A Conceptual Overview of asyncio

Release 3.13.7

Guido van Rossum and the Python development team

September 19, 2025

Python Software Foundation
Email: docs@python.org

Contents

This HOWTO article seeks to help you build a sturdy mental model of how `asyncio` fundamentally works, helping you understand the how and why behind the recommended patterns.

You might be curious about some key `asyncio` concepts. You'll be comfortably able to answer these questions by the end of this article:

- What's happening behind the scenes when an object is awaited?
- How does `asyncio` differentiate between a task which doesn't need CPU-time (such as a network request or file read) as opposed to a task that does (such as computing n-factorial)?
- How to write an asynchronous variant of an operation, such as an async sleep or database request.

See also

- The guide that inspired this HOWTO article, by Alexander Nordin.
- This in-depth YouTube tutorial series on `asyncio` created by Python core team member, Łukasz Langa.
- 500 Lines or Less: A Web Crawler With asyncio Coroutines by A. Jesse Jiryu Davis and Guido van Rossum.

1 A conceptual overview part 1: the high-level

In part 1, we'll cover the main, high-level building blocks of `asyncio`: the event loop, coroutine functions, coroutine objects, tasks and `await`.

1.1 Event Loop

Everything in `asyncio` happens relative to the event loop. It's the star of the show. It's like an orchestra conductor. It's behind the scenes managing resources. Some power is explicitly granted to it, but a lot of its ability to get things done comes from the respect and cooperation of its worker bees.

In more technical terms, the event loop contains a collection of jobs to be run. Some jobs are added directly by you, and some indirectly by `asyncio`. The event loop takes a job from its backlog of work and invokes it (or "gives it control"), similar to calling a function, and then that job runs. Once it pauses or completes, it returns control to the event loop. The event loop will then select another job from its pool and invoke it. You can *roughly* think of the collection of jobs as a queue: jobs are added and then processed one at a time, generally (but not always) in order. This process repeats indefinitely with the event loop cycling endlessly onwards. If there are no more jobs pending execution, the event loop is smart enough to rest and avoid needlessly wasting CPU cycles, and will come back when there's more work to be done.

Effective execution relies on jobs sharing well and cooperating; a greedy job could hog control and leave the other jobs to starve, rendering the overall event loop approach rather useless.

```
import asyncio

# This creates an event loop and indefinitely cycles through
# its collection of jobs.
event_loop = asyncio.new_event_loop()
event_loop.run_forever()
```

1.2 Asynchronous functions and coroutines

This is a basic, boring Python function:

```
def hello_printer():
    print(
        "Hi, I am a lowly, simple printer, though I have all I "
        "need in life -- \nfresh paper and my dearly beloved octopus "
        "partner in crime."
    )
```

Calling a regular function invokes its logic or body:

```
>>> hello_printer()
Hi, I am a lowly, simple printer, though I have all I need in life --
fresh paper and my dearly beloved octopus partner in crime.
```

The async def, as opposed to just a plain `def`, makes this an asynchronous function (or "coroutine function"). Calling it creates and returns a coroutine object.

```
async def loudmouth_penguin(magic_number: int):
    print(
     "I am a super special talking penguin. Far cooler than that printer. "
     f"By the way, my lucky number is: {magic_number}."
    )
```

Calling the async function, `loudmouth_penguin`, does not execute the print statement; instead, it creates a coroutine object:

```
>>> loudmouth_penguin(magic_number=3)
<coroutine object loudmouth_penguin at 0x104ed2740>
```

The terms "coroutine function" and "coroutine object" are often conflated as coroutine. That can be confusing! In this article, coroutine specifically refers to a coroutine object, or more precisely, an instance of `types.CoroutineType` (native coroutine). Note that coroutines can also exist as instances of `collections.abc.Coroutine` – a distinction that matters for type checking.

A coroutine represents the function's body or logic. A coroutine has to be explicitly started; again, merely creating the coroutine does not start it. Notably, the coroutine can be paused and resumed at various points within the function's body. That pausing and resuming ability is what allows for asynchronous behavior!

Coroutines and coroutine functions were built by leveraging the functionality of generators and generator functions. Recall, a generator function is a function that `yield`s, like this one:

```
def get_random_number():
    # This would be a bad random number generator!
    print("Hi")
    yield 1
    print("Hello")
    yield 7
    print("Howdy")
    yield 4
    ...
```

Similar to a coroutine function, calling a generator function does not run it. Instead, it creates a generator object:

```
>>> get_random_number()
<generator object get_random_number at 0x1048671c0>
```

You can proceed to the next `yield` of a generator by using the built-in function `next()`. In other words, the generator runs, then pauses. For example:

```
>>> generator = get_random_number()
>>> next(generator)
Hi
1
>>> next(generator)
Hello
7
```

1.3 Tasks

Roughly speaking, tasks are coroutines (not coroutine functions) tied to an event loop. A task also maintains a list of callback functions whose importance will become clear in a moment when we discuss `await`. The recommended way to create tasks is via `asyncio.create_task()`.

