `readuntil()` that will read bytes until a specified character in byte form is read.

Finally, the `asyncio.StreamReader` implements the asynchronous iterator interface that can be used via the `async for` expression. This can be used to read lines of byte data.

For example:

```
...
# read lines of byte data
async for line_bytes in reader:
    # decode bytes into a string
    line_string = line_bytes.decode()
    # remove white space
    line_data = line_string.strip()
    # ...
```

33.5 How To Close A Socket Connection

The socket can be closed via the `asyncio.StreamWriter`.

The `close()` method can be called which will close the socket.

This method does not block.

For example:

```
...
# close the socket
writer.close()
```

Although the `close()` method does not block, we can wait for the socket to close completely before continuing on.

This can be achieved via the `wait_closed()` method.

This is a coroutine that can be awaited.

For example:

```
...
# close the socket
writer.close()
# wait for the socket to close
await writer.wait_closed()
```

We can check if the socket has been closed or is in the process of being closed via the `is_closing()` method.

For example:

```
...
# check if the socket is closed or closing
if writer.is_closing():
    # ...
```

33.6 Takeaways

You now know how to use streams in asyncio.

Specifically, you know:

- What are asyncio streams and how to open a TCP socket connection.
- How to read and write data using asyncio streams and ensure write buffers are emptied.
- How to close an open connection and ensure the connection is closed before moving on.

33.6.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

33.6.1.1 References

- Network socket, Wikipedia.
  https://en.wikipedia.org/wiki/Network_socket

33.6.1.2 APIs

- Streams.
  https://docs.python.org/3/library/asyncio-stream.html
- Asyncio Event Loop.
  https://docs.python.org/3/library/asyncio-eventloop.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

33.6.2 Next

In the next tutorial, we will explore how to use what we have learned about streams to develop an asyncio program to check the status of many webpages asynchronously.

Chapter 34

Check Website Status

We can query the HTTP status of websites using asyncio by opening a stream sending HTTP requests, and then reading the response.

Once developed, we can use this mechanism to query the status of many websites concurrently, and even report the results dynamically.

In this tutorial, you will discover how to develop a concurrent client program to check the HTTP status of multiple web pages.

After completing this tutorial, you will know:

- How to manually manage the life-cycle of opening, sending, receiving, and closing an HTTP connections in asyncio.
- How to sequentially and then concurrently query the status of multiple websites.
- How to concurrently query and dynamically report responses from servers as they are received with asyncio.

Let's get started.

34.1 How To Check HTTP Status With Asyncio

The `asyncio` module provides support for opening socket connections and reading and writing data via streams without blocking.

We can use this capability to check the status of web pages.

This involves perhaps four steps, they are:

1. Open a socket connection.
2. Write a request.
3. Read a response.
4. Close the socket connection.

Let's take a closer look at each part in turn.

34.1.1 Open An HTTP Connection

A connection can be opened in asyncio using the `open_connection()` function.

Among many arguments, the function takes the string host name and integer port number

This is a coroutine that must be awaited and returns a `StreamReader` and a `StreamWriter` for reading and writing with the socket.

As a reminder of what we saw in the previous chapter, this can be used to open an HTTP connection on port 80.

For example:

```
...
# open a socket connection
reader, writer = await asyncio.open_connection(
    'www.google.com', 80)
```

We can also open an SSL connection using the `ssl=True` argument. This can be used to open an HTTPS connection on port 443.

For example:

```
...
# open a socket connection
reader, writer = await asyncio.open_connection(
    'www.google.com', 443)
```

34.1.2 Write An HTTP Request

Once open, we can write a query to the `StreamWriter` to make an HTTP request of the server.

For example, an HTTP version 1.1 request is in plain text. We can request the file path `'/'`, which may look as follows:

```
GET / HTTP/1.1
Host: www.google.com
```

Importantly, there must be a carriage return and a line feed (`\r\n`) at the end of each line, and an empty line at the end.

As Python strings this may look as follows:

```
'GET / HTTP/1.1\r\n' + \
    'Host: www.google.com\r\n' + \
    '\r\n'
```

This string must be encoded as bytes before being written to the `StreamWriter`.

This can be achieved using the `encode()` method on the string itself.

The default `'utf-8'` encoding may be sufficient.

For example:

```
...
# encode string as bytes
byte_data = string.encode()
```

The bytes can then be written to the socket via the `StreamWriter` via the `write()` method.

For example:

```
...
# write query to socket
writer.write(byte_data)
```

After writing the request, it is a good idea to wait for the byte data to be sent, the buffer emptied, and for the socket to be ready.

This can be achieved by the `drain()` method.

This is a coroutine that must be awaited.

For example:

```
...
# wait for the socket to be ready.
await writer.drain()
```

34.1.3 Read An HTTP Response

Once the HTTP request has been made, we can read the response.

This can be achieved via the `StreamReader` for the socket.

The response can be read using the `read()` method which will read a chunk of bytes, or the `readline()` method which will read one line of bytes.

We might prefer the `readline()` method because we are using the text-based HTTP protocol which sends HTML data one line at a time.

The `readline()` method is a coroutine and must be awaited.

For example:

```
...
# read one line of response
line_bytes = await reader.readline()
```

HTTP 1.1 responses are composed of two parts, a header separated by an empty line, then the body terminating with an empty line.

The header has information about whether the request was successful and what type of file will be sent, and the body contains the content of the file, such as an HTML webpage.

The first line of the HTTP header contains the HTTP status for the requested page on the server.

Each line must be decoded from bytes into a string.

This can be achieved using the `decode()` method on the byte data. Again, the default encoding is `'utf_8'` which should be sufficient for our simple use case.

For example:

```
...
# decode bytes into a string
line_data = line_bytes.decode()
```

34.1.4 Close An HTTP Connection

We can close the socket connection by closing the `StreamWriter`.

This can be achieved by calling the `close()` method.

For example:

```
...
# close the connection
writer.close()
```

This does not block and may not close the socket immediately. We can choose to wait for the socket connection to close if we desire, via the `wait_closed()` method. This may not be needed in this use case.

Now that we know how to make HTTP requests and read responses using asyncio, let's look at some worked examples of checking web page statuses.

34.2 Example Of Checking HTTP Status Sequentially

We can develop an example to check the HTTP status for multiple websites using asyncio.

In this example, we will first develop a coroutine that will check the status of a given URL. We will then call this coroutine once for each of the top 10 websites.

Firstly, we can define a coroutine that will take a URL string and return the HTTP status.

```
# get the HTTP/S status of a webpage
async def get_status(url):
    # ...
```

The URL must be parsed into its constituent components.

We require the hostname and file path when making the HTTP request. We also need to know the URL scheme (HTTP or HTTPS) in order to determine whether SSL is required nor not.

This can be achieved using the `urllib.parse.urlsplit()` function that takes a URL string and returns a named tuple of all the URL elements.

```
...
# split the url into components
url_parsed = urlsplit(url)
```

We can then open the HTTP connection based on the URL scheme and use the URL hostname.

```
...
# open the connection
if url_parsed.scheme == 'https':
    reader, writer = await asyncio.open_connection(
        url_parsed.hostname, 443, ssl=True)
else:
    reader, writer = await asyncio.open_connection(
        url_parsed.hostname, 80)
```

Next, we can create the HTTP GET request using the hostname and file path and write the encoded bytes to the socket using the `StreamWriter`.

```
...
# send GET request
query = f'GET {url_parsed.path} HTTP/1.1\r\n' + \
    f'Host: {url_parsed.hostname}\r\n\r\n'
# write query to socket
writer.write(query.encode())
# wait for the bytes to be written to the socket
await writer.drain()
```

Next, we can read the HTTP response.

We only require the first line of the response that contains the HTTP status.

```
...
# read the single line response
response = await reader.readline()
```

The connection can then be closed.

```
...
# close the connection
writer.close()
```

Finally, we can decode the bytes read from the server, remove trailing white space, and return the HTTP status string.

```
...
# decode and strip white space
status = response.decode().strip()
# return the response
```

```
    return status
```

Tying this together, the complete `get_status()` coroutine is listed below.

It does not have any error handling, such as the case where the host cannot be reached or is slow to respond.

These additions would make a nice extension, if you're interested.

```
# get the HTTP/S status of a webpage
async def get_status(url):
    # split the url into components
    url_parsed = urlsplit(url)
    # open the connection
    if url_parsed.scheme == 'https':
        reader, writer = await asyncio.open_connection(
            url_parsed.hostname, 443, ssl=True)
    else:
        reader, writer = await asyncio.open_connection(
            url_parsed.hostname, 80)
    # send GET request
    query = f'GET {url_parsed.path} HTTP/1.1\r\n' + \
        f'Host: {url_parsed.hostname}\r\n\r\n'
    # write query to socket
    writer.write(query.encode())
    # wait for the bytes to be written to the socket
    await writer.drain()
    # read the single line response
    response = await reader.readline()
    # close the connection
    writer.close()
    # decode and strip white space
    status = response.decode().strip()
    # return the response
    return status
```

Next, we can call the `get_status()` coroutine for multiple web pages or websites we want to check.

