SuperFastPython
making python developers awesome at concurrency (https://superfastpython.com/)

Python Asyncio: The Complete Guide

NOVEMBER 10, 2022 *by* JASON BROWNLEE *in* ASYNCIO (HTTPS://SUPERFASTPYTHON.COM/CATEGORY/ASYNCIO/)

Asyncio allows us to use asynchronous programming with coroutine-based concurrency in Python.

Although asyncio has been available in Python for many years now, it remains one of the most interesting and yet one of the most frustrating areas of Python.

It is just plain hard to get started with asyncio for new developers.

This guide provides a detailed and comprehensive review of asyncio in Python, including how to define, create and run coroutines, what is asynchronous programming, what is non-blocking-io, concurrency primitives used with coroutines, common questions, and best practices.

This is a massive 29,000+ word guide. You may want to bookmark it so you can refer to it as you develop your concurrent programs.

Let’s dive in.

Skip the tutorial. Master asyncio today. Learn how (https://superfastpython.com/paj-incontent)

Table of Contents

What is Asynchronous Programming

Asynchronous programming is a programming paradigm that does not block.

Instead, 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.

Let’s get a handle on asynchronous programming before we dive into asyncio.

Asynchronous Tasks

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

> “*asynchronous: not simultaneous or concurrent in time*
>
> — MERRIAM-WEBSTER DICTIONARY (HTTPS://WWW.MERRIAM-WEBSTER.COM/DICTIONARY/ASYNCHRONOUS)

When programming, asynchronous (https://en.wikipedia.org/wiki/Asynchrony_(computer_programming)) means that the action is requested, although not performed at the time of the request. It is performed later.

> “*Asynchronous*: *Separate execution streams that can run concurrently in any order relative to each other are asynchronous.*
>
> — **PAGE 265, THE ART OF CONCURRENCY (HTTPS://AMZN.TO/3TKCUWX), 2009.**

For example, we can make an asynchronous function call (https://en.wikipedia.org/wiki/Asynchronous_procedure_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.

- **Asynchronous Function Call**: Request that a function is called at some time and in some manner, allowing the caller to resume and perform other activities.

The function call will happen somehow and at some time, in the background, and the program can perform other tasks or respond to other events.

This is key. We don’t have control over how or when the request is handled, only that we would like it handled while the program does other things.

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.

- **Future**: A handle on an asynchronous function call allowing the status of the call to be checked and results to be retrieved.

The combination of the asynchronous function call and future together is often referred to as an asynchronous task. This is because it is more elaborate than a function call, such as allowing the request to be canceled and more.

- **Asynchronous Task**: Used to refer to the aggregate of an asynchronous function call and resulting future.

Asynchronous Programming

Issuing asynchronous tasks and making asynchronous function calls is referred to as asynchronous programming.

> *"So what is asynchronous programming? It means that a particular long-running task can be run in the background separate from the main application. Instead of blocking all other application code waiting for that long-running task to be completed, the system is free to do other work that is not dependent on that task. Then, once the long-running task is completed, we'll be notified that it is done so we can process the result.*
>
> **— PAGE 3, PYTHON CONCURRENCY WITH ASYNCIO (HTTPS://AMZN.TO/3CZ7ZH6), 2022.**

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

Asynchronous programming is primarily used with non-blocking I/O, such as reading and writing from socket connections with other processes or other systems.

> *"In non-blocking mode, when we write bytes to a socket, we can just fire and forget the write or read, and our application can go on to perform other tasks.*
>
> **— PAGE 18, PYTHON CONCURRENCY WITH ASYNCIO (HTTPS://AMZN.TO/3CZ7ZH6), 2022.**

Non-blocking I/O is a way of performing I/O where reads and writes are requested, although performed asynchronously. The caller does not need to wait for the operation to complete before returning.

The read and write operations are performed somehow (e.g. by the underlying operating system or systems built upon it), and the status of the action and/or data is retrieved by the caller later, once available, or when the caller is ready.

- **Non-blocking I/O**: Performing I/O operations via asynchronous requests and responses, rather than waiting for operations to complete.

As such, we can see how non-blocking I/O is related to asynchronous programming. In fact, we use non-blocking I/O via asynchronous programming, or non-blocking I/O is implemented via asynchronous programming.

The combination of non-blocking I/O with asynchronous programming is so common that it is commonly referred to by the shorthand of asynchronous I/O (https://en.wikipedia.org/wiki/Asynchronous_I/O).

- **Asynchronous I/O**: A shorthand that refers to combining asynchronous programming with non-blocking I/O.

Next, let's consider asynchronous programming support in Python.

Asynchronous Programming in Python

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

We can implement asynchronous programming in Python in various ways, although a few are most relevant for Python concurrency.

The first and obvious example is the **asyncio** module (https://docs.python.org/3/library/asyncio.html). This module directly offers an asynchronous programming environment using the async/await syntax and non-blocking I/O with sockets and subprocesses.

> *asyncio is short for asynchronous I/O. It is a Python library that allows us to run code using an asynchronous programming model. This lets us handle multiple I/O operations at once, while still allowing our application to remain responsive.*
>
> **— PAGE 3, PYTHON CONCURRENCY WITH ASYNCIO (HTTPS://AMZN.TO/3CZ7ZH6), 2022.**

It is implemented using coroutines that run in an event loop that itself runs in a single thread.

- **Asyncio**: An asynchronous programming environment provided in Python via the asyncio module.

More broadly, Python offers threads and processes that can execute tasks asynchronously.

For example, one thread can start a second thread to execute a function call and resume other activities. The operating system will schedule and execute the second thread at some time and the first thread may or may not check on the status of the task, manually.

> “*Threads are asynchronous, meaning that they may run at different speeds, and any thread can halt for an unpredictable duration at any time.*
>
> **— PAGE 76, THE ART OF MULTIPROCESSOR PROGRAMMING (HTTPS://AMZN.TO/3CC82J2), 2020.**

More concretely, Python provides executor-based thread pools and process pools in the **ThreadPoolExecutor (https://superfastpython.com/threadpoolexecutor-in-python/)** and **ProcessPoolExeuctor (https://superfastpython.com/processpoolexecutor-in-python/)** classes.

These classes use the same interface and support asynchronous tasks via the **submit()** method that returns a **Future** object.

> “*The concurrent.futures module provides a high-level interface for asynchronously executing callables. The asynchronous execution can be performed with threads, using ThreadPoolExecutor, or separate processes, using ProcessPoolExecutor.*
>
> **— CONCURRENT.FUTURES — LAUNCHING PARALLEL TASKS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/CONCURRENT.FUTURES.HTML)**

The **multiprocessing** module (https://docs.python.org/3/library/multiprocessing.html) also provides pools of workers using processes and threads in the **Pool (https://superfastpython.com/multiprocessing-pool-python/)** and **ThreadPool (https://superfastpython.com/threadpool-python/)** classes, forerunners to the **ThreadPoolExecutor** and **ProcessPoolExeuctor** classes.

The capabilities of these classes are described in terms of worker execution tasks asynchronously. They explicitly provide synchronous (blocking) and asynchronous (non-blocking) versions of each method for executing tasks.

For example, one may issue a one-off function call synchronously via the **apply()** method or asynchronously via the **apply_async()** method.

> *"A process pool object which controls a pool of worker processes to which jobs can be submitted. It supports asynchronous results with timeouts and callbacks and has a parallel map implementation.*
>
> — **MULTIPROCESSING — PROCESS-BASED PARALLELISM (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/MULTIPROCESSING.HTML)**

There are other aspects of asynchronous programming in Python that are less strictly related to Python concurrency.

For example, Python processes receive or handle signals asynchronously. Signals are fundamentally asynchronous events sent from other processes.

This is primarily supported by the **signal** module (https://docs.python.org/3/library/signal.html).

Now that we know about asynchronous programming, let's take a closer look at asyncio.

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

What is Asyncio

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

Specifically, it refers to two elements:

1. The addition of the “**asyncio**” module to the Python standard library in Python 3.4.
2. The addition of **async/await** expressions to the Python language in Python 3.5.

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

> “*Python 3.4 introduced the asyncio library, and Python 3.5 produced the async and await keywords to use it palatably. These new additions allow so-called asynchronous programming.*
>
> — **PAGE VII, USING ASYNCIO IN PYTHON (HTTPS://AMZN.TO/3MQC92E), 2020.**

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

Changes to Python to add 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.

> “***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 (HTTPS://DOCS.PYTHON.ORG/3/GLOSSARY.HTML#TERM-COROUTINE)**

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

For example:

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

Calling a coroutine function will create a coroutine object, this is a new class. It does not execute the coroutine function.

```python
...
# 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.

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

An asynchronous iterator is an iterator that yields awaitables.

> *“asynchronous iterator: An object that implements the __aiter__() and __anext__() methods. __anext__ must return an awaitable object. async for resolves the awaitables returned by an asynchronous iterator’s __anext__() method until it raises a StopAsyncIteration exception.*
>
> — **PYTHON GLOSSARY (HTTPS://DOCS.PYTHON.ORG/3/GLOSSARY.HTML)**

An asynchronous iterator can be traversed using the “**async for**” expression.

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

This does not execute the for-loop in parallel.

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

An asynchronous context manager is a context manager that can await the enter and exit methods.

> *“An asynchronous context manager is a context manager that is able to suspend execution in its enter and exit methods.*
>
> — **ASYNCHRONOUS CONTEXT MANAGERS AND “ASYNC WITH” (HTTPS://PEPS.PYTHON.ORG/PEP-0492/#ASYNCHRONOUS-CONTEXT-MANAGERS-AND-ASYNC-WITH)**

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.

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

Next, let’s look at the asyncio module.

The asyncio Module

The “**asyncio**” module 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 TCP socket programming).

> *“asyncio is a library to write concurrent code using the async/await syntax.*
>
> — **ASYNCIO — ASYNCHRONOUS I/O (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO.HTML)**

Central to the asyncio module is the event loop.

This is the mechanism that runs a coroutine-based program and implements cooperative multitasking between coroutines.

> *“The event loop is the core of every asyncio application. Event loops run asynchronous tasks and callbacks, perform network IO operations, and run subprocesses.*
>
> — **ASYNCIO EVENT LOOP (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-EVENTLOOP.HTML)**

The module provides both a high-level and low-level API.

The high-level API is for us Python application developers. The low-level API is for framework developers, not us, in most cases.

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

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.

> *“… there are low-level APIs for library and framework developers*
>
> — **ASYNCIO — ASYNCHRONOUS I/O (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO.HTML)**

Now that we know what asyncio is, broadly, and that it is for Asynchronous programming.

Next, let’s explore when we should consider using asyncio in our Python programs.

Confused by the asyncio module API?
Download my FREE PDF cheat sheet (https://marvelous-writer-6152.ck.page/d29b7d8dfb)

When to Use Asyncio

Asyncio, broadly, is new, popular, much discussed, and exciting.

Nevertheless, there is a lot of confusion over when it should be adopted in a project.

When should we use asyncio in Python?

Reasons to Use Asyncio in Python

There are perhaps 3 top-level reasons to use asyncio in a Python project.

They are:

1. Use asyncio in order to adopt coroutines in your program.
2. Use asyncio in order to use the asynchronous programming paradigm.
3. Use asyncio in order to use non-blocking I/O.

Reason 1: To Use Coroutines

We may choose to use asyncio because we want to use coroutines.

We may want to use coroutines because we can have many more concurrent coroutines in our program than concurrent threads.

Coroutines are another unit of concurrency, like threads and processes.

Thread-based concurrency is provided by the threading module and is supported by the underlying operating system. It is suited to blocking I/O tasks such reading and writing from files, sockets, and devices.

Process-based concurrency is provided by the multiprocessing module and is also supported by the underlying operating system, like threads. It is suited to CPU-bound tasks that do not require much inter-process communication, such as compute tasks.

Coroutines are an alternative that is provided by the Python language and runtime (standard interpreter) and further supported by the asyncio module. They are suited to non-blocking I/O with subprocesses and sockets, however, blocking I/O and CPU-bound tasks can be used in a simulated non-blocking manner using threads and processes under the covers.

This last point is subtle and key. Although we can choose to use coroutines for the capability for which they were introduced into Python, non-blocking, we may in fact use them with any tasks. Any program written with threads or processes can be rewritten or instead written using coroutines if we so desire.

Threads and processes achieve multitasking via the operating system that chooses which threads and processes should run, when, and for how long. The operating switches between threads and processes rapidly, suspending those that are not running and resuming those granted time to run. This is called preemptive multitasking.

Coroutines in Python provide an alternative type of multitasking called cooperating multitasking.

A coroutine is a subroutine (function) that can be suspended and resumed. It is suspended by the await expression and resumed once the await expression is resolved.

This allows coroutines to cooperate by design, choosing how and when to suspend their execution.

It is an alternate, interesting, and powerful approach to concurrency, different from thread-based and process-based concurrency.

This alone may make it a reason to adopt it for a project.

Another key aspect of coroutines is that they are lightweight.

They are more lightweight than threads. This means they are faster to start and use less memory. Essentially a coroutine is a special type of function, whereas a thread is represented by a Python object and is associated with a thread in the operating system with which the object must interact.

As such, we may have thousands of threads in a Python program, but we could easily have tens or hundreds of thousands of coroutines all in one thread.

We may choose coroutines for their scalability.

Reason 2: To Use Asynchronous Programming

We may choose to use asyncio because we want to use asynchronous programming in our program.

That is, we want to develop a Python program that uses the asynchronous programming paradigm.

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. It is performed later.

Asynchronous programming often means going all in and designing the program around the concept of asynchronous function calls and tasks.

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.

> *It is a Python library that allows us to run code using an asynchronous programming model.*
>
> — **PAGE 3, PYTHON CONCURRENCY WITH ASYNCIO (HTTPS://AMZN.TO/3CZ7ZH6), 2022.**

We may choose to use asyncio because we want to use the asynchronous programming module in our program, and that is a defensible reason.

To be crystal clear, this reason is independent of using non-blocking I/O. Asynchronous programming can be used independently of non-blocking I/O.

As we saw previously, coroutines can execute non-blocking I/O asynchronously, but the asyncio module also provides the facility for executing blocking I/O and CPU-bound tasks in an asynchronous manner, simulating non-blocking under the covers via threads and processes.

Reason 3: To Use Non-Blocking I/O

We may choose to use asyncio because we want or require non-blocking I/O in our program.

Input/Output or I/O for short means reading or writing from a resource.

Common examples include:

- **Hard disk drives**: Reading, writing, appending, renaming, deleting, etc. files.
- **Peripherals**: mouse, keyboard, screen, printer, serial, camera, etc.
- **Internet**: Downloading and uploading files, getting a webpage, querying RSS, etc.
- **Database**: Select, update, delete, etc. SQL queries.
- **Email**: Send mail, receive mail, query inbox, etc.

These operations are slow, compared to calculating things with the CPU.

The common way these operations are implemented in programs is to make the read or write request and then wait for the data to be sent or received.

As such, these operations are commonly referred to as blocking I/O tasks.

The operating system can see that the calling thread is blocked and will context switch to another thread that will make use of the CPU.

This means that the blocking call does not slow down the entire system. But it does halt or block the thread or program making the blocking call.

You can learn more about blocking calls in the tutorial:

- Thread Blocking Call in Python (https://superfastpython.com/thread-blocking-call-in-python/)

Non-blocking I/O is an alternative to blocking I/O.

It requires support in the underlying operating system, just like blocking I/O, and all modern operating systems provide support for some form of non-blocking I/O.

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

The operating system will handle the request and notify the calling program when the results are available.

- **Non-blocking I/O**: Performing I/O operations via asynchronous requests and responses, rather than waiting for operations to complete.

As such, we can see how non-blocking I/O is related to asynchronous programming. In fact, we use non-blocking I/O via asynchronous programming, or non-blocking I/O is implemented via asynchronous programming.

The combination of non-blocking I/O with asynchronous programming is so common that it is commonly referred to by the shorthand of asynchronous I/O (https://en.wikipedia.org/wiki/Asynchronous_I/O).

- **Asynchronous I/O**: A shorthand that refers to combining asynchronous programming with non-blocking I/O.

The asyncio module in Python was added specifically to add support for non-blocking I/O with subprocesses (e.g. executing commands on the operating system) and with streams (e.g. TCP socket programming) to the Python standard library.

We could simulate non-blocking I/O using threads and the asynchronous programming capability provided by Python thread pools or thread pool executors.

The asyncio module provides first-class asynchronous programming for non-blocking I/O via coroutines, event loops, and objects to represent non-blocking subprocesses and streams.

We may choose to use asyncio because we want to use asynchronous I/O in our program, and that is a defensible reason.

Other Reasons to Use Asyncio

Ideally, we would choose a reason that is defended in the context of the requirements of the project.

Sometimes we have control over the function and non-functional requirements and other times not. In the cases we do, we may choose to use asyncio for one of the reasons listed above. In the cases we don't, we may be led to choose asyncio in order to deliver a program that solves a specific problem.

Some other reasons we may use asyncio include:

1. Use asyncio because someone else made the decision for you.
2. Use asyncio because the project you have joined is already using it.
3. Use asyncio because you want to learn more about it.

We don't always have full control over the projects we work on.

It is common to start a new job, new role, or new project and be told by the line manager or lead architect of specific design and technology decisions.

Using asyncio may be one of these decisions.

We may use asyncio on a project because the project is already using it. You must use asyncio, rather than you choose to use asyncio.

A related example might be the case of a solution to a problem that uses asyncio that you wish to adopt.

For example:

- Perhaps you need to use a third-party API and the code examples use asyncio.
- Perhaps you need to integrate an existing open-source solution that uses asyncio.
- Perhaps you stumble across some code snippets that do what you need, yet they use asyncio.

For lack of alternate solutions, asyncio may be thrust upon you by your choice of solution.

Finally, we may choose asyncio for our Python project to learn more about.

You may scoff, "*what about the requirements*?"

You may choose to adopt asyncio just because you want to try it out and it can be a defensible reason.

Using asyncio in a project will make its workings concrete for you.

When to Not Use Asyncio

We have spent a lot of time on reasons why we should use asyncio.

It is probably a good idea to spend at least a moment on why we should not use it.

One reason to not use asyncio is that you cannot defend its use using one of the reasons above.

This is not foolproof. There may be other reasons to use it, not listed above.

But, if you pick a reason to use asyncio and the reason feels thin or full of holes for your specific case. Perhaps asyncio is not the right solution.

I think the major reason to not use asyncio is that it does not deliver the benefit that you think it does.

There are many misconceptions about Python concurrency, especially around asyncio.

For example:

- Asyncio will work around the global interpreter lock.
- 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.

Only a single coroutine can run at a time by design, they cooperate to execute. This is just like threads under the GIL. In fact, the GIL is an orthogonal concern and probably irrelevant in most cases when using asyncio.

Any program you can write with asyncio, you can write with threads and it will probably be as fast or faster. It will also probably be simpler and easier to read and interpret by fellow developers.

Any concurrency failure mode you might expect with threads, you can encounter with coroutines. You must make coroutines safe from deadlocks and race conditions, just like threads.

Another reason to not use asyncio is that you don’t like asynchronous programming.

Asynchronous programming has been popular for some time now in a number of different programming communities, most notably the JavaScript community.

It is different from procedural, object-oriented, and functional programming, and some developers just don’t like it.

No problem. If you don’t like it, don’t use it. It’s a fair reason.

You can achieve the same effect in many ways, notably by sprinkling a few asynchronous calls in via thread or process executors as needed.

Now that we are familiar with when to use asyncio, let’s look at coroutines in more detail.

Free Python Asyncio Course

Download my asyncio API cheat sheet and as a bonus you will get FREE access to my 7-day email course on asyncio.

Discover how to use the Python asyncio module including how to define, create, and run new coroutines and how to use non-blocking I/O.

Learn more (https://marvelous-writer-6152.ck.page/d29b7d8dfb)

Coroutines in Python

Python provides first-class coroutines with a “**coroutine**” type and new expressions like “**async def**” and “**await**“.

It provides the “**asyncio**” module for running coroutines and developing asynchronous programs.

In this section, we will take a much closer look at coroutines.

What is a Coroutine

A coroutine is a function (https://en.wikipedia.org/wiki/Coroutine) that can be suspended and resumed.

It is often defined 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.

Specifically, coroutines have control over when exactly they suspend their execution.

This may involve the use of a specific expression, such as an “await” expression in Python, like a yield expression in a Python generator.

> “*A coroutine is a method that can be paused when we have a potentially long-running task and then resumed when that task is finished. In Python version 3.5, the language implemented first-class support for coroutines and asynchronous programming when the keywords async and await were explicitly added to the language.*
>
> **— PAGE 3, PYTHON CONCURRENCY WITH ASYNCIO (HTTPS://AMZN.TO/3ENILNS), 2022.**

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.

Coroutines are used for concurrency.

> *Coroutines let you have a very large number of seemingly simultaneous functions in your Python programs.*
>
> — **PAGE 267, EFFECTIVE PYTHON (HTTPS://AMZN.TO/3ECLKFE), 2019.**

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 (https://en.wikipedia.org/wiki/Cooperative_multitasking) and is different from the multitasking typically used with threads called preemptive multitasking tasking.

> *… 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 (HTTPS://EN.WIKIPEDIA.ORG/WIKI/COOPERATIVE_MULTITASKING)**

Preemptive multitasking involves the operating system choosing what threads to suspend and resume and when to do so, as opposed to the tasks themselves deciding in the case of cooperative multitasking.

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

Coroutine vs 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 (https://en.wikipedia.org/wiki/Function_(computer_programming)) 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.

- **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 extension 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 modules of expressions.
- They both can take arguments, or not.
- They both can return a value, or not.

The main difference is that it chooses to suspend and resume its execution many times before returning and exiting.

Both coroutines and subroutines can call other examples of themselves. A subroutine can call other subroutines. A coroutine executes other coroutines. However, a coroutine can also execute other subroutines.

When a coroutine executes another coroutine, it must suspend its execution and allow the other coroutine to 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 to run as well.

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

Coroutine vs 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 (HTTPS://DOCS.PYTHON.ORG/3/GLOSSARY.HTML)**

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 (HTTPS://DOCS.PYTHON.ORG/3/GLOSSARY.HTML)**

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.

> *Using this paradigm, an await statement is similar in function to a yield statement; the execution of the current function gets paused while other code is run. Once the await or yield resolves with data, the function is resumed.*
>
> — **PAGE 218, HIGH PERFORMANCE PYTHON (HTTPS://AMZN.TO/3RY7CQE), 2020.**

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 (HTTPS://EN.WIKIPEDIA.ORG/WIKI/COROUTINE).**

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.

> *To implement your own concurrency using generators, you first need a fundamental insight concerning generator functions and the yield statement. Specifically, the fundamental behavior of yield is that it causes a generator to suspend its execution. By suspending execution, it is possible to write a scheduler that treats generators as a kind of "task" and alternates their execution using a kind of cooperative task switching.*
>
> — **PAGE 524, PYTHON COOKBOOK (HTTPS://AMZN.TO/3D002LI), 2013.**

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 async/await expressions.

Coroutine vs 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 (https://docs.python.org/3/library/asyncio-task.html#asyncio.Task).

> *"A Future-like object that runs a Python coroutine. […] Tasks are used to run coroutines in event loops.*
>
> — **ASYNCIO TASK OBJECT (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-TASK.HTML#ASYNCIO.TASK)**

A coroutine can be wrapped in an **asyncio.Task** object and executed independently, as opposed to being executed directly within a coroutine. The **Task** object provides a handle on the asynchronously execute 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.

You can learn more about **asyncio.Task** objects in the tutorial:

Coroutine vs Thread

A coroutine is more lightweight than a 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.

> “*The cost of starting a coroutine is a function call. Once a coroutine is active, it uses less than 1 KB of memory until it’s exhausted.*
>
> **— PAGE 267, EFFECTIVE PYTHON (HTTPS://AMZN.TO/3ECLKFE), 2019.**

Coroutines execute within one thread, therefore a single thread may execute many coroutines.

> “*Many separate async functions advanced in lockstep all seem to run simultaneously, mimicking the concurrent behavior of Python threads. However, coroutines do this without the memory overhead, startup and context switching costs, or complex locking and synchronization code that’s required for threads.*
>
> **— PAGE 267, EFFECTIVE PYTHON (HTTPS://AMZN.TO/3ECLKFE), 2019.**

You can learn more about threads in the guide:

- Python Threading: The Complete Guide (https://superfastpython.com/threading-in-python/)

Coroutine vs Process

A coroutine is more lightweight than a process.

In fact, a thread is more lightweight than a process.

A process is a computer program. It may have one or many threads.

A Python process is in fact a separate instance of the Python interpreter.

Processes, like threads, are created and managed by the underlying operating system and are represented by a **multiprocessing.Process** object.

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

This means that coroutines are significantly faster than a process to create and start and take up much less memory.

A coroutine is just a special function, whereas a Process is an instance of the interpreter that has at least one thread.

You can learn more about Python processes in the guide:

- Python Multiprocessing: The Complete Guide (https://superfastpython.com/multiprocessing-in-python/)

When Were Coroutines Added to Python

Coroutines extend generators in 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 probability addition of asyncio.

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

These were added in Python 2.5 and described in PEP 342 (https://peps.python.org/pep-0342/).

> “*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 (HTTPS://PEPS.PYTHON.ORG/PEP-0342/)**

Later, allowing generators to emit a suspension exception as well as a stop exception described in PEP 334 (https://peps.python.org/pep-0334/).

> “*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 (HTTPS://PEPS.PYTHON.ORG/PEP-0334/)**

The vast majority of the capabilities for working with modern coroutines in Python via the asyncio module were described in PEP 3156 (https://peps.python.org/pep-3156/), added in Python 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: THE “ASYNCIO” MODULE (HTTPS://PEPS.PYTHON.ORG/PEP-3156/)**

A second approach to coroutines, based on generators, was added to Python 3.4 (https://docs.python.org/3.4/library/asyncio-task.html) as an extension to Python generators.

A coroutine was defined as a function that used the **@asyncio.coroutine** decorator.

Coroutines were executed using an asyncio event loop, via the asyncio module.

A coroutine could suspend and execute another coroutine via the "**yield from**" expression

For example:

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

The "**yield from**" expression was defined in PEP 380 (https://peps.python.org/pep-0380/).

> *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 (HTTPS://PEPS.PYTHON.ORG/PEP-0380/)**

The "**yield from**" expression (https://docs.python.org/3/reference/expressions.html#yield-expressions) is still available for use in generators, although is a deprecated approach to suspending execution in coroutines, in favor of the "**await**" expression (https://docs.python.org/3/reference/expressions.html#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.*
>
> — **ASYNCIO COROUTINES AND TASKS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-TASK.HTML)**

We might say that 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**“, “**await**“, “**async with**“, and “**async for**” expressions, as well as a coroutine type.

These changes were described in PEP 492 (https://peps.python.org/pep-0492/).

> *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 (HTTPS://PEPS.PYTHON.ORG/PEP-0492/)**

Now that we know what a coroutine is, let’s take a closer look at how to use them in Python.

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

Define, Create and Run Coroutines

We can define coroutines in our Python programs, just like defining new subroutines (functions).

Once defined, a coroutine function can be used to create a coroutine object.

The “**asyncio**” module provides tools to run our coroutine objects in an event loop, which is a runtime for coroutines.

How to Define a Coroutine

A coroutine can be defined via the “**async def**” expression.

This is an extension of the “**def**” expression for defining subroutines.

It defines a coroutine that can be created and returns a coroutine object.

For example:

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

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

> *coroutine function: A function which returns a coroutine object. A coroutine function may be defined with the async def statement, and may contain await, async for, and async with keywords.*
>
> — **PYTHON GLOSSARY (HTTPS://DOCS.PYTHON.ORG/3/GLOSSARY.HTML)**

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

> *Execution of Python coroutines can be suspended and resumed at many points (see coroutine). await expressions, async for and async with can only be used in the body of a coroutine function.*
>
> — **COROUTINE FUNCTION DEFINITION (HTTPS://DOCS.PYTHON.ORG/3/REFERENCE/COMPOUND_STMTS.HTML#ASYNC-DEF)**

For example:

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

How to 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 returns a “**coroutine**” object (https://docs.python.org/3/reference/datamodel.html#coroutines).

> *“You can think of a coroutine function as a factory for coroutine objects; more directly, remember that calling a coroutine function does not cause any user-written code to execute, but rather just builds and returns a coroutine object.*
>
> **— PAGE 516, PYTHON IN A NUTSHELL (HTTPS://AMZN.TO/3TAZSBW), 2017.**

A “**coroutine**” Python object has methods, such as **send()** and **close()**. It is a type.

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

For example:

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

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

# 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 RuntimeError because the coroutine was created but never executed, we will explore that in the next section.

```
<class 'coroutine'>
sys:1: RuntimeWarning: coroutine 'custom_coro' was never awaited
```

A coroutine object is an awaitable.

This means it is a Python type that implements the **__await__()** method.

> *An awaitable object generally implements an __await__() method. Coroutine objects returned from async def functions are awaitable.*
>
> — **AWAITABLE OBJECTS (HTTPS://DOCS.PYTHON.ORG/3/REFERENCE/DATAMODEL.HTML#COROUTINES)**

You can learn more about awaitables in the tutorial:

- What is an Asyncio Awaitable in Python (/asyncio-awaitable)

How to Run a Coroutine From Python

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

> *The event loop is the core of every asyncio application. Event loops run asynchronous tasks and callbacks, perform network IO operations, and run subprocesses.*
>
> — **ASYNCIO EVENT LOOP (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-EVENTLOOP.HTML)**

The event loop that executes coroutines, manages the cooperative multitasking between coroutines.

> *Coroutine objects can only run when the event loop is running.*
>
> — **PAGE 517, PYTHON IN A NUTSHELL (HTTPS://AMZN.TO/3TAZSBW), 2017.**

The typical way to start a coroutine event loop is via the **asyncio.run()** function (https://docs.python.org/3/library/asyncio-task.html#asyncio.run).

This function takes one coroutine and returns the value of the coroutine. The provided coroutine can be used as the entry point into the coroutine-based program.

For example:

```python
# SuperFastPython.com
# example of running a coroutine
import asyncio
# define a coroutine
async def custom_coro():
    # await another coroutine
    await asyncio.sleep(1)

# main coroutine
async def main():
    # execute my custom coroutine
    await custom_coro()

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

Running the example

Now that we know how to define, create, and run a coroutine, let's take a moment to understand the event loop.