Creating a task automatically schedules it for execution (by adding a callback to run it in the event loop's to-do list, that is, collection of jobs).

Since there's only one event loop (in each thread), `asyncio` takes care of associating the task with the event loop for you. As such, there's no need to specify the event loop.

```
coroutine = loudmouth_penguin(magic_number=5)
# This creates a Task object and schedules its execution via the event loop.
task = asyncio.create_task(coroutine)
```

Earlier, we manually created the event loop and set it to run forever. In practice, it's recommended to use (and common to see) `asyncio.run()`, which takes care of managing the event loop and ensuring the provided coroutine finishes before advancing. For example, many async programs follow this setup:

```
import asyncio

async def main():
    # Perform all sorts of wacky, wild asynchronous things...
    ...

if __name__ == "__main__":
    asyncio.run(main())
    # The program will not reach the following print statement until the
    # coroutine main() finishes.
    print("coroutine main() is done!")
```

It's important to be aware that the task itself is not added to the event loop, only a callback to the task is. This matters if the task object you created is garbage collected before it's called by the event loop. For example, consider this program:

```
1  async def hello():
2      print("hello!")
3
4  async def main():
5      asyncio.create_task(hello())
6      # Other asynchronous instructions which run for a while
7      # and cede control to the event loop...
8      ...
9
10 asyncio.run(main())
```

Because there's no reference to the task object created on line 5, it *might* be garbage collected before the event loop invokes it. Later instructions in the coroutine `main()` hand control back to the event loop so it can invoke other jobs. When the event loop eventually tries to run the task, it might fail and discover the task object does not exist! This can also happen even if a coroutine keeps a reference to a task but completes before that task finishes. When the coroutine exits, local variables go out of scope and may be subject to garbage collection. In practice, `asyncio` and Python's garbage collector work pretty hard to ensure this sort of thing doesn't happen. But that's no reason to be reckless!

1.4 await

`await` is a Python keyword that's commonly used in one of two different ways:

```
await task
await coroutine
```

In a crucial way, the behavior of `await` depends on the type of object being awaited.

Awaiting a task will cede control from the current task or coroutine to the event loop. In the process of relinquishing control, a few important things happen. We'll use the following code example to illustrate:

```
async def plant_a_tree():
    dig_the_hole_task = asyncio.create_task(dig_the_hole())
    await dig_the_hole_task

    # Other instructions associated with planting a tree.
    ...
```

In this example, imagine the event loop has passed control to the start of the coroutine `plant_a_tree()`. As seen above, the coroutine creates a task and then awaits it. The `await dig_the_hole_task` instruction adds a callback (which will resume `plant_a_tree()`) to the `dig_the_hole_task` object's list of callbacks. And then, the instruction cedes control to the event loop. Some time later, the event loop will pass control to `dig_the_hole_task` and the task will finish whatever it needs to do. Once the task finishes, it will add its various callbacks to the event loop, in this case, a call to resume `plant_a_tree()`.

Generally speaking, when the awaited task finishes (`dig_the_hole_task`), the original task or coroutine (`plant_a_tree()`) is added back to the event loops to-do list to be resumed.

This is a basic, yet reliable mental model. In practice, the control handoffs are slightly more complex, but not by much. In part 2, we'll walk through the details that make this possible.

Unlike tasks, awaiting a coroutine does not hand control back to the event loop! Wrapping a coroutine in a task first, then awaiting that would cede control. The behavior of `await coroutine` is effectively the same as invoking a regular, synchronous Python function. Consider this program:

```
import asyncio

async def coro_a():
   print("I am coro_a(). Hi!")

async def coro_b():
   print("I am coro_b(). I sure hope no one hogs the event loop...")

async def main():
   task_b = asyncio.create_task(coro_b())
   num_repeats = 3
   for _ in range(num_repeats):
      await coro_a()
   await task_b

asyncio.run(main())
```