In this case, we will define a list of the top 10 web pages in the world at the time of writing, as described by Wikipedia's *List of most-visited websites*.

```
...
# list of top 10 websites to check
sites = ['https://www.google.com/',
    'https://www.youtube.com/',
    'https://www.facebook.com/',
    'https://twitter.com/',
```

```
    'https://www.instagram.com/',
    'https://www.baidu.com/',
    'https://www.wikipedia.org/',
    'https://yandex.ru/',
    'https://yahoo.com/',
    'https://www.whatsapp.com/'
    ]
```

We can then query each, in turn, using our `get_status()` coroutine.

In this case, we will do so sequentially in a loop, and report the status of each in turn.

```
...
# check the status of all websites
for url in sites:
    # get the status for the url
    status = await get_status(url)
    # report the url and its status
    print(f'{url:30}:\t{status}')
```

We can do better than sequential when using asyncio, but this provides a good starting point that we can improve upon later.

Tying this together, the `main()` coroutine queries the status of the top 10 websites.

```
# main coroutine
async def main():
    # list of top 10 websites to check
    sites = ['https://www.google.com/',
        'https://www.youtube.com/',
        'https://www.facebook.com/',
        'https://twitter.com/',
        'https://www.instagram.com/',
        'https://www.baidu.com/',
        'https://www.wikipedia.org/',
        'https://yandex.ru/',
        'https://yahoo.com/',
        'https://www.whatsapp.com/'
        ]
    # check the status of all websites
    for url in sites:
        # get the status for the url
        status = await get_status(url)
        # report the url and its status
        print(f'{url:30}:\t{status}')
```

Finally, we can create the `main()` coroutine and use it as the entry point to the asyncio

program.

We will wrap this in code to record and report the overall execution time via the `time.perf_counter()` function, preferred for reliable benchmarking in Python.

```
...
# record start time
time_start = perf_counter()
# start the asyncio event loop
asyncio.run(main())
# calculate duration
time_duration = perf_counter() - time_start
# report duration
print(f'Took {time_duration:.3f} seconds')
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# check the status of many webpages
import asyncio
from urllib.parse import urlsplit
from time import perf_counter

# get the HTTP/S status of a webpage
async def get_status(url):
    # split the url into components
    url_parsed = urlsplit(url)
    # open the connection
    if url_parsed.scheme == 'https':
        reader, writer = await asyncio.open_connection(
            url_parsed.hostname, 443, ssl=True)
    else:
        reader, writer = await asyncio.open_connection(
            url_parsed.hostname, 80)
    # send GET request
    query = f'GET {url_parsed.path} HTTP/1.1\r\n' + \
        f'Host: {url_parsed.hostname}\r\n\r\n'
    # write query to socket
    writer.write(query.encode())
    # wait for the bytes to be written to the socket
    await writer.drain()
    # read the single line response
    response = await reader.readline()
    # close the connection
    writer.close()
    # decode and strip white space
    status = response.decode().strip()
```

```
    # return the response
    return status

# main coroutine
async def main():
    # list of top 10 websites to check
    sites = ['https://www.google.com/',
        'https://www.youtube.com/',
        'https://www.facebook.com/',
        'https://twitter.com/',
        'https://www.instagram.com/',
        'https://www.baidu.com/',
        'https://www.wikipedia.org/',
        'https://yandex.ru/',
        'https://yahoo.com/',
        'https://www.whatsapp.com/'
        ]
    # check the status of all websites
    for url in sites:
        # get the status for the url
        status = await get_status(url)
        # report the url and its status
        print(f'{url:30}:\t{status}')

# record start time
time_start = perf_counter()
# start the asyncio event loop
asyncio.run(main())
# calculate duration
time_duration = perf_counter() - time_start
# report duration
print(f'Took {time_duration:.3f} seconds')
```

Running the example first creates the `main()` coroutine and uses it as the entry point into the program.

The `main()` coroutine runs, defining a list of the top 10 websites.

The list of websites is then traversed sequentially. The `main()` coroutine suspends and calls the `get_status()` coroutine to query the status of one website.

The `get_status()` coroutine runs, parses the URL, and opens a connection. It constructs an HTTP GET query and writes it to the host. A response is read, decoded, and returned.

The `main()` coroutine resumes and reports the HTTP status of the URL.

This is repeated for each URL in the list.

The program takes about 5.520 seconds to complete, or about half a second per URL on average.

This highlights how we can use asyncio to query the HTTP status of multiple webpages.

Nevertheless, it does not take full advantage of the asyncio to execute tasks concurrently.

```
https://www.google.com/       : HTTP/1.1 200 OK
https://www.youtube.com/      : HTTP/1.1 200 OK
https://www.facebook.com/     : HTTP/1.1 302 Found
https://twitter.com/          : HTTP/1.1 200 OK
https://www.instagram.com/    : HTTP/1.1 200 OK
https://www.baidu.com/        : HTTP/1.1 200 OK
https://www.wikipedia.org/    : HTTP/1.1 200 OK
https://yandex.ru/            : HTTP/1.1 302 Moved
https://yahoo.com/            : HTTP/1.1 301 Moved
https://www.whatsapp.com/     : HTTP/1.1 302 Found
Took 5.520 seconds
```

Next, let's look at how we might update the example to execute the coroutines concurrently.

34.3 Example Of Checking Website Status Concurrently

A benefit of asyncio is that we can execute many coroutines concurrently.

There are a number of ways that this can be achieved.

We will explore two examples in this section.

1. Issues independent tasks and waits.
2. Gather the coroutines.

Let's dive in.

34.3.1 Example Of Concurrent Tasks With Wait

We can update the above example to query the status of URLs concurrently using asyncio.

One approach is to issue each `get_status()` coroutine as an independent task. We can then wait for all of the issued tasks to complete. Once complete, we can report the status of each URL.

Firstly, we can schedule coroutines as independent tasks using the `asyncio.create_task()` function that returns an `asyncio.Task` object.

We need to associate the URL with each task. This can be achieved using a `dict` where the `asyncio.Task` objects are taken as keys and the URL string is taken as the value. We can then look up the URL for a task later when reporting results.

```
...
# create all task requests
tasks_to_url = {asyncio.create_task(
    get_status(url)):url for url in sites}
```

This allows all tasks to be executed concurrently.

Next, we can wait for all issued tasks to complete.

This can be achieved using the `asyncio.wait()` function. It takes a collection of awaitables, suspends the caller, and by default returns once all tasks are complete.

```
...
# wait for all tasks to complete
_ = await asyncio.wait(tasks_to_url)
```

This allows the scheduled tasks to run.

Once all tasks are complete, the status of each URL can be reported.

We can traverse the `dict` of tasks, retrieve the URL for each, retrieve the status from the task and report the result.

```
...
# report all results
for task in tasks_to_url:
    # get the url
    url = tasks_to_url[task]
    # get the status
    status = task.result()
    # report status
    print(f'{url:30}:\t{status}')
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# check the status of many webpages
import asyncio
from urllib.parse import urlsplit
from time import perf_counter

# get the HTTP/S status of a webpage
async def get_status(url):
    # split the url into components
    url_parsed = urlsplit(url)
    # open the connection
    if url_parsed.scheme == 'https':
        reader, writer = await asyncio.open_connection(
            url_parsed.hostname, 443, ssl=True)
    else:
```

```
        reader, writer = await asyncio.open_connection(
            url_parsed.hostname, 80)
    # send GET request
    query = f'GET {url_parsed.path} HTTP/1.1\r\n' + \
        f'Host: {url_parsed.hostname}\r\n\r\n'
    # write query to socket
    writer.write(query.encode())
    # wait for the bytes to be written to the socket
    await writer.drain()
    # read the single line response
    response = await reader.readline()
    # close the connection
    writer.close()
    # decode and strip white space
    status = response.decode().strip()
    # return the response
    return status

# main coroutine
async def main():
    # list of top 10 websites to check
    sites = ['https://www.google.com/',
        'https://www.youtube.com/',
        'https://www.facebook.com/',
        'https://twitter.com/',
        'https://www.instagram.com/',
        'https://www.baidu.com/',
        'https://www.wikipedia.org/',
        'https://yandex.ru/',
        'https://yahoo.com/',
        'https://www.whatsapp.com/'
        ]
    # create all task requests
    tasks_to_url = {asyncio.create_task(
        get_status(url)):url for url in sites}
    # wait for all tasks to complete
    _ = await asyncio.wait(tasks_to_url)
    # report all results
    for task in tasks_to_url:
        # get the url
        url = tasks_to_url[task]
        # get the status
        status = task.result()
        # report status
```

```
        print(f'{url:30}:\t{status}')

# record start time
time_start = perf_counter()
# start the asyncio event loop
asyncio.run(main())
# calculate duration
time_duration = perf_counter() - time_start
# report duration
print(f'Took {time_duration:.3f} seconds')
```

Running the example executes the `main()` coroutine as before.