What is the Event Loop

The heart of asyncio programs is the event loop.

In this section, we will take a moment to look at the asyncio event loop.

What is the Asyncio Event Loop

The event loop is an environment for executing coroutines in a single thread.

> *"asyncio is a library to execute these coroutines in an asynchronous fashion using a concurrency model known as a single-threaded event loop.*
>
> **— PAGE 3, PYTHON CONCURRENCY WITH ASYNCIO (HTTPS://AMZN.TO/3VVAQ59), 2022.**

The event loop is the core of an asyncio program.

It does many things, such as:

1. Execute coroutines.

2. Execute callbacks.
3. Perform network input/output.
4. Run subprocesses.

> *The event loop is the core of every asyncio application. Event loops run asynchronous tasks and callbacks, perform network IO operations, and run subprocesses.*
>
> — **ASYNCIO EVENT LOOP (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-EVENTLOOP.HTML)**

Event loops are a common design pattern and became very popular in recent times given their use in JavaScript.

> *JavaScript has a runtime model based on an event loop, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks. This model is quite different from models in other languages like C and Java.*
>
> — **THE EVENT LOOP, MOZILLA (HTTPS://DEVELOPER.MOZILLA.ORG/EN-US/DOCS/WEB/JAVASCRIPT/EVENTLOOP).**

The event loop, as its name suggests, is a loop. It manages a list of tasks (coroutines) and attempts to progress each in sequence in each iteration of the loop, as well as perform other tasks like executing callbacks and handling I/O.

The “**asyncio**” module provides functions for accessing and interacting with the event loop.

This is not required for typical application development.

Instead, access to the event loop is provided for framework developers, those that want to build on top of the asyncio module or enable asyncio for their library.

> *Application developers should typically use the high-level asyncio functions, such as asyncio.run(), and should rarely need to reference the loop object or call its methods.*
>
> — **ASYNCIO EVENT LOOP (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-EVENTLOOP.HTML)**

The asyncio module provides a low-level API for getting access to the current event loop object, as well as a suite of methods that can be used to interact with the event loop.

The low-level API is intended for framework developers that will extend, complement and integrate asyncio into third-party libraries.

We rarely need to interact with the event loop in asyncio programs, in favor of using the high-level API instead.

Nevertheless, we can briefly explore how to get the event loop.

How To Start and Get An Event Loop

The typical way we create an event loop in asyncio applications is via the **asyncio.run()** function.

> *This function always creates a new event loop and closes it at the end. It should be used as a main entry point for asyncio programs, and should ideally only be called once.*
>
> — **ASYNCIO COROUTINES AND TASKS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-TASK.HTML)**

The function takes a coroutine and will execute it to completion.

We typically pass it to our main coroutine and run our program from there.

There are low-level functions for creating and accessing the event loop.

The **asyncio.new_event_loop()** function (https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.new_event_loop) will create a new event loop and return access to it.

> “*Create and return a new event loop object.*
>
> — **ASYNCIO EVENT LOOP (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-EVENTLOOP.HTML)**

For example:

```
...
# create and access a new asyncio event loop
loop = asyncio.new_event_loop()
```

We can demonstrate this with a worked example.

In the example below we will create a new event loop and then report its details.

```
# SuperFastPython.com
# example of creating an event loop
import asyncio

# create and access a new asyncio event loop
loop = asyncio.new_event_loop()
# report defaults of the loop
print(loop)
```

Running the example creates the event loop, then reports the details of the object.

We can see that in this case the event loop has the type **_UnixSelectorEventLoop** and is not running, but is also not closed.

```
<_UnixSelectorEventLoop running=False closed=False debug=False>
```

If an asyncio event loop is already running, we can get access to it via the **asyncio.get_running_loop()** function (https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.get_running_loop).

> “*Return the running event loop in the current OS thread. If there is no running event loop a RuntimeError is raised. This function can only be called from a coroutine or a callback.*
>
> — **ASYNCIO EVENT LOOP (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-EVENTLOOP.HTML)**

For example:

```
...
# access he running event loop
loop = asyncio.get_running_loop()
```

There is also a function for getting or starting the event loop called **asyncio.get_event_loop() (https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.get_event_loop)**, but it was deprecated in Python 3.10 and should not be used.

What is an Event Loop Object

An event loop is implemented as a Python object.

The event loop object defines how the event loop is implemented and provides a common API for interacting with the loop, defined on the **AbstractEventLoop** class (https://github.com/python/cpython/blob/3.10/Lib/asyncio/events.py#L204).

There are different implementations (https://docs.python.org/3/library/asyncio-eventloop.html#event-loop-implementations) of the event loop for different platforms.

For example, Windows and Unix-based operations systems will implement the event loop in different ways, given the different underlying ways that non-blocking I/O is implemented on these platforms.

The **SelectorEventLoop** type event loop (https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.SelectorEventLoop) is the default on Unix-based operating systems like Linux and macOS.

The **ProactorEventLoop** type event loop (https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.ProactorEventLoop) is the default on Windows.

Third-party libraries may implement their own event loops to optimize for specific features.

Why Get Access to The Event Loop

Why would we want access to an event loop outside of an asyncio program?

There are many reasons why we may want access to the event loop from outside of a running asyncio program.

For example:

1. To monitor the progress of tasks.
2. To issue and get results from tasks.
3. To fire and forget one-off tasks.

An asyncio event loop can be used in a program as an alternative to a thread pool for coroutine-based tasks.

An event loop may also be embedded within a normal asyncio program and accessed as needed.

Now that we know a little about the event loop, let's look at asyncio tasks.

Create and Run Asyncio Tasks

You can create Task objects from coroutines in asyncio programs.

Tasks provide a handle on independently scheduled and running coroutines and allow the task to be queried, canceled, and results and exceptions to be retrieved later.

The asyncio event loop manages tasks. As such, all coroutines become and are managed as tasks within the event loop.

Let's take a closer look at asyncio tasks.

What is an Asyncio Task

A 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.

> *A Task is an object that manages an independently running coroutine.*
>
> — **PEP 3156 – ASYNCHRONOUS IO SUPPORT REBOOTED: THE "ASYNCIO" MODULE (HTTPS://PEPS.PYTHON.ORG/PEP-3156/)**

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.

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 wait for it to complete.

> *Tasks are used to schedule coroutines concurrently. When a coroutine is wrapped into a Task with functions like asyncio.create_task() the coroutine is automatically scheduled to run soon*
>
> — **COROUTINES AND TASKS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-TASK.HTML)**

The **asyncio.Task** class (https://docs.python.org/3/library/asyncio-task.html#task-object) extends the **asyncio.Future** class (https://docs.python.org/3/library/asyncio-future.html#asyncio.Future) and an instance are awaitable.

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

> *A Future is a special low-level awaitable object that represents an eventual result of an asynchronous operation.*
>
> — **COROUTINES AND TASKS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-TASK.HTML)**

Classes that extend the Future class are often referred to as **Future**-like.

“*A Future-like object that runs a Python coroutine.*

— **COROUTINES AND TASKS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-TASK.HTML)**

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

For example:

```
...
# wait for a task to be done
await task
```

Now that we know what an asyncio task is, let’s look at how we might create one.

How to Create a Task

A task is created using a provided coroutine instance.

Recall that a coroutine is defined using the async def expression and looks like a function.

For example:

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

A task can only be created and scheduled within a coroutine.

There are two main ways to create and schedule a task, they are:

1. Create Task With High-Level API (preferred)
2. Create Task With Low-Level API

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

Create Task With High-Level API

A task can be created using the **asyncio.create_task()** function (https://docs.python.org/3/library/asyncio-task.html#asyncio.create_task).

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

For example:

```
...
# create a coroutine
coro = task_coroutine()
# create a task from a coroutine
task = asyncio.create_task(coro)
```

This can be achieved with a compound statement on a single line.

For example:

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

This will do a few things:

1. Wrap the coroutine in a Task instance.
2. Schedule the task for execution in the current event loop.
3. Return a Task instance

The task instance can be discarded, interacted with via methods, and awaited by a coroutine.

This is the preferred way to create a Task from a coroutine in an asyncio program.

Create Task With Low-Level API

A task can also be created from a coroutine using the lower-level asyncio API.

The first way is to use the **asyncio.ensure_future()** function (https://docs.python.org/3/library/asyncio-future.html#asyncio.ensure_future).

This function takes a **Task**, **Future**, or **Future**-like object, such as a coroutine, and optionally the loop in which to schedule it.

If a loop is not provided, it will be scheduled in the current event loop.

If a coroutine is provided to this function, it is wrapped in a Task instance for us, which is returned.

For example:

```
...
# create and schedule the task
task = asyncio.ensure_future(task_coroutine())
```

Another low-level function that we can use to create and schedule a **Task** is the **loop.create_task()** method (https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.loop.create_task).

This function requires access to a specific event loop in which to execute the coroutine as a task.

We can acquire an instance to the current event loop within an asyncio program via the **asyncio.get_event_loop()** function (https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.get_event_loop).

This can then be used to call the **create_task()** method to create a **Task** instance and schedule it for execution.

For example:

```
...
# get the current event loop
loop = asyncio.get_event_loop()
# create and schedule the task
task = loop.create_task(task_coroutine())
```

When Does a Task Run?

A common question after creating a task is when does it run?

This is a good question.

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.

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

For example, if we had an asyncio program with one coroutine that created and scheduled a task, the scheduled task will not run until the calling coroutine that created the task is suspended.

This may happen if the calling coroutine chooses to sleep, chooses to await another coroutine or task, or chooses to await the new task that was scheduled.

For example:

```
...
# create a task from a coroutine
task = asyncio.create_task(task_coroutine())
# await the task, allowing it to run
await task
```

Now that we know what a task is and how to schedule them, next, let's look at how we may use them in our programs

Work With and Query Tasks

Tasks are the currency of asyncio programs.

In this section, we will take a closer look at how to interact with them in our programs.

Task Life-Cycle

An asyncio Task has a life cycle.

Firstly, a task is created from a coroutine.

It is then scheduled for independent execution within the event loop.

At some point, it will run.

While running it may be suspended, such as awaiting another coroutine or task. It may finish normally and return a result or fail with an exception.

Another coroutine may intervene and cancel the task.

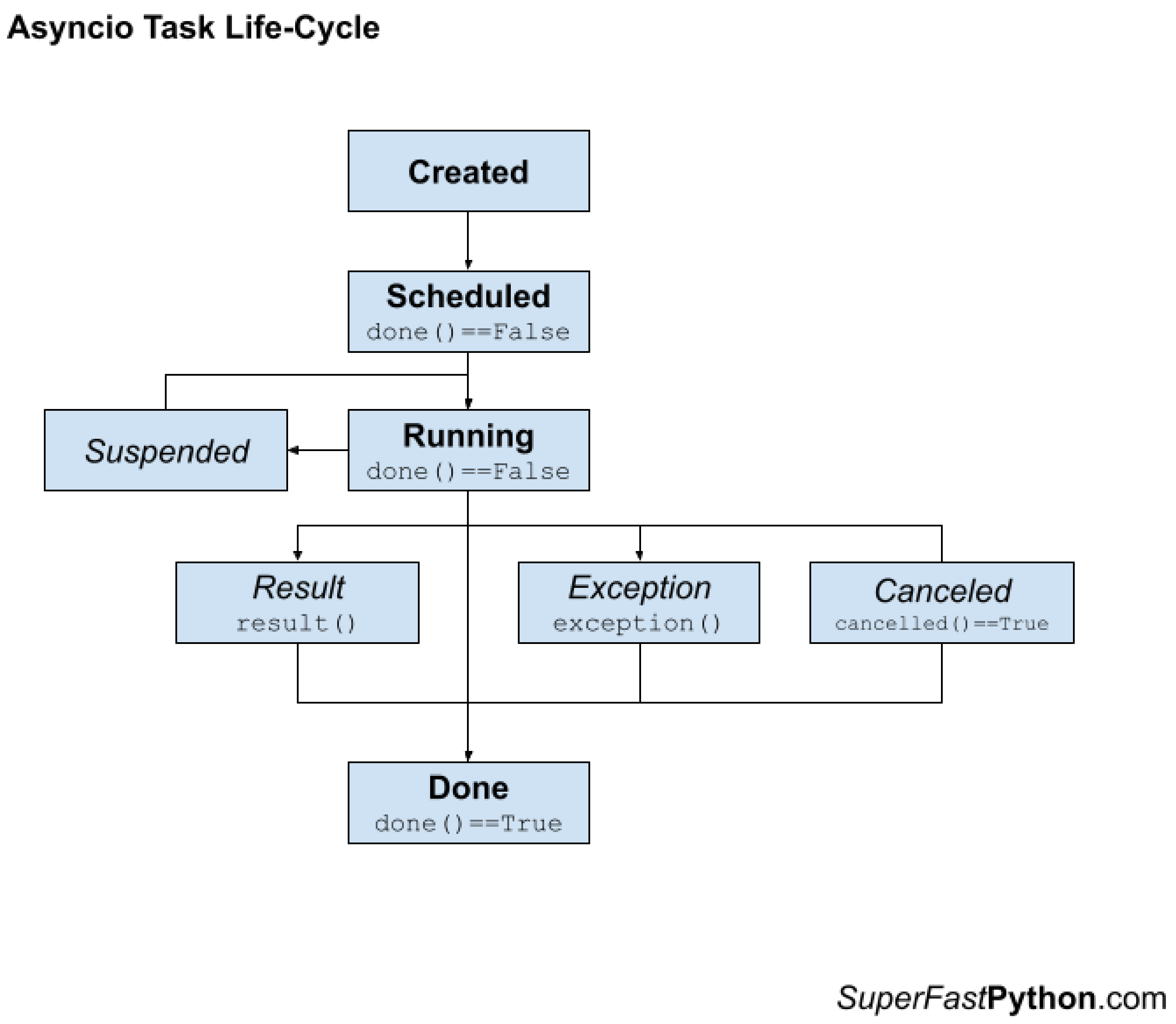
Eventually, it will be done and cannot be executed again.

We can summarize this life-cycle as follows:

- 1. Created
- 2. Scheduled
    - 2a Canceled
- 3. Running
    - 3a. Suspended
    - 3b. Result
    - 3c. Exception
    - 3d. Canceled
- 4. Done

Note that Suspended, Result, Exception, and Canceled are not states per se, they are important points of transition for a running task.

The diagram below summarizes this life cycle showing the transitions between each phase.



ASYNCIO TASK LIFE-CYCLE

Now that we are familiar with the life cycle of a task from a high level, let's take a closer look at each phase.