The first statement in the coroutine `main()` creates `task_b` and schedules it for execution via the event loop. Then, `coro_a()` is repeatedly awaited. Control never cedes to the event loop which is why we see the output of all three `coro_a()` invocations before `coro_b()`'s output:

```
I am coro_a(). Hi!
I am coro_a(). Hi!
I am coro_a(). Hi!
I am coro_b(). I sure hope no one hogs the event loop...
```

If we change `await coro_a()` to `await asyncio.create_task(coro_a())`, the behavior changes. The coroutine `main()` cedes control to the event loop with that statement. The event loop then proceeds through its backlog of work, calling `task_b` and then the task which wraps `coro_a()` before resuming the coroutine `main()`.

```
I am coro_b(). I sure hope no one hogs the event loop...
I am coro_a(). Hi!
I am coro_a(). Hi!
I am coro_a(). Hi!
```

This behavior of `await coroutine` can trip a lot of people up! That example highlights how using only `await coroutine` could unintentionally hog control from other tasks and effectively stall the event loop. `asyncio.run()` can help you detect such occurences via the `debug=True` flag which accordingly enables debug mode. Among other things, it will log any coroutines that monopolize execution for 100ms or longer.

The design intentionally trades off some conceptual clarity around usage of `await` for improved performance. Each time a task is awaited, control needs to be passed all the way up the call stack to the event loop. That might sound minor, but in a large program with many `await`'s and a deep callstack that overhead can add up to a meaningful performance drag.

2 A conceptual overview part 2: the nuts and bolts

Part 2 goes into detail on the mechanisms `asyncio` uses to manage control flow. This is where the magic happens. You'll come away from this section knowing what `await` does behind the scenes and how to make your own asynchronous operators.

2.1 The inner workings of coroutines

`asyncio` leverages four components to pass around control.

`coroutine.send(arg)` is the method used to start or resume a coroutine. If the coroutine was paused and is now being resumed, the argument `arg` will be sent in as the return value of the `yield` statement which originally paused it. If the coroutine is being used for the first time (as opposed to being resumed) `arg` must be `None`.

```
1  class Rock:
2      def __await__(self):
3          value_sent_in = yield 7
4          print(f"Rock.__await__ resuming with value: {value_sent_in}.")
5          return value_sent_in
6  
7  async def main():
8      print("Beginning coroutine main().")
9      rock = Rock()
10     print("Awaiting rock...")
11     value_from_rock = await rock
12     print(f"Coroutine received value: {value_from_rock} from rock.")
13     return 23
14 
15 coroutine = main()
16 intermediate_result = coroutine.send(None)
17 print(f"Coroutine paused and returned intermediate value: {intermediate_result}.")
18 
19 print(f"Resuming coroutine and sending in value: 42.")
20 try:
21     coroutine.send(42)
22 except StopIteration as e:
23     returned_value = e.value
24 print(f"Coroutine main() finished and provided value: {returned_value}.")
```

yield, like usual, pauses execution and returns control to the caller. In the example above, the `yield`, on line 3, is called by `... = await rock` on line 11. More broadly speaking, `await` calls the `__await__()` method of the given object. `await` also does one more very special thing: it propagates (or "passes along") any `yield`s it receives up the call-chain. In this case, that's back to `... = coroutine.send(None)` on line 16.

The coroutine is resumed via the `coroutine.send(42)` call on line 21. The coroutine picks back up from where it `yield`ed (or paused) on line 3 and executes the remaining statements in its body. When a coroutine finishes, it raises a `StopIteration` exception with the return value attached in the `value` attribute.

That snippet produces this output:

```
Beginning coroutine main().
Awaiting rock...
Coroutine paused and returned intermediate value: 7.
Resuming coroutine and sending in value: 42.
Rock.__await__ resuming with value: 42.
Coroutine received value: 42 from rock.
Coroutine main() finished and provided value: 23.
```

It's worth pausing for a moment here and making sure you followed the various ways that control flow and values were passed. A lot of important ideas were covered and it's worth ensuring your understanding is firm.

The only way to yield (or effectively cede control) from a coroutine is to `await` an object that `yield`s in its `__await__` method. That might sound odd to you. You might be thinking:

1. What about a `yield` directly within the coroutine function? The coroutine function becomes an async generator function, a different beast entirely.

2. What about a yield from within the coroutine function to a (plain) generator? That causes the error: `SyntaxError: yield from not allowed in a coroutine`. This was intentionally designed for the sake of simplicity – mandating only one way of using coroutines. Initially `yield` was barred as well, but was re-accepted to allow for async generators. Despite that, `yield from` and `await` effectively do the same thing.

2.2 Futures

A future is an object meant to represent a computation's status and result. The term is a nod to the idea of something still to come or not yet happened, and the object is a way to keep an eye on that something.