In this case, a `get_status()` task is created and scheduled for execution for each URL in the list.

The `dict` is created, mapping each `asyncio.Task` object to its URL string.

The `main()` coroutine then suspends and waits for all issued tasks to complete.

The tasks complete as fast as they are able. The suspend when awaiting the connection to open and when reading a response from the server. This allows cooperative multitasking between the tasks.

Once all tasks are complete, the `main()` coroutine resumes and results are reported.

This highlights how we can query website status concurrently using asyncio by scheduling tasks and waiting for them to be completed.

It is faster than the sequential version above, completing in about 1.088 seconds, a speedup of about 5x.

```
https://www.google.com/       : HTTP/1.1 200 OK
https://www.youtube.com/      : HTTP/1.1 200 OK
https://www.facebook.com/     : HTTP/1.1 302 Found
https://twitter.com/          : HTTP/1.1 200 OK
https://www.instagram.com/    : HTTP/1.1 200 OK
https://www.baidu.com/        : HTTP/1.1 200 OK
https://www.wikipedia.org/    : HTTP/1.1 200 OK
https://yandex.ru/            : HTTP/1.1 302 Moved
https://yahoo.com/            : HTTP/1.1 301 Moved
https://www.whatsapp.com/     : HTTP/1.1 302 Found
Took 1.088 seconds
```

Next, let's look at another way to execute the coroutines concurrently.

34.3.2 Example Of Concurrent Tasks With Gather

We can query the status of websites concurrently in asyncio using the `asyncio.gather()` function.

This function takes one or more coroutines, suspends executing the provided coroutines, and returns the results from each as an iterable. We can then traverse the list of URLs and iterable of return values from the coroutines and report results.

This may be a simpler approach than the above.

First, we can create a list of coroutines, one for each URL to query.

```
...
# create all coroutine requests
coros = [get_status(url) for url in sites]
```

Next, we can execute the coroutines and get the iterable of results using `asyncio.gather()`.

Note that we cannot provide the list of coroutines directly, but instead must unpack the list into separate expressions that are provided as positional arguments to the function.

```
...
# execute all coroutines and wait
results = await asyncio.gather(*coros)
```

This will execute all of the coroutines concurrently and retrieve their results.

We can then traverse the list of URLs and returned status and report each in turn.

```
...
# process all results
for url, status in zip(sites, results):
    # report status
    print(f'{url:30}:\t{status}')
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# check the status of many webpages
import asyncio
from urllib.parse import urlsplit
from time import perf_counter

# get the HTTP/S status of a webpage
async def get_status(url):
    # split the url into components
    url_parsed = urlsplit(url)
    # open the connection
    if url_parsed.scheme == 'https':
        reader, writer = await asyncio.open_connection(
            url_parsed.hostname, 443, ssl=True)
```

```
    else:
        reader, writer = await asyncio.open_connection(
            url_parsed.hostname, 80)
    # send GET request
    query = f'GET {url_parsed.path} HTTP/1.1\r\n' + \
        f'Host: {url_parsed.hostname}\r\n\r\n'
    # write query to socket
    writer.write(query.encode())
    # wait for the bytes to be written to the socket
    await writer.drain()
    # read the single line response
    response = await reader.readline()
    # close the connection
    writer.close()
    # decode and strip white space
    status = response.decode().strip()
    # return the response
    return status

# main coroutine
async def main():
    # list of top 10 websites to check
    sites = ['https://www.google.com/',
        'https://www.youtube.com/',
        'https://www.facebook.com/',
        'https://twitter.com/',
        'https://www.instagram.com/',
        'https://www.baidu.com/',
        'https://www.wikipedia.org/',
        'https://yandex.ru/',
        'https://yahoo.com/',
        'https://www.whatsapp.com/'
        ]
    # create all coroutine requests
    coros = [get_status(url) for url in sites]
    # execute all coroutines and wait
    results = await asyncio.gather(*coros)
    # process all results
    for url, status in zip(sites, results):
        # report status
        print(f'{url:30}:\t{status}')

# record start time
time_start = perf_counter()
```

```
# start the asyncio event loop
asyncio.run(main())
# calculate duration
time_duration = perf_counter() - time_start
# report duration
print(f'Took {time_duration:.3f} seconds')
```

Running the example executes the `main()` coroutine as before.

In this case, a list of coroutines is created in a list comprehension.

The `asyncio.gather()` function is then called, passing the coroutines and suspending the `main()` coroutine until they are all complete.

The coroutines execute, querying each website concurrently and returning their status.

The `main()` coroutine resumes and receives an iterable of status values. This iterable along with the list of URLs is then traversed using the `zip()` built-in function and the statuses are reported.

This highlights a simpler approach to executing the coroutines concurrently and reporting the results after all tasks are completed.

It is also faster than the sequential version above, completing in about 1.125 seconds.

Any speed difference with the above version that uses `asyncio.wait()` is likely due to the natural variability in querying remote servers.

```
https://www.google.com/       : HTTP/1.1 200 OK
https://www.youtube.com/      : HTTP/1.1 200 OK
https://www.facebook.com/     : HTTP/1.1 302 Found
https://twitter.com/          : HTTP/1.1 200 OK
https://www.instagram.com/    : HTTP/1.1 200 OK
https://www.baidu.com/        : HTTP/1.1 200 OK
https://www.wikipedia.org/    : HTTP/1.1 200 OK
https://yandex.ru/            : HTTP/1.1 302 Moved
https://yahoo.com/            : HTTP/1.1 301 Moved
https://www.whatsapp.com/     : HTTP/1.1 302 Found
Took 1.125 seconds
```

Next, let's explore how we might query HTTP statuses concurrently and report results dynamically, as they become available.

34.4 Example Of Reporting Statuses Dynamically

We can execute many tasks concurrently and traverse them in the order that they are completed.

This approach can then be used to report website status dynamically, as the statuses become available.

This can be achieved using the `asyncio.as_completed()` function.

This function takes a collection of iterables as an argument, such as a list of coroutines. It then wraps each iterable in a coroutine for execution and returns an iterable that yields these new coroutines in the order that they are completed.

Although we can use the `as_completed()` method to report HTTP statuses dynamically, we have no easy way to relate the new wrapped coroutines to the URL provided as an argument.

Instead, we can update the `get_status()` coroutine to return the status and the URL together.

First, we can create a list of coroutines, one for each URL using a list comprehension.

```
...
# create all coroutine requests
coros = [get_status(url) for url in sites]
```

We can then call the `asyncio.as_completed()` function with the list of coroutines. This returns a list of awaitables that yield coroutines in the order they are completed.

We can await each in turn to get the return value, in this case, the HTTP status and URL, which can be reported.

```
...
# traverse tasks in completion order
for coro in asyncio.as_completed(coros):
    # get status from task
    status, url = await coro
    # report status
    print(f'{url:30}:\t{status}')
```

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# check the status of many webpages
import asyncio
from urllib.parse import urlsplit
from time import perf_counter

# get the HTTP/S status of a webpage
async def get_status(url):
    # split the url into components
    url_parsed = urlsplit(url)
    # open the connection
    if url_parsed.scheme == 'https':
        reader, writer = await asyncio.open_connection(
```

```
            url_parsed.hostname, 443, ssl=True)
    else:
        reader, writer = await asyncio.open_connection(
            url_parsed.hostname, 80)
    # send GET request
    query = f'GET {url_parsed.path} HTTP/1.1\r\n' + \
        f'Host: {url_parsed.hostname}\r\n\r\n'
    # write query to socket
    writer.write(query.encode())
    # wait for the bytes to be written to the socket
    await writer.drain()
    # read the single line response
    response = await reader.readline()
    # close the connection
    writer.close()
    # decode and strip white space
    status = response.decode().strip()
    # return the response
    return status, url

# main coroutine
async def main():
    # list of top 10 websites to check
    sites = ['https://www.google.com/',
        'https://www.youtube.com/',
        'https://www.facebook.com/',
        'https://twitter.com/',
        'https://www.instagram.com/',
        'https://www.baidu.com/',
        'https://www.wikipedia.org/',
        'https://yandex.ru/',
        'https://yahoo.com/',
        'https://www.whatsapp.com/'
        ]
    # create all coroutine requests
    coros = [get_status(url) for url in sites]
    # traverse tasks in completion order
    for coro in asyncio.as_completed(coros):
        # get status from task
        status, url = await coro
        # report status
        print(f'{url:30}:\t{status}')

# record start time
```

```
time_start = perf_counter()
# start the asyncio event loop
asyncio.run(main())
# calculate duration
time_duration = perf_counter() - time_start
# report duration
print(f'Took {time_duration:.3f} seconds')
```

Running the example executes the `main()` coroutine as before.

In this case, a list of coroutines is created.

This is then provided to the `asyncio.as_completed()` function. This wraps each coroutine in a new coroutine (in case a timeout is needed) and returns an iterable.

We can traverse the iterable using a normal for loop which will automatically await the coroutines and yield them in the order they are completed.

Once yielded, we can retrieve the return values by awaiting them and reporting the website status.

This highlights how we can check HTTP status concurrently with asyncio and report results dynamically as tasks are completed.

Any speed difference with the above versions that use `asyncio.wait()` or `asyncio.gather()` is likely due to the natural variability in querying remote servers.