How to Check Task Status

After a Task is created, we can check the status of the task.

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

- Whether the task is done.
- Whether the task was canceled.

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

Check if a Task is Done

We can check if a task is done via the **done()** method (https://docs.python.org/3/library/asyncio-task.html#asyncio.Task.done).

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

For example:

```
...
# check if a task is done
if task.done():
    # ...
```

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.

Similarly, a task that is running is not done.

A task is done if:

- The coroutine finishes normally.
- The coroutine returns explicitly.
- An unexpected error or exception is raised in the coroutine
- The task is canceled.

Check if a Task is Canceled

We can check if a task is canceled via the **cancelled()** method (https://docs.python.org/3/library/asyncio-task.html#asyncio.Task.cancelled).

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.

How to Get Task Result

We can get the result of a task via the **result()** method (https://docs.python.org/3/library/asyncio-task.html#asyncio.Task.result).

This returns the return value of the coroutine wrapped by the **Task** or **None** if the wrapped coroutine does not explicitly return a value.

For example:

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

If the coroutine raises an unhandled error or 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
```

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:

```python
...
# 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
```

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:

```python
...
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:

```python
...
# check if the task is not done
if not task.done():
    await task
# get the return value from the wrapped coroutine
value = task.result()
```

How to Get Task Exception

A coroutine wrapped by a task may raise an exception that is not handled.

This will cancel the task, in effect.

We can retrieve an unhandled exception in the coroutine wrapped by a task via the **exception()** method (https://docs.python.org/3/library/asyncio-task.html#asyncio.Task.exception).

For example:

```python
...
# get the exception raised by a task
exception = task.exception()
```

If an unhandled exception was not raised in the wrapped coroutine, then a value of None is returned.

If the task was canceled, then a **CancelledError** 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.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
```

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()
```

How to Cancel a Task

We can cancel a scheduled task via the **cancel()** method (https://docs.python.org/3/library/asyncio-task.html#asyncio.Task.cancel).

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

For example:

```
...
# cancel the task
was_cancelled = task.cancel()
```

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.

The next time the task is given an opportunity to run, it will raise a **CancelledError** exception.

If the **CancelledError** exception is not handled within the wrapped coroutine, the task will be canceled.

Otherwise, if the **CancelledError** exception is handled within the wrapped coroutine, the task will not be canceled.

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

How to Use Callback With a Task

We can add a done callback function to a task via the **add_done_callback()** method (https://docs.python.org/3/library/asyncio-task.html#asyncio.Task.add_done_callback).

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.

For example:

```
# done callback function
def handle(task):
    print(task)

...
# register a done callback function
task.add_done_callback(handle)
```

Recall that a task may be done when the wrapped coroutine finishes normally when it returns, 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.

We can also remove or de-register a callback function via the **remove_done_callback()** function (https://docs.python.org/3/library/asyncio-task.html#asyncio.Task.remove_done_callback).

For example:

```
...
# remove a done callback function
task.remove_done_callback(handle)
```

How to Set the Task Name

A task may have a name.

This name can be helpful if multiple tasks are created from the same coroutine and we need some way to tell them apart programmatically.

The name can be set when the task is created from a coroutine via the “**name**” argument.

For example:

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

The name for the task can also be set via the **set_name()** method (https://docs.python.org/3/library/asyncio-task.html#asyncio.Task.set_name).

For example:

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

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()
```

Current and Running Tasks

We can introspect tasks running in the asyncio event loop.

This can be achieved by getting an **asyncio.Task** object for the currently running task and for all tasks that are running.

How to Get the Current Task

We can get the current task via the **asyncio.current_task()** function (https://docs.python.org/3/library/asyncio-task.html#asyncio.current_task).

This function will return a **Task** object for the task that is currently running.

For example:

```python
...
# get the current task
task = asyncio.current_task()
```

This will return a **Task** object for the currently running task.

This may be:

- The main coroutine passed to **asyncio.run()**.
- A task created and scheduled within the asyncio program via **asyncio.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.

Getting the current task can be helpful if a coroutine or task requires details about itself, such as the task name for logging.

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 first example, as it highlights that all coroutines can be accessed as tasks within the asyncio event loop.

The complete example is listed below.

```python
# SuperFastPython.com
# example of getting the current task from 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 program
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 ...]>
```

How to Get All 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.
- To find a task that can be queried or canceled.

We can get a set 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 a set of all tasks in the asyncio program.

It is a set 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 for the currently running task, e.g. the task that is executing the coroutine that calls the **asyncio.all_tasks()** function.

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.

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.

```python
# SuperFastPython.com
# example of starting many tasks and getting access to all 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 program
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 0x10e186e30>
> Task-2, <coroutine object task_coroutine at 0x10e184e40>
> Task-11, <coroutine object task_coroutine at 0x10e186f10>
> Task-7, <coroutine object task_coroutine at 0x10e186d50>
> Task-4, <coroutine object task_coroutine at 0x10e185700>
> Task-10, <coroutine object task_coroutine at 0x10e186ea0>
> Task-8, <coroutine object task_coroutine at 0x10e186dc0>
> Task-5, <coroutine object task_coroutine at 0x10e186ab0>
> Task-1, <coroutine object main at 0x10e1847b0>
> Task-3, <coroutine object task_coroutine at 0x10e184f90>
> Task-6, <coroutine object task_coroutine at 0x10e186ce0>
```

Next, we will explore how to run many coroutines concurrently.

Run Many Coroutines Concurrently

A benefit of asyncio is that we can run many coroutines concurrently.

These coroutines can be created in a group and stored, then executed all together at the same time.

This can be achieved using the **asyncio.gather()** function.

Let’s take a closer look.

What is Asyncio gather()

The **asyncio.gather()** module (https://docs.python.org/3/library/asyncio-task.html#asyncio.gather) function allows the caller to group multiple awaitables together.

Once grouped, the awaitables can be executed concurrently, awaited, and canceled.

> “*Run awaitable objects in the aws sequence concurrently.*
>
> — **COROUTINES AND TASKS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-TASK.HTML)**

It is a helpful utility function for both grouping and executing multiple coroutines or multiple tasks.

For example:

```
...
# run a collection of awaitables
results = await asyncio.gather(coro1(), asyncio.create_task(coro2()))
```

We may use the **asyncio.gather()** function in situations where we may create many tasks or coroutines up-front and then wish to execute them all at once and wait for them all to complete before continuing on.

This is a likely situation where the result is required from many like-tasks, e.g. same task or coroutine with different data.

The awaitables can be executed concurrently, results returned, and the main program can resume by making use of the results on which it is dependent.

The **gather()** function is more powerful than simply waiting for tasks to complete.

It allows a group of awaitables to be treated as a single awaitable.

This allows:

- Executing and waiting for all awaitables in the group to be done via an await expression.
- Getting results from all grouped awaitables to be retrieved later via the result() method.
- The group of awaitables to be canceled via the cancel() method.
- Checking if all awaitables in the group are done via the done() method.
- Executing callback functions only when all tasks in the group are done.

And more.

How to use Asyncio gather()

In this section, we will take a closer look at how we might use the **asyncio.gather()** function.

The **asyncio.gather()** function takes one or more awaitables as arguments.

Recall an awaitable may be a coroutine, a **Future** or a **Task**.

Therefore, we can call the **gather()** function with:

- Multiple tasks
- Multiple coroutines
- 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 running because **Tasks** are scheduled as part of being created.

The **asyncio.gather()** function takes awaitables as position arguments.

We cannot create a list or collection of awaitables and provide it to gather, as this will result in an error.

For example:

```
...
# cannot provide a list of awaitables directly
asyncio.gather([coro1(), coro2()])
```

A list of awaitables can be provided if it is first unpacked into separate expressions using the star operator (*).

For example:

```
...
# gather with an unpacked list of awaitables
asyncio.gather(*[coro1(), coro2()])
```

If coroutines are provided to **gather()**, they are wrapped in **Task** objects automatically.

The **gather()** function does not block.

Instead, it returns an **asyncio.Future (https://docs.python.org/3/library/asyncio-future.html#asyncio.Future)** object that represents the group of awaitables.

For example:

```
...
# get a future that represents multiple awaitables
group = asyncio.gather(coro1(), coro2())
```

Once the **Future** 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 you do not have to await the **Future** that is returned from **gather()**.

For example:

```
...
# get a future that represents multiple awaitables
group = asyncio.gather(coro1(), coro2())
# suspend and wait a while, the group may be executing..
await asyncio.sleep(10)
```

The returned Future 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 Future returned from **gather()** will return a list of return values from the awaitables.

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())
```

Example of gather() For Many Coroutines in a List

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 many coroutines
coros = [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:

```
...
# run the tasks
await asyncio.gather(*coros)
```

Tying this together, the complete example of running a list of pre-prepared coroutines with **gather()** is listed below.

```python
# 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 many coroutines
    coros = [task_coro(i) for i in range(10)]
    # run the tasks
    await asyncio.gather(*coros)
    # report a message
    print('main done')

# start the asyncio program
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 Future 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
```

Next, we will explore how to wait on a group of asyncio tasks.

Wait for A Collection of Tasks

We can wait for asyncio tasks to complete via the **asyncio.wait()** function.

Different conditions can be waited for, such as all tasks to complete, the first task to complete, and the first task to fail with an exception.

Let's take a closer look.

What is asyncio.wait()

The **asyncio.wait()** function (https://docs.python.org/3/library/asyncio-task.html#asyncio.wait) can be used to wait for a collection of asyncio tasks to complete.

Recall that an asyncio task is an instance of the **asyncio.Task** class that wraps a coroutine. It allows a coroutine to be scheduled and executed independently, and the **Task** instance provides a handle on the task for querying status and getting results.

You can learn more about asyncio tasks in the tutorial:

- What is an Asyncio Task (/asyncio-task)

The wait() function allows us to wait for a collection of tasks to be done.

The call to wait can be configured to wait for different conditions, such as all tasks being completed, the first task completed and the first task failing with an error.

Next, let’s look at how we might use the wait() function.

How to Use asyncio.wait()

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)
```

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.

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.

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.

For example:

```python
...
# wait for all tasks to complete with a timeout
done, pending = await asyncio.wait(tasks, timeout=3)
```

If the timeout is reached before the condition is met, an exception is not raised and the remaining tasks are not canceled.

Now that we know how to use the **asyncio.wait()** function, let's look at some worked examples.

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 be completed.

The complete example is listed below.

```python
# 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 program
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, we will explore how to wait for a single coroutine with a time limit.

Wait for a Coroutine with a Time Limit

We can wait for an asyncio task or coroutine to complete with a timeout using the **asyncio.wait_for()** function.

If the timeout elapses before the task completes, the task is canceled.

Let’s take a closer look.

What is Asyncio wait_for()

The **asyncio.wait_for()** function allows the caller to wait for an asyncio task or coroutine to complete with a timeout.

If no timeout is specified, the **wait_for()** function will wait until the task is completed.

If a timeout is specified and elapses before the task is complete, then the task is canceled.

> *"Wait for the aw awaitable to complete with a timeout.*
>
> — COROUTINES AND TASKS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-TASK.HTML)

This allows the caller to both set an expectation about how long they are willing to wait for a task to complete, and to enforce the timeout by canceling the task if the timeout elapses.

Now that we know what the **asyncio.wait_for()** function is, let's look at how to use it.

How to Use Asyncio wait_for()

The **asyncio.wait_for()** function (https://docs.python.org/3/library/asyncio-task.html#asyncio.wait_for) takes an awaitable and a timeout.

The awaitable may be a coroutine or a task.

A timeout must be specified and may be **None** for no timeout, an integer 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, 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 waited-for 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 how we can call **wait_for()** with a timeout.

Example of Asyncio wait_for() 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.

```python
# 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 program
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, we will explore how we might protect an asyncio task from being canceled.

Shield Tasks from Cancellation

Asyncio tasks can be canceled by calling their **cancel()** method.

We can protect a task from being canceled by wrapping it in a call to **asyncio.shield()**.

Let's take a closer look.

What is Asyncio shield()

The asyncio.shield() function wraps an awaitable in Future that will absorb requests to be canceled.

> “*Protect an awaitable object from being cancelled.*
>
> — **COROUTINES AND TASKS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-TASK.HTML)**

This means the shielded future can be passed around to tasks that may try to cancel it and the cancellation request will look like it was successful, except that the Task or coroutine that is being shielded will continue to run.

It may be useful in asyncio programs where some tasks can be canceled, but others, perhaps with a higher priority cannot.

It may also be useful in programs where some tasks can safely be canceled, such as those that were designed with asyncio in mind, whereas others cannot be safely terminated and therefore must be shielded from cancellation.

Now that we know what **asyncio.shield()** is, let's look at how to use it.

How to Use Asyncio shield()

The **asyncio.shield()** function (https://docs.python.org/3/library/asyncio-task.html#asyncio.shield) will protect another **Task** or coroutine from being canceled.

It takes an awaitable as an argument and returns an **asyncio.Future** object.

The Future 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
```

The returned **Future** 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 **Future** object will raise an **asyncio.CancelledError**, 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 **Future** object is not propagated to the inner task.

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()
```

If a coroutine is provided to the **asyncio.shield()** function it is wrapped in an **asyncio.Task()** and scheduled immediately.

This means that the shield does not need to be awaited for the inner coroutine to run.

> “*If aw is a coroutine it is automatically scheduled as a Task.*
>
> — **COROUTINES AND TASKS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-TASK.HTML)**

If the task that is being shielded is canceled, the cancellation request will be propagated up to the shield, which will also be canceled.

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.

Example of Asyncio shield() for a Task

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.

```python
# SuperFastPython.com
# example of using asyncio shield to protect 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
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 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 **Future**, 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 future and the inner task. We can see that the shielded future 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, we will explore how to run a blocking task from an asyncio program.

Run a Blocking Task in Asyncio

A blocking task is a task that stops the current thread from progressing.

If a blocking task is executed in an asyncio program it stops the entire event loop, preventing any other coroutines from progressing.

We can run blocking calls asynchronously in an asyncio program via the **asyncio.to_thread()** and **loop.run_in_executor()** functions.

Need to Run Blocking Tasks in Asyncio

The focus of asyncio is asynchronous programming and non-blocking IO.

Nevertheless, we often need to execute a blocking function call within an asyncio application.

This could be for many reasons, such as:

- To execute a CPU-bound task like calculating something.
- To execute a blocking IO-bound task like reading or writing from a file.
- To call into a third-party library that does not support asyncio yet.

Making a blocking call directly in an asyncio program will cause the event loop to stop while the blocking call is executing. It will not allow other coroutines to run in the background.

How can we execute a blocking call in an asyncio program asynchronously?

How to Run Blocking Tasks

The asyncio module provides two approaches for executing blocking calls in asyncio programs.

The first is to use the **asyncio.to_thread()** function (https://docs.python.org/3/library/asyncio-task.html#asyncio.to_thread).

This is in the high-level API and is intended for application developers.

The **asyncio.to_thread()** function takes a function name to execute and any arguments.

The function is executed in a separate thread. It returns a coroutine that can be awaited or scheduled as an independent task.