A future has a few important attributes. One is its state which can be either "pending", "cancelled" or "done". Another is its result, which is set when the state transitions to done. Unlike a coroutine, a future does not represent the actual computation to be done; instead, it represents the status and result of that computation, kind of like a status light (red, yellow or green) or indicator.

`asyncio.Task` subclasses `asyncio.Future` in order to gain these various capabilities. The prior section said tasks store a list of callbacks, which wasn't entirely accurate. It's actually the `Future` class that implements this logic, which `Task` inherits.

Futures may also be used directly (not via tasks). Tasks mark themselves as done when their coroutine is complete. Futures are much more versatile and will be marked as done when you say so. In this way, they're the flexible interface for you to make your own conditions for waiting and resuming.

2.3 A homemade asyncio.sleep

We'll go through an example of how you could leverage a future to create your own variant of asynchronous sleep (`async_sleep`) which mimics `asyncio.sleep()`.

This snippet registers a few tasks with the event loop and then awaits a coroutine wrapped in a task: `async_sleep(3)`. We want that task to finish only after three seconds have elapsed, but without preventing other tasks from running.

```
async def other_work():
    print("I like work. Work work.")

async def main():
    # Add a few other tasks to the event loop, so there's something
    # to do while asynchronously sleeping.
    work_tasks = [
        asyncio.create_task(other_work()),
        asyncio.create_task(other_work()),
        asyncio.create_task(other_work())
    ]
    print(
        "Beginning asynchronous sleep at time: "
        f"{datetime.datetime.now().strftime("%H:%M:%S")}."
    )
    await asyncio.create_task(async_sleep(3))
    print(
        "Done asynchronous sleep at time: "
        f"{datetime.datetime.now().strftime("%H:%M:%S")}."
    )
    # asyncio.gather effectively awaits each task in the collection.
    await asyncio.gather(*work_tasks)
```

Below, we use a future to enable custom control over when that task will be marked as done. If `future.set_result()` (the method responsible for marking that future as done) is never called, then this task will never finish. We've also enlisted the help of another task, which we'll see in a moment, that will monitor how much time has elapsed and, accordingly, call `future.set_result()`.

```python
async def async_sleep(seconds: float):
    future = asyncio.Future()
    time_to_wake = time.time() + seconds
    # Add the watcher-task to the event loop.
    watcher_task = asyncio.create_task(_sleep_watcher(future, time_to_wake))
    # Block until the future is marked as done.
    await future
```

Below, we'll use a rather bare object, `YieldToEventLoop()`, to `yield` from `__await__` in order to cede control to the event loop. This is effectively the same as calling `asyncio.sleep(0)`, but this approach offers more clarity, not to mention it's somewhat cheating to use `asyncio.sleep` when showcasing how to implement it!

As usual, the event loop cycles through its tasks, giving them control and receiving control back when they pause or finish. The `watcher_task`, which runs the coroutine `_sleep_watcher(...)`, will be invoked once per full cycle of the event loop. On each resumption, it'll check the time and if not enough has elapsed, then it'll pause once again and hand control back to the event loop. Eventually, enough time will have elapsed, and `_sleep_watcher(...)` will mark the future as done, and then itself finish too by breaking out of the infinite `while` loop. Given this helper task is only invoked once per cycle of the event loop, you'd be correct to note that this asynchronous sleep will sleep *at least* three seconds, rather than exactly three seconds. Note this is also of true of `asyncio.sleep`.

```python
class YieldToEventLoop:
    def __await__(self):
        yield

async def _sleep_watcher(future, time_to_wake):
    while True:
        if time.time() >= time_to_wake:
            # This marks the future as done.
            future.set_result(None)
            break
        else:
            await YieldToEventLoop()
```

Here is the full program's output:

```
$ python custom-async-sleep.py
Beginning asynchronous sleep at time: 14:52:22.
I like work. Work work.
I like work. Work work.
I like work. Work work.
Done asynchronous sleep at time: 14:52:25.
```

You might feel this implementation of asynchronous sleep was unnecessarily convoluted. And, well, it was. The example was meant to showcase the versatility of futures with a simple example that could be mimicked for more complex needs. For reference, you could implement it without futures, like so:

```python
async def simpler_async_sleep(seconds):
    time_to_wake = time.time() + seconds
    while True:
        if time.time() >= time_to_wake:
            return
        else:
            await YieldToEventLoop()
```

But, that's all for now. Hopefully you're ready to more confidently dive into some async programming or check out advanced topics in the `rest of the documentation`.