```
https://www.youtube.com/      : HTTP/1.1 200 OK
https://www.facebook.com/     : HTTP/1.1 302 Found
https://www.google.com/       : HTTP/1.1 200 OK
https://www.whatsapp.com/     : HTTP/1.1 302 Found
https://twitter.com/          : HTTP/1.1 200 OK
https://www.instagram.com/    : HTTP/1.1 200 OK
https://www.wikipedia.org/    : HTTP/1.1 200 OK
https://yahoo.com/            : HTTP/1.1 301 Moved
https://www.baidu.com/        : HTTP/1.1 200 OK
https://yandex.ru/            : HTTP/1.1 302 Moved
Took 1.161 seconds
```

34.5 Takeaways

You now know how to check the status of multiple web pages concurrently with asyncio.

Specifically, you know:

- How to manually manage the life-cycle of opening, sending, receiving and closing a HTTP connections in asyncio.
- How to sequentially and then concurrently query the status of multiple websites.

- How to concurrently query and dynamically report responses from servers as they are received with asyncio.

34.5.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

34.5.1.1 References

- Network socket, Wikipedia.
  https://en.wikipedia.org/wiki/Network_socket
- HTTP, Wikipedia.
  https://en.wikipedia.org/wiki/HTTP
- List of most-visited websites, Wikipedia.
  https://en.wikipedia.org/wiki/List_of_most_visited_websites

34.5.1.2 APIs

- `urllib.parse` – Parse URLs into components.
  https://docs.python.org/3/library/urllib.parse.html
- `codecs` – Codec registry and base classes.
  https://docs.python.org/3/library/codecs.html
- Asyncio Streams.
  https://docs.python.org/3/library/asyncio-stream.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

34.5.2 Next

In the next tutorial, we will explore how to develop socket servers in asyncio programs.

Chapter 35

Server Sockets

We can create an asyncio server to accept and manage client socket connections.

An asyncio server is not created directly, instead, we can use a factory function to configure, create, and start a socket server. We can then use the server or accept client connections forever.

In this tutorial, you will discover how to create and use asyncio socket servers.

After completing this tutorial, you will know:

- How to create socket servers in asyncio.
- How to define the behavior and manage client connections via callback functions.
- How to safely manage the life-cycle of a server, including the handling of unexpected failures.

Let's get started.

35.1 Asyncio Servers

An asyncio server accepts incoming client connections.

It is represented in asyncio Python programs via the `asyncio.Server` class.

We use an `asyncio.Server` in our asyncio programs when we want to accept connections from other programs.

There are two main servers we can create, each supporting different type of client socket connections, they are:

1. TCP Socket Server.
2. UNIX Socket Server.

TCP (Transmission Control Protocol) sockets are a fundamental communication mechanism in computer networks. They provide a reliable, connection-oriented channel for data exchange between applications.

In TCP socket communication, two entities, often referred to as the client and the server, establish a connection before exchanging data. The connection is established through a process known as the three-way handshake, which involves the exchange of control messages to synchronize and establish a reliable connection. Once the connection is established, data can be sent bidirectionally.

UNIX sockets, also known as IPC (Inter-Process Communication) sockets or Unix Domain sockets, provide a communication mechanism between processes on the same Unix-like operating system.

Unlike TCP sockets, UNIX sockets operate locally within a single machine, offering a fast and efficient way for processes to exchange data without the overhead associated with network communication.

UNIX sockets leverage the file system to establish communication channels between processes. They are represented as special files in the file system hierarchy, residing in the file namespace. Processes can communicate through these sockets using standard file I/O operations, such as reading and writing.

Now that we are familiar with the two types of asyncio socket servers, let's look at how we might create them.

35.2 How To Create An Asyncio Server

We do not create an `asyncio.Server` instance directly.

Instead, we can use helper functions to create an asyncio server and return an `asyncio.Server` instance.

These factory functions include:

- `asyncio.start_server()`: to create a TCP socket server.
- `asyncio.start_unix_server()` to create a UNIX socket server.

Creating a server requires that we define a callback handler coroutine that is specified to the factory function that will handle incoming client connections.

The handler must take two arguments, a `asyncio.StreamReader` for reading data from the client and an `asyncio.StreamWriter` for writing data to the client.

A handler coroutine may look as follows:

```
# handler for client connections
async def handler(reader, writer):
    pass
```

We will focus our attention in this chapter on the more common TCP socket server created with the `asyncio.start_server()`, although most of the discussion in this chapter is just applicable for UNIX servers that use a socket file rather than a server address and port.

The `asyncio.start_server()` function takes the name of the handler as an argument, and is typically configured with a default address and port number.

For example:

```
...
# create an asyncio server
server = await asyncio.start_server(
    handler, '127.0.0.1', port=8888)
```

By default, the server will begin listening and accepting client connections immediately.

The example below creates a server and reports the details of the server.

```
# SuperFastPython.com
# example of creating a server
import asyncio

# handler for client connections
async def handler(reader, writer):
    pass

# main coroutine
async def main():
    # create an asyncio server
    server = await asyncio.start_server(
        handler, '127.0.0.1', port=8888)
    # report the details of the server
    print(server)

# start the asyncio event loop
asyncio.run(main())
```

Running the program we can see that an asyncio TCP socket server was created with the specified port number.

```
<Server sockets=(<asyncio.TransportSocket fd=6,
    family=2, type=1, proto=6,
    laddr=('127.0.0.1', 8888)>,)>
```

35.3 How To Start Accepting Client Connections

An `asyncio.Server` will begin accepting client connections by default as soon as it is created.

For example:

```
...
# create an asyncio server
server = await asyncio.start_server(handler, port=8888)
```

Alternatively, we can create an `asyncio.Server` that does not start serving by setting the `start_serving` argument to `False`, then manually start serving when we are ready.

This can be achieved via the `start_serving()` method.

For example:

```
...
# create an asyncio server
server = await asyncio.start_server(
    handler, '127.0.0.1', port=8888,
    start_serving=False)
# start serving
await server.start_serving()
```

We need a way to ensure that the coroutine remains running so that we can continue to accept client connections for the duration of our program.

This can be achieved via the `serve_forever()` method.

This will allow the server to begin accepting client connections, if not already configured to do so, and will never return, allowing the program to accept client connections for as long as the program runs.

For example:

```
...
# create an asyncio server
server = await asyncio.start_server(
    handler, '127.0.0.1', port=8888)
# accept client connections forever
await server.serve_forever()
```

The complete example below demonstrates this.

```
# SuperFastPython.com
# example of creating a server and serving forever
import asyncio

# handler for client connections
async def handler(reader, writer):
    pass

# main coroutine
async def main():
    # create an asyncio server
    server = await asyncio.start_server(
        handler, '127.0.0.1', port=8888)
    # report the details of the server
    print(server)
```

```
    # accept clients forever (kill with control-c)
    await server.serve_forever()

# start the asyncio event loop
asyncio.run(main())
```

Running the program creates the server and reports its details.

The program then begins serving and waiting for client connections forever.

The program must be stopped manually, such as via the Control-C key combination that sends a `SIGINT` signal (signal interrupt) to the process.

```
<Server sockets=(<asyncio.TransportSocket fd=6,
    family=2, type=1, proto=6,
    laddr=('127.0.0.1', 8888)>,)>
```

35.4 How To Check If A Server Is Serving

We can check if an `asyncio.Server` is currently serving or not via the `is_serving()` method.

Nevertheless, a server may not be serving for a number of reasons, such as:

- It was configured to not serve when it was created.
- It has been closed.
- It failed with an error and was closed.
- The `serve_forever()` task was canceled and was closed.

We can explore this with a worked example.

In this case, we cam create an asyncio server that is not serving, checks its status, starts serving, then checks its status again.

```
# SuperFastPython.com
# example of checking if server is serving
import asyncio

# handler for client connections
async def handler(reader, writer):
    pass

# main coroutine
async def main():
    # create an asyncio server
    server = await asyncio.start_server(
        handler, '127.0.0.1', port=8888,
        start_serving=False)
    # report the details of the server
```

```
    print(server)
    # check if it is serving
    print(f'Serving: {server.is_serving()}')
    # start serving
    await server.start_serving()
    # check if it is serving
    print(f'Serving: {server.is_serving()}')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the server and reports its details.

The status is checked and we can see that it is not serving.