For example:

```
...
# execute a function in a separate thread
await asyncio.to_thread(task)
```

The task will not begin executing until the returned coroutine is given an opportunity to run in the event loop.

The **asyncio.to_thread()** function creates a **ThreadPoolExecutor** behind the scenes to execute blocking calls.

As such, the **asyncio.to_thread()** function is only appropriate for IO-bound tasks.

An alternative approach is to use the **loop.run_in_executor()** function (https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.loop.run_in_executor).

This is in the low-level asyncio API and first requires access to the event loop, such as via the **asyncio.get_running_loop()** function (https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.get_running_loop).

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 **ThreadPoolExecutor**.

The **loop.run_in_executor()** function returns an awaitable that can be awaited if needed. 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:

```
...
# get the event loop
loop = asyncio.get_running_loop()
# execute a function in a separate thread
await loop.run_in_executor(None, task)
```

Alternatively, an executor can be created and passed to the **loop.run_in_executor()** function, which will execute the asynchronous call in the executor.

The caller must manage the executor in this case, shutting it down once the caller is finished with it.

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...
```

These two approaches allow a blocking call to be executed as an asynchronous task in an asyncio program.

Now that we know how to execute blocking calls in an asyncio program, let's look at some worked examples.

Example of Running I/O-Bound Task in Asyncio with to_thread()

We can explore how to execute a blocking IO-bound call in an asyncio program using **asyncio.to_thread()**.

In this example, we will define a function that blocks the caller for a few seconds. We will then execute this function asynchronously in a thread pool from asyncio using the **asyncio.to_thread()** function.

This will free the caller to continue with other activities.

The complete example is listed below.

```python
# SuperFastPython.com
# example of running a blocking io-bound task in asyncio
import asyncio
import time

# a blocking io-bound task
def blocking_task():
    # report a message
    print('Task starting')
    # block for a while
    time.sleep(2)
    # report a message
    print('Task done')

# main coroutine
async def main():
    # report a message
    print('Main running the blocking task')
    # create a coroutine for  the blocking task
    coro = asyncio.to_thread(blocking_task)
    # schedule the task
    task = asyncio.create_task(coro)
    # report a message
    print('Main doing other things')
    # allow the scheduled task to start
    await asyncio.sleep(0)
    # await the task
    await task

# run the asyncio program
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 issues a call to the blocking function call to the thread pool. This returns a coroutine,

The coroutine is then wrapped in a **Task** and executed independently.

The **main()** coroutine is free to continue with other activities. In this case, it sleeps for a moment to allow the scheduled task to start executing. This allows the target function to be issued to the **ThreadPoolExecutor** behind the scenes and start running.

The **main()** coroutine then suspends and waits for the task to complete.

The blocking function reports a message, sleeps for 2 seconds, then reports a final message.

This highlights how we can execute a blocking IO-bound task in a separate thread asynchronously from an asyncio program.

```
Main running the blocking task
Main doing other things
Task starting
Task done
```

Next, we will explore how to develop and use asynchronous iterators.

Asynchronous Iterators

Iteration is a basic operation in Python.

We can iterate lists, strings, and all manner of other structures.

Asyncio allows us to develop asynchronous iterators.

We can create and use asynchronous iterators in asyncio programs by defining an object that implements the **__aiter__()** and **__anext__()** methods.

Let’s take a closer look.

What Are Asynchronous Iterators

An asynchronous iterator is an object that implements the **__aiter__()** and **__anext__()** methods.

Before we take a close look at asynchronous iterators, let's review classical iterators.

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 (HTTPS://DOCS.PYTHON.ORG/3/GLOSSARY.HTML)**

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 containers such as lists.

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 (HTTPS://DOCS.PYTHON.ORG/3/GLOSSARY.HTML)**

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.

Asynchronous iterators were introduced in PEP 492 – Coroutines with async and await syntax (https://peps.python.org/pep-0492/).

An asynchronous iterator can be stepped using the **anext()** built-in function (https://docs.python.org/3/library/functions.html#anext) 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.

> *An asynchronous iterable is able to call asynchronous code in its iter implementation, and asynchronous iterator can call asynchronous code in its next method.*
>
> — **PEP 492 – COROUTINES WITH ASYNC AND AWAIT SYNTAX (HTTPS://PEPS.PYTHON.ORG/PEP-0492/)**

What is the "async for" loop?

The async for expression is used to traverse an asynchronous iterator.

It is an asynchronous for-loop statement.

An asynchronous iterator is an iterator that yields awaitables.

You may recall that an awaitable is an object that can be waited for, such as a coroutine or a task.

“*awaitable: An object that can be used in an await expression.*

— **PYTHON GLOSSARY (HTTPS://DOCS.PYTHON.ORG/3/GLOSSARY.HTML)**

An asynchronous generator will automatically implement the asynchronous iterator methods, allowing it to be iterated like an asynchronous iterator.

The await for expression allows the caller to traverse an asynchronous iterator of awaitables and retrieve the result from each.

This is not the same as traversing a collection or list of awaitables (e.g. coroutine objects), instead, the awaitables returned must be provided using the expected asynchronous iterator methods.

Internally, the async for loop will automatically resolve or await each awaitable, scheduling coroutines as needed.

Because it is a for-loop, it assumes, although does not require, that each awaitable being traversed yields a return value.

The async for loop must be used within a coroutine because internally it will use the await expression, which can only be used within coroutines.

The async for expression can be used to traverse an asynchronous iterator within a coroutine.

For example:

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

This does not execute the for-loop in parallel. The asyncio is unable to execute more than one coroutine at a time within a Python thread.

Instead, this is an asynchronous for-loop.

The difference is that the coroutine that executes the for loop will suspend and internally await for each awaitable.

Behind the scenes, this may require coroutines to be scheduled and awaited, or tasks to be awaited.

We may also use the async for expression in a list comprehension.

For example:

```
...
# build a list of results
results = [item async for item async_iterator]
```

This would construct a list of return values from the asynchronous iterator.

Next, let's look at how to define, create and use asynchronous iterators.

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.

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.

> “*object.__anext__(self): Must return an awaitable resulting in a next value of the iterator. Should raise a StopAsyncIteration error when the iteration is over.*
>
> — **ASYNCHRONOUS ITERATORS (HTTPS://DOCS.PYTHON.ORG/3/REFERENCE/DATAMODEL.HTML#ASYNCHRONOUS-ITERATORS)**

When the iteration is complete, the **__anext__()** method must raise a **StopAsyncIteration** exception.

For example:

```python
# 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:

```python
...
# 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.

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*".

Step an Asynchronous Iterator

One step of the iterator can be traversed using the **anext()** built-in function (https://docs.python.org/3/library/functions.html#anext), just like a classical iterator using the **next()** function.

The result is an awaitable that is awaited.

For example:

```
...
# get an awaitable for one step of the iterator
awaitable = anext(it)
# execute the 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)
```

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():
    print(result)
```

You can learn more about the “**async for**” expression in the tutorial:

- Asyncio async for loop (/asyncio-async-for/)

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()]
```

Example of an Asynchronous Iterator

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.

```python
# SuperFastPython.com
# example of an asynchronous iterator with async for loop
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)

# execute the asyncio program
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, we will explore asynchronous generators.

Asynchronous Generators

Generators are a fundamental part of Python.

A generator is a function that has at least one “**yield**” expression. They are functions that can be suspended and resumed, just like coroutines.

In fact, Python coroutines are an extension of Python generators.

Asyncio allows us to develop asynchronous generators.

We can create an asynchronous generator by defining a coroutine that makes use of the “**yield**” expression.

Let’s take a closer look.

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.

Generators

A generator is a Python function that returns a value via a yield expression.

For example:

```python
# 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 (HTTPS://DOCS.PYTHON.ORG/3/GLOSSARY.HTML)**

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 (HTTPS://DOCS.PYTHON.ORG/3/GLOSSARY.HTML)**

A generator can be executed in steps by using the **next()** built-in function (https://docs.python.org/3/library/functions.html#next).

For example:

```python
...
# 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.

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 (HTTPS://DOCS.PYTHON.ORG/3/GLOSSARY.HTML)**

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 (HTTPS://DOCS.PYTHON.ORG/3/GLOSSARY.HTML)**

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.

You can learn more about the “**async for**” expression in the tutorial:

- Asyncio async for loop (/asyncio-async-for/)

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.

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:

```python
# 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:

```python
# 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.

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.

Step an Asynchronous Generator

One step of the generator can be traversed using the **anext()** built-in function (https://docs.python.org/3/library/functions.html#anext), 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 the 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)
```

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)
```

You can learn more about the “**async for**” expression in the tutorial:

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()]
```

Example of an Asynchronous Generator

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)

# execute the asyncio program
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, we will explore asynchronous context managers.

Asynchronous Context Managers

A context manager is a Python construct that provides a try-finally like environment with a consistent interface and handy syntax, e.g. via the “with” expression.

It is commonly used with resources, ensuring the resource is always closed or released after we are finished with it, regardless of whether the usage of the resources was successful or failed with an exception.

Asyncio allows us to develop asynchronous context managers.

We can create and use asynchronous context managers in asyncio programs by defining an object that implements the **__aenter__()** and **__aexit__()** methods as coroutines.

Let's take a closer look.

What is an Asynchronous Context Manager

An asynchronous context manager is a Python object that implements the **__aenter__()** and **__aexit__()** methods.

Before we dive into the details of asynchronous context managers, let's review classical context managers.

Context Manager

A context manager is a Python object that implements the **__enter__()** and **__exit__()** methods.

> *A context manager is an object that defines the runtime context to be established when executing a with statement. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code.*
>
> — **WITH STATEMENT CONTEXT MANAGERS (HTTPS://DOCS.PYTHON.ORG/3/REFERENCE/DATAMODEL.HTML#CONTEXT-MANAGERS)**

- 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.

> *Typical uses of context managers include saving and restoring various kinds of global state, locking and unlocking resources, closing opened files, etc.*
>
> — **WITH STATEMENT CONTEXT MANAGERS (HTTPS://DOCS.PYTHON.ORG/3/REFERENCE/DATAMODEL.HTML#CONTEXT-MANAGERS)**

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 mirrors a try-finally expression.

For example:

```
...
# create the object
manager = ContextManager()
try:
    manager.__enter__()
    # ...
finally:
    manager.__exit__()
```

Next, let's take a look at asynchronous context managers.

Asynchronous Context Manager

Asynchronous context managers were introduced in "PEP 492 – Coroutines with async and await syntax (https://peps.python.org/pep-0492/)".

They provide a context manager that can be suspended when entering and exiting.

> *"An asynchronous context manager is a context manager that is able to suspend execution in its __aenter__ and __aexit__ methods.*
>
> — **ASYNCHRONOUS CONTEXT MANAGERS (HTTPS://DOCS.PYTHON.ORG/3/REFERENCE/DATAMODEL.HTML#ASYNCHRONOUS-CONTEXT-MANAGERS)**

The __**aenter**__ and __**aexit**__ methods are defined as coroutines and are awaited by the caller.

This is achieved using the “**async with**” expression.

You can learn more about the “async with” expression in the tutorial:

- What is Asyncio async with (/asyncio-async-with/)

As such, asynchronous context managers can only be used within asyncio programs, such as within calling coroutines.

What is “async with”

The “**async with**” expression is for creating and using asynchronous context managers.

It is an extension of the “**with**” expression for use in coroutines within asyncio programs.

The “**async with**” expression is just like the “**with**” expression used for context managers, except it allows asynchronous context managers to be used within coroutines.

In order to better understand “**async with**“, let’s take a closer look at asynchronous context managers.

The async with expression allows a coroutine to create and use an asynchronous version of a context manager.

For example:

```
...
# create and use an asynchronous context manager
async with AsyncContextManager() as manager:
    # ...
```

This is equivalent to something like:

```
...
# create or enter the async context manager
manager = await AsyncContextManager()
try:
    # ...
finally:
    # close or exit the context manager
    await manager.close()
```

Notice that we are implementing much the same pattern as a traditional context manager, except that creating and closing the context manager involve awaiting coroutines.

This suspends the execution of the current coroutine, schedules a new coroutine and waits for it to complete.

As such an asynchronous context manager must implement the **__aenter__()** and **__aexit__()** methods that must be defined via the async def expression. This makes them coroutines themselves which may also await.

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.

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**" 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:

```python
# 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)
```

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:

```python
...
# 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 a worked example.

Example of an Asynchronous Context Manager and "async with"

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 asynchronous 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 program
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 will explore asynchronous comprehensions.

Asynchronous Comprehensions

Comprehensions, like list and dict comprehensions are one feature of Python when we think of “*pythonic*“.

It is a way we do loops that is different to many other languages.

Asyncio allows us to use asynchronous comprehensions.

We can traverse an asynchronous generators and asynchronous iterators using an asynchronous comprehension via the “**async for**” expression.

Let’s take a closer look.

What are Asynchronous Comprehensions

An async comprehension is an asynchronous version of a classical comprehension.

Asyncio supports two types of asynchronous comprehensions, they are the “**async for**” comprehension and the “**await**” comprehension.

> *PEP 530 adds support for using async for in list, set, dict comprehensions and generator expressions*
>
> — **PEP 530: ASYNCHRONOUS COMPREHENSIONS, WHAT'S NEW IN PYTHON 3.6 (HTTPS://DOCS.PYTHON.ORG/3/WHATSNEW/3.6.HTML#PEP-530-ASYNCHRONOUS-COMPREHENSIONS).**

Before we look at each, let's first recall classical comprehensions.

Comprehensions

Comprehensions allow data collections like lists, dicts, and sets to be created in a concise way.

> *List comprehensions provide a concise way to create lists.*
>
> — **LIST COMPREHENSIONS (HTTPS://DOCS.PYTHON.ORG/3/TUTORIAL/DATASTRUCTURES.HTML#LIST-COMPREHENSIONS)**

A list comprehension allows a list to be created from a for expression within the new list expression.

For example:

```
...
# create a list using a list comprehension
result = [a*2 for a in range(100)]
```

Comprehensions are also supported for creating dicts and sets.

For example:

```
...
# create a dict using a comprehension
result = {a:i for a,i in zip(['a','b','c'],range(3))}
# create a set using a comprehension
result = {a for a in [1, 2, 3, 2, 3, 1, 5, 4]}
```

Asynchronous Comprehensions

An asynchronous comprehension allows a list, set, or dict to be created using the "**async for**" expression with an asynchronous iterable.

> "*We propose to allow using async for inside list, set and dict comprehensions.*
>
> — PEP 530 – ASYNCHRONOUS COMPREHENSIONS (HTTPS://PEPS.PYTHON.ORG/PEP-0530/)

For example:

```
...
# async list comprehension with an async iterator
result = [a async for a in aiterable]
```

This will create and schedule coroutines or tasks as needed and yield their results into a list.

Recall that the "**async for**" expression (/asyncio-async-for/) may only be used within coroutines and tasks.

Also, recall that an asynchronous iterator is an iterator that yields awaitables.

The "**async for**" expression allows the caller to traverse an asynchronous iterator of awaitables and retrieve the result from each.

Internally, the async for loop will automatically resolve or await each awaitable, scheduling coroutines as needed.

An async generator automatically implements the methods for the async iterator and may also be used in an asynchronous comprehension.

For example:

```
...
# async list comprehension with an async generator
result = [a async for a in agenerator]
```

Await Comprehensions

The "**await**" expression may also be used within a list, set, or dict comprehension, referred to as an await comprehension.

“We propose to allow the use of await expressions in both asynchronous and synchronous comprehensions

— **PEP 530 – ASYNCHRONOUS COMPREHENSIONS (HTTPS://PEPS.PYTHON.ORG/PEP-0530/)**

Like an async comprehension, it may only be used within an asyncio coroutine or task.

This allows a data structure, like a list, to be created by suspending and awaiting a series of awaitables.

For example:

```
...
# await list compression with a collection of awaitables
results = [await a for a in awaitables]
```

This will create a list of results by awaiting each awaitable in turn.

The current coroutine will be suspended to execute awaitables sequentially, which is different and perhaps slower than executing them concurrently using **asyncio.gather()**.

Next, we will explore how to run commands using subprocesses from asyncio.

Run Commands in Non-Blocking Subprocesses

We can execute commands from asyncio.

The command will run in a subprocess that we can write to and read from using non-blocking I/O.

Let’s take a closer look.

What is asyncio.subprocess.Process

The **asyncio.subprocess.Process** class (https://docs.python.org/3/library/asyncio-subprocess.html#asyncio.subprocess.Process) provides a representation of a subprocess run by asyncio.

It provides a handle on a subprocess in asyncio programs, allowing actions to be performed on it, such as waiting and terminating it.

> “*Process is a high-level wrapper that allows communicating with subprocesses and watching for their completion.*
>
> — **INTERACTING WITH SUBPROCESSES (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-SUBPROCESS.HTML#INTERACTING-WITH-SUBPROCESSES)**

The API is very similar to the **multiprocessing.Process** class (https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Process) and perhaps more so with the **subprocess.Popen** class (https://docs.python.org/3/library/subprocess.html#subprocess.Popen).

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 what the **asyncio.subprocess.Process** class is, let’s look at how we might use it in our asyncio programs.

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.

> “*An object that wraps OS processes created by the create_subprocess_exec() and create_subprocess_shell() functions.*
>
> — **INTERACTING WITH SUBPROCESSES (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-SUBPROCESS.HTML#INTERACTING-WITH-SUBPROCESSES)**

There are two ways to execute an external program as a subprocess and acquire a Process instance, they are:

- **asyncio.create_subprocess_exec()** for running commands directly.
- **asyncio.create_subprocess_shell()** for running commands via the shell.

Let’s look at examples of each in turn.

How to Run a Command Directly

A command (https://en.wikipedia.org/wiki/Command-line_interface) is a program executed on the command line (terminal or command prompt). It is another program that is run directly.

Common examples on Linux and macOS might be:

- ‘**ls**‘ to list the contents of a directory
- ‘**cat**‘ to report the content of a file
- ‘**date**‘ to report the date
- ‘**echo**‘ to report back a string
- ‘**sleep**‘ to sleep for a number of seconds

And so on.

We can execute a command from an asyncio program via the **create_subprocess_exec()** function.

The **asyncio.create_subprocess_exec()** function takes a command and executes it directly.

This is helpful as it allows the command to be executed in a subprocess and for asyncio coroutines to read, write, and wait for it.

“Because all asyncio subprocess functions are asynchronous and asyncio provides many tools to work with such functions, it is easy to execute and monitor multiple subprocesses in parallel.

— ASYNCIO SUBPROCESSES (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-SUBPROCESS.HTML)

Unlike the **asyncio.create_subprocess_shell()** function, the **asyncio.create_subprocess_exec()** will not execute the command using the shell.

This means that the capabilities provided by the shell, such as shell variables, scripting, and wildcards are not available when executing the command.

It also means that executing the command may be more secure as there is no opportunity for a shell injection (https://en.wikipedia.org/wiki/Code_injection#Shell_injection).

Now that we know what **asyncio.create_subprocess_exec()** does, let’s look at how to use it.

How to Use Asyncio create_subprocess_exec()

The **asyncio.create_subprocess_exec()** function (https://docs.python.org/3/library/asyncio-subprocess.html#asyncio.create_subprocess_exec) will execute a given string command in a subprocess.

It returns a **asyncio.subprocess.Process** object (https://docs.python.org/3/library/asyncio-subprocess.html#asyncio.subprocess.Process) that represents the subprocess.

“Process is a high-level wrapper that allows communicating with subprocesses and watching for their completion.

— INTERACTING WITH SUBPROCESSES (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-SUBPROCESS.HTML#ASYNCIO.SUBPROCESS.PROCESS)

The **create_subprocess_exec()** function is a coroutine, which means we must await it. It will return once the subprocess has been started, not when the subprocess is finished.

For example:

```
...
# execute a command in a subprocess
process = await asyncio.create_subprocess_exec('ls')
```

Arguments to the command being executed must be provided as subsequent arguments to the **create_subprocess_exec()** function.

For example:

```
...
# execute a command with arguments in a subprocess
process = await asyncio.create_subprocess_exec('ls', '-l')
```

We can wait for the subprocess to finish by awaiting the **wait()** method.

For example:

```
...
# wait for the subprocess to terminate
await process.wait()
```

We can stop the subprocess directly by calling the **terminate()** or **kill()** methods, which will raise a signal in the subprocess.

For example:

```
...
# terminate the subprocess
process.terminate()
```

The input and output of the command will be handled by **stdin**, **stderr**, and **stdout**.

We can have the asyncio program handle the input or output for the subprocess.

This can be achieved by specifying the input or output stream and specifying a constant to redirect, such as asyncio.**subprocess.PIPE**.

For example, we can redirect the output of a command to the asyncio program:

```
...
# start a subprocess and redirect output
process = await asyncio.create_subprocess_exec('ls', stdout=asyncio.subprocess.PIPE)
```

We can then read the output of the program via the **asyncio.subprocess.Process** instance via the **communicate()** method.

This method is a coroutine and must be awaited. It is used to both send and receive data with the subprocess.

For example:

```
...
# read data from the subprocess
line = process.communicate()
```

We can also send data to the subprocess via the **communicate()** method by setting the "**input**" argument in bytes.

For example:

```
...
# start a subprocess and redirect input
process = await asyncio.create_subprocess_exec('ls', stdin=asyncio.subprocess.PIPE)
# send data to the subprocess
process.communicate(input=b'Hello\n')
```

Behind the scenes the **asyncio.subprocess.PIPE** configures the subprocess to point to a **StreamReader** or **StreamWriter** for sending data to or from the subprocess, and the **communicate()** method will read or write bytes from the configured reader.

> *"If PIPE is passed to stdin argument, the Process.stdin attribute will point to a StreamWriter instance. If PIPE is passed to stdout or stderr arguments, the Process.stdout and Process.stderr attributes will point to StreamReader instances.*
>
> — **ASYNCIO SUBPROCESSES (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-SUBPROCESS.HTML)**

We can interact with the StreamReader or StreamWriter directly via the subprocess via the stdin, stdout, and stderr attributes.

For example:

```
...
# read a line from the subprocess output stream
line = await process.stdout.readline()
```

Now that we know how to use the **create_subprocess_exec()** function, let's look at some worked examples.

Example of Asyncio create_subprocess_exec()

We can explore how to run a command in a subprocess from asyncio.

In this example, we will execute the "**echo**" command (https://en.wikipedia.org/wiki/Echo_(command)) to report back a string.

The echo command will report the provided string on standard output directly.

The complete example is listed below.

Note, this example assumes you have access to the "**echo**" command, I'm not sure it will work on Windows.

```python
# SuperFastPython.com
# example of executing a command as a subprocess with asyncio
import asyncio

# main coroutine
async def main():
    # start executing a command in a subprocess
    process = await asyncio.create_subprocess_exec('echo', 'Hello World')
    # report the details of the subprocess
    print(f'subprocess: {process}')

# entry point
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 calls the **create_subprocess_exec()** function to execute a command.

The **main()** coroutine suspends while the subprocess is created. A Process instance is returned.

The **main()** coroutine resumes and reports the details of the subprocess. The **main()** process terminates and the asyncio program terminates.

The output of the echo command is reported on the command line.

This highlights how we can execute a command from an asyncio program.

```
Hello World
subprocess: <Process 50249>
```

How to Run a Command Via the Shell

We can execute commands using the shell (https://en.wikipedia.org/wiki/Shell_(computing)).

The shell is a user interface for the command line, called a command line interpreter (CLI).

It will interpret and execute commands on behalf of the user.

It also offers features such as a primitive programming language for scripting, wildcards, piping, shell variables (e.g. PATH), and more.

For example, we can redirect the output of one command as input to another command, such as the contents of the “**/etc/services**” file into the word count “**wc**” command and count the number of lines:

```
cat /etc/services | wc -l
```

Examples of shells in the Unix based operating systems include:

- ‘sh’
- ‘bash’
- ‘zsh’
- And so on.

On Windows, the shell is probably cmd.exe (https://en.wikipedia.org/wiki/Cmd.exe).

See this great list of command line shells:

- List of command-line interpreters, Wikipedia (https://en.wikipedia.org/wiki/List_of_command-line_interpreters)

The shell is already running, it was used to start the Python program.

You don’t need to do anything special to get or have access to the shell.

We can execute a command from an asyncio program via the **create_subprocess_shell()** function.

The **asyncio.create_subprocess_shell()** function takes a command and executes it using the current user shell.

This is helpful as it not only allows the command to be executed, but allows the capabilities of the shell to be used, such as redirection, wildcards and more.

> “*… the specified command will be executed through the shell. This can be useful if you are using Python primarily for the enhanced control flow it offers over most system shells and still want convenient access to other shell features such as shell pipes, filename wildcards, environment variable expansion, and expansion of ~ to a user’s home directory.*
>
> — **SUBPROCESS — SUBPROCESS MANAGEMENT (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/SUBPROCESS.HTML)**

The command will be executed in a subprocess of the process executing the asyncio program.

Importantly, the asyncio program is able to interact with the subprocess asynchronously, e.g. via coroutines.

> “*Because all asyncio subprocess functions are asynchronous and asyncio provides many tools to work with such functions, it is easy to execute and monitor multiple subprocesses in parallel.*
>
> — **ASYNCIO SUBPROCESSES (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-SUBPROCESS.HTML)**

There can be security considerations when executing a command via the shell instead of directly.

This is because there is at least one level of indirection and interpretation between the request to execute the command and the command being executed, allowing possible malicious injection.

> “*Important It is the application’s responsibility to ensure that all whitespace and special characters are quoted appropriately to avoid shell injection vulnerabilities.*
>
> — **ASYNCIO SUBPROCESSES (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-SUBPROCESS.HTML)**

Now that we know what **asyncio.create_subprocess_shell()** does, let’s look at how to use it.

How to Use Asyncio create_subprocess_shell()

The **asyncio.create_subprocess_shell()** function (https://docs.python.org/3/library/asyncio-subprocess.html#asyncio.create_subprocess_shell) will execute a given string command via the current shell.

It returns a **asyncio.subprocess.Process** object (https://docs.python.org/3/library/asyncio-subprocess.html#asyncio.subprocess.Process) that represents the process.

It is very similar to the **create_subprocess_shell()** function we saw in a previous section. Nevertheless, we will review how to use the function and interact with the process via the Process instance (in case you skipped straight to this section).

The **create_subprocess_shell()** function is a coroutine, which means we must await it. It will return once the subprocess has been started, not when the subprocess is finished.

For example:

```python
...
# start a subprocess
process = await asyncio.create_subprocess_shell('ls')
```

We can wait for the subprocess to finish by awaiting the **wait()** method.

For example:

```
...
# wait for the subprocess to terminate
await process.wait()
```

We can stop the subprocess directly by calling the **terminate()** or **kill()** methods, which will raise a signal in the subprocess.

The input and output of the command will be handled by the shell, e.g. **stdin**, **stderr**, and **stdout**.

We can have the asyncio program handle the input or output for the subprocess.

This can be achieved by specifying the input or output stream and specifying a constant to redirect, such as **asyncio.subprocess.PIPE**.

For example, we can redirect the output of a command to the asyncio program:

```
...
# start a subprocess and redirect output
process = await asyncio.create_subprocess_shell('ls', stdout=asyncio.subprocess.PIPE)
```

We can then read the output of the program via the **asyncio.subprocess.Process** instance via the **communicate()** method.

This method is a coroutine and must be awaited. It is used to both send and receive data with the subprocess.

For example:

```
...
# read data from the subprocess
line = process.communicate()
```

We can also send data to the subprocess via the **communicate()** method by setting the “input” argument in bytes.

For example:

```
...
# start a subprocess and redirect input
process = await asyncio.create_subprocess_shell('ls', stdin=asyncio.subprocess.PIPE)
# send data to the subprocess
process.communicate(input=b'Hello\n')
```

Behind the scenes the **asyncio.subprocess.PIPE** configures the subprocess to point to a **StreamReader** or **StreamWriter** for sending data to or from the subprocess, and the **communicate()** method will read or write bytes from the configured reader.

> *"If PIPE is passed to stdin argument, the Process.stdin attribute will point to a StreamWriter instance. If PIPE is passed to stdout or stderr arguments, the Process.stdout and Process.stderr attributes will point to StreamReader instances.*
>
> — **ASYNCIO SUBPROCESSES (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-SUBPROCESS.HTML)**

We can interact with the **StreamReader** or **StreamWriter** directly via the subprocess via the stdin, stdout, and stderr attributes.

For example:

```
...
# read a line from the subprocess output stream
line = await process.stdout.readline()
```

Now that we know how to use the **create_subprocess_shell()** function, let's look at some worked examples.

Example of Asyncio create_subprocess_shell()

We can explore how to run a command in a subprocess from asyncio using the shell.

In this example, we will execute the "**echo**" command (https://en.wikipedia.org/wiki/Echo_(command)) to report back a string.

The echo command will report the provided string on standard output directly.

The complete example is listed below.

Note, this example assumes you have access to the "**echo**" command, I'm not sure it will work on Windows.

```python
# SuperFastPython.com
# example of executing a shell command as a subprocess with asyncio
import asyncio

# main coroutine
async def main():
    # start executing a shell command in a subprocess
    process = await asyncio.create_subprocess_shell('echo Hello World')
    # report the details of the subprocess
    print(f'subprocess: {process}')

# entry point
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 calls the **create_subprocess_shell()** function to execute a command.

The **main()** coroutine suspends while the subprocess is created. A **Process** instance is returned.

The **main()** coroutine resumes and reports the details of the subprocess. The **main()** process terminates and the asyncio program terminates.

The output of the echo command is reported on the command line.

This highlights how we can execute a command using the shell from an asyncio program.