The server is then configured to start serving and the status is checked again, confirming that it is indeed now serving.

```
<Server sockets=(<asyncio.TransportSocket fd=6,
    family=2, type=1, proto=6,
    laddr=('127.0.0.1', 8888)>,)>
Serving: False
Serving: True
```

35.5 How To Access A Client Connections

A single `asyncio.Server` may manage many client connections.

There could be hundreds or even thousands of clients connected to a single server.

The `asyncio.Server` instance provides access to the client socket connections if needed.

This is provided via the `sockets` attribute that provides a list of `TransportSocket` instances.

For example:

```
...
# access list of all client sockets
sockets = server.sockets
```

This might be helpful if we need to report the number of currently connected clients.

For example:

```
...
# report current client connections
print(f'There are {len(server.sockets)} clients')
```

35.6 How To Close An Asyncio Server

Once we are finished with the `asyncio.Server` in our program, we can close it.

This can be achieved by calling the `close()` method manually.

For example:

```
...
# close the server
server.close()
```

This will mean that the server is no longer accepting new client connections, although it may still have existing open client connections.

We can then call the `wait_closed()` to suspend the caller until the server is closed completely. This means that all client connections are now closed.

The idiom for manually closing the `asyncio.Server` may look as follows:

```
...
# request that the server close
server.close()
# wait for the server to complete closing
await server.wait_closed()
```

The example below creates a server and starts serving, waits a moment, then closes the server again.

```
# SuperFastPython.com
# example of creating a server and closing it again
import asyncio

# handler for client connections
async def handler(reader, writer):
    pass

# main coroutine
async def main():
    # create an asyncio server
    server = await asyncio.start_server(
        handler, '127.0.0.1', port=8888)
    # report the details of the server
    print(server)
    # wait a moment
    await asyncio.sleep(2)
    # close the server
    server.close()
    # wait for the server to close
    await server.wait_closed()
```

```
    # report the details of the server
    print(server)

# start the asyncio event loop
asyncio.run(main())
```

Running the program first starts a server that accepts client connections.

The program then suspends for two seconds.

Finally, the server is requested to close, then suspends and waits for all client connections to close. There are no client connections so it closes quickly.

Reporting the final status of the server, we can see that it is no longer listening for incoming connections.

```
<Server sockets=(<asyncio.TransportSocket fd=6,
    family=2, type=1, proto=6,
    laddr=('127.0.0.1', 8888)>,)>
<Server sockets=()>
```

We can also close the server automatically.

This can be achieved by the asynchronous context manager interface implemented by the `asyncio.Server` class.

For example:

```
...
# use the server
async with server:
    # ...
    # close automatically
```

This is helpful to ensure the server is closed correctly if some unexpected error occurs within the body of the `async with` block.

We can demonstrate this with a worked example, listed below.

```
# SuperFastPython.com
# example of managing a server via context manager
import asyncio

# handler for client connections
async def handler(reader, writer):
    pass

# main coroutine
async def main():
    # create an asyncio server
```

```
    server = await asyncio.start_server(
        handler, '127.0.0.1', port=8888)
    # report the details of the server
    print(server)
    # start using the server
    async with server:
        # wait a moment
        await asyncio.sleep(2)
        # server is closed automatically
    # report the details of the server
    print(server)
    # check if it is serving
    print(f'Serving: {server.is_serving()}')

# start the asyncio event loop
asyncio.run(main())
```

Running the example first creates the server that starts serving.

It is then used within the asynchronous context manager, blocking for two seconds.

The context manager is exited and the server is closed and is no longer serving.

```
<Server sockets=(<asyncio.TransportSocket fd=6,
    family=2, type=1, proto=6,
    laddr=('127.0.0.1', 8888)>,)>
<Server sockets=()>
Serving: False
```

We can use the asynchronous context manager interface with the `server_forever()` method to ensure that the server is closed correctly if there is an error serving or if the `server_forver()` task is canceled.

For example:

```
...
# use the server
async with server:
    # accept clients forever (kill with control-c)
    await server.serve_forever()
    # close automatically
```

35.7 Takeaways

You now know how to create and use socket servers in asyncio.

Specifically, you know:

- How to create a socket servers in asyncio.
- How to define the behavior and manage client connections via callback functions.
- How to safely manage the life-cycle of a server, including the handling of unexpected failures.

35.7.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

35.7.1.1 References

- Network socket, Wikipedia.
  https://en.wikipedia.org/wiki/Network_socket
- Unix domain socket, Wikipedia.
  https://en.wikipedia.org/wiki/Unix_domain_socket
- Signal (IPC), Wikipedia.
  https://en.wikipedia.org/wiki/Signal_(IPC)

35.7.1.2 APIs

- PEP 3156 – Asynchronous IO Support Rebooted.
  https://peps.python.org/pep-3156/
- Asyncio Streams.
  https://docs.python.org/3/library/asyncio-stream.html
- Asyncio Event Loop.
  https://docs.python.org/3/library/asyncio-eventloop.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

35.7.2 Next

In the next tutorial, we will explore what we have learned about asyncio servers to develop asynchronous client and server chat programs.

Chapter 36

Group Chat Client And Server

We can develop a group chat client and server with asyncio.

A chat room allows multiple clients to connect to the same server and chat together. Each client message is transmitted to all connected clients.

It is a great exercise in network programming and a great way to showcase asynchronous programming with asyncio with non-blocking TCP streams.

In this tutorial, you will discover how to develop a group chat room using asyncio.

After completing this tutorial, you will know:

- What is a multi-user chatroom and how we might develop a chat room client and server in asyncio.
- How to implement a specific chatroom server and client.
- How to test the developed chatroom programs with one and multiple connected clients.

Let's get started.

36.1 Develop Group Chat Client And Server

A group chat program allows multiple clients to connect and communicate via text in real-time.

Each client is able to send a message to the server and all clients are shown the message, allowing a group of clients to chat at the same time.

> The primary use of a chat room is to share information via text with a group of other users. Generally speaking, the ability to converse with multiple people in the same conversation differentiates chat rooms from instant messaging programs, which are more typically designed for one-to-one communication.

– Chat room, Wikipedia.

Developing a chat client is an excellent case study for network programming and asyncio in particular as it requires that both the client program and the server program be event-driven, responding to messages on demand.

Specifically:

- The client program must display messages as they are received from the server and must transmit messages as they are typed in on standard input.
- The server must accept and manage new client connections and must broadcast messages from each client to all clients.

Let's explore how to develop a chat room for group chat using asyncio.

36.2 How To Develop An Asyncio Chatroom

Developing an asyncio chat room involves developing client and server programs.

Before we develop the client and server programs, let's look at how we might decompose each program into subtasks and how we might solve these using what we have learned about asyncio.

Let's take a closer look at the problems to solve when developing a chat server and chat client respectively.

36.2.1 Develop Asyncio Chat Server

An asyncio group chat server has several requirements, they are:

1. Accept connections from new clients.
2. Ask the client for their name and announce it to all other clients.
3. Read messages from each client and broadcast them to all other clients.
4. Determine when a client wants to exit and broadcast their disconnection.

There are several ways that we could implement these aspects.

We will explore perhaps the simplest approach and ignore error handling.

Firstly, we must create the server and accept incoming client connections.

This can be achieved using the `asyncio.start_server()` function and specify the coroutine to call to handle each incoming connection and the hostname and port on which to handle connections.

For example:

```
...
# define the local host
host_address, host_port = '127.0.0.1', 8888
# create the server
server = await asyncio.start_server(
    handle_chat_client, host_address, host_port)
```

We can then use the context manager on the `asyncio.Server` object instance and call the `serve_forever()` method, allowing the server to accept connections until the program is terminated.

```
...
# run the server
async with server:
    # accept connections
    await server.serve_forever()
```

Next, we can define the `handle_chat_client()` coroutine for handling new connections.

This coroutine must take a `StreamReader` and `StreamWriter` as arguments.

Firstly, it must retrieve a username from the client and store it for use when broadcasting all messages from the client.

We can send a message asking for the name. We will read and write each message as a line terminated with a new line character (`'\n'`).

```
...
# write a message
writer.write('Enter your name:\n'.encode())
# wait for buffer to empty
await writer.drain()
```

Next, the user name can be read and used as a key in a global variable `dict` that stores a tuple of the client `StreamReader` and `StreamWriter` against the client name.

```
...
# ask the user for their name
name_bytes = await reader.readline()
# convert name to string
name = name_bytes.decode().strip()
# store the user details
global ALL_USERS
ALL_USERS[name] = (reader, writer)
```

We can then announce that a client with this name has connected.

This can be achieved by defining a coroutine that takes a string message and loops over all clients in the `ALL_USERS` `dict` and shares the message with each in turn.

The `broadcast_message()` coroutine below implements this.

```
# send a message to all connected users
async def broadcast_message(message):
    # report locally
    print(f'Broadcast: {message.strip()}')
    # convert to bytes
```

```
    msg_bytes = message.encode()
    # enumerate all users and broadcast the message
    global ALL_USERS
    for name, (reader, writer) in ALL_USERS.items():
        # write message to this user
        writer.write(msg_bytes)
        # wait for the buffer to empty
        await writer.drain()
```

We can use this coroutine to broadcast the connection of the new client.

```
...
# announce the user
await broadcast_message(f'{name} has connected\n')
```

Next, we can loop forever, reading messages from the client and broadcasting them to all other clients.

```
...
# read messages from the user
while True:
    # read a line of data
    line_bytes = await reader.readline()
    # convert to string
    line = line_bytes.decode().strip()
    # check for exit
    if line == 'QUIT':
        break
    # broadcast message
    await broadcast_message(f'{name}: {line}\n')
```

If the user sends a special message, e.g. `"QUIT"`, the loop is broken and the user is choosing to disconnect.

To disconnect the user, we can close the `StreamWriter` and broadcast a message to all other clients that the user has disconnected.

For example:

```
...
# close the user's connection
writer.close()
await writer.wait_closed()
# remove from the dict of all users
global ALL_USERS
del ALL_USERS[name]
# broadcast the user has left
await broadcast_message(f'{name} has disconnected\n')
```

That covers the main functionality of the server, next, let's look at the client.

36.2.2 Develop Asyncio Chat Client

An asyncio chat client has a number of requirements, they are:

1. Open a connection to the server.
2. Read and report messages from the server.
3. Read and transmit messages from the user.
4. Close the connection to the server.

The first step is to open a connection to the server.

This can be achieved via the `asyncio.open_connection()` function and specifying the hostname and port number. The function then returns a `StreamReader` and a `StreamWriter` for interacting with the server.

```
...
# open a connection to the server
reader, writer = await asyncio.open_connection(
    server_address, server_port)
```

Next, we can loop forever and read messages from the server and report them on standard output.

```
...
# read messages from the server and print to user
while True:
    # read a message
    result_bytes = await reader.readline()
    # decode response
    response = result_bytes.decode()
    # report the response
    print(response.strip())
```

This can be wrapped in a task and executed for the duration of the program. It ensures that all messages from the server, including broadcasts from other clients, are received and reported on demand.

Next, we need to read messages from the user and transmit them to the server.

This may be the user name at the beginning of the session or chat messages during the session.

We can implement this with a second loop that reads lines of text from standard input (stdin) and transmits them to the server.

If the user enters a special command, e.g. `"QUIT"`, the loop is exited.

Reading lines of text from stdin is a blocking activity, which should not be executed in the asyncio event loop.

For example:

```
...
# read from stdin
message = sys.stdin.readline()
```

If executed, it will stop all activity in the program, preventing any messages from being read from the server and printed.

Instead, we must read from standard input and block in a thread and then read the result in the asyncio event loop when it is available.

This can be achieved via the `asyncio.to_thread()` function.

For example:

```
...
# read from stdin
message = await asyncio.to_thread(sys.stdin.readline)
```

Using this, we can then define the loop that reads messages from the user and transmits them to the server.

```
...
# read messages from the user and transmit to server
while True:
    # read from stdin
    message = await asyncio.to_thread(
        sys.stdin.readline)
    # send message to the server
    msg_bytes = message.encode()
    writer.write(msg_bytes)
    await writer.drain()
    # check for stop
    if message.strip() == 'QUIT':
        break
```

Finally, we can disconnect the client from the server.

```
...
# close the connection
writer.close()
await writer.wait_closed()
```

Now that we know how to develop a chat client and server, let's look at a worked example.

36.3 Example Asyncio Chat Server

We can explore how to develop an asyncio group chat server.

This program has 6 main parts, they are:

1. A coroutine to broadcast to all users.
2. A coroutine for managing a new client connection.
3. A coroutine for managing a client disconnection.
4. A coroutine for managing the client connection overall.
5. A coroutine for driving the server.
6. The start of the event loop.

Let's dive in.

36.3.1 Broadcast To All Clients

Firstly, we can define a coroutine that broadcasts a message to all connected clients.

We will maintain a global variable `dict` of user name mapped to a tuple of `StreamReader` and `StreamWriter`. Therefore, to broadcast a message to all clients means traversing this `dict` and writing a message to each client.

The `broadcast_message()` coroutine below implements this, taking the string message as an argument.

```
# send a message to all connected users
async def broadcast_message(message):
    # report locally
    print(f'Broadcast: {message.strip()}')
    # convert to bytes
    msg_bytes = message.encode()
    # enumerate all users and broadcast the message
    global ALL_USERS
    for _, (_, writer) in ALL_USERS.items():
        # write message to this user
        writer.write(msg_bytes)
        # wait for the buffer to empty
        await writer.drain()
```

We could improve this coroutine in a few ways.

For example, we could protect the `ALL_USERS dict` from concurrent updates while transmitting messages, in case a user disconnects while we are broadcasting.

We could also transmit the messages concurrently rather than sequentially through all clients.

The first idea is left as an extension, but we can implement the second suggestion.

This can be achieved by first defining a new coroutine that writes a byte message to a single `StreamWriter`.

The `write_message()` coroutine below implements this.

```
# write a message to a stream writer
async def write_message(writer, msg_bytes):
```

```
    # write message to this user
    writer.write(msg_bytes)
    # wait for the buffer to empty
    await writer.drain()
```

Next, the `broadcast_message()` coroutine can be updated to call the `write_message()` coroutine concurrently.

This can be achieved by creating one `asyncio.Task` for each client in the `ALL_USERS` `dict`, then awaiting all tasks via `asyncio.wait()`.

For example:

```
...
# create a task for each write to client
tasks = [asyncio.create_task(
    write_message(w, msg_bytes)) for _,(_,w) in
        ALL_USERS.items()]
# wait for all writes to complete
_ = await asyncio.wait(tasks)
```

The updated version of the `broadcast_message()` coroutine with this change is listed below.

The benefit is that it will broadcast messages to all clients concurrently, rather than sequentially.

```
# send a message to all connected users
async def broadcast_message(message):
    # report locally
    print(f'Broadcast: {message.strip()}')
    # convert to bytes
    msg_bytes = message.encode()
    # enumerate all users and broadcast the message
    global ALL_USERS
    # create a task for each write to client
    tasks = [asyncio.create_task(
        write_message(w, msg_bytes)) for _,(_,w) in
            ALL_USERS.items()]
    # wait for all writes to complete
    _ = await asyncio.wait(tasks)
```

36.3.2 Connect A New Client

Next, we need to manage the connection of a new client.

This involves first transmitting a welcome message and asking the client for their name. The name is then read and used to update the `ALL_USERS` `dict`, storing the connection details for broadcasting later.

A broadcast message is then reported to all clients that a new client with a given name has arrived and a direct message is then sent to the client mentioning the command to terminate the chat session.

The `connect_user()` coroutine below implements this.

```
# connect a user
async def connect_user(reader, writer):
    # get name message
    writer.write('Asyncio Chat Server\n'.encode())
    writer.write('Enter your name:\n'.encode())
    await writer.drain()
    # ask the user for their name
    name_bytes = await reader.readline()
    # convert name to string
    name = name_bytes.decode().strip()
    # store the user details
    global ALL_USERS
    ALL_USERS[name] = (reader, writer)
    # announce the user
    await broadcast_message(f'{name} has connected\n')
    # welcome message
    welcome = f'Welcome {name}. ' + \
        'Send QUIT to disconnect.\n'
    writer.write(welcome.encode())
    await writer.drain()
    return name
```

36.3.3 Disconnect A Client

Next, we can define a coroutine to handle the disconnection of a client.

This involves closing the `StreamWriter`, and then deleting the user from the `ALL_USERS` dict.

Finally, a broadcast message can be sent to all remaining clients that the client with the given name has disconnected.

The `disconnect_user()` coroutine below implements this.

```
# disconnect a user
async def disconnect_user(name, writer):
    # close the user's connection
    writer.close()
    await writer.wait_closed()
    # remove from the dict of all users
    global ALL_USERS
    del ALL_USERS[name]
    # broadcast the user has left
```

```
    await broadcast_message(
        f'{name} has disconnected\n')
```

36.3.4 Manage A Client

Next, we need to define a coroutine to manage a single client connection.

This includes calling the `connect_user()` coroutine to formally connect the client, read all messages from the client, and then disconnect the client when they are done.

The core of this is a loop that reads messages transmitted from the client and broadcasts them to all other clients.

The `handle_chat_client()` coroutine below implements this.

```
# handle a chat client
async def handle_chat_client(reader, writer):
    print('Client connecting...')
    # connect the user
    name = await connect_user(reader, writer)
    try:
        # read messages from the user
        while True:
            # read a line of data
            line_bytes = await reader.readline()
            # convert to string
            line = line_bytes.decode().strip()
            # check for exit
            if line == 'QUIT':
                break
            # broadcast message
            await broadcast_message(f'{name}: {line}\n')
    finally:
        # disconnect the user
        await disconnect_user(name, writer)
```

A limitation of this implementation is that all reads from the client are blocked until the broadcast to all other clients is done.