```
subprocess: <Process 43916>
Hello World
```

Non-Blocking Streams

A major benefit of asyncio is the ability to use non-blocking streams.

Let’s take a closer look.

Asyncio Streams

Asyncio provides non-blocking I/O socket programming.

This is provided via streams.

> “*Streams are high-level async/await-ready primitives to work with network connections. Streams allow sending and receiving data without using callbacks or low-level protocols and transports.*
>
> — **ASYNCIO STREAMS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-STREAM.HTML)**

Sockets can be opened that provide access to a stream writer and a stream writer.

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.

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 your own application-specific protocol.

Now that we know what asyncio streams are, let’s look at how to use them.

How to Open a Connection

An asyncio TCP client socket connection can be opened using the **asyncio.open_connection()** function (https://docs.python.org/3/library/asyncio-stream.html#asyncio.open_connection).

> “*Establish a network connection and return a pair of (reader, writer) objects. The returned reader and writer objects are instances of StreamReader and StreamWriter classes.*
>
> — **ASYNCIO STREAMS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-STREAM.HTML)**

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)
```

How to Start a Server

An asyncio TCP server socket can be opened using the **asyncio.start_server()** function (https://docs.python.org/3/library/asyncio-stream.html#asyncio.start_server).

> *Create a TCP server (socket type SOCK_STREAM) listening on port of the host address.*
>
> — **ASYNCIO EVENT LOOP (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-EVENTLOOP.HTML)**

This is a coroutine that must be awaited.

The function returns an **asyncio.Server** object (https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.Server) that represents the running server.

For example:

```
...
# start a tcp server
server = await asyncio.start_server(...)
```

The three required arguments are the callback function, the host, and the port.

The callback function is a custom function specified by name that will be called each time a client connects to the server.

> *The client_connected_cb callback is called whenever a new client connection is established. It receives a (reader, writer) pair as two arguments, instances of the StreamReader and StreamWriter classes.*
>
> — **ASYNCIO STREAMS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-STREAM.HTML)**

The host is the domain name or IP address that clients will specify to connect. The port is the socket port number on which to receive connections, such as 21 for FTP or 80 for HTTP.

For example:

```
# handle connections
async def handler(reader, writer):
    # ...

...
# start a server to receive http connections
server = await asyncio.start_server(handler, '127.0.0.1', 80)
```

How to Write Data with the StreamWriter

We can write data to the socket using an **asyncio.StreamWriter (https://docs.python.org/3/library/asyncio-stream.html#streamwriter)**.

> *Represents a writer object that provides APIs to write data to the IO stream.*
>
> — **ASYNCIO STREAMS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-STREAM.HTML)**

Data is written as bytes.

Byte data can be written to the socket using the **write()** method (https://docs.python.org/3/library/asyncio-stream.html#asyncio.StreamWriter.write).

> *The method attempts to write the data to the underlying socket immediately. If that fails, the data is queued in an internal write buffer until it can be sent.*
>
> — **ASYNCIO STREAMS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-STREAM.HTML)**

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 (https://docs.python.org/3/library/asyncio-stream.html#asyncio.StreamWriter.writelines).

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 drain the socket via the **drain()** method (https://docs.python.org/3/library/asyncio-stream.html#asyncio.StreamWriter.drain).

> *Wait until it is appropriate to resume writing to the stream.*
>
> — **ASYNCIO STREAMS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-STREAM.HTML)**

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()
```

How to Read Data with the StreamReader

We can read data from the socket using an **asyncio.StreamReader (https://docs.python.org/3/library/asyncio-stream.html#streamreader)**.

> *Represents a reader object that provides APIs to read data from the IO stream.*
>
> — **ASYNCIO STREAMS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-STREAM.HTML)**

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).

```
...
# read byte data
byte_data = await reader.read()
```

Additionally, the number of bytes to read can be specified via the “**n**” argument.

> *Read up to n bytes. If n is not provided, or set to -1, read until EOF and return all read bytes.*
>
> — **ASYNCIO STREAMS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-STREAM.HTML)**

This may be helpful if you 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 EOF.

> *Read one line, where “line” is a sequence of bytes ending with \n. If EOF is received and \n was not found, the method returns partially read data. If EOF is received and the internal buffer is empty, return an empty bytes object.*
>
> — **ASYNCIO STREAMS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-STREAM.HTML)**

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.

How to Close Connection

The socket can be closed via the **asyncio.StreamWriter**.

The **close()** method can be called which will close the socket.

> *The method closes the stream and the underlying socket.*
>
> — **ASYNCIO STREAMS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-STREAM.HTML)**

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.

> *Wait until the stream is closed. Should be called after close() to wait until the underlying connection is closed.*
>
> — **ASYNCIO STREAMS (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO-STREAM.HTML)**

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():
    # ...
```

Now that we know how to use asyncio streams, let's look at a worked example.

Example of Checking Website Status

We can query the HTTP status of websites using asyncio by opening a stream and writing and reading HTTP requests and responses.

We can then use asyncio to query the status of many websites concurrently, and even report the results dynamically.

Let's get started.

How to Check HTTP Status with Asyncio

The asyncio module provides support for opening socket connections and reading and writing data via streams.

We can use this capability to check the status of web pages.

This involves perhaps four steps, they are:

1. Open a connection
2. Write a request

3. Read a response
4. Close the connection

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

Open HTTP Connection

A connection can be opened in asyncio using the **asyncio.open_connection()** function (https://docs.python.org/3/library/asyncio-stream.html#asyncio.open_connection).

Among many arguments, the function takes the string hostname 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.

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)
```

Write HTTP Request

Once open, we can write a query to the **StreamWriter** to make an HTTP request.

For example, an HTTP version 1.1 request (https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) 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'
```

You can learn more about HTTP v1.1 request messages here:

- HTTP/1.1 request messages (https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#HTTP/1.1_request_messages)

This string must be encoded as bytes before being written to the **StreamWriter (https://docs.python.org/3/library/asyncio-stream.html#streamwriter)**.

This can be achieved using the **encode()** method (https://docs.python.org/3/library/stdtypes.html#str.encode) on the string itself.

The default '**utf-8**' encoding may be sufficient.

For example:

```
...
# encode string as bytes
byte_data = string.encode()
```

You can see a listing of encodings here:

- Python Standard Encodings (https://docs.python.org/3/library/codecs.html#standard-encodings)

The bytes can then be written to the socket via the **StreamWriter** via the **write()** method (https://docs.python.org/3/library/asyncio-stream.html#asyncio.StreamWriter.write).

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 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()
```

Read HTTP Response

Once the HTTP request has been made, we can read the response.

This can be achieved via the **StreamReader (https://docs.python.org/3/library/asyncio-stream.html#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 (https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#HTTP/1.1_response_messages) 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.

You can learn more about HTTP v1.1 responses here:

- HTTP/1.1 response messages (https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#HTTP/1.1_response_messages)

Each line must be decoded from bytes into a string.

This can be achieved using the **decode()** method (https://docs.python.org/3/library/stdtypes.html#bytes.decode) on the byte data. Again, the default encoding is '**utf_8**'.

For example:

```
...
# decode bytes into a string
line_data = line_bytes.decode()
```

Close HTTP Connection

We can close the socket connection by closing the **StreamWriter**.

This can be achieved by calling the **close()** method (https://docs.python.org/3/library/asyncio-stream.html#asyncio.StreamWriter.close).

For example:

```
...
# close the connection
writer.close()
```

This does not block and may not close the socket immediately.

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.

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 (https://en.wikipedia.org/wiki/List_of_most_visited_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 (https://docs.python.org/3/library/urllib.parse.html) 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**.

```python
...
# send GET request
query = f'GET {url_parsed.path} HTTP/1.1\r\nHost: {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.

```python
...
# read the single line response
response = await reader.readline()
```

The connection can then be closed.

```python
...
# close the connection
writer.close()
```

Finally, we can decode the bytes read from the server, remote trailing white space, and return the HTTP status.

```python
...
# 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 for the reader.

```
# 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\nHost: {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.

```
...
# 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.

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

Tying this together, the complete example is listed below.

```
# SuperFastPython.com
# check the status of many webpages
import asyncio
from urllib.parse import urlsplit

# 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\nHost: {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}')

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

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 5.6 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 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 temporarily
https://yahoo.com/             : HTTP/1.1 301 Moved Permanently
https://www.whatsapp.com/      : HTTP/1.1 302 Found
```

Next, let’s look at how we might update the example to execute the coroutines concurrently.

Example of Checking Website Status Concurrently

A benefit of asyncio is that we can execute many coroutines concurrently.

We can query the status of websites concurrently in asyncio using the **asyncio.gather() function (https://docs.python.org/3/library/asyncio-task.html#asyncio.gather)**.

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.

```
...
# 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.

```python
# SuperFastPython.com
# check the status of many webpages
import asyncio
from urllib.parse import urlsplit

# 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\nHost: {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}')

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

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 1.4 seconds on my system.

```
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 temporarily
https://yahoo.com/            : HTTP/1.1 301 Moved Permanently
https://www.whatsapp.com/     : HTTP/1.1 302 Found
```

Next, let’s explore common errors when getting started with asyncio.

Python Asyncio Common Errors

This section gives examples of general errors encountered by developers when using asyncio in Python.

The 5 most common asyncio errors are:

1. Trying to run coroutines by calling them.
2. Not letting coroutines run in the event loop.
3. Using the asyncio low-level API.
4. Exiting the main coroutine too early.
5. Assuming race conditions and deadlocks are not possible.

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

Error 1: Trying to Run Coroutines by Calling Them

The most common error encountered by beginners to asyncio is calling a coroutine like a function.

For example, we can define a coroutine using the “async def” expression:

```python
# custom coroutine
async def custom_coro():
    print('hi there')
```

The beginner will then attempt to call this coroutine like a function and expect the print message to be reported.

For example:

```python
...
# error attempt at calling a coroutine like a function
custom_coro()
```

Calling a coroutine like a function will not execute the body of the coroutine.

Instead, it will create a coroutine object.

This object can then be awaited within the asyncio runtime, e.g. the event loop.

We can start the event loop to run the coroutine using the asyncio.run() function.

For example:

```python
...
# run a coroutine
asyncio.run(custom_coro())
```

Alternatively, we can suspend the current coroutine and schedule the other coroutine using the “await” expression.

For example:

```python
...
# schedule a coroutine
await custom_coro()
```

You can learn more about running coroutines in the tutorial:

- How to Run an Asyncio Coroutine in Python (/asyncio-run-coroutine)

Error 2: Not Letting Coroutines Run in the Event Loop

If a coroutine is not run, you will get a runtime warning as follows:

```
sys:1: RuntimeWarning: coroutine 'custom_coro' was never awaited
```

This will happen if you create a coroutine object but do not schedule it for execution within the asyncio event loop.

For example, you may attempt to call a coroutine from a regular Python program:

```
...
# attempt to call the coroutine
custom_coro()
```

This will not call the coroutine.

Instead, it will create a coroutine object.

For example:

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

If you do not allow this coroutine to run, you will get a runtime error.

You can let the coroutine run, as we saw in the previous section, by starting the asyncio event loop and passing it the coroutine object.

For example:

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

Or, on one line in a compound statement:

```
...
# run a coroutine
asyncio.run(custom_coro())
```

You can learn more about running coroutines in the tutorial:

- How to Run an Asyncio Coroutine in Python (/asyncio-run-coroutine)

If you get this error within an asyncio program, it is because you have created a coroutine and have not scheduled it for execution.

This can be achieved using the await expression.

For example:

```
...
# create a coroutine object
coro = custom_coro()
# suspend and allow the other coroutine to run
await coro
```

Or, you can schedule it to run independently as a task.

For example:

```
...
# create a coroutine object
coro = custom_coro()
# schedule the coro to run as a task interdependently
task = asyncio.create_task(coro)
```

You can learn more about creating tasks in the tutorial:

- How to Create an Asyncio Task in Python (/asyncio-create-task)

Error 3: Using the Low-Level Asyncio API

A big problem with beginners is that they use the wrong asyncio API.

This is common for a number of reasons.

- The API has changed a lot with recent versions of Python.
- The API docs page makes things confusing, showing both APIs.
- Examples elsewhere on the web mix up using the different APIs.

Using the wrong API makes things more verbose (e.g. more code), more difficult, and way less understandable.

Asyncio offers two APIs (https://docs.python.org/3/library/asyncio.html).

1. High-level API for application developers (us)

2. Low-level API for framework and library developers (not us)

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.

> *"… there are low-level APIs for library and framework developers*
>
> — **ASYNCIO — ASYNCHRONOUS I/O (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/ASYNCIO.HTML)**

We should almost always stick to the high-level API.

We absolutely must stick to the high-level API when getting started.

We may dip into the low-level API to achieve specific outcomes on occasion.

If you start getting a handle on the event loop or use a "loop" variable to do things, you are doing it wrong.

I am not saying don't learn the low-level API.

Go for it. It's great.

Just don't start there.

Drive asyncio via the high-level API for a while. Develop some programs. Get comfortable with asynchronous programming and running coroutines at will.

Then later, dip in and have a look around.

Error 4: Exiting the Main Coroutine Too Early

A major point of confusion in asyncio programs is not giving tasks enough time to complete.

We can schedule many coroutines to run independently within an asyncio program via the **asyncio.create_task()** method.

The main coroutine, the entry point for the asyncio program, can then carry on with other activities.

If the main coroutine exits, then the asyncio program will terminate.

The program will terminate even if there are one or many coroutines running independently as tasks.

This can catch you off guard.

You may issue many tasks and then allow the main coroutine to resume, expecting all issued tasks to complete in their own time.

Instead, if the main coroutine has nothing else to do, it should wait on the remaining tasks.

This can be achieved by first getting a set of all running tasks via the **asyncio.all_tasks()** function, removing itself from this set, then waiting on the remaining tasks via the **asyncio.wait()** function.

For example:

```
...
# get a set of all running tasks
all_tasks = asyncio.all_tasks()
# get the current tasks
current_task = asyncio.current_task()
# remove the current task from the list of all tasks
all_tasks.remove(current_task)
# suspend until all tasks are completed
await asyncio.wait(all_tasks)
```

Error 5: Assuming Race Conditions and Deadlocks are Impossible

Concurrent programming has the hazard of concurrency-specific failure modes.

This includes problems such as race conditions and deadlocks.

A race condition involves two or more units of concurrency executing the same critical section at the same time and leaving a resource or data in an inconsistent or unexpected state. This can lead to data corruption and data loss.

A deadlock is when a unit of concurrency waits for a condition that can never occur, such as for a resource to become available.

Many Python developers believe these problems are not possible with coroutines in asyncio.

The reason being that only one coroutine can run within the event loop at any one time.

It is true that only one coroutine can run at a time.

The problem is, coroutines can suspend and resume and may do so while using a shared resource or shared variable.

Without protecting critical sections, race conditions can occur in asyncio programs.

Without careful management of synchronization primitives, deadlocks can occur

As such, it is important that asyncio programs are created ensuring coroutine-safety, a concept similar to thread-safety and process-safety, applied to coroutines.

Python Asyncio Common Questions

This section answers common questions asked by developers when using asyncio in Python.

Do you have a question about asyncio?
Ask your question in the comments below and I will do my best to answer it and perhaps add it to this list of questions.

How to Stop a Task?

We can cancel a task via the **cancel()** method on an **asyncio.Task** object (https://docs.python.org/3/library/asyncio-task.html#asyncio.Task.cancel).

The **cancel()** method returns **True** if the task was canceled, or **False** otherwise.

For example:

```
...
# cancel the task
was_cancelled = task.cancel()
```

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.

The next time the task is given an opportunity to run, it will raise a **CancelledError** exception.

If the **CancelledError** exception is not handled within the wrapped coroutine, the task will be canceled.

Otherwise, if the **CancelledError** exception is handled within the wrapped coroutine, the task will not be canceled.

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

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 a moment.

The main coroutine then resumes and cancels the task while it is running. It waits a moment more to allow the task to respond to the request to cancel. The main coroutine then reports whether the request to cancel the task was successful.

The task is canceled and is then done.

The main coroutine then reports whether the status of the task is canceled before closing the program.

The complete example is listed below.