This may limit the scalability and responsiveness of the server to rapid messages from a client,

We can improve the design by decoupling the broadcasting of messages from the client management. One approach would be to have the client put the broadcast message on a queue and have another coroutine read messages from the queue and transmit them to all clients.

This is left as a fun exercise extension.

36.3.5 Drive The Server

Next, we can create a coroutine to drive the main server.

This will define the details of the host, such as its port number, create the server and serve forever until the program is terminated.

The `main()` coroutine below implements this.

```
# chat server
async def main():
    # define the local host
    host_address, host_port = '127.0.0.1', 8888
    # create the server
    server = await asyncio.start_server(
        handle_chat_client, host_address, host_port)
    # run the server
    async with server:
        # report message
        print('Chat Server Running\n' +
            'Waiting for chat clients...')
        # accept connections
        await server.serve_forever()
```

36.3.6 Start The Event Loop

Finally, we can define the global variable that holds all client connections and starts the event loop.

```
...
# dict of all current users
ALL_USERS = {}
# start the event loop
asyncio.run(main())
```

36.3.7 Asyncio Chat Server Complete Example

Tying this together, the complete example of the asyncio chat server is listed below.

```
# SuperFastPython.com
# example of a chat server using streams
import asyncio

# write a message to a stream writer
async def write_message(writer, msg_bytes):
    # write message to this user
    writer.write(msg_bytes)
    # wait for the buffer to empty
```

```
    await writer.drain()

# send a message to all connected users
async def broadcast_message(message):
    # report locally
    print(f'Broadcast: {message.strip()}')
    # convert to bytes
    msg_bytes = message.encode()
    # enumerate all users and broadcast the message
    global ALL_USERS
    # create a task for each write to client
    tasks = [asyncio.create_task(
        write_message(w, msg_bytes)) for _,(_,w) in
            ALL_USERS.items()]
    # wait for all writes to complete
    _ = await asyncio.wait(tasks)

# connect a user
async def connect_user(reader, writer):
    # get name message
    writer.write('Asyncio Chat Server\n'.encode())
    writer.write('Enter your name:\n'.encode())
    await writer.drain()
    # ask the user for their name
    name_bytes = await reader.readline()
    # convert name to string
    name = name_bytes.decode().strip()
    # store the user details
    global ALL_USERS
    ALL_USERS[name] = (reader, writer)
    # announce the user
    await broadcast_message(
        f'{name} has connected\n')
    # welcome message
    welcome = f'Welcome {name}. ' + \
        'Send QUIT to disconnect.\n'
    writer.write(welcome.encode())
    await writer.drain()
    return name

# disconnect a user
async def disconnect_user(name, writer):
    # close the user's connection
    writer.close()
```

```
    await writer.wait_closed()
    # remove from the dict of all users
    global ALL_USERS
    del ALL_USERS[name]
    # broadcast the user has left
    await broadcast_message(
        f'{name} has disconnected\n')

# handle a chat client
async def handle_chat_client(reader, writer):
    print('Client connecting...')
    # connect the user
    name = await connect_user(reader, writer)
    try:
        # read messages from the user
        while True:
            # read a line of data
            line_bytes = await reader.readline()
            # convert to string
            line = line_bytes.decode().strip()
            # check for exit
            if line == 'QUIT':
                break
            # broadcast message
            await broadcast_message(f'{name}: {line}\n')
    finally:
        # disconnect the user
        await disconnect_user(name, writer)

# chat server
async def main():
    # define the local host
    host_address, host_port = '127.0.0.1', 8888
    # create the server
    server = await asyncio.start_server(
        handle_chat_client, host_address, host_port)
    # run the server
    async with server:
        # report message
        print('Chat Server Running\n' +
              'Waiting for chat clients...')
        # accept connections
        await server.serve_forever()
```

```
# dict of all current users
ALL_USERS = {}
# start the asyncio event loop
asyncio.run(main())
```

Next, let's explore an example of a chat client.

36.4 Example Asyncio Chat Client

We can explore how to develop an asyncio group chat client.

This is simpler than the server we developed in the previous section.

This program has 3 main parts, they are:

1. A coroutine to read messages from the user and transmit to the server.
2. A coroutine to read messages from the server and report to the user.
3. A main coroutine to drive the connection to the server.

Let's dive in.

36.4.1 Read And Transmit User Messages

Firstly, we can develop a coroutine that reads messages from standard input and transmits them to the server.

Reads from standard input are blocking, therefore we must perform them in a separate thread via the `asyncio.to_thread()` function.

This coroutine loops until the user enters a message `"QUIT"`, which then exits the loop.

The `write_messages()` coroutine below implements this.

```
# send message to server
async def write_messages(writer):
    # read messages from the user and transmit to server
    while True:
        # read from stdin
        message = await asyncio.to_thread(
            sys.stdin.readline)
        # encode the string message to bytes
        msg_bytes = message.encode()
        # transmit the message to the server
        writer.write(msg_bytes)
        # wait for the buffer to be empty
        await writer.drain()
        # check if the user wants to quit the program
        if message.strip() == 'QUIT':
```

```
            # exit the loop
            break
    # report that the program is terminating
    print('Quitting...')
```

36.4.2 Read And Report Server Messages

Next, we need to develop a coroutine that will read messages from the server and report them to the user.

This task operates in a loop, forever reading messages and printing them to standard output. This can be run as a background task in our client application.

The `read_messages()` coroutine below implements this.

```
# read messages from the server
async def read_messages(reader):
    # read messages from the server and print to user
    while True:
        # read a message from the server
        result_bytes = await reader.readline()
        # decode response from bytes to a string
        response = result_bytes.decode()
        # report the response to the user
        print(response.strip())
```

36.4.3 Drive Connection With Server

Next, we need to drive the connection with the server.

This involves first opening a connection to the server on the required host and port number.

Next, we can create the `read_messages()` coroutine and schedule it as a background task, forever reading and reporting messages from the server.

We can then create and await the `write_messages()` coroutine that will read input from the user and transmit it to the server. This coroutine will return when the user decides to quit.

The `read_messages()` task can then be canceled and we can then close the connection to the server.

The `main()` coroutine below implements this.

```
# echo client
async def main():
    # define the server details
    host, port = '127.0.0.1', 8888
    # report progress to the user
```

```
    print(f'Connecting to {host}:{port}...')
    # open a connection to the server
    reader, writer = await asyncio.open_connection(
        server_address, server_port)
    # report progress to the user
    print('Connected.')
    # read and report messages from the server
    read_task = asyncio.create_task(
        read_messages(reader))
    # write messages to the server
    await write_messages(writer)
    # cancel the read messages task
    read_task.cancel()
    # report progress to the user
    print('Disconnecting from server...')
    # close the stream writer
    writer.close()
    # wait for the tcp connection to close
    await writer.wait_closed()
    # report progress to the user
    print('Done.')
```

Finally, we can start the event loop.

```
...
# run the event loop
asyncio.run(main())
```

36.4.4 Asyncio Chat Client Complete Example

Tying this together, the complete asyncio chat client example is listed below.

```
# SuperFastPython.com
# example of a chat client using streams
import sys
import asyncio

# send message to server
async def write_messages(writer):
    # read messages from the user and transmit to server
    while True:
        # read from stdin
        message = await asyncio.to_thread(
            sys.stdin.readline)
        # encode the string message to bytes
        msg_bytes = message.encode()
```

```
        # transmit the message to the server
        writer.write(msg_bytes)
        # wait for the buffer to be empty
        await writer.drain()
        # check if the user wants to quit the program
        if message.strip() == 'QUIT':
            # exit the loop
            break
    # report that the program is terminating
    print('Quitting...')

# read messages from the server
async def read_messages(reader):
    # read messages from the server and print to user
    while True:
        # read a message from the server
        result_bytes = await reader.readline()
        # decode response from bytes to a string
        response = result_bytes.decode()
        # report the response to the user
        print(response.strip())

# echo client
async def main():
    # define the server details
    host, port = '127.0.0.1', 8888
    # report progress to the user
    print(f'Connecting to {host}:{port}...')
    # open a connection to the server
    reader, writer = await asyncio.open_connection(
        server_address, server_port)
    # report progress to the user
    print('Connected.')
    # read and report messages from the server
    read_task = asyncio.create_task(
        read_messages(reader))
    # write messages to the server
    await write_messages(writer)
    # cancel the read messages task
    read_task.cancel()
    # report progress to the user
    print('Disconnecting from server...')
    # close the stream writer
    writer.close()
```

```
    # wait for the tcp connection to close
    await writer.wait_closed()
    # report progress to the user
    print('Done.')

# start the asyncio event loop
asyncio.run(main())
```

Next, let's explore how to use our chat server and client.

36.5 Example Chat Session

Now that we have developed our chat server and client, let's look at how we can use them with an example chat session.

Each program must be run as a separate program and the client programs require user input.

I recommend saving each program as a separate file and running them on the command line (terminal or prompt).

Firstly, we can run the server.

Save the chat server to a file like `server.py` and then run it on the command line to start the server.

For example:

```
python server.py
```

Next, save the chat client program in a file like `client.py` and then run it on the command line to start the client for example:

```
python client.py
```

The client must then enter their name and can begin chatting.

For example:

```
Connecting to 127.0.0.1:8888...
Connected.
Asyncio Chat Server
Enter your name:
Jason
Jason has connected
Welcome Jason. Send QUIT to disconnect.
Hello there!
Jason: Hello there!
```

On the server, we can see the client connect and each broadcast message being sent.

```
Chat Server Running
Waiting for chat clients...
Client connecting...
Broadcast: Jason has connected
Broadcast: Jason: Hello there!
```

Next, open another window and start a second chat client.

For example:

```
python client.py
```

Enter a different name and begin chatting.

For example:

```
Connecting to 127.0.0.1:8888...
Connected.
Asyncio Chat Server
Enter your name:
Tom
Tom has connected
Welcome Tom. Send QUIT to disconnect.
Hi there, this is very cool!
Tom: Hi there, this is very cool!
```

Now check the first chat client. You will see that the second client was announced and their chat messages were broadcast.

```
...
Tom has connected
Tom: Hi there, this is very cool!
```

Similarly, the server sees the second client.

```
...
Client connecting...
Broadcast: Tom has connected
Broadcast: Tom: Hi there, this is very cool!
```

Carry on and test the clients.

Each client can then terminate their session by sending the message: `QUIT`

For example:

```
...
QUIT
Quitting...
Disconnecting from server...
Done.
```

The first client will see the second client disconnect.

For example:

```
...
Tom has disconnected
```

The first client can then disconnect in the same manner.

For example:

```
...
QUIT
Quitting...
Disconnecting from server...
Done.
```

The server will see and report both client disconnections.

```
...
Broadcast: Tom has disconnected
Broadcast: Jason has disconnected
```

Finally, the server process can be forcefully terminated with the Control-C key combination.

This highlights how we can use our group chat or chat room program by running separate server and client programs.

36.6 Extensions

This section lists some extensions you may wish to explore.

- **Use Queue in the Server**. The server could put messages from clients on a queue. A broadcast coroutine could loop forever and read messages from the queue and broadcast them to all clients. This separation of concerns may be more efficient and make the server more responsive to rapid messages from a client.
- **Add Coroutine Safety**. Protect all changes to the `ALL_USERS dict` from race conditions via concurrent updates using a mutex lock.
- **Add Error Handling**. Check for and handle common errors, such as server already running, unable to connect to server, client failing to connect properly, client disconnected unexpectedly, etc.

If you explore any of these extensions, reach out and let me know. I'd love to see what you come up with.

36.7 Takeaways

You now know how to develop a group chat room using asyncio.

Specifically, you know:

- What is a multi-user chatroom and how we might develop a chat room client and server in asyncio.
- How to implement a specific chatroom server and client.
- How to test the developed chatroom programs with one and multiple connected clients.

36.7.1 Further Reading

This section provides resources for you to learn more about the topics covered in this chapter.

36.7.1.1 References

- Network socket, Wikipedia.
  https://en.wikipedia.org/wiki/Network_socket
- Chat room, Wikipedia.
  https://en.wikipedia.org/wiki/Chat_room

36.7.1.2 APIs

- `sys` – System-specific parameters and functions.
  https://docs.python.org/3/library/sys.html
- Asyncio Streams.
  https://docs.python.org/3/library/asyncio-stream.html
- Asyncio Event Loop.
  https://docs.python.org/3/library/asyncio-eventloop.html
- Asyncio Runners.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks.
  https://docs.python.org/3/library/asyncio-task.html

36.7.2 Next

This was the final tutorial. Next, we will review how far you have come and resources for further reading.

Conclusions

Chapter 37

Conclusions

37.1 How Far You've Come

Congratulations, you made it to the end.

Let's take a look back and review what you now know.

1. You know how to confidently develop Python programs with the asynchronous programming paradigm, including:

1. How to structure and run an asyncio programs.
2. How to create and execute coroutines.
3. How to use the expressions added to the Python language to implement cooperative multitasking.

2. You know how to confidently create, run, and query asyncio tasks, including:

1. How to create and schedule coroutines as tasks and run them in the foreground or background.
2. How to query task status and assign them meaningful names.
3. How to cancel tasks, handle task cancellation, and check if a task has been canceled.
4. How to retrieve results from tasks and exceptions raised in tasks.
5. How to add and mange done callback functions executed by tasks.
6. How to introspect the current and all running tasks and know the role and importance of the main task.

3. You know how to confidently manage groups of asyncio tasks, including:

1. How to execute multiple tasks concurrently and retrieve their results once they are all done.
2. How to wait for a condition on a group of tasks, such as all done, first done, or first to fail.
3. How to process task results in the order that tasks are completed rather than the order they were issued.

4. How to define a related collection of tasks as a group and operate upon the group, such as cancel all if one fails.
5. How to execute tasks and blocks of asyncio code with timeouts.

4. You know how to confidently implement more complex task management, including:

1. How to shield tasks from cancellation.
2. How to yield control with sleep to allow scheduled tasks to execute.
3. How to wait for a single task to complete with a timeout.
4. How to execute a blocking task without stopping the asyncio event loop.

5. You know how to confidently use asynchronous control flow and data structures, including:

1. How to develop and traverse asynchronous iterators and generators
2. How to develop and use asynchronous context managers.
3. How to communicate between coroutines and tasks using queues.

6. You know how to confidently synchronize the behavior between tasks, including:

1. How to use mutex locks to avoid race conditions and deadlocks, ensuring coroutine safety.
2. How to synchronize behavior with shared events and use the wait/notify pattern with condition variables.
3. How to limit access to shared resources by tasks using semaphores.
4. How to coordinate behavior between tasks using barriers.

7. You know how to confidently communicate with other processes and networked programs efficiently, including:

1. How to create, manage and perform non-blocking reads and writes with subprocesses.
2. How to read and write with non-blocking network sockets.
3. How to develop an asynchronous website status checking application.
4. How to develop and manage asynchronous socket servers
5. How to develop an asynchronous group chat application.

Thank you for letting me help you on your journey into Python concurrency.

Jason Brownlee, Ph.D.
SuperFastPython.com
2023.

37.2 Resources

This section lists some useful additional resources for further reading.

37.2.1 APIs

- `asyncio` API - Asynchronous I/O.
  https://docs.python.org/3/library/asyncio.html
- Asyncio Runners API.
  https://docs.python.org/3/library/asyncio-runner.html
- Asyncio Coroutines and Tasks API.
  https://docs.python.org/3/library/asyncio-task.html
- Asyncio Streams API.
  https://docs.python.org/3/library/asyncio-stream.html
- Asyncio Synchronization Primitives API.
  https://docs.python.org/3/library/asyncio-sync.html
- Asyncio Subprocesses API.
  https://docs.python.org/3/library/asyncio-subprocess.html
- Asyncio Queues API.
  https://docs.python.org/3/library/asyncio-queue.html
- Asyncio Exceptions API.
  https://docs.python.org/3/library/asyncio-exceptions.html
- Asyncio High-level API Index.
  https://docs.python.org/3/library/asyncio-api-index.html
- Developing with asyncio.
  https://docs.python.org/3/library/asyncio-dev.html

37.2.2 Books

- Python Concurrency with asyncio, Matt Fowler, 2022.
  https://amzn.to/3LZvxNn
- Using AsyncIO in Python, Caleb Hattingh, 2020.
  https://amzn.to/3lNp2ml
- asyncio Recipes, Mohamed Mustapha Tahrioui, 2019.
  https://amzn.to/3sz3zoj

37.2.3 Getting More Help

Do you have any questions?

Below provides some great places online where you can ask questions about Python programming and Python concurrency:

- Stack Overview.
  https://stackoverflow.com/
- Python Subreddit.
  https://www.reddit.com/r/python
- LinkedIn Python Developers Community.
  https://www.linkedin.com/groups/25827

- Quora Python (programming language).
  https://www.quora.com/topic/Python-programming-language-1

37.2.4 Contact The Author

You are not alone.

If you ever have any questions about the lessons in this book, please contact me directly:

- Super Fast Python - Contact Page
  https://SuperFastPython.com/contact/

I will do my best to help.

About the Author

Jason Brownlee, Ph.D. helps Python developers bring modern concurrency methods to their projects with hands-on tutorials. Learn more at SuperFastPython.com.

Jason is a software engineer and research scientist with a background in artificial intelligence and high-performance computing. He has authored more than 20 technical books on machine learning and has built, operated, and exited online businesses.