```python
# 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
    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 program
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 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 that 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 normal case of canceling a running task.

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

How to Wait for a Task To Finish?

We can wait for a task to finish by awaiting the **asyncio.Task** object directly.

For example:

```
...
# wait for the task to finish
await task
```

We may create and wait for the task in a single line.

For example:

```
...
# create and wait for the task to finish
await asyncio.create_task(custom_coro())
```

How to Get a Return Value from a Task?

We may need to return values from coroutines to the caller.

We can retrieve a return value from a coroutine by awaiting it.

It assumes that the other coroutine being awaited returns a value.

For example:

```
# coroutine that returns a value
async def other_coro():
    return 100
```

Awaiting the other coroutine will suspend the calling coroutine and schedule the other coroutine for execution. Once the other coroutine has been completed, the calling coroutine will resume. The return value will be passed from the other coroutine to the caller.

For example:

```
...
# execute coroutine and retrieve return value
value = await other_coro()
```

A coroutine can be wrapped in an **asyncio.Task** object.

This is helpful for independently executing the coroutine without having the current coroutine await it.

This can be achieved using the **asyncio.create_task()** function (https://docs.python.org/3/library/asyncio-task.html#asyncio.create_task).

For example:

```
...
# wrap coroutine in a task and schedule it for execution
task = asyncio.create_task(other_coro())
```

You can learn more about how to create tasks in the tutorial:

- How to Create an Asyncio Task in Python (/asyncio-create-task)

There are two ways to retrieve the return value from an **asyncio.Task**, they are:

1. Await the task.
2. Call the result() method.

We can await the task to retrieve the return value.

If the task is scheduled or running, then the caller will suspend until the task is complete and the return value will be provided.

If the task is completed, the return value will be provided immediately.

For example:

```
...
# get the return value from a task
value = await task
```

Unlike a coroutine, we can await a task more than once without raising an error.

For example:

```
...
# get the return value from a task
value = await task
# get the return value from a task
value = await task
```

We can also get the return value from the task by calling the **result()** method (https://docs.python.org/3/library/asyncio-task.html#asyncio.Task.result) on the **asyncio.Task** object.

For example:

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

This requires that the task is done. If not, an **InvalidStateError** exception will be raised.

If the task was canceled a **CancelledError** exception will be raised.

You can learn more about getting the result from tasks in the tutorial:

- How to Get Asyncio Task Results (/asyncio-task-result)

How to Run a Task in the Background?

We can run a coroutine in the background by wrapping it in an **asyncio.Task** object.

This can be achieved by calling the **asyncio.create_task()** function and passing it the coroutine.

The coroutine will be wrapped in a Task object and will be scheduled for execution. The task object will be returned and the caller will not suspend.

For example:

```
...
# schedule the task for execution
task = asyncio.create_task(other_coroutine())
```

The task will not begin executing until at least the current coroutine is suspended, for any reason.

We can help things along by suspending for a moment to allow the task to start running.

This can be achieved by sleeping for zero seconds.

For example:

```
...
# suspend for a moment to allow the task to start running
await asyncio.sleep(0)
```

This will suspend the caller only for a brief moment and allow the ask an opportunity to run.

This is not required as the caller may suspend at some future time or terminate as part of normal execution.

We may also await the task directly once the caller has run out of things to do.

For example:

```
...
# wait for the task to complete
await task
```

How to Wait for All Background Tasks?

We can wait for all independent tasks in an asyncio program.

This can be achieved by first getting a set of all currently running tasks via the **asyncio.all_tasks()** function.

For example:

```
...
# get a set of all running tasks
all_tasks = asyncio.all_tasks()
```

This will return a set that contains one **asyncio.Task** object for each task that is currently running, including the **main()** coroutine.

We cannot wait on this set directly, as it will block forever as it includes the task that is the current task.

Therefore we can get the **asyncio.Task** object for the currently running task and remove it from the set.

This can be achieved by first calling the **asyncio.current_task()** method to get the task for the current coroutine and then remove it from the set via the **remove()** method.

For example:

```
...
# get the current tasks
current_task = asyncio.current_task()
# remove the current task from the list of all tasks
all_tasks.remove(current_task)
```

Finally, we can wait on the set of remaining tasks.

This will suspend the caller until all tasks in the set are complete.

For example:

```
...
# suspend until all tasks are completed
await asyncio.wait(all_tasks)
```

Tying this together, the snippet below added to the end of the **main()** coroutine will wait for all background tasks to complete.

```
...
# get a set of all running tasks
all_tasks = asyncio.all_tasks()
# get the current tasks
current_task = asyncio.current_task()
# remove the current task from the list of all tasks
all_tasks.remove(current_task)
# suspend until all tasks are completed
await asyncio.wait(all_tasks)
```

Does a Running Task Stop the Event Loop from Exiting?

No.

A task that is scheduled and run independently will not stop the event loop from exiting.

If your main coroutine has no other activities to complete and there are independent tasks running in the background, you should retrieve the running tasks and wait on them

The previous question/answer shows exactly how to do this.

How to Show Progress of Running Tasks?

We can show progress using a done callback function on each task.

A done callback is a function that we can register on an **asyncio.Task**.

It is called once the task is done, either normally or if it fails.

The done callback function is a regular function, not a coroutine, and takes the **asyncio.Task** that it is associated with as an argument.

We can use the same callback function for all tasks and report progress in a general way, such as by reporting a message.

For example:

```python
# callback function to show progress of tasks
def progress(task):
    # report progress of the task
    print('.', end='')
```

We can register a callback function on each **asyncio.Task** that we issue.

This can be achieved using the **add_done_callback()** method (https://docs.python.org/3/library/asyncio-task.html#asyncio.Task.add_done_callback) on each task and passing it the name of the callback function.

For example:

```python
...
# add a done callback to a task
task.add_done_callback(progress)
```

How to Run a Task After a Delay?

We can develop a custom wrapper coroutine to execute a target coroutine after a delay.

The wrapper coroutine may take two arguments, a coroutine and a time in seconds.

It will sleep for the given delay interval in seconds, then await the provided coroutine.

The **delay()** coroutine below implements this.

```python
# coroutine that will start another coroutine after a delay in seconds
async def delay(coro, seconds):
    # suspend for a time limit in seconds
    await asyncio.sleep(seconds)
    # execute the other coroutine
    await coro
```

To use the wrapper coroutine, a coroutine object can be created and either awaited directly or executed independently as a task.

For example, the caller may suspend and schedule the delayed coroutine and wait for it to be done:

```python
...
# execute a coroutine after a delay
await delay(coro, 10)
```

Alternatively, the caller may schedule the delayed coroutine to run independently:

```python
...
# execute a coroutine after a delay independently
_ = asyncio.create_task(delay(coro, 10))
```

How to Run a Follow-Up Task?

There are three main ways to issue follow-up tasks in asyncio.

They are:

1. Schedule the follow-up task from the completed task itself.
2. Schedule the follow-up task from the caller.
3. Schedule the follow-up task automatically using a done callback.

Let’s take a closer look at each approach.

The task that is completed can issue its own follow-up task.

This may require checking some state in order to determine whether the follow-up task should be issued or not.

The task can then be scheduled via a call to **asyncio.create_task()**.

For example:

```
...
# schedule a follow-up task
task = asyncio.create_task(followup_task())
```

The task itself may choose to await the follow-up task or let it complete in the background independently.

For example:

```
...
# wait for the follow-up task to complete
await task
```

The caller that issued the task can choose to issue a follow-up task.

For example, when the caller issues the first task, it may keep the **asyncio.Task** object.

It can then check the result of the task or whether the task was completed successfully or not.

The caller can then decide to issue a follow-up task.

It may or may not await the follow-up task directly.

For example:

```
...
# issue and await the first task
task = await asyncio.create_task(task())
# check the result of the task
if task.result():
    # issue the follow-up task
    followup = await asyncio.create_task(followup_task())
```

We can execute a follow-up task automatically using a done callback function.

For example, the caller that issues the task can register a done callback function on the task itself.

The done callback function must take the **asyncio.Task** object as an argument and will be called only after the task is done. It can then choose to issue a follow-up task.

The done callback function is a regular Python function, not a coroutine, so it cannot await the follow-up task

For example, the callback function may look as follows:

```
# callback function
def callback(task):
    # schedule and await the follow-up task
    _ = asyncio.create_task(followup())
```

The caller can issue the first task and register the done callback function.

For example:

```
...
# schedule and the task
task = asyncio.create_task(work())
# add the done callback function
task.add_done_callback(callback)
```

How to Execute a Blocking I/O or CPU-bound Function in Asyncio?

The asyncio module provides two approaches for executing blocking calls in asyncio programs.

The first is to use the **asyncio.to_thread()** function (https://docs.python.org/3/library/asyncio-task.html#asyncio.to_thread).

This is in the high-level API and is intended for application developers.

The **asyncio.to_thread()** function takes a function name to execute and any arguments.

The function is executed in a separate thread. It returns a coroutine that can be awaited or scheduled as an independent task.

For example:

```
...
# execute a function in a separate thread
await asyncio.to_thread(task)
```

The task will not begin executing until the returned coroutine is given an opportunity to run in the event loop.

The **asyncio.to_thread()** function creates a **ThreadPoolExecutor** behind the scenes to execute blocking calls.

As such, the **asyncio.to_thread()** function is only appropriate for IO-bound tasks.

An alternative approach is to use the **loop.run_in_executor()** function (https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.loop.run_in_executor).

This is in the low-level asyncio API and first requires access to the event loop, such as via the **asyncio.get_running_loop()** function (https://docs.python.org/3/library/asyncio-eventloop.html#asyncio.get_running_loop).

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 **ThreadPoolExecutor**.

The **loop.run_in_executor()** function returns an awaitable that can be awaited if needed. 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:

```
...
# get the event loop
loop = asyncio.get_running_loop()
# execute a function in a separate thread
await loop.run_in_executor(None, task)
```

Alternatively, an executor can be created and passed to the **loop.run_in_executor()** function, which will execute the asynchronous call in the executor.

The caller must manage the executor in this case, shutting it down once the caller is finished with it.

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...
```

These two approaches allow a blocking call to be executed as an asynchronous task in an asyncio program.

Common Objections to Using Asyncio

Asyncio and coroutines may not be the best solution for all concurrency problems in your program.

That being said, there may also be some misunderstandings that are preventing you from making full and best use of the capabilities of the asyncio in Python.

In this section, we review some of the common objections seen by developers when considering using the asyncio.

What About the Global Interpreter Lock (GIL)?

The GIL protects the internals of the Python interpreter from concurrent access and modification from multiple threads.

The asyncio event loop runs in one thread.

This means that all coroutines run in a single thread.

As such the GIL is not an issue when using asyncio and coroutine.

Are Python Coroutines "*Real*"?

Coroutines are managed in software.

Coroutines run and are managed (switched) within the asyncio event loop in the Python runtime.

They are not a software representation of a capability provided by the underlying operating system, like threads and processes.

In this sense, Python does not have support for "native coroutines", but I'm not sure such things exist in modern operating systems.

Isn't Python Concurrency Buggy?

No.

Python provides first-class concurrency with coroutines, threads, and processes.

It has for a long time now and it is widely used in open source and commercial projects.

Isn’t Python a Bad Choice for Concurrency?

Developers love python for many reasons, most commonly because it is easy to use and fast for development.

Python is commonly used for glue code, one-off scripts, but more and more for large-scale software systems.

If you are using Python and then you need concurrency, then you work with what you have. The question is moot.

If you need concurrency and you have not chosen a language, perhaps another language would be more appropriate, or perhaps not. Consider the full scope of functional and non-functional requirements (or user needs, wants, and desires) for your project and the capabilities of different development platforms.

Why Not Use Threads Instead?

You can use threads instead of asyncio.

Any program developed using threads can be rewritten to use asyncio and coroutines.

Any program developed using coroutines and asyncio can be rewritten to use threads.

Adopting asyncio in a project is a choice, the rationale is yours.

For the most part, they are functionally equivalent.

Many use cases will execute faster using threads and may be more familiar to a wider array of Python developers.

Some use cases in the areas of network programming and executing system commands may be simpler (less code) when using asyncio, and significantly more scalable than using threads.

Further Reading

This section lists helpful additional resources on the topic.

Python Asyncio Books

This section lists my books on Python asyncio, designed to help you get started and get good, super fast.

- Asyncio Module API Cheat Sheet (https://superfastpython.gumroad.com/l/pacs)
- Python Asyncio Jump-Start (https://superfastpython.com/paj-further-reading), Jason Brownlee, 2022. (**my book!**)

Other Books

Other books on asyncio include:

- Python Concurrency with asyncio (https://amzn.to/3LZvxNn), Matthew Fowler, 2022.
- Using Asyncio in Python (https://amzn.to/3lNp2ml), Caleb Hattingh, 2020.

APIs

- asyncio — Asynchronous I/O (https://docs.python.org/3/library/asyncio.html)
- Asyncio Coroutines and Tasks (https://docs.python.org/3/library/asyncio-task.html)
- Asyncio Streams (https://docs.python.org/3/library/asyncio-stream.html)
- Asyncio Subprocesses (https://docs.python.org/3/library/asyncio-subprocess.html)
- Asyncio Queues (https://docs.python.org/3/library/asyncio-queue.html)
- Asyncio Synchronization Primitives (https://docs.python.org/3/library/asyncio-sync.html)

References

- Asynchronous I/O, Wikipedia (https://en.wikipedia.org/wiki/Asynchronous_I/O).

- Coroutine, Wikipedia (https://en.wikipedia.org/wiki/Coroutine).

Conclusions

This is a large guide, and you have discovered in great detail how asyncio and coroutines work in Python and how to best use them in your project.

Did you find this guide useful?
I'd love to know, please share a kind word in the comments below.

Have you used asyncio on a project?
I'd love to hear about it, please let me know in the comments.

Do you have any questions?
Leave your question in a comment below and I will reply fast with my best advice.

Join the discussion on reddit (https://www.reddit.com/r/Python/comments/yqrr94/python_asyncio_the_complete_guide/) and hackernews (https://news.ycombinator.com/item?id=33547323).

Learn Asyncio Fast (without the frustration)

(https://superfastpython.com/paj-footer)

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

The **asyncio module** provides easy-to-use coroutine-based concurrency for asynchronous programming.

Introducing: "Python Asyncio Jump-Start (https://superfastpython.com/paj-footer